TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

EINE DIPLOMARBEIT ÜBER

Modularisierung und Schutz von Applikationen auf einem ressourcenarmen Mikrocontroller mit Embedded Betriebssystem

ZUR ERBRINGUNG DER ANFORDERUNGEN FÜR DEN GRAD DES

Diplom-Ingenieur

in

Embedded Systems 066 504

von

Daniel Aspodinger, BSc

01618886

Betreuer:

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thilo Sauter

Wien, Österreich

Juni 2024

Kurzfassung

Bei der Entwicklung von Software für Embedded Systems, welche nur sehr eingeschränkte Ressourcen besitzen, wird meistens der gesamte Programm-Code in einer Binär-Datei zusammengefügt. Dabei werden alle Sprünge im Code und alle Daten-Abhängigkeiten während des Kompilier- und Linking-Vorganges ermittelt und aufgelöst.

Dieses statische Linken ermöglicht es dem Compiler und dem Linker den Code in seiner Gesamtheit besser zu optimieren, damit die bereits eingeschränkten Ressourcen geschont werden können. Dies birgt jedoch den Nachteil, dass für eine Code-Änderung die gesamte Binär-Datei neu kompiliert, gelinkt und dann ausgetauscht werden muss. Dies führt dazu, dass das Programm nicht in Module gegliedert werden kann, welche unabhängig voneinander sind und nur durch ein vordefiniertes Interface kommunizieren können. Eine solche Modularisierung des Programms wäre vor allem für Safety-Entwicklungen wichtig, damit die safety-kritischen und die nicht safety-kritischen Module getrennt werden können und eine gegenseitige Beeinflussung der Module verhindert werden kann.

Durch den in dieser Arbeit beschriebenen, modularen Ansatz für das Entwickeln einer Software, können für die safety-kritischen und die nicht safety-kritischen Code-Teile getrennte Binär-Dateien erstellt werden und damit verhindert werden, dass während der Laufzeit die nicht safety-kritischen Teile Fehler in den safety-kritischen Teilen verursachen. Dadurch bleiben die safety-kritischen Teile operationsfähig, auch wenn in einem oder mehreren der nicht safety-kritischen Teilen ein Fehler entsteht, welcher zu einem Absturz oder einem Hängenbleiben des nicht safety-kritischen Code-Teils führt. Eine solche Trennung von den safety-kritischen und den nicht safety-kritischen Teilen ist bei vielen Systemen gängige Praxis. Jedoch bei Embedded Systems mit sehr eingeschränkten Ressourcen ist dies nicht der Fall, da hier noch kein Betriebssystem oder keine Betriebssystemerweiterung vorhanden ist, welche eine Modularisierung, inklusive Schutz des Betriebssystems vor den Modulen und den Modulen gegeneinander, vorhanden ist.

Eine weitere Möglichkeit zur Nutzung dieser Modularisierung ist, die Module zur Laufzeit auf das Embedded System zu kopieren und zu installieren, um eine Anpassung oder eine Individualisierung zu vereinfachen.

Abstract

When developing software for a resource constrained embedded system, most often the entire program code is combined in a single binary file. All jumps in the code and all data dependencies are determined and resolved during compilation and linking.

This static linkage allows the compiler and the linker to improve the optimization of the code as a whole and thus conserve the already constrained resources. However, this has the disadvantage that the entire binary file has to be recompiled, linked and then replaced for any code change. This means that the program cannot be divided into modules that are independent of each other, only connected via a predefined interface as sole communication method. Such a modularization of the program would be particularly important for safety critical developments, so that the safety-critical and the non-safety-critical modules can be separated and mutual influencing of the modules can be prevented.

Using the modular approach for developing software, as described in this work, allows the creation of separate binary files for the safety-critical and the non-safety-critical code parts and can prevent the non-safety-critical parts from causing errors in the safety-critical parts during runtime. In this way, the safety-critical parts remain operable even if an error occurs in one or more non-safety-critical parts, which leads to a crash or a hang of the non-safety-critical code part. Such a separation of the safety-critical and the non-safety-critical parts is common practice in many systems. However, this is not the case for resources constrained embedded systems, since there is no operating system or operating system extension available, which allows the modularization including the protection of the operating system from the modules and the modules amongst themselves.

Another way to use this modularization is to copy and install the modules to the embedded system during runtime to make alterations and customizations easier.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Copyright Statement

I, Daniel Aspodinger, BSc, hereby declare that this thesis is my own original work and, to the best of my knowledge and belief, it does not:

- Breach copyright or other intellectual property rights of a third party.
- Contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
- Contain material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.
- Contain substantial portions of third party copyright material, including but not limited to charts, diagrams, graphs, photographs or maps, or in instances where it does, I have obtained permission to use such material and allow it to be made accessible worldwide via the Internet.

Signature: _____

Vienna, Austria, Juni 2024 Daniel Aspodinger, BSc

Danksagung

Als Erstes möchte ich mich von ganzem Herzen bei meinen Eltern, Gerlinde und Ernst, dafür bedanken, dass sie es mir ermöglicht haben, meine schulische Ausbildung voranzutreiben und dieses Studium zu absolvieren. Außerdem will ich mich bei meiner Mutter für das Korrekturlesen dieser Arbeit bedanken.

Bedanken will ich mich auch bei meinem Arbeitskollegen Roland Wintersteller und bei meinem Chef Josef Wallner für ihre Unterstützung bei dieser Arbeit.

Last but not least will ich mich bei meinem Betreuer an der Technischen Universität Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thilo Sauter vom Institut für Computertechnik (ICT) (E384) [1] für seine Hilfe und vor allem für seine Unterstützung bei der Formulierung dieser Arbeit bedanken.

Inhaltsverzeichnis

Tabellenverzeichnis **xi**

Abbildungsverzeichnis **xiv**

Programmcodeverzeichnis **xvi**

Abkürzungsverzeichnis **xix**

1 Einleitung **1**

1.1 Motivation 2
1.2 Problemstellung 4
1.3 Beitrag der Arbeit 5
1.4 Aufbau der Arbeit 6

2 Stand der Technik **7**

2.1 Betriebssysteme mit Application Programming Interface 8
2.2 Virtuelle Maschinen und Sandboxes 21
2.3 Embedded Betriebssysteme 34
2.4 Fazit 45

3 Konzept und Umsetzung **47**

3.1 Application Programming Interface 48
3.2 Schutz des Betriebssystems vor Fehlern, Abstürzen und Endlosschleifen der Module . . 51
3.3 Aufbau eines Moduls 53
3.4 Übertragung, Speicherung und Ausführung der Module 54
3.5 Debugger Unterstützung 55

4 Implementierung **57**

4.1 Implementierung eines Beispiel-Moduls 59

4.2 Installieren eines Moduls 67
4.3 Behandlung von Fehlern eines Moduls 70
4.4 Anpassungen des Schedulers, Aktivierung der Memory Protection Unit und Starten eines Moduls 73
4.5 Modul Wrapper 77
4.6 Abarbeitung von Supervisor-Calls 83

5 Evaluierung **91**
5.1 Validierung 92
5.2 Verifizierung 93

6 Fazit und Ausblick **99**
6.1 Fazit 99
6.2 Ausblick 100

Literaturverzeichnis **102**

Tabellenverzeichnis

2.1 Wichtige Hersteller, Mikrocontroller und Compiler mit Demo-Projekt für FreeRTOS [81] 35
2.2 Wichtige Hersteller und Mikrocontroller mit Portierungen von μC/OSIII [90] [92] . . . 42

Abbildungsverzeichnis

1.1 Gliederung der Software im Mikrocontroller: Mehrere Kompilier-Einheiten werden in ein einziges Binär-File zusammengefügt . . . 2
1.2 Gliederung der Software im Mikrocontroller: mit Modulen und API . . . 3

2.1 UNIX [9] . . . 8
2.2 Vereinfachte historische Darstellung der wichtigsten UNIX-Derivate [14] . . . 9
2.3 Android [15] . . . 10
2.4 Android Software Stack [25] . . . 11
2.5 Linux [29] . . . 12
2.6 Linux Software Stack [30] . . . 13
2.7 macOS [42] . . . 15
2.8 Windows [51] . . . 16
2.9 Oracle VirtualBox [66] . . . 22
2.10 Java [68] . . . 23
2.11 Java Plattform Mikro Edition Embedded 8 Stack [78] . . . 31
2.12 FreeRTOS [79] . . . 34
2.13 Keil RTX [84] . . . 40
2.14 μC/OS-III [88] . . . 41

3.1 Gliederung der Software im Mikrocontroller: Mehrere Kompilier-Einheiten werden in ein einziges Binär-File zusammengefügt . . . 47
3.2 Gliederung der Software im Mikrocontroller: Mehrere eigenständige Module mit API . 48
3.3 Schutz des Betriebssystems vor den Applikationsmodulen und den Applikationsmodulen gegeneinander . . . 51
3.4 Übertragung, Speicherung und Ausführung der Module . . . 54
3.5 Softwareunterstütztes Debugging ohne externe Hardware . . . 55

4.1 Development Board von STM mit STM32L476VG. Logos der verwendeten Systeme (a [96], b [88], c [97], d [98], e [99], f [100], g [101], h [102]) 57
4.2 Zusammenspiel von User-Space (unprivilegiert) und Kernel-Space (privilegiert) mittels SV-Call 65
4.3 Darstellung des MTP-Gerätes mit Anzeige vom gesamten Speicher und freiem Speicher 67
4.4 Inhalt des MTP-Gerätes ohne Module 67
4.5 Readme-Datei mit Informationen zum Kopieren von Modulen 68
4.6 Inhalt des MTP-Gerätes mit zwei Modulen 68
4.7 Informationen zu Fehlern beim Kopieren von Modulen 68
4.8 Informationen über die Position der Module im RAM 69
4.9 Modul-Descriptor angezeigt in einem Text-Editor 69

5.1 Software-Konzept aus Kapitel 3 92
5.2 Beendigung eines fehlerhaften Moduls aufgrund einer Speicherzugriffsverletzung . . . 96
5.3 Beendigung eines fehlerhaften Moduls aufgrund einer zu langen Zykluszeit 97

6.1 Zugriff auf eine Variable ohne Verwendung von PIE (gelb: Code im Flash oder RAM; grün: Variablen im RAM) [104] 101
6.2 Zugriff auf eine Variable unter Verwendung von PIE und GOT (gelb: Code im Flash oder RAM; grün: GOT und Variablen im RAM) [105] 101

Programmcodeverzeichnis

3.1 Datei-Operationen in C/C++ [93] . . . 49
3.2 Datei-Operationen in POSIX [94] . . . 49
3.3 Verwendetes API . . . 50
3.4 Funktionen, in denen die Software des Applikationsmoduls ausgeführt wird . . . 53
4.1 Pseudo-globale Datenstruktur . . . 59
4.2 Funktion zur Initialisierung des Moduls . . . 60
4.3 Periodisch ausgeführte Funktion des Moduls . . . 60
4.4 Funktion zur Deinitialisierung des Moduls . . . 61
4.5 Abstrahierte Implementierung der roten LED . . . 62
4.6 Implementierung der SV-Calls im Modul . . . 63
4.7 Eigentlicher Einstiegspunkt in die Module . . . 64
4.8 Logging Funktion der Module . . . 66
4.9 Fehlerfunktion zur Behandlung von allen Fehlern . . . 70
4.10 Fehlerfunktion zur Behandlung von unzulässigen Speicherzugriffen . . . 71
4.11 Fehlerfunktion, welche einen fehlerhaften Thread stoppt . . . 72
4.12 Zusätzlicher Code für den Context-Switch . . . 74
4.13 Starten der Module in einzelnen Threads . . . 76
4.14 Destruktor von der Klasse `ModuleWrapper` . . . 77
4.15 Funktion zum Umkopieren von Elementen, welche an das Modul übergeben werden sollen . . . 77
4.16 Funktion zum Beenden eines Moduls . . . 78
4.17 Funktion zum Starten eines Moduls . . . 79
4.18 Funktion, welche aufgerufen wird, wenn das Modul die Rechenzeit abgibt . . . 80
4.19 Funktion, welche aufgerufen wird, bevor das Modul einen neuen Zyklus beginnt . . . 80
4.20 Timer-Funktion zum Aktivieren der Module und zur Überprüfung der Rechenzeit . . . 81
4.21 Interrupt-Funktion des SV-Calls . . . 83

4.22 C-Funktion zur Abarbeitung der SV-Calls 84
4.23 Funktion zum Öffnen einer Geräte-Datei 86
4.24 Funktion zum Lesen von Geräte-Dateien 87
4.25 Funktion zum Schreiben von Geräte-Dateien 88
4.26 Funktion zum Konfigurieren von Geräte-Dateien 89
4.27 Funktion, welche zum Verlassen eines Moduls aufgerufen wird 90

Abkürzungsverzeichnis

μC/OS-II Micro-Controller Operating Systems-II. 42

μC/OS-III Micro-Controller Operating Systems-III. xiii, 41, 42, 43, 44, 45, 57, 110, 111

μC/linux Micro-Controller Linux. 14

ABI Application Binary Interface. 104

AMP Asymmetric Multiprocessing. 34

AOT Ahead-of-Time. 11, 24

API Application Programming Interface. ix, xiii, xv, 3, 4, 7, 8, 9, 11, 12, 13, 15, 17, 19, 18, 19, 20, 26, 27, 29, 31, 47, 48, 49, 50, 51, 52, 63, 92, 99

ART Android Runtime. 11

ASCII American Standard Code for Information Interchange. 44, 60, 69

AT&T American Telephone and Telegraph Company. 8

BSD Berkeley Software Distribution. 22

BSS Block-Starting-Symbol. 100

CPU Central Processing Unit. 9, 10, 20

CRC Cyclic Redundancy Check. 75

DOS Disk Operating System. 22

ELKS Embeddable Linux Kernel Subset. 14, 106

FIFO First In/First Out. 36

FPGA Field-Programmable Gate Array. 24

GCC GNU-Compiler-Collection. 57, 111

GDB GNU-Debugger. 55, 57, 111

GNOME GNU Network Object Model Environment. 12

GNU GNU's Not UNIX!. xvii, 12, 13, 14, 55, 57, 105, 111

GNU GRUB GNU Grand Unified Bootloader. 13

GOT Global Offset Table. xiv, 58, 100, 101, 112

GPIO General-Purpose Input/Output. 19, 29

GUI Graphical User Interface. 29, 42

HAL Hardware Abstraction Layer. 11

HID Human Interface Device. 54

I²C Inter-Integrated Circuit. 19, 29

IBM International Business Machines Corporation. 14

ICT Institut für Computertechnik. vii, 103

Inc. Incorporated. 15, 103, 111

IoT Internet-of-Things. 1, 7, 17, 19, 20, 21, 103

JIT Just-In-Time. 11, 23, 24, 25, 26

KDE Kool Desktop Environment. 12

LIFO Last In/First Out. 36

LILO Linux Loader. 13

M2M Machine-To-Machine. 29

macOS mac Operating System. xiii, 4, 7, 15, 16, 22, 106, 107

MCU Microcontroller Unit. 57, 111

MFC Microsoft Foundation Class Library. 19

MIT Massachusetts Institute of Technology. 34

MMU Memory Management Unit. 7, 14

MPU Memory Protection Unit. x, 51, 73, 75, 93

MRI Monitor for Remote Inspection. 55, 100

MSD Mass Storage Device. 54, 67

MTP Media Transfer Protocol. xiv, 54, 67, 68, 99

Mutex Mutual Exclusion. 20, 34, 40, 42, 43, 44, 52

OEM Original Equipment Manufacturer. 31

OS Operating System. xiii, xvii, xviii, xix, 4, 14, 15, 22, 34, 41, 106, 107, 110, 111

OTG On-The-Go. 42

PE Preinstallation Environment. 17

PendSV Pendable-Service-Call. 73, 93

PIE Position Independent Executable. xiv, 58, 100, 101, 112

POSIX Portable Operating System Interface. xv, 15, 16, 49, 50

PROM Programmable Read-Only Memory. 1

PWM Pulse-Width Modulation. 19

QSPI Quad Serial Peripheral Interface. 54, 94

RAM Random-Access Memory. xiv, xix, 1, 14, 19, 20, 29, 30, 37, 38, 40, 41, 54, 58, 69, 75, 76, 93, 94, 95, 100, 101

RT Real-Time. xiii, xix, 20, 25, 34, 110

RTOS Real-Time Operating System. 34, 36, 37, 38, 39, 43

SDRAM Synchronous Dynamic Random-Access Memory. 30

SMP Symmetric Multiprocessing. 34

SPI Serial Peripheral Interface. xix, 19, 29, 54

SRAM Static Random-Access Memory. 1, 7, 14, 30, 47, 57

STM STMicroelectronics. xiii, 30, 34, 41, 57, 111

SV-Call Supervisor-Call. x, xiv, xv, 51, 63, 64, 66, 75, 79, 83, 84, 85, 86, 87, 89, 90

TCB Thread Control Block. 73, 74, 78, 79, 80, 86

TÜV Technischer Überwachungsverein. 3

UART Universal Asynchronous Receiver-Transmitter. 29, 54, 65

UNIX Uniplexed Information Computing System. xiii, xvii, xix, 4, 5, 7, 8, 9, 12, 14, 15, 16, 48, 61, 66, 103, 104

VTable Virtual Table. 58, 101

XNU X is Not UNIX. 15

Kapitel 1

Einleitung

Durch die fortschreitende Digitalisierung und der steigenden Beliebtheit von Internet-of-Things (IoT) Geräten, werden immer mehr Alltagsgegenstände mit einem Mikrocontroller ausgestattet. Dieses Wachstum beschränkt sich nicht nur auf den privaten Bereich, sondern erstreckt sich auch auf Industrie-, Automotive- und Aerospace-Anwendungen [2] [3]. Diese Embedded Systems müssen häufig mit sehr eingeschränkten Ressourcen auskommen. Dies betrifft sowohl die Rechenleistung des Prozessors als auch den Programm-Speicher und den Laufzeitspeicher. Der Programm-Speicher ist abhängig von der produzierten Stückzahl und kann entweder als Flash, für geringe Stückzahlen, oder als Programmable Read-Only Memory (PROM), für sehr hohe Stückzahlen, ausgeführt sein. Für diese Arbeit wurde ein Mikrocontroller mit einem Flash als Programm-Speicher benutzt. Der Laufzeitspeicher ist bei den meisten Mikrocontrollern als Static Random-Access Memory (SRAM) ausgeführt.

Um sowohl Speicherbedarf als auch Rechenleistung zu minimieren, führt der Compiler und der Linker Optimierungen im Code aus. Damit dieser Optimierungsvorgang bestmöglich ausgeführt werden kann, muss dem Compiler und dem Linker der gesamte Code in seiner finalen Form zur Verfügung stehen. Außerdem lösen der Compiler und der Linker alle Sprünge und Datenabhängigkeiten auf und produzieren absolute Adressen oder Adress-Offsets relativ zum Program Counter. Da der gesamte Code zum Kompilierzeitpunkt vorhanden ist, produziert der Compiler normalerweise ein einziges Binärfile, welches in den Programm-Speicher des Mikrocontrollers geladen wird.

Im Bereich der Industrie-, Automotive- und Aerospace-Anwendungen spielt ein weiterer Faktor eine wichtige Rolle: Safety[1]. Safety-kritische Systeme beinhalten meist auch nicht safety-kritische Funktionalitäten. Können diese jedoch nicht von den safety-kritischen Funktionalitäten getrennt werden, so müssen auch die nicht safety-kritischen Funktionalitäten als safety-kritisch betrachtet werden.

[1] Im Deutschen ist der Begriff Sicherheit zweideutig, darum werden hierfür die englischen Begriffe verwendet. *Safety*: Sicherheit vor Schäden an Personen und Gütern, welche vom Gerät verursacht werden könnten. *Security*: Sicherheit vor (Digitalen-) Attacken und Zugriffen von außen auf das Gerät.

1.1 Motivation

Auf Grund der steigenden Anforderungen an Embedded Systems und der stetig wachsenden Ressourcen von Mikrocontrollern werden immer häufiger Betriebssysteme im Embedded Bereich eingesetzt. Im Gegensatz zu bekannten Betriebssystemen auf Computern, Servern, Tablets und Smartphones, bestehen die Embedded Betriebssysteme meist nur aus einem einfachen Scheduler mit Prozessprioritäten, einer einfachen Speicherverwaltung für den Laufzeitspeicher und den Stacks[2] der einzelnen Prozesse. Auf Grund dieses einfachen Aufbaus und der kurzen Zeiten, welche für das Wechseln von einem Prozess zu einem anderen benötigt wird, sind sie meistens echtzeitfähig[3], was eine wesentliche Eigenschaft von vielen Embedded Systems ist.

In Abbildung 1.1 ist ein weiterer großer Unterschied zu herkömmlichen Betriebssystemen dargestellt. Die einzelnen Funktionalitäten sind zwar in logische Komponenten, sogenannten Kompilier-Einheiten, gegliedert, diese sind aber oft nicht unabhängig voneinander, wie es bei normalen Computer-Programmen der Fall ist, und werden deshalb vom Compiler und Linker in einem einzelnen Binär-File zusammengefügt.

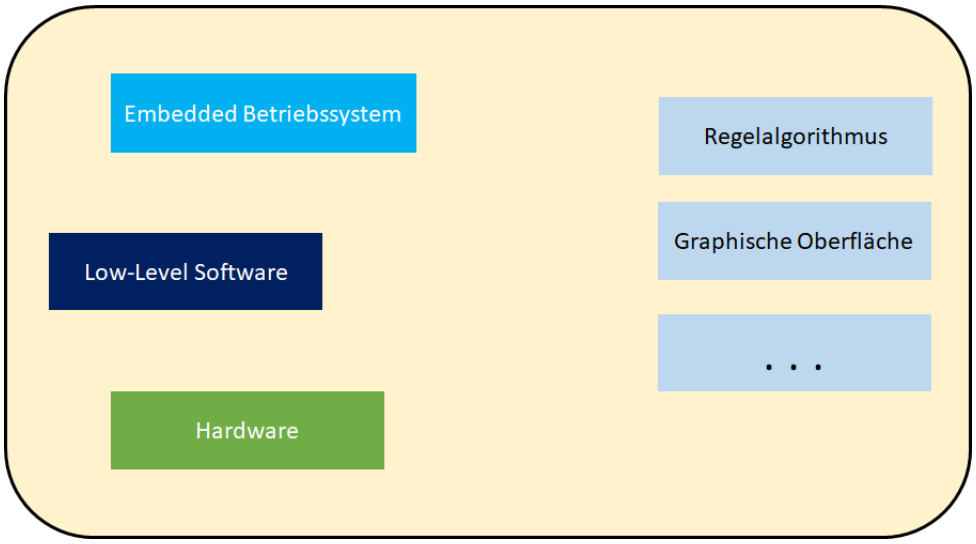

Abbildung 1.1: Gliederung der Software im Mikrocontroller: Mehrere Kompilier-Einheiten werden in ein einziges Binär-File zusammengefügt

Ein solcher Aufbau der Software bringt zwar viele Vorteile, vor allem bei der Optimierung der Laufzeit und des Speicherverbrauchs, mit sich, hat aber den Nachteil, dass es nicht möglich ist, einzelne

[2] Auf dem Stack (engl. für Stapel) werden die lokalen Variablen gespeichert. Wenn eine Funktion aufgerufen wird, dann werden auch die aktuellen Werte der Register auf dem Stack abgelegt, damit die aufgerufene Funktion die Register verwenden und deren Werte verändern kann.

[3] Wenn ein Programm auf einem normalen Betriebssystem ausgeführt wird, benötigt das Programm eine nicht deterministische Zeit, um ein deterministisches Ergebnis zu liefern. In einem Echtzeitbetriebssystem wird das deterministische Ergebnis in einer (zumindest teilweise) deterministischen Zeit geliefert. Echtzeitfähigkeit ist nicht gleichzusetzen mit besserer Performance, es handelt sich nur um vorhersagbare Performance [4]. Meist führt die Einführung der Echtzeitfähigkeit in einem Betriebssystem zu reduzierter Performance.

Teile der Software anzupassen oder zu ersetzen, ohne den gesamten Code neu zu kompilieren und zu linken. Dies mag zwar auf den ersten Blick kein großes Problem darstellen, bedeutet jedoch vor allem bei der Safety-Entwicklung mehr Arbeitsaufwand beim Testen. Dieser Mehraufwand entsteht, weil auch die nicht safety-kritischen Funktionalitäten für die Safety-Zertifizierung getestet und zertifiziert werden müssen, da eine Beeinflussung der safety-kritischen Teile durch die nicht safety-kritischen Teile nicht ausgeschlossen werden kann.

Aus diesem Grund wäre es wünschenswert, wenn ein modularer Aufbau mit einzelnen, unabhängigen Programmen möglich wäre. Diese Module könnten wie auf einem Computer nachträglich installiert oder angepasst werden, ohne den gesamten Code neu kompilieren und linken zu müssen.

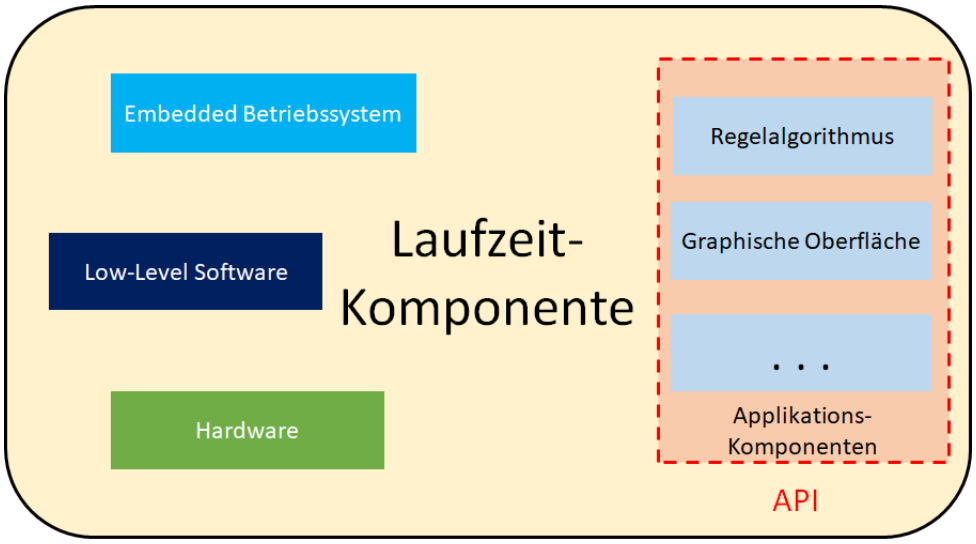

Abbildung 1.2: Gliederung der Software im Mikrocontroller: mit Modulen und API

In Abbildung 1.2 ist ein solcher modularer Aufbau dargestellt. Zusätzlich dazu ist eine definierte Schnittstelle (Application Programming Interface (API)) eingeführt worden, welche die Software in Laufzeit- und Applikations-Komponenten unterteilt. Die Laufzeit-Komponente beinhaltet sowohl das Embedded Betriebssystem als auch alle Low-Level Funktionalitäten, welche Zugriff auf die Hardware oder spezielle Hardware-Register benötigen. In der Laufzeit-Komponente werden alle High-Level Funktionalitäten ausgeführt, welche in einzelne, voneinander unabhängige Module unterteilt sind. Diese Module können über das API mit dem Embedded Betriebssystem kommunizieren und Funktionalitäten aus der Low-Level Schicht anfordern.

Natürlich reicht ein solcher modularer Aufbau mit Gliederung in Laufzeit- und Applikations-Komponenten nicht aus, um die safety-kritischen Module vor den nicht safety-kritischen Modulen zu schützen, schafft aber die nötigen Voraussetzungen, um eine Schutzschicht zwischen den beiden Systemteilen einziehen zu können.

1.2 Problemstellung

Bei safety-kritischen Anwendungen werden vermehrt auch Mikrocontroller eingesetzt, um die Sicherheitsfunktionen zu implementieren. Solche Systeme bestehen nicht nur aus safety-kritischen Funktionalitäten, sondern besitzen auch nicht safety-kritische Funktionalitäten wie Displays, Eingabemöglichkeiten, Kommunikationsschnittstellen, etc.. Wenn diese nicht safety-kritischen Funktionalitäten nicht von den safety-kritischen Funktionalitäten getrennt werden können, kann es passieren, dass aufgrund eines Fehlers im nicht safety-kritischen Teil auch der safety-kritische Teil nicht mehr fehlerfrei funktioniert. Dies kann zum Beispiel ein fehlerhafter Speicherzugriff oder eine Endlosschleife sein, wodurch die safety-kritischen Daten korrumpiert werden oder die Ausführung der safety-kritischen Funktionalität blockiert wird. Wenn dies der Fall ist, müssen auch die nicht safety-kritischen Teile als safety-kritisch betrachtet werden, was zu einem erheblich größeren Aufwand beim Testen und beim Abnahmeverfahren durch eine externe Prüfstelle (zum Beispiel: Technischer Überwachungsverein (TÜV)) führt.

Daher ist die Aufgabe dieser Arbeit, ein bestehendes Embedded Betriebssystem so zu erweitern, dass die Software in einzelne Module gegliedert werden kann. Diese Module werden in dieser Arbeit als Applikations-Komponenten bezeichnet und jedes dieser Module implementiert einen Teil der Logik für die gewünschte Anwendung (siehe Abbildung 1.2). Jede einzelne dieser Applikations-Komponenten sollen entweder safety-kritische oder nicht safety-kritische Funktionalitäten beinhalten.[4] In der sogenannten Laufzeit-Komponente sollen sowohl das Embedded Betriebssystem als auch alle Low-Level Funktionalitäten enthalten sein. Als Low-Level Funktionalitäten werden in dieser Arbeit alle Funktionalitäten bezeichnet, welche auf die Hardware oder Hardware-Register des Mikrocontrollers zugreifen und somit inhärent safety-kritisch sind. Während des Kompilier-Vorgangs sollen die Applikations-Komponenten unabhängig voneinander und unabhängig von der Laufzeit-Komponente sein.

Des Weiteren soll ein Schutzmechanismus implementiert werden, welcher die Laufzeit-Komponente vor den Applikations-Komponenten und jede Applikations-Komponente vor den anderen Applikations-Komponenten schützt. Hierfür sollen die Applikations-Komponenten nur über ein API mit der Laufzeit-Komponente kommunizieren können. Über dieses API sollen die Applikations-Komponenten auch Low-Level Funktionalitäten der Laufzeit-Komponente anfordern und Daten mit anderen Applikations-Komponenten austauschen können. Da die gesamte Kommunikation über dieses API geschieht, soll die Laufzeit-Komponente diese überwachen und bei Fehlern das betreffende Modul beenden oder neu starten, um einen fehlerfreien Betrieb der safety-kritischen Funktionalitäten zu gewährleisten, oder bei einem Fehler in einem safety-kritischen Modul das System in einen Fail-Safe Zustand[5] bringen.

[4] Bis auf einfache Applikations-Komponenten zu Test- und Demonstrationszwecken ist die Implementierung dieser Module nicht Teil der Arbeit.

[5] Ein Fail-Safe Zustand ist ein Zustand des Systems, der nach einem safety-kritischen Fehler eingenommen wird und in dem kein Schaden angerichtet werden kann.

1.3 Beitrag der Arbeit

In vielen Bereichen der Software-Entwicklung ist es gängige Praxis, dass es eine Abstraktions-Ebene gibt, welche die Hardware von den applikationsspezifischen Softwareteilen trennt. Bei den bekannten Betriebssystemen von PCs, Servern und Smartphones - wie Windows, mac Operating System (macOS), Linux oder Android - wird diese Abstraktion von Treibern übernommen, welche im Betriebssystem verwaltet werden. Applikationen können über Funktionen des Betriebssystems auf diese Treiber zugreifen und mit diesen, die Hardwarefunktionalitäten nutzen. Außerdem ermöglicht diese Abstraktion, dass Applikationen nur die Funktionen des Betriebssystems kennen müssen und keine Information über die spezielle Hardware benötigen. Damit werden die Applikationen unabhängig von der Hardware und können auf verschiedenen Geräten installiert werden. Vor allem Uniplexed Information Computing System (UNIX)-basierte Betriebssysteme werden immer häufiger in Embedded Anwendungen eingesetzt. Da diese Betriebssysteme Abwandlungen von einem ursprünglich für PCs entwickelten Betriebssystem sind, benötigen sie erhebliche Rechenleistung und Speicherbedarf, wodurch sie für Anwendungen auf ressourcenarmen Mikrocontrollern nicht geeignet sind. Zudem haben diese Betriebssysteme einen viel größeren Funktionsumfang, als für die meisten Embedded Anwendungen benötigt wird.

Es gibt auch spezielle Embedded Betriebssysteme für ressourcenarme Mikrocontroller, jedoch bestehen diese mehrheitlich nur aus einem rudimentären Scheduler mit Prozessprioritäten, einer einfachen Speicherverwaltung für den Laufzeitspeicher und den Stacks der einzelnen Prozesse. Bei diesen wird das Betriebssystem selbst, die Treiber für die Hardware und die Applikations-Software gemeinsam kompiliert, da sie nicht unabhängig voneinander sind. Dadurch können zwar der Compiler und der Linker Optimierungen besser durchführen, jedoch ist der Code von der Hardware abhängig und das kompilierte Programm kann nicht auf einer anderen Hardware verwendet werden.

Das Novum dieser Arbeit ist, diese beiden Ideen zu verbinden und ein Embedded Betriebssystem zu schaffen, bei welchem die Applikations-Software unabhängig vom Betriebssystem und von den Hardware-Treibern entwickelt und kompiliert werden kann, ohne dabei den großen, meist unnötigen Funktionsumfang eines UNIX-Betriebssystems zu besitzen. Dies soll drei wesentliche Vorteile mit sich bringen:

1. Applikationen sollen nachträglich installiert oder ausgetauscht werden können.
2. Applikationen sollen auf verschiedenen Geräten mit unterschiedlicher Hardware-Konfiguration verwendet werden können.
3. Es soll eine Schutzschicht zwischen den Applikationen und dem Betriebssystem eingeführt werden können.

1.4 Aufbau der Arbeit

Diese Arbeit gliedert sich in sechs Kapitel. Zuerst wird in Kapitel 2 der Stand der Technik erläutert und verschiedene, bestehende Betriebssystemvarianten miteinander verglichen. In Kapitel 3 wird das Konzept erläutert, nach welchem ein schlankes Embedded Betriebssystem mit austauschbaren Applikationen erstellt und implementiert werden soll. Kapitel 4 enthält Erklärungen zur genauen Implementierung, welche Probleme dabei aufgetreten sind, wie diese gelöst werden könnten und welcher Lösungsansatz dieser Probleme in der Implementierung verwendet wird. Danach wird in Kapitel 5 das System und die Implementierung der Embedded Betriebssystemerweiterung evaluiert, auf seine praktische Funktionalität untersucht und die Ergebnisse diskutiert. Den Abschluss bildet Kapitel 6 mit einem Fazit und einem Ausblick über weitere Verbesserungsmöglichkeiten und Einsatzgebiete.

Kapitel 2

Stand der Technik

Durch das Fortschreiten der Digitalisierung und der steigenden Anzahl an IoT- und Embedded Geräten steigt auch die Zahl der entwickelten Betriebssysteme. Ein kompletter Überblick über alle existierenden (Embedded) Betriebssysteme wäre ein Thema für eine eigenständige Arbeit und würde daher den Rahmen dieser Arbeit um ein Vielfaches sprengen. Aus diesem Grunde beschränkt sich das Kapitel „Stand der Technik“ auf die bekanntesten/relevantesten Betriebssysteme. Diese wurden in drei Gruppen unterteilt:

- Abschnitt 2.1 Betriebssysteme mit API
- Abschnitt 2.2 Virtuelle Maschinen und Sandboxes
- Abschnitt 2.3 Embedded Betriebssysteme

Um eine weitere Eingrenzung der Betriebssysteme durchführen zu können, werden die betrachteten Betriebssysteme auf ihren Ressourcenbedarf untersucht. Hierfür soll der Mikrocontroller STM32L476VG (80 MHz, 1 MB Flash und 128 kB SRAM) die Referenz sein. Hierbei handelt es sich um einen Mikrocontroller auf Basis des Cortex-M4 Designs von Arm. Arm unterteilt ihre Prozessoren der Cortex-Reihe in drei Gruppen: A, M und R [5].

Cortex-A: Auf Performance getrimmte Prozessoren, speziell für rechenintensive Applikationen. Im Gegensatz zu den anderen beiden Architekturen besitzen sie eine Memory Management Unit (MMU) [6].

Cortex-M: Günstige, energieeffiziente Prozessoren für den Einsatz in Mikrocontrollern und Embedded Systems [7].

Cortex-R: Spezielle Prozessoren mit der Fähigkeit, Anwendungen mit harten Echtzeitanforderungen ausführen zu können [8].

2.1 Betriebssysteme mit Application Programming Interface

Die bekanntesten Vertreter von Betriebssystemen sind Windows und die UNIX-basierten Betriebssysteme Android, Linux und macOS. Bei Windows, Linux und macOS handelt es sich um Betriebssysteme, welche für PCs entwickelt wurden. Android ist ein Betriebssystem für Smartphones und andere Mobile-Devices.

Diese Betriebssysteme besitzen eine Abstraktions-Ebene, ein sogenanntes API, welche die Software, die die Hardware anspricht, von der Software der Applikations-Programme trennt. Die Applikations-Programme können über dieses API auf Funktionen der Hardware oder des Betriebssystems zugreifen oder Funktionen für andere Applikations-Programme zur Verfügung stellen. Durch diese klar definierte Trennung dieser beiden Ebenen, kann das Betriebssystem Kommandos, welche über dieses API gesendet werden, überprüfen und bei Bedarf den Zugriff auf gewisse Bereiche verweigern oder den Benutzer auffordern, diesen Zugriff explizit zu gewähren.

Da diese Betriebssysteme für PCs entwickelt wurden, eignen sie sich nicht für ressourcenarme Mikrocontroller, jedoch sind sie ein gutes Beispiel für die Funktionsweise einer Abstraktion der Hardware-nahen Software von der Applikations-Software. Des Weiteren besitzen einige dieser Betriebssysteme spezielle Derivate, welche für den Einsatz in Embedded Systems entwickelt wurden.

2.1.1 Uniplexed Information Computing System

UNIX-basierte Betriebssysteme bilden eine Gruppe an Multithreading[1] und Multi-User fähigen Betriebssystemen, welche die Konzepte, des ursprünglich in 1969 von American Telephone and Telegraph Company (AT&T) entwickeltem UNIX, welches nicht portabel, Multithreading und Multi-User fähig war, implementieren [10].

Abbildung 2.1: UNIX [9]

UNIX-Systeme sind durch ihren modularen Aufbau charakterisiert. Brian W. Kernighan und Rob Pike haben die Idee des modularen Aufbaus eines Betriebssystems in [11] durch den Satz „[...] the idea that the power of a system comes more from the relationships among programs than from the programs themselves“ zum Ausdruck gebracht. Das Betriebssystem soll nur ein kleines Set an einfachen Programmen zur Verfügung stellen, welche genau definierte Aufgaben erfüllen [12].

Die Philosophie von UNIX ist es, alles als Dateien zu behandeln. Das bedeutet, dass es aus Sicht des Applikations-Programmierers keinen Unterschied macht, ob Daten in eine Datei im Speicher geschrie-

[1] Thread (engl. für Faden) ist ein Teil eines Programms, welcher sequenziell abgearbeitet wird. Beim Multithreading werden mehrere Threads gleichzeitig ausgeführt und müssen sich die Rechenzeit teilen.

ben, über eine Schnittstelle (z.B. RS232) gesendet oder damit Hardware-Schalter geschalten werden. Diese Philosophie gilt auch für die Inter-Prozess-Kommunikation, sogenannten „pipes“. Ein Programm kann Daten auf eine „pipe“ schreiben, als würde es auf das Datei-System im Speicher schreiben. Diese Daten werden jedoch nur temporär in einem Puffer gespeichert, bis ein anderes Programm die Daten dieser „pipe“ ausliest. Dieser Lesezugriff ist, aus Sicht des Applikations-Programmierers, ident dem Lesezugriff auf eine Datei im Speicher [12] [13].

UNIX-basierte Betriebssysteme bestehen im Wesentlichen aus vielen verschiedenen Bibliotheken[2] und Diensten und einem mächtigen Scheduler, dem sogenannten Kernel. Dieser Kernel ist für das Starten, Stoppen und bei Multithreading für das abwechselnde Ausführen von Programmen zuständig. Er ist nicht nur zuständig für das Aufteilen der Central Processing Unit (CPU) Zeit, sondern auch um Konflikte aufzulösen, welche entstehen, wenn mehrere Programme versuchen, die gleiche Ressource zur selben Zeit zu benutzen. Für diesen Zweck besitzt der Kernel spezielle Rechte und daher wird zwischen Kernel-Space und User-Space unterschieden [10].

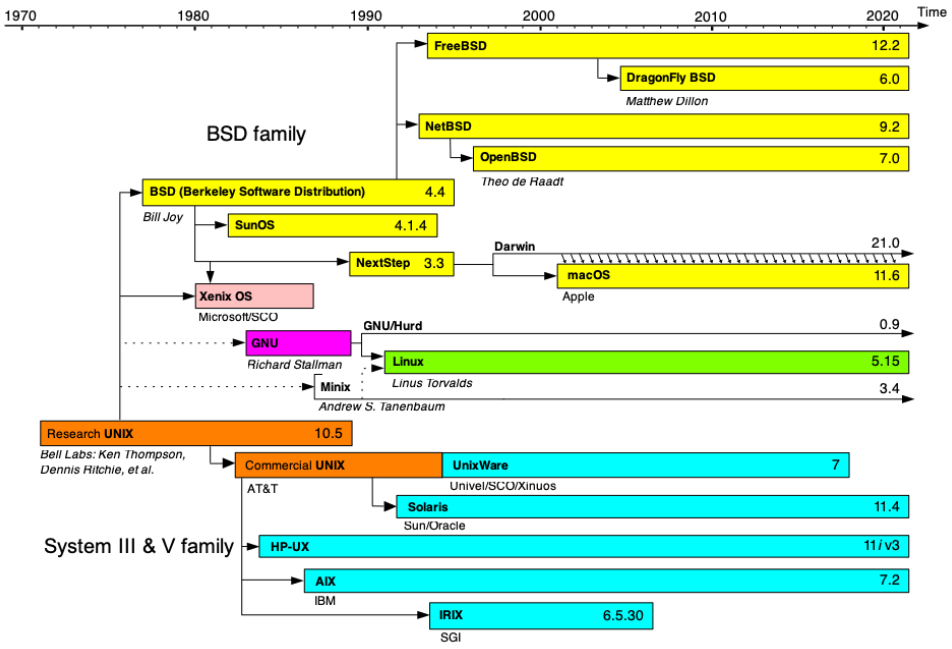

Abbildung 2.2: Vereinfachte historische Darstellung der wichtigsten UNIX-Derivate [14]

Es gibt sehr viele unterschiedliche Betriebssysteme, welche entweder nur das Konzept oder sogar Teile des Codes vom ursprünglichen UNIX übernommen haben. In Abbildung 2.2 kann man eine vereinfachte Darstellung der wichtigsten UNIX-Derivate und deren Relation zueinander sehen.

[2]Bibliotheken (engl. Libraries) sind vorkompilierte Software-Teile, welche Software-Funktionen zur Verfügung stellen, die von vielen anderen Programmen benötigt werden. Sie werden entweder beim Kompilieren mit gelinkt oder während der Laufzeit des Programms durch einen dynamischen Linker gelinkt.

2.1.1.1 Andriod

Abbildung 2.3: Android [15]

Android basiert auf Linux [16] und ist speziell für Smartphones und andere Mobile Devices entwickelt worden [17]. Linux- und Android-Entwickler sind sich jedoch uneinig, ob es sich bei Android um ein Linux-Derivat handelt. So hat Patrick Brady, Android-Entwickler bei Google, bei einer Google IO-Konferenz gesagt „Android is not Linux“ [18]. Daraufhin hat Steven J. Vaughan-Nichols auf „Coumputerworld“ in einem Artikel geschrieben „To quote from the Android developer page, [...][Android] uses the 'Linux kernel for underlying functionality such as threading and low-level memory management.' Let me make it simple for you, without Linux, there is no Android“ [19]. Android baut zwar auf dem Linux-Kernel auf, ist aber abgeändert, um den Anforderungen von Android zu entsprechen, und hat daher wenig mit dem Linux-Kernel von Desktop-Linux-Betriebssystemen gemeinsam [18].

Android kann auch als Kombination aus PC-Betriebssystem und Embedded Betriebssystem gesehen werden. Es gibt jedoch zwei wesentliche Unterschiede zu anderen Embedded Betriebssystemen, die starke Kapselung der Apps und der vergleichsweise große Ressourcenbedarf. Android kann sowohl auf Arm-Prozessoren als auch auf Intel-Prozessoren, mit x86 oder x86-64 Architektur, ausgeführt werden. Die meisten Android-Geräte haben Arm-Prozessoren verbaut, mit entweder 32-Bit-Architektur(ARMv7a) oder 64-Bit-Architektur(ARMv8a). Hierbei handelt es sich um Prozessoren der Cortex-A Familie [20].

Eine weitere, besondere Eigenschaft von Android ist, dass das Haupteingabe-Medium ein Touchscreen ist. Auf diesem werden die Apps und das Betriebssystem über Gesten, wie Wischen, Berühren, Auf- und Zusammenziehen, und über eine virtuelle Tastatur gesteuert [21]. Außerdem besitzen Android-Geräte eine Vielzahl an Sensoren, wie Beschleunigungs-, Gyro- und Näherungs-Sensoren, welche ebenfalls als Steuerungen für Apps verwendet werden können [22].

Da beinahe alle Android-Geräte batteriebetrieben sind, wurde der Prozess-Manager von Android so designt, dass Apps nicht unnötig CPU-Zeit und damit Energie verbrauchen. Hierfür besitzt der Prozess-Manager von Android eine spezielle Routine für den Umgang mit Apps, welche zurzeit nicht in Verwendung sind. Sobald eine App verlassen wird, pausiert Android jegliche Operationen dieser App. Da die App nicht geschlossen, sondern nur pausiert wird, bleibt sie bereit für den sofortigen Wiedereinsatz, spart aber CPU-Zeit und damit Energie [23]. Hierfür müssen die Apps den internen Zustand speichern, bevor sie pausiert werden, um später den Betrieb an der Stelle fortzuführen, an der sie pausiert wurden. Dieser Zustandsspeicher der Apps wird von Android vergeben und überwacht. Wenn

dieser Speicher voll wird, werden pausierte Apps geschlossen und der gespeicherte Zustand gelöscht. Dabei gehen alle Daten im Zustandsspeicher verloren, da Android die entsprechende App nicht benachrichtigt, dass sie geschlossen und ihr Speicher gelöscht wird. Um zu vermeiden, dass relevante Daten verloren gehen, müssen Apps wichtige Daten immer in Dateien abspeichern und nur temporäre Daten über den Zustand der App in den Zustandsspeicher schreiben [24].

Über dem Linux-Kernel laufen die Android Runtime (ART), verschiedenste Bibliotheken und APIs, welche alle in der Programmiersprache C geschrieben sind. Die Applikations-Software läuft auf einem Applikations-Framework mit Java-Bibliotheken [17].

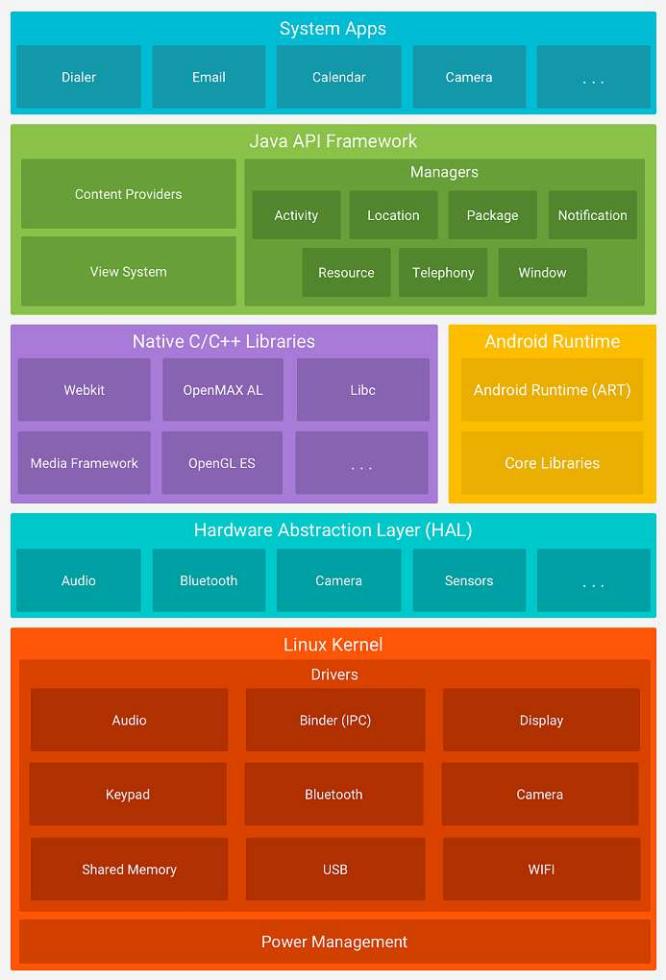

Abbildung 2.4: Android Software Stack [25]

Linux Kernel

Der Linux Kernel bildet, in seiner abgewandelten Form, die Basis von Android und höhere Schichten benutzen Kernel-Funktionalitäten, wie Multithreading und Low-Level Speichermanagement [26].

Hardware Abstraction Layer

Der Hardware Abstraction Layer (HAL) stellt ein Interface zur Verfügung, mit welchem das Java API Framework auf Hardware-Funktionen zugreifen kann. Es besteht aus vielen Bibliotheks-Modulen, wobei jedes einzelne Modul ein Interface für eine einzige Hardwarekomponente implementiert [26].

Android Runtime

ART ist so designt, dass mehrere virtuelle Maschinen auf speicherarmen Geräten ausgeführt werden können. Jede App läuft in einem eigenen Thread und hat seine eigene Instanz der ART. Es verwendet eine Kombination aus Ahead-of-Time (AOT) und Just-In-Time (JIT) [26], dabei wird ein Teil der App als fertig kompilierter Maschinen-Code installiert und der Rest der App als Dalvik Executables gespeichert, welche erst kompiliert werden, wenn dieser Teil der App auch wirklich ausgeführt werden muss [27].

Native C/C++ Bibliotheken

Viele Systemkomponenten, wie ART und HAL, sind als native Software[3] gebaut und benötigen somit auch native Bibliotheken in C/C++. Das Java API Framework stellt einige dieser nativen Bibliotheks-komponenten den Apps zur Verfügung [26].

Java Application Programming Interface Framework

Alle Funktionen von Android werden über APIs, geschrieben in der Programmiersprache Java, den Apps zur Verfügung gestellt. Diese APIs sind die Bausteine zur App-Entwicklung und vereinfachen die Wiederverwendbarkeit von Kernmodulen der Systemkomponenten und Systemdienste [26].

System Apps

Die System Apps sind vorinstallierte Apps, welche häufig benötigte Anwendungen implementieren. Beispiele hierfür sind Programme zum Senden und Empfangen von E-Mails und SMS, Kalender mit Terminbenachrichtigungen, Internetbrowser und Telefonbuch [26].

2.1.1.2 Linux

Linux ist ein modulares UNIX-basiertes Betriebs-system, welches viele wesentliche Teile seiner Design-Prinzipien von dem in den 1970er und 1980er Jahren entwickelten UNIX übernommen hat. Es besitzt einen monolithischen Kernel, den Linux-Kernel, welcher für das Managen und das Scheduling der Prozesse, die Kommunikation über das Netzwerk, den Zugriff auf periphere Hardware und das Schreiben und Lesen des Da-tei-Systems zuständig ist [30]. Linux und dessen unzähligen Derivate beinhalten meist nicht nur den Linux-Kernel, sondern auch unterstützende System-Software, Systemdienste und Bibliothe-ken, welche zu großen Teilen vom GNU's Not UNIX! (GNU)-Projekt stammen [31].

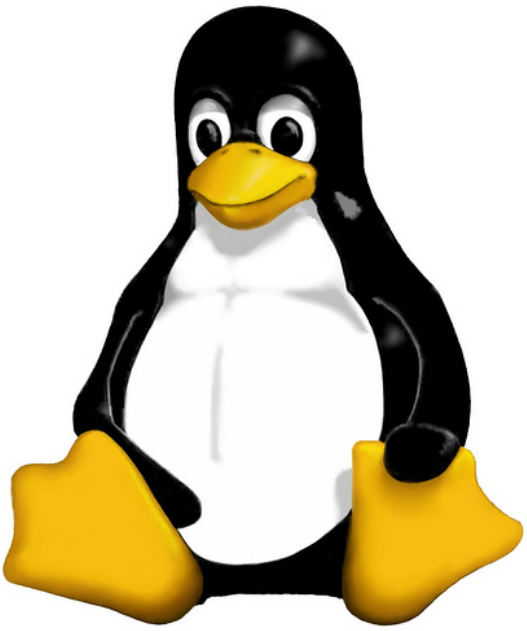

Abbildung 2.5: Linux [29]

Linux wurde ursprünglich für PCs mit Prozessoren der Intel x86-Architektur entwickelt und später auf viele weitere Architekturen portiert [32] und besitzt für Desktop-Anwendungen ein Fenstersystem

[3] Native Software ist Software, welche für einen bestimmten Prozessor gebaut wurde und somit nur auf diesem einen Prozessor ausgeführt werden kann [28].

(engl. windowing system), wie X11 oder Wayland, und eine Desktop-Umgebung (engl. desktop environment), wie GNU Network Object Model Environment (GNOME) oder Kool Desktop Environment (KDE) Plasma [30].

Linux kann aber auch auf Embedded Systems verwendet werden, auf denen es dann oft Teil der Firmware ist und speziell auf den Anwendungsbereich optimiert wird. Typische Embedded Anwendungen mit Linux sind Router, industrielle Automatisierung, Smart-Home-Geräte, Videospielkonsolen und Smart-TVs [30].

Wie bei den meisten modernen Betriebssystemen spielt die Unterscheidung zwischen dem GNU-User-Space und dem Kernel-Space eine wichtige Rolle. Bei Linux fungiert die GNU-C-Bibliothek als API zwischen User-Space und Kernel-Space [33]. Dies wird gemacht, um den Speicher und die Hardware vor bösartiger oder schadhafter Software zu schützen. Hierfür wird der virtuelle Speicher in User-Space und Kernel-Space geteilt und mit unterschiedlichen Berechtigungen ausgestattet. Im Kernel-Space laufen nur privilegierte Anwendungen und Dienste, wie der Kernel und Geräte-Treiber. Im User-Space laufen alle Applikations-Programme im nicht privilegierten Modus, das bedeutet, dass diese Anwendungen nicht direkt auf die Hardware oder gewisse Speicherbereiche zugreifen können [34].

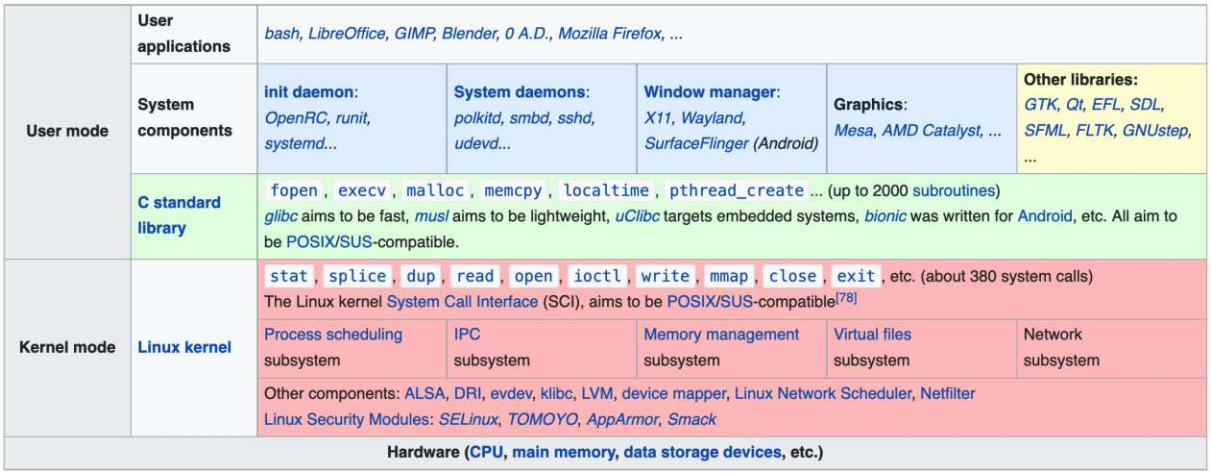

Abbildung 2.6: Linux Software Stack [30]

In Abbildung 2.6 ist die Unterteilung zwischen User-Space und Kernel-Space zu sehen. Des Weiteren sind die einzelnen Schichten eines Linux-Betriebssystems, welche aufeinander aufbauen, zu erkennen. So bauen zum Beispiel User-Applikationen auf den Systemkomponenten und der C-Standard-Bibliothek auf, wobei die Systemkomponenten ebenfalls die C-Standard-Bibliothek verwenden.

Weitere wichtige Komponenten sind [33] [35]:

- Ein **Bootloader**, welcher den Linux-Kernel in den Arbeitsspeicher lädt. Zum Beispiel GNU Grand Unified Bootloader (GNU GRUB), Linux Loader (LILO), SYSLINUX und systemd-boot.

- Das **Initialisierungsprogramm (init)** wird als erstes gestartet und alle anderen Prozesse werden von diesem heraus abgezweigt, es bildet somit die Wurzel des Prozess-Baumes. Beispiele hierfür sind sysvinit, systemd, OpenRC und Upstart.
- **Software-Bibliotheken** enthalten bestehenden Code, welcher von den Prozessen verwendet werden kann. Der bekannteste Vertreter ist die GNU-C-Bibliothek (glibc), welche die Standard-C-Bibliothek von Linux ist. Eine C-Bibliothek wird von fast allen Programmen benötigt, welche in der Programmiersprache C geschrieben sind.

Linux auf Embedded Systems

Linux kann auch auf Embedded Systems installiert werden, es benötigt jedoch große Mengen an Speicher für den Linux-Kernel und das Datei-System. Außerdem muss der Mikrocontroller über eine MMU verfügen, um die virtuelle Adressierung zu ermöglichen. Diese virtuelle Adressierung ist nötig, damit Software, welche für Linux entwickelt wurde, auch auf dem Embedded System ausgeführt werden kann [36].

Laut [36] sind die Mindestanforderungen für ein Embedded Linux 8 GB an externem nicht-volatilem Programm-Speicher, 16 MB Random-Access Memory (RAM), 8 MB internem Flash-Speicher und ein Prozessortakt von mindestens 500 MHz. Zum Vergleich soll hier nochmals der Mikrocontroller STM32L476VG erwähnt werden. Dieser ist ein Mikrocontroller der Arm Cortex—M Familie mit 1 MB internem Flash-Speicher (wird als nicht-volatiler Programm-Speicher genutzt), 128 kB SRAM und einem maximalen Prozessortakt von 80 MHz. Prozessoren der Arm Cortex-M Familie besitzen zudem keine MMU.

Weiters soll hier noch Micro-Controller Linux (μC/linux) erwähnt werden. μC/linux wurde als Embedded Variante von Linux entwickelt, wurde aber nicht mehr gewartet, seit der Embedded Support in den Linux-Mainline-Kernel übernommen wurde.

Embeddable Linux Kernel Subset

Das Embeddable Linux Kernel Subset (ELKS) ist ein Linux-basiertes Betriebssystem, welches für die Intel IA16-Architektur entwickelt wurde. Bei der Intel IA16-Architektur handelt es sich um 16-Bit-Prozessoren. Die Hauptanwendungsbereiche dieser Linux-Variante sind „altertümliche“ Computer wie der International Business Machines Corporation (IBM)-PX XT/AT [37]. Nach eigenen Angaben der Programmierer in der Dokumentation [38] von ELKS gibt es nur mehr sehr wenige Systeme, auf denen ELKS benötigt wird, da die meisten Mikrocontroller viel besser geworden sind, seit ELKS entwickelt wurde.

Raspberry Pi Operating System

Raspberry Pis sind eine Familie an kompakten Einplatinen-Computern. Das Raspberry Pi Operating System (OS) ist ein weiteres UNIX-basiertes Betriebssystem, welches auf dem Linux-Derivat Debian aufbaut. Raspberry Pi OS läuft auf allen Raspberry Pis bis auf den Raspberry Pi Pico [39]. Alle Raspberry Pis, auf denen Raspberry Pi OS installiert werden kann, laufen auf relativ leistungsstarken Mikrocontrollern (Raspberry Pi1: Single-Core mit 700 MHz, mindestens 256 MB SRAM; ab Raspberry Pi2: Quad-Core mit mindestens 900 MHz, mindestens 512 MB SRAM) [40]. Im Gegensatz dazu besitzt der Raspberry Pi Pico nur einen Mikrocontroller mit einem Dual-Core Prozessor der Arm Cortex-M0+ Familie mit maximal 133 MHz (übertaktet 400 MHz) und 264 kB SRAM [41].

2.1.1.3 mac Operating System

macOS ist ein Portable Operating System Interface (POSIX)-kompatibles Betriebssystem und zählt daher auch zu den UNIX-Derivaten. Es baut auf dem X is Not UNIX (XNU)-Kernel auf und wurde von der Firma Apple Incorporated (Inc.) für die von Apple produzierten Computer und Laptops entwickelt [43]. Teile von macOS sind als freie und offene Software als eigenständiges Betriebssystem mit dem Namen „Darwin“ veröffentlicht [44]. Aufbauend auf Darwin hat Apple noch eigene, proprietäre Komponenten zu macOS hinzugefügt, wie das Aqua Interface, welches für die grafische Darstellung zuständig ist, und den Finder, welcher die Visualisierung des Datei-Systems übernimmt. Das Management des Datei-Systems ist hingegen in Darwin enthalten und verwendet die UNIX-Zugriffsrechte [45].

Abbildung 2.7: macOS [42]

macOS ist auf allen drei Prozessor-Architekturen gelaufen, welche seit 1999 in Apple PCs und Laptops verwendet wurden. Diese sind PowerPC, von 1999 bis 2006, Intel, von 2006 bis 2020, und seit 2020 der von Apple eigenes entwickelte 64-Bit, Arm-basierte Apple M1 Prozessor [45].

Die Architektur von macOS ist in Schichten unterteilt. Dies hilft bei der Entwicklung von Programmen, da existierender Code durch das Betriebssystem bereitgestellt wird und so der Programmieraufwand reduziert wird. Des Weiteren stellt Apple eigene Software zum Entwickeln von Programmen zur Verfügung. Das Wichtigste davon ist die integrierte Entwicklungsumgebung „Xcode“. Es beinhaltet Compiler, welche verschiedene Programmiersprachen unterstützen, wie zum Beispiel C, C++, Objecti-

ve-C und Swift [46]. Es gibt zwei APIs für die native Entwicklung von Programmen: Cocoa und Carbon. Carbon ist eine Adaptierung des alten APIs vom klassischen Mac OS, um den Änderungsaufwand für bestehende Programme, damit diese auf dem neueren Mac OS X (später macOS) ausgeführt werden können, gering zu halten. Cocoa ist ein API, welche von OPENSTEP übernommen wurde und keine Abhängigkeit zu dem API des klassischen Mac OS aufweist [45]. Ursprünglich war das API Carbon für macOS nicht geplant gewesen, damit jegliche Software, welche für Apple PCs existierte, neu geschrieben werden muss. Jedoch hat dies zu einem massiven Aufschrei unter den Entwicklern geführt, welche damit drohten, keine Software für macOS zu schreiben, was wiederum dazu führte, dass diese Idee verworfen wurde und Carbon als kompatibles API entwickelt wurde [45].

Aufgrund der Implementierung von POSIX, können viele verschiedene Programme, welche für andere UNIX-basierte Betriebssysteme entwickelt wurden, auch für macOS neu kompiliert werden und somit auch auf Apple-Geräten ausgeführt werden [47]. Es gibt auch viele Third-Party Projekte, wie Homebrew, Fink, MacPorts und pkgsrc, welche vorkompilierte Software-Pakete zum Download zur Verfügung stellen. Programme, welche direkt für macOS entwickelt wurden, können auf verschiedenste Weise den Benutzern zur Verfügung gestellt werden. Eine Möglichkeit ist der Mac App Store, in dem alle Programme von Apple verifiziert sind. Programme, welche über den Mac App Store installiert wurden, werden in einer Sandbox[4] ausgeführt und können dadurch nicht mit anderen Programmen kommunizieren oder Änderungen an Betriebssystem-Einstellungen vornehmen. Dies ist ein zweischneidiges Schwert: Zum einen bringt es den Vorteil, dass Benutzer Software ohne Bedenken installieren können, da diese Software das System nicht beschädigen oder kompromittieren kann. Zum anderen können gewisse Programme, welche erweiterte Rechte benötigen, nicht über den Mac App Store verteilt werden. Solche Programme müssen dann auf anderen Wegen zur Verfügung gestellt werden. Zum Beispiel als downloadbare Installations-Datei, welche mit einem Apple-Entwickler-Account signiert oder auch unsigniert sein kann [49] [50].

2.1.2 Windows

Windows ist eine Reihe von proprietären, grafischen Betriebssystemen, welche von dem Unternehmen Microsoft entwickelt wurden und werden. Windows-Betriebssysteme können wie folgt in Gruppen unterteilt werden [52]:

Abbildung 2.8: Windows [51]

[4]Eine Sandbox ist ein Separationsmechanismus, welcher die laufenden Programme voneinander und vom Betriebssystem trennt. Die Metapher kommt von Kindern, welche nicht streitfrei gemeinsam spielen können und deshalb jedes Kind seinen eigenen Sandkasten bekommt [48].

- **Windows NT**[5] ist eine Familie an Multithreading und Multi-User Betriebssystemen für Server- und PC-Anwendungen. Die wichtigste Eigenschaft von Windows NT ist, dass verschiedenste Software auf unterschiedlichster Hardware ausgeführt werden kann. So läuft Windows unter anderem auf Intel x86-Prozessoren (auf der 32-Bit-Architektur IA-32 und auf der 64-Bit-Architektur x64) und auf Prozessoren von Arm [53].

  Betriebssysteme der Windows NT Familie können wiederum in drei Untergruppen unterteilt werden, welche alle den gleichen Kernel verwenden [52]:

  - **Windows** ist das Betriebssystem für PCs und Tablets.
  - **Windows Server** wird als Betriebssystem auf Servern eingesetzt.
  - **Windows Preinstallation Environment (PE)** ist eine abgespeckte Version von Windows, welche direkt von CDs oder USB-Sticks gebootet werden kann. Es wird verwendet, um Windows auf neuen PCs, zum Beispiel in einer Fabrik eines Computer-Herstellers vor der Auslieferung, automatisiert zu installieren. Es kann auch verwendet werden, um Daten zu retten, falls die Windows-Installation auf einem PC beschädigt ist.

- **Windows IoT** (früher Windows Embedded genannt) ist für Geräte entwickelt worden, welche zu wenig Ressourcen besitzen, um als vollwertiger Computer zu zählen [52]. Siehe dazu auch Abschnitt 2.1.2.2.

2.1.2.1 Windows NT Application Programming Interface

Der Windows-Kernel beinhaltet ein Native-API, welche im Kernel-Space ausgeführt wird. Auf dieses API setzen weitere User-Space-APIs, wie das Windows API, auf [53]. Das Windows API stellt einen Großteil der unteren Schicht den Programmen zur Verfügung. Dies bringt den Vorteil, dass die Programme nicht durch das API in ihrer Funktion eingeschränkt werden. Jedoch bringt dies auch große Verantwortung mit sich, da die Programme auf viele Low-Level Funktionalitäten uneingeschränkt zugreifen können [54]. Normalerweise wird das Windows API von allen Programmen verwendet, Ausnahmen stellen nur Programme dar, welche sehr früh im Start-Up-Prozess des Betriebssystems gestartet werden, diese müssen das Native-API verwenden, da das Windows API noch nicht verfügbar ist [55].

Die Funktionen des Windows APIs können in zwölf Gruppen unterteilt werden [56]:

User Interface ist für die Interaktion mit dem Benutzer verantwortlich. Es erstellt Fenster, um Informationen darzustellen, Benutzereingaben anzufordern und erfüllt andere Aufgaben in Zusammenhang mit der Interaktion mit dem Benutzer [56].

[5] NT war früher eine Abkürzung für „New Technology“, wird aber jetzt nur mehr in der abgekürzten Form verwendet und hat keine nähere Bedeutung [53].

Windows Shell stellt Elemente, welche auch über die grafische Benutzeroberfläche erreichbar sind, in einem organisierten, hierarchischen Namespace dar und stellt einen konsistenten und effizienten Weg für den Zugriff und das Bearbeiten dieser Elemente zur Verfügung. Bei diesen Elementen handelt es sich unter anderem um Dateien, Ordner und virtuelle Elemente, welche zum Beispiel das Senden von Dateien an einen Drucker ermöglichen [57].

Benutzereingaben und Benachrichtigungen stellt zum einen Funktionen zur Verfügung, mit welchen Eingaben des Benutzers über Maus, Tastatur, Touchscreen und anderen Eingabecontrollern an das Programm weitergeleitet werden, zum anderen Funktionen, mit welchen Nachrichten dem Benutzer angezeigt werden können, welche meist wieder eine Eingabe durch den Benutzer benötigen.

Datenzugriff und Speicherung ist nicht nur für den Zugriff und die Speicherung auf dem Hauptspeicher verantwortlich, sondern hat auch Funktionen zum Datenaustausch zwischen Programmen, zum Sichern von Daten und zum temporären Speichern von Daten und Dateien, dem sogenannten Clipboard.

Diagnose-Funktionen helfen dem Programmierer Fehler in seinem Programm oder im System zu lokalisieren und Leistungsdaten zu ermitteln.

Grafik und Multimedia beinhaltet Funktionen, um visuelle und auditive Daten an den Benutzer zu senden. Darunter fallen Videos, Audios, statische Grafiken, formatierter Text und andere multimediale Informationen.

Geräte-Manager ist dafür zuständig, externe Geräte zu verwalten und deren Dienste den Programmen bereitzustellen. Dazu zählen Drucker, USB-Sticks, Headsets und andere, zusätzliche Hardware, welche an den Computer angeschlossen wird.

Systemdienste bieten den Programmen Zugriff auf die Ressourcen des Computers und den Funktionalitäten des Betriebssystems wie Speicher, Datei-System, Prozess-Management und Threads.

Security und Identität ermöglicht den Login beim Systemstart, den Schutz von geteilten Systemobjekten, Rechtemanagement und Sicherheitsüberprüfungen.

Programminstallation und Wartung ermöglicht nicht nur das Installieren und Warten von Programmen, sondern bietet auch andere Dienste wie das Überprüfen von Entwickler-Lizenzen von Programmen.

Systemadministration und Management ermöglicht die Konfiguration und Installation von Systemdiensten, wie das Updaten von Windows oder das Einteilen von Benutzern in Gruppen und der Zuweisung gruppenspezifischer Rechte.

Netzwerk und Internet ermöglicht Programmen die Kommunikation über ein Netzwerk oder dem Internet. Darunter fällt auch der Zugriff auf Netzwerklaufwerke und Netzwerkdrucker.

Die Hauptaufgabe des Windows APIs ist die Kommunikation zwischen den Programmen und dem Betriebssystem. Für die Kommunikation zwischen zwei oder mehreren Programmen waren in der Vergangenheit verschiedene Dienste zuständig. Der aktuelle Standard für die Inter-Programm-Kommunikation ist das .NET Framework [54].

Für das Windows API gibt es auch viele Wrapper[6] Bibliotheken, welche die Windows API Funktionalitäten in neue Funktionen verpacken, um deren Verwendbarkeit zu verbessern. Beispiele hierfür sind Bibliotheken, welche Low-Level Zugriffe vereinfachen, indem sie einen Teil der Arbeit übernehmen oder die Bibliotheken, wie die Microsoft Foundation Class Library (MFC), welche die C-Funktionen des APIs als C++-Funktionen zur Verfügung stellt und somit einen objektorientierten Umgang mit dem Windows API ermöglicht [54].

2.1.2.2 Windows Internet-of-Things

Windows IoT, früher Windows Embedded genannt, ist eine Gruppe von Betriebssystemen, welche für den Einsatz in Embedded Systems von Microsoft entwickelt wurden. Es gibt drei Untergruppen von Windows IoT: Windows IoT Enterprise (mit Windows 10 IoT Enterprise und Windows 11 IoT Enterprise), Windows 10 IoT Core und Windows Server IoT 2019 [58].

Windows Internet-of-Things Enterprise

Windows IoT Enterprise ist eine vollständige Version von Windows Enterprise mit allen Funktionalitäten der Desktop-Version. Es ist „binary equivalent“ zu Windows Enterprise und somit ist auch der Entwicklungsprozess für Programme und das Systemmanagement gleich wie bei PCs und Laptops [59].

Beginnend mit Version 2004 (April 2020) werden alle neuen Versionen von Windows 10 nur mehr als 64-Bit Versionen veröffentlicht. Für ältere 32-Bit Versionen werden jedoch weiterhin Updates mit neuen Eigenschaften und Security-Verbesserungen veröffentlicht [60]. Die Mindestanforderungen an das System, auf dem Windows IoT Enterprise laufen soll, sind ein Prozessortakt von 1 GHz, 1 GB RAM (bzw. 2 GB RAM für 64-Bit Versionen) und 16 GB nicht-volatiler Speicher (bzw. 20 GB nicht-volatiler Speicher für 64Bit Versionen) [61].

Es besitzt auch einen Embedded Mode, welcher explizit gestartet werden kann und folgende Erweiterungen aktiviert [62]:

- **Hintergrund-Programme:** Hintergrund-Programme laufen ohne zu stoppen und ohne Ressourcenlimitierung. Sollte ein Hintergrund-Programm aus irgendeinem Grund stoppen, wird es automatisch vom Betriebssystem neu gestartet.

[6] engl. für Verpackung/Umschlag.

- **„lowLevel devices“ und „lowLevelDevices“:** Mit „lowLevel devices“ und „lowLevelDevices“ bezeichnet Microsoft zwei unterschiedliche Funktionen. „lowLevel devices“ ermöglicht den Zugriff auf Low-Level Hardware, wie zum Beispiel General-Purpose Inputs/Outputs (GPIOs), Inter-Integrated Circuit (I^2C), Serial Peripheral Interface (SPI) und Pulse-Width Modulation (PWM). „lowLevelDevices“ ermöglicht den Zugriff auf andere Low-Level Hardware, welche nicht von „lowLevel devices“ unterstützt wird.
- **„systemManagement“:** „systemManagement“ ermöglicht erweiterte Systemeinstellungen und Systemmanagement-Funktionen, welche von den Programmen verwendet werden können.

Seit Version 21H2 von Windows 10 IoT Enterprise ist es auch möglich, Windows 10 IoT Enterprise als Echtzeitbetriebssystem auszuführen. Dabei handelt es sich um Soft-Real-Time (RT). Im Gegensatz zu Hard-RT Systemen, bei welchen die Ausführungszeit deterministisch auf einen exakten Zeitpunkt ist, besitzen Soft-RT Systeme einen gewissen Jitter, um welchen die Ausführungszeit schwankt. Der Vorteil von Soft-RT Systemen ist, dass sie auf Multi-Core Systemen laufen können und die Programme in ihrer Implementierung weniger eingeschränkt sind. Windows IoT Enterprise besitzt vier Einstellungen, die nötig sind, um Soft-RT zu aktivieren [4]:

- **Core Isolation:** CPU-Kerne können isoliert und ausschließlich für die Abarbeitung von RT-Aufgaben verwendet werden.
- **Umleitung von Interrupts:** Hardware-Interrupts werden standardmäßig an die nicht-RT-Kerne gesendet. Wenn ein Interrupt jedoch von einem RT-Kern abgearbeitet werden soll, so kann eine benutzerdefinierte Einstellung vorgenommen werden, mit welcher der Interrupt auf den RT-Kern umgeleitet wird.
- **Prioritätsvererbung von Mutual Exclusions (Mutexes):** Damit RT-Threads nicht durch reservierte Ressourcen von niederprioren oder nicht-RT-Threads blockiert werden, werden diese Threads, welche die benötigte Ressource reserviert haben, auf die gleiche Prioritätsstufe wie der wartende Thread gestellt.
- **RT-Thread Prioritäten:** Um auch eine Priorisierung der RT-Threads zu ermöglichen, besitzt Windows IoT Enterprise 16 Priorisierungslevels.

Windows 10 Internet-of-Things Core

Windows 10 IoT Core ist eine Version von Windows 10, welche für kleine Geräte mit oder ohne Display entwickelt wurde. Es kann auf Geräten mit Prozessoren der Architekturen ARMv7, x86 und x86-64 ausgeführt werden [63]. Die Mindestanforderungen an das System sind ein Prozessortakt von 400 MHz, 256 MB RAM (32- und 64-Bit Versionen) bzw. 512 MB RAM für 32-Bit Geräte mit Display und 768 MB

für 64-Bit Geräte mit Display. Des Weiteren wird ein nicht-volatiler Speicher von mindestens 2 GB benötigt. Die Angaben für den Speicherbedarf beinhalten ausschließlich den Bedarf des Betriebssystems, jeglicher Speicherbedarf der Programme muss zu diesen Angaben addiert werden [61]. Um die Größe von Windows 10 IoT Core zu verringern, sind einige Teile des Windows APIs nicht implementiert. Zum Beispiel ist das FileOpenPicker API, welche für den Zugriff auf den lokalen und externen Speicher zuständig ist, nicht inkludiert und muss bei Bedarf vom Entwickler selbst implementiert werden. Auch bei den Gerätetreibern wird Platz gespart, so sind viele Treiber, welche in der Desktop-Version inkludiert sind, nicht automatisch bei Windows 10 IoT Core verfügbar. Diese können entweder direkt aus den Quell-Dateien für Windows 10 IoT Core gebaut werden oder es muss ein eigener Treiber geschrieben werden. Ein weiterer Unterschied zur Windows Desktop-Version, welcher das Datei-System betrifft, sind die eingeschränkten Zugriffsrechte von Programmen auf Dateien und Ordner. Um den Programmen Zugriff auf Dateien oder Ordner zu gewähren, muss dies explizit erlaubt werden. Damit können Dateien und Ordner nicht nur vor Programmen, sondern auch vor den Benutzern geschützt werden. Bei Windows 10 IoT Core wird standardmäßig keine Desktop-Umgebung, sondern ein festgelegtes Programm gestartet [63].

Windows Server Internet-of-Things 2019

Windows Server IoT 2019 ist eine vollständige Version des Windows Server 2019 mit all seinen Funktionalitäten und ist auch dazu „binary equivalent“. Der einzige Unterschied besteht in den Lizenzierungsvarianten [64].

2.2 Virtuelle Maschinen und Sandboxes

Virtuelle Maschinen

Eine virtuelle Maschine ist eine Virtualisierung oder Emulation eines Computersystems und stellt die Funktionalität einer physikalischen Computer-Architektur zur Verfügung. Virtuelle Maschinen können grob in zwei Gruppen unterteilt werden [65]:

- **System virtuelle Maschinen** sind ein Ersatz eines kompletten realen Systems. Sie stellen alle benötigten Funktionen des realen Systems bereit, welche für die Ausführung eines vollständigen Betriebssystems nötig sind. Ein Hypervisor, welcher als native Software[7] ausgeführt wird, kümmert sich um die Nutzung und Verteilung der Hardware und ermöglicht eine isolierte Ausführung von mehreren Umgebungen/Betriebssystemen auf demselben physikalischen Computer. Ein bekannter Vertreter ist die VirtualBox von Oracle. Siehe auch Abschnitt 2.2.1.

[7] Native Software ist Software, welche für einen bestimmten Prozessor-Typ gebaut wurde und somit nur auf diesem Prozessor-Typ ausgeführt werden kann [28].

- **Prozess virtuelle Maschinen**, auch Applikations virtuelle Maschinen genannt, ermöglichen die Ausführung von plattformunabhängiger Software auf einer beliebigen Prozessor-Architektur und ermöglichen somit Programmen auf den unterschiedlichsten Betriebssystemen und Hardware-Konfigurationen zu funktionieren, ohne dabei auf die Eigenheiten jedes Betriebssystems achten und eine eigene Version für jedes Betriebssystem bauen zu müssen. Ein bekannter Vertreter ist die Java virtuelle Maschine. Siehe auch Abschnitt 2.2.2.

Sandboxes

Eine Sandbox ist ein Separationsmechanismus, welcher die laufenden Programme voneinander und vom Betriebssystem trennt, um das Betriebssystem und andere Programme vor schadhaften Programmen zu schützen. Die Metapher kommt von Kindern, welche nicht streitfrei gemeinsam spielen können und deshalb jedes Kind seinen eigenen Sandkasten bekommt. Sandboxes können als Software-Virtualisierung mit sehr eingeschränkten Zugriffsrechten auf das Datei-System und andere wichtige Komponenten gesehen werden [48].

2.2.1 Oracle VirtualBox

Oracle VirtualBox ist eine freie, Open-Source Software[8] zur Virtualisierung von Intel x86, AMD64 und Intel64 Architekturen. Bei diesem Open-Source-Projekt handelt es sich um Software, zu welcher jeder beitragen kann, jedoch diese Beiträge von der Firma Oracle überprüft werden, um die Funktion und Qualität der Software zu garantieren. Betriebssysteme, auf denen VirtualBox ausgeführt werden kann, sind Windows, Linux, macOS und Solaris. Das Betriebssystem auf dem VirtualBox ausgeführt wird, wird Host-OS genannt. Das Guest-OS ist das Betriebssystem, welches in der VirtualBox läuft. Als Guest-OS unterstützt VirtualBox unter anderem Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7, Windows 8, Windows 10), Disk Operating System (DOS)/Windows 3.x, Linux (2.4, 2.6, 3.x und 4.x), Solaris und OpenSolaris, OS/2 und OpenBerkeley Software Distribution (BSD) [67].

Abbildung 2.9: Oracle VirtualBox [66]

[8] Open-Source Software ist Software, bei der der Source-Code zugänglich ist und von jedem bei Bedarf verändert werden kann.

2.2.2 Java virtuelle Maschine

Eine Prozess virtuelle Maschine bietet eine High-Level-Abstraktions-Ebene, in den meisten Fällen in der Form einer High-Level-Programmiersprache, zur Entwicklung und Ausführung von Applikationen an. Diese Applikationen werden während der Ausführung in der virtuellen Maschine entweder durch einen Interpreter oder einen Just-In-Time (JIT) Compiler[9] in Maschinencode[10], entsprechend der Host-Plattform, auf der die virtuelle Maschine ausgeführt wird, umgewandelt [65].

Abbildung 2.10: Java [68]

In Zusammenhang mit der Java virtuellen Maschine wird in der Literatur nicht immer klar zwischen Java-Code als Programmiercode der Programmiersprache Java (und dem objektorientierten Programmierkonzept der Klassen) und dem Java-Bytecode (auch oft nur als Java-Code oder Java-Klassen-Dateien), welcher das Ergebnis des Java-Compilers ist, unterschieden. In dieser Arbeit wird der Begriff „Java“ nie alleine verwendet. Es wird immer ausdrücklich erwähnt, wenn es sich um die „Programmiersprache Java“ handelt. Für den Code, welcher in der Java virtuellen Maschine ausgeführt wird, wird immer der Begriff „Java-Bytecode“ verwendet. Programme, welche in der Java virtuellen Maschine ausgeführt werden können, also als Java-Bytecode gespeichert sind, werden hier „Java-Programme“ genannt, egal in welcher Programmiersprache diese Programme geschrieben wurden, solange diese mit einem entsprechenden Compiler zu Java-Bytecode kompiliert wurden.

Die Java virtuelle Maschine ist ein abstrakter, virtueller Computer, welcher durch eine Spezifikation beschrieben ist. Jede konkrete Implementierung dieser Spezifikation, welche auf einem physikalischen Computer ausgeführt wird, ermöglicht es, Java-Programme auf diesem Computer auszuführen. Es muss für jede Host-Plattform eine eigene Version implementiert werden, da der Java-Bytecode für jede Host-Plattform in einen anderen Maschinencode übersetzt werden muss. Jedoch kann dadurch jeder Computer, auf welchem eine Java virtuelle Maschine läuft, jedes Java-Programm ausführen und

[9] Ein JIT Compiler kompiliert die auszuführende Software während diese ausgeführt wird und kompiliert nur benötigte Softwareteile kurz bevor diese ausgeführt werden.

[10] Maschinencode oder Maschinensprache sind die konkreten Assemblerbefehle, welche von einem Prozessor-Typ ausgeführt werden können. Da jeder Prozessor-Typ ein unterschiedliches Set an Assemblerbefehlen unterstützt, können Programme, welche in Maschinencode vorliegen, nur auf diesem Prozessor-Typ ausgeführt werden.

Java-Programme können ohne jegliche Adaption auf allen Computern mit Java virtuellen Maschinen ausgeführt werden. Der Java-Bytecode kann als Maschinensprache der Java virtuellen Maschine angesehen werden und weil es sich bei dieser Maschinensprache um eine plattformunabhängige Spezifikation handelt, muss diese auch Definitionen beinhalten, welche normalerweise von der Plattform vorgegeben sind. Zum Beispiel ist die Byte-Reihenfolge im Java-Bytecode immer Big-Endian[11], egal welche Byte-Reihenfolge die Host-Plattform hat, auf welcher die Java virtuelle Maschine ausgeführt wird [69].

In der Spezifikation der Java virtuellen Maschine ist nicht definiert, wie der Java-Bytecode in Maschinencode der ausführenden Host-Plattform übersetzt wird. So gibt es mehrere Möglichkeiten, um dies umzusetzen [69]:

- **Interpreter:** Dieser arbeitet jeden Befehl des Java-Bytecode einzeln ab und wandelt den Java-Bytecode Befehl für Befehl einzeln in die entsprechenden Maschinenbefehle der Host-Plattform um.
- **JIT Compiler:** Dieser übersetzt beim ersten Aufruf einer Funktion die gesamte Funktion in die entsprechenden Maschinenbefehle der Host-Plattform und speichert diese ab, damit sie bei einem erneuten Aufruf der Funktion wiederverwendet werden kann. Dies führt zu einer Verbesserung der Ausführungszeit gegenüber dem Interpreter, benötigt jedoch um ein Vielfaches mehr an Speicher.
- **Adaptiver Optimierer:** Dieser startet bei der Ausführung eines Programms als einfacher Interpreter, welcher aber zusätzlich die Aktivität des Programms aufzeichnet. Mit diesen Aufzeichnungen kann der Adaptive Optimierer während der Ausführung des Programms feststellen, welche Teile sehr oft ausgeführt werden. Diese häufig verwendeten Teile werden dann in Maschinencode der Host-Plattform übersetzt und gespeichert. Der Rest des Programms bleibt als Java-Bytecode gespeichert und wird bei Bedarf interpretiert. Somit ist der Adaptive Optimierer eine Kombination aus einem Interpreter und einem JIT Compiler. Damit kann das Programm typischerweise zu 80 % bis 90 % der Zeit vorkompilierten Code ausführen und der Speicherbedarf drastisch reduziert werden, da meist nur 10 % bis 20 % des gesamten Programmcodes als Maschinencode der Host-Plattform gespeichert werden muss.
- **Hardware-Unterstützung:** Es gibt auch einige verschiedene Prozessoren, meist als Implementierungen für Field-Programmable Gate Arrays (FPGAs), welche Java-Bytecode als Maschinencode verwenden und daher diesen direkt und ohne vorherige Interpretierung oder Kompilierung ausführen können.

[11] Big-Endian bedeutet, dass das höchstwertige Byte auf der kleinsten Speicheradresse steht. Im Gegensatz dazu steht bei Little-Endian das niederwertigste Byte auf der kleinsten Speicheradresse.

Programme, welche in der Programmiersprache Java geschrieben sind, können, wie auch Programme anderer Programmiersprachen, direkt in den Maschinencode der gewünschten Plattform kompiliert werden. Dies wird auch Ahead-of-Time (AOT) Kompilierung genannt. Dies beschleunigt zwar die Ausführung, macht aber auch das kompilierte Programm plattformabhängig und kann dadurch nicht mehr in der Java virtuellen Maschine ausgeführt werden und somit auch nicht mehr auf anderen Host-Plattformen. Eine weitere Einschränkung der AOT Kompilierung ist das statische Linken. Normalerweise werden Java-Programme dynamisch, also während der Ausführung des Programms, gelinkt. Dadurch können dynamische Erweiterungen verwendet werden und der Speicher des Programms reduziert werden. Des Weiteren können durch das dynamische Linken einzelne Bibliotheken oder sogar Teile des Programms durch neuere Versionen ersetzt werden und somit ein partielles Update durchgeführt werden. Dies alles ist mit der AOT Kompilierung und dem statischen Linken nicht möglich und es muss jeglicher benötigter Code während der Kompilierung vorliegen und kann auch später nicht mehr ausgetauscht werden [69].

Neben der reduzierten Ausführungsgeschwindigkeit durch den Interpreter bzw. JIT Compiler gibt es auch Nachteile bei der Kontrolle über das Speichermanagement und das Thread Scheduling. Da in der Java virtuellen Maschine ein Garbage-Collector[12], und nicht der Programmierer, für das Zerstören von Programm-Objekten zuständig ist, ist es nicht deterministisch, wann welches nicht mehr genutzte Objekt gelöscht wird. Dadurch kann aber der Programmierer sicher sein, dass es zu keinem Memory Leak[13] durch nicht gelöschte Objekte kommt. Um die Portierung der Java virtuellen Maschine auf andere Host-Plattformen zu erleichtern, ist das Thread Scheduling nicht sehr genau definiert. Dadurch kann das Thread Scheduling, auf unterschiedlichen Host-Plattformen und unterschiedlichen Implementierungen der Java virtuellen Maschine, variieren. Die Java virtuelle Maschine bietet auch keine Möglichkeit, das Scheduling der einzelnen Threads zu beeinflussen und Prioritäten für Threads zu geben. Dies führt vor allem bei Real-Time-Anwendungen zu Problemen [69].

Diese fehlende Kontrolle beim Speichermanagement und dem Thread Scheduling bringt jedoch auch wesentliche Vorteile. So sind Java-Programme in der Java virtuellen Maschine so stark gekapselt, dass sie kaum Schaden am Host verursachen können. So kann ein Java-Programm nicht auf fremden Speicher zugreifen, indem es auf einen bereits freigegebenen Zeiger zugreift. Durch die Typ-Kontrolle des Java-Bytecode ist es auch nicht möglich den Speicher durch Array-Überläufe, fehlerhafte Zeiger-Arithmetik oder fehlerhafte Zeiger-Typ-Konvertierung zu korrumpieren und so einen Absturz des

[12] Der Garbage-Collector (Garbage-Collection engl. für Müllabfuhr) ist ein Teil eines Programms, welches Objekte, auf die keine Referenz mehr verweist, zerstört und damit den Speicher freigibt.

[13] Ein Memory Leak (engl. für Speicherleck) ist ein Problem in Programmiersprachen, bei denen allokierter Speicher vom Programmierer wieder freigegeben werden muss, wenn dieser nicht mehr benötigt wird. Wird dieser Speicher nicht freigegeben und der Zeiger auf diesen Speicher zerstört, zum Beispiel durch Verlassen des Gültigkeitsbereiches des Zeigers, so kann dieser Speicherbereich nicht mehr erreicht und auch nicht mehr freigegeben werden, solange das Programm ausgeführt wird. Dadurch geht Speicher verloren, das Programm verursacht somit ein Leck im Speicher.

Programms oder des gesamten Systems zu verursachen. Des Weiteren überprüft die Java virtuelle Maschine alle Sprünge im Java-Bytecode, ob diese auch auf eine korrekte Adresse im eigenen Programm zeigen [69].

Class-Loader

Neben dem Interpreter bzw. dem JIT Compiler benötigt die Java virtuelle Maschine auch einen Class-Loader, welcher die Java-Klassen-Dateien mit dem Java-Bytecode lädt. Eine Java virtuelle Maschine kann mehrere verschiedene Class-Loader besitzen. Diese können in zwei Gruppen unterteilt werden: „bootstrap“ Class-Loader und benutzerdefinierte Class-Loader. Jede Java virtuelle Maschine benötigt genau einen „bootstrap“ Class-Loader, welcher benutzerdefinierte Klassen und Klassen des Java APIs auf einem standardmäßigen Weg, zum Beispiel vom lokalen, nicht-volatilen Speicher, lädt. Vom Java API werden immer nur jene Klassen geladen, welche auch im auszuführenden Programm wirklich benötigt werden. Benutzerdefinierte Class-Loader sind Teile eines auszuführenden Programms, welche Java-Klassen-Dateien auf einem benutzerdefinierten Weg, zum Beispiel durch Herunterladen der Java-Klassen-Dateien von einem Server über das lokale Netzwerk oder das Internet, laden. Im Gegensatz zum „bootstrap“ Class-Loader sind benutzerdefinierte Class-Loader nicht Teil der Java virtuellen Maschine, sondern liegen als Teil eines Java-Programms in Form von Java-Klassen-Dateien vor und müssen selbst durch den „bootstrap“ Class-Loader oder einem benutzerdefinierten Class-Loader geladen werden [69].

Mithilfe der benutzerdefinierten Class-Loader müssen nicht alle Klassen, welche später zur Laufzeit benötigt werden, zum Kompilierzeitpunkt bekannt sein. Mit den benutzerdefinierten Class-Loadern können Java-Programme dynamisch, also zur Laufzeit, erweitert werden. Wenn ein Java-Programm während der Laufzeit zusätzliche Klassen benötigt, können ein oder mehrere benutzerdefinierte Class-Loader verwendet werden, um diese Klassen über verschiedenste Wege zu laden. Da die benutzerdefinierten Class-Loader ebenfalls aus Java-Bytecode bestehen, können die zu ladenden Klassen über jeden Weg geladen werden, der in Java-Bytecode darstellbar ist, zum Beispiel über das lokale Netzwerk oder das Internet heruntergeladen, aus einer Datenbank extrahiert oder vom Class-Loader selbst berechnet und implementiert werden [69].

Die Java virtuelle Maschine speichert, welche Klasse von welchem Class-Loader geladen wurde. Wenn eine Klasse auf eine neue Klasse verweist und diese noch nicht geladen wurde, versucht die Java virtuelle Maschine diese neue Klasse mit dem gleichen Class-Loader, mit der auch die aufrufende Klasse geladen wurde, zu laden [69]. Wenn zum Beispiel die Klasse „Baum“ mit dem Class-Loader „Wachstum“ geladen wurde und die Klasse „Baum“ zum ersten Mal eine Instanz der Klasse „Apfel“ generieren möchte, so wird die Klasse „Apfel“ mit dem Class-Loader „Wachstum“ geladen und dynamisch zur Klasse „Baum“ gelinkt.

Klassen können nur andere Klassen sehen, welche vom selben Class-Loader geladen wurden, somit gibt es in jedem Java-Programm einen eigenen Namensbereich für jeden Class-Loader. Ein Java-Programm kann mehrere Class-Loader, vom gleichen Typ oder von verschiedenen Typen, beinhalten und kann damit so viele Namensbereiche realisieren, wie es benötigt. Dadurch können Programmteile von anderen Programmteilen isoliert werden und bei Bedarf explizit Zugriffsrechte für einzelne Klassen erteilt werden, um auf Klassen in anderen Namensbereichen zuzugreifen [69].

2.2.2.1 Java Application Programming Interface

Zusätzlich zur Java virtuellen Maschine benötigen Java-Programme auch das Java API und zusammen werden sie Java Plattform oder Java-Runtime-System genannt. Das Java API bietet ein Set an Bibliotheken an, welche den Zugriff auf die Systemressourcen der Host-Plattform ermöglichen. Da diese Bibliotheken auf die Host-Plattform zugreifen müssen, sind die Java-Klassen-Dateien plattformabhängig und müssen für jede Plattform implementiert oder angepasst werden. Methoden, welche auf plattformspezifische Ressourcen zugreifen, werden in dem Java API native Methoden genannt [69].

Bevor eine potenziell schädliche Methode, wie zum Beispiel der Zugriff auf den lokalen, nicht-volatilen Speicher, des Java APIs ausgeführt wird, wird überprüft, ob das Programm bzw. der Programmteil die nötigen Rechte dazu besitzt. Der Security Controller und der Access Controller verwalten die Rechte der einzelnen Programme und ermöglichen gemeinsam mit dem Java API das ungefährliche Ausführen potenziell schadhafter Programme [69].

Java Plattform Standard Edition

Die Java Plattform Standard Edition definiert ein Set an Standard-APIs, welche von allen Implementierungen der Java virtuellen Maschine unterstützt werden müssen [70].

- Das Java-Paket **java.lang** beinhaltet wichtige Klassen und Interfaces für die Programmiersprache Java und das Java-Runtime-System. Es beinhaltet zum Beispiel Stammklassen, von welchen alle anderen Klassen abgeleitet werden müssen, Typdefinitionen für Standarddatentypen, Mathematikfunktionen, Klassen zum Starten, Stoppen und Managen von Threads, Klassen mit Security-Funktionen und einen Teil der Informationen über die Host-Plattform [70].
- Das **java.io** Paket enthält Klassen für den Zugriff auf Input- und Output-Geräte. Diese Klassen beinhalten großteils Stream[14]-Funktionen, mit Ausnahme der Klasse für den Zugriff auf Random-Access Dateien. Das zentrale Stück dieses Pakets sind die abstrakten Klassen InputStream und OutputStream, welche für das Lesen bzw. Schreiben von Bytestreams zuständig sind und

[14] Ein Stream (engl. für Strom/Fluss) ist ein Konzept, um Daten zu behandeln, welche als Datenfluss nacheinander bearbeitet werden sollen und welches keine Sprünge in der Datenmenge zulässt.

die abstrakten Klassen Reader und Writer für das Lesen bzw. Schreiben von Characterstreams. Außerdem beinhaltet es auch Klassen für die Interaktion mit dem Host-Datei-System wie die File Klasse, welche für das Erstellen, Löschen und Umbenennen von Dateien und Ordnern sowie dem Manipulieren von Datei-Attributen verwendet wird. Instanzen der File Klasse, welche eine Datei repräsentieren, können als Übergabeparameter an Klassen wie Input/OutputStream, Reader, Writer und RandomAccessFile dienen. Instanzen der File Klasse, welche einen Ordner repräsentieren, können zum Auflisten der in diesem Ordner enthaltenen Dateien und Ordner genutzt werden [70].

- Das Java-Paket **java.nio** (Non-blocking Inputs and Outputs) ermöglicht einen verbesserten Zugriff auf memory-mapped[15] Eingänge und Ausgänge. Diese Verbesserung wird erzielt, indem die Funktionalitäten dieses Pakets viel enger mit der darunter liegenden Host-Plattform verbunden sind und, wenn vorhanden, die Hardware-Unterstützung für diese Operationen verwendet wird [70].
- Das Paket **java.math** unterstützt Multipräzisions[16]-Arithmetik für Berechnungen mit sehr großen Zahlen und Multipräzisions-Primzahlgeneration für die Erstellung von kryptografischen Schlüsseln [70].
- Mit dem **java.applet** Paket können Java-Applets erstellt werden und/oder über das Netzwerk heruntergeladen und in einer geschützten Sandbox ausgeführt werden. Für diese Sandbox können spezielle Restriktionen erstellt werden, um die Security zu erhöhen und zu vermeiden, dass potenziell schadhafte Software Schaden anrichten kann [70].

Java Plattform Mikro Edition

Die Java Plattform Mikro Edition ist eine abgespeckte Version der Java Plattform Standard Edition, entwickelt für Embedded und Mobile-Geräte, wie Mikrocontroller, Sensoren, Gateways, Smartphones und Drucker. Es ist weiter unterteilt in Konfigurationen, welche wiederum in Profile unterteilt sind. Konfigurationen definieren ein Minimalset an Bibliotheken und Funktionalitäten der Java virtuellen Maschine für einen großen Bereich an Anwendungen und Geräten. Profile stellen eine Spezialisierung von Konfigurationen dar und sind für einen engeren Bereich an Anwendungen und Geräten gedacht. Zusätzlich können optionale Pakete hinzugefügt werden, wenn diese in speziellen Anwendungen oder Geräten benötigt werden [71].

[15] Memory-mapping (engl. für Speicherzuordnung) ist ein Konzept, bei dem Geräte Speicheradressen erhalten, obwohl diese Geräte keinen Speicherbereich darstellen. Damit können Funktionen zur Manipulation des Speichers auch auf Geräte, welche keine Speicherbereiche sind, angewendet werden.

[16] In Zusammenhang mit der Darstellung von Zahlen in einem Computer ist Multipräzision (engl. multiprecision) eine Darstellungsform, bei der die Anzahl der Ziffern nur durch den vorhandenen Speicher limitiert ist. Im Gegensatz dazu steht fixierte Präzision (engl. fixed-precision), welche nur eine limitierte Anzahl an Ziffern besitzt und unter Umständen überlaufen kann. Je nach System kann diese Limitierung der Genauigkeit 8, 16, 32, 64 oder 128 Bit betragen.

Es gibt unter anderem folgende Konfigurationen [72]:

- **Connected Device Configuration:** Diese Konfiguration beinhaltet fast alle Bibliotheken der Java Plattform Standard Edition, ausgenommen der Bibliotheken für das Graphical User Interface (GUI).
- **Connected Limited Device Configuration:** Diese Konfiguration beinhaltet nur jene Bibliotheken, welche unbedingt zur Ausführung der Java virtuellen Maschine nötig sind.

Das **Mobile Information Device Profil** ist ein Profil der Connected Limited Device Configuration und ist speziell für Smartphones entwickelt und enthält ein API für das GUI und ein API für den Zugriff auf den Speicher. Das **Information Module Profil** ist ein Subset des Mobile Information Device Profils und besitzt kein API für das GUI, da es für Embedded Geräte ohne Display konzipiert ist [72].

Ein weiteres Profil der Connected Limited Device Configuration ist das **Embedded Profil** für ressourcenarme Embedded Geräte, wie Wireless-Module für Machine-To-Machine (M2M) Kommunikation, Automatisierung, Umweltsensoren und Positionsverfolgung. Es wurde von Oracle implementiert und wird als Oracle Java Mikro Edition vertrieben. Es beinhaltet unter anderem folgende Funktionalitäten der Spezifikation [71]:

- **Anpassung der Größe** der Java virtuellen Maschine, um es für mehr Anwendungsbereiche nutzbar zu machen, indem nicht benötigte Funktionen der Java virtuellen Maschine entfernt werden. So sind die Minimalanforderungen an das System 1 MB nicht-volatiler Programm-Speicher und 128 kB RAM, wenn die Java virtuelle Maschine nur eine einzelne Applikation ausführen können muss, nur die obligatorischen Pakete beinhaltet und keine optionalen Pakete von der Applikation benötigt werden.
- **Zugriff auf spezielle Hardware**, wie GPIOs, I^2C, SPI und Universal Asynchronous Receiver-Transmitter (UART), passiert direkt aus der Programmiersprache Java und somit kann auch auf alle Geräte, welche über diese Schnittstellen angeschlossen sind, zugegriffen werden.

Das **Java Mikro Edition Embedded Profile 8 „Minimal Profile Set“** beinhaltet nur die obligatorischen Pakete und keine oder nur sehr wenige optionale Pakete. Des Weiteren kann die Java virtuelle Maschine nur eine einzelne Applikation ausführen, wenn das „Minimal Profile Set“ ausgewählt ist. Diese Applikation besitzt maximale Rechte, ohne jegliche Möglichkeit diese einzuschränken und, da es nur diese eine Applikation gibt, ohne Ressourcen mit anderen Applikationen teilen zu müssen. Diese Applikation wird automatisch beim Starten des Gerätes ausgeführt und läuft, bis das Gerät abgeschaltet wird. Für das „Minimal Profile Set“ werden mindestens 2 MB nicht-volatiler Programm-Speicher und 256 kB RAM, für die Java virtuelle Maschine inklusive der Applikation, empfohlen [73].

Das **Java Mikro Edition Embedded Profile 8 „Standard Profile Set“** kann mehrere Applikationen ausführen und es beinhaltet, neben den obligatorischen Paketen, eine größere Anzahl an optionalen Paketen. Dadurch können erweiterte Funktionalitäten des Mikro Edition Embedded Profile 8, wie Applikations-Management, Bibliotheken, Security-Funktionen, Kommunikationsschnittstellen und Ereignis-basierte Programmabarbeitung, genutzt werden. Für das „Standard Profile Set“ werden mindestens 3 MB nicht-volatiler Programm-Speicher und 1 MB RAM, für die Java virtuelle Maschine inklusive der Applikationen, empfohlen [73].

Das **Java Mikro Edition Embedded Profile 8 „Full Profile Set“** kann ebenfalls mehrere Applikationen ausführen und es beinhaltet fast alle optionalen Pakete, außer jene Pakete, welche nicht kombinierbar sind und sich gegenseitig ausschließen, weil sie unterschiedliche Implementierungen derselben Funktionalität darstellen. Dies ermöglicht die Nutzung aller Mikro Edition Embedded Profile 8 Funktionalitäten, wie Applikations-Management, Bibliotheken, Security-Funktionen, Kommunikationsschnittstellen, Ereignis-basierte Programmabarbeitung und Inter-Applikation-Kommunikation. Die Minimalanforderungen für das „Full Profile Set“ sind 4 MB nicht-volatiler Programm-Speicher und 2 MB RAM, für die Java virtuelle Maschine inklusive der Applikationen [73].

Oracle Java Mikro Edition ist für folgende Hardware-Plattformen verfügbar [71]:

- **Raspberry Pi Modell B:** Mindestens 700 MHz Prozessortakt, nicht-volatiler Programm-Speicher auf externer Speicherkarte und mindestens 512 MB Synchronous Dynamic Random-Access Memory (SDRAM) [40]. Das benötigte Betriebssystem ist Debian Linux [71].
- **STMicroelectronics (STM)32429I-EVAL:** Der verbaute Prozessor (STM32F429NIH6) ist ein Cortex-M4 mit 180 MHz Prozessortakt, 2 MB internem, nicht-volatilem Programm-Speicher (Flash) und 256 kB interner SRAM [74]. Auf der Platine sind zusätzlich 16 MB externes Flash, 32 MB externer SDRAM und 2 MB externer SRAM verbaut [75]. Das benötigte Betriebssystem ist Keil RTX [71].
- **STM32F746G-DISCOVERY:** Der verbaute Prozessor (STM32F746NGH6) ist ein Cortex-M7 mit 216 MHz Prozessortakt, 1 MB internem, nicht-volatilem Programm-Speicher (Flash), 320 kB interner SRAM, 4 kB Daten L1-Cache und 4 kB Instruktion L1-Cache [76]. Auf der Platine sind zusätzlich 16 MB externes Flash und 16 MB externer SDRAM verbaut [77]. Das benötigte Betriebssystem ist Keil RTX [71].

Wie in Abbildung 2.11 zu sehen ist, ist die Java virtuelle Maschine Mikro Edition in einzelne Schichten unterteilt, von welchen einige durch den Programmierer des Gesamtsystems angepasst werden

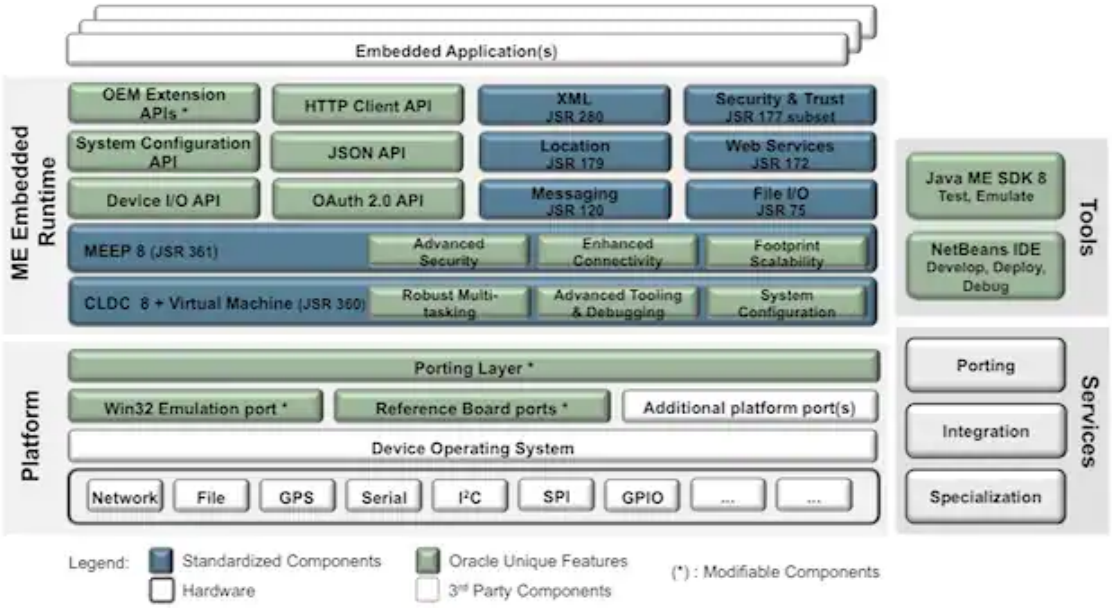

Abbildung 2.11: Java Plattform Mikro Edition Embedded 8 Stack [78]

müssen, um sie für andere Hardware-Plattformen und Betriebssysteme verwendbar zu machen. Dafür können alle Applikationen, welche für die Java virtuelle Maschine Mikro Edition entwickelt wurden, auf allen Hardware-Plattformen und Betriebssystemen ausgeführt werden, welche diese virtuelle Maschine enthalten. Die Java virtuelle Maschine ist in drei Ebenen unterteilt:

- **Plattform:** Die unterste Ebene bildet die Hardware-Plattform und das Betriebssystem mit den Hardware-Treibern. Darauf bauen die Portierungsschichten von Oracle auf, welche von den Programmierern des Gesamtsystems angepasst werden müssen und als definierte Schnittstelle zwischen dem Betriebssystem mit den Hardware-Treibern und der virtuellen Maschine dient.

- **Mikro Edition Embedded Runtime:** Diese Ebene ist die virtuelle Maschine mit ihren einzelnen Komponenten, den obligatorischen und den optionalen Paketen. Einige dieser Elemente sind im Java-Standard definiert und einige sind speziell von Oracle entwickelt worden. Bis auf die „Original Equipment Manufacturer (OEM) Extension APIs“ sind alle Elemente dieser Schicht nicht durch die Programmierer des Gesamtsystems veränderbar.

- **Embedded Applikationen:** Die oberste Ebene sind die Java Embedded Applikationen, welche komplett hardware- und betriebssystemunabhängig sind. Dadurch können diese Applikationen auf allen Geräten mit einer Java virtuellen Maschine Mikro Edition ausgeführt werden und auch die Abhängigkeit von einer einzelnen Hardware-Plattform oder einem einzelnen Prozessor-Hersteller kann entfernt werden.

2.2.2.2 Java Programmiersprache

Die Programmiersprache Java ist eine sehr allgemein einsetzbare Programmiersprache, welche einige sehr wichtige Konzepte zur Entwicklung von Software inkludiert [69]:

- **Objektorientierung** ist ein Programmierkonzept, bei dem Funktionen und Variablen in logische, oft der echten Welt nachempfundenen Klassen gekapselt werden, welche durch Vererbung spezialisiert werden können.
- **Multithreading:** Es können verschiedene Programmteile gleichzeitig ausgeführt werden. Diese müssen sich jedoch die Rechenzeit und alle anderen Ressourcen teilen.
- **Strukturierte Fehlerbehandlung:** In der Programmiersprache Java können Fehler von Funktionen generieren werden, welche dazu führen, dass die Funktion, in der der Fehler generiert wurde, verlassen wird. Diese Fehler können von den aufrufenden Funktionen abgefangen und speziell behandelt werden. Behandelt eine Funktion den Fehler einer aufgerufenen Funktion nicht, so wird auch diese Funktion an der Stelle, an der die Unterfunktion aufgerufen wurde, verlassen. Dies geht die gesamte Liste der aufrufenden Funktionen zurück, bis entweder eine Funktion diesen Fehler behandelt oder, im Falle, dass keine Funktion den Fehler behandelt, eine generische Fehlerbehandlung ausgeführt wird.
- **Garbage-Collection:** Objekte, auf welche keine Referenzen mehr verweisen und dadurch auch nicht mehr auf diese zugegriffen werden kann, werden automatisch vom Garbage-Collector zerstört und der zugehörige Speicher freigegeben.
- **Dynamisches Linken:** Dabei werden Verlinkungen zu Objekten, welche von anderen Objekten referenziert werden, erst zur Laufzeit des Programms aufgelöst und die Ressourcen der verlinkten Objekte erst geladen, wenn diese im Programm auch tatsächlich benötigt werden.
- **Dynamische Erweiterungen:** Dies sind Teile eines Programms, welche erst zur Laufzeit des Programms hinzugefügt werden. Solche dynamischen Erweiterungen können zum Beispiel über das lokale Netzwerk oder das Internet geladen werden.

Eine besondere Eigenschaft der Programmiersprache Java ist der Schutz des Speichers. So gibt es in der Programmiersprache Java keine Zeigerarithmetik, wie es sie in C, C++ oder anderen Sprachen gibt. Zeigerarithmetik wird zum Beispiel verwendet, um von einem Speicherplatz zum nächsten zu gehen, wie beim Iterieren durch ein Array, oder um auf einen bestimmten Speicherplatz zuzugreifen, wie es beim Zugriff auf ein bestimmtes Element in einem Array der Fall ist. Außerdem gibt es strenge Regeln zum Verändern des Datentyps von Objekten. In der Programmiersprache Java kann eine Referenz auf ein Objekt des Typs „Berg“ nur als „Berg“ verändert werden, es kann nicht zu einer Referenz des gänzlich verschiedenen Typs „Lava“ geändert und der Speicher verändert werden, als ob es sich dabei um

ein Objekt des Typs „Lava“ handeln würde. Sind die Typen jedoch kompatibel, kann der Datentyp einer Referenz verändert werden. Wenn der Typ „Vulkan“ eine Spezialisierung des Typs „Berg“ ist, kann eine Referenz auf ein Objekt des Typs „Vulkan“ auch als Referenz eines Objektes des Typs „Berg“ behandelt und der Speicher verändert werden. Eine solche Referenz auf ein Objekt des Typs „Berg“ kann aber nur dann wieder in eine Referenz auf ein Objekt des Typs „Vulkan“ geändert werden, wenn es sich bei dem Objekt wirklich um das Objekt des Typs „Vulkan“ und nicht um eine andere Spezialisierung des Typs „Berg“ handelt. Durch diese strengen Regeln für die Typ-Konvertierung wird verhindert, dass auf Speicher in einer falschen oder böswilligen Variante zugegriffen wird und dadurch der Speicher korrumpiert wird [69].

Wie bereits erwähnt verhindert auch der Garbage-Collector, dass Speicher korrumpiert wird. Wie auch in C++ und anderen Sprachen gibt es auch in der Programmiersprache Java einen new-Operator, um eine neue Referenz am Heap[17] zu erzeugen, jedoch muss dieser Speicher nicht vom Programmierer mit dem „delete“-Operator wieder freigegeben werden, wie es bei C++ und anderen Sprachen der Fall ist. In der Programmiersprache Java ist der Garbage-Collector dafür zuständig, dass belegter, aber nicht mehr genutzter Speicher freigegeben wird. Sobald der Programmierer nicht mehr auf eine Referenz zugreift, bleibt dieser Speicher zwar vorerst belegt, aber der Inhalt dieses Speichers wird nicht mehr benötigt. Dies erkennt der Garbage-Collector und gibt diesen Speicher selbstständig wieder frei. Die Zeit, welche der Garbage-Collector braucht um zu erkennen, dass ein Element am Heap nicht mehr genutzt wird bis dieser Speicher freigegeben wird, hängt von vielen Faktoren ab und muss aus Sicht des Applikationsprogrammierers als nicht deterministisch angesehen werden. Dies verhindert Memory Leaks, da der Programmierer nicht mehr vergessen kann, dass er den Speicher freigibt und es verhindert Speicherkorruption, da Objekte nicht mehrfach freigegeben werden können, wobei bereits neu allokierter Speicher eines anderen Objektes versehentlich freigegeben wird [69].

In C, C++ und anderen Programmiersprachen sind Arrays nur eine vereinfachte Darstellung von Zeigerarithmetik. Dadurch kann man über die Grenzen eines Arrays hinaus auf den Speicher zugreifen und diesen korrumpieren. Zum Beispiel kann man bei einem Array mit zehn Elementen auf das elfte Element zugreifen, ohne dabei eine Fehlermeldung zu erhalten. In der Programmiersprache Java hingegen werden Arrays als Objekte behandelt, welche zusätzliche Informationen besitzen, so auch die Anzahl ihrer Elemente. Mit dieser Information können die Grenzen des Arrays während der Laufzeit überprüft werden und ein illegaler Speicherzugriff verhindert werden [69].

[17] Auf dem Heap (engl. für Haufen) liegen alle Objekte, deren Speicher mit dem new-Operator angefordert wurde. Im Gegensatz zum Stack, wo die Elemente in einer geordneten Reihenfolge liegen, liegen die Elemente am Heap in keiner bestimmten Reihenfolge, sondern hängen nur davon ab, wo ausreichend zusammenhängender Speicher frei ist.

2.3 Embedded Betriebssysteme

Unter Embedded Betriebssystemen werden in dieser Arbeit spezielle Betriebssysteme verstanden, welche vor allem für die Verwendung im Embedded Bereich entwickelt wurden, also für Geräte, welche auf eine oder eine geringe Anzahl an Anwendungen spezialisiert sind und im Allgemeinen eine Veränderung bzw. ein Austausch der Anwendung durch den Benutzer nicht vorgesehen ist. Dies bedeutet jedoch nicht zwangsläufig, dass diese Betriebssysteme für ressourcenarme Mikrocontroller geeignet sind, daher beschränkt sich diese Arbeit bei der Betrachtung von Embedded Betriebssystemen auf jene, welche auf einem ressourcenarmen Mikrocontroller, wie einem Cortex-M4 oder ähnlichem, ausgeführt werden können.

2.3.1 FreeReal-Time Operating System

FreeReal-Time Operating System (RTOS) ist ein Echtzeitbetriebssystem für Mikrocontroller, welches als Open-Source-Software unter der Massachusetts Institute of Technology (MIT) Open-Source-Lizenz verfügbar ist. Es besteht aus einem Kernel und vielen Erweiterungsbibliotheken, welche stetig weiterentwickelt werden. Bei der Entwicklung von FreeRTOS wurde speziell auf die Zuverlässigkeit (engl. reliability) und die einfache Verwendbarkeit geachtet. Damit FreeRTOS für eine große Anzahl an unterschiedlichen Mikrocontrollern nutzbar ist, kann die Größe des Kernels reduziert werden, indem nicht benötigte Funktionalitäten entfernt werden. Damit kann die Größe des Kernels auf 9 KB nicht-volatilem Programmspeicher reduziert werden. Durch die vielen, modularen Erweiterungsbibliotheken kann auf fertige Funktionalitäten für viele Anwendungsbereiche zurückgegriffen werden, darunter auch sichere (im Sinne von Security) Verbindung zu lokalen Netzen oder dem Internet [80].

Abbildung 2.12: FreeRTOS [79]

Für FreeRTOS gibt es Demo-Projekte für viele unterschiedliche Mikrocontroller von vielen verschiedenen Herstellern und je nach Hersteller können verschiedene Compiler verwendet werden. Diese Demo-Projekte können auch angepasst werden, um diese für andere Mikrocontroller derselben Familie verwendbar zu machen. Eine komplette Liste an Herstellern, Mikrocontroller und Compiler kann hier [81] gefunden werden. Einige wichtige sind in Tabelle 2.1 aufgelistet.

FreeRTOS ist ein Embedded Betriebssystem mit Multithreading für Mikrocontroller mit einem oder mehreren Prozessorkernen. Die wählbaren Regeln des Schedulers sind für Mikrocontroller mit einem und mit mehreren Kernen gleich. Bei mehreren Kernen kann zwischen Asymmetric Multiprocessing

Tabelle 2.1: Wichtige Hersteller, Mikrocontroller und Compiler mit Demo-Projekt für FreeRTOS [81]

Hersteller	Mikrocontroller	Compiler
Atmel	SAMV7 (Arm Cortex-M7), SAM3 (Arm Cortex-M3), SAM4 (Arm Cortex-M4), SAMD20 (Arm Cortex-M0+), SAMA5 (Arm Cortex-A5), SAM7 (ARM7), SAM9 (ARM9), AT91, AVR, AVR32 UC3	IAR, GCC, Keil, Rowley CrossWorks
Infineon	TriCore, XMC4000 (Arm Cortex-M4F), XMC1000 (Arm Cortex-M0)	GCC, Keil, Tasking, IAR
Luminary Micro/Texas Instruments	RM48, TMS570, Arm Cortex-M4F MSP432, MSP430, MSP430X, SimpleLink, Stellaris (Arm Cortex-M3, Arm Cortex-M4F)	Rowley CrossWorks, IAR, GCC, Code Composer Studio
NXP	VEGAboard (RISC-V), LPC55S6x(Arm Cortex-M33), LPC1500 (Arm Cortex-M3), LPC1700 (Arm Cortex-M3), LPC1800 (Arm Cortex-M3), LPC1100 (Arm Cortex-M0), LPC2000 (ARM7), LPC4000 (Arm Cortex-M4F, Arm Cortex-M0)	GCC, Rowley CrossWorks, IAR, Keil, LPCXpresso IDE, Eclipse, MCUXpresso IDE
STM	STM32 (Arm Cortex-M0, Arm Cortex-M7, Arm Cortex-M3, Arm Cortex-M4F), STR7 (ARM7), STR9 (ARM9)	IAR, Atollic TrueStudio, GCC, Keil, Rowley CrossWorks

(AMP)[18] und Symmetric Multiprocessing (SMP)[19] ausgewählt werden. Der Standardalgorithmus des FreeRTOS Schedulers ist ein präemptiver[20] Scheduler mit zeitmultiplexiertem[21] Round-Robin[22]-Verfahren für Threads mit gleicher Priorität. Die Priorität der Threads ist dabei statisch und kann vom Programmierer fix gewählt werden, mit der Ausnahme von Prioritätsvererbung[23]. Für das Zeitmultiplexen besitzt FreeRTOS einen System-Timer[24], welcher deaktiviert werden kann, um die Stromaufnahme zu reduzieren. Optional kann die Präemption und das Zeitmultiplexen deaktiviert werden, wodurch der

[18] Beim Asymmetric Multiprocessing (AMP) läuft auf jedem Prozessorkern eine eigene, unabhängige Instanz des Betriebssystems und das Scheduling in jedem einzelnen Prozessorkern ist gleich wie bei einem Prozessor mit einem einzelnen Kern. Jedoch können durch einen gemeinsam genutzten Speicher Daten zwischen den Prozessorkernen ausgetauscht werden und eine Synchronisation der Prozessorkerne ermöglicht werden.

[19] Beim Symmetric Multiprocessing (SMP) wird eine Instanz des Betriebssystems auf mehreren Prozessorkernen ausgeführt und es werden mehrere Threads gleichzeitig ausgeführt. Dadurch kann es dazu kommen, dass zwei oder mehrere Threads unterschiedlicher Priorität gleichzeitig ausgeführt werden und daher alleine durch die Threadpriorität keine definitive Aussage über die Ausführungsreihenfolge getroffen werden kann.

[20] Beim präemptiven Scheduling wird die Rechenzeit vom Betriebssystem verwaltet. So wird ein Thread, welcher zur Ausführung bereit ist, für eine bestimmte Zeit ausgeführt und danach wird er vom Betriebssystem unterbrochen und ein anderer Thread ausgeführt. Dadurch kann ein einzelner Thread nicht die Rechenzeit horten und nicht die anderen Threads um ihre Ausführungszeit berauben.

[21] Beim Zeitmultiplexen von Threads in einem Prozessorkern, wird die Rechenzeit gleich auf die einzelnen Threads verteilt und nach einer bestimmten Zeit der laufende Thread unterbrochen und der nächste Thread ausgeführt, sofern der laufende Thread nicht selbstständig die Rechenzeit abgegeben hat.

[22] Beim Round-Robin-Scheduling handelt es sich um einen Scheduler, bei welchem alle Threads die gleiche Priorität besitzen und der Reihe nach ausgeführt werden.

[23] Wartet ein höherpriorer Thread auf eine Ressource, welche von einem niederprioren Thread belegt ist, so wird die Priorität des Threads, welcher die Ressource belegt, solange auf die Priorität des höchstprioren und wartenden Threads angehoben, bis der ursprünglich niederpriore Thread die Ressource wieder freigegeben hat.

[24] Ein System-Timer ist meist ein Hardware-Timer, bei welchem ein Zähler im Hintergrund läuft, der periodische Ereignisse auslöst und somit eine Zeitbasis für das System bildet.

FreeRTOS Scheduler zu einem kooperativen Scheduler wird, bei dem Threads höherer Priorität Threads mit niedrigerer Priorität unterbrechen können. Als Synchronisierungsmechanismen bietet FreeRTOS mehrere Möglichkeiten: Zähler-Semaphoren[25], Binäre-Semaphoren[26] und Mutexes[27]. In FreeRTOS ist der große Unterschied zwischen Mutexes und Binären-Semaphoren, dass Mutexes eine Prioritätsvererbung besitzen und Binäre-Semaphoren nicht. Bei der Verwendung von Binären-Semaphoren für das Management einer Ressource kann es daher zu Prioritätsinvertierung[28] kommen, da ein niederpriorer Thread, welcher die Ressource besitzt, die Ausführung eines hochprioren Threads blockieren kann.

FreeRTOS besitzt auch verschiedene Möglichkeiten, um Daten zwischen zwei oder mehreren Threads oder einem Interrupt und einem oder mehreren Threads auszutauschen [82]:

- **Queues**[29] sind bei FreeRTOS der Grundbaustein aller Inter-Thread-Kommunikationen. Zum Beispiel verwendet FreeRTOS Queues, um Zähler-Semaphoren zu implementieren, indem die Threads nicht die Daten in der Queue bekommen, sondern nur darüber informiert werden, ob Daten in der Queue verfügbar sind. Daten, welche mittels Queue verschickt werden, werden als Kopie versendet. Dies bringt den Vorteil, dass der lokale Puffer sofort überschrieben werden kann. Wenn große Mengen an Daten verschickt werden und das Kopieren dieser nicht erwünscht ist, so kann ein Zeiger auf die Daten per Queue verschickt werden, dadurch wird nur der Zeiger kopiert.
- **Direct-To-Thread Benachrichtigung:** In FreeRTOS besitzt jeder Thread ein Array an Benachrichtigungen, ein Thread kann immer nur auf das Eintreffen einer einzigen Benachrichtigung in diesem Array blockieren. Die Größe dieses Arrays kann statisch gewählt werden und ein Eintrag im Array benötigt 8 Byte. Jede dieser Benachrichtigungen hat einen Status (Bearbeitet, Unbearbeitet) und eine 32-Bit Nachricht. Eine Direct-To-Thread Benachrichtigung kann direkt an einen Thread geschickt werden, ohne ein vermittelndes Objekt wie eine Queue, ein Event-Bit oder eine

[25] Eine Zähler-Semaphore (engl. counter semaphore) ist ein Konzept in Multithreading Systemen, um beschränkte Anzahl an Ressourcen nur an so viele Threads zu vergeben, wie tatsächliche Ressourcen verfügbar sind, und alle anderen Threads, welche ebenfalls diese Ressource benötigen, auf eine frei werdende Ressource warten zu lassen.

[26] Eine Binäre-Semaphore (engl. binary semaphore) ist eine Zähler-Semaphore mit einem maximalen Zählerwert von eins und kann verwendet werden, wenn von einer Ressource nur eine einzige Instanz existiert. Das Haupteinsatzgebiet von Binären-Semaphoren ist jedoch die Synchronisation zwischen zwei Threads oder einem Thread und einem Interrupt.

[27] Mit einer Mutual Exclusion (Mutex) (engl. für gegenseitiger Ausschluss) können Ressourcen, welche nur einmal verfügbar sind, geschützt werden, damit nur ein Thread darauf zugreifen kann.

[28] Die Prioritätsinvertierung ist ein Phänomen, welches auftreten kann, wenn Threads unterschiedlicher Priorität dieselbe Ressource verwenden möchten. Dabei muss ein hochpriorer Thread auf einen niederprioren Thread warten, bis dieser die Ressource wieder freigibt. Das Problem hierbei ist, dass der hochpriore Thread in dieser Situation implizit eine niedrigere Priorität erhält als der Thread, welcher die Ressource besitzt. Wenn ein oder mehrere mittelpriore Threads den niederprioren Thread mit der Ressource unterbrechen, verzögert dies die Ausführung des hochprioren Threads weiter. Eine Abhilfe kann hier die Prioritätsvererbung bieten, dadurch wird der hochpriore Thread nur mehr von dem niederprioren Thread mit der Ressource, dieser besitzt nun die gleiche Priorität wie der hochpriore Thread, blockiert und nicht mehr von den mittelprioren Threads.

[29] In eine Queue (engl. für Warteschlange) können Daten von einem Thread hineingeschrieben und von einem oder mehreren Threads gelesen werden. Dabei kann sowohl der lesende Thread, wenn die Queue leer ist, und der schreibende Thread, wenn die Queue voll ist, blockieren. Queues können auch als Thread-sichere First In/First Out (FIFOs) betrachtet werden, wobei in manchen Betriebssystemen eine Queue sowohl als FIFO als auch als Last In/First Out (LIFO) oder als Mischung aus beiden benutzt werden kann.

Semaphore zu benötigen, jedoch kann damit nur ein einziger Thread blockieren, und zwar jener, welcher das Array besitzt. Wenn die Nachricht vom Thread ausgelesen wird, wird der Status von Unbearbeitet auf Bearbeitet gesetzt. Es gibt vier Möglichkeiten, wie der Wert einer solchen Nachricht gesetzt werden kann:

- Die Nachricht wird komplett geschrieben, egal ob der Wert bereits gelesen wurde oder nicht.
- Die Nachricht wird komplett geschrieben, falls der Wert bereits gelesen wurde und nicht geschrieben, wenn der Wert noch nicht gelesen wurde.
- Es werden ein oder mehrere Bits in der Nachricht gesetzt.
- Es wird der Wert der Nachricht um eins inkrementiert.

Der Vorteil von Direct-To-Thread Benachrichtigungen gegenüber der Verwendung von vermittelnden Objekten ist, dass diese um bis zu 45 % schneller sind und weniger RAM benötigen.[30]

- **Stream- und Nachrichten-Puffer** sind, im Gegensatz zu den anderen Inter-Thread-Kommunikationsmechanismen, für genau einen Schreiber und einen Leser pro Puffer vorgesehen. Sie verwenden den Array-Index Null der Direct-To-Thread Benachrichtigungen, um Daten per Kopie zu versenden. Bei den Stream-Puffern werden Daten als kontinuierlicher Datenstrom versendet. Sind keine Daten im Puffer vorhanden, so wird der Thread beim Lesevorgang blockiert, ist der Puffer voll, so wird der Thread beim Schreibvorgang blockiert. Es kann auch ein Trigger-Level definiert werden, bis zu welchem der Lesevorgang blockiert wird, damit ein Thread nicht für jedes Byte extra ausgeführt und dann wieder blockiert wird. Die Nachrichten-Puffer bauen auf den Stream-Puffern auf, es werden jedoch diskrete Nachrichten mit variabler Länge versendet und diese müssen immer als gesamte Nachricht gelesen werden. Dazu wird als erstes Datenelement die Länge als 32-Bit-Wert mitgesendet.

- **Event[31]-Bits (oder -Flags) und Event-Gruppen:** Eine Event-Gruppe ist ein Set an Event-Bits auf welche mittels einer Bit-Nummer zugegriffen werden kann. Threads können eine oder mehrere Event-Bits in einer Event-Gruppe gleichzeitig behandeln. Das bedeutet, dass ein oder mehrere Event-Bits gleichzeitig gesetzt oder gelöscht werden können und vor allem, dass ein Thread auf das aktiv Werden von einem oder mehreren Event-Bits warten kann und somit Events miteinander verknüpft werden können. Dadurch können sogenannte Thread-Rendezvous ermöglicht werden, bei welchen zwei oder mehrere Threads darauf warten, dass alle diese Threads ihre Aufgaben abgearbeitet haben und den Rendezvous-Punkt erreicht haben.

[30] „Measured using the binary semaphore implementation from FreeRTOS V8.1.2, compiled with GCC at -O2 optimisation, and without `configASSERT()` defined. A 35 % improvement can still be obtained using the improved binary semaphore implementation found in FreeRTOS V8.2.0 and higher." [83]

[31] Ein Event (engl. für Ereignis) ist ein Synchronisierungsmechanismus zwischen Threads, bei denen ein oder mehrere Threads auf ein oder mehrere Ereignisse warten, welche durch andere Threads verursacht werden.

Eine weitere, wichtige Funktionalität eines RTOS sind die Software-Timer[32]. In FreeRTOS werden Software-Timer hauptsächlich dazu verwendet, um Funktionen zu einem bestimmten Zeitpunkt in der Zukunft auszuführen. Die Funktionen, welche vom Software-Timer ausgeführt werden, werden Callback-Funktionen[33] genannt. Bei der internen Implementierung von Software-Timern in FreeRTOS wurde besonders darauf geachtet, dass diese möglichst effizient sind. Daher werden die Callback-Funktionen nicht in einem Interrupt ausgeführt, es wird keine Rechenzeit für laufende Software-Timer verbraucht (erst wenn die eingestellte Zeit erreicht ist), es wird keine zusätzliche Rechenzeit im Interrupt des System-Timers benötigt und es werden keine verketteten Listen durchlaufen, während die Interrupts deaktiviert sind. Es werden jedoch alle Callback-Funktionen im selben Thread, dem Software-Timer-Thread, ausgeführt. Daher ist es wichtig, dass die Callback-Funktionen keine blockierenden Aufrufe verwenden und die Priorität des Software-Timer-Threads richtig gewählt wird. Der Software-Timer-Thread ist nicht nur für die Ausführung der Callback-Funktionen zuständig, sondern auch für das Aufsetzen und Konfigurieren von neuen Software-Timern. Da es sein kann, dass ein Thread mit einer höheren Priorität als der Software-Timer einen solchen aufsetzen möchte, berechnet sich der Zeitpunkt für das Ablaufen des Software-Timers relativ zum Zeitpunkt, wenn das Kommando zum Aufsetzen des Software-Timers gesendet wird und nicht wenn das Kommando tatsächlich bearbeitet wird. Für die Größe des Stacks des Software-Timer-Threads sind die Stackanforderungen der einzelnen Callback-Funktionen zu beachten, da diese in diesem Thread ausgeführt werden. Es gibt zwei verschiedene Arten an Software-Timern in FreeRTOS: singuläre Timer (engl. one-shot timer) und periodische Timer. Bei singulären Timern wird die Callback-Funktion nur einmal ausgeführt und muss dann manuell neu gestartet werden, falls der Timer nochmals laufen soll. Im Gegensatz dazu werden periodische Timer automatisch neu gestartet und die Callback-Funktion wird regelmäßig mit der eingestellten Periodendauer ausgeführt [82].

FreeRTOS bietet verschiedene Möglichkeiten, wie dynamischer Speicher aus dem Heap allokiert werden kann. Neben den fünf bereitgestellten Allokationsmechanismen bietet FreeRTOS auch die Möglichkeit eigene Speichermanagementalgorithmen zu verwenden. Es können auch zwei unterschiedliche Allokationsmechanismen gleichzeitig verwendet werden. Damit kann zum Beispiel der schnellere, interne RAM für den Stack und die Objekte des Betriebssystems und ein langsamerer, externer RAM für die Speicherung der Daten der Applikation genutzt werden. Die fünf Allokationsmechanismen von FreeRTOS sind [82]:

[32] Software-Timer sind Timer, welche die Zeitbasis des System-Timers verwenden, um Ereignisse nach definierbaren Zeiten zu generieren. Dazu wird keine zusätzliche Hardware benötigt.

[33] Eine Callback-Funktion (engl. für Rückruffunktion) ist eine Funktion, welche zu einem bestimmten Teil in der Software gehört und von anderen Teilen der Software ausgeführt werden kann, um dem Softwareteil der Callback-Funktion etwas mitzuteilen (zum Beispiel, dass ein Event aufgetreten ist oder dass Daten zur Verarbeitung vorhanden sind). Durch die Verwendung von Callback-Funktionen muss die Software, die die Callback-Funktion besitzt, nicht blockierend warten.

1. Einfachste Variante, mit der Speicher nur allokiert und nicht wieder freigegeben werden kann. Dadurch kann es zu keiner Fragmentierung des Speichers kommen und die Zeit für die Allokation eines Speicherblocks ist statisch.
2. Speicher kann allokiert und freigegeben werden, aber freigegebene, benachbarte Blöcke werden nicht zu einem Block zusammengefasst. Es kann zu Speicherfragmentierung kommen, wenn immer wieder Speicherblöcke unterschiedlicher Größe allokiert werden.
3. Wrapper für die Standard-C-Funktionen, `malloc()` und `free()` zur Allokation und Freigabe von Speicherblöcken, damit diese Thread-sicher werden.
4. Speicher kann allokiert und freigegeben werden und freigegebene, benachbarte Blöcke werden wieder zu einem Block zusammengefasst, damit die Fragmentierung des Speichers reduziert wird.
5. Ebenso wie bei 4. kann Speicher allokiert und freigegeben werden und freigegebene, benachbarte Blöcke werden wieder zu einem Block zusammengefasst, jedoch muss der Heap nicht ein zusammenhängender Block sein, sondern kann in mehrere, nicht zusammenhängende Bereiche unterteilt sein.

Bei Betriebssystemen mit Multithreading besitzt jeder Thread seinen eigenen Stack, auf dem lokale Variablen abgelegt werden, wenn diese keinen Platz in den Registern haben oder eine Unterfunktion aufgerufen wird. Außerdem müssen die Registerwerte gespeichert werden, wenn der Thread unterbrochen wird und ein anderer Thread ausgeführt wird. Wenn viele Unterfunktionen aufgerufen und/oder viele lokale Variablen verwendet werden, kann es dazu kommen, dass der allokierte Speicher für den Stack zu klein wird. Solche Stack-Überläufe (engl. stack overflow) sind Fehler, welche sehr schwer zu finden und zu reproduzieren sind, da ihre Auswirkungen davon abhängen, welcher und wie viel Speicher außerhalb des Stacks überschrieben wird. Um das Suchen nach solchen Fehlern zu vereinfachen, bietet FreeRTOS zwei unterschiedliche Varianten für einen Stack-Check an [82]:

- **Methode 1:** Diese Methode trifft die Annahme, dass der Stack am vollsten ist, nachdem der Thread unterbrochen wurde, um einen anderen Thread auszuführen. Zu diesem Zeitpunkt enthält der Stack neben den normalerweise gespeicherten, lokalen Variablen auch den Zustand des Threads mit den Werten, welche in den Registern gespeichert waren. Zu diesem Zeitpunkt überprüft FreeRTOS, ob der Wert des Stackpointers[34] noch innerhalb des gültigen Stackbereiches liegt. Liegt der Stackpointer außerhalb des gültigen Stackbereiches so wird eine Fehlerfunktion aufgerufen, welche vom Entwickler geschrieben wird und Informationen über den Thread mit dem übergelaufenen Stack beinhaltet. Diese Methode des Stack-Checks ist sehr schnell, kann jedoch nicht alle möglichen Stack-Überläufe erkennen.

[34]Der Stackpointer (engl. für Zeiger auf den Stack) ist ein Register oder eine Variable, welches die Speicheradresse des nächsten Elements am Stack beinhaltet.

- **Methode 2:** Bei dieser Methode wird der Stack mit einem bekannten Wert befüllt, wenn der Thread erzeugt wird. Bevor ein Thread unterbrochen wird, überprüft FreeRTOS, ob die letzten 16 Bytes im Stackbereich noch den ursprünglichen Wert besitzen oder bereits überschrieben worden sind. Wurde mindestens ein Byte dieser 16 Bytes überschrieben, so wird die Fehlerfunktion aufgerufen. Da bei jedem Wechsel zu einem neuen Thread 16 Bytes überprüft werden müssen, ist diese Methode nicht so effizient wie Methode 1, es ist jedoch wahrscheinlicher, aber nicht garantiert, dass diese Methode den Stack-Überlauf erkennt.

Diese Methoden für den Stack-Check benötigen zusätzliche Rechenzeit, wenn von einem Thread in einen anderen gewechselt wird. Daher ist die Aktivierung des Stack-Checks nur während des Entwicklungsprozesses zu empfehlen und sollte bei der fertigen Software entfernt werden [82].

2.3.2 Keil RTX

Keil RTX ist ein deterministisches Echtzeitbetriebssystem mit Multithreading für Arm Cortex-M Geräte. Es ist als Open-Source-Software unter der Apache 2.0 Open-Source-Lizenz verfügbar und bietet alle benötigten Funktionalitäten eines Embedded Betriebssystems, wie dynamisches Speichermanagement, unbegrenzte[35] Anzahl an Software-Timern und Inter-Thread-Kommunikation, mit einer unbegrenzten[35] Anzahl an Semaphoren, Mutexes und Queues. Eine Anwendung mit Keil RTX kann eine unbegrenzte[35] Anzahl an Threads mit 254 unterschiedlichen Prioritäts-Levels erstellen, welche per präemptivem, Round-Robin- oder kooperativem Scheduling abgearbeitet werden können. Der verwendete Scheduling Algorithmus kann entweder eine dieser genannten Varianten oder einer Kombination aus präemptivem Scheduling mit Round-Robin- oder kooperativem Scheduling sein. So kann zum Beispiel präemptives Scheduling verwendet werden und wenn zwei Threads die gleiche Priorität besitzen, werden diese mit Round-Robin-Scheduling im Zeitmultiplex-Verfahren abgearbeitet. Eine andere Variante ist, dass Threads gleicher Priorität per kooperativem Scheduling abgearbeitet werden und die Threads die Rechenzeit selbstständig abgeben müssen. Round-Robin- und kooperatives Scheduling schließen sich gegenseitig aus und können daher nicht kombiniert werden. Da es für Mikrocontroller entwickelt wurde, lässt sich der benötigte, nicht-volatile Programmspeicher auf bis zu 5 KB reduzieren [85] [86].

Abbildung 2.13: Keil RTX [84]

[35] In der Theorie ist die Anzahl nicht durch Keil RTOS begrenzt. In der Praxis ist jedoch der Speicher begrenzt und somit auch die tatsächlich verwendbare Anzahl.

Threads, Software-Timer, Semaphoren, Mutexes, Queues und Event-Flags benötigen Speicher im RAM. Keil RTX bietet drei verschiedene Möglichkeiten, welche auch kombiniert werden können, an, wie der Speicher für diese Objekte im RAM organisiert wird [87]:

- **Globaler Speicher-Pool:** Keil RTX verwendet standardmäßig diesen Algorithmus. Hier wird der Speicher für alle Objekte aus einem Speicher-Pool entnommen und die Ausführungszeit hierfür ist nicht deterministisch. Wenn Speicher für unterschiedlich große Objekte allokiert wird und später wieder im Pool freigegeben wird, so kommt es zu Speicherfragmentierung.
- **Objektspezifischer Speicher-Pool:** Hierbei wird jeweils ein eigener Speicher-Pool für jeden Objekt-Typ verwendet, um Speicherfragmentierung zu verhindern. Dadurch können die Blöcke in jedem Speicher-Pool eine fixe, aber unterschiedliche Größe besitzen und dies macht die Zeit für das Allokieren von Speicher konstant und deterministisch.
- **Statischer Objektspeicher:** Für alle benötigten Objekte wird schon während der Kompilierung Speicher reserviert und somit verhindert, dass Speicherfragmentierung auftritt oder kein Speicher mehr allokiert werden kann.

Jeder Thread in Keil RTX besitzt seinen eigenen Stack, auf welchem die lokalen Variablen gespeichert werden und die Register des Prozessors gespeichert werden, wenn ein anderer Thread ausgeführt wird. Die Größe des Stacks kann für jeden Thread individuell gewählt werden und Keil RTX bietet auch einen konfigurierbaren Stack-Check, welcher nicht nur Stack-Überläufe, sondern auch die laufende Stack-Auslastung überprüft [87].

2.3.3 Micro-Controller Operating Systems-III

Micro-Controller Operating Systems-III (μC/OS-III) ist ein Embedded Betriebssystem, welches von der Firma Micriμm entwickelt wurde und jetzt von der Firma Silicon Labs, welche Micriμm aufgekauft hat, gewartet wird. μC/OS-III ist als Open-Source-Software unter der Apache 2.0 Open-Source-Lizenz verfügbar und nahezu der gesamte Code ist in ANSI-C geschrieben [89]. Es handelt sich um ein deterministisches Echtzeitbetriebssystem mit Multithreading und unlimitierter Anzahl an Thread-Prioritäten. Der Scheduler ist ein präemptiver Scheduler, welcher Threads mit gleicher Priorität mittels Round-Robin-Scheduling bearbeitet. Da es sich um ein Betriebssystem für Mikrocontroller handelt, wurde besonderes Augenmerk auf die Portierbarkeit und Skalierbarkeit (Speicherbedarf: 6 KB - 24 KB nicht-volatiler Programmspeicher und 2 KB - 5 KB RAM [90]) gelegt [91]. Dadurch gibt es viele

Abbildung 2.14: μC/OS-III [88]

unterschiedliche Mikrocontroller von vielen verschiedenen Herstellern für die μC/OS-III bereits angepasst wurde. Eine komplette Liste an Hersteller, Mikrocontroller und Compiler kann hier [92] gefunden werden. Einige wichtige sind in Tabelle 2.2 aufgelistet.

Tabelle 2.2: Wichtige Hersteller und Mikrocontroller mit Portierungen von μC/OSIII [90] [92]

Hersteller	Mikrocontroller	Compiler
Altera (Intel)	Nios II SoC FPGA (Cortex-A)	GNU
Arm	ARM7, ARM9, ARM11, Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17 Cortex-A53, Cortex-A57, Cortex-R4, Cortex-R5, Cortex-R7, Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4(F), Cortex-M7	Arm, CCS, GNU, IAR, RealView
Atmel (Microchip)	AVR, AVR32, SAM3, SAM4, SAM7, SAM9, SAMA5 (Arm Cortex-based)	IAR, C30
Infineon/ Cypress Semiconductors	PSoC 4, PSoC 5 (Cortex-M), XMC4000 (Cortex-M)	Arm, CCS, GNU, IAR, RealView
NXP	ColdFire, HCS12, i.MX, Kinetis (Cortex-M), LPC (Cortex-M), LPC (ARM7 / ARM9) MPC5xxx, MPC8xxx, VFxxx (Cortex-A & Cortex-M)	CodeWarrior, GNU, IAR
STM	STM32F (Cortex-M), STM32L (Cortex-M), STM32H (Cortex-M), STR9	Arm, CCS, GNU, IAR, RealView
Texas Instruments	C28x, MSP430 (Cortex-M), MSP432 (Cortex-M), Hercules RM (Cortex-R), Hercules TMS570 (Cortex-R)	Arm, CCS, GNU, IAR, RealView
Xilinx	MicroBlaze, Zynq-7000 (Cortex-A), Zynq Ultrascale+ MPSoC (Cortex-A & Cortex-R)	Arm, CCS, GNU, IAR, RealView

μC/OS-III ist die dritte Generation des Mikrocontroller-Betriebssystems von Micriμm und beinhaltet alle wichtigen Funktionalitäten eines Embedded Betriebssystems, wie unter anderem Software-Timer, Ressourcen-Management, Thread-Management, Thread-Synchronisation und Inter-Thread-Kommunikation mittels Semaphoren, Event-Flags, Mutexes und Queues. Darüber hinaus besitzt es noch weitere Funktionalitäten, welche in vielen anderen Embedded Betriebssystemen nicht vorhanden sind, wie zum Beispiel Performance-Messungen während der Laufzeit und Blockieren auf mehrere Semaphoren, Event-Flags oder Mutexes. Außerdem können ein Dateisystem (μC/FS), ein TCP/IP-Stack (μC/TCP-IP), ein GUI (μC/GUI), ein USB-Stack (μC/USB) für USB-Device, USB-Host und USB-On-The-Go (OTG) und andere Funktionalitäten optional hinzugefügt werden [91].

Im Gegensatz zu seinem Vorgänger μC/OS-II können in μC/OS-III eine unbegrenzte[36] Anzahl an Threads mit einer unbegrenzten[36] Anzahl an Prioritäten erstellt werden. Jeder Thread besitzt seinen eigenen Stack mit beliebiger[37] Größe und mit der Möglichkeit, den tatsächlich benötigten Platz am

[36] In der Theorie ist die Anzahl nicht durch μC/OS-III begrenzt. In der Praxis ist jedoch der Speicher begrenzt und somit auch die tatsächlich verwendbare Anzahl.

[37] Nach oben nur durch den physikalisch verfügbaren Speicher begrenzt und nach unten begrenzt durch die Prozessor-Architektur, da die Register des Prozessors in den Stack geschrieben werden müssen, um einen anderer Thread auszuführen.

Stack während der Laufzeit zu überwachen. Die Anzahl der Semaphoren, Event-Flags, Mutexes, Queues und Software-Timer ist ebenfalls unbegrenzt[36] [91].

In μC/OS-III kommt es zu keinem Problem, wenn Zugriffe auf Mutexes verschachtelt werden. Ein Thread kann bis zu 250 Mal dieselbe Mutex anfordern. Dies erleichtert das Schreiben eines Programms, da der Entwickler sich keine Gedanken darüber machen muss, ob eine aufgerufene Funktion dieselbe Mutex anfordert, welche bereits von der aufrufenden Funktion angefordert ist. Wichtig ist nur, dass die Mutex genau so oft freigegeben wird, wie sie angefordert wurde. Das ist aber kein Problem, da dazu kein Wissen über die aufgerufenen Funktionen benötigt wird, weil jede Funktion für sich alleine für das Freigeben der in dieser Funktion verwendeten Mutexes verantwortlich ist. Die Pausierung eines Threads kann ebenfalls verschachtelt werden und der pausierte Thread muss wieder genau so oft fortgesetzt werden, wie er pausiert wurde. Ein Thread kann sich selbst pausieren oder von einem oder mehreren anderen Threads bis zu 250 Mal pausiert werden. Wenn ein Thread pausiert ist, so wird er nicht mehr ausgeführt, bis dieselbe Anzahl an Aufrufen zum Fortsetzten des Threads getätigt wurde. Wenn ein Thread sich selbst pausiert, kann sich dieser nicht mehr selbst fortsetzen, da er keine Rechenzeit mehr bekommt. Das heißt, dass er von einem anderen Thread fortgesetzt werden muss, was dazu führen kann, dass der pausierte Thread nie wieder fortgesetzt wird [91].

Der System-Timer in μC/OS-III generiert periodisch einen Interrupt, welcher ein Signal/Event an den Tick-Thread sendet, damit dieser wieder ausgeführt wird. Dadurch wird die Zeit, in welcher sich das System in eine Interrupt-Routine befindet, stark reduziert und andere Interrupts bzw. hochpriore Threads werden nicht verzögert. In diesem Tick-Thread werden die Listen der Threads, welche auf das Ablaufen eines Software-Timers oder eines Time-outs einer blockierenden Funktion warten, durchlaufen. Um das Verarbeiten dieser Listen zu beschleunigen, verwendet μC/OS-III einen Hashed-Delta-List-Mechanismus. Wie bei FreeRTOS gibt es zwei verschiedene Arten von Software-Timern: singuläre Timer (engl. one-shot timer) und periodische Timer. Software-Timer sind als Abwärtszähler (engl. countdown counters) implementiert und wenn ein solcher abläuft, also den Wert Null erreicht, wird eine Callback-Funktion im Tick-Thread ausgeführt. Aus diesem Grund dürfen keine blockierenden Funktionen oder Funktionen, welche den Thread stoppen oder löschen, in diesen Callback-Funktionen verwendet werden. Für Zeitmessungen benötigt μC/OS-III einen 16-Bit oder 32-Bit frei laufenden Hardware-Timer. Der Wert dieses Timers kann ausgelesen und als Zeitstempel (engl. timestamp) verwendet werden. Diese Zeitstempel werden automatisch mitgesendet, wenn eine Semaphore oder eine Mutex freigegeben oder ein Event-Flag gesetzt oder eine Nachricht in einer Queue gesendet wird. Wenn dieser Zeitstempel empfangen wird, kann dieser mit dem aktuellen Zeitstempel verglichen werden und somit die Zeit berechnet werden, welche zwischen dem Freigeben/Setzen/Senden und dem aktiv Werden des Threads vergangen ist [91].

In μC/OS-III ist es auch möglich, auf mehrere unterschiedliche Semaphoren und/oder Queues gleichzeitig zu warten, jedoch nicht auf Event-Flags oder Mutexes. Ein wartender Thread wird wieder aktiviert, wenn mindestens eine seiner Semaphoren freigegeben oder eine Nachricht in einer seiner Queues gesendet wird. Damit kann zum Beispiel erreicht werden, dass ein Thread auf Nachrichten in verschiedenen Queues von verschiedenen Threads und auf die Semaphore einer Kommunikationsschnittstelle, über welcher die Nachrichten in einem organisierten Schema (Zum Beispiel kann eine Priorisierung der Nachrichten in Abhängigkeit der Queue, in welcher sie sich befinden, vorgenommen werden.) übertragen werden sollen, wartet [91].

Nachrichten und Events können auch direkt an einen bestimmten Thread gesendet werden. Hierbei werden keine Semaphoren, Event-Flags oder Queues benötigt und dies resultiert in einer besseren Performance [91].

Alle Funktion von μC/OS-III besitzen zusätzliche Laufzeitüberprüfungen für die Übergabeparameter. Unter anderem wird überprüft, ob Zeiger auf die Adresse Null (null pointer) übergeben werden, ob Funktionen in einer Interrupt-Routine aufgerufen werden, welche dort nicht aufgerufen werden dürfen, ob die Parameter in einem gültigen Wertebereich liegen und ob gültige Konfigurationsoptionen übergeben werden. Tritt ein Fehler auf, so wird die Funktion verlassen und ein Fehler-Code zurückgegeben. Diese Überprüfungen können deaktiviert werden, um die Code-Größe zu verringern und die Performance zu erhöhen [91].

μC/OS-III bietet auch einige Funktionalitäten, welche das Entwickeln und vor allem das Debuggen vereinfachen [91]:

- **Performance-Messungen:** Es können zur Laufzeit unter anderem die Ausführungszeit jedes einzelnen Threads, dessen Stack-Auslastung, wie oft der Thread vom Scheduler aktiviert wurde, die Auslastung des Prozessors gesamt und pro Thread, die Zeit, in der die Interrupts deaktiviert waren und die Zeit, in der der Scheduler gesperrt war gesamt und pro Thread, die Zeit, welche benötigt wird, um von einer Interrupt-Routine zu einem Thread und zwischen zwei Threads zu wechseln, gemessen werden.
- **Unterstützung von Kernel-Aware-Debugging:** Gemeinsam mit einem Kernel-Aware-Debugger können Variablen und Datenstrukturen von μC/OS-III benutzerfreundlich dargestellt werden, wenn der Prozessor an einem Breakpoint[38] angehalten wird. Wird als Debugger die μC/Probe verwendet, so können gewisse Daten auch zur Laufzeit dargestellt werden, ohne dass der Prozessor angehalten werden muss.

[38] Ein Breakpoint ist ein spezielles Assembler-Kommando, bei welchem der Prozessor anhält und keine weiteren Operationen ausführt. Mit einem Debugger können dann Daten (Werte in den Registern, im Stack und im Speicher) vom Mikrocontroller geladen und spezielle Kommandos gesendet werden, um Befehle einzeln abzuarbeiten oder das Programm normal weiterlaufen zu lassen.

- **Benennung von μC/OS-III-Objekten:** Es können für Threads, Semaphoren, Mutexes, Event-Flag-Gruppen, Queues und Software-Timer Namen vergeben werden, welche in diesen Objekten gespeichert werden. Diese Namen bestehen aus American Standard Code for Information Interchange (ASCII)-Zeichen mit beliebiger Länge und müssen mit Null terminiert sein.

In μC/OS-III ist es zwar möglich, die C-Funktionen `malloc()` und `free()` zur dynamischen Speicherallokierung zu verwenden, aber es ist nicht empfohlen, da die Ausführungszeit dieser Funktionen nicht deterministisch ist und es zu Speicherfragmentierung kommen kann. Aus diesem Grund gibt es einen eigenen Algorithmus, um dynamisch Speicher zu allokieren. Dieser liefert Speicherblöcke fixer Größe und besitzt dadurch eine konstante und somit deterministische Ausführungszeit und es kann zu keiner Fragmentierung kommen. Der Nachteil ist, dass meistens mehr Speicher in einem Block ist als für das dynamisch allokierte Objekt benötigt wird. Um diesen Nachteil zu minimieren, kann in μC/OS-III eine unbegrenzte[39] Anzahl an Partitionen für den dynamischen Speicher erstellt werden, wobei jede Partition eine unterschiedliche, aber fixe Blockgröße besitzt. Wenn mehrere Partitionen verwendet werden, ist es wichtig, dass ein Speicherblock an dieselbe Partition freigegeben wird, aus der er allokiert wurde [91].

2.4 Fazit

Da jetzt ein Überblick über die verschiedensten Betriebssystemtypen und den wichtigsten Vertretern dieser Betriebssystemtypen gegeben wurde, kann betrachtet werden, ob es bereits eine Lösung für die Problemstellung gibt beziehungsweise welcher Betriebssystemtyp sich am besten eignet, um die Problemstellung zu lösen.

Alle betrachteten PC-Betriebssysteme und deren Embedded Varianten benötigen mehr Speicher und eine höhere Prozessorfrequenz, als der Referenz-Mikrocontroller besitzt, und sind somit für diesen Einsatzbereich nicht geeignet.

Bei den virtuellen Maschinen würden sich Prozess virtuelle Maschinen hervorragend dafür eignen, unabhängige Applikationsmodule auf einem Betriebssystem auszuführen. Solche Prozess virtuellen Maschinen bedeuten jedoch einen nicht unerheblichen Mehraufwand für den Prozessor, da die Kompilierung erst am Mikrocontroller stattfindet und diese auch Rechenzeit benötigt, was bei Echtzeitsystemen zu Problemen führen kann. Ein weiteres Problem, vor allem bei der Java virtuellen Maschine, ist, dass eine zu grobe Abstraktion vorgenommen wird und dass, für Anwendungen mit Echtzeitanforderungen, auf wichtige Teile, wie dem Scheduling und Priorisieren von Threads, kein Einfluss

[39] In der Theorie ist die Anzahl nicht durch μC/OS-III begrenzt. In der Praxis ist jedoch der Speicher begrenzt und somit auch die tatsächlich verwendbare Anzahl.

genommen werden kann.

Daher bleiben nur mehr die Embedded Betriebssysteme übrig, für welche eine Erweiterung entwickelt werden muss. Diese Erweiterung muss es ermöglichen, Applikationsmodule zu installieren und auszuführen. Des Weiteren muss eine Abstraktionsebene definiert und implementiert werden, welche das Betriebssystem von den Applikationsmodulen trennt und als Grundlage für den Schutz des Betriebssystems vor den Applikationsmodulen und den Applikationsmodulen untereinander dient.

Kapitel 3

Konzept und Umsetzung

Nachdem ein Überblick über viele verschiedene Betriebssysteme und virtuelle Maschinen geschaffen wurde, kann nun die Umsetzung dieser Arbeit konkretisiert werden. Ziel ist es, ein Embedded Betriebssystem für einen ressourcenarmen Mikrocontroller[1] zu schaffen, bei dem Programme ähnlich wie bei einem PC installiert, ausgetauscht und ausgeführt werden können. Dafür muss zuallererst eine Trennung zwischen den Applikationsmodulen und dem Betriebssystem geschaffen werden, da, wie in Abbildung 3.1 zu sehen ist, die meisten Embedded Projekte beim Kompilieren und Linken zu einer einzigen Binär-Datei zusammengeführt werden. Zusätzlich zu dieser Trennung muss ein API hinzugefügt werden, um den Applikationsmodulen Zugriff auf Betriebssystemfunktionen und Hardware-Funktionalitäten zu bieten. Diese Zugriffe sollen vom Betriebssystem überwacht und kontrolliert werden, damit keine ungültigen Werte verwendet werden und somit der Schutz des Betriebssystems, der Hardware und der anderen Module umgangen werden kann. Um das Betriebssystem vor den Modulen und die Module gegeneinander zu schützen, dürfen die Applikationsmodule nicht im privilegierten Modus, dem Kernel-Space, ausgeführt werden. Daher muss der Scheduler in den unprivilegierten Modus, dem User-Space, schalten, wenn der Thread eines Applikationsmoduls ausgeführt werden soll. Außerdem muss

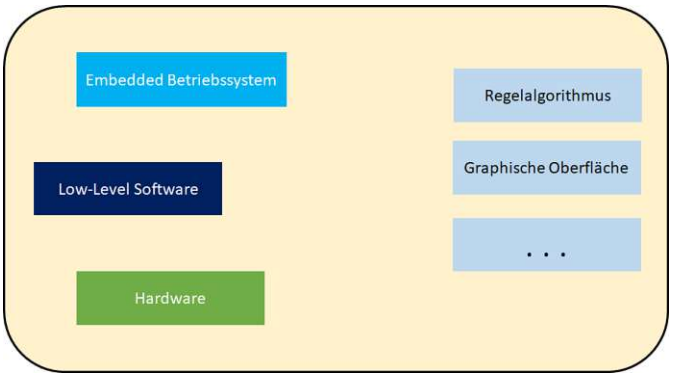

Abbildung 3.1: Gliederung der Software im Mikrocontroller: Mehrere Kompilier-Einheiten werden in ein einziges Binär-File zusammengefügt

[1] Hierfür soll der bereits erwähnte Mikrocontroller STM32L476VG (80 MHz, 1 MB Flash als nicht-volatiler Programmspeicher und 128 kB SRAM) die Referenz sein. Hierbei handelt es sich um einen Mikrocontroller auf Basis des Cortex-M4 Designs von Arm.

verhindert werden, dass Applikationsmodule über Zeiger direkt auf Speicherbereiche zugreifen, für welche sie keine Zugriffsrechte besitzen. Des Weiteren muss ein Weg geschaffen werden, über welchen die Applikationsmodule auf die Geräte geladen, gelöscht oder upgedatet werden können. Da es sich bei den Applikationsmodulen auch um Software handelt, muss eine Möglichkeit geschaffen werden, diese Software auch debuggen zu können. Dabei ist darauf zu achten, dass dadurch nicht das Betriebssystem blockiert und kein Speicher verändert wird, der zu dem geschützten Bereich des Betriebssystems und der Hardware gehört.

3.1 Application Programming Interface

Wie in Abbildung 3.2 zu sehen, werden die Applikationsmodule vom Betriebssystem getrennt und als eigenständige Projekte kompiliert, damit entstehen eigene Binär-Dateien für das Betriebssystem und jedes Applikationsmodul. Damit diese weiterhin Funktionalitäten des Betriebssystems und der Hardware verwenden können, wird ein API als Kommunikationsschnittstelle eingeführt.

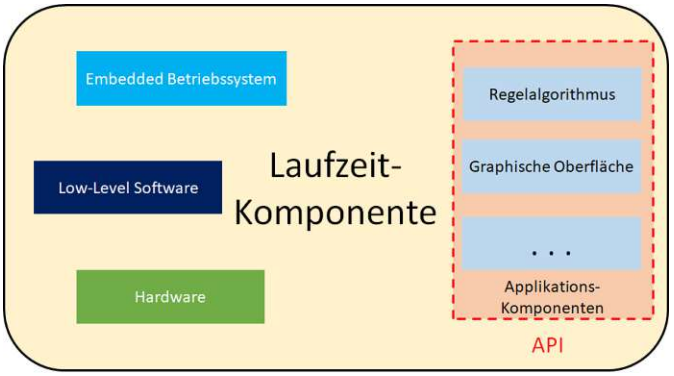

Abbildung 3.2: Gliederung der Software im Mikrocontroller: Mehrere eigenständige Module mit API

Dieses API soll möglichst allgemein gehalten sein, damit sowohl Rückwärts- als auch Vorwärtskompatibilität ermöglicht wird. Um das zu garantieren, ist es von Vorteil, das API von den Funktionalitäten des Betriebssystems und der Hardware weitestgehend zu trennen. Hierfür bieten sich „Geräte-Dateien“ (engl. device files) an. Bei diesen „Geräte-Dateien“ handelt es sich um keine echten Dateien in einem Speichermedium, sondern um Pseudo-Dateien, welche vom Programmierer geöffnet, geschrieben, gelesen und geschlossen werden können, als ob es normale Dateien wären. So können externe Geräte, aber auch Kommunikationsschnittstellen, vom Programmierer behandelt werden, als wären sie Dateien im lokalen Speicher. Das Betriebssystem weiß jedoch, dass hinter der Pseudo-Datei ein Gerät oder eine Kommunikationsschnittstelle steht und muss die Daten des Programmierers dem Gerät oder der Kommunikationsschnittstelle entsprechend verarbeiten. Diese Idee, alles wie eine Datei zu behandeln, stammt von UNIX.

Funktionsdefinitionen für Datei-Operationen gibt es viele verschiede und obwohl sie sich sehr ähneln, gibt es Unterschiede in den Parametern und den Konfigurationsmöglichkeiten. So bietet zum Bei-

spiel C/C++ die in Programmcode 3.1 aufgelisteten Funktionen zum Öffnen, Schließen, Lesen, Schreiben und Umsetzen des Lese-Schreib-Zeigers an.

Programmcode 3.1: Datei-Operationen in C/C++ [93]

```
FILE* fopen( const char *filename, const char *mode );
int fclose( FILE *stream );
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
int fseek( FILE *stream, long offset, int origin );
```

Wobei `FILE` ein Objekt ist, welches eine Datei repräsentiert, `filname` der Name der Datei inklusive Pfad ist, `mode` der Modus ist, wie die Datei geöffnet wird („r“ nur lesen; „w“ nur schreiben, existierende Dateien werden gelöscht; „a“ nur schreiben, existierende Dateien werden nicht gelöscht; „r+“ lesen und schreiben; „w+“ lesen und schreiben, existierende Dateien werden gelöscht; „a+“ lesen und schreiben, existierende Dateien werden nicht gelöscht), `buffer` ein Zeiger auf einen Speicherbereich mit Objekten der Größe `size` in Byte und der Objektanzahl `count` ist, `offset` eine Anzahl an Zeichen ist, um welche der Lese-Schreib-Zeiger in der Datei verschoben werden soll, beginnend bei `origin` (SEEK_SET startet am Anfang der Datei, SEEK_CUR startet bei der derzeitigen Position des Lese-Schreib-Zeigers, SEEK_END startet am Ende der Datei und der Lese-Schreib-Zeiger, im Gegensatz zu den anderen beiden Optionen, wird nach vorne verschoben) [93].

Im POSIX-Standard sind ebenfalls Datei-Operationen definiert, welche in Programmcode 3.2 aufgelistet sind.

Programmcode 3.2: Datei-Operationen in POSIX [94]

```
int open(const char *pathname, int flags, ... /* mode_t mode */);
int close(int fd);
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, const void *buffer, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int ioctl(int fd, int request, ... /* argp */);
```

Der erste Unterschied zwischen C/C++ und dem POSIX-Standard ist, dass bei POSIX kein eigenes Objekt für die Repräsentation von Dateien in der Software verwendet wird, sondern ein Integer[2]. Wenn dieser Unterschied und die unterschiedlichen Namen außen vor gelassen werden, so sind die Funktionen `close()` bzw. `fclose()` und `lseek()` bzw. `fseek()` bei beiden gleich und die Funktionen `read()` bzw. `fread()` und `write()` bzw. `fwrite()` unterscheiden sich nur dadurch, dass bei POSIX die Länge der Daten gesamt in Bytes angegeben ist und nicht wie bei C/C++ in Objektgröße und

[2]Ein Integer ist ein Ganzzahl-Datentyp ($\mathbb{Z}$).

Anzahl der Objekte getrennt ist. Die POSIX-Funktion `open()` unterscheidet sich etwas mehr von der C/C++-Funktion `fopen()`, vor allem in Bezug auf die Parameter `flags` und `mode`, wobei `mode` ein optionaler Parameter ist, welcher auch weggelassen werden kann, wenn dieser nicht benötigt wird. Der POSIX-Parameter `flags` entspricht dem C/C++-Parameter `mode` und gibt an, wie die Datei geöffnet werden soll. Es gibt verschiedene Optionen, welche als Bitmaske dargestellt sind und mit einer Bitweise-Oder-Verknüpfung verbunden werden können (Unter anderem: `O_RDONLY` nur lesen, `O_WRONLY` nur schreiben, `O_RDWR` lesen und schreiben; `O_CREAT` Datei wird erstellt, wenn sie nicht bereits existiert, `O_DIRECTORY` zum Bearbeiten/Erstellen von Ordnern, `O_TRUNC` löscht alle Daten in der Datei, wenn diese bereits existiert; `O_APPEND` der Lese-Schreib-Zeiger wird vor jedem Aufruf zu `write()` ans Ende der Datei gesetzt, `O_DIRECT` umgeht den Zwischenspeicher und Daten werden sofort in die Datei geschrieben). Der Parameter `mode` wird nur in Zusammenhang mit `O_CREAT` benötigt und definiert, mit welchen Datei-Zugriffsbeschränkungen (engl. file permissions) die erstellte Datei oder der erstellte Ordner ausgestattet ist. Die Funktion `ioctl()` (ioctl steht für I/O Control) gibt es bei C/C++ nicht, da bei C/C++ nicht vorgesehen ist, die Datei-Operationen für Geräte-Dateien zu verwenden. Mit der Funktion `ioctl()` können bei POSIX-Systemen Geräte-Dateien konfiguriert und Einstellungen vorgenommen werden. Der Parameter `request` definiert, welche Konfigurationen oder Einstellungen in der Datei vorgenommen werden sollen. Der Parameter `argp` kann jeder beliebige Typ sein und hängt von dem Parameter `request` ab. Meistens handelt es sich um einen Integerwert oder einen Zeiger auf eine Struktur mit Daten und kann, wenn er nicht benötigt wird, auch weggelassen werden [94].

Das verwendete API ist stark an die POSIX-Funktionen angelehnt und ist in Programmcode 3.3 zu sehen.

Programmcode 3.3: Verwendetes API

```
int32_t lomo_open(const char *name, int32_t flags, int32_t mode);
int32_t lomo_close(int32_t handle);
int32_t lomo_read(int32_t handle, void *buffer, size_t bytes);
int32_t lomo_write(int32_t handle, const void *buffer, size_t bytes);
int32_t lomo_seek(int32_t handle, size_t offset, int32_t origin);
int32_t lomo_ioctl(int32_t handle, uint32_t function, ioctlArg arg);
int32_t lomo_sleep(int32_t microseconds);
typedef union {
  uint32_t uint32;
  int32_t int32;
  void *ptr;
} ioctlArg;
```

Bis auf einige geänderte Namen, dem Präfix `lomo_` bei den Funktionen, die fixe Anzahl an Funk-

tionsparametern bei `lomo_open()` und `lomo_ioctl()` und der eigene Datentyp für den dritten Parameter von `lomo_ioctl()` sind die Funktionen des APIs gleich wie die Funktionen im POSIX-Standard definiert. Zusätzlich gibt es die Funktion `lomo_sleep()`, welche den Thread für den Wert, spezifiziert im Parameter `microseconds`, pausiert. Diese Funktion ist zwar Teil des APIs, ist aber kein Teil der Operationen für Geräte-Dateien.

Jetzt ist zwar das API definiert, aber es gibt noch keine Verknüpfung zwischen den Applikationsmodulen und dem Betriebssystem, da diese in getrennten Binär-Dateien sind und somit keine Funktionen aus der anderen aufrufen können. Eine Möglichkeit wäre, die Funktionen des APIs im Betriebssystem auf definierte Speicheradressen zu legen und diese Adressen beim Linken der Applikationsmodule fix anzugeben. Dies ist zwar ein einfacher Weg, er bereitet aber einige Probleme. Zum Beispiel würde im Fall eines Funktionsaufrufs des APIs der Thread des Applikationsmoduls weiter laufen und es käme zu einer Speicherverletzung, da noch nicht in den Kernel-Space gewechselt wurde. Außerdem kann es zu Problemen mit fixen Speicheradressen für Funktionen kommen, wenn Funktionalitäten hinzugefügt werden und sich die Code-Größe ändert und dann nicht mehr genügend Platz vor den fixen Adressen ist. Eine nachträgliche Änderung dieser Adressen ist nur schwer möglich, da alle bestehenden Applikationsmodule angepasst werden müssten und somit die Vorwärts-Kompatibilität nicht gegeben wäre. Eine bessere Variante ist ein Software-Interrupt. Bei einem Interrupt werden automatisch die Prozessor-Register auf dem Stack gespeichert und es wird in den Kernel-Space gewechselt. Arm Cortex-M Prozessoren bieten hierfür den Supervisor-Call (SV-Call)-Interrupt an, welcher für solche Anwendungen, für den Wechsel vom User-Space in den Kernel-Space, als Software-Interrupt genutzt werden kann.

3.2 Schutz des Betriebssystems vor Fehlern, Abstürzen und Endlosschleifen der Module

Da nun die Voraussetzungen geschaffen sind, die Applikationsmodule vom Betriebssystem zu trennen, kann eine Schutzschicht, wie in Abbildung 3.3, eingeführt werden. Eine solche Schutzschicht muss aus mehreren unterschiedlichen Schutzmechanismen bestehen. So muss als Allererstes gewährleistet werden, dass jedes Applikationsmodul einen eigenen Speicher-

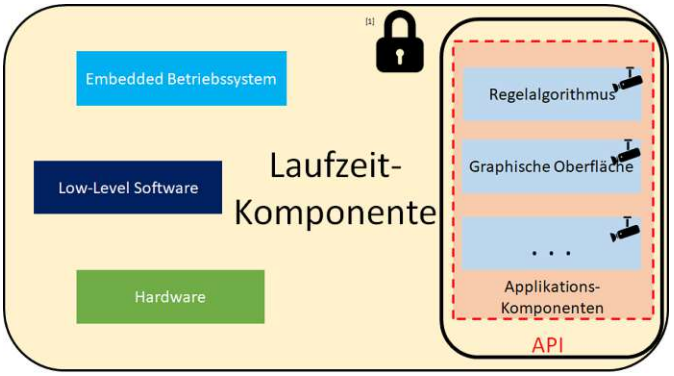

Abbildung 3.3: Schutz des Betriebssystems vor den Applikationsmodulen und den Applikationsmodulen gegeneinander

bereich besitzt und ausschließlich auf diesen Speicherbereich zugreifen kann. So ein Speicherschutz ist kaum in Software zu realisieren und muss somit als Hardware-Funktionalität zur Verfügung stehen. Arm Cortex-M Prozessoren besitzen hierfür eine sogenannte Memory Protection Unit (MPU), welche den erlaubten Speicherbereich für User-Space-Anwendungen einschränken kann und bei Verletzung des Speicherbereichs einen Interrupt generiert.

Die Applikationsmodule können nun nicht mehr direkt auf Funktionalitäten des Betriebssystems und der Hardware zugreifen und all diese Zugriffe müssen über das API geschehen. Damit durch die Zugriffe über das API keine Fehler produziert werden oder über Umwege versucht wird, auf Speicher außerhalb des erlaubten Speicherbereichs zuzugreifen, muss das Betriebssystem bei der Abarbeitung der Befehle darauf achten, dass nur erlaubte Werte und Befehle über das API gesendet werden. Dabei muss nicht nur darauf geachtet werden, dass die übergebenen Werte stimmen, sondern auch darauf, dass es zu keinen Konflikten zwischen zwei oder mehreren Modulen kommt, welche dieselbe Hardware verwenden möchten. Dazu zählt zum einen die Thread-sichere Zugriffskontrolle mit Semaphoren oder Mutexes, zum anderen aber auch die Überprüfung/Speicherung der eingestellten Konfiguration der Hardware, damit es zu keiner Verwendung der Hardware mit einer falschen Konfiguration kommt.

Applikationsmodule können nicht nur über das API Fehler produzieren, sondern es kann auch zu internen Fehlern kommen. Zu diesen Fehlern zählen zum Beispiel ungültige Assemblerbefehle oder inkorrekte Rechenoperationen, wie eine Division durch Null. Solche Fehler lösen bei Mikrocontrollern einen Hardwarefehler aus, welcher ähnlich einem Interrupt funktioniert. Tritt ein solcher Fehler auf, werden die Prozessor-Register auf dem Stack gespeichert, es wird in den Kernel-Space gewechselt und eine spezielle Fehlerfunktion aufgerufen, welche meistens die auf dem Stack gespeicherten Prozessor-Register über eine Debug-Schnittstelle ausgibt und bei einem Breakpoint stehen bleibt oder in einer Endlosschleife gefangen wird. Dieses Verhalten ist nicht wünschenswert, da damit auch das Betriebssystem und alle anderen Applikationsmodule gestoppt werden. Daher müssen diese Fehlerfunktionen überarbeitet werden, sodass nur das fehlerhafte Modul gestoppt wird und das restliche System ohne Beeinträchtigung weiter ausgeführt werden kann.

Ein weiteres Problem, das auftreten kann, sind Endlosschleifen und Applikationsmodule, welche die Rechenzeit nicht mehr abgeben und somit andere Applikationsmodule blockieren können. Um dieses Problem zu lösen, wird die Ausführungszeit überwacht. Des Weiteren müssen die Applikationsmodule eine statische Zeit angeben, mit der die tatsächlich benötigte Ausführungszeit verglichen wird. Mehr dazu in Abschnitt 3.3. Wenn ein Modul zu viel Zeit in Anspruch nimmt, weil es in einer Endlosschleife hängt oder die Rechenzeit nicht abgibt, so soll dies erkannt und das Modul beendet werden.

3.3 Aufbau eines Moduls

Da in vielen Embedded Anwendungen die meisten Funktionalitäten nicht dauerhaft ausgeführt werden müssen, sondern immer nach einer bestimmten Zeit oder wenn ein Ereignis auftritt, soll dies auch mit den Applikationsmodulen möglich sein. Dazu wird ein Applikationsmodul in drei Teile unterteilt: Der erste Teil wird beim Starten einmalig ausgeführt und trifft Vorbereitungen zur Ausführung des zweiten Teils. Der zweite Teil wird zyklisch ausgeführt und beinhaltet die eigentliche Applikation. Im dritten Teil werden Aufräumarbeiten nach Beendigung des Moduls vorgenommen, wie zum Beispiel das Freigeben von dynamisch allokiertem Speicher.

Programmcode 3.4: Funktionen, in denen die Software des Applikationsmoduls ausgeführt wird

```
1 void* init(int argc, char *argv[]);
2 int cyclic(void *param);
3 int deinit(void *param);
```

In Programmcode 3.4 sind die Deklarationen der Funktionen für die Applikationsmodule zu sehen. Die Funktion `cyclic()` ist der Hauptteil des Moduls, welcher regelmäßig ausgeführt wird. Die zwei anderen Funktionen `init()` und `deinit()` werden jeweils nur einmal beim Start bzw. am Ende des Moduls ausgeführt. Mit den Parametern `argc` und `argv` können beim Starten des Moduls Daten vom Betriebssystem an das Modul übergeben werden. Der Rückgabewert der Funktion `init()` und der Parameter `param` von `cyclic()` und `deinit()` werden in Abschnitt 4.1 genauer erläutert und deren Notwendigkeit erklärt. Es sei hier nur so viel gesagt: Mithilfe des Rückgabewertes und der Parameter können die Adressen von dynamisch allokierten Objekten zwischen den Funktionen ausgetauscht werden.

Die Zeit, nach welcher die Funktion `cyclic()` aufgerufen werden soll, muss zum Kompilierzeitpunkt bekannt sein und die vom Modul tatsächlich verbrauchte Rechenzeit darf einen gewissen Prozentsatz der Zykluszeit nicht übersteigen, damit das Betriebssystem und andere Module nicht blockiert oder verzögert werden. Um dem Betriebssystem die Zykluszeit eines Moduls mitzuteilen, wird diese fix in das Applikationsmodul geschrieben. Dazu besitzt jedes Modul einen Descriptor, in welchem Informationen über das Modul und andere benötigte Daten gespeichert werden. Ein solcher Descriptor ist bereits im System, auf dem die Arbeit aufbaut, vorhanden und es bedarf nur ein paar Anpassungen, um ihn zu verwenden. In Abschnitt 4.4 sind die Änderungen am Descriptor näher erläutert.

3.4 Übertragung, Speicherung und Ausführung der Module

In Abbildung 3.4 ist schematisch dargestellt, dass die Applikationsmodule vom Entwicklungscomputer über das Embedded Betriebssystem auf den Mikrocontroller geladen werden. Es gibt viele Möglichkeiten, wie ein Mikrocontroller mit einem PC verbunden werden kann. Zum Beispiel UART oder SPI, aber bei diesen Schnittstellen wird ein Konverter auf USB benötigt. Aus Anwendersicht wäre es das einfachste, wenn der Mikrocontroller direkt per USB an den PC angeschlossen werden kann. Dazu wird ein USB-Stack und ein USB-Anschluss am Mikrocontroller benötigt. Der USB-Stack kümmert sich um die gesamte Abwicklung der Kommunikation inklusive Synchronisation und Time-outs. USB definiert mehrere verschiedene Protokolle, welche verwendet werden können. Die zwei bekanntesten Protokolle zur Datenübertragung sind Mass Storage Device (MSD) und Media Transfer Protocol (MTP). Ein weiteres Protokoll, das häufig eingesetzt wird, aber nicht so bekannt ist, da es meistens ohne Wissen des Anwenders verwendet wird, ist das Human Interface Device (HID). Mit diesem kommunizieren zum Beispiel Tastaturen, Mäuse und Game-Controller. MTP hat den Nachteil, dass Dateien nur übertragen und nicht bearbeitet werden können. Im Gegensatz dazu benötigt MSD ein Datei-System, wodurch aber das Bearbeiten von Dateien am Gerät möglich wird. Ein Datei-System zu integrieren bedeutet einen zusätzlichen Aufwand, welcher nicht nötig ist, da es sich bei den zu übertragenden Dateien um Binär-Dateien handelt. Binär-Dateien können nicht bearbeitet werden, daher stellt der Nachteil von MTP kein Problem dar. Aus diesem Grund wird bei diesem Projekt das MTP verwendet, um Module auf den Mikrocontroller zu laden.

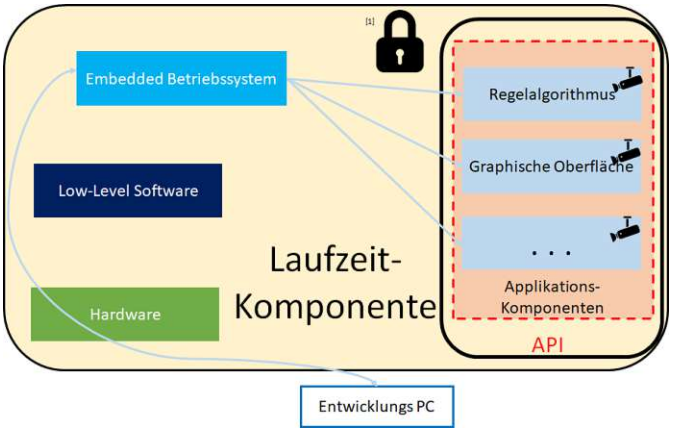

Abbildung 3.4: Übertragung, Speicherung und Ausführung der Module

Natürlich müssen diese Applikationsmodule auch auf dem Mikrocontroller in einem nicht-volatilen Programmspeicher gespeichert werden. Hierfür gibt es grundsätzlich zwei Varianten: der interne, nicht-volatile Speicher und ein externer, nicht-volatiler Speicher. Als externer, nicht-volatiler Speicher kann zum Beispiel ein externer Flash-Speicher dienen, welcher über ein Quad Serial Peripheral Interface (QSPI) angebunden ist. Wenn der interne, nicht-volatile Speicher verwendet wird, muss darauf geachtet werden, dass mehrere Speicherbänke[3] vorhanden sind, da es meist nicht möglich ist, in der-

[3]Speicher sind oft in mehrere Speicherbänke (engl. memory banks) unterteilt, um das Löschen und Schreiben zu ermöglichen, während ein Programm im selben Speicher in einer anderen Bank ausgeführt wird. Dadurch kann sich ein Mikrocon-

selben Speicherbank zu löschen und zu schreiben, in der auch das laufende Programm gespeichert ist. Wenn ein externer, nicht-volatiler Speicher verwendet wird, so muss darauf geachtet werden, ob ein Programm direkt aus dem externen Speicher exekutiert werden kann. Ist eine direkte Exekution nicht möglich oder ist die Ausführung durch die Übertragungsgeschwindigkeit zu stark beeinträchtigt, so muss der Programmcode in den RAM geladen werden und von dort aus exekutiert werden. Dies bringt zwar Vorteile in der Ausführungsgeschwindigkeit, ist aber nur für eine geringe Anzahl an kleinen Modulen möglich, da der RAM meist viel kleiner ist als der Flash-Speicher.

3.5 Debugger Unterstützung

Wie bereits erwähnt ist es wichtig, dass beim Debuggen das Betriebssystem und die restlichen Applikationsmodule nicht angehalten werden. Wenn ein normaler Debugger, wie der Segger J-Link in Verbindung mit GNU-Debugger (GDB), verwendet wird, ist dies nicht möglich, denn dieser hält mittels Hardware-Breakpoint den Prozessor an. Daher wird eine Softwarelösung, wie in Abbildung 3.5, benötigt, wodurch das Debugging Teil des Systems wird und zum Beispiel über USB ohne zusätzliche, externe Hardware gesteuert werden kann. Die einfachste Variante ist, das Programm nicht anzuhalten und nur „`printf`“-Debugging[4] zu ermöglichen. Eine weitere Möglichkeit wäre es, das Projekt „Monitor for Remote Inspection (MRI)“ von Adam Green anzupassen, welches ein Software-Debugger ist und keine zusätzliche Hardware benötigt [95]. Dieses Projekt wurde für Mikrocontroller-Programme ohne Betriebssystem entwickelt. Daher müsste die Software insofern angepasst werden, als dass der MRI die unterschiedlichen Threads erkennt und nur den Thread anhält und nur dessen Code anzeigt, in dem der Breakpoint gesetzt wurde. Außerdem muss der Scheduler des Betriebssystems angepasst werden, damit der gestoppte Thread nur dann wieder aktiviert wird, wenn beim Debuggen um einen oder mehrere Befehle weiter gegangen wird.

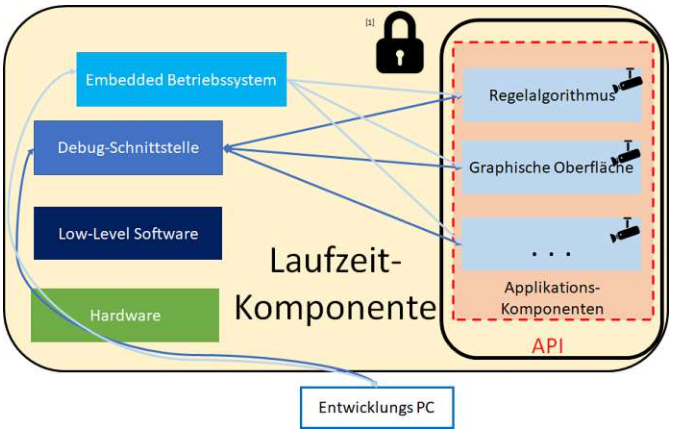

Abbildung 3.5: Softwareunterstütztes Debugging ohne externe Hardware

troller selbst programmieren.

[4] `printf()` ist die Ausgabefunktion der C-Standard-Bibliothek. Beim „`printf`“-Debugging werden Textausgaben verwendet, um den Programmablauf und die Daten im Programm für den Programmierer sichtbar zu machen, ohne dabei das Programm anzuhalten.

Kapitel 4

Implementierung

Als Basis für die Entwicklung dieser Betriebssystemerweiterung wird ein Development Kit von STM verwendet. Dieses ist in Abbildung 4.1 zu sehen und ist mit einem Arm Cortex-M4 (Abbildung 4.1a) ausgestattet (STM32L476VG: 80 MHz; 1 MB Flash; 128 kB SRAM). Als Grundlage für die Betriebssystemerweiterung dient das Embedded Betriebssystem μC/OS-III (Abbildung 4.1b). Da μC/OS-III keine Implementierung für eine USB-Schnittstelle besitzt, wird der proprietäre USB-Device-Stack (Abbildung 4.1d) von Segger (Abbildung 4.1c) zum Betriebssystem hinzugefügt. Als Entwicklungsumgebung kommt Eclipse (Abbildung 4.1e) mit einer speziellen Erweiterung für die Entwicklung von Mikrocontroller-Software zum Einsatz (Abbildung 4.1f): GNU-Microcontroller Unit (MCU)-Eclipse. Für das Kompilieren und das Linken wird die GNU-Compiler-Collection (GCC) (Abbildung 4.1g) benutzt und für das Debuggen wird der GDB (Abbildung 4.1h) verwendet.

Abbildung 4.1: Development Board von STM mit STM32L476VG. Logos der verwendeten Systeme (a [96], b [88], c [97], d [98], e [99], f [100], g [101], h [102])

Um den Code in diesem Kapitel übersichtlicher zu gestalten und die Anzahl der Zeilen zu reduzieren, wurde dieser an vielen Stellen gekürzt und vereinfacht. So wurden zum Beispiel Logging[1]-Informationen, deren Zusammenstellung und Übermittlung aus dem Code entfernt. Des Weiteren wurden die meisten Typkonvertierungen[2] aus dem Code entfernt, da diese meist nur Formalitäten für den Compiler sind und keinen Beitrag zur Verständlichkeit des Codes leisten. Beispiele hierfür sind Typkonvertierungen von unsigned[3] 32-Bit auf unsigned 16-Bit, wenn der gültige Wertebereich kleiner als 65536 ist, Typkonvertierungen von unsigned auf signed[3], wenn das höchstwertige Bit in der unsigned Variable nicht benötigt wird oder Typkonvertierungen eines unsigned 32-Bit Wertes auf einen Zeiger eines bestimmten Typs, um auf Hardware zugreifen zu können.

Damit die Module auf einem beliebigen Speicherplatz gespeichert werden können, werden diese mit der Compiler- und Linker-Option -fPIE kompiliert. Position Independent Executable (PIE) besitzt keine direkten Sprünge im Programmspeicher. Das bedeutet, dass bei allen Sprüngen eine Adressdistanz zur derzeitigen Adresse angegeben ist. Wenn ein solcher indirekter Sprung ausgeführt werden soll, wird diese Adressdistanz einfach zum „Program Counter“[4]-Register addiert.

Ein Problem, das sich aus dem beliebigen Speicherplatz eines Moduls ergibt, ist, dass auch die Speicheradressen der Variablen und virtuellen Funktionen im RAM nicht bekannt sind. Die Adressen der globalen und statischen Variablen stehen im Global Offset Table (GOT) und die Adressen der virtuellen Funktionen stehen im Virtual Table (VTable), welche beide Teil des Programmcodes sind und damit nicht veränderbar sind. Würde man nun globale und/oder statische Variablen und/oder virtuelle Funktionen benötigen, so müssten die Werte im GOT und im VTable bei der Speicherzuweisung am Mikrocontroller dynamisch geändert werden. Nähere Informationen zu diesem Thema und die benötigten Änderungen, um dies zu ermöglichen, sind in Abschnitt 6.2.2 beschrieben. Um als ersten Schritt die nötigen Änderungen gering zu halten, wurde sowohl auf globale und statische Variablen als auch auf virtuelle Funktionen verzichtet. Dadurch wird beim Kompilieren und Linken kein Speicher im RAM des Mikrocontrollers reserviert.

[1] Logging (engl. für protokollieren/Protokollierung) bezeichnet einen Mechanismus, mit dem das Fortschreiten des Programms, dessen interne Zustände und mögliche Probleme aufgezeichnet werden.

[2] Eine Typkonvertierung ist eine Anweisung an den Compiler, keine Überprüfung des Datentyps zu machen. Damit weiß der Compiler, dass eine Zuweisung von unterschiedlichen Datentypen vom Programmierer gewollt ist und der Programmierer für die Richtigkeit der Zuweisung verantwortlich ist.

[3] unsigned und signed geben an, ob ein Integerdatentyp nur positive (unsigned) oder positive und negative (signed) Werte beinhalten kann. Bei unsigned Datentypen können alle Bits zur Speicherung des Zahlenwerts verwendet werden. Bei signed Datentypen wird im höchstwertigen Bit das Vorzeichen gespeichert (1 bedeutet negatives Vorzeichen, 0 bedeutet positives Vorzeichen). Dadurch ist der Betrag der höchsten und niedrigsten Zahl nur halb so groß wie die höchste Zahl der unsigned Variante des gleichen Datentyps.

[4] Der Program Counter (engl. für Programm Zähler) ist ein Prozessorregister, welcher die Speicheradresse des derzeitig ausgeführten Befehls enthält. Ist dieser abgearbeitet, so wird der Program Counter inkrementiert.

4.1 Implementierung eines Beispiel-Moduls

In diesem Abschnitt wird die Implementierung eines Beispiel-Moduls erklärt, welches als Grundlage für die nötigen Anpassungen des Betriebssystems verwendet wird.

Aufgabe dieses Beispiel-Moduls ist es, zwei LEDs, eine rote und eine grüne LED, abwechselnd blinken zu lassen und die Tasten eines Joysticks abzufragen. In einer Variable wird mitgezählt, wie oft das Modul bereits ausgeführt wurde und wenn der Taster '0' des Joysticks betätigt wird, wird das Modul beendet. Außerdem wird beim Start des Moduls überprüft, ob das Modul einen korrekten Modul Descriptor besitzt. Falls der Descriptor nicht vorhanden oder falsch ist, wird ein Fehlerzustand angezeigt, indem beide LEDs gleichzeitig und dauerhaft leuchten.

Programmcode 4.1: Pseudo-globale Datenstruktur

```
1 struct Data {
2   module_descriptor_v02_t *moduleDescriptor;
3   LedGreen *ledGreen;
4   LedRed *ledRed;
5   Joystick *joystick;
6   uint32_t cycleCount;
7   bool toggle;
8 };
```

Wie bereits erklärt, können in den Modulen keine globalen und statischen Variablen verwendet werden. Diese lassen sich beim Programmieren aber sehr häufig nicht gänzlich vermeiden. Darum gibt es in den Modulen eine Datenstruktur, wie in Programmcode 4.1, welche alle Daten beinhaltet, welche nicht am Ende einer Funktion zerstört werden sollen. Diese pseudo-globale Datenstruktur kann sowohl Standard-Datentypen wie `int`, `char` und `bool` enthalten als auch Datenstrukturen wie `module_descriptor_v02_t` oder Instanzen von Klassen wie `LedGreen`, `LedRed` und `Joystick`. Diese Datentypen, Strukturen und Klassen können direkt als Instanzen, mit oder ohne Initialisierung, oder als Zeiger in der Struktur gespeichert werden.

Alle Module sind in drei Teile unterteilt: `init()`, `cyclic()` und `deinit()`. Die Funktionen `init()` und `deinit()` werden jeweils nur einmal am Beginn bzw. am Ende des Moduls ausgeführt. Die Funktion `cyclic()` wird regelmäßig ausgeführt, bis das Modul sich selbst beendet oder das Betriebssystem das Modul beendet. Informationen, wie das Modul beendet wird, sind in Abschnitt 4.1.2 zu finden.

In der Funktionsdefinition der Funktion `init()` in Programmcode 4.2 sind zwei Parameter definiert, welche zur Übergabe von Argumenten vom Betriebssystem an das Modul dienen. Diese Argumente werden in dem Array `argv` mit der Länge `argc` übergeben. Jedes dieser Elemente ist ein Zeiger auf einen Null-terminierten String. Als Erstes wird in der `init()`-Funktion eine Instanz der pseudo-

globalen Datenstruktur erstellt. Diese wird mit dem `new`-Operator auf dem Heap angelegt. Der `new`-Operator verwendet im Hintergrund die C-Funktion `malloc()`. Diese wiederum fordert Speicher vom Betriebssystem auf dem Heap, welcher für User-Space-Anwendungen reserviert ist, an.

Programmcode 4.2: Funktion zur Initialisierung des Moduls

```
void* init(int argc, char *argv[]) {
  Data *data = new Data;
  uint32_t descriptorAddress = 0;
  data->cycleCount = 0;
  for (uint8_t argi = 0; argi < argc; ) {
    if (strcmp(argv[argi++], "-D") == 0) {
      descriptorAddress = strtol(argv[argi], nullptr, 16);
  } }
  data->moduleDescriptor = descriptorAddress;
  if (module_verifyDescriptor(data->moduleDescriptor) != 0) {
    data->moduleDescriptor = nullptr;
  }
  data->toggle = true;
  data->ledGreen = new LedGreen { data->toggle };
  data->ledRed = new LedRed { !data->toggle };
  data->joystick = new Joystick;
  return data;
}
```

Weiters werden in Programmcode 4.2 die Daten, welche als Argumente an das Modul übergeben werden, ausgelesen. Als Beispiel dient hier die Adresse, auf der der Modul Descriptor gespeichert ist. Wenn das Element `argi` den Null-terminierten String `"-D"` enthält, ist im Element `argi + 1` die Adresse des Modul-Descriptors als ASCII-formatierter Hexadezimalwert gespeichert und es wird überprüft, ob auf dieser Adresse ein gültiger Descriptor liegt. Danach werden Instanzen für die restlichen Elemente der pseudo-globalen Datenstruktur erstellt und die `init()`-Funktion wird, mit der pseudo-globalen Datenstruktur als Rückgabewert, verlassen.

In Programmcode 4.3 ist die Funktion `cyclic()` zu sehen, welche regelmäßig ausgeführt wird. Der Übergabeparameter `param` ist der Zeiger auf die pseudo-globale Datenstruktur, welche in der Funktion `init()` erstellt wurde. Sollte der Modul-Descriptor ein Zeiger auf die Adresse Null, also ein ungültiger Zeiger, sein, so werden beide LEDs eingeschalten und das Modul mit dem Fehlercode '-1' beendet. Ansonsten werden die beiden LEDs abwechselnd ein und ausgeschalten. Der Rückgabewert der Funktion ist das Bit '0' welches vom Joystick gelesen wird. Dies führt dazu, dass das Modul beendet wird, wenn der Taster am Joystick betätigt wird, welcher auf dem Bit '0' gespeichert wird.

Programmcode 4.3: Periodisch ausgeführte Funktion des Moduls

```
int cyclic(void *param) {
  Data *data = param;
  if (data->moduleDescriptor == nullptr) {
    data->ledRed->write(true);
    data->ledGreen->write(true);
    return -1;
  }
  data->toggle = !data->toggle;
  data->ledRed->write(data->toggle);
  data->ledGreen->write(!data->toggle);
  ++cycleCount;
  return data->joystick->read() & 0b00001;
}
```

In der Funktion `deinit()` in Programmcode 4.4, welche ebenfalls den Zeiger auf die pseudo-globale Datenstruktur als Parameter bekommt, werden abschließende Aufgaben nach Beendigung des Moduls durchgeführt. Hier müssen alle Instanzen, welche in der Funktion `init()` mit dem `new`-Operator angelegt wurden, mit dem Operator `delete` gelöscht werden. Der `delete`-Operator verwendet im Hintergrund die C-Funktion `free()`. Diese wiederum fordert das Betriebssystem auf, den Speicher am Heap der User-Space-Anwendungen freizugeben.

Programmcode 4.4: Funktion zur Deinitialisierung des Moduls

```
int deinit(void *param) {
  Data *data = param;
  delete data->ledGreen;
  delete data->ledRed;
  delete data->joystick;
  delete data;
  return 0;
}
```

4.1.1 Beispiel für die Abstraktion einer Hardware

Die abstrahierte Implementierung der Hardware von `LedRed`, `LedGreen` und `Joystick` wurde bereits verwendet. Mit Programmcode 4.5 wird die abstrahierte Implementierung von Hardware anhand der roten LED erklärt. Im Konstruktor wird als Erstes ein Handle[5], welcher für alle LEDs gleich ist, mit `lomo_open()` angefordert. Der String `"/dev/leds"` ist an die Geräte-Dateien in UNIX angelehnt.

[5]Ein Handle (engl. für Griff/Henkel) ist eine abstrakte Repräsentation einer Geräte-Datei.

Programmcode 4.5: Abstrahierte Implementierung der roten LED

```
class LedRed {
public:
  explicit LedRed(bool startValue = false) : m_values{0} {
    const char *red = "Red";
    ioctlArg arg;
    m_handle = lomo_open("/dev/leds", lomo_flags_notCared, lomo_modes_notCared);
    arg.ptr = malloc(strlen(red) + 1);
    strcpy((char*)arg.ptr, red);
    m_pos = lomo_ioctl(m_handle, lomo_ioctl_getLedPos, arg);
    write(startValue);
  }
  ~LedRed() {
    lomo_close(m_handle);
  }
  void write(bool value) {
    if(read()) {
      if(value) {
        m_values |= 1U << m_pos;
      }
      else {
        m_values &= ~(1U << m_pos);
      }
      lomo_write(m_handle, &m_values, sizeof(m_values));
  } }
  bool read() {
    return lomo_read(m_handle, &m_values, sizeof(m_values)) >= 0;
  }
private: int32_t m_handle; uint32_t m_values; uint32_t m_pos;
};
```

Danach wird in Programmcode 4.5 Speicher für einen String allokiert und `"Red"` hineingeschrieben. Der Zeiger auf diesen String wird in die Variable `ioctlArg` geschrieben, welche als Übergabewert an die Funktion `lomo_ioctl()` dient. Zusätzlich werden der vorher erhaltene Handle und der auszuführende Befehl übergeben. Die Funktion `lomo_ioctl()` sendet diesen Parameter an das Betriebssystem und dieses gibt einen Positionswert für die LED `"Red"` zurück. Im Destruktor wird die Hardware freigegeben, indem `lomo_close()` mit dem Handle als Parameter aufgerufen wird. In der Funktion `write()` wird der Zustand aller LEDs dieser Geräte-Datei gelesen und nur wenn dieser korrekt gelesen werden konnte, wird der Wert der LED mit der Position `m_pos` verändert. Dieser neue Wert wird dann mit der Funktion `lomo_write()` geschrieben. Beim Lesen der Zustände der LEDs in `read()` wird zusätzlich

der Rückgabewert der Funktion `lomo_read()` überprüft, ob die Funktion korrekte Werte zurückgeliefert hat. Die Zustandswerte der LEDs werden in der Variable `m_values`, welche als Zeiger übergeben wird, gespeichert. Zeile 28 in Programmcode 4.5 enthält die Definitionen der privaten[6] Member-Variablen[7], auf welche nur Funktionen dieser Klasse zugreifen können.

4.1.2 Unterstützende Funktionen eines Moduls

Die Module benötigen auch einige Funktionen, welche standardmäßig bereitgestellt werden müssen, damit die nötigen Funktionalitäten vorhanden sind. Die Funktionen `lomo_open()`, `lomo_close()`, `lomo_read()`, `lomo_write()`, `lomo_ioctl()`, `lomo_seek()` und `lomo_sleep()` sind das API, welches von den Programmierern der Module verwendet wird. Diese Funktionen sind nur eine Zwischenschicht, um die Funktionen mit den SV-Calls zu verbergen. So ruft `lomo_open()` die Funktion `svc_open()` auf, welche die gleichen Parameter besitzt. Dies gilt auch für alle anderen Funktionen mit dem Präfix `lomo_`.

Programmcode 4.6: Implementierung der SV-Calls im Modul

```
1  #define svc(code) __asm volatile ("svc %[immediate]"::[immediate] "I" (code))
2  int32_t svc_open(const char *name, int32_t flags, int32_t mode) {
3    svc(svc_command_open);
4  }
5  int32_t svc_sync(int32_t value) {
6    __asm volatile ("mov r1, 1");
7    svc(svc_command_synch);
8    __asm volatile ("mov r1, 0");
9    svc(svc_command_synch);
10 }
```

In der ersten Zeile von Programmcode 4.6 ist ein `#define` [8], welches den Assemblerbefehl für den SV-Call im Code leichter lesbar macht. Die Funktion `svc_open` dient als Beispiel für alle anderen SV-Call-Funktionen(`svc_close()`, `svc_read()`, `svc_write()`, `svc_seek()`, `svc_ioctl()`, `svc_malloc()`, `svc_free()`, `svc_sleep()` und `svc_exit()`). Die einzigen Unterschiede dieser Funk-

[6] Die Schlüsselwörter `private` und `public` legen die Zugriffsrechte auf Funktionen und Variablen einer Klasse fest. Auf Funktionen und Variablen, welche als `public` definiert sind, kann von außerhalb der Klasse zugegriffen werden. Im Gegensatz dazu können auf Funktionen und Variablen, welche als `private` definiert sind, nur Funktionen, welche Teil der Klasse sind, zugreifen. Des Weiteren gibt es das Schlüsselwort `protected`. Bei diesem sind die Zugriffsrechte auf Funktionen und Variablen gleich wie bei `private`, mit dem Unterschied, dass andere Klassen, welche eine Spezialisierung (Vererbung) dieser Klasse darstellen, ebenfalls auf diese Funktionen und Variablen zugreifen können.

[7] Member (engl. für Mitglied). Member-Variablen sind Variablen, welche Teil einer Klasse sind. Member-Variablen werden im Code dieser Arbeit immer mit dem Präfix `m_` ergänzt.

[8] Ein `#define` (engl. für definieren) ist eine Präprozessor-Anweisung. Der Präprozessor durchsucht den Code vor dem Kompilieren nach den mit `#define` definierten Wörtern und ersetzt diese durch die definierten Werte. Zum Beispiel wird hier `svc(code)` durch `__asm volatile ("svc %[immediate]"::[immediate] "I" (code))` ersetzt, wobei `code` ein variabler Parameter ist.

tionen sind die Übergabeparameter und die definierten Konstanten, welche mit dem SV-Call an das Betriebssystem übergeben werden. Diese Konstanten bestimmen, welche Funktionalität im Betriebssystem abgearbeitet werden soll. Eine Ausnahme bildet hier die Funktion `svc_sync()`, welche zwei SV-Calls benötigt. Beim ersten SV-Call wartet das Modul auf den Beginn des nächsten Zyklus und beim zweiten SV-Call werden spezielle Aufgaben vom Betriebssystem erledigt, welche vor dem Beginn des Zyklus eines Moduls ausgeführt werden müssen. Um diese beiden SV-Calls zu unterscheiden, wird ein zusätzlicher Parameter mit übergeben. Parameter werden bei Arm Cortex-M Prozessoren in den ersten fünf Registern (`R0`, `R1`, `R2`, `R3`, und `R4`) gespeichert. Daher wird auch der neue Parameter einfach ins Register `R1` geschrieben.

Programmcode 4.7: Eigentlicher Einstiegspunkt in die Module

```
1  void entryPoint(int argc, char *argv[]) {
2    void *param = init(argc, argv);
3    int retVal = svc_sync(retVal);
4    while (retVal > 0) {
5      retVal = cyclic(param);
6      retVal = svc_sync(retVal);
7    }
8    retVal = deinit(param);
9    exit(retVal);
10 }
```

Die Funktion `entryPoint()` in Programmcode 4.7 ist der eigentliche Einstiegspunkt in die Module, welcher die drei Funktionen `init()`, `cyclic()` und `exit()` verbindet. Diese Funktion darf durch den Programmierer der Module nicht verändert werden und kann zu diesem Zweck mit den `svc_`-Funktionen in eine vorkompilierte Bibliothek verpackt werden. Hier ist auch ersichtlich, wie der Zeiger auf die pseudo-globale Datenstruktur von der `init()`-Funktion erhalten wird und als Parameter an die Funktionen `cyclic()` und `deinit()` übergeben wird. Nach der `init()`-Funktion wird das erste Mal die Funktion `svc_sync()` aufgerufen, damit zwischen der `init()`-Funktion und der Funktion `cyclic()` andere Threads ausgeführt werden können und das Modul nicht zu lange den Prozessor für sich beansprucht. Der Rückgabewert der Funktion `cyclic()` wird mit dem SV-Call an das Betriebssystem weitergeleitet und im Gegenzug ein Rückgabewert vom Betriebssystem erhalten, welcher die Beendigung des Moduls veranlassen kann. Wird das Modul beendet, so wird die `deinit()`-Funktion aufgerufen und deren Rückgabewert an die Funktion `exit()` weitergegeben. Diese Funktion beinhaltet ebenfalls einen SV-Call, welcher einen großen Unterschied zu den anderen SV-Calls besitzt: Dieser SV-Call besitzt keinen Rücksprungbefehl, weil der Thread des Moduls mit diesem SV-Call im Betriebssystem beendet wird.

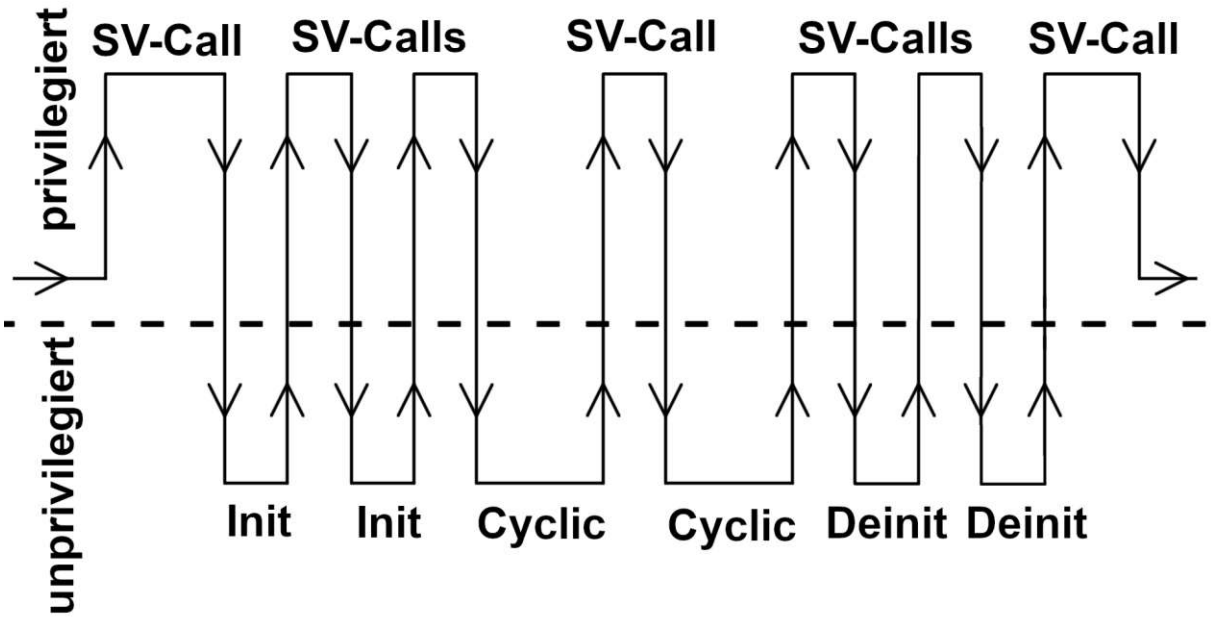

Abbildung 4.2: Zusammenspiel von User-Space (unprivilegiert) und Kernel-Space (privilegiert) mittels SV-Call

In Abbildung 4.2 sind die Sprünge zwischen einem Modul und dem Betriebssystem beispielhaft dargestellt. Der unprivilegierte Bereich ist der User-Space, in welchem die Module ausgeführt werden. Anwendungen, die im unprivilegierten Bereich ausgeführt werden, können nur auf Speicheradressen des unprivilegierten Bereichs zugreifen. Da auf die Hardware eines Mikrocontrollers ebenfalls über Speicheradressen zugegriffen wird, können die Module durch diesen Speicherschutz nicht direkt auf die Hardware zugreifen. Das Betriebssystem und die Interrupts werden im privilegierten Bereich ausgeführt, dem Kernel-Space. Anwendungen in diesem Bereich können auf jede Speicheradresse zugreifen. Nachdem das Betriebssystem Initialisierungsaufgaben für die Module durchgeführt hat, können die Module gestartet werden, indem das Betriebssystem einen SV-Call ausführt, welcher im unprivilegierten Modus in die Funktion `entryPoint()` des Moduls springt. In der Funktion `init()` werden ebenfalls SV-Calls, zum Beispiel um Speicher zu allokieren, ausgeführt. Am Ende der `init()`-Funktion wird das erste Mal ein `svc_sync()` ausgeführt, damit das Modul die Rechenzeit abgibt. Von diesem `svc_sync()` geht das Modul in die Funktion `cyclic()` und führt dort die Hauptaufgaben des Moduls aus und beendet jeden Zyklus mit einem `svc_sync()`. Wenn das Modul beendet wird, geht es nach dem letzten `svc_sync()` in die Funktion `deinit()`, wo ebenfalls wieder SV-Calls, zum Beispiel um Speicher wieder freizugeben, benutzt werden. Am Ende des Moduls wird die Funktion `exit()` aufgerufen, welche mit einem SV-Call in das Betriebssystem springt, wo der Thread des Moduls beendet wird.

Für das bereits erwähnte `printf`-Debugging benötigen die Module eine Möglichkeit, die gewünschten Daten in ein Ausgabemedium zu schreiben. Da ein Mikrocontroller meist keine Standardausgabe, wie eine Konsole auf einem Bildschirm, besitzt, wird hierfür eine Logging-Ausgabe verwendet. Diese Logging-Ausgaben können vom Betriebssystem in ein verfügbares Ausgabemedium geschrieben werden. Hierfür kann zum Beispiel eine Datei in einem externen Flash-Speicher oder ein PC dienen. Der PC kann über USB oder eine UART-Schnittstelle mit einem UART-USB-Adapter an den Mikrocontroller angeschlossen sein. Auf dem PC können die empfangenen Daten mit einer Terminal-Konsole, wie TeraTerm, angezeigt werden.

Programmcode 4.8: Logging Funktion der Module

```
int log_write(const char *buf, int nbyte) {
  char *buffer;
  int retVal = 0;
  buffer = (char*)malloc((size_t)nbyte + 1);
  strncpy(buffer, buf, (unsigned int)nbyte);
  buffer[nbyte] = '\0';
  retVal = svc_write(svc_handle_log, buffer, (size_t)nbyte);
  free(buffer);
  return retVal;
}
```

Die Module beinhalten verschiedene Funktionen, mit welchen Daten über das Logging geschrieben werden können. Mit den Funktionen `log_putchar()`, `log_puts()`, `log_vprintf()` und `log_printf()` kann die Nachricht zusätzlich verarbeitet und formatiert werden. Jedoch verwenden diese Funktionen alle die Funktion `log_write()` aus Programmcode 4.8, um die Daten zu versenden. In dieser Funktion werden die zu sendenden Daten in einen neuen Puffer umkopiert, damit sichergestellt werden kann, dass dieser Null terminiert ist. Dieser Puffer und dessen Länge werden mit einem SV-Call an das Betriebssystem gesendet. Wenn dieser SV-Call retourniert, wird dieser Puffer wieder freigegeben. Aus diesem Grund muss das Betriebssystem nochmals den Puffer umkopieren, bevor der SV-Call retourniert. Ein weiterer Vorteil von diesem Umkopieren ist, dass das Betriebssystem die Nachricht verändern oder erweitern kann. Dies ist vor allem wichtig, wenn die Logging-Nachrichten von mehreren Modulen in dasselbe Ausgabemedium geschrieben werden, da das Betriebssystem Informationen des Moduls zu der Logging-Nachricht hinzufügen kann. Eine weitere Besonderheit ist, dass kein `svc_open()` und kein `svc_close()` für den Handle der Logging-Ausgabe aufgerufen wird. Dies liegt daran, dass die Logging-Ausgabe immer geöffnet ist, wenn diese im Betriebssystem vorhanden ist, und somit auch nicht vom Modul geöffnet oder geschlossen werden muss. Dasselbe gilt für die Handles der Standard-Eingabe, der Standard-Ausgabe und der Standard-Fehlerausgabe. Diese Handles sind als

konstante Werte in den Modulen definiert und besitzen folgende Werte:

```
svc_handle_stdin = 0
svc_handle_stdout=1
svc_handle_stderr=2
svc_handle_log=3
```

Die Werte für `stdin`, `stdout` und `stderr` sind die gleichen Werte, wie sie auch im C-Standard und in UNIX definiert sind.

Damit ist der Code, welcher in einem Modul enthalten sein muss, und der Code eines Beispiel-Moduls fertig erklärt und es können nun die Änderungen und Erweiterungen im Betriebssystem behandelt werden.

4.2 Installieren eines Moduls

Wie bereits erwähnt sollen Module möglichst einfach auf dem Mikrocontroller installiert, gelöscht und upgedatet werden können. Für diese Arbeit wurde entschieden, dass auf dem Mikrocontroller ein USB-Stack von der Firma Segger mit einem MTP und einem virtuellen Dateisystem verwendet wird. Dadurch können die Binär-Dateien vom PC einfach auf den Mikrocontroller kopiert werden. Der Vorteil von MTP in Verbindung mit einem virtuellen Dateisystem ist, dass die Module mit Drag-and-Drop oder Copy-and-Paste kopiert werden können und im Gegensatz zu MSD kein vollständiges Dateisystem implementiert werden muss. Der Nachteil von MTP ist, dass die Module nicht mit Befehlen in der Kommandozeile kopiert werden können und dass die Dateien nicht direkt am Mikrocontroller verändert werden können.

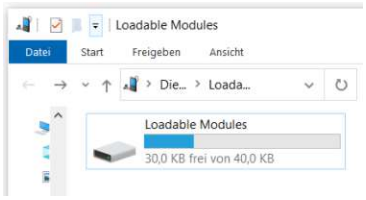

Abbildung 4.3: Darstellung des MTP-Gerätes mit Anzeige vom gesamten Speicher und freiem Speicher

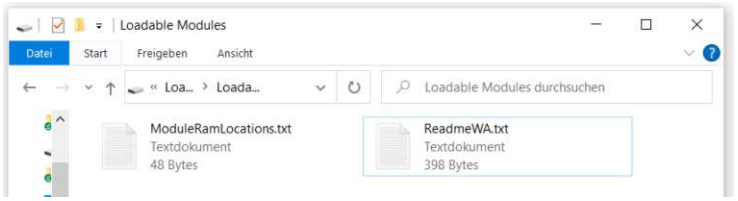

Abbildung 4.4: Inhalt des MTP-Gerätes ohne Module

In Abbildung 4.3 ist die Ansicht des Speichermediums mit einer Anzeige über den gesamten Speicher und den freien Speicher zu sehen. Wenn diese geöffnet wird, kommt man zu Abbildung 4.4. Hier sind zwei Dateien zu sehen, welche standardmäßig auf dem Mikrocontroller gespeichert und schreibgeschützt sind, also weder gelöscht noch verändert oder überschrieben werden können.

Der Inhalt der Datei `ReadmeWA.txt` ist in Abbildung 4.5 zu sehen und beinhaltet Informationen zum Kopieren von Modulen.

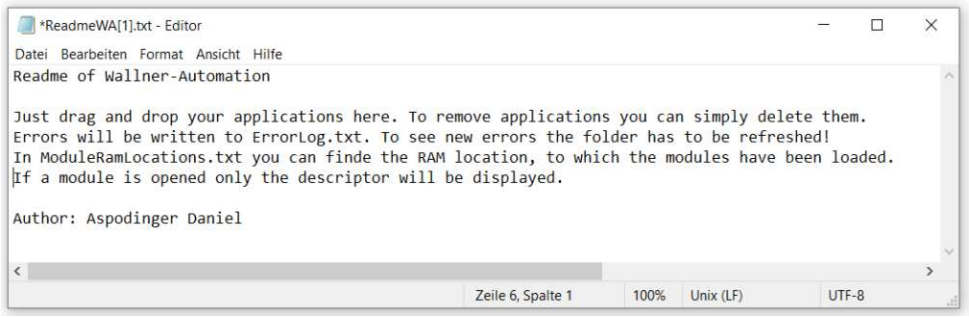

Abbildung 4.5: Readme-Datei mit Informationen zum Kopieren von Modulen

In Abbildung 4.6 ist der Inhalt des Ordners zu sehen, wenn bereits zwei Module auf den Mikrocontroller kopiert worden sind. Zusätzlich zu den beiden Modulen ist eine dritte Textdatei zu sehen. Sie wurde vom Mikrocontroller erstellt und beinhaltet Fehlerinformationen, da versucht wurde, ein ungültiges Modul auf den Mikrocontroller zu kopieren. Diese Fehlerinformationen sind in Abbildung 4.7 zu sehen. Diese Datei ist ebenfalls schreibgeschützt und kann nicht gelöscht, verändert oder überschrieben werden.

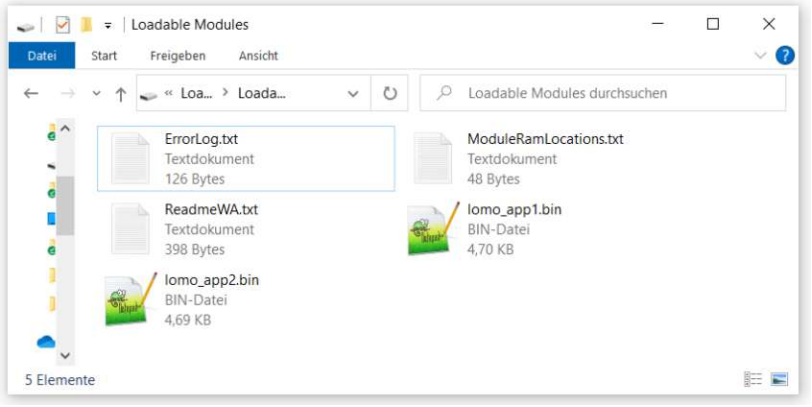

Abbildung 4.6: Inhalt des MTP-Gerätes mit zwei Modulen

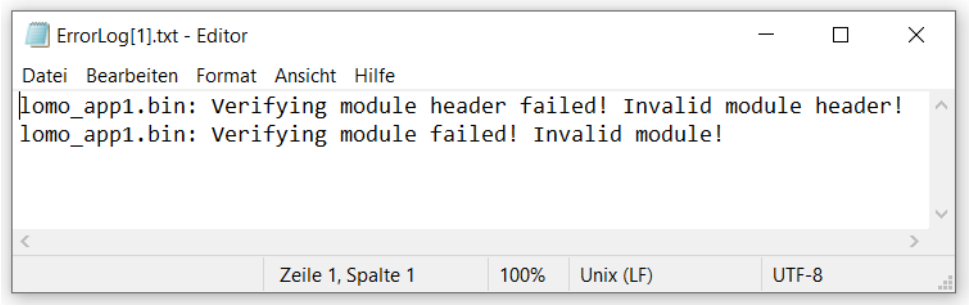

Abbildung 4.7: Informationen zu Fehlern beim Kopieren von Modulen

Außerdem wurden Informationen in der Datei `ModuleRamLocations.txt` hinzugefügt. Diese sind in Abbildung 4.8 zu sehen. In dieser Datei stehen die Speicheradressen, auf welchen die Module im RAM abgelegt sind. Diese werden benötigt, um dem Debugger mitzuteilen, auf welcher Adresse welches Modul startet.

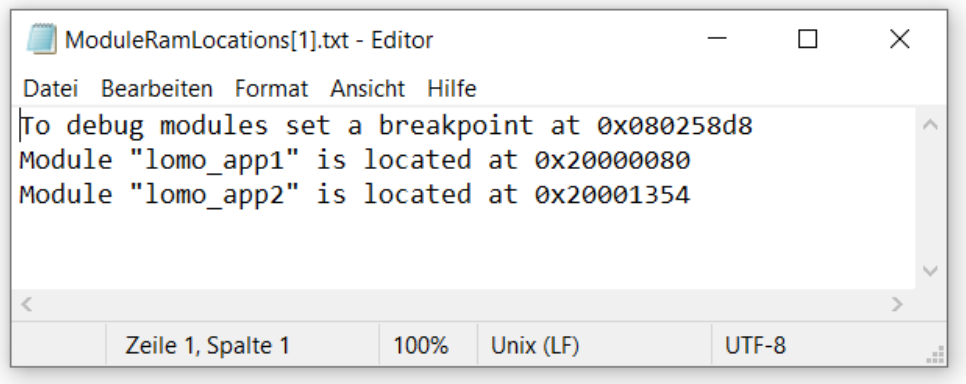

Abbildung 4.8: Informationen über die Position der Module im RAM

Als Größe der Module wird deren tatsächliche Größe auf dem Mikrocontroller verwendet. Versucht man jedoch ein Modul, welches am Mikrocontroller gespeichert ist, zu öffnen, so wird nur der Descriptor des Moduls angezeigt. Dieser enthält einige Informationen, welche als Text gespeichert sind und in einem Text-Editor dargestellt werden können. Die restlichen Informationen sind ohne Wissen über den Aufbau des Descriptors nicht entzifferbar und werden in einem Text-Editor nicht ordentlich dargestellt, da dieser versucht, die Informationen als ASCII zu interpretieren.

Abbildung 4.9: Modul-Descriptor angezeigt in einem Text-Editor

In Abbildung 4.9 sind die Informationen eines Modul-Descriptors zu sehen, wenn dieser in einem Text-Editor angezeigt wird. Der Beginn und das Ende sind zu erkennen, da diese mit vier Zeichen markiert sind: `^wa^` und `$wa$`. Weiters sind der Name des Moduls (`lomo_app1`), der Name der Compiler-Einstellung (`Debug`) und der verwendete Compiler (`GNU GCC/G++`) in Klartext im Modul gespeichert.

4.3 Behandlung von Fehlern eines Moduls

Beim Ausführen von Software können verschiedenste Fehler auftreten. So kann es zum Beispiel dazu kommen, dass probiert wird eine Division durch Null durchzuführen oder die Software einen ungültigen Assemblerbefehl enthält. Solche Fehler lösen bei Mikrocontrollern einen Hardwarefehler aus, welcher ähnlich einem Interrupt funktioniert. Die zugehörige Fehlerfunktion heißt `HardFault_Handler()`. Ein weiterer Fehler, welcher in Zusammenhang mit den Modulen auftreten kann, ist ein Zugriff auf eine Speicheradresse, auf die das Modul keine Zugriffsrechte besitzt. Hier wird ebenfalls ein Hardwarefehler ausgelöst, jedoch wird bei diesem eine andere Fehlerfunktion (`MemManage_Handler()`) ausgeführt. Es gibt noch weitere Fehlerfunktionen für Busfehler und Verwendungsfehler, welche wie der Speicherfehler aktiviert werden können. Sind diese Fehlerfunktionen nicht aktiviert, so wird an ihrer Stelle der `HardFault_Handler()` ausgeführt.

In Programmcode 4.9 ist der `HardFault_Handler()` in Assembler ausprogrammiert und es gibt eine zugehörige C-Funktion, welche aus dem Assemblercode aufgerufen wird. In Zeile drei wird das Link-Register mit der Rücksprungadresse auf dem Stack gespeichert. Danach wird das Bit '2' im Link-Register überprüft. Dieses gibt an, ob der Main-Stackpointer[9] oder der Prozess-Stackpointer[9] verwendet wurde, bevor der Fehler aufgetreten ist. Der verwendete Stackpointer wird in das Register `R0` geschrieben. Dieses Register wird ebenfalls auf den Stack gespeichert, da dieses später nochmals benötigt wird. Danach wird das Link-Register in das Register `R1` geschrieben und in die Funktion `HardFault_Handler_C()` gesprungen. Dabei dienen die Werte in den Registern `R0` und `R1` als Übergabeparameter. In dieser Funktion werden zusätzliche Informationen ausgelesen und mit der Funktion `dumpExceptionStack()` eine Fehlerausgabe durchgeführt, bei der die Registerwerte, bevor der Fehler aufgetreten ist, ausgegeben werden. Danach wird im Assemblercode der Wert des Registers `R0` vom Stack wiederhergestellt, damit in diesem Register wieder der richtige Stackpointer gespeichert ist, denn dieser wird in der Funktion `Fault_Handler_Trap_C()` in Programmcode 4.11 benötigt. In Zeile acht wird das Link-Register vom Stack wiederhergestellt und ein Sprung zu dieser Adresse durchgeführt. Im Link-Register steht jedoch keine gültige Speicheradresse, sondern ein spezieller Wert, welcher angibt, dass die derzeitige Funktion eine Interrupt-Funktion ist. Wird ein Sprung auf eine solche Adresse ausgeführt, so werden die Register vom Stack wiederhergestellt und auf die Adresse im Stack gespeicherten Link-Register gesprungen. Daher ist es wichtig, dass die Funktion `Fault_Handler_Trap_C()` entweder nicht retourniert oder den Stack ändert. Würde diese Funktion mit demselben Stackpointer retournieren, würde wieder auf die Adresse gesprungen werden, auf welcher der Fehler aufgetreten ist.

[9] Arm Cortex-M4 Prozessoren besitzen zwei getrennte Stackpointer, wobei immer nur einer in Verwendung ist. Der Main-Stackpointer wird vom Betriebssystem und von den Interrupts verwendet. Der Prozess-Stackpointer wird von den Applikationen verwendet [103].

Programmcode 4.9: Fehlerfunktion zur Behandlung von allen Fehlern

```
void HardFault_Handler(void) {
  __asm__ volatile(
      " push {lr} \n"
      " tst lr,#4 \n      ite eq \n        mrseq r0,msp \n       mrsne r0,psp \n"
      " push {r0} \n"
      " mov r1,lr \n      ldr r2,=HardFault_Handler_C \n       blx r2 \n"
      " pop {r0} \n"
      " ldr r2,=Fault_Handler_Trap_C \n    blx r2 \n"
      " pop {lr} \n"
      " bx lr \n"
      " .ltorg \n"
      : : :
  );
}
void HardFault_Handler_C(ExceptionStackFrame *frame, uint32_t lr) {
  uint32_t mmfar = SCB->MMFAR; // MemManage Fault Address
  uint32_t bfar = SCB->BFAR; // Bus Fault Address
  uint32_t cfsr = SCB->CFSR; // Configurable Fault Status Registers
  dumpExceptionStack(frame, cfsr, mmfar, bfar, lr);
}
```

Der Code für den `MemManage_Handler()` in Programmcode 4.10 ist sehr ähnlich dem Code für den `HardFault_Handler()`. Nur `Fault_Handler_Trap_C()` wird nicht im Assemblercode, sondern in der C-Funktion aufgerufen, daher muss das Register `R0` nicht auf dem Stack gespeichert werden.

Programmcode 4.10: Fehlerfunktion zur Behandlung von unzulässigen Speicherzugriffen

```
void MemManage_Handler(void) {
  __asm__ volatile(
      " push {lr} \n"
      " tst lr,#4 \n      ite eq \n      mrseq r0,msp \n       mrsne r0,psp \n"
      " mov r1,lr \n      ldr r2,=MemFault_Handler_C \n       blx r2 \n"
      " pop {lr} \n"
      " bx lr \n"
      " .ltorg \n"
      : : :
  );
}
void MemFault_Handler_C(ExceptionStackFrame *frame, uint32_t lr) {
  Fault_Handler_Trap_C(frame);
}
```

Die Funktion `Fault_Handler_Trap_C()` in Programmcode 4.11 beinhaltet den Code zum Beenden eines fehlerhaften Threads. Zuerst wird überprüft, wo der Fehler aufgetreten ist. Wenn der Fehler in einem kritischen Bereich, in dem die Interrupts deaktiviert sind, oder in einem Interrupt, erkennbar durch den speziellen Wert im Link-Register auf dem Stack, aufgetreten ist, so kann kein Thread deaktiviert werden und es wird bei einem Breakpoint angehalten und dann in eine Dauerschleife gegangen. Dies geschieht ebenfalls, wenn kein Betriebssystem ausgeführt wird oder der Fehler im `IdleTask`[10] aufgetreten ist.

Programmcode 4.11: Fehlerfunktion, welche einen fehlerhaften Thread stoppt

```
void Fault_Handler_Trap_C(ExceptionStackFrame *frame) {
  if (  CPU_IS_CRITICAL() ||
        (frame->lr & 0xFFFFFFE8u) == 0xFFFFFFE8u ||
        OSRunning != OS_STATE_OS_RUNNING ||
        OSTCBCurPtr == &OSIdleTaskTCB) {
    __DEBUG_BKPT();
    while (1) { }
  }
  else {
    OS_ERR os_err;
    OSTaskSuspend(OSTCBCurPtr, &os_err);
    OSIntCtxSw();
    __DEBUG_BKPT();
    return;
} }
```

Ist der Fehler jedoch in einem normalen Thread aufgetreten, so wird dieser mit der Funktion `OSTaskSuspend()` pausiert und der Scheduler aufgerufen. Dieser tauscht den verwendeten Stackpointer aus und stellt die Register des neuen Threads wieder her. Wenn ein Debugger angeschlossen ist, wird hier gestoppt, damit der Programmierer sofort mitbekommt, dass einer seiner Threads abgestürzt ist. Danach wird die Funktion verlassen und die jeweilige Fehlerfunktion kann ebenfalls verlassen werden, da bereits ein anderer Stackpointer gespeichert ist und somit nicht wieder auf die Adresse mit dem Fehler gesprungen wird.

[10] Der `IdleTask` ist ein Thread mit der niedrigsten Priorität, welcher immer dann ausgeführt wird, wenn kein anderer Thread ausgeführt werden muss. In diesem Thread werden auch die Daten für die verbrauchte Rechenzeit und den verbrauchten Stack der einzelnen Threads berechnet.

4.4 Anpassungen des Schedulers, Aktivierung der Memory Protection Unit und Starten eines Moduls

Die Änderungen am Scheduler sind nicht im eigentlichen Scheduler, welcher die Auswahl des als nächsten auszuführenden Threads trifft, nötig, sondern in dem Teil des Schedulers, welcher den sogenannten Context-Switch[11] durchführt. Bevor die Änderungen in der Funktion, welche den Context-Switch ausführt, erklärt werden, wird die Funktionsweise des bestehenden Context-Switch erläutert.

Der Interrupt eines Hardware-Timers, welcher mit einer Frequenz von 1 MHz zählt, wird vom Betriebssystem dynamisch aufgesetzt und kann zwischen 100 μs und fünf Minuten liegen. In diesem Interrupt wird unter anderem die Funktion `OSCtxSw()` aufgerufen, welche wiederum einen Trigger für den Pendable-Service-Call (PendSV)-Interrupt generiert. Da der PendSV-Interrupt die niedrigste Interrupt-Priorität besitzt, wird dieser erst ausgeführt, wenn alle anderen Interrupts bereits abgearbeitet sind. Damit der dieser Interrupt nicht unterbrochen wird, muss am Beginn der Funktion die Priorität des PendSV-Interrupts auf die höchste Priorität gesetzt werden. Im PendSV-Interrupt wird der Context-Switch durchgeführt.

Da bei Cortex-M Prozessoren die Hälfte der Register (xPSR, PC, LR, R12, R0, R1, R2 und R3) automatisch auf dem Stack gespeichert werden, wenn in einen Interrupt gewechselt wird, müssen diese Register bei einem Context-Switch nicht per Software gespeichert werden. Dasselbe gilt beim Verlassen eines Interrupts: Die vorher automatisch gespeicherten Register werden auch automatisch wiederhergestellt. Damit die restlichen Register (R4, R5, R6, R7, R8, R9, R10, R11, R14) auf dem Stack gespeichert werden können, wird als erstes der Stackpointer benötigt. Wenn diese Register am Stack gespeichert sind, wird der Stackpointer im Thread Control Block (TCB), einer internen Struktur im Thread, gespeichert. Danach wird der TCB des Threads mit der höchsten Priorität, welcher zur Ausführung bereit ist, benötigt, damit dessen Stackpointer ausgelesen werden kann. Von diesem Stack können nun die Register, welche per Software gespeichert wurden, wiederhergestellt werden.

Am Ende wird die Priorität des PendSV-Interrupts wieder auf die niedrigste Priorität gesetzt und der Interrupt verlassen. Dadurch werden die restlichen Register automatisch wiederhergestellt.

[11] Bei einem Context-Switch (engl. für Kontextwechsel) wird der Zustand des aktuellen Threads gespeichert und der Zustand eines neuen Threads geladen

Programmcode 4.12: Zusätzlicher Code für den Context-Switch

```
1  OS_CPU_PendSVHandler:
2    @ * Registers R0, R1, R2, R3, xPSR, PC, LR and R12 are saved on stack
3    @   automatically
4    @ * Interrupt priority is increadsed to maximum
5    @ * Registers R4, R5, R6, R7, R8, R9, R10, R11 and R14 are saved on stack
6    @   manually
7    @ * Stack pointer is saved manually in a register in TCB
8    @ * TCB of pending thread with highest priority is loaded
9      LDR      R6, [R2, #12]   @ OSTCBCurPtr->Opt
10     MRS      R7, CONTROL
11     TST      R6, #0x40      @ (OSTCBCurPtr->Opt & 0x40) ?, OS_OPT_TASK_UNPRIVILEGED
12     ITE      NE
13     ORRNE    R7, R7, #0x1    @ Set privileged bit in CONTROL (unprivileged mode)
14     BICEQ    R7, R7, #0x1    @ Reset privileged bit in CONTROL (privileged mode)
15     MSR      CONTROL, R7
16     @ * Load stack pointer from register in TCB
17     @ * Registers R4, R5, R6, R7, R8, R9, R10, R11 and R14 are loaded from stack
18     @   manually
19     @ * Interrupt priority is decreased to minimum
20     @ * Registers R0, R1, R2, R3, xPSR,PC,LR and R12 are loaded from stack
21     @   automatically
22   .end
```

In Programmcode 4.12 ist die Funktion `OS_CPU_PendSVHandler()`, welche den Context-Switch durchführt, zu sehen. Der Code, welcher bereits erklärt wurde, wurde durch Kommentare ersetzt, um die Länge des Codes, ursprünglich über 60 Zeilen, zu kürzen. Die Assemblerbefehle in Programmcode 4.12 wurden eingefügt, nachdem der Stackpointer des neuen Stacks bereits ausgelesen wurde und bevor die Register wieder hergestellt werden. Zuerst werden die Optionen des Threads aus dem TCB und das spezielle Prozessorregister CONTROL ausgelesen. Das Bit '6' in den Optionen wird überprüft, da dieses Bit angibt, ob der Thread im privilegierten oder unprivilegierten Zustand ausgeführt werden soll. Ist dieses Bit gesetzt, so wird das Bit '0' im Wert des Registers CONTROL auf 1 gesetzt und dadurch der unprivilegierte Modus aktiviert, sobald der Interrupt verlassen wird. Ist das Bit nicht gesetzt, so wird das Bit '0' im Wert des Registers CONTROL auf 0 gesetzt und der Thread wird im privilegierten Modus ausgeführt. In Zeile 15 in Programmcode 4.12 wird der veränderte Wert in das spezielle Prozessorregister zurückgeschrieben.

Damit der Unterschied zwischen privilegiertem und unprivilegiertem Modus auch einen Effekt hat, muss die MPU aktiviert werden. Dazu muss die MPU so konfiguriert werden, dass ein unprivilegierter Thread standardmäßig keinen Zugriff auf den gesamten Speicher besitzt. Im nächsten Schritt kann eine eigene Region für den Speicherbereich des Programmcodes der Module angelegt werden, auf welchen ein unprivilegierter Thread nur lesenden Zugriff besitzt. Dadurch können die Module ausgeführt werden, aber ohne dass sie ihren eigenen Programmcode verändern können. Eine weitere Region muss für den Heap und den Stack, welche von den Modulen verwendet werden, im RAM angelegt werden. Für diese müssen die unprivilegierten Threads Lese- und Schreibrechte besitzen, damit sie die Werte der Variablen am Heap verändern können und die lokalen Variablen, für die kein Platz in den Registern ist, am Stack abgelegt werden können. Weiters muss der SV-Call-Interrupt aktiviert werden, damit die Module mit dem Betriebssystem kommunizieren können.

Wie bereits erwähnt, werden die Module über USB auf den Mikrocontroller übertragen und dort in einem virtuellen Filesystem verwaltet. Damit beim Beschreiben des Flashs keine Fehler im Betriebssystem verursacht werden, werden die Module in einem externen Flash gespeichert. Es ist zwar möglich, dass die Module direkt aus dem externen Flash exekutiert werden, jedoch bringt dies starke Einschränkungen in der Ausführungsgeschwindigkeit mit sich. Das liegt an zwei Gründen: Erstens ist die Geschwindigkeit der Datenübertragung geringer (Das interne Flash arbeitet mit 80 MHz Datenrate und das externe Flash nur mit 40 MHz.). Zweitens ist das externe Flash nur mit vier Datenleitungen, im Gegensatz zu 32 Datenleitungen des internen Flashs, mit dem Mikrocontroller verbunden. Wenn man die Steuerzeichen, welche für das externe Flash benötigt werden, vernachlässigt, so wäre die Ausführung eines Moduls aus dem externen Flash um einen Faktor von 16 langsamer, als wenn es aus dem internen Flash ausgeführt werden würde.

Da die Speicherung im internen Flash nicht erwünscht ist, bleibt noch die Möglichkeit, die Module vor der Ausführung in den RAM zu kopieren und von dort auszuführen. Dazu muss das externe Flash nach den Modulen durchsucht werden. Da bei der Speicherung der Module im externen Flash jedes Modul in einen neuen Block gelegt wurde, müssen nur die Startadressen der Blöcke überprüft werden, ob dort ein gültiger Descriptor gespeichert ist. Wenn ein gültiger Descriptor gefunden wurde, wird überprüft, ob der Cyclic Redundancy Check (CRC) des zugehörigen Moduls korrekt ist. Wird ein gültiges Modul gefunden, so kann es in den RAM kopiert werden und die Startadresse des Moduls kann gespeichert werden. Diese gespeicherten Startadressen können verwendet werden, um die Module zu starten. Wie die Module gestartet werden, ist in Programmcode 4.13 zu sehen.

Programmcode 4.13: Starten der Module in einzelnen Threads

```
void startModules(uint8_t *moduleRamAddresses[]) {
  char s[k_maxModuleRamLocationTextLength];
  for (uint8_t i = 0; i < k_maxNumModules && moduleRamAddresses[i]!=nullptr; ++i) {
    module_descriptor_v02_t *moduleDescriptor = moduleRamAddresses[i];
    uint32_t entryPoint = moduleRamAddresses[i] + moduleDescriptor->entryPoint;

    lomo::ModuleWrapper *wrapper=lomo::ModuleWrapperFactory::createModuleWrapper();
    wrapper->copyToArgv("-D");
    SOPHIA_SPRINTF(s, "%08lx", moduleRamAddresses[i]);
    wrapper->copyToArgv(s);

    sophia::thread moduleTask { moduleDescriptor->moduleName };
    moduleTask.set_option(sophia::thread_option_flag::stack_check);
    moduleTask.set_option(sophia::thread_option_flag::stack_user);
    moduleTask.configure(moduleDescriptor->execPriority,
      moduleDescriptor->execStack);
    moduleTask.start(&lomo::ModuleWrapper::start, wrapper, entryPoint);
    moduleTask.detach();
  }
  lomo::ModuleSynchTimer::start();
}
```

Aus dem Descriptor der Module können der Name des Moduls und dessen Einstiegspunkt ausgelesen werden. Die Startadresse des Moduls wird als String an das Modul mit übergeben, dazu wird es in den Parameter `argv` kopiert. Dann wird ein neuer Thread, der den Namen des Moduls trägt, gestartet, welcher nicht direkt in das Modul springt, sondern zuerst in die `start()`-Funktion der Klasse `ModuleWrapper`. Dann wird der Thread von der lokalen Variable `moduleTask` mit der Funktion `detach()` getrennt, damit der Thread weiter läuft, wenn die lokale Variable zerstört wird. Wenn alle Module gestartet sind, wird ein Timer gestartet, welcher für das periodische Aktivieren der Module verantwortlich ist.

Ein weiterer wichtiger Punkt ist die Option `stack_user` des Threads. Diese bewirkt, dass der Stack des Threads in einem RAM-Bereich angelegt wird, auf den im unprivilegierten Modus zugegriffen werden kann, da sonst die Module keine Variablen auf dem Stack ablegen könnten.

4.5 Modul Wrapper

Um die Organisation der Module zu vereinfachen, wurde dafür eine eigene Klasse entwickelt. Die Klasse `ModuleWrapper` ist dafür zuständig, dass die Parameter richtig an das Modul übergeben werden und die Module zu den richtigen Zeitpunkten ausgeführt werden und deren zeitliches Verhalten überprüft wird. Außerdem ist es dafür zuständig, dass die im Modul benötigten Daten richtig allokiert und dass beim Beenden eines Moduls jeglicher, allokierter Speicher freigegeben wird.

Programmcode 4.14: Destruktor von der Klasse `ModuleWrapper`

```
lomo::ModuleWrapper::~ModuleWrapper() {
  OS_ERR os_err;
  for (int i = 0; i < m_argc; ++i) {
    mem2_free(m_argv[i]);
  }
  mem2_free(m_argv);
  ...
  OSTaskDel(0, &os_err);
}
```

Im Konstruktor werden nur interne Datentypen angelegt, daher wurde dieser hier weggelassen. Im Destruktor werden jedoch einige wichtige Tätigkeiten durchgeführt. Wie in Programmcode 4.14 zu sehen, wird hier der Speicher der Elemente in `m_argv` und `m_argv` selbst freigegeben, da diese in Programmcode 4.15 dynamisch allokiert worden sind. Außerdem wird das Modul, für das der Destruktor aufgerufen wurde, aus der Liste der laufenden Module entfernt.

Programmcode 4.15: Funktion zum Umkopieren von Elementen, welche an das Modul übergeben werden sollen

```
int lomo::ModuleWrapper::copyToArgv(const char *element) {
  ++m_argc;
  if (m_argv == nullptr) {
    m_argv = mem2_malloc(m_argc * sizeof(char*));
  }
  else {
    m_argv = mem2_realloc(m_argv, m_argc * sizeof(char*));
  }
  m_argv[m_argc - 1] = mem2_malloc((strlen(element) + 1) * sizeof(char));
  strcpy(m_argv[m_argc - 1], element);
  return 0;
}
```

Mit der Funktion `copyToArgv()` in Programmcode 4.15 kann ein Element zu `m_argv` hinzugefügt werden und diese Elemente werden dann beim Start des Moduls übergeben. Diese Systematik mit `argv` und `argc` kommt von den Kommandozeilen-Argumenten, welche beim Start eines PC-Programms aus der Kommandozeile mit übergeben werden können. Die Variable `argv` ist ein Array, welches Zeiger auf Null-terminierte Strings beinhaltet. Die Variable `argc` gibt an, wie viele Elemente in der Variable `argv` enthalten sind. Wenn ein neues Element hinzugefügt wird, muss zuerst `argc` erhöht werden und Speicher für `argv` allokiert werden, falls es das erste Element ist, oder der Speicher von `argv` vergrößert werden. Danach wird an die neue Position in `argv` die Adresse des Speicherbereichs, welcher für das Element neu allokiert wird, geschrieben und das Element wird in diesen Speicherbereich kopiert. Dieses Umkopieren ist nötig, damit die Funktion, welche das Element erstellt hat, den Speicherbereich wieder freigeben kann bzw. eine lokale Variable verwendet werden kann. Würde man dies nicht machen, wäre es sehr schwierig für die aufrufende Funktion zu wissen, wann das Element nicht mehr benötigt wird und gelöscht werden kann.

Programmcode 4.16: Funktion zum Beenden eines Moduls

```
1 class ModuleWrapperFactory {
2 public:
3   static ModuleWrapper* createModuleWrapper(){ return new ModuleWrapper { }; }
4 };
5 void lomo::ModuleWrapper::stopHook() {
6   OS_ERR os_err;
7   ModuleWrapper *tmp = OSTaskRegGet(nullptr, s_taskRegIDModuleWrapper, &os_err);
8   delete tmp;
9 }
```

In Programmcode 4.16 ist die Klasse `ModuleWrapperFactory` und die statische Funktion[12] `stopHook()` der Klasse `ModuleWrapper` zu sehen. Soll ein Modul beendet werden, so kann in dem Thread, in dem das Modul ausgeführt wird, die Funktion `stopHook()` aufgerufen werden. Diese kann den Zeiger auf die Instanz der Klasse `ModuleWrapper`, welche zu dem Modul gehört, aus einem Register im TCB lesen. Dieser Zeiger kann nun mit dem `delete`-Operator freigegeben werden. Damit sichergestellt wird, dass die Instanz der Klasse `ModuleWrapper` mit dem `new`-Operator angelegt wurde, wird für die Klasse `ModuleWrapper` das Factory[13]-Pattern[14] verwendet.

[12] Statische Funktionen einer Klasse können nicht auf Member-Variablen der Klasse zugreifen, da diese Funktion der Klasse und nicht einer bestimmten Instanz einer Klasse zugeordnet ist.

[13] Beim Factory-Pattern wird eine Klasse nicht durch den direkten Aufruf des Konstruktors erstellt, sondern indirekt durch den Aufruf einer Factory-Funktion, welche dann den korrekten Konstruktor der richtigen Klasse mit den richtigen Parametern aufruft.

[14] Patterns (engl. für Muster) sind Softwarekonstrukte, welche eine definierte Lösung für spezielle Probleme in der Softwareentwicklung bieten.

Die Funktion `start()` in Programmcode 4.17 ist der Einstiegspunkt für den Thread, in welchem das Modul ausgeführt werden soll. Es verwendet den im Descriptor gespeicherten Wert für den Zyklus des Moduls, um es in die Liste der laufenden Module einzusortieren. Außerdem wird der im Descriptor gespeicherte Wert für die maximale Rechenzeit pro Zyklus in der Variable `m_cycleTimeout` der Klasse `ModuleWrapper` gespeichert. Damit diese Werte im Descriptor gespeichert werden können, mussten diese hinzugefügt werden. Außerdem wurde eine zusätzliche Option hinzugefügt, damit angegeben werden kann, dass es sich um ein Modul handelt, welches auf einer beliebigen Speicheradresse abgelegt werden kann.

Programmcode 4.17: Funktion zum Starten eines Moduls

```
void lomo::ModuleWrapper::start(uint32_t entryPoint) {
  OS_ERR os_err;
  m_tcb = OSTCBCurPtr;
  ...
  if (moduleDescriptor->execCycle > 0) {
    m_cycleTimeout = moduleDescriptor->execCycleTimeout;
    for (uint8_t i = 0; i < k_counterBits; ++i) {
      if (moduleDescriptor->execCycle <= 1U << i) {
        m_cycleExponent = i;
        m_listItem.next = s_moduleWrapperList[i];
        s_moduleWrapperList[i] = &m_listItem;
        break;
  } } }
  OSTaskRegSet(nullptr, s_taskRegIDModuleWrapper, this, &os_err);
  svc_enterUnprivileged(m_argc, m_argv, entryPoint);
}
void svc_enterUnprivileged(int argc, char *argv[], uint32_t entryPoint) {
  __asm volatile ("svc #0xFF"); //svc_command_reserved
}
```

Am Ende der Funktion `start()` in Programmcode 4.17 wird der Zeiger auf die Instanz der Klasse `ModuleWrapper` in einem Register des TCB gespeichert und `svc_enterUnprivileged()` aufgerufen. Diese Funktion löst einen SV-Call-Interrupt aus, um in den unprivilegierten Modus und in das Modul zu wechseln. Der Aufruf des Assemblerbefehls kann nicht direkt in der Funktion `start()` aufgerufen werden, da durch diesen zusätzlichen Funktionsaufruf die Parameter in die Register `R0`, `R1` und `R2` geschrieben werden, welche bei einem Interrupt automatisch auf dem Stack abgelegt werden. Dadurch kann im SV-Call-Interrupt auf den Wert von `entryPoint` zugegriffen werden, und die anderen beiden Werte werden beim Verlassen des Interrupts automatisch wieder in die Register geschrieben und können in der Funktion `entryPoint()` im Modul verwendet werden.

Programmcode 4.18: Funktion, welche aufgerufen wird, wenn das Modul die Rechenzeit abgibt

```
int lomo::ModuleWrapper::synch(int param) {
  OS_ERR os_err;
  ModuleWrapper *tmp = OSTaskRegGet(nullptr, s_taskRegIDModuleWrapper, &os_err);
  tmp->m_isInitFinished = true;
  tmp->m_isCycleRunning = false;
  OS_FlagPendTry(&s_flagGroup, 1 << tmp->m_flagPos, 0,
    OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_FLAG_CONSUME + OS_OPT_PEND_BLOCKING,
    (CPU_TS *) 0, &os_err);
  return param;
}
```

Die Funktion `synch()` in Programmcode 4.18 wird aufgerufen, wenn ein Modul seine Rechenzeit abgibt. Da es sich um eine statische Funktion handelt, muss der Zeiger auf die Instanz der Klasse `ModuleWrapper` aus dem Register im TCB ausgelesen werden. Die Variable `m_isInitFinished` wird auf `true` gesetzt, da diese Funktion auch am Ende der Initialisierung aufgerufen wird. Die Variable `m_isCycleRunning` wird auf `false` gesetzt, da die Abarbeitung der Funktion `cyclic` im Modul mit diesem Aufruf abgeschlossen ist. Diese wird verwendet, um die benötigte Rechenzeit zu ermitteln. Am Ende wird der Thread in den Wartezustand versetzt, indem mit der Funktion `OS_FlagPendComplete` auf ein Flag in einer Flag-Gruppe gewartet wird. Da diese Funktion in einem Interrupt aufgerufen wird, wird nicht direkt hier blockiert, sondern erst, wenn der Interrupt verlassen wird. Der Parameter `param`, welcher vom Modul kommt, wird hier nur durchgeschliffen, da dieser Parameter erst in der Funktion `synchComplete` in Programmcode 4.19 benötigt wird.

Programmcode 4.19: Funktion, welche aufgerufen wird, bevor das Modul einen neuen Zyklus beginnt

```
int lomo::ModuleWrapper::synchComplete(int param) {
  OS_ERR os_err;
  ModuleWrapper *tmp = OSTaskRegGet(nullptr, s_taskRegIDModuleWrapper, &os_err);
  if (param > 0) {
    tmp->m_isCycleRunning = true;
  }
  OS_FlagPendComplete(&s_flagGroup, 1 << tmp->m_flagPos, 0,
    OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_FLAG_CONSUME + OS_OPT_PEND_BLOCKING,
    (CPU_TS *) 0, &os_err);
  return param;
}
```

In der statischen Funktion `synchComplete()` in Programmcode 4.19 wird ebenfalls die Instanz der Klasse `ModuleWrapper` aus dem Register im TCB ausgelesen. Ist der Parameter `param` kleiner oder

gleich Null, so wird das Modul beendet und die Variable `m_isCycleRunning` nicht mehr auf `true` gesetzt. Der Thread wurde durch das aktiv Werden des Flags in der Flag-Gruppe aktiviert und das Betriebssystem muss jetzt noch einige Aufgaben in der Funktion `OS_FlagPendComplete()` durchführen. Am Ende wird der Parameter `param` zurückgegeben, damit dieser an das Modul weitergeleitet wird und das Modul die Funktion `deinit()` ausführen kann.

Bei der Funktion `synchTimer()` in Programmcode 4.20 handelt es sich ebenfalls, um eine statische Funktion der Klasse `ModuleWrapper`. Diese Funktion ist verantwortlich für das Aktivieren der Module, wenn deren Zyklus kommt, und das Überprüfen der benötigten Rechenzeit. Eine Einschränkung ist, dass die Module nur eine Zeitbasis besitzen können, welche eine Multiplikation der Zykluszeit des Timers mit einer Zweierpotenz ist.[15] Am Anfang wird die Funktion immer wieder verlassen und nichts gemacht, bis alle Module ihre Initialisierungsfunktion ausgeführt haben, damit alle Module synchron laufen. Der Code hierfür wurde aus der Funktion `synchTimer` entfernt und mit `...` ersetzt, um den Code kürzer und übersichtlicher zu gestalten. Außerdem wurden die Variablennamen `m_moduleWrapper` zu `m_modWrap` und `s_moduleWrapperList` zu `s_modWrapList` gekürzt, um automatische Zeilenumbrüche im Code so weit wie möglich zu vermeiden.

Von Zeile sechs bis neun in Programmcode 4.20 wird der letzte Wert der Variable `counter` gespeichert, dekrementiert und bei einem Überlauf der Maximalwert (2047 = 0b11111111111) in die Variable geschrieben. Da die Funktion `synchTimer` durch einen Timer alle 500 μs aufgerufen wird, können Module mit Zykluszeiten von 500 μs bis 1,024 s ausgeführt werden. Diese Werte können in beide Richtungen noch erweitert werden, sollten aber für fast alle Module ausreichend kleine bzw. große Zykluszeiten bieten. In der äußeren `for`-Schleife und den äußeren `if`-Bedingungen wird mithilfe der Variable `mask` überprüft, welche Bits sich in der Variable `counter` gegenüber der Variable `lastCounter` geändert haben.

Programmcode 4.20: Timer-Funktion zum Aktivieren der Module und zur Überprüfung der Rechenzeit

```
1  void lomo::ModuleWrapper::synchTimer() {
2    static uint16_t counter = k_counterRefillValue;
3    uint16_t lastCounter;
4    OS_ERR os_err;
5    ...
6    lastCounter = counter--;
7    if (counter > k_counterRefillValue) {
8      counter = k_counterRefillValue;
9    }
10
```

[15] Einfache Zykluszeit des Timers, zweifache Zykluszeit des Timers, vierfache Zykluszeit des Timers, achtfache Zykluszeit des Timers, usw.

```
11  for (uint8_t i = 0; i < k_counterBits; ++i) {
12    uint32_t mask = 1 << i;
13    if ((counter & mask) == mask && (~lastCounter & mask) == mask) {
14      for (ListItem *item = s_modWrapList[i]; item != nullptr; item = item->next) {
15        if ((item->m_modWrap->m_tcb->TaskState & TaskSuspended ) != TaskSuspended) {
16          if (item->m_modWrap->m_skipNextCycle) {
17            if (item->m_modWrap->m_isCycleRunning) {
18              item->m_modWrap->m_execTime += 1 << item->m_modWrap->m_cycleExponent;
19              if (item->m_modWrap->m_execTime > item->m_modWrap->m_cycleTimeout) {
20                OSTaskSuspend(item->m_modWrap->m_tcb, &os_err);
21          } } }
22          else {
23            item->m_modWrap->m_execTime = 0;
24            OSFlagPost(&s_flagGroup, 1 << item->m_modWrap->m_flagPos,
25              OS_OPT_POST_FLAG_SET, &os_err);
26    } } } }
27
28    if ((~counter & mask) == mask && (lastCounter & mask) == mask) {
29      for (ListItem *item = s_modWrapList[i]; item != nullptr; item = item->next) {
30        if ((item->m_modWrap->m_tcb->TaskState & TaskSuspended ) != TaskSuspended) {
31          if (item->m_modWrap->m_isCycleRunning) {
32            item->m_modWrap->m_skipNextCycle = true;
33          }
34          else {
35            item->m_modWrap->m_skipNextCycle = false;
36  } } } } } }
```

In der `if`-Abfrage von Zeile 13 bis 26 werden die Module aktiviert, die zu der Zykluszeit gehören, bei welcher sich das Bit von 0 auf 1 geändert hat. Dafür werden in der `for`-Schleife alle Module der zugehörigen Zykluszeit durchlaufen und überprüft, ob der Thread des jeweiligen Moduls pausiert ist.[16] Wenn die Variable `m_skipNextCycle` `false` ist, so wird der Wert der benötigten Rechenzeit in der Variable `m_execTime` zurückgesetzt und das entsprechende Flag in der Flag-Gruppe gesetzt, damit der Thread des Moduls aktiviert wird. Ist die Variable `m_skipNextCycle` jedoch `true`, so wird die Ausführung des Moduls in diesem Durchgang übersprungen. Ist zusätzlich die Variable `m_isCycleRunning` auf `true` gesetzt, das Modul benötigt also noch immer Rechenzeit für diesen Zyklus, wird dessen Zykluszeit verdoppelt. Ist die neue Zykluszeit größer als die maximale Zykluszeit des Moduls, so wird das Modul pausiert.[16]

[16] Pausierte Threads sollen hier nicht wieder ausgeführt werden. Es sollen nur jene Threads aktiviert werden, die auf ein Flag in der Flag-Gruppe warten. Pausierter Thread $\neq$ Wartender Thread

In der `if`-Abfrage von Zeile 28 bis 36 wird nach der halben Zykluszeit, also wenn sich ein Bit von 1 auf 0 geändert hat, überprüft, ob die Module noch ausgeführt werden. Dafür werden in der `for`-Schleife alle Module der zugehörigen Zykluszeit durchlaufen und überprüft, ob der Thread des jeweiligen Moduls pausiert ist. [16] Wenn die Variable `m_isCycleRunning` `true` ist, also das Modul noch immer Rechenzeit für diesen Zyklus benötigt, so wird die Variable `m_skipNextCycle` auf `true` gesetzt, damit das Modul den nächsten Zyklus überspringt und nicht ausgeführt wird. Ansonsten wird die Variable `m_skipNextCycle` auf `false` gesetzt, da das Modul rechtzeitig fertig geworden ist.

Durch dieses Auslassen von Zyklen und dem Erhöhen der Zykluszeit von Modulen kann die Ausführungshäufigkeit von niederprioren Modulen und somit die Auslastung des Prozessors automatisch reduziert werden. Damit kann erreicht werden, dass auch die Zyklen von niederprioren Modulen fertig ausgeführt werden. Natürlich muss darauf geachtet werden, dass zeitkritische Module eine Priorität besitzen, welche hoch genug ist, dass ihr Zyklus nicht verlängert wird.

4.6 Abarbeitung von Supervisor-Calls

Für die Kommunikation mit dem Betriebssystem verwenden die Module SV-Calls. Wird ein solcher SV-Call aufgerufen, so wird ein SV-Call-Interrupt ausgelöst und der Prozessor wechselt in den privilegierten Modus. Beim Eintritt in die Interrupt-Funktion werden automatisch die Hälfte der Register (`xPSR`, `PC`, `LR`, `R12`, `R0`, `R1`, `R2` und `R3`) auf dem Stack abgelegt, damit diese Register in der Interrupt-Funktion verwendet werden können, ohne diese extra auf dem Stack ablegen zu müssen.

Programmcode 4.21: Interrupt-Funktion des SV-Calls

```
1 void SVC_Handler(void) {
2   __asm(
3       " TST lr, #4 \n     ITE EQ \n     MRSEQ r0, MSP \n     MRSNE r0, PSP \n"
4       " B SVC_Handler_C\n" );
5 }
```

In Programmcode 4.21 ist die Interrupt-Funktion des SV-Calls zu sehen. Wie bei den anderen, bereits besprochenen Interrupt-Funktionen, wird zuerst überprüft, ob der Main-Stackpointer oder der Process-Stackpointer verwendet wurde und der verwendete Stackpointer wird in das Register `R0` kopiert. Das Register `R0` wird bei Funktionsaufrufen verwendet, um den ersten Parameter zu übergeben. Darum kann in der Funktion `SVC_Handler_C()` in Programmcode 4.22, welche von der Interrupt-Funktion aufgerufen wird, der Stackpointer aus dem Übergabeparameter verwendet werden und es muss nicht direkt auf das Register `R0` zugegriffen werden.

In der Funktion SVC_Handler_C() in Programmcode 4.22 werden die am Stack abgelegten Register R0, R1, R2, LR, PC und die Variable command, welche für die Unterscheidung der SV-Call-Funktionen verantwortlich ist, ausgelesen. Das Auslesen der Register vom Stack ist nicht kompliziert, es muss nur bekannt sein, in welcher Reihenfolge die Register auf dem Stack abgelegt wurden. Das Auslesen des Kommandos, welches beim SV-Call angegeben wurde, ist etwas trickreicher, da dieses Kommando Teil des Assemblerbefehls ist. Daher wird der Program Counter (PC) benötigt. Dieser wurde nach Abarbeitung des Assemblerbefehls bereits inkrementiert. Der Maschinencode für den SV-Call besteht aus zwei Byte, im Byte auf der höheren Speicheradresse ist der Maschinencode für das Auslösen des SV-Call-Interrupts und im Byte auf der niedrigeren Speicheradresse ist das Kommando gespeichert. Aus diesem Grund wird der Zeiger des Program Counters auf einen Zeiger des Typs uint8_t, welcher ein Datentyp mit der Größe von einem Byte ist, konvertiert. Mit diesem Zeiger kann auf den Speicher zugegriffen werden, als ob es sich um ein Array handelt. Da in der Programmiersprache C++ keine automatische Überprüfung der Arraygrenzen gemacht wird, kann auf das Kommando, welches um zwei Byte weiter unten im Speicher liegt, mit dem Array-Index -2 zugegriffen werden. Diese Information und die Information über die Reihenfolge der Register am Stack stammen aus „The definitive guide to Arm Cortex-M3 and Cortex-M4 Processors“ [103] von Joseph Yiu.

In der Switch-Case-Anweisung werden die Funktionen zur Abarbeitung der einzelnen Kommandos aufgerufen. Diesen Funktionen werden die Parameter übergeben, welche beim Aufrufen einer svc_ -Funktion im Modul als Parameter verwendet wurden. Der Rückgabewert einer Funktion wird im Register R0 gespeichert und, da nicht die Interrupt-Funktion, sondern die svc_-Funktion den Rückgabewert liefern soll, wird der Rückgabewert in das am Stack gespeicherte Register R0 geschrieben. Das am Stack gespeicherte Register R0 wird beim Verlassen der Interrupt-Funktion und der Rückkehr in das Modul wiederhergestellt. Dadurch besitzen die svc_-Funktionen einen Rückgabewert, welcher, wie bei jeder anderen Funktion auch, verwendet werden kann.

Mithilfe des Kommandos svc_command_synch können zwei verschiedene Funktionen aus der Klasse ModuleWrapper aufgerufen werden: synch() und synchComplete(). Welche dieser Funktionen aufgerufen wird, wird durch das im Stack gespeicherte Register R1 unterschieden. Wie bereits bei der Erklärung der beiden Funktionen erwähnt, wird diese Zweiteilung benötigt, da vor Beginn eines Zyklus und am Ende eines Zyklus Überprüfungen durchgeführt werden müssen.

Programmcode 4.22: C-Funktion zur Abarbeitung der SV-Calls

```
1 void SVC_Handler_C(uint32_t *svc_args) {
2   int32_t *R0 = svc_args;
3   int32_t *R1 = svc_args + 1;
4   int32_t *R2 = svc_args + 2;
```

```
5  uint32_t *LR = svc_args + 5;
6  uint32_t *PC = svc_args + 6;
7  uint8_t command = reinterpret_cast<uint8_t*>(PC)[-2];
8  OS_ERR os_err;
9
10 switch (command) {
11 case svc_command_open:
12   *R0 = openHandler(*R0, *R1, *R2);
13   break;
14 case svc_command_close:
15   *R0 = closeHandler(*R0);
16   break;
17 case svc_command_read:
18   *R0 = readHandler(*R0, *R1, *R2);
19   break;
20 case svc_command_write:
21   *R0 = writeHandler(*R0, *R1, *R2);
22   break;
23 case svc_command_seek:
24   *R0 = seekHandler(*R0, *R1, *R2);
25   break;
26 case svc_command_ioctl:
27   *R0 = ioctlHandler(*R0, *R1, *R2);
28   break;
29 case svc_command_synch:
30   if (*R1) {
31     *R0 = lomo::ModuleWrapper::synch(*R0);
32   }
33   else {
34     *R0 = lomo::ModuleWrapper::synchComplete(*R0);
35   }
36   break;
37 case svc_command_malloc:
38   *R0 = mem2_malloc(*R0);
39   break;
40 case svc_command_free:
41   mem2_free(*R0);
42   break;
43 case svc_command_sleep:
44   OSTimeDly((OS_TICK) *R0, OS_OPT_TIME_DLY, &os_err);
45   *R0 = (os_err == OS_ERR_NONE ? 0 : -1);
46   break;
```

```
47   case 0xFF: //svc_command_reserved
48     *PC = *R2; // R0: argc; R1: argv; R2: entryPoint;
49     *LR = *R2;
50     OSTCBCurPtr->Opt |= OS_OPT_TASK_UNPRIVILEGED;
51     OSIntCtxSw();
52     break;
53   case svc_command_exit:
54     svc_exitHandler(LR, PC);
55     break;
56   default: // Unknown SVC request
57     *R0 = -1;
58     assert(false && "Undefined SVC request!");
59     break;
60   }
61   OS_TASK_SW_SYNC();
62 } // value in R0 is the return value
```

Ein spezielles Kommando hat den Wert 0xFF = 255 und besitzt, wie die anderen Kommandos auch, eine benannte Konstante. Diese heißt `svc_command_reserved` und ist nur für die Verwendung beim Starten eines Moduls in der Klasse `ModuleWrapper` vorgesehen. Bei diesem SV-Call werden drei Parameter übergeben: `argc` in `R0`, `argv` in `R1` und `entryPoint` in `R2`. Da das Verlassen einer Interrupt-Funktion durch einen Sprung auf das Link Register oder dem Wiederherstellen des Program Counters vom Stack ausgeführt werden kann, müssen beide Register mit der Adresse des Einstiegspunktes des Moduls beschrieben werden. Außerdem wird im TCB das Options-Bit für den unprivilegierten Modus gesetzt und ein Context-Switch durchgeführt, damit in den unprivilegierten Zustand gewechselt wird.

Programmcode 4.23: Funktion zum Öffnen einer Geräte-Datei

```
1  int32_t svc::openHandler(const char *name, int32_t flags, int32_t mode) {
2    void *handle = nullptr;
3    ValidHandles::DeviceType deviceType = ValidHandles::DeviceType::notValid;
4    if (strcmp(name, "/dev/leds") == 0) {
5      LedsHandler *tmp = new LedsHandler();
6      deviceType = ValidHandles::DeviceType::leds;
7      handle = tmp;
8    }
9    else if (strcmp(name, "/dev/joystick") == 0) {
10     JoystickHandler *tmp = new JoystickHandler();
11     deviceType = ValidHandles::DeviceType::joystick;
12     handle = tmp;
13   }
```

```
  else {
    return svc_errorCode_invalidOperation;
  }
  ValidHandles::add(handle, deviceType);
  return handle;
}
```

Die Funktion `openHandler()` in Programmcode 4.23 ist für das Öffnen von bzw. das Erstellen eines Handles für Geräte-Dateien zuständig. Je nachdem welche Geräte-Datei mit dem Null-terminierten String `name` angefordert wird, wird ein anderes Objekt erstellt. Diese Objekte werden mit der statischen Funktion `add()` der Klasse `ValidHandles` gespeichert. Sollte hier kein Platz mehr vorhanden sein, so wird das Objekt mit dem `delete`-Operator wieder freigegeben und ein Fehler retourniert. Diese Überprüfung und Unterscheidung der verschiedenen Objekte wurde hier wieder entfernt, um den Code übersichtlicher zu gestalten.

In der Funktion `closeHandler()` werden die Objekte mit dem `delete`-Operator freigegeben und aus der Klasse `ValidHandles` entfernt, wenn diese vom Modul geschlossen werden. Dafür müssen die Handles auf den Typ der richtigen Klasse konvertiert werden, damit der richtige Destruktor aufgerufen wird. Dazu wird in `openHandler()` auch der Typ des Handles in der Klasse `ValidHandles` gespeichert. Die Handles von `stdin`, `stdout`, `stderr` und `log` können nicht geschlossen werden, da diese immer geöffnet bleiben, weil sich alle Module diese Handles teilen.

Programmcode 4.24: Funktion zum Lesen von Geräte-Dateien

```
int32_t svc::readHandler(int32_t handle, void *buffer, size_t bytes) {
  ValidHandles::DeviceType deviceType;
  ...
  else if (ValidHandles::check(handle, deviceType)) {
    switch (deviceType) {
    case ValidHandles::DeviceType::leds:
      if (bytes != 4) { return svc_errorCode_invalidParameter; }
      *buffer = reinterpret_cast<LedsHandler*>(handle)->read();
      return bytes;
    case ValidHandles::DeviceType::joystick:
      if (bytes != 4) { return svc_errorCode_invalidParameter; }
      *buffer = reinterpret_cast<JoystickHandler*>(handle)->read();
      return bytes;
    default: return svc_errorCode_invalidHandle;
  } }
  return svc_errorCode_invalidHandle;
}
```

Die Überprüfungen für den `stdin`-, `stdout`-, `stderr`- und `log`-Handle wurden in `readHandler()` in Programmcode 4.24 durch `...` ersetzt, da der Standard-Input noch nicht implementiert ist und von den anderen drei Handles nicht gelesen werden kann. Wenn ein Handle übergeben wird, um von den LEDs oder dem Joystick zu lesen, wird überprüft, ob die Größe des Puffers korrekt ist. Danach wird der Handle auf die entsprechende Klasse konvertiert und die Daten von der Hardware ausgelesen.

Programmcode 4.25: Funktion zum Schreiben von Geräte-Dateien

```
int32_t svc::writeHandler(int32_t handle, const void *buffer, size_t bytes) {
  ValidHandles::DeviceType deviceType;
  char *tmp;
  ...
  else if (handle == svc_handle_log) {
    tmp = new char[bytes];
    strcpy(tmp, buffer);
    cdcLoggingQueue.put(tmp, bytes);
    return bytes;
  }
  else if (ValidHandles::check(handle, deviceType)) {
    switch (deviceType) {
    case ValidHandles::DeviceType::leds:
      if (bytes != 4) {
        return svc_errorCode_invalidParameter;
      }
      reinterpret_cast<LedsHandler*>(handle)->write(*buffer);
      return bytes;
    case ValidHandles::DeviceType::joystick: return svc_errorCode_invalidOperation;
    default:
      return svc_errorCode_invalidHandle;
    }
    return svc_errorCode_noError;
  }
  return svc_errorCode_invalidHandle;
}
```

In der Funktion `writeHandler` in Programmcode 4.25 wird die Überprüfung für `stdin`-, `stdout`- und `stderr`-Handle ebenfalls ersetzt, da der Standard-Input nicht geschrieben werden kann und die anderen beiden Handles noch nicht implementiert sind. Wenn der `log`-Handle geschrieben werden soll, werden hier die Daten in ein neues Array am Heap umkopiert und dann in eine Queue geschrieben, welche in einem anderen Thread periodisch ausgelesen und über die Hardware übertragen wird. Der

Speicher des hier erstellten Arrays muss in dem Thread freigegeben werden, welcher die Daten aus der Queue ausliest. Das Umkopieren ist hier notwendig, da nicht sichergestellt werden kann, dass die Daten, welche vom Modul gesendet wurden, nicht verändert werden oder der Speicher nicht freigegeben wird, bevor die Daten aus der Queue ausgelesen werden können. Die Hardware der LEDs wird beschrieben, wenn der Handle für die LEDs übergeben wird und das übergebene Array mit den zu schreibenden Daten die richtige Größe besitzt. Dazu muss der Handle wieder auf die entsprechende Klasse konvertiert werden. Der Handle des Joysticks kann nicht geschrieben werden, da dies aus Sicht der Hardware nicht möglich ist und auch keinen Sinn macht.

In der Funktion `seekHandler()` ist nichts implementiert und es wird für alle Handles ein Fehlercode für eine unzulässige Operation zurückgegeben. Diese Funktion ist nicht implementiert, weil sie für die verwendete Hardware bis jetzt nicht anwendbar ist. Wird jedoch eine neue Hardware hinzugefügt, welche eine solche Operation benötigt, muss die Funktion `seekHandler()` für den neuen Handle-Typ implementiert werden.

Programmcode 4.26: Funktion zum Konfigurieren von Geräte-Dateien

```
int32_t svc::ioctlHandler(int32_t handle, int32_t function, ioctlArg arg) {
  ValidHandles::DeviceType deviceType;
  ...
  else if (ValidHandles::check(handle, deviceType)) {
    switch (deviceType) {
    case ValidHandles::DeviceType::leds:
      if(function == lomo_ioctl_getLedPos && arg.ptr != nullptr) {
        if(strcmp(arg.ptr, "Green") == 0) {
          return LedsHandler::LedPos::k_Green;
        }
        else if(strcmp(arg.ptr, "Red") == 0) {
          return LedsHandler::LedPos::k_Red;
      } }
      return svc_errorCode_invalidParameter;
    default:
      return svc_errorCode_invalidHandle;
    }
    return svc_errorCode_noError;
  }
  return svc_errorCode_invalidHandle;
}
```

Die Funktion `ioctlHandler()` in Programmcode 4.26 ist nur für den Handle der LEDs implementiert. Für alle anderen Handles wird ein Fehlercode für eine unzulässige Operation zurückgegeben. Für

die LEDs ist eine Funktionalität implementiert, welche die Position der LED im Ausgabe-Array liefert. Die Unterscheidung der LEDs wird durch einen Namen gemacht, welcher als Null-terminierter String in der Variable `arg` des Typs `ioctlArg` übergeben wird. Wenn das Ausgabe-Array mit einem SV-Call gelesen wird, kann diese Position verwendet werden, um die gewünschte LED ein- oder auszuschalten, indem der entsprechende Wert im Ausgabe-Array verändert wird und das Ausgabe-Array mit einem SV-Call geschrieben wird.

In der Klasse `ValidHandles` werden die zur Zeit verwendeten und somit aktiven Handels und der Typ dieser Handles gespeichert. Hierzu werden zwei dynamisch allokierte Arrays verwendet, die vergrößert werden, wenn ein neuer Handle hinzugefügt werden soll, aber kein Platz mehr frei ist. Beim Löschen von Handles, welche nicht mehr in Verwendung sind, werden die entsprechenden Einträge in den Arrays als unbenutzt markiert und können wiederverwendet werden, wenn ein neuer Handle hinzugefügt wird. Die Klasse `ValidHandles` besitzt auch eine Funktion `check()` welche verwendet wird, um zu überprüfen, ob der vom Modul übergebene Handle korrekt ist. Außerdem liefert diese Funktion den Typ des Handles im, als Referenz übergebenen Parameter, `deviceType` zurück.

Programmcode 4.27: Funktion, welche zum Verlassen eines Moduls aufgerufen wird

```
1 void svc_exitHandler(uint32_t *LR, uint32_t *PC) {
2   *PC = &lomo::ModuleWrapper::stopHook;
3   *LR = &lomo::ModuleWrapper::stopHook;
4   OSTCBCurPtr->Opt &= ~OS_OPT_TASK_UNPRIVILEGED;
5   OSIntCtxSw();
6 }
```

Wenn das Modul sich selbst beenden will, kann es dies mit einem SV-Call machen. Dieser SV-Call wird in `svc_exitHandler()` in Programmcode 4.27 abgearbeitet, dazu wird in die statische Funktion `stopHook()` der Klasse `ModuleWrapper` gewechselt und die Option, dass der Thread im unprivilegierten Modus ausgeführt wird, wird entfernt, damit nach dem nächsten Context-Switch der Thread im privilegierten Modus ausgeführt wird. Damit nicht zurück ins Modul, sondern in die Funktion `stopHook()` gesprungen wird, werden die am Stack gespeicherten Register des Link-Registers und des Program Counters verändert. Das Verlassen einer Interrupt-Funktion kann durch einen Sprung auf das Link Register oder dem Wiederherstellen des Program Counters vom Stack ausgeführt werden.

Kapitel 5

Evaluierung

Zwei wichtige Schritte bei jeder Arbeit sind die Validierung und die Verifizierung. Vor allem bei Software-Projekten wird hier klar zwischen den beiden Begriffen unterschieden. Unter Validierung versteht man die Tätigkeit, die (Software-) Spezifikation auf seine Korrektheit in Bezug auf die Anforderungen, Bedürfnisse und Wünsche des „Auftraggebers“[1] zu prüfen. Im Gegensatz dazu steht die Verifizierung. Hierbei wird überprüft, ob das entwickelte System den, in der (Software-) Spezifikation beschriebenen, Anforderungen entspricht. Dieser Unterschied kann in den beiden folgenden, sehr ähnlichen Fragestellungen verdeutlicht werden:

- **Validierung:** Wird das richtige System entwickelt?
- **Verifizierung:** Wird das System richtig entwickelt?

Diese zwei Aspekte sollen in diesem Kapitel betrachtet werden. Zuerst soll geprüft werden, ob die Bedürfnisse, welche aus der Problemstellung hervorgehen, durch das erstellte Konzept erfüllt werden. Danach wird die Implementierung überprüft, ob sie die Anforderungen des Konzepts erfüllt.

Ein wichtiger Hinweis hierzu ist, dass die Validierung vor der Implementierung stattfinden muss, da es sonst sein kann, dass die Implementierung auf Basis einer unvollständigen oder fehlerhaften (Software-) Spezifikation durchgeführt wird. Um den Lesefluss dieser Arbeit zu verbessern, wird jedoch die Validierung gemeinsam mit der Verifizierung in diesem Kapitel behandelt, obwohl diese bereits vor der Implementierung durchgeführt wurde.

[1] Der Begriff „Auftraggeber“ wird hier als sehr loser Begriff verwendet. Ein „Auftraggeber“ kann sowohl ein Kunde sein, wenn es sich um ein kommerzielles Projekt handelt, als auch um den Entwickler oder den Forscher selbst, welcher ein Problem entdeckt, daraus eine Problemstellung formuliert und diese Problemstellung zu lösen versucht.

5.1 Validierung

Da normalerweise sowohl die Anforderungen des „Auftraggebers“ als auch die (Software-) Spezifikation als textuelle Beschreibung vorliegen, ist es nicht möglich hierfür Tests zu entwickelt. Meistens wird hierfür ein Review[2] durch eine andere Person oder in manchen Fällen auch durch dieselbe Person durchgeführt.

Zusammenfassend kann die Problemstellung als Schutz der safety-kritischen Funktionalitäten vor nicht safety-kritischen Funktionalitäten in einem Software-System mit Betriebssystem beschrieben werden. Dies ist eine sehr vereinfachte Beschreibung und zieht eine Reihe an weiteren Anforderungen nach sich:

1. Zerteilung in einzelne Module, mit ausschließlich safety-kritischen oder nicht safety-kritischen Funktionalitäten
2. Definierte Kommunikationsschnittstelle für die Module mit dem Betriebssystem
3. Überwachung der Kommunikationsschnittstelle durch das Betriebssystem
4. Schutz vor illegalen Speicherzugriffen und Dauerschleifen
5. Installation und Deinstallation von Modulen
6. Debuggen von Modulen

In Abbildung 5.1 ist nochmals das Software-Konzept aus Kapitel 3 zu sehen. Punkt 1 der Anforderungen ist im Konzept berücksichtigt, jedoch ist die Zerteilung in einzelne Module Aufgabe des Entwicklers, welcher die Module schreibt. Dieser muss auch darauf achten, dass safety-kritische und nicht safety-kritische Funktionalitäten in getrennten Modulen implementiert werden. Für diese Arbeit ist lediglich die Unterstützung von mehreren Modulen wichtig. Punkt 2, die Kommunikationsschnittstelle, wird im Software-Konzept mit dem API umgesetzt. Punkt 3 und 4 sind abstrakt als Vorhängeschloss und Überwachungskamera dargestellt. Für Punkt 5 und 6 muss es eine externe Schnittstelle mit dem Entwicklungs-PC geben. Diese Schnittstelle ist mit Pfeilen dargestellt und geht vom Entwicklungs-PC über

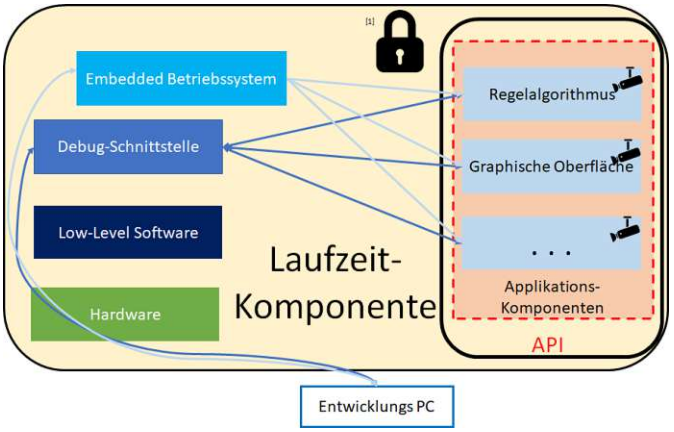

Abbildung 5.1: Software-Konzept aus Kapitel 3

[2] Bei einem Review wird das erstellte Dokument/der erstellte Schaltplan/der erstellte Softwarecode/... durchgesehen und auf die Richtigkeit überprüft. Dies wird meist von einer anderen Person durchgeführt, da eine andere Sichtweise die Wahrscheinlichkeit erhöht, Fehler zu finden.

das Betriebssystem zu den Modulen für die Installation und die Deinstallation und über eine Debug-Schnittstelle in der Laufzeit-Komponente für das Debugging. Die Schnittstelle kann für beide Funktionen dieselbe sein oder es können zwei getrennte Schnittstellen verwendet werden. Damit sind alle Anforderungen, die in der Problemstellung definiert wurden, im Software-Konzept vorhanden.

5.2 Verifizierung

Für die Verifikation können aus dem Software-Konzept Testfälle erstellt werden, welche auf der realen Hardware ausgeführt werden und Aufschluss über die Korrektheit und Effizienz der Software geben.

Um die Effizienz der Softwareerweiterung zu quantifizieren, wird die Zeit, welche benötigt wird, um von einem Thread in einen anderen Thread zu wechseln (Context-Switch), herangezogen. Des Weiteren werden die Vor- und Nachteile von der Ausführung der Module im RAM und im externen Flash betrachtet.

Für die Überprüfung der Korrektheit der Implementierung werden die implementierten Schutzmechanismen getestet. Hierfür wird zum einen versucht, auf eine illegale Speicheradresse zu schreiben, und zum anderen, die Rechenzeit mittels einer Dauerschleife nicht wieder abzugeben und somit zu testen, ob dieses Modul dauerhaft alle anderen Module blockieren kann.

Benötigte Zeit für den Context-Switch

In Abschnitt 4.4 wurde bereits erklärt, dass der Scheduler im PendSV-Interrupt, welcher von einem Timer ausgelöst wird, ausgeführt wird. Im Code des PendSV-Interrupts wurden die Änderungen, welche in Programmcode 4.12 zu sehen sind, eingefügt und diese ändern den Zustand zu privilegiert oder unprivilegiert, je nachdem, welcher Thread als nächstes ausgeführt wird. Der gesamte Context-Switch benötigt 2 μs inklusive der Änderungen[3]. Da die Änderung nur aus sieben einfachen Assemblerbefehlen besteht, benötigt diese kaum mehr Zeit. Um genau zu sein, benötigt sie sieben Prozessorzyklen mehr, was bei 80 MHz eine Zeit von 87,5 ns ergibt. Dieser Mehraufwand im Context-Switch hat kaum einen Einfluss auf die Auslastung des Mikrocontrollers.

Wenn mehr als ein Modul verwendet wird, so müssen zusätzlich die benötigten MPU-Regionen des neuen Threads aktiviert und die MPU-Regionen des alten Threads deaktiviert werden. Jedes Modul benötigt zwei Regionen, eine für den Bereich des Programmcodes und eine für den Bereich im RAM. Das Aktivieren und Deaktivieren dauert pro MPU-Region 14 μs und somit insgesamt 56 μs. Werden zusätzlich mehr als vier Module gleichzeitig verwendet, so müssen die MPU-Regionen umkonfiguriert werden, da der Mikrocontroller nur acht konfigurierbare MPU-Regionen besitzt. Dieses Umkonfigu-

[3]Die Zeitbasis des System-Timers, welcher für diese Messung verwendet wurde, läuft mit 1 MHz und damit liegt die Genauigkeit dieser Messung bei 1 μs.

rieren benötigt zusätzlich 19 μs, dies ergibt eine gesamt Zeit von 77 μs für den Context-Switch. Dies ist ein erheblicher Mehraufwand im Context-Switch und kann zu einer signifikanten Steigerung der Auslastung des Mikrocontrollers führen. Es ist vor allem dann ein Problem, wenn sehr viele Module mit kurzen Zykluszeiten ausgeführt werden, da in diesem Fall sehr oft der Context-Switch ausgeführt werden muss. Dies ist eine Einschränkung des gewählten Mikrocontrollers und kann mittels Software nicht umgangen werden. Daher muss diese Einschränkung bei der Erstellung und Implementierung von Modulen berücksichtigt werden.

RAM-Bedarf versus Ausführungszeit

Wie bereits in Kapitel 4 erwähnt, können die Module im RAM oder im externen Flash ausgeführt werden. Der große Nachteil bei der Ausführung der Module im RAM ist, dass der RAM-Speicher sehr begrenzt ist und dadurch nur sehr wenige oder sehr kompakte Module dort gespeichert und ausgeführt werden können. Ein weiterer Nachteil ist, dass die Module vor dem Starten aus dem externen Flash in den RAM kopiert werden müssen. Dies ist meistens jedoch ein akzeptabler Nachteil, da dieses Umkopieren vor dem Starten der Module passiert, wenn das System noch in keinem safety-kritischen Zustand ist und dadurch diese Zeit keinen Einfluss auf das Echtzeitverhalten hat. Der Vorteil von der Ausführung im RAM ist jedoch, dass nur minimale Geschwindigkeitsverluste aufgrund des Speicherzugriffes entstehen. Diese Geschwindigkeitsverluste sind aber nicht vermeidbar und treten auch bei dem restlichen Code auf, welcher im internen Flash gespeichert ist.

Der Nachteil der Ausführung aus dem externen Flash ist die Reduktion der Ausführungsgeschwindigkeit aufgrund der reduzierten Übertragungsgeschwindigkeit und der geringeren Anzahl an gleichzeitig übertragenen Bits. Das verwendete, externe Flash, auf dem die Module gespeichert sind, ist über eine QSPI-Schnittstelle mit einer maximalen Frequenz von 40 MHz angebunden. Dies reduziert die Ausführungsgeschwindigkeit um den Faktor 16. Diese Reduktion teilt sich auf die reduzierte Übertragungsgeschwindigkeit (Faktor 2) und die reduzierte Datenbreite (Faktor 8) auf.

Welche dieser zwei Varianten zu bevorzugen ist, hängt von der Anwendung ab, für die es eingesetzt werden soll. So wird für ein System mit sehr kurzen Zykluszeiten (< 10 ms) und wenigen, kleinen Modulen die Ausführung im RAM besser sein, wohingegen bei einem System mit längeren Zykluszeiten (> 100 ms) und mehreren und größeren Modulen die Ausführung aus dem externen Flash zu bevorzugen sein wird. Unter Umständen kann es auch nötig sein, eine Mischlösung zu implementieren, bei der ein Teil der Module im RAM und der Rest der Module im externen Flash ausgeführt werden.

Zusätzlicher Speicherbedarf durch Modularisierung

Aufgrund der Modularisierung ergeben sich weitere Einschränkungen im Hinblick auf den Speicherbedarf. Da in der Implementierung noch keine gemeinsam genutzten Bibliotheken (engl. shared libraries) berücksichtigt werden, kann dies dazu führen, dass der Programmcode von Bibliotheksfunktionen in mehreren Modulen enthalten ist, weil dieser Code beim Kompilieren der einzelnen Module vom Compiler hinzugefügt wird. Durch die Verwendung von gemeinsam genutzten Bibliotheken kann dem Compiler mitgeteilt werden, wo diese Bibliotheksfunktionen im Betriebssystem zu finden sind. Dadurch werden diese Funktionen vom Compiler nicht mehr den Modulen hinzugefügt und dies kann unter Umständen den Speicherbedarf der Module drastisch reduzieren.

Auch die Implementierung der Betriebssystemerweiterung benötigt zusätzlichen Speicher, sowohl im Flash als auch im RAM. Der Programmcode selbst ist kein Problem, da dieser sowieso im internen Flash gespeichert ist und dieses ausreichend groß ist, um die wenigen Kilo-Byte an zusätzlichem Programmcode aufzunehmen. Jedoch wird auch zusätzlicher Speicher im RAM benötigt. Dieser Speicherbedarf kann in zwei Gruppen unterteilt werden. Die erste Gruppe ist jener Speicherbedarf, welcher von der Änderung benötigt wird und unabhängig von der Anzahl der installierten Module ist. Die zweite Gruppe beinhaltet den Speicherbedarf, welcher direkt von der Anzahl der installierten Module abhängt. Der RAM-Speicherbedarf der ersten Gruppe beläuft sich auf etwas mehr als 1 kB. Der RAM-Bedarf der zweiten Gruppe beträgt ca. 300 Byte pro Modul. Dieser RAM-Speicherbedarf ist unabhängig davon, ob die Module im RAM oder in einem externen Flash gespeichert sind.

Schutz vor illegalem Speicherzugriff

Zum Testen des Schutzes vor illegalen Speicherzugriffen wurde in einem Modul folgende Zeile hinzugefügt:

```
reinterpret_cast<Data*>(0)->cycleCount = 0;
```

Diese Speicherzugriffsverletzung wird in dieser exakten Form nie in einem Programmcode vorkommen, da leicht zu erkennen ist, dass dies eine Speicherzugriffsverletzung ist. Jedoch gibt es einen ähnlichen Fall, welcher sehr häufig auftreten kann. Hierbei wird versucht, neuen Speicher zu allokieren. Sollte kein Speicher mehr allokiert werden können, wird ein Fehler signalisiert, indem ein Zeiger auf die Adresse Null (null pointer) retourniert wird. Wenn dies nicht überprüft wird und versucht wird, auf diese Adresse zuzugreifen, so führt dies zu derselben Situation wie bei der simulierten Speicherzugriffsverletzung. Der einzige Unterschied ist, dass der Zeiger auf die Adresse Null in einer Variable gespeichert ist und daher nicht bei einer statischen Analyse des Programmcodes ersichtlich ist, da der Fehler nur bei einer Ausführung des Programms auftritt.

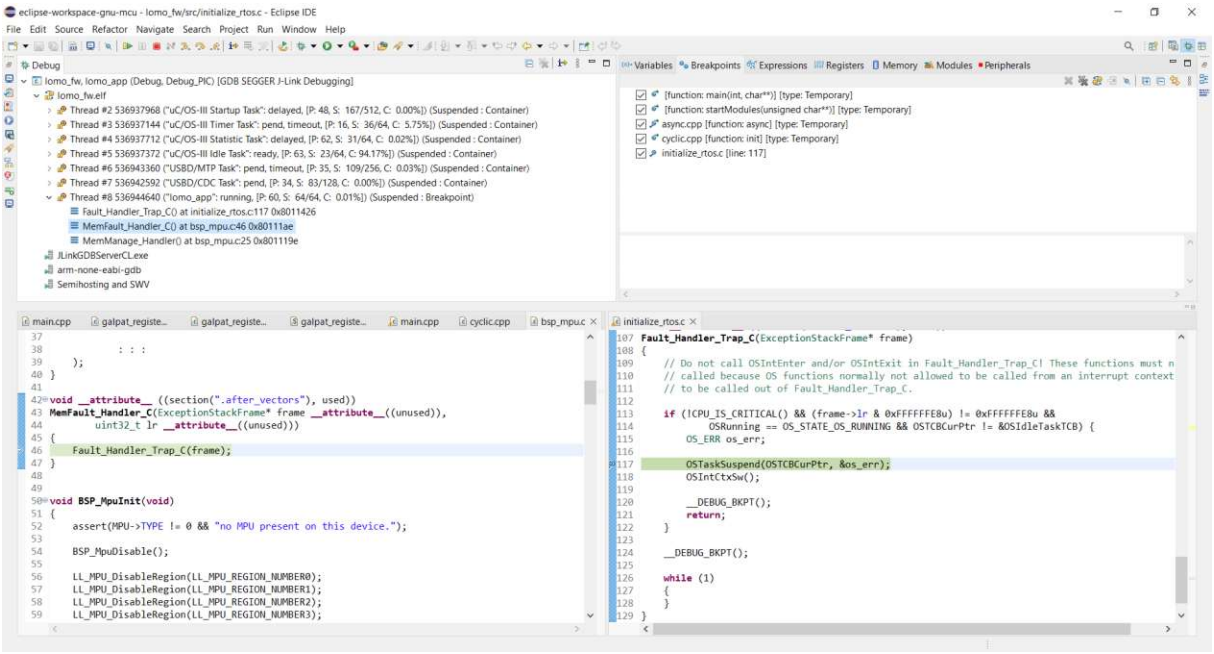

Abbildung 5.2: Beendigung eines fehlerhaften Moduls aufgrund einer Speicherzugriffsverletzung

In Abbildung 5.2 ist zu sehen, dass ein spezieller Interrupt, eine sogenannte Exception, ausgelöst wird. Diese Exception wird in `MemFault_Handler_C` abgefangen und die Funktion `Fault_Handler_Trap_C` aufgerufen. Die Funktion `Fault_Handler_Trap_C` wurde bereits in Abschnitt 4.3 erklärt. Sie überprüft, ob es sich bei dem Thread, welcher den Fehler ausgelöst hat, um ein Modul handelt und wenn ein Modul den Fehler verursacht hat, dann wird dieses Modul beendet. Im oberen linken Teil von Abbildung 5.2 sind die einzelnen Threads und der Call-Stack des Moduls zu sehen. Links unten ist die Funktion `MemFault_Handler_C` und rechts unten die Funktion `Fault_Handler_Trap_C` zu sehen. Diese ist grün markiert, da der Mikrocontroller hier mit einem Break-Point angehalten wurde. Der Break-Point wurde auf die Funktion gesetzt, welche den Thread des Moduls suspendiert, damit dieser nicht wieder ausgeführt wird und danach wird ein Context-Switch durchgeführt, damit das fehlerhafte Modul nicht weiter ausgeführt wird.

Schutz vor blockierenden Modulen

Zum Testen des Schutzes vor blockierenden Modulen wurde in einem Modul eine Dauerschleife hinzugefügt. Dadurch gibt das Modul die Rechenzeit nicht rechtzeitig ab und der Scheduler der Module erkennt, dass das Modul mehr als 50 % seiner Zykluszeit aktiv ist. Damit die Auslastung des Mikrocontrollers nicht zu hoch wird, wird die Zykluszeit des Moduls verdoppelt. Wenn die neue Zykluszeit die maximale Zykluszeit des Moduls überschreitet, so wird das Modul beendet. Dies ist in Abbildung 5.3 zu sehen. Links unten ist die Dauerschleife zu sehen, welche verhindert, dass die Rechenzeit abgegeben

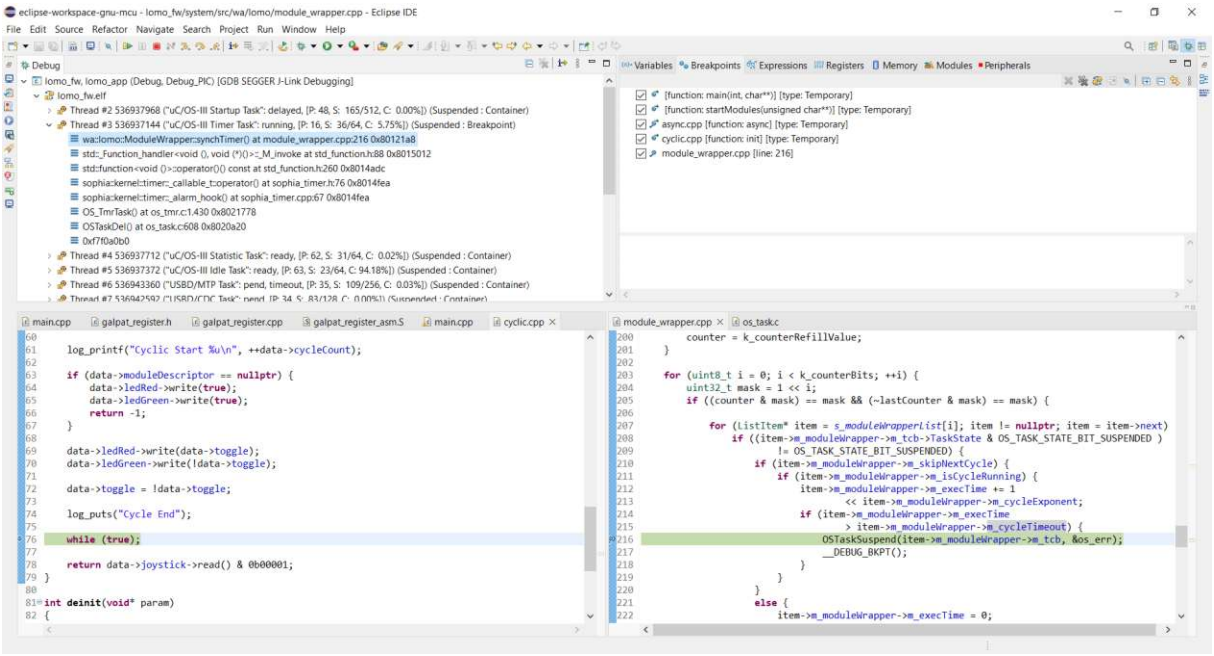

Abbildung 5.3: Beendigung eines fehlerhaften Moduls aufgrund einer zu langen Zykluszeit

wird. Rechts unten ist der Code des Modul-Schedulers zu sehen, welcher das Modul suspendiert, weil die maximale Zykluszeit des Moduls überschritten wurde. Dadurch wird dieses fehlerhafte Modul nicht weiter ausgeführt.

Kapitel 6

Fazit und Ausblick

Am Ende dieser Arbeit soll in diesem Kapitel zuerst ein kurzer Rückblick auf die Problemstellung und die Ziele dieser Arbeit gegeben werden und ein Fazit gezogen werden, ob diese Probleme gelöst und die Ziele erreicht wurden. Danach soll ein Ausblick auf Themen gegeben werden, welche diese Arbeit erweitern könnten.

6.1 Fazit

In Abschnitt 1.2 wurde die Problemstellung definiert, welche in dieser Arbeit behandelt werden soll. Für safety-kritische Anwendungen auf ressourcenarmen Mikrocontrollern gibt es noch kein Betriebssystem, welches die Trennung von Applikations-Komponenten von der Laufzeit-Komponente[1] ermöglicht. Diese Applikations-Komponenten oder Module sollen unabhängige Programme sein und auf dem Gerät installiert und deinstalliert werden können. Diese Trennung und das Installieren/Deinstallieren wurden umgesetzt und es können verschiedenste Module und auch mehrere unterschiedliche Module gleichzeitig installiert werden. Das Übertragen und Installieren/Deinstallieren der Module funktioniert per Kopieren/Löschen über das Dateisystem des Entwicklungs-PCs mittels MTP einer USB-Schnittstelle.

Um diese Trennung zwischen Applikations-Komponenten und Laufzeit-Komponente für safety-kritische Anwendungen verwenden zu können, muss die Laufzeit-Komponente und alle safety-relevanten Applikations-Komponenten vor den nicht safety-kritischen Applikations-Komponenten geschützt werden. Dieser Schutz gliedert sich in Überwachung der Kommunikation über das definierte API, Schutz vor illegalen Speicherzugriffen und Schutz vor Modulen, welche zu viel Rechenzeit benötigen. Diese Anforderungen wurden mit der Evaluierung aus Kapitel 5 überprüft und bewiesen, dass die in Abschnitt 1.2 definierten Aufgaben umgesetzt wurden.

[1] Die Laufzeit-Komponente beinhaltet sowohl das Betriebssystem als auch die Low-Level Funktionalitäten mit den Hardware-Treibern.

6.2 Ausblick

Diese Arbeit beinhaltet ein paar Einschränkungen, welche in zukünftigen Projekten behandelt werden könnten. Zum einen gibt es nur eine rudimentäre Debug-Unterstützung und zum anderen können in den Modulen keine globalen Variablen und keine virtuellen Funktionen verwendet werden.

6.2.1 Debugging

Zu jeder Software-Entwicklung gehört das Debugging, um Fehler im Code zu finden. Die implementierte Debug-Unterstützung beschränkt sich auf das sogenannte „printf"-Debugging, bei welchem der Zustand des Programms und der Wert von Variablen als Text zur Laufzeit ausgegeben werden können, ohne dabei den Mikrocontroller anhalten zu müssen. Dies ist eine sehr rudimentäre Debug-Unterstützung und es ist oft sehr aufwändig, damit einen Fehler zu finden, da man an jeder Stelle die Funktion `printf()` einfügen muss, an der man den Zustand des Programms aufzeichnen oder den Wert einer Variable wissen will. Des Weiteren muss man das Modul immer wieder neu kompilieren und auf dem Mikrocontroller installieren, um einen weiteren Zustand oder eine weitere Variable zu beobachten.

Es wurde bereits in Abschnitt 3.5 erwähnt, dass ein Debugging mit einem normalen Debugger nicht möglich ist, da dabei alle Threads am Mikrocontroller angehalten werden. Um dieses Problem zu lösen, gibt es im Internet bereits einen Lösungsansatz: Monitor for Remote Inspection (MRI) [95].

MRI ermöglicht es bei einem Break-Point den aktuellen Context abzuspeichern und in einen Debug-Code zu wechseln, welcher die Steuerung des Debuggings übernimmt, ohne den Prozessor dauerhaft anzuhalten. Das Problem dabei ist, dass diese Implementierung nicht für Betriebssysteme mit mehreren Threads entwickelt wurde. Die Kompatibilität mit einem Betriebssystem mit mehreren Threads müsste hinzugefügt werden, um für diese Arbeit geeignet zu sein. Außerdem wird der Debugger im Interrupt Context ausgeführt, was dazu führt, dass alle Threads angehalten werden. Daher muss der Debugger-Code umgeschrieben werden, damit dieser in einem eigenen Thread abgearbeitet wird und alle höherprioren Threads weiterhin ausgeführt werden. Da nur für Module das Debugging ermöglicht werden soll, dürfen nur jene Threads angezeigt und gestoppt werden, welche zu einem Modul gehören.

6.2.2 Position Independent Executable

Eine weitere Einschränkung ist, dass keine globalen Variablen verwendet werden können, um genau zu sein, dürfen sich keine Variablen im RAM befinden außer am Stack. Das heißt, es müssen die Data- und Block-Starting-Symbol (BSS)-Section[2] leer sein und es dürfen keine virtuellen Funktionen verwendet werden.

[2] In der Data-Section werden globale Variablen abgelegt, welche einen Initialisierungswert besitzen, wohingegen in der BSS-Section globale Variablen abgelegt werden, welche keinen Initialisierungswert besitzen.

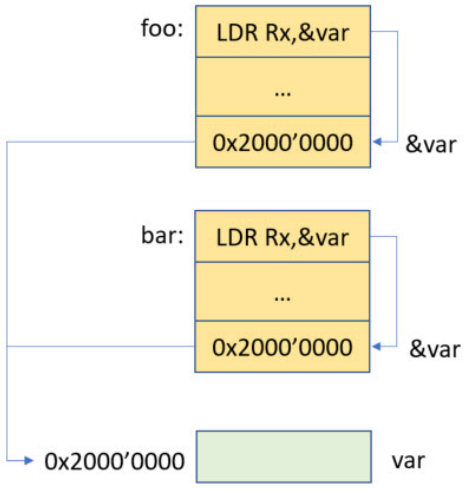

Abbildung 6.1: Zugriff auf eine Variable ohne Verwendung von PIE (gelb: Code im Flash oder RAM; grün: Variablen im RAM) [104]

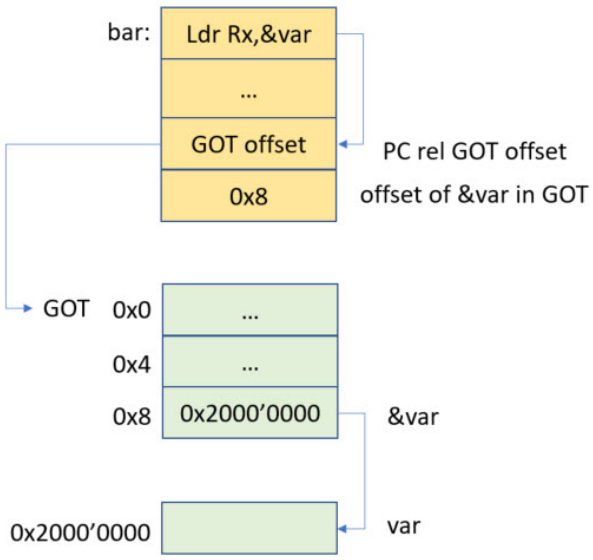

Abbildung 6.2: Zugriff auf eine Variable unter Verwendung von PIE und GOT (gelb: Code im Flash oder RAM; grün: GOT und Variablen im RAM) [105]

In Abbildung 6.1 ist der Zugriff auf Variablen ohne die Verwendung von Position Independent Executable (PIE) illustriert. Hier wird über eine fixe Adresse, welche im Programmcode hinterlegt ist, zugegriffen.

In Abbildung 6.2 ist die Änderung des Zugriffs auf eine globale Variable veranschaulicht, wenn PIE verwendet wird. Hier gibt es einen Bereich im Code, in welchem die Adressen der globalen Variablen gespeichert sind, und im Code ist der Offset in Bezug auf den Program-Counter des Global Offset Table (GOT) gespeichert. Dies ist auf den ersten Blick noch keine Verbesserung, da hierfür die Adressen wieder bei der Kompilierung festgelegt werden müssen. ARM Cortex-M Prozessoren können aber ein Register für die Speicherung der Adresse des GOT verwenden, damit kann der GOT in den RAM kopiert werden und dort die Adressen der Variablen angepasst werden, sodass diese auf die richtige Adresse zeigen, nachdem die Variablen vom Betriebssystem in den RAM kopiert wurden. Für die Verwendung von virtuellen Funktionen muss die Adresse des Virtual Table (VTable)s in dem GOT gespeichert werden und die Adressen im VTable angepasst werden.

Um dies zu nutzen, muss diese Arbeit um ein paar Schritte beim Umkopieren der Module erweitert werden. Es müssen der GOT, der VTable und die globalen Variablen in den RAM kopiert werden. Danach müssen die Adressen im GOT auf die Adressen der globalen Variablen und die Adresse des VTable im RAM gesetzt werden. Daraufhin müssen die Adressen der virtuellen Funktionen im VTable auf die richtigen Adressen gesetzt werden, auf denen die entsprechenden Funktionen des Moduls nach dem Umkopieren gespeichert sind.

Literaturverzeichnis

[1] "Institut für Computertechnik [homepage]," last accessed: Februar 15, 2022. [Online]. Available: https://www.ict.tuwien.ac.at

[2] L. S. Vailshery, "Global IoT end-user spending worldwide 2017-2025," Statista [Website], January 2021, last accessed: Mai 05, 2022. [Online]. Available: https://www.statista.com/statistics/976313/global-iot-market-size/#statisticContainer

[3] M. Placek, "Industrial IoT - market size worldwide 2020-2028," Statista [Website], March 2022, last accessed: Mai 05, 2022. [Online]. Available: https://www.statista.com/statistics/611004/global-industrial-internet-of-things-market-size/

[4] rsameser, "Soft real-time on windows iot enterprise," Microsoft Documentation [Website], October 2021, last accessed: November 29, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/soft-real-time/soft-real-time

[5] "Arm cortex-family," last accessed: November 07, 2022. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu

[6] "Arm cortex-a," last accessed: November 07, 2022. [Online]. Available: https://en.wikipedia.org/wiki/ARM_Cortex-A

[7] "Arm cortex-m," last accessed: November 07, 2022. [Online]. Available: https://en.wikipedia.org/wiki/ARM_Cortex-M

[8] "Arm cortex-r," last accessed: November 07, 2022. [Online]. Available: https://en.wikipedia.org/wiki/ARM_Cortex-R

[9] "UNIX [image]," last accessed: November 08, 2022. [Online]. Available: https://unix.org/images/unix-an-open-group-standard.png

[10] "UNIX," last accessed: November 10, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Unix

[11] B. W. Kernighan and R. Pike, *The UNIX Programming Environment.* Prentice-Hall, Inc., 1984, ch. Preface, p. viii, last accessed: November 09, 2022. [Online]. Available: http://files.catwell.info/misc/mirror/the-unix-programming-environment-kernighan-pike.pdf

[12] E. S. Raymond, *The Art of UNIX Programming.* Addison-Wesley, September 1984, archived from the original: October 31, 2022. Last accessed: November 09, 2022. [Online]. Available: https://web.archive.org/web/20221031052418/http://www.catb.org/~esr/writings/taoup/html/

[13] D. M. Ritchie and K. L. Thompson, "The UNIX time-sharing system," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, July-August 1978, archived from the original: January 19, 2013. Last accessed March: November 10, 2022. [Online]. Available: https://archive.org/details/bstj57-6-1905

[14] "UNIX timeline [image]," last accessed: November 16, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/c/cd/Unix_timeline.en.svg

[15] "Android [image]," last accessed: November 08, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/3/3b/Android_new_logo_2019.svg/1280px-Android_new_logo_2019.svg.png

[16] "Android common kernels," Android Developers [Website], October 2022, last accessed: November 14, 2022. [Online]. Available: https://source.android.com/docs/core/architecture/kernel/android-common

[17] "Android," last accessed: November 10, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Android_(operating_system)

[18] R. Paul, "Dream(sheep++): A developer's introduction to google android," Ars Technica [Website], Febuary 2009, last accessed: November 14, 2022. [Online]. Available: https://arstechnica.com/gadgets/2009/02/an-introduction-to-google-android-for-developers/

[19] S. J. Vaughan-Nichols, "Android/linux kernel fight continues," Computerworld [Website], September 2010, last accessed: November 14, 2022. [Online]. Available: https://www.computerworld.com/article/2469087/android-linux-kernel-fight-continues.html

[20] "Android Application Binary Interface," Android Developers [Website], August 2022, last accessed: November 14, 2022. [Online]. Available: https://developer.android.com/ndk/guides/abis

[21] "Touch devices," Android Open Source Project, archived from the original: January 25, 2012. Last accessed: November 10, 2022. [Online]. Available: https://web.archive.org/web/20120125061950/http://source.android.com/tech/input/touch-devices.html

[22] "Sensors overview," Android Open Source Project, archived from the original: November 09, 2012. Last accessed: November 10, 2022. [Online]. Available: https://web.archive.org/web/20221109034820/https://developer.android.com/guide/topics/sensors/sensors_overview

[23] V. Matos, "Application's life cycle," [grail.cba.csuohio.edu], January 2012, archived from the original: February 22, 2014. Last accessed: November 14, 2022. [Online]. Available: https://web.archive.org/web/20140222153131/http://grail.cba.csuohio.edu/~matos/notes/cis-493/lecture-notes/Android-Chapter03-Life-Cycle.pdf

[24] R. Meier, *Files, Saving State and Preferences*, 2nd ed. John Wiley Sons, March 2012, ch. 7, pp. 221–250, last accessed: November 14, 2022. [Online]. Available: https://books.google.at/books?id=g3hAdK1IBkYC&pg=PT53&redir_esc=y#v=onepage&q&f=false

[25] "Android software stack [image]," last accessed: November 10, 2022. [Online]. Available: https://developer.android.com/static/guide/platform/images/android-stack_2x.png

[26] "Android software stack," last accessed: November 15, 2022. [Online]. Available: https://developer.android.com/guide/platform

[27] C. Toombs, "Meet art, part 1: The new super-fast android runtime google has been working on in secret for over 2 years debuts in kitkat," Android Police [Website], November 2013, last accessed: November 14, 2022. [Online]. Available: https://www.androidpolice.com/2013/11/06/meet-art-part-1-the-new-super-fast-android-runtime-google-has-been-working-on-in-secret-for-over-2-years-debuts-in-kitkat/

[28] "Native (computing)," last accessed: November 15, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Native_(computing)

[29] "Linux [image]," last accessed: November 08, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/dd/Linux_logo.jpg

[30] "Linux," last accessed: November 15, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Linux

[31] "Linux and GNU," last accessed: November 16, 2022. [Online]. Available: https://www.gnu.org/gnu/linux-and-gnu.html

[32] B. Levine, "Linux' 22th birthday is commemorated - subtly - by creator," CMSWIRE [Website], August 2013, last accessed: November 16, 2022. [Online]. Available: https://www.cmswire.com/cms/information-management/linux-22th-birthday-is-commemorated-subtly-by-creator-022244.php

[33] T. O'Reilly, C. Toporek, and et.al., "Anatomy of a linux system," July 2001, last accessed: November 17, 2022. [Online]. Available: https://personalpages.manchester.ac.uk/staff/m.dodge/cybergeography/atlas/linux_anatomy.pdf

[34] "User space and kernel space," last accessed: November 17, 2022. [Online]. Available: https://en.wikipedia.org/wiki/User_space_and_kernel_space

[35] M. T. Jones, "Inside the linux boot process," IBM Developer [Website], May 2006, archived from the original: April 02, 2019. Last accessed: November 17, 2022. [Online]. Available: https://personalpages.manchester.ac.uk/staff/m.dodge/cybergeography/atlas/linux_anatomy.pdf

[36] J. Beningo, "How to select your embedded systems Operating System: OS characteristics," Embedded.com [Website], May 20022, last accessed: November 17, 2022. [Online]. Available: https://www.embedded.com/how-to-select-your-embedded-systems-operating-system-os-characteristics/

[37] "Embeddable Linux Kernel Subset," [GitHub Repository], last accessed: November 17, 2022. [Online]. Available: https://github.com/jbruchon/elks

[38] "Getting started with Embeddable Linux Kernel Subset," [GitHub Repository], last accessed: November 17, 2022. [Online]. Available: https://htmlpreview.github.io/?https://raw.githubusercontent.com/jbruchon/elks/master/Documentation/html/user/getting_started_with_elks.htmls

[39] "Raspberry pi OS," last accessed: November 17, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi_OS

[40] "Raspberry pi," last accessed: November 17, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi

[41] "Raspbery pi pico rp2040," last accessed: November 17, 2022. [Online]. Available: https://en.wikipedia.org/wiki/RP2040

[42] "macOS [image]," last accessed: November 08, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/3/30/MacOS_logo.svg/1024px-MacOS_logo.svg.png

[43] Lucy, "Inside the mac OS x kernel," in *24th Chaos Communication Congress 24C3*, 2007, last accessed: November 21, 2022. [Online]. Available: https://fahrplan.events.ccc.de/congress/2007/Fahrplan/attachments/986_inside_the_mac_osx_kernel.pdf

[44] "Darwin (Operating System)," last accessed: November 21, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Darwin_(operating_system)#cite_note-5

[45] "macOS," last accessed: November 21, 2022. [Online]. Available: https://en.wikipedia.org/wiki/MacOS#cite_note-aqua23-58

[46] "About developing for mac," Apple Developer [Website], September 2015, last accessed: November 21, 2022. [Online]. Available: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html

[47] B. Steele, "Nasa wise deputy project scientist amy mainzer on the apple //e and kinect-powered laptops," engadget [Website], May 2013, last accessed: November 22, 2022. [Online]. Available: https://www.engadget.com/2013-05-24-engadget-questionnaire-nasa-amy-mainzer.html

[48] "Sandbox (computer security)," last accessed: November 22, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Sandbox_(computer_security)

[49] D. Counsell, "Not on the mac app store," Dan Counsell [Website], November 2015, archived from the original: June 14, 2017. Last accessed: November 22, 2022. [Online]. Available: https://web.archive.org/web/20151208082109/https://dancounsell.typed.com/articles/not-on-the-mac-app-store

[50] "Distribute outside the mac app store (macOS)," Apple Xcode Help [Website], 2020, last accessed: November 22, 2022. [Online]. Available: https://help.apple.com/xcode/mac/current/#/dev033e997ca

[51] "Windows [image]," last accessed: November 23, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e2/Windows_logo_and_wordmark_-_2021.svg/2560px-Windows_logo_and_wordmark_-_2021.svg.png

[52] "Microsoft windows," last accessed: November 22, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Windows

[53] "Windows nt," last accessed: November 22, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Windows_NT

[54] "Windows api," last accessed: November 22, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Windows_API

[55] M. Russinovich, "Inside native applications," Microsoft Documentation [Website], November 2006, last accessed: November 24, 2022. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/resources/inside-native-applications

[56] "Windows api index," Microsoft Documentation [Website], May 2022, last accessed: November 23, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

[57] "Windows shell," Microsoft Documentation [Website], July 2021, last accessed: November 23, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/shell/shell-entry

[58] "Windows iot," last accessed: November 28, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Windows_IoT

[59] rsameser, "Getting started with windows iot enterprise," Microsoft Documentation [Website], October 2022, last accessed: November 28, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/getting_started

[60] rsameser and sydbruck, "Minimum hardware requirements for windows iot enterprise," Microsoft Documentation [Website], July 2022, last accessed: November 29, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/hardware-guidance/hardware_requirements

[61] "Windows 10 minimum hardware requirements," December 2019, last accessed: November 29, 2022. [Online]. Available: https://download.microsoft.com/download/c/1/5/c150e1ca-4a55-4a7e-94c5-bfc8c2e785c5/Windows%2010%20Minimum%20Hardware%20Requirements.pdf

[62] rsameser and D. Jahiu, "Embedded mode," Microsoft Documentation [Website], June 2022, last accessed: November 29, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/os-features/embedded-mode

[63] T. Warwick, C. McClister, v chmccl, rsameser, D. Lee, nehinnan, M. Wojciakowski, S. Cai, Sara, H. B., N. Schonning, K. Baker, and N. R. Kedia, "An overview of windows 10 iot core," Microsoft Documentation [Website], November 2022, last accessed: November 29, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot-core/windows-iot-core

[64] T. Warwick, v chmccl, rsameser, M. Wojciakowski, and Sara, "An overview of windows server iot 2019," Microsoft Documentation [Website], November 2022, last accessed: December 01, 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot-core/windows-server

[65] "Virtual machine," last accessed: December 01, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Virtual_machine

[66] "Virtualbox [image]," last accessed: November 08, 2022. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/d5/Virtualbox_logo.png?20150209215936

[67] "Welcome to virtualbox.org!" VirtualBox [Website], last accessed: December 02, 2022. [Online]. Available: https://www.virtualbox.org/

[68] "Java [image]," last accessed: Dezember 15, 2022. [Online]. Available: https://dev.java/assets/images/jug-program-logo.png

[69] B. Venners, *Inside the Java Virtual Machine*, 2nd ed. Computing McGraw-Hill, January 2000, ch. Chapter 1: Introduction to Java's Architecture, last accessed: December 06, 2022. [Online]. Available: https://www.artima.com/insidejvm/ed2/index.html

[70] "Java platform, standard edition," last accessed: December 12, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition

[71] "Oracle java me embedded getting started," Oracle [Website], last accessed: December 12, 2022. [Online]. Available: https://www.oracle.com/java/technologies/javame-embedded/javame-embedded-getstarted.html

[72] "Java platform, micro edition," last accessed: December 12, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Java_Platform,_Micro_Edition

[73] "Javatm me embedded profile version 8," Oracle [Website], 2014, last accessed: December 15, 2022. [Online]. Available: https://docs.oracle.com/javame/8.0/api/meep/api/index.html

[74] "Stm32f427xx stm32f429xx," January 2018, last accessed: December 15, 2022. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f429ni.pdf

[75] "Stm32429i-eval," September 2013, last accessed: December 15, 2022. [Online]. Available: https://www.st.com/resource/en/data_brief/stm32429i-eval.pdf

[76] "Stm32f745xx stm32f746xx," February 2016, last accessed: December 15, 2022. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f746ng.pdf

[77] "32f746gdiscovery," December 2019, last accessed: December 15, 2022. [Online]. Available: https://www.st.com/resource/en/data_brief/32f746gdiscovery.pdf

[78] "Java platform micro edition embedded 8 stack [image]," last accessed: December 12, 2022. [Online]. Available: https://www.oracle.com/ocom/groups/public/@ocompublic/documents/webcontent/1844141.jpeg

[79] "freeRTOS™ [image]," last accessed: November 08, 2022. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/logo-1.jpg

[80] "Freertos™," freeRTOS [Website], last accessed: December 23, 2022. [Online]. Available: https://www.freertos.org/

[81] "Freertos™ kernel ports," freeRTOS [Website], last accessed: December 23, 2022. [Online]. Available: https://www.freertos.org/RTOS_ports.html

[82] "Freertos kernel developer docs[including subchapters accessible via sitemap links]," freeRTOS [Website], last accessed: January 09, 2023. [Online]. Available: https://www.freertos.org/features.html

[83] "Rtos task notifications," freeRTOS [Website], last accessed: January 16, 2023. [Online]. Available: https://www.freertos.org/RTOS-task-notifications.html

[84] "Keil rtx [image]," last accessed: January 19, 2023. [Online]. Available: https://www.psacertified.org/app/uploads/2019/02/Keil-rtx5-final-600x0-c-default.png

[85] "Rtx real-time operating system," Keil [Website], last accessed: January 19, 2023. [Online]. Available: https://www.keil.com/arm/rl-arm/kernel.asp

[86] "Rtx5 rtos," Arm Developer [Website], last accessed: January 19, 2023. [Online]. Available: https://developer.arm.com/Tools%20and%20Software/Keil%20MDK/RTX5%20RTOS#Overview

[87] "Rtx v5 implementation," Arm [GitHub Repository], May 2022, last accessed: January 19, 2023. [Online]. Available: https://arm-software.github.io/CMSIS_5/RTOS2/html/theory_of_operation.html

[88] "Micro-Controller Operating Systems-III [image]," last accessed: October 12, 2022. [Online]. Available: https://weston-embedded.com/images/micrium/product_lo_ucos.png

[89] "What is micrium?" Weston Embedded [Website], last accessed: January 17, 2023. [Online]. Available: https://weston-embedded.com/about-micrium

[90] "Real-time kernels: µc/os-ii and µc/os-iii," Weston Embedded [Website], last accessed: January 17, 2023. [Online]. Available: https://weston-embedded.com/micrium-kernels

[91] *User's Manual.* Micriμm Press, 2010, last accessed: January 17, 2023. [Online]. Available: https://www.analog.com/media/en/dsp-documentation/software-manuals/Micrium-uCOS-III-UsersManual.pdf

[92] "Micro-Controller Operating Systems-III ports and drivers," Micriμm [Website], last accessed: January 17, 2023. [Online]. Available: https://micrium.atlassian.net/wiki/spaces/osiiidoc/pages/132550/C+OS-III+Ports+and+Drivers

[93] "File input/output," cppreference [Website], last accessed: January 24, 2023. [Online]. Available: https://en.cppreference.com/w/c/io

[94] M. Kerrisk, *FILE I/O: THE UNIVERSAL I/O MODEL.* No Starch Press, October 2010, ch. 4, last accessed: January 24, 2023. [Online]. Available: https://man7.org/tlpi/download/TLPI-04-File_IO_The_Universal_IO_Model.pdf

[95] A. Green, "Mri - monitor for remote inspection," [Github Repository], last accessed: January 25, 2023. [Online]. Available: https://github.com/adamgreen/mri

[96] "STMicroelectronics arm prozessor [image]," last accessed: October 12, 2022. [Online]. Available: https://www.st.com/content/dam/arm-cortex-m/M4-Corenew.jpg

[97] "Segger [image]," last accessed: October 12, 2022. [Online]. Available: https://c.a.segger.com/fileadmin/images/about-us/Media/Preview/segger-logo-outlines-the-embedded-experts_600x.png

[98] "Embedded usb-device stack von segger [image]," last accessed: October 12, 2022. [Online]. Available: https://www.segger.com/products/connectivity/emusb-device/

[99] "Eclipse [image]," last accessed: October 12, 2022. [Online]. Available: https://www.eclipse.org/org/artwork/

[100] "GNU-MCU-eclipse [image]," last accessed: October 12, 2022. [Online]. Available: https://avatars.githubusercontent.com/u/26934776?s=200&v=4

[101] "GNU-Compiler-Collection [image]," last accessed: October 12, 2022. [Online]. Available: https://de.wikipedia.org/wiki/GNU_Compiler_Collection#/media/Datei:GNU_Compiler_Collection_logo.svg

[102] "GNU-Debugger [image]," last accessed: October 12, 2022. [Online]. Available: https://www.logolynx.com/images/logolynx/1d/1d12643eaacd5b3183eae12407bf4891.jpeg

[103] J. Yiu, *The definitive guide to Arm Cortex-M3 and Cortex-M4 Processors*, 3rd ed. Elsevier Inc., 2014.

[104] "Variable access with Position Independent Executable [image]," last accessed: November 11, 2023. [Online]. Available: https://mcuoneclipse.files.wordpress.com/2021/06/variable-access-without-pic-1.jpg

[105] "Variable access with Global Offset Table [image]," last accessed: November 11, 2023. [Online]. Available: https://mcuoneclipse.files.wordpress.com/2021/06/variable-access-with-got.jpg