

Jasper Lorenz

Mindstorming mit Erika: Portierung eines OSEK/AUTOSAR konformen Betriebssystems auf ARM9

Bachelorarbeit im Fach Informatik

9. September 2022

Please cite as:
Jasper Lorenz, "Mindstorming mit Erika: Portierung eines OSEK/AUTOSAR konformen Betriebssystems auf ARM9" Bachelor's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, September 2022.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

Mindstorming mit Erika: Portierung eines OSEK/AUTOSAR konformen Betriebssystems auf ARM9

Bachelorarbeit im Fach Informatik

vorgelegt von

Jasper Lorenz

geb. am 29. Februar 2000
in Hannover

angefertigt am

**Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur**

**Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover**

Erstprüfer: **Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer: **Prof. Dr.-Ing. Bernardo Wagner**
Betreuer: **Tim-Marek Thomas, M.Sc.**
Gerion Entrup, M.Sc.

Beginn der Arbeit: **9. Mai 2022**
Abgabe der Arbeit: **9. September 2022**

Erklärung

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

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Jasper Lorenz)
Hannover, 9. September 2022

ABSTRACT

LEGO EV3 MINDSTORMS is a modular system to build robots. It comes with an ARM9 based microcontroller and includes a firmware designed for playful programming. However, this firmware is not suited for programming real-time applications. Other works have already provided an implementation of the real-time operating system OSEK/OS, which is, however incomplete and has serious bugs. The goal of this thesis is to port the widespread ErikaOS (an implementation of the OSEK/OS) onto the ARM9 microcontroller. Due to the incomplete documentation of ErikaOS the porting of ErikaOS is based on the analysis of existing ErikaOS ports for other CPUs. Further, pre-existing open source software for the LEGO EV3 MINDSTORMS has been reused for the implementation. To validate the implementation the conformance test cases defined for OSEK/OS have been used. These tests confirm that the implementation of ErikaOS for the LEGO EV3 MINDSTORMS is correct and complete.

KURZFASSUNG

LEGO EV3 MINDSTORMS ist ein modulares System zum Bauen von Robotern. Es besteht aus einem ARM9 basierten Mikrokontroller, einschließlich Firmware, welche eine spielerische Programmierung ermöglicht. Die Firmware ist jedoch nicht zur Programmierung von Echtzeitanwendungen geeignet. Andere Arbeiten bieten bereits eine Implementierung eines OSEK/OS Echtzeitbetriebssystems, welche jedoch unvollständig und fehlerbehaftet ist. Das Ziel dieser Arbeit ist das weitverbreitete ErikaOS (eine Implementierung von OSEK/OS) auf den ARM9 Mikrokontroller zu portieren. Aufgrund der unvollständigen Dokumentation von ErikaOS erfolgt die Portierung von ErikaOS basierend auf der Analyse von existierenden Portierungen von ErikaOS auf andere CPUs. Weiterhin wurde Open Source Software für den LEGO EV3 MINDSTORMS für die Implementierung wiederverwendet. Zur Validierung der Implementierung wurden die OSEK/OS-Konformitätstestfälle genutzt. Diese bestätigen, dass die Implementierung von ErikaOS für den LEGO EV3 MINDSTORMS korrekt und vollständig ist.

INHALTSVERZEICHNIS

Abstract	v
Kurzfassung	vii
1 Einleitung	1
2 Grundlagen	3
2.1 Echtzeitbetriebssysteme	3
2.2 OSEK/AUTOSAR	4
2.3 ErikaOS	14
2.3.1 Verzeichnisstruktur	15
2.3.2 Generierungsprozess	16
2.4 EV3-Baustein	17
2.4.1 AM1808	17
2.4.2 ARM9	18
2.4.3 Bootvorgang	19
2.5 Verwandte Arbeiten	19
2.6 Zusammenfassung	20
3 Umsetzung	21
3.1 Entwicklungsumgebung	21
3.1.1 Kommunikation über UART	21
3.1.2 Kompilierung	23
3.1.3 Binden	23
3.1.4 Deployment	24
3.1.5 Debuggen	25
3.1.6 Reset-Schaltung	27
3.2 Implementierung	28
3.2.1 RT-Druid Anpassungen	29
3.2.2 Starten und Terminieren von Tasks	29
3.2.3 Bootfunktion	30
3.2.4 Präemptives Scheduling	30
3.2.5 Interrupts	30
3.2.5.1 AM1808	31
3.2.5.2 Versatilepb	33
3.2.6 Multistack und Kontextwechsel	34

Inhaltsverzeichnis

3.2.7 EV3 Motoren und Sensoren	35
3.3 Zusammenfassung	35
4 Evaluation	37
4.1 Testanwendungen	37
4.2 OSEK-Konformitätstestfälle	37
4.3 Umsetzung	38
4.4 Ergebnis	39
5 Zusammenfassung	41
6 Anhang	43
Verzeichnisse	46
Abkürzungsverzeichnis	46
Abbildungsverzeichnis	49
Tabellenverzeichnis	51
Codeverzeichnis	53
Literatur	55

1

EINLEITUNG

LEGO EV3 MINDSTORMS [LEG] ist ein System zum Bauen von Robotern mithilfe von LEGO. Kern dieses Systems ist der EV3-Baustein, an welchen Sensoren und Motoren angeschlossen werden können. Der EV3-Baustein basiert auf dem von Texas Instruments entwickelten AM1808 Mikrokontroller. Dieser stellt, neben einer ARM9-CPU, Schnittstellen zur Ansteuerung der Sensoren und Motoren und eine dazugehörige Firmware bereit. Die Firmware eignet sich jedoch nicht, um zeitkritische Anwendungen auf dem EV3-Baustein auszuführen.

Für zeitkritische Anwendungen wird ein Echtzeitbetriebssystem benötigt. Im Rahmen von OSEK/AUTOSAR [gro04a], den Bestrebungen der Automobilindustrie Architektur und Schnittstellen für Steuergeräte in Fahrzeugen zu vereinheitlichen, wurde ein solches Echtzeitbetriebssystem als OSEK/OS spezifiziert.

Ziel dieser Arbeit ist ErikaOS [Gai] auf den EV3-Baustein zu portieren. ErikaOS ist eine Open Source Implementierung eines OSEK/OS konformen Echtzeitbetriebssystems, die bisher keine Unterstützung für ARM9-CPUs bietet.

Der gewählte Standard und die gewählte Zielhardware der Portierung sind auf zwei wesentliche Gründe zurückzuführen. So eignet sich der EV3-Baustein gut zum Entwickeln von Prototypen und es handelt sich bei OSEK um einen Industriestandard. Dadurch kann der EV3-Baustein durch diese Portierung im akademischen Umfeld sinnvoller genutzt werden.

In Kapitel 2 werden die Grundlagen für die Portierung dargelegt. Dazu gehören eine Einführung in OSEK/OS, die Beschreibung des Aufbaus von ErikaOS und die Vorstellung der Hardware des EV3-Bausteins. Ebenso wird ein Überblick über verwandte Arbeiten gegeben. In Kapitel 3 werden die Details der Portierung beschrieben. Es wird auf die Entwicklungsumgebung und die Besonderheiten bei der Entwicklung von Anwendungen für eingebettete Systeme eingegangen. Anschließend werden die Implementierung und die wesentlichen Funktionen wie Starten und Terminieren von *Tasks*, Bootfunktion, präemptives Scheduling, Interrupts und dem Kontextwechsel beschrieben. In Kapitel 4 erfolgt die Evaluation der Portierung basierend auf standardisierten Testfällen. Kapitel 5 fasst diese Arbeit zusammen.

2

GRUNDLAGEN

Im Rahmen dieser Arbeit wurde das OSEK/AUTOSAR konforme Echtzeitbetriebssystem ErikaOS auf den EV3-Baustein, der auf einem ARM9 Prozessor basiert, portiert. In diesem Kapitel werden die Grundlagen, die zum Verständnis des Portierungsprozesses erforderlich sind, vorgestellt. Dazu gehören generelle Betriebssystemkonzepte, der OSEK/AUTOSAR-Standard, sowie grundlegende Informationen zur Hardware des EV3-Bausteins. Außerdem werden verwandte Arbeiten vorgestellt.

2.1 Echtzeitbetriebssysteme

Es gibt keine allgemeingültige Definition des Begriffes Betriebssystem. Allgemein zählt aber die Abstraktion und Verteilung der Hardwareressourcen zu den wesentlichen Aufgaben von Betriebssystemen. Je nach Einsatzfeld lassen sich Betriebssysteme verschiedenen Klassen zuordnen.

Im Rahmen dieser Arbeit geht es um eingebettete Echtzeitbetriebssysteme. Innerhalb dieser existieren gebündelte Aufgabenpakete mit einer zeitlichen Frist, bis wann diese erledigt sein müssen. Diese Aufgabenpakete werden im weiteren Verlauf als *Tasks* bezeichnet. *Tasks* in Echtzeitbetriebssystemen unterscheiden sich zu anderen Betriebssystemen darin, dass die Zeit eine funktionale Eigenschaft darstellt. Das bedeutet, dass das Einhalten der zeitlichen Fristen für die Korrektheit des Systems erforderlich ist [CW19, S. 11]. Man unterscheidet zwischen harter und weicher Echtzeit. Harte Echtzeit ist dadurch geprägt, dass *Tasks* zu einem gewissen Zeitpunkt erledigt sein müssen, während bei weicher Echtzeit ein Verpassen dieses Zeitpunktes nicht erwünscht ist, aber akzeptiert werden kann. Der Unterschied liegt also in den Konsequenzen, wenn eine Frist nicht eingehalten wird. In der harten Echtzeit hat dies schwere Folgen und in der weichen Echtzeit nicht [TBP16, S. 70-71]. Das hier portierte ErikaOS unterstützt harte Echtzeit.

Ein eingebettetes Gerät beschreibt eine Steuereinheit zum Kontrollieren von anderen Geräten. Dabei ist diese Steuereinheit nach außen nicht erkennbar, und der Sicherheitsaspekt ein relevanter Faktor. So darf nur vertrauenswürdige Software auf diesen Steuereinheiten ausgeführt werden und das Ausführen von eigenen neuen Anwendungen ist nicht ohne weiteres möglich. Eingebettete Systeme besitzen häufig eine eher kleinere Rechenleistung [TBP16, S. 1147].

Echtzeitbetriebssysteme für eingebettete Systeme werden aufgrund der geringen Rechenleistung häufig als Bibliothek bereitgestellt. Man spricht in solch einem Fall von einem Bibliotheksbetriebs-

2.1 Echtzeitbetriebssysteme

system [TBP16, S. 70]. Dies ist auch für das in dieser Arbeit portierte ErikaOS der Fall.

Damit Fristen eingehalten werden können, ist es häufig nötig im laufenden Betrieb von einer *Task* zu einer anderen *Task* zu wechseln.

Dies wird deutlich, wenn man folgende Situation betrachtet: Das System arbeitet an einer *Task T1* ohne Frist. Nun wird eine weitere *Task T2* mit Frist lauffähig. *Task T1* ist so umfangreich, dass das vollständige Durchlaufen dieser ein Verpassen der Frist von *Task T2* bedeutet. Daher muss *Task T1* im laufenden Betrieb die Central Processing Unit (CPU) an *Task T2* abgeben.

Wann und zu welcher *Task* so ein Wechsel stattfindet, entscheidet der sogenannte Scheduler. Wie genau dieser agiert, hängt von der gewählten Scheduler-Strategie ab. Scheduler lassen sich im Wesentlichen in nicht-präemptive und präemptive Scheduler unterteilen. Bei Ersteren muss sich eine *Task* aktiv dazu entscheiden, die CPU freizugeben. Bei Letzterem kann einer *Task* nach jeder Instruktion die CPU entzogen werden. Häufig werden *Tasks* in Echtzeitbetriebssystemen Prioritäten zugewiesen, anhand welcher der Scheduler entscheiden kann, welche *Task* er bevorzugen soll [TBP16, S. 192ff]. Dies ist auch für das in dieser Arbeit portierte ErikaOS der Fall.

Beim Verwalten von Ressourcen durch das Betriebssystem unterscheidet man zwischen unterbrechbaren und nicht unterbrechbaren Ressourcen. Unterbrechbare Ressourcen können ihrem Besitzer jederzeit ohne negativen Effekt entzogen werden. Nicht unterbrechbare Ressourcen hingegen können ihrem Besitzer nicht entzogen werden [TBP16, S. 513ff].

Ein wichtiges Konzept und häufiges Problem im Zusammenhang mit nicht unterbrechbaren Ressourcen sind Deadlocks. Dafür gilt es zunächst den Ablauf der Benutzung von nicht unterbrechbaren Ressourcen nachzuvollziehen. Dieser besteht aus drei Schritten: Als Erstes der Anforderung der Ressource. Sollte diese belegt sein, blockiert die anfordernde *Task*. Als Zweites das Erhalten und Nutzen der Ressource. Und abschließend die Freigabe der Ressource [TBP16, S. 513ff].

Ein Deadlock entsteht, wenn mindestens zwei unabhängige *Tasks* wechselseitig auf die Freigabe einer Ressource warten. Dies wird deutlich, wenn man folgende Situation betrachtet: Ein System besteht aus zwei *Tasks* (*T1, T2*). *T1* fordert erst Ressource A und dann Ressource B an, während *T2* erst Ressource B und dann Ressource A anfordert. Entsteht nun die Situation, dass beide *Tasks* (*T1, T2*) jeweils ihre erste Ressource angefordert haben, so entsteht ein Deadlock. *T1* belegt Ressource A und wartet auf die Freigabe von Ressource B, während *T2* Ressource B belegt und auf die Freigabe von Ressource A wartet [TBP16, S.515].

Ein mit dem Deadlock verwandtes Problem ist die Prioritätsinversion. Dabei verhindert eine *Task* mit niedriger Priorität das Ausführen einer *Task* mit höherer Priorität durch das Belegen einer Ressource [TBP16, S. 170].

2.2 OSEK/AUTOSAR

Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) steht für die um 1993 initiierten Bestrebungen der Automobilindustrie Architektur und Schnittstellen für Steuergeräte in Fahrzeugen zu standardisieren [gro04a]. 2005 wurde OSEK von der ISO in sechs ISO-Standards aufgenommen (ISO 17356-1 bis ISO 17356-6) [Oseb]. Seit 2003 werden diese Bestrebungen im

Rahmen von Autosar [ATU] (Automotive Open System Architecture) fortgesetzt. Für diese Arbeit relevant ist das von OSEK standardisierte Echtzeitbetriebssystem OSEK/OS beziehungsweise seine offene Implementierung ErikaOS.

Ziel dieser Standardisierung ist die Portabilität von Anwendungen. Dazu wurde die sogenannte **OSEK Implementation Language** (OIL), sowie das OSEK/OS-Interface definiert. Eine OIL-Datei enthält Informationen über das genutzte System, deren Ressourcen und die Anwendung. Das OSEK/OS-Interface besteht aus *System Services* und weiteren Konzepten wie *Tasks*, *Interruptbehandlungen*, *Eventmechanismen*, *Ressourcenmanagement*, *Alarmen* und *Counter*, sowie *Hookfunktionen* [gro04a]. *System Services* werden im weiteren Verlauf als Systemaufrufe bezeichnet.

Im Folgenden werden die Konzepte von OSEK/OS anhand von Beispielen¹ erläutert.

Tasks

Eine Echtzeitanwendung unter OSEK/OS unterteilt sich in *Tasks*, welche individuelle Echtzeitanforderungen, wie zum Beispiel Priorität oder Scheduler-Strategie besitzen. Auf Implementierungsebene sind *Tasks* Funktionen, ergänzt um Laufzeitinformationen. Anders als C-Funktionen bieten *Tasks* keine Übergabeparameter. *Tasks* werden über eine ID identifiziert [gro05].

OSEK/OS unterscheidet zwischen zwei Typen von Tasks. *Basic Tasks* und *Extended Tasks*. Diese unterscheiden sich darin, wann sie die CPU abgeben. *Basic Tasks* tun dies nur, wenn sie terminieren, eine Task mit höherer Priorität bereit ist und der Scheduler aktiv wird oder ein Interrupt ihnen die CPU entzieht. *Extended Tasks* können die CPU zusätzlich noch an *Tasks* mit einer niedrigeren Priorität abgeben, wenn sie in den Zustand *Waiting* wechseln [gro05].

Allgemein kann sich eine *Task* in einem von den folgenden Zuständen befinden [gro05].

Running Die *Task* nutzt gerade die CPU.

Ready Die *Task* ist bereit die CPU zu übernehmen, beansprucht diese aber gerade noch nicht.

Suspended Die *Task* ist noch nicht aktiviert oder bereits terminiert worden.

Waiting Die *Task* wurde explizit durch Systemaufruf *WaitEvent* in diesen Zustand versetzt. Dieser Zustand kann nur von *Extended Tasks* eingenommen werden.

Neben ihrem Zustand besitzen *Tasks* noch statisch zugewiesene Eigenschaften. Diese werden in der OIL-Datei angegeben und auch zur Laufzeit für eine bessere Performance nicht mehr modifiziert. Zum Beispiel kann die Priorität oder die maximale Anzahl der zeitgleichen Aktivierungen festgelegt werden [gro05].

Nach dem Starten des Betriebssystems befinden sich alle *Tasks* im Zustand *Suspended*. Damit eine *Task* gestartet wird, müssen zwei Zustandswechsel erfolgen. Als Erstes muss eine *Task* durch den Systemaufruf *ActivateTask* oder den Systemaufruf *ChainTask* von dem *Suspended* in den *Ready* Zustand versetzt werden. Dies setzt voraus, dass die *Task* nicht schon öfters zeitgleich aktiviert wurde als erlaubt. Dieses Limit wird in der OIL-Datei jeder *Task* zugewiesen. Als Nächstes muss die *Task* die CPU erhalten. Wann dies geschieht, entscheidet der Scheduler [gro05].

Wie und wann der Scheduler aktiv wird, hängt von der Scheduler-Strategie ab. Diese ergibt sich

¹Alle Anwendungsbeispiele sind ebenfalls im praktischen Teil im Verzeichnis *erika/thesis_example* zu finden.

2.2 OSEK/AUTOSAR

über das für alle *Task* in der OIL individuell festgelegte, „SCHEDULE“ Attribut. Daraus können drei Scheduler-Strategien entstehen [Gai].

preemptive Alle *Tasks* sind präemptiv

non-preemptive Keine *Task* ist präemptiv

mixed-preemptive Es existieren sowohl präemptive als auch nicht-präemptive *Tasks*

Der Scheduler wird bei präemptiven *Tasks* am Ende von bestimmten Systemaufrufen oder beim Verlassen einer Interruptserviceroutine aktiv. Bei nicht-präemptiven *Tasks* ist dies nicht der Fall und der Scheduler muss durch den Systemaufruf Schedule selbstständig aufgerufen werden [gro05]. OSEK/OS spezifiziert kein zeitbasiertes Aktivieren des Schedulers, wobei in regelmäßigen Intervallen der Scheduler aufgerufen wird. Erreicht eine *Task* ihr Ende, so muss diese sich selbst terminieren. Dies erfolgt durch den Systemaufruf TerminateTask oder ChainTask. Wichtig hierbei ist, dass *Tasks* sich nur selbstständig terminieren können, aber auch müssen. ChainTask terminiert die laufende *Task* und garantiert, dass eine weitere *Task* lauffähig ist, zu der gewechselt werden kann. Endet eine *Task* ohne den Aufruf einer dieser Systemaufrufe so ist das Verhalten undefiniert [gro05].

Der Beispielcode aus Listing 2.1 zeigt eine minimale OSEK/OS Anwendung mit nur einer *Task*, die den Lebenszyklus einer *Task* verdeutlichen soll. Die Anwendung startet „Task1“ mit ActivateTask (Zeile 15). Diese terminiert und startet sich selbst zehnmal neu über den Systemaufruf ChainTask (Zeile 9). Wurden diese zehn Aktivierungen überschritten, terminiert die *Task* endgültig (Zeile 11). In der OIL werden statisch die Priorität, Scheduler-Strategie und Aktivierungsanzahl für die *Task* definiert. Anzumerken ist, dass durch das Neustarten der *Task* mit ChainTask die maximale Anzahl der erlaubten Aktivierungen nicht überschritten wird, da ChainTask die laufende *Task* zunächst terminiert und die übergebene *Task* erst im Anschluss startet.

```
1 #include "ee.h" // ErikaOS Header          1 CPU test_application {
2 unsigned int task1_started = 0;              2   OS EE {
3 TASK(Task1) {                                3     CPU_DATA = <genutzte CPU> {
4   task1_started++;                           4       // Codedateien der Anwendung
5   if (task1_started <= 10) {                  5     APP_SRC = "main.c";
6     // Gib Anzahl Aktivierung+'\n' aus      6   };
7     put_uint(task1_started); __nl;           7     // Konformitätsklasse
8     // Terminiere und Starte Task1 neu      8     KERNEL_TYPE = BCC1;
9     ChainTask(Task1);                      9   };
10 } else { // Terminiere endgültig            10  TASK Task1 {
11   TerminateTask();                         11    PRIORITY = 0x01;
12 }                                         12    // Anzahl erlaubter Aktivierungen
13 }                                         13    ACTIVATION = 1;
14 int main(void) {                           14    // Scheduler ist preemptive
15   ActivateTask(Task1);                   15    SCHEDULE = FULL;
16   return 0;                               16  };
17 }
```

((a)) main.c

((b)) conf.oil

Listing 2.1 – OSEK/OS Beispianwendung Minitask

Konformitätsklassen

OSEK/OS wird in vier unterschiedliche Konformitätsklassen unterteilt. Je nachdem, mit welchem Umfang OSEK/OS genutzt wird, entspricht das resultierende Betriebssystem dieser Konformitätsklasse [gro05].

BCC1 Es werden nur *Basic Tasks* genutzt. Jede Priorität darf nur einmalig genutzt werden und *Tasks* können zeitgleich maximal einmal aktiviert werden.

BCC2 Es werden nur *Basic Tasks* genutzt. Jede Priorität kann mehrfach genutzt werden und alle *Tasks* können zeitgleich mehr als einmal aktiviert werden.

ECC1 Entspricht BCC1, wobei *Extended Tasks* genutzt werden dürfen.

ECC2 Entspricht BCC2, wobei *Extended Tasks* genutzt werden dürfen.

Da ErikaOS ein Bibliotheksbetriebssystem ist, werden die Konformitätsklassen so genutzt, dass nur die Teile der Bibliothek übersetzt werden, die benötigt werden. Ebenso ermöglichen die Konformitätsklassen, dass ein OSEK konformes Betriebssystem nicht den gesamten Standard implementieren muss.

Interruptbehandlung

OSEK/OS kategorisiert Interrupts in zwei unterschiedliche Kategorien. Die erste Kategorie *Interrupt Kategorie 1* zeichnet sich durch zwei Eigenschaften aus. Als Erstes, dass keine Systemaufrufe in der Interruptserviceroutine genutzt werden dürfen und als Zweites, dass nach Abschluss der Interruptserviceroutine zum selben Kontext gewechselt wird, welcher unterbrochen wurde. Demnach wird nach einem Interrupt der Kategorie eins der Scheduler nicht aktiviert.

Die zweite Kategorie *Interrupt Kategorie 2*, erlaubt das Aufrufen von Systemaufrufen. Ebenso wird der Scheduler vor dem Wechseln zum unterbrochenen Kontext aktiviert. Somit ist es möglich, dass im Anschluss an die Interruptserviceroutine zu einem neuen Kontext gewechselt wird. Interrupts gehören zu allen Konformitätsklassen dazu [gro05].

Interrupts und ihren Routinen werden genau wie *Tasks* über die OIL-Datei statische Eigenschaften zugewiesen. Dazu zählen, welcher Kategorie sie angehören, was Ihre Hardwarequelle ist, aber auch welche Priorität sie besitzen. Die Interruptserviceroutine wird ähnlich, wie eine *Task* als Funktion implementiert. Jedoch wird das Schlüsselwort „ISR1“ oder „ISR2“ statt „TASK“ genutzt [gro05].

Der Beispielcode aus Listing 2.2 zeigt eine minimale OSEK/OS Anwendung mit Interruptbehandlung. Die Anwendung startet „Task1“, welche in einer „While-Schleife“ verbleibt, bis die Variable *isr_fired* auf eins gesetzt wird (Zeile 8). Diese Variable wird ausschließlich von der Interruptserviceroutine auf eins gesetzt (Zeile 4). Somit terminiert die Anwendung nur, wenn der Interrupt ausgelöst wurde. In der OIL werden statisch die Kategorie, Hardwarequelle und Priorität für die Interruptserviceroutine definiert [gro05].

2.2 OSEK/AUTOSAR

```
1 CPU test_application {
2   OS EE {
3     CPU_DATA = <genutzte CPU> {
4       APP_SRC = "main.c";
5     };
6     KERNEL_TYPE = BCC1;
7   };
8   TASK Task1 {
9     PRIORITY = 0x01;
10    ACTIVATION = 1;
11    SCHEDULE = FULL;
12  };
13  // Interruptserviceroutinenname
14  ISR timer_handler {
15    // Kategorie eins oder zwei
16    CATEGORY = 2;
17    // Hardwarequelle des Interrupt
18    ENTRY = "TIMER";
19    // Priorität des Interrupt
20    PRIORITY = 1;
21  };
22};
```

((a)) main.c

((b)) conf.oil

Listing 2.2 – OSEK/OS Beispielanwendung Interrupt

Eventmechanismen

OSEK/OS bietet das Konzept des *Events*. Diese können nur mit *Extended Tasks* und somit in den Konformitätsklassen *ECC1* und *ECC2* genutzt werden. Sie dienen der Synchronisation und dem Zulassen von komplexeren Programmflüssen. Über die OIL-Datei wird angegeben, welche *Task* welches *Event* empfangen kann. Eine solche *Task* wird als Besitzer dieses *Events* bezeichnet. Ein *Event* wird über seine ID und die zugehörige *Task* eindeutig identifiziert. Mehrere *Tasks* dürfen dasselbe *Event* besitzen.

Ein Event kann mittels des Systemaufrufs *SetEvent* synchron gesetzt werden. Alternativ kann es auch asynchron, zum Beispiel durch einen abgelaufenen Alarm, (siehe 2.2) gesetzt werden. Das Setzen eines *Events* kann auch von einer *Basic Task* erfolgen [gro05].

Soll eine *Extended Task* auf ein Event warten, muss diese den Systemaufruf *WaitEvent* nutzen, welcher sie in den Zustand *Waiting* versetzt und den Scheduler aktiviert. So kann an dieser Stelle zu *Tasks* mit einer niedrigeren Priorität gewechselt werden. Wird im weiteren Verlauf das *Event* gesetzt, wechselt die *Extended Task* vom Zustand *Waiting* in den Zustand *Ready* [gro05].

Der Beispielcode aus Listing 2.3 zeigt eine minimale OSEK/OS Anwendung mit Eventmechanismus. Die Anwendung startet „Task2“ (Zeile 28), welche nach dem Aktivieren einer *Task* mit niedrigerer Priorität (Zeile 10) direkt auf das Setzen von einem *Event* wartet (Zeile 14). Da „Task2“ *WaitEvent* aufgerufen hat und „Task1“ sich im Zustand *Ready* befindet, startet der Scheduler „Task1“. Diese setzt das *Event* „MyEvent“, wodurch der Scheduler zurück zu „Task2“ wechselt, welche vom Zustand

Waiting in den Zustand *Ready* beziehungsweise direkt *Running* versetzt wird. Dies passiert, da sie die *Task* mit der höchsten Priorität ist. Wäre dies nicht der Fall, würde sie nur in den Zustand *Ready* versetzt werden. Anschließend prüft „Task2“, ob es sich bei dem gesetzten *Event* um das „MyEvent“ Event handelt (Zeile 18) und terminiert (Zeile 23). „Task1“ befindet sich noch im Zustand *Ready*, da sie nicht terminiert wurde. Der Scheduler wurde mit dem Aufruf von SetEvent aktiviert und daher wurde zu „Task2“ gewechselt, welche terminiert und somit die Anwendung beendet.

In der OIL wird das *Event* „MyEvent“ deklariert und „Task2“ zugewiesen. Da nur „Task2“ ein Event zugeordnet wird, ist folglich „Task1“ eine *Basic* und „Task2“ eine *Extended Task*.

<pre> 1 #include "ee.h" // ErikaOS Header 2 TASK(Task1) { 3 // Setzt Event für Task2 4 SetEvent(Task2, MyEvent) 5 // Terminiere 6 TerminateTask(); 7 } 8 TASK(Task2) { 9 // Setzte Task1 in Zustand Ready 10 ActivateTask(Task1); 11 // Warte bis Event korrekt gesetzt 12 while (1) { 13 // Wechsel zu Task1 14 WaitEvent(MyEvent); 15 EventMaskType mask; 16 // Prüfe ob Event gesetzt 17 GetEvent(Task2, &mask); 18 if (mask & MyEvent) { 19 // Setze Event auf nicht gesetzt 20 ClearEvent(MyEvent); 21 put_string("Finished!\n"); 22 // Terminiere 23 TerminateTask(); 24 } 25 } 26 } 27 int main(void) { 28 ActivateTask(Task2); 29 return 0; 30 }</pre>	<pre> 1 CPU test_application { 2 OS EE { 3 CPU_DATA = <genutzte CPU> { 4 APP_SRC = "main.c"; 5 }; 6 KERNEL_TYPE = ECC1; 7 }; 8 TASK Task1 { 9 PRIORITY = 0x01; 10 ACTIVATION = 1; 11 SCHEDULE = FULL; 12 }; 13 TASK Task2 { 14 PRIORITY = 0x02; 15 ACTIVATION = 1; 16 SCHEDULE = FULL; 17 // Zugehörigkeit des Event 18 EVENT = MyEvent; 19 }; 20 // Definiere Event mit Eventmaske 21 EVENT MyEvent { MASK = AUTO; }; 22 }</pre>
--	---

((a)) main.c

((b)) conf.foil

Listing 2.3 – OSEK/OS Beispielanwendung Event

2.2 OSEK/AUTOSAR

Ressourcenmanagement

Neben *Events*, welche der Synchronisation des Programmflusses dienen, definiert OSEK/OS *Ressourcen*, welche genutzt werden können, um neben dem Programmfluss auch unter *Tasks* geteilte Komponenten des Systems zu synchronisieren. Dazu zählt zum Beispiel der Scheduler. Ressourcen gehören zu allen Konformitätsklassen [gro05].

Eine *Ressource* wird durch den Systemaufruf `GetResource` belegt. Dieser Aufruf darf nur aus dem Kontext einer *Task* erfolgen. Solange diese *Ressource* belegt ist, können andere *Tasks* nicht auf diese zugreifen. Nur die *Task*, welche die *Ressource* belegt hat, kann diese auch wieder freigeben. Dies geschieht durch den Systemaufruf `ReleaseResource` [gro05].

OSEK/OS garantiert folgende Eigenschaften beim Belegen von *Ressourcen*. Zwei unterschiedliche *Tasks* können niemals dieselbe *Ressource* zeitgleich belegen. Durch das Belegen von *Ressourcen* kann es nicht zur Prioritätsinversion oder Deadlocks kommen [gro05].

Soll eine *Ressource* dafür genutzt werden, den Programmfluss zu kontrollieren, so kann man den Scheduler als *Ressource* belegen und somit dafür sorgen, dass dieser nicht aktiv wird. Dies verhindert, dass einer *Task* durch den Scheduler die CPU entzogen wird, nicht jedoch durch Interrupts. Der Scheduler wird mit dem Systemaufruf `GetResource` belegt, wobei die vordefinierte *Ressource* des Schedulers „RES_SCHEDULER“, als *Ressource* angegeben wird [gro05].

Da der Scheduler nach OSEK/OS, wenn er aktiv wird, immer die *Task* mit der höchsten Priorität im Zustand *Ready* aktiviert, und es *Ressourcen* gibt, welche belegt werden können, könnte es zur Prioritätsinversion kommen. Damit dies nicht eintritt, nutzt OSEK/OS das „OSEK Priority Ceiling Protocol“ (OSEK Prioritätsslimitprotokoll). Dieses verhindert das Eintreten von Deadlocks und Prioritätsinversion indem die Priorität einer *Task* temporär auf die Priorität der *Task* angehoben wird, welche die höchste Priorität besitzt und auf die *Ressource* zugreifen kann.

Der Beispielcode aus Listing 2.4 zeigt eine minimale OSEK/OS Anwendung mit Ressourcenmanagement. Die Anwendung startet „Task1“, welche die *Ressource* des Schedulers belegt und somit dessen Aktivierung verhindert (Zeile 5). Als Nächstes wird „Task2“, welche eine höhere Priorität als „Task1“ besitzt, aktiviert (Zeile 7). Da beide *Tasks* präemptiv sind, würde an dieser Stelle normalerweise ein Kontextwechsel zu „Task2“ erfolgen. Dies ist nicht der Fall, da der Scheduler belegt ist. Folgend wird geprüft, ob kein Kontextwechsel passierte und wenn dies der Fall ist der Scheduler wieder freigegeben, wodurch „Task2“ die CPU erhält (Zeile 15). Beide *Tasks* terminieren im Anschluss. In der OIL wird der Scheduler als *Ressource* deklariert und „Task2“ eine höhere Priorität als „Task1“ zugewiesen.

```

1 #include "ee.h" // ErikaOS Header
2 unsigned int task2_started = 0;
3 TASK(Task1) {
4     // Sperre den Scheduler
5     GetResource(RES_SCHEDULER);
6     // Aktiviere (ohne Wechsel) Task2
7     ActivateTask(Task2);
8     // Prüfe ob Task2 gelaufen ist
9     while(task2_started) {
10         // NOP
11         (void) 0;
12     }
13     // Gebe den Scheduler frei
14     // Wechsel zu Task2
15     ReleaseResource(RES_SCHEDULER);
16     TerminateTask();
17 }
18 TASK(Task2) {
19     task2_started = 1;
20     put_string("Finished!\n");
21     TerminateTask();
22 }
23 int main(void) {
24     ActivateTask(Task1);
25     return 0;
26 }
```

((a)) main.c

```

1 CPU test_application {
2     OS EE {
3         CPU_DATA = <genutzte CPU> {
4             APP_SRC = "main.c";
5         };
6         KERNEL_TYPE = BCC1;
7         // Nutze Scheduler als Ressource
8         USERESSCHEDULER = TRUE;
9     };
10    TASK Task1 {
11        PRIORITY = 0x01;
12        ACTIVATION = 1;
13        SCHEDULE = FULL;
14    };
15    TASK Task2 {
16        PRIORITY = 0x02;
17        ACTIVATION = 1;
18        SCHEDULE = FULL;
19    };
20 }
```

((b)) conf.oil

Listing 2.4 – OSEK/OS Beispielanwendung Ressource

Alarme und Counter

OSEK spezifiziert *Alarme* und *Counter*. *Counter* können rein in Software umgesetzt oder auch in Hardware realisiert werden. Ein *Counter* wird über seinen Maximalwert und die Anzahl von Ticks, die erforderlich sind um den *Counter* um eins zu erhöhen, definiert. Bei Softwarezählern werden Ticks durch den Systemaufruf IncrementCounter ausgelöst. Die Eigenschaften eines *Counters* werden statisch über die OIL-Datei spezifiziert [gro05].

Basierend auf *Countern* lassen sich *Alarme* definieren. Dazu muss dem *Alarm* ein *Counter*, sowie eine Aktion, welche beim Ablauen des *Alarms* ausgelöst wird, zugeordnet werden. Ein *Alarm* läuft ab, wenn der dazugehörige *Counter* einen gewissen Wert erreicht. Dieser Wert wird beim Aktivieren des *Alarms* absolut oder relativ festgelegt. Beim Aktivieren mittels des Systemaufrufs SetAbsAlarm wird ein absoluter und mittels des Systemaufrufs SetRelAlarm ein relativer Alarm gesetzt. Beim Auslösen des *Alarms* kann entweder eine vordefinierte Funktion aufgerufen, eine *Task* aktiviert, ein *Event* gesetzt oder ein *Counter* inkrementiert werden. [gro05].

Der Beispielcode aus Listing 2.5 zeigt eine minimale OSEK/OS Anwendung mit einem absoluten *Alarm* der ein Event setzt. Die Anwendung startet „Task2“, welche „Task1“ aktiviert und dann auf das „AlarmEvent“ wartet (Zeile 18). Daher startet „Task1“, welche einen absoluten *Alarm* startet (Zeile 8), der auslöst, wenn der *Counter* den Wert eins erreicht. Anschließend erhöht „Task1“ den

2.2 OSEK/AUTOSAR

Counter mit dem Systemaufruf CounterTick um einen „Tick“, wodurch der *Alarm* ausgelöst und somit das Event „*AlarmEvent*“ gesetzt wird (Zeile 10). Es wird wieder zu „*Task2*“ gewechselt, welche prüft, ob das Event „*AlarmEvent*“ gesetzt wurde (Zeile 21). Nur wenn dies der Fall ist, terminieren beide *Tasks* nacheinander.

```
1 CPU test_application {
2   OS EE {
3     CPU_DATA = <genutzte CPU> {
4       APP_SRC = "main.c";
5     };
6     KERNEL_TYPE = ECC1;
7   };
8
9   TASK Task1 {
10    PRIORITY = 0x01;
11    ACTIVATION = 1;
12    SCHEDULE = FULL;
13  };
14
15   TASK Task2 {
16    PRIORITY = 0x02;
17    ACTIVATION = 1;
18    SCHEDULE = FULL;
19    // Zugehörigkeit des Event
20    EVENT = AlarmEvent;
21  };
22
23   COUNTER MyCounter {
24    // Starte neu, wenn Maximalwert \
25    // erreicht
26    MINCYCLE = 1;
27    // 4095 Ticks ist der Maximalwert
28    MAXALLOWEDVALUE = 0xFFFF;
29    // Anzahl Ticks für ein Inkrement
30    TICKSPERBASE = 1;
31  };
32
33   ALARM MyAlarm {
34    // Weise dem Alarm den Zähler zu
35    COUNTER = MyCounter;
36    // Setzte "AlarmEvent" für Task2
37    ACTION = SETEVENT {
38      TASK = Task2; EVENT = AlarmEvent;
39    };
40  };
41
42  EVENT AlarmEvent { MASK = AUTO; };
43};
```

((a)) main.c

((b)) conf.oi

Listing 2.5 – OSEK/OS Beispielanwendung Alarm

Hookfunktionen

OSEK/OS erlaubt mittels sogenannter *Hookfunktionen* beim Erreichen bestimmter Zustände oder beim Eintreten bestimmter Ereignisse vordefinierte Funktionen aufzurufen. Es gibt fünf solcher *Hookfunktionen*. Die *Startup- und Shutdown-Hookfunktion*, welche zu Systemstart beziehungsweise Systemende aufgerufen werden, die *Error-Hookfunktion*, die aufgerufen wird, sollte ein Systemfehler auftreten und die *Pretask- und Posttask-Hookfunktion*, welche zu Beginn und Ende einer *Task* aufgerufen werden. Sollen *Hookfunktionen* genutzt werden, so muss dies in der OIL angegeben werden [gro05].

Der Beispielcode aus Listing 2.6 zeigt eine minimale OSEK/OS Anwendung, bei der alle Hookfunktionen einmal durchlaufen werden. Die Anwendung startet und die *Startup-Hookfunktionen* wird ausgeführt (Zeile 3). Bevor „Task1“ startet, wird die *Pretask-Hookfunktionen* ausgeführt (Zeile 6). „Task1“ provoziert einen Systemfehler, indem versucht wird, eine *Task* zu aktivieren, die nicht existiert (Zeile 19). Dadurch wird die *Error-Hookfunktion* ausgelöst (Zeile 9). Nach der Terminierung wird die *Posttask-Hookfunktion* ausgeführt (Zeile 12).

Der Systemaufruf *ShutdownOS*, erwartet einen Fehlercode, welcher hier willkürlich ausgewählt wurde. Vor dem Beenden (Zeile 23) des Betriebssystems wird noch die *Shutdown-Hookfunktion* ausgeführt (Zeile 15).

```

1 #include "ee.h" // ErikaOS Header
2 #define INVALID_TASK_ID 1234
3 void StartupHook(void) {
4     put_string("StartupHook\n");
5 }
6 void PreTaskHook(void) {
7     put_string("PreTaskHook\n");
8 }
9 void ErrorHook(StatusType Error) {
10    put_string("ErrorHook\n");
11 }
12 void PostTaskHook(void) {
13    put_string("PostTaskHook\n");
14 }
15 void ShutdownHook(StatusType Error) {
16    put_string("ShutdownHook\n");
17 }
18 TASK(Task1) {
19     ActivateTask(INVALID_TASK_ID)
20 }
21 int main(void) {
22     ActivateTask(Task1);
23     ShutdownOS(E_OS_ID);
24     return 0;
25 }
```

```

1 CPU test_application {
2     OS EE {
3         CPU_DATA = <genutzte CPU> {
4             APP_SRC = "main.c";
5         };
6         KERNEL_TYPE = BCC1;
7         // Hookfunktionen sollen genutzt \
8         // werden (hier alle)
9         STARTUPHOOK = TRUE;
10        ERRORHOOK = TRUE;
11        SHUTDOWNHOOK = TRUE;
12        PRETASKHOOK = TRUE;
13        POSTTASKHOOK = TRUE;
14    };
15    TASK Task1 {
16        PRIORITY = 0x01;
17        ACTIVATION = 1;
18        SCHEDULE = FULL;
19    };
}
```

(a) main.c

(b) conf.foil

Listing 2.6 – OSEK/OS Beispieldokumentation Hookfunktionen

2.2 OSEK/AUTOSAR

OSEK Implementierungssprache

Die OIL wird genutzt, um die vom Anwender benötigten Betriebssystemkomponenten und ihre statischen Eigenschaften zu spezifizieren.

Die Syntax ist an die von C++ angelehnt. Dies gilt sowohl für Kommentare, Einbindungen als auch den eigentlichen Inhalt. Die Grundidee der OIL besteht darin, sogenannte OIL-Objekte zu deklarieren und zu beschreiben. Jedes Objekt hat einen eindeutigen Namen, welcher Groß- und Kleinschreibung beachtet und dem ein Wert zugewiesen wird. Dieser Wert kann sich aus gebündelten Attributen zusammensetzen. Jedes Attribut eines Objektes hat einen Standardwert, welchen das Attribut erhält, sollte kein expliziter Wert gegeben werden. Objekte können Referenzen zu anderen Objekten beinhalten. Ein Beispiel dafür wäre eine *Task*, welche ein *Event* besitzt (siehe Listing 2.7). Literale sind somit entweder ein Name zum Identifizieren eines Objekts oder Attributs oder ein String beziehungsweise eine Nummer, um einem Objekt oder einem Attribut einen Wert zuzuweisen [gro04b].

Um das Ziel der Portabilität nicht einzuschränken, dürfen implementierungsspezifische Parameter nur für optionale Attribute genutzt werden [gro04b].

Damit die Informationen aus der OIL auch im laufenden System genutzt werden können, muss ein OIL-Kompilierer bereitgestellt werden. Wie genau dieser zu realisieren ist, wird von OSEK nicht spezifiziert [gro04b].

```
1      [...]
2      TASK <TaskName> {
3          [...]
4          // Referenz zu anderem Objekt <EventName>
5          EVENT = <EventName>;
6      };
7
8      EVENT <EventName> { MASK = AUTO; };
```

Listing 2.7 – OIL Beispiel: Objektreferenzen

2.3 ErikaOS

ErikaOS ist ein Open Source OSEK/OS konformes Echtzeitbetriebssystem [Gai]. Es wurde im universitären Umfeld entwickelt als ErikaOS Version 2 [Eria] und später von der Firma Evidence [Evi21] als ErikaOS Version 3 fortgeführt. Ziel von ErikaOS ist ein leichtgewichtiges, skalierbares, OSEK konformes Echtzeitbetriebssystem für eingebettete Systeme bereitzustellen. Es existieren Portierungen für kleine 8 Bit Mikrocontroller (Atmel/AVR8), aber auch größere Mikrocontroller werden unterstützt (STM32/Cortex-M4) [Gai].

Um die Skalierbarkeit zu erhöhen, lässt sich ErikaOS als Monostack oder als Multistack konfigurieren. Bei der Monostackvariante, teilen sich alle *Tasks* einen Stack und bei der Multistackvariante kann *Tasks* ein individueller Stack zugewiesen werden. [Evi14].

Neben den von OSEK definierten Konformitätsklassen *BCC1*, *BCC2*, *ECC1* und *ECC2* implementiert ErikaOS noch weitere eigene Konformitätsklassen. Diese unterscheiden sich bezüglich der angewendeten Scheduler-Strategie. So können auch „Fixed Priority Scheduling“, „Immediate Priority Ceiling“ und „Earliest Deadline First“ als Scheduler-Strategie genutzt werden [Ava+15].

Zu ErikaOS gehört auch der OIL-Kompilierer *RT-Druid*. Dieser ist nur für ErikaOS Version 2 Open Source. Dies ist der Grund, warum im Rahmen dieser Arbeit ErikaOS Version 2 portiert wurde. *RT-Druid* nimmt als Eingabe eine OIL-Datei und erzeugt daraus von ErikaOS benötigte Konfigurationsdateien. RT-Druid kann als Kommandozeilenanwendung oder Plugin für Eclipse [Fou] genutzt werden [Gai].

2.3.1 Verzeichnisstruktur

ErikaOS ist über ein öffentliches SVN-Repository [Erib] erhältlich. Darin ist das Verzeichnis *ee/pkg* enthalten. Dieses enthält alle für die Entwicklung von ErikaOS-Anwendungen relevanten Dateien. Die Verzeichnisstruktur ist in Abbildung 2.1 dargestellt. Im Verzeichnis *kernel* stellt ErikaOS alle hardwareunabhängigen Codedateien des Betriebssystems (ErikaOS-Kern) bereit. Die hardwareabhängigen Codedateien sind in den Verzeichnissen *board*, *mcu* und *cpu* zu finden. Die Struktur dient der Wiederverwendbarkeit. So wird zum Beispiel CPU spezifischer Code im Verzeichnis *avr8* sowohl für die ATtiny Microcontroller Unit (MCU), als auch die ATmega MCU genutzt. Ähnlich wird MCU spezifischer Code für verschiedene Boards mit gleicher MCU wiederverwendet.

Jedes dieser Verzeichnisse enthält das Unterverzeichnis *common*. Die Verzeichnisse *board/common* und *mcu/common* sind für diese Arbeit nicht weiter relevant. Dort sind zum Beispiel generische Hardwaretreiber für Peripheriebausteine implementiert. Das Verzeichnis *cpu/common* enthält Implementierungsvorlagen in Form von Code und Pseudocode, welche als Grundlage für eine Portierung genutzt werden können. Teilweise können Funktionen aus *common* unverändert übernommen werden. Für viele ist jedoch eine Anpassung erforderlich. Dies gilt insbesondere für Funktionen mit dem Prefix „EE_hal“. Eine vollständige Spezifikation, welche Funktionen ErikaOS von einer Portierung bereitgestellt werden müssen, existiert jedoch nicht.

Das Verzeichnis *cfg* enthält die Makefiles zur Generierung einer ErikaOS-Anwendung. Die Generierung ist hierarchisch organisiert. So werden Definitionen aus den spezifischen Verzeichnissen für CPU, MCU und Board automatisch eingebunden. Das Verzeichnis *cfg/arch* enthält CPU spezifische Makefiles zum Konfigurieren der Generierwerkzeuge (zum Beispiel Kompilierer) und die Makefileregeln zum Erzeugen der Anwendung.

2.3 ErikaOS

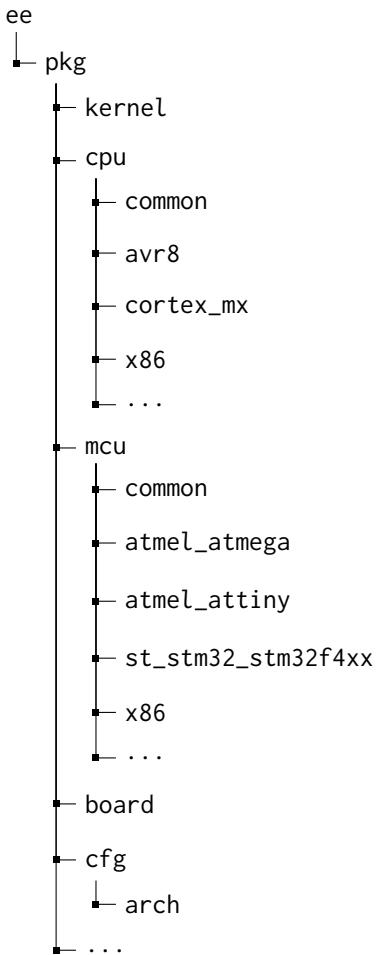


Abbildung 2.1 – ErikaOS: Verzeichnisstruktur

2.3.2 Generierungsprozess

Der Generierungsprozess einer ErikaOS-Anwendung ist in Abbildung 2.2 dargestellt. Als Erstes wird RT-Druid aufgerufen (1). RT-Druid erzeugt ein Makefile, eine C-Quelldatei und eine C-Headerdatei. Das Makefile enthält Informationen dazu, welche Komponenten vom Betriebssystem für die Anwendung benötigt werden. Nutzt die Anwendung zum Beispiel keine Events, so muss dieser Teil des Betriebssystems auch nicht in die resultierende Anwendung eingebunden werden. Dies soll die Größe der resultierenden Anwendungsdatei minimieren. Die generierte C-Datei enthält die benötigten Betriebssystemkomponenten, wie *Tasks* oder *Ressourcen*. Im nächsten Schritt (2) wird das generierte Makefile aufgerufen. Dieses steuert den weiteren Generierungsprozess. Im nächsten Schritt (3) werden die notwendigen ErikaOS-Dateien übersetzt und in der Bibliothek „libee.a“ abgelegt. Der Code der Anwendung, welcher wie hier aus einer einzigen Datei, aber auch aus mehreren Dateien bestehen kann wird übersetzt (4). Die Konfigurationsdatei „eecfg.c“ wird ebenfalls übersetzt (5). Im letzten Schritt (6) werden alle bis hier erzeugten Komponenten zusammengebunden.

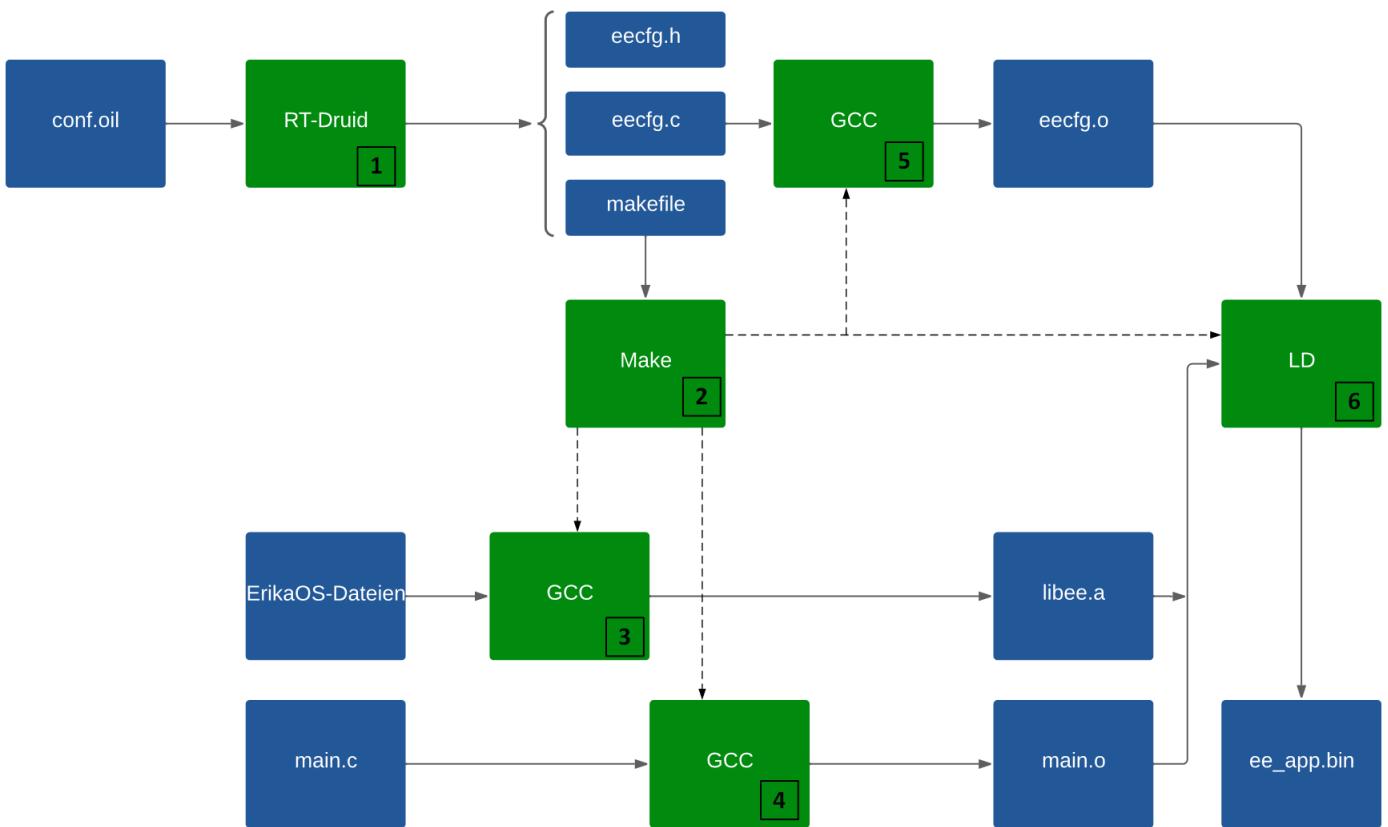


Abbildung 2.2 – ErikaOS: Übersetzungsprozess

2.4 EV3-Baustein

Der EV3-Baustein ist ein Legobaustein, an welchen Motoren und Sensoren (ebenfalls als Legobausteine) angeschlossen werden können. Dabei können bis zu vier Motoren und Sensoren angeschlossen werden. Es gibt einen Farb-, Ultraschall-, Tast- und Girosensor. Weiter bietet der EV3-Baustein einen USB-Port, WiFi und Bluetooth für drahtlose Verbindungen, einen Lautsprecher. Programme können auf einer SD-Karte gespeichert werden [LEG]. Er basiert auf der AM1808 MCU von Texas Instruments, welche eine ARM9-CPU beinhaltet [Ins14, S. 1]. Im weiteren Verlauf werden diese beiden Komponenten vorgestellt.

2.4.1 AM1808

Der AM1808 ist eine MCU, welche neben der CPU weitere Hardwarekomponenten bereitstellt. Die für diese Arbeit relevanten Hardwarekomponenten des AM1808 sind der Timer, der Interruptkontroller und ein Universal Asynchronous Receiver / Transmitter (UART). Der Timer kann Interrupts auslösen. Bei dem Interruptkontroller handelt es sich um den **Arm Interrupt Controller** (AINTC) [Ins14, S. 81]. Der UART ist so verschaltet, dass er über einen Sensorport als UART zur Ein- und Ausgabe verwendet werden kann [HRP14]. Ein UART ist eine serielle Schnittstelle. Wie genau diese Komponenten funktionieren und genutzt werden, ist in Abschnitt 3.2 beschrieben.

2.4 EV3-Baustein

Im Rahmen dieser Arbeit wird ErikaOS nicht nur auf den AM1808, sondern auch auf die Quick Emulator (QEMU) Version des Versatilepb [Ver] portiert. Das Versatilepb ist ein ARM Evaluierungsboard, welches die gleiche CPU wie der AM1808 nutzt. Es wird im weiteren Verlauf als MCU bezeichnet, auch wenn es sich dabei um ein Board handelt. Grund dafür ist, dass es im Rahmen dieser Arbeit nur als MCU genutzt wird. Es verwendet allerdings einen anderen Timer [ARM04b], Interruptkontroller [ARM04c] und UART [ARM]. Daher können die Anteile der Portierung, welche nicht von der spezifischen Hardware der MCUs abhängen, erleichtert innerhalb QEMU debuggt werden.

2.4.2 ARM9

Die vom AM1808 genutzte CPU ist der ARM926RJ-S. Diese gehört der ARM9 Familie an und wird als Allzweck-Mikroprozessor beschrieben. Sie nutzt die ARM Architektur v5TEJ [ARM08, S. 26].

Eine ARM9-CPU besitzt dreizehn Allzweck-Register (r0 bis r12), sowie drei spezielle Kontrollregister. Der Stackpointer oder auch SP (r13) enthält den Zeiger auf den aktuellen Stack. Das Linkregister oder auch LR (r14), welches Rücksprungadressen speichert und den Programmzähler oder auch PC (r15), welches die Adresse der aktuellen Instruktion enthält. Das Linkregister kann ebenfalls als ein Allzweck-Register genutzt werden. Dies gilt auch für den Stackpointer, jedoch nur im ARM-Modus [ARM16, S. 42ff].

Neben den Allzweck-Registern besitzt die CPU noch das sogenannte „Current Program Status Register“ (CPSR), sowie das „Saved Program Status Register“ (SPSR). Das CPSR enthält Informationen, wie zum Beispiel die Bedingungsflaggen für Vergleichsoperationen, die Interruptmaske oder ob sich die CPU im ARM- oder Thumbmodus befindet [ARM08, S. 26]. Das CPSR gehört zu den Status beziehungsweise Kontrollregistern des CP15. Der CP15 ist der Ko-Prozessor des ARM9, welcher dazu dient den eigentlichen Prozessor den ARM926EJ-S, sowie weitere Hardwarekomponenten wie Caches oder die Memory-Management-Unit zu konfigurieren [ARM08, S. 32]. Diese Status- und Kontrollregister können nur über die dafür vorgesehenen Instruktionen „MCR“, „MSR“ und „MRC“, „MRS“ geschrieben beziehungsweise gelesen werden [ARM16, S. 397, 414, 408, 410].

Die CPU unterstützt sechs unterschiedliche Betriebsmodi. Davon werden in der Arbeit nur der Supervisor-Modus und IRQ-Modus genutzt. Im normalen Betrieb ist die CPU in dieser Arbeit stets im Supervisor-Modus. Bei der Interruptbehandlung wechselt diese in den IRQ-Modus. Der ARM9 unterscheidet zwischen schnellen Interrupts (FIQ) und normalen Interrupts (IRQ), für welche jeweils auch ein eigener Betriebsmodus existiert. Diese unterscheiden sich in ihrer Priorität und darin wie viele Register durch die Hardware gesichert werden [ARM02, S. 52].

Für das CPSR, das Linkregister und den Stackpointer existieren Schattenregister. Diese werden automatisch aktiviert, wenn die CPU in den IRQ-Modus wechselt. Das CPSR des Supervisor-Modus wird im IRQ-Modus als SPSR bezeichnet. Die Betriebsmodi korrespondieren mit den *Exceptions* des ARM9. Eine *Exception* kann durch unterschiedliche Zustände ausgelöst werden und besitzt stets ihren eigenen Betriebsmodus. Zu *Exceptions* zählen zum Beispiel Interrupts, ein Reset oder das Erreichen einer undefinierten Instruktion. Der ARM9 besitzt eine Vektortabelle im Hauptspeicher, welche Instruktionen beinhaltet, die ausgeführt werden, wenn die dementsprechende *Exception* ausgelöst wird.

Um ARM-Assembler mit C-Funktionen zu nutzen, muss die korrekte Aufrufkonvention beachtet werden. Diese ist im sogenannten Extended Application Binary Interface (EABI) definiert [ARM22]. Gemäß EABI werden Argumente an Funktionen über die ersten vier Allzweck-Register (r0-r3) übergeben. Weitere Argumente müssen über den Stack übergeben werden. Die nächsten sieben Register (r4-r11) werden als Register für lokale Variablen genutzt, wobei r11 auch als Framepointer dienen kann. Der Rückgabewert einer Funktion wird in r0 übergeben.

Der Aufrufer muss daher r0 bis r3 vor dem Aufrufen einer C-Funktion sichern, wenn deren Inhalte auch nach dem Aufruf noch benötigt werden. Dies gilt nicht für r4 bis r11, da der Aufgerufene diese nicht verändert hinterlassen darf [ARM22].

Die CPU unterstützt zwei unterschiedliche Instruktionsmodi. Den 32 Bit ARM-Modus und den 16 Bit Thumbmodus. Dabei sind im ARM-Modus alle Instruktionen 32 Bit und im Thumbmodus alle Instruktionen 16 Bit breit. Es ist möglich im laufenden Betrieb zwischen diesen Modi zu wechseln. Somit können entweder schnellere, aber dafür größerer Anwendungen oder kleinere, aber dafür langsamere Anwendungen erzeugt werden [ARM08, S. 26].

2.4.3 Bootvorgang

Details zu dem Bootvorgang sind im technischen Bericht Nr. 90 des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam [HRP14] zu finden. Nach dem Einschalten des EV3-Baustein startet der Bootlader aus dem ROM. Bei diesem handelt es sich um U-Boot [Uboa]. U-Boot prüft zunächst, ob auf einer SD-Karte ein bootbare Datei („boot.scr“) vorhanden ist. Ist dies nicht der Fall, so wird die originale EV3-Baustein Firmware gestartet. Liegt eine „boot.scr“ Datei vor, so werden die darin enthaltenen U-Boot-Befehle ausgeführt. Mittels U-Boot-Befehlen kann zum Beispiel Speicher gelesen und modifiziert, aber auch Daten von der SD-Karte in den Speicher geladen werden. Auch kann eine Anwendung, die im Speicher liegt, gestartet werden.

U-Boot protokolliert den Bootvorgang mittels Ausgaben über einen UART, welcher physikalisch mit dem Sensorport eins verbunden ist. Ebenso ist es möglich über den UART Befehle an U-Boot zu senden [HRP14].

2.5 Verwandte Arbeiten

Neben dem originalen Betriebssystem von Lego gibt es für den EV3-Baustein viele Betriebssysteme, welche als Hilfestellung für den Portierungsvorgang von ErikaOS genutzt werden können. Dazu zählen unter anderem die Debian basierte Linuxversion ev3dev [Ev3] oder die Echtzeitbetriebssysteme nxtOSEK [Lej] und Ninjastorm [HRP14]. Die Vielzahl der Betriebssysteme kann darauf zurückgeführt werden, dass der EV3-Baustein Open Source ist.

Die hilfreichste Arbeit ist der technische Bericht Nr. 90 des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Dieser umfasst die Entwicklung einer bare-metal Hallo-Welt-Anwendung, sowie des Echtzeitbetriebssystems Ninjastorm für den EV3-Baustein. Zusätzlich gibt es eine Variante des Ninjastorm Betriebssystems, die auf dem Versatilepb mit QEMU lauffähig ist [HRP14].

Mit den Betriebssystemen nxtOSEK existiert eine Implementierung des OSEK/OS-Standards für den EV3-Baustein. nxtOSEK ist jedoch aufgrund von schwerwiegenden Fehlern nur eingeschränkt

2.5 Verwandte Arbeiten

nutzbar [Hö+18].

Neben den bisher genannten Betriebssystemen stellt Texas Instruments mit der Starterware [Ins11] ein umfangreiches Softwarepaket zur Verfügung. Dieses stellt Treiber für die Hardware des AM1808 einschließlich Beispielanwendungen bereit. Die Starterware wurde bei der Portierung teilweise verwendet, zum Beispiel für die Implementierung eines Watchdogs.

Für ErikaOS existieren bereits verschiedene Portierungen. Insbesondere auch für andere ARM Prozessoren, wie ARM7 oder Cortex-M. Deren Code ist Bestandteil des ErikaOS Codes [Gai]. Allerdings gibt es für diese Portierungen keine Beschreibung zum Vorgehen des Portierungsprozesses. Weiterhin sind diese Portierungen nicht geeignet, um ErikaOS auf ARM9 laufen zu lassen.

2.6 Zusammenfassung

In dem Kapitel 2 wurde der Echtzeitbetriebssystemstandard OSEK/OS, die grundlegende Struktur von ErikaOS, sowie der EV3-Baustein und dessen Hardware, die MCU AM1808 und die CPU ARM926RJS, vorgestellt. Weiterhin wurden verwandte Arbeiten zum EV3-Baustein betrachtet. Obwohl viele Softwareprojekte für den EV3-Baustein und viele Portierungen von ErikaOS existieren, gibt es keine Arbeiten, die die Portierung von ErikaOS auf den EV3-Baustein zum Ziel haben.

3

UMSETZUNG

Dieses Kapitel beschreibt die für die Portierung benötigte Entwicklungsumgebung. Sie wird anhand einer Hallo-Welt-Anwendung vorgestellt. Des Weiteren werden die für die Portierung erforderlichen Anpassungen an ErikaOS beschrieben. Es wird im Detail dargestellt, welche Funktionalität für den ErikaOS-Kern bereitgestellt werden muss.

3.1 Entwicklungsumgebung

Beim EV3-Baustein handelt es sich um ein eingebettetes Gerät. Daraus ergeben sich Besonderheiten bei der Entwicklung von Software für diesen. Insbesondere sind die folgenden Aspekte zu berücksichtigen.

1. Wie kann eine Anwendung übersetzt werden?
2. Wie kann eine Anwendung gebunden und geladen werden?
3. Wie kann die Ausführung einer Anwendung von außen protokolliert werden?
4. Wie kann eine Anwendung debuggt werden?

Den ersten drei Fragen wird auch in dem Bericht [HRP14] nachgegangen. Die in dieser Arbeit verwendeten Lösungen basieren zum Teil auf diesem Bericht. Sie werden mittels einer Hallo-Welt-Anwendung vorgestellt. Die Hallo-Welt-Anwendung ist im Verzeichnis *hello_world* des praktischen Teils zu finden.

3.1.1 Kommunikation über UART

Um eine klassische Hallo-Welt-Anwendung zu erstellen, muss diese eine Ausgabe erzeugen. In der Beispielanwendung wird hierzu der mit dem EV3-Sensorport verbundene UART verwendet. Die Anwendung ist in Listing 3.1 dargestellt. Softwaretechnisch ist eine Ausgabe über den UART einfach zu realisieren. Dazu muss lediglich das passende UART Register mit dem auszugebenden Byte beschrieben werden (Zeile 7). Vorher muss gegebenenfalls gewartet werden, bis der UART signalisiert, dass er bereit ist (Zeile 5).

3.1 Entwicklungsumgebung

```
1 #define UART_THR (volatile char *) (0x01D0C000)
2 #define UART_LSR (volatile char *) (0x01D0C014)
3
4 void put_char(int c) {
5     while (!(*UART_LSR & (1 << 5))) {} // Warte bis UART bereit
6
7     *UART_THR = c;                      // Gebe Byte aus
8 }
9
10 void put_string(const char *s) {
11     for (; *s != 0; s++) {
12         putchar(*s);
13     }
14 }
15
16 void hello_world(void) {           // Entry point
17     put_string("Hello World!\n");
18 }
```

Listing 3.1 – ARM9 bare-metal Hello World Code

Hardwaretechnisch muss der UART mit der Entwicklungsmaschine verbunden werden. Dazu ist ein passendes Verbindungskabel nötig. Auf der Seite des EV3-Baustein muss dieses einen EV3-Sensorstecker besitzen. Über diesen müssen die UART-Signale *Receive* (Rx), *Transmit* (Tx) und *Ground* (GND) des EV3-Baustein mit einem UART der Entwicklungsmaschine verbunden werden. Hierfür eignet sich ein USB-UART-Dongle [Uar]. Der im Rahmen dieser Arbeit erstellte UART-Anschluss ist in Abbildung 3.1 zu sehen. Eine Anleitung zur Erstellung eines solchen ist online [Sol13] zu finden.

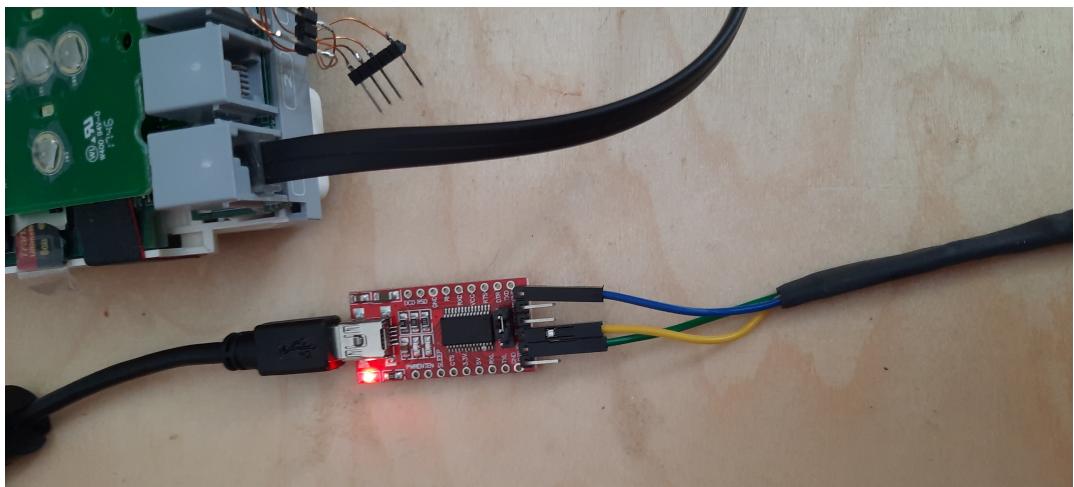


Abbildung 3.1 – EV3-Baustein: UART-Anschluss

3.1.2 Kompilierung

Typischerweise unterschiedet sich die Rechnerarchitektur der Entwicklungsmaschine (x86) von der Zielmaschine (ARM9). Daher wird für die Übersetzung ein Cross-Compiler benötigt, der über Argumente so gesteuert wird, dass auf der Zielmaschine ablauffähiger Code entsteht.

Die Beispielanwendung verwendet das Kommandozeilentool make. Listing 3.2 zeigt den für die Übersetzung relevanten Ausschnitt des Makefiles. Als Cross-Compiler wird arm-none-eabi-gcc verwendet (Zeile 13).

```
1 # Wie agiert Übersetzer
2 COMPILED_FLAGS = -g -O2 -pipe
3
4 # Welche Warnungen und Fehler sollen wie behandelt werden
5 WARNING_FLAGS = -Wall -Wextra -Wstrict-prototypes -Werror
6
7 # ARM spezifische Übersetzerargumente, wie zum Beispiel welche ARM \
8 # Architektur
9 ARM_FLAGS = -marm -mabi=aapcs-linux -march=armv5te \
10 -mno-thumb-interwork
11
12 # In welcher Umgebung wird die Anwendung laufen
13 ENVIRONMENT_FLAGS = -fno-common -msoft-float -fno-builtin \
-ffreestanding -nostdinc -fno-stack-protector
14
15 arm-none-eabi-gcc $(COMPILED_FLAGS) $(ENVIRONMENT_FLAGS) -isystem \
$(INC_GCCDIR) $(WARNING_FLAGS) -o hello_world.o hello_world.c -c
```

Listing 3.2 – EV3-Baustein: Übersetzeraufruf für Hallo-Welt-Anwendung

Der Aufruf des Übersetzers benötigt Argumente, welche die Zielmaschine und die Umgebung der Anwendung auf der Zielmaschine beschreiben. Davon unabhängig sind die Argumente „COMPILED_FLAGS“ und „WARNING_FLAGS“. Diese sorgen zum Beispiel für die Ausgabe von Warnungen oder setzen die Optimierungsstufe. Sie haben jedoch nichts mit der eigentlichen Kompilierung zu tun. Die „ARM_FLAGS“ legen die für die Übersetzung für ARM9 relevanten Komplieroptionen fest. Die „ENVIRONMENT_FLAGS“ beschreiben die Eigenschaften der Umgebung, in der die Anwendung laufen soll. Sie stellen sicher, dass die generierte Anwendung keine Abhängigkeiten zu Bibliotheken der Entwicklungsmaschine enthält. Damit der Linker nicht automatisch aufgerufen wird, bekommt der Übersetzer das Argument „-c“ übergeben, sodass lediglich eine Objektdatei erzeugt wird. Die genaue Beschreibung der Optionen findet sich in Kapitel 6.

3.1.3 Binden

Eine Anwendung für den EV3-Baustein muss so gebunden werden, dass diese zum Speicherlayout des EV3-Bausteins passt. Der Hauptspeicher vom EV3-Baustein startet bei der Adresse 0xC0000000 und endet bei der Adresse 0xFFFFFFFF [Ins14, S. 13].

3.1 Entwicklungsumgebung

Der Linkeraufruf erzeugt eine ELF-Datei der Anwendung und legt das Textsegment an die Adresse 0xC1000000 (-Ttext 0xC1000000), also im Hauptspeicher des EV3-Bausteins. Zudem wird libgcc eingebunden. Damit der EV3-Baustein die Anwendung ausführen kann, muss aus der ELF-Datei eine Binärdatei erzeugt werden. Der genaue Aufruf des Linkers ist in Kapitel 6 abgebildet.

3.1.4 Deployment

Der einfachste Weg, die Anwendung zu laden und zu starten, besteht darin, diese mit einer „boot.scr“ zusammen auf einer SD-Karte abzulegen. Die „boot.scr“ muss hierzu U-Boot-Befehle [Ubob] enthalten, welche die Anwendung an die Adresse 0xC1000000 laden und starten. Dazu können die Befehle *fatload* und *go* genutzt werden. *fatload* muss der Gerätename und Instanz der SD-Karte, der Name der zu ladenden Datei und die Zieladresse übergeben werden. *go* muss die Adresse der Eintrittsfunktion übergeben werden. Das Erzeugen der „boot.scr“ ist in Kapitel 6 genau beschrieben.

Wird der EV3-Baustein mit einer SD-Karte mit der „boot.scr“ und „hello_world.bin“ gestartet, so wird die Anwendung ausgeführt. Über den UART wird „hello world“ ausgegeben.

Der im vorherigen Abschnitt beschriebene Deploymentvorgang erfordert, dass nach jeder Änderung am Anwendungscode die neu erzeugte Binärdatei auf die SD-Karte kopiert wird. Dazu muss die SD-Karte jedesmal entnommen und in einen Kartenleser an der Entwicklungsmaschine gesteckt werden. Dieser Ansatz ist nicht geeignet um ein komplexes Projekt wie die Portierung von ErikaOS effizient durchzuführen. Neben der langen Zeit von der Codeänderung bis zum Ausführen der Anwendung, ist auch das häufige Stecken und Ziehen der SD-Karte für die Lebensdauer der Kontaktflächen des EV3-Bausteins, der SD-Karte und des Kartenlesers nicht gut. Aus diesem Grund wurde im Rahmen dieser Arbeit der folgende Deploymentvorgang entwickelt. Die entsprechende Hallo-Welt-Anwendung ist im Verzeichnis *hello_world_v2.0* des praktischen Teils zu finden.

Die Idee für diesen Deploymentvorgang besteht darin, eine unveränderliche „boot.scr“ zu verwenden und die Anwendung über den UART zu laden, anstatt sie auf der SD-Karte abzulegen. Dazu kann der U-Boot-Befehl *loadb* genutzt werden. Dieser verwendet C-Kermit [Cke]. C-Kermit ist ein Programm zur Datenübertragung über eine serielle Schnittstelle. Die Daten werden als ASCII-Zeichen kodiert zwischen zwei C-Kermit Instanzen übertragen.

Damit dieses Vorgehen möglich ist, muss die Adresse der Eintrittsfunktion der Anwendung immer gleich bleiben, da andernfalls die „boot.scr“ Datei anzupassen wäre. Um sicherzustellen, dass die Adresse der Eintrittsfunktion sich nicht ändert, wird dieser eine eigene Sektion *start* zugewiesen (*__attribute__((section("start")))*). Über die Linkeroption *-section-start=start=0xC1000000* wird festgelegt, an welche Adresse diese Sektion und somit auch die Eintrittsfunktion gebunden wird.

Wenn der EV3-Baustein mit einer „boot.scr“ gestartet wird, welche *loadb* aufruft startet C-Kermit. Dieses wartet auf das Empfangen einer Datei. Zum Senden der Datei muss auf der Entwicklungsmaschine C-Kermit gestartet werden. Die zum Senden notwendigen C-Kermit-Kommandos können in einem Skript übergeben werden. Das in dieser Arbeit verwendete Skript wird im Verzeichnis *erika/hello_world_v2.0* erzeugt.

Ist das Hochladen der Datei abgeschlossen, startet der *go* Befehl die Anwendung. Um eine geänderte Anwendung zu laden, muss der EV3-Baustein lediglich neugestartet und die neue Anwendung über C-Kermit geladen werden.

3.1.5 Debuggen

In diesem Kapitel werden die in dieser Arbeit betrachteten und eingesetzten Debugverfahren vorgestellt.

printf-Debuggen

Über die serielle Schnittstelle kann printf-Debuggen erfolgen. Die Realisierung ist einfach, da dazu lediglich der UART angesprochen werden muss. In Listing 3.1 wurde bereits eine Implementierung für eine Ausgabefunktion vorgestellt. printf-Debuggen ermöglicht die Beobachtung des Programmablaufs und das Ausgeben von Speicherinhalten. Dazu müssen an geeigneten Stellen Ausgaben eingefügt werden.

Jedoch kann das Einfügen von Ausgaben zu einem veränderten Zustand führen. Insbesondere ist printf-Debuggen nicht geeignet um Assemblersequenzen zu debuggen. So verändert die Branch-instruktion BL das Linkregister, sowie möglicherweise r0 bis r3. Um komplexere Funktionen in Assembler, wie zum Beispiel einen Kontextwechsel, zu debuggen, sind daher verbesserte Debugmöglichkeiten erforderlich.

QEMU

QEMU [QEM] ist ein Program zur Emulation der Hardware eines Rechners. Es werden verschiedenste Rechnerarchitekturen (Zielhardware) unterstützt. QEMU erlaubt es Anwendungen für eine Zielhardware auf der Entwicklungsmaschine auszuführen. Weiterhin stellt QEMU einen GDB-Server bereit, sodass Anwendungen mittels GDB debuggt werden können. Der AM1808 wird von QEMU nicht unterstützt. Allerdings existiert für QEMU eine Implementierung für das Versatilepb. Somit ist es möglich, Anwendungen für ARM926EJ-S über den QEMU und GDB zu debuggen. Bezüglich der ErikaOS Portierung, welche auch AM1808 spezifische Hardware nutzt, ergibt sich lediglich folgende Einschränkung. Die AM1808 hardwarespezifischen Codeanteile lassen sich nicht mit QEMU debuggen. Dazu zählen der Interruptkontroller, der Timer und die Ausgabe über den UART. Um die übrigen Codeanteile der Portierung auf QEMU debuggen zu können, müssen für Interruptkontroller, Timer und UART entsprechende Treiber für das Versatilepb realisiert werden. Der Vorteil des debuggen mit QEMU ist, dass QEMU die vollständige Kontrolle über die CPU hat. Somit kann die Anwendung zu jeder Zeit unterbrochen und Register und Speicherinhalte geprüft werden. Ebenso erlaubt QEMU ein schnelles Laden, Starten und Ausführen der Anwendung. Wie QEMU und GDB genutzt werden können um ErikaOS-Anwendungen zu debuggen ist in Kapitel 6 vorgestellt.

JTAG

Der EV3-Baustein besitzt ein JTAG-Interface. Über dieses kann ein JTAG-Debugger angeschlossen werden. Ein solcher ermöglicht, ähnlich wie QEMU, die vollständige Kontrolle über die CPU. Auch das Laden von Programmen ist über JTAG möglich. Aus diesem Grund wurde im Rahmen dieser Arbeit versucht, ein JTAG-Debugger einzusetzen. Um über JTAG zu debuggen, wird dementsprechende Hardware (JTAG-Dongle) und dazu passende Software benötigt.

Die JTAG-Pins sind im Schaltplan mit „TP24“ bis „TP33“ bezeichnet [LEG13]. Auf der Platine des EV3-Baustein sind die Namen dieser Pins ebenfalls zu erkennen (siehe Abbildung 3.2). Die Pins sind zusätzlich von eins bis zehn durchnummieriert. Die Zuordnung dieser Pins zu den JTAG-Signalen ist in Tabelle 3.1 dargestellt. Zum Debuggen des EV3-Baustein sind mindestens die JTAG-Signale „GND“, „TCK“, „TDO“, „TDI“ und „TMS“ erforderlich.

3.1 Entwicklungsumgebung

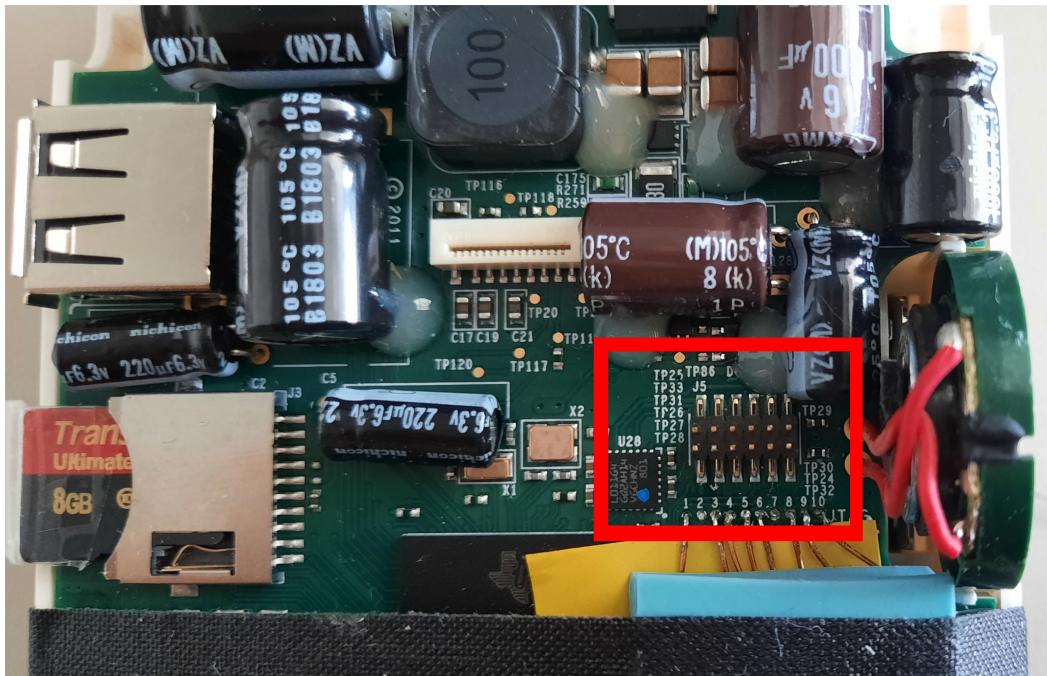


Abbildung 3.2 – EV3-Baustein: JTAG Pins

Nummer	TP-Name	JTAG-Signal
1	TP25	nTRST
2	TP33	RESET
3	TP31	RESET_1
4	TP26	TDI
5	TP27	TMS
6	TP28	TCK
7	TP29	RTCK
8	TP30	TDO
9	TP24	VCC
10	TP32	GND

Tabelle 3.1 – EV3-Baustein: JTAG Pin Zuordnung

Texas Instruments stellt drei JTAG-Dongle zur Verfügung, welche für den AM1808 ausgelegt sind [Am1]. Im Rahmen dieser Arbeit stand jedoch nur ein HS3 JTAG-Dongle von Diligent zur Verfügung [Dil21]. Für diesen stand keine vom Hersteller bereitgestellte Software zur Verfügung, sodass OpenOCD (Open On-Chip-Debugger) [Ope] genutzt wurde. Mit diesem Ansatz ließ sich jedoch keine Verbindung zum EV3-Baustein aufbauen. Als Grund hierfür kommen unter anderem folgende Ursachen in Frage: OpenOCD benötigt an JTAG-Dongle und Zielmaschine angepasste Konfigurationsdateien. Während OpenOCD eine Konfigurationsdatei für den HS3 JTAG-Dongle beinhaltet, fehlt diese für den AM1808. Es existiert lediglich eine Konfigurationsdatei für den Nachfolger des AM1808 (OMAP-L138), welche probeweise verwendet wurde. Grundsätzlich lassen sich diese Dateien manuell anpassen, was jedoch im Rahmen dieser Arbeit als nicht zielführend erachtet wurde. Texas Instruments weist in dem Datenblatt des AM1808 ausdrücklich darauf hin, dass JTAG-Dongle die „TRST“ Leitung aktiv treiben müssen, was nicht für alle JTAG-Dongle der Fall ist. Grund dafür ist, dass der AM1808 interne Pulldown-Widerstände verwendet [Ins14, S. 71].

Da es nicht möglich war, den JTAG in angemessener Zeit in Betrieb zu nehmen und mit QEMU eine ausreichend gute Debugmöglichkeit existiert, wurde dieser Ansatz nicht weiter verfolgt.

3.1.6 Reset-Schaltung

Fehler in der Anwendung führen sehr leicht dazu, dass der EV3-Baustein in einen undefinierten Zustand wechselt. Um in solchen Fällen eine einfache Möglichkeit zu haben, die komplette Hardware zurückzusetzen, wurde eine einfache Reset-Schaltung verwendet (siehe Abbildung 3.3). Diese besteht aus einem Taster (1) (Öffner) der die Stromversorgung bei Betätigung unterbricht. Dazu wurde ein Pol des Akkus mit Klebeband abgeklebt (2) und dessen Kontaktfläche stattdessen über den Taster (1) mit dem dazugehörigen Pol (3) am EV3-Baustein verbunden. Normalerweise ist der Pol (2) über die Feder (3) direkt mit dieser verbunden, sodass für einen vollständigen Reset jedes Mal der Akku entnommen werden müsste.

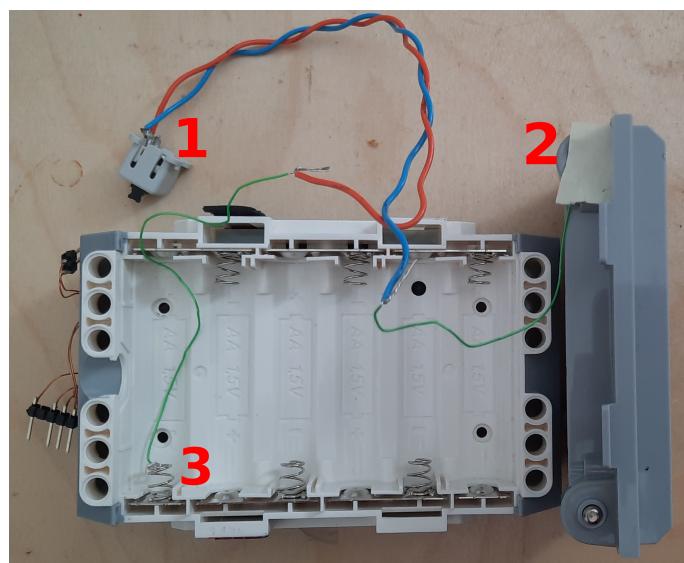


Abbildung 3.3 – EV3-Baustein: Reset-Taster

3.2 Implementierung

3.2 Implementierung

Dieses Kapitel dokumentiert die Portierung von ErikaOS. Während im vorherigen Kapitel allgemein die Entwicklungsumgebung für ARM9-Anwendungen dargestellt wurde, werden in diesem Kapitel die Details der Portierung erläutert. In Abbildung 3.4 sind die zur Portierung gehörenden Verzeichnisse dargestellt. Die den EV3-Baustein betreffenden Anpassungen sind in den Verzeichnissen *arm9* und *am1808* enthalten. Das Verzeichnis *versatilepb* enthält Anpassungen, die nötig sind, um ErikaOS auf QEMU laufen zu lassen. Zusätzlich sind im Verzeichnis *pkg* angepasste Header (zum Beispiel *ee.h*) enthalten. Diese Header binden die neuen Dateien der Portierung ein. Im Verzeichnis *apps* sind einfache ErikaOS-Anwendungen zu finden. Diese wurden verwendet, um einzelne Schritte der Portierung zu testen. Das Verzeichnis *testcases* enthält die Implementierung der Testfälle, welche zur Evaluation genutzt wurden.

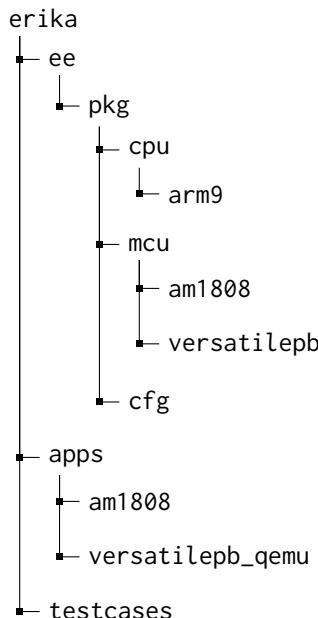


Abbildung 3.4 – ErikaOS Portierung: Verzeichnisstruktur

Für die Portierung müssen Funktionen bereitgestellt werden, die CPU und MCU spezifische Eigenschaften kapseln. Der Umfang der benötigten Funktionen hängt davon ab wie ErikaOS genutzt wird. Zum Beispiel werden für Anwendungen, die der Konformitätsklasse *BCC1* in der Monotstackvariante angehören, keine Funktionen zum Kontextwechsel benötigt. Aus diesem Grund erfolgte die Portierung in den folgenden Schritten. Diese entsprechen den in den folgenden Unterkapiteln beschriebenen Anpassungen.

1. RT-Druid Anpassungen
2. Starten und Terminieren von *Tasks*
3. Bootfunktion
4. präemptives Scheduling

5. Interrupts
6. Multistack und Kontextwechsel

3.2.1 RT-Druid Anpassungen

RT-Druid erwartet, dass in der OIL-Datei die CPU spezifiziert wird. Dies erfolgt über das Attribut `CPU_DATA = CPU_NAME`. Diesem Attribut dürfen nur Werte zugewiesen werden, die RT-Druid bekannt sind. Ohne Anpassungen von RT-Druid werden somit keine neuen CPUs unterstützt. RT-Druid ist eine in Java programmierte Anwendung. Da diese Open Source geführt wird, ist es grundsätzlich möglich Anpassungen vorzunehmen. Diese Option wurde nach Analyse des Codes verworfen. Der Grund dafür ist, dass RT-Druid mittels eines einfachen Workarounds nutzbar gemacht werden konnte. Dieser Workaround führt zu keinen Einschränkungen für den Nutzer dieser Portierung. Vor diesem Hintergrund wäre der Aufwand für das Anpassen von RT-Druid für diese Arbeit nicht angemessen gewesen. Dieser Workaround besteht darin in die OIL-Datei die von RT-Druid unterstützte Cortex-M CPU einzutragen und zusätzlich mittels Attribute „EE_OPT“ den Wert „__ARM9__“ festzulegen. Die Headerdateien der Portierung wurden so angepasst, dass sie in diesem Fall die von RT-Druid generierten Dateien, so auswerten, als wäre `CPU_DATA = ARM9` gesetzt worden.

Die Details zum Workaround finden sich in Kapitel 6

3.2.2 Starten und Terminieren von Tasks

ErikaOS erwartet, dass von der Portierung jeweils eine Funktion zum Starten (`EE_hal_terminate_savestk`) beziehungsweise Terminieren (`EE_hal_terminate_task`) einer *Task* bereitgestellt wird. Für diese sind in der Datei `erika/ee/pkg/cpu/arm9/inc/ee_internal.h` Wrapper-Funktionen definiert. Die eigentliche Implementierung ist in Assembler geschrieben und findet sich in `erika/ee/pkg/cpu/arm9/src/ee_terminate.S`.

Die Startfunktion sichert den aktuellen Kontext. Dieser besteht aus dem Stackpointer, sowie den Registern r4 bis r12 und dem Linkregister. Die Register r4 bis r12 und das Linkregister werden auf dem Stack gesichert. Der Stackpointer wird in dem globalen Array `EE_terminate_data` gesichert, welches in „`eecfg.c`“ deklariert ist. Die Register r0 bis r3 müssen nicht gesichert werden, da diese gemäß der Aufrufkonvention vom Aufrufer, also in diesem Fall ErikaOS, gesichert werden müssen. Als letztes wird die zur *Task* gehörende Anwenderfunktion gestartet.

Die Terminierungsfunktion stellt den gesicherten Kontext wieder her. Dafür wird der Stackpointer aus dem globalen Array `EE_terminate_data` geladen und anschließend die Register wiederhergestellt. Als letztes wird zur Adresse des Linkregisters gesprungen.

Bereits mit Bereitstellung dieser Funktionen liegt eine funktionsfähige Portierung vor, sofern die ErikaOS Anwendung der Konformitätsklasse *BCC1* mit Monostackvariante entspricht und keine Interrupts nutzt. Deshalb wird keine Bootfunktion benötigt, die *Exception-Vektortabelle* oder MCU Hardware, wie Interruptkontroller oder Timer initialisiert. Daher kann unmittelbar nach dem Starten der Systemaufruf `StartOS` erfolgen, welcher ErikaOS startet. Die Anwendung im Verzeichnis `erika/apps/am1808/compile_test` enthält ein Beispiel hierzu.

3.2 Implementierung

3.2.3 Bootfunktion

Damit auch Anwendungen mit *Extended Tasks*, die ErikaOS als Multistackvariante oder Interrupts nutzen, eingesetzt werden können, ist eine Bootfunktion erforderlich, welche die hierfür notwendigen Initialisierungen vornimmt. Der Code der Bootfunktion verteilt sich auf die Dateien *reset.S* und *init.c* im Verzeichnis *erika/ee/pkg/cpu/arm9/src*. „*reset.S*“ enthält die Eintrittsfunktion *reset_handler*, in welche von U-Boot aus gesprungen wird. Diese Funktion setzt die CPU in den Supervisor-Modus, initialisiert die Stackpointer für den Supervisor-Modus und den IRQ-Modus sowie die *Exception*-Vektortabelle. Die Portierung verwendet nur die *Exception*-Vektoren für den Reset (0) und Interrupts (6). Für die restlichen Vektoren wird ein Dummy-Handler eingetragen, welcher den Namen der *Exception* ausgibt und anschließend in einer Endlosschleife verbleibt. Am Ende übergibt der Reset-Handler die Kontrolle an die C-Funktion *system_init*. Diese initialisiert den Interruptkontroller des AM1808 und startet ErikaOS mit dem Systemaufruf *Startos*.

Im Gegensatz zu dem Beispiel aus Abschnitt 3.1.4 bei dem die Adresse der Eintrittsfunktion über eine eigene Sektion festgelegt wurde, wird dies für die Eintrittsfunktion *reset_handler* über ein Linkerskript realisiert. Die Linkerskripte sind jeweils in *erika/ee/pkg/mcu/am1808/cfg/am1808_linker.ld* und *erika/ee/pkg/mcu/versatilepb/cfg/versatilepb_linker.ld* zu finden. Hintergrund dafür ist, dass sich die unterschiedlichen Speicherlayouts des AM1808 und Versatilepb einfacher im Generierungsprozess berücksichtigen lassen. Auch das Zuweisen der Speicherbereiche für die Stacks lässt sich so übersichtlicher realisieren.

3.2.4 Präemptives Scheduling

Die Aktivierung des präemptiven Scheduler erfolgt, abgesehen von einer Aktivierung, aus dem ErikaOS-Kern heraus. Der Scheduler muss auch aktiviert werden, wenn aus einer Interruptbehandlung zurückgekehrt wird, was nicht Teil des ErikaOS-Kerns ist. Der Grund dafür ist, dass nach einem Interrupt potenziell eine neue *Task* lauffähig sein könnte. Da die Behandlung von Interrupts zum hardwarespezifischen Code gehört, muss diese Aktivierung des Schedulers über den Code der Portierung erfolgen. Dieser ist in *erika/ee/pkg/cpu/arm9/src/ee_irqsched.S* und *erika/ee/pkg/mcu/am1808/src/interrupt_handler.S* beziehungsweise *erika/ee/pkg/mcu/versatilepb/src/interrupt_handler.S* zu finden.

Der Wechsel aus der Interruptbehandlung zur Scheduler-Funktion muss für die Scheduler-Funktion so aussehen, als hätte die unterbrochene *Task* sie aufgerufen. Der Stack der unterbrochenen *Task* muss dazu so modifiziert werden, dass dieser den selben Zustand hat, als wäre die Scheduler-Funktion unmittelbar nach der unterbrochenen Instruktion aufgerufen worden. Gemäß der Aufrufkonvention bedeutet dies, dass das Linkregister und r0 bis r3 auf dem Stack abgelegt werden müssen. Bis dieser Zustand erreicht ist, müssen Interrupts gesperrt bleiben.

Nach der Interruptbehandlung wird in die Scheduler-Funktion gewechselt, indem das Linkregister auf die Adresse der Funktion *EE_arm9_after_IRQ_schedule* gesetzt wird.

3.2.5 Interrupts

Dieses Kapitel beschreibt die Implementierung von Interrupts für ErikaOS für sowohl den AM1808, als auch das Versatilepb. Dazu werden zunächst die dafür relevanten Hardwarekomponenten (Timer

und Interruptkontroller) beschrieben.

Der AM1808 stellt den sogenannten AINTC und den 64 Bit Timer Puls bereit [Ins16, S. 282, 1467]. Das Versatilepb nutzt den PL190 Interruptkontroller und den ARM Dual-Timer SP804 [Ver]. Um Interrupts global für die CPU zu aktivieren beziehungsweise deaktivieren nutzt der ARM9 das CPSR (Current Programm Status Register). Dieses Register kann nicht direkt, sondern nur über den Ko-Prozessor CP15 geschrieben werden [ARM04a, S. 31]. Der Timer wird in dieser Arbeit verwendet, um eine Interruptquelle bereitzustellen und so die korrekte Behandlung von Interruptserviceroutinen, die von der Anwendung definiert werden, zu prüfen.

Die Implementierung der Interruptbehandlung von AM1808 und Versatilepb unterscheiden sich hinsichtlich des Umfangs. Da die Implementierung des Versatilepb lediglich den Zweck hat das Debuggen zu vereinfachen, wurde eine möglichst einfache Implementierung gewählt und zum Beispiel auf das Zuweisen eines Vektors für Interrupts verzichtet.

3.2.5.1 AM1808

Dieses Kapitel beschreibt die AM1808 spezifische Implementierung der Interruptbehandlung.

Arm Interrupt Controller (AINTC)

Der AINTC dient als ein Interface zwischen externen Interruptquellen und der CPU. Während der AM1808 101 verschiedene Interruptquellen unterstützt, besitzt der AINTC jedoch nur 32 Interruptkanäle. Somit muss eine Zuordnung von Interruptquellen zu Interruptkanälen vorgenommen werden. Die Interruptkanäle wiederum sind den *Exception*-Vektoren für schnelle (FIQ) und normale (IRQ) Interrupts fest zugeordnet. Die Zuordnung erfolgt implizit über die Kanalnummer. Kanal null und eins sind dem FIQ und die restlichen den IRQ zugeordnet [Ins16, S. 282ff]. Schnelle Interrupts werden von dieser Portierung nicht unterstützt.

Der AINTC unterstützt Interruptpriorisierung und verschachtelte Interrupts. Die Priorisierung kann über Software oder über Hardware erfolgen. Verschachtelte Interrupts und Priorisierung in Software werden von dieser Portierung nicht unterstützt. Die Hardwarepriorisierung erfolgt über die Kanalnummer des AINTC. Je kleiner die Kanalnummer desto höher die Priorität [Ins16, S. 282ff].

Interrupts kann ein Vektor zugewiesen werden (Vektorisierung). Dabei kann jedem der 101 Interruptquellen eine Interruptserviceroutine zugewiesen werden. Die Nummer des eingehenden Interrupts, sowie die Adresse der zugewiesenen Interruptserviceroutine können, während der Interruptbehandlung aus einem Register des AINTC gelesen werden.

Dazu muss die Adresse für die Interruptserviceroutine aus dem „Host Interrupt Prioritized Vector Register“ und die Interruptnummer aus dem „Host Interrupt Prioritized Index Register“ gelesen werden. Einzelne Interruptquellen können so individuelle Interruptserviceroutine zugewiesen werden. Jede Interruptquelle kann am AINTC deaktiviert oder aktiviert werden. Löst eine Interruptquelle einen Interrupt aus, so wird diese Interruptquelle automatisch deaktiviert und muss nach der Interruptbehandlung wieder aktiviert werden [Ins16, S. 282ff].

3.2 Implementierung

64 Bit Timer Puls

Der 64 Bit Timer Puls unterstützt unterschiedliche Timermodi. Dazu zählen ein 64 Bit, ein verketteter 32 Bit, ein unverketteter 32 Bit und der Watchdog Modus. Im verketteten 32 Bit Modus werden die ersten 32 Bits als Vorskalierer für den aus den restlichen 32 Bits bestehenden Timer genutzt. Der Vorskalierer verlangsamt die Frequenz, womit die restlichen 32 Bits inkrementiert werden. Der Watchdog nutzt den 64 Bit Modus unterliegt aber Restriktionen. So darf zum Beispiel keine externe *Clock* genutzt werden [Ins16, S. 1467ff].

Der Timer kann als Interruptquelle konfiguriert werden und einen Interrupt auslösen. Dies ist jedoch nicht notwendig, um den Timer zu nutzen. Neben den Timermodi existieren noch drei Operationsmodi. Diese bestimmen das Verhalten beim Ablaufen des Timers. Im „One-time operation“ Modus läuft der Timer einmalig, bis er den konfigurierten Maximalwert erreicht hat. Im „Continuous operation“ Modus setzt der Timer beim Ablaufen seinen Wert auf null zurück und beginnt erneut bis zum Maximalwert zu zählen. Der „Continuous operation with period reload“ Modus unterscheidet sich vom „Continuous operation“ Modus darin, dass nach Ablauf des Timers ein geänderter Maximalwert geladen werden kann [Ins16, S. 1467ff].

In dieser Arbeit wird der unverketteten 32 Bit Timer im „Continuous operation“ Modus genutzt. Die Konfiguration ist statisch und lediglich der Maximalwert, kann in der Anwendung mittels der Funktion `EE_timer_init`, festgelegt werden. Zu beachten ist, dass der Timer seinen Interruptausgang solange aktiv hält, bis die Interruptserviceroutine den Interrupt quittiert. Wird der Interrupt nicht quittiert, so löst der Timer unmittelbar nach beenden der Interruptserviceroutine einen weiteren Interrupt aus.

Timerkonfiguration

Die Implementierung des Timers findet sich in `erika/ee/pkg/mcu/am1808/src/timer.c`. Der Timer wird über vier Kontrollregister konfiguriert. Das „Timer Controll Register“ (TCR) erlaubt es den Timer zu aktivieren und zu deaktivieren, sowie die *Clock* auszuwählen. Über das „Timer Global Control Register“ (TGCR) werden Timer- und Operationsmodus gewählt. Das „Timer Period Register 34“ (PRD34) bestimmt den Maximalwert. Das „Timer Interrupt Control and Status Register“ (INTCTLSTAT) legt fest, ob der Timer Interrupts generieren soll. Weiterhin werden Interrupts über dieses Register quittiert.

Um dem Timer als Interruptquelle einem Kanal zuzuordnen, muss das dazugehörige „Channel Map Register“ (CMR) beschrieben werden. Insgesamt gibt es 26 CMR. Jedes CMR kann dafür genutzt werden vier Interruptquellen einen Interruptkanal zuzuordnen. So kann für jede der 101 möglichen Interruptquellen ein Kanal gewählt werden. Der Kanal für Interruptquelle null wird im ersten Byte von CMR0 abgelegt. Die Interruptquelle des Timers hat die Nummer 22 und daher muss der Kanal für diese im dritten Byte von CMR5 eingetragen werden.

Interruptkonfiguration

Die Konfiguration der Interrupts im AINTC erfolgt über fünf Register. In dem „System Interrupt Status Enabled/Clear Register“ (SECR) müssen alle Interruptquellen initial quittiert werden. Mit dem „Host Interrupt Enable Register“ (HIER) werden die 32 Interruptkanäle aktiviert. Damit die

Interruptbehandlung vektorisiert werden kann, muss im „Vector Base Register“ (VBR) noch die Basisadresse der Vektortabellen eingetragen werden, sowie im „Vector Size Register“ (VSR) die Größe jedes Eintrags der Vektortabelle. Über das „Global Enable Register“ (GER) werden Interrupts am AINTC aktiviert.

Interruptbehandlung

Beim Auftreten eines Interrupts schaltet die CPU in den IRQ-Modus. Dadurch werden der Stackpointer und das Linkregister des IRQ-Modus aktiviert. Im CPSR werden Interrupts automatisch gesperrt, während diese im SPSR weiter zugelassen sind. Das Linkregister enthält die Rücksprungadresse der unterbrochenen Funktion. Die Implementierung der Interruptbehandlung ist in *erika/ee/pkg/mcu/am1808/src/interrupt_handler.S* zu finden. Die Interruptbehandlung sichert als Erstes von ihr genutzte Register auf dem eigenen Stack und stellt die Register vor Abschluss wieder her. Die Aufgabe der Interruptbehandlung ist es die von der Anwendung definierte Interruptserviceroutine aufzurufen. ErikaOS muss in der Lage sein zu unterscheiden, ob ein Systemaufruf von einer *Task* oder einer Interruptserviceroutine erfolgt. Daher erwartet ErikaOS, dass die globale Variable *EE_IRQ_nesting_level* vor dem Aufrufen der Interruptserviceroutine inkrementiert und nach der Rückkehr dekrementiert wird. Die Adresse der Interruptserviceroutine kann direkt über die Vektorisierung ermittelt werden. Um die Interruptserviceroutine aufzurufen, muss der Programmzähler auf die Adresse der Interruptserviceroutine gesetzt werden. Damit für Interruptserviceroutine eine gültige Rücksprungadresse existiert muss diese im Linkregister abgelegt werden. Daher wird in das Linkregister die Adresse der übernächsten Instruktion geschrieben (die nächste Instruktion ist das Setzen des Programmzählers). Beim Setzen des Linkregisters wird eine Besonderheit der ADD Instruktion ausgenutzt. Addiert man den Programmzähler, so wird stets ein Versatz von mindestens acht Byte aufaddiert. Somit wird die Rücksprungadresse mit der Instruktion ADD *lr, pc, #0x0* ermittelt.

Nach Durchlaufen der Interruptserviceroutine, wird die Interruptsperrre am AINTC des behandelten Interrupts aufgehoben. Die Interruptbehandlung kehrt mit der Instruktion SUBS *pc, lr, #0x0* zurück. Das „S“ am Ende vom SUBS sorgt dafür, dass das CPSR vom SPSR wiederhergestellt wird und erlaubt somit global wieder Interrupts an der CPU. Da der Betriebsmodus der CPU im CPSR kodiert ist, wird hierdurch auch in den Supervisor-Modus zurückgewechselt.

3.2.5.2 Versatilepb

Dieses Kapitel beschreibt die Versatilepb spezifische Implementierung der Interruptbehandlung.

PL190 Interruptkontroller

Der PL190 erlaubt 32 Interruptquellen. Diese können konfigurationsabhängig einen FIQ oder IRQ auslösen. Im Fall von IRQ können 16 Interruptquellen vektorisiert werden, wobei dies von der Portierung nicht genutzt wird. Die Interruptpriorisierung erfolgt über die Nummer der Interruptquelle (0-31). Je kleiner die Nummer, desto höher die Priorität [ARM04c].

ARM Dual-Timer SP804

Der SP804 unterstützt unterschiedliche Timermodi. Dazu zählen 32 Bit und 16 Bit Timer im freilaufenden, periodischem oder „one-shot“ Modus. Im freilaufenden Modus zählt der Timer fortlaufend abwärts. Erreicht er den Wert null, wird der Timer auf den maximal möglichen Wert gesetzt und

3.2 Implementierung

beginnt erneut abwärtszuzählen. Im periodischen Modus verhält sich der Timer ähnlich zum freilaufenden Modus. Jedoch wird beim Erreichen von null ein konfigurierter Startwert gesetzt. Im „one-shot“ Modus läuft der Timer einmalig bis er null erreicht.

Der Timer kann als Interruptquelle konfiguriert werden und einen Interrupt auslösen. Dies ist jedoch nicht notwendig, um den Timer zu nutzen.

In dieser Arbeit wird der 32 Bit Timer im periodischen Modus genutzt. Die Konfiguration ist statisch und lediglich der Startwert, kann in der Anwendung mittels der Funktion `EE_timer_init`, festgelegt werden. Auch hier muss der Interrupt von der Interruptserviceroutine quittiert werden [ARM04b].

Timerkonfiguration

Die Implementierung des Timer findet sich in `erika/ee/pkg/mcu/versatilepb/src/timer.c`.

Der Timer wird über drei Kontrollregister konfiguriert. Das „Timer1Control“ Register (CTRL) erlaubt es den Timer zu aktivieren und zu deaktivieren und den Timermodus zu wählen. Über das „Timer1Load“ Register (LOAD) wird der Startwert des Timers festgelegt. Über das „Timer1IntClr“ Register (INTCLR) können Interrupts quittiert werden.

Interruptkonfiguration

Die Konfiguration des Timerinterrupt im PL190 erfordert lediglich das Aktivieren von Interrupts generell, sowie die Freischaltung des Timerinterrupts. Beides erfolgt über das „VICINTENABLE“ Register (INTENABLE).

Interruptbehandlung

Der Ablauf der Interruptbehandlung ist im Wesentlichen identisch zu dem vom AM1808. Der einzige Unterschied besteht darin, dass die Adresse der Interruptserviceroutine nicht über Vektorisierung bestimmt wird. Stattdessen wird diese aus der globalen Variable `TIMER_ISR_PTR` übernommen. In dieser wird die Adresse der Interruptserviceroutine hinterlegt. Dieser Ansatz resultiert in der Einschränkung, dass sich ohne größere Anpassungen an der Versatilepb Portierung nur eine Interruptserviceroutine nutzen lässt. Da das Versatilepb nur als Hilfsmittel zum Debuggen verwendet wird, ist dies ausreichend. Die Implementierung der Interruptbehandlung ist in `erika/ee/pkg/mcu/versatilepb/src/interrupt_handler.S` zu finden

3.2.6 Multistack und Kontextwechsel

Die *Tasks* von Anwendungen der Konformitätsklassen *BCC1* und *BCC2* benötigen keinen eigenen Stack. Dies liegt daran, dass *Task*-Wechsel lediglich über Funktionsaufrufen realisiert werden. Im Gegensatz dazu benötigen *Extended Tasks* der Konformitätsklassen *ECC1* und *ECC2* einen eigenen Stack. Dieser ist erforderlich, damit *Tasks* sich wechselseitig Aufrufen können. Daher muss ErikaOS für diese Konformitätsklassen als Multistackvariante bereitgestellt werden. Dazu muss ein Kontextwechsel implementiert werden, welcher beim Wechseln zwischen zwei *Tasks* den Stack wechselt. Die Implementierung ist in `erika/ee/pkg/cpu/arm9/src/ee_context.S` enthalten.

Weiterhin müssen *Tasks* als gestartet markiert werden können. Hintergrund dafür ist, dass innerhalb des Kontextwechsels erkannt werden muss, ob eine *Task* neugestartet oder fortgeführt wird. Das Markieren erfolgt über das globale Array `EE_stack_is_marked`.

Der Kontextwechsel besteht aus der Funktion `EE_std_change_context`. Dieser wird als Argument die ID der *Task* übergeben, zu welcher gewechselt werden soll. Dies ist ausreichend, da die aktive *Task* und der aktive Stack bekannt sind. Zusammengefasst besteht der Kontextwechsel aus dem Sichern der Register auf dem Stack der aktuellen *Task*, dem Wechseln zum Stack der neuen *Task* und dem Wiederherstellen der Register der neuen *Task*.

Im Detail wird zunächst geprüft, ob der aktive Stack dem Stack entspricht zu dem gewechselt werden soll. Nur wenn dies nicht der Fall ist, wird der Stack tatsächlich gewechselt und die Register r4 bis r11 und das Linkregister auf dem Stack gesichert. Die Register r0 bis r3 sind bereits vom Aufrufer der Funktion gesichert. Der Stackwechsel erfolgt, in dem die Adresse des Stacks der zu aktivierenden *Task* als Stackpointer gesetzt wird. Zum Wiederherstellen des Kontexts dieser *Task* werden die Register r4 bis r11 und das Linkregister vom neuen Stack geladen. Die Register r0 bis r3 werden durch das Zurückkehren aus der Funktion vom Aufrufer wiederhergestellt.

Unabhängig davon, ob der Stack gewechselt wurde oder nicht, wird geprüft, ob die zu aktivierende *Task* markiert ist. Eine neu zu startende *Task* wird durch das Aufrufen dieser gestartet. Eine fortzuführende *Task* wird aktiviert indem der Programmzähler auf den Wert des wiederherstellten Linkregisters gesetzt wird.

3.2.7 EV3 Motoren und Sensoren

Die Ansteuerung von Hardwarekomponenten, wie Motoren und Sensoren erfolgt über Hardwareregister des AM1808. Über die Register werden zum Beispiel General Purpose Input/Output (GPIO)-Pins oder Schnittstellen, wie Serial Peripheral Interface (SPI) angesprochen.

OSEK/OS spezifiziert keine standardisierten Systemaufrufe, die Zugriffe auf Hardwarekomponenten abstrahieren. Aus diesem Grund ist für die vollständige Portierung von ErikaOS keine Unterstützung der Hardwarekomponenten erforderlich. Die Ansteuerung der Hardwarekomponenten kann aus der ErikaOS-Anwendung heraus über die Hardwareregister erfolgen. Da es bereits existierende Software mit Treibern für die Motoren und Sensoren [HRP14] beziehungsweise GPIO und SPI [Ins11] gibt, stellt dies kein Problem dar.

Im Rahmen dieser Arbeit wurde daher nur eine Ansteuerung der Motoren zu Testzwecken bereitgestellt. Die Implementierung findet sich in `erika/ee/pkg/board/ev3_mindstorm/src/motor.c`.

3.3 Zusammenfassung

In diesem Kapitel wurde die Entwicklungsumgebung anhand einer Hallo-Welt-Anwendung beschrieben. Es wurde gezeigt, wie diese übersetzt und ohne ein Betriebssystem auf dem EV3-Baustein geladen und dort gestartet werden kann. Für diese Arbeit wurde ein automatisierter Ladevorgang über den UART entwickelt, welcher ebenfalls vorgestellt wurde.

Weiterhin wurden die während der Portierung entstandenen Anpassungen beschrieben. Dazu gehören das Anpassen an RT-Druid, Anpassungen zum Starten und Terminieren von *Tasks*, die Entwicklung einer Bootfunktion, sowie Anpassungen die präemptives Scheduling, Interrupts und die ErikaOS Multistackvariante zulassen.

4

EVALUATION

Dieses Kapitel beschreibt die Evaluation der Portierung. Um die Qualität der Portierung sicher zu stellen, wurden sowohl eigene Testanwendungen erstellt als auch formale OSEK-Konformitätstestfälle angewendet. Die Testanwendungen haben den Zweck zu prüfen, ob die Portierung grundsätzlich funktioniert. Die OSEK-Konformitätstestfälle erlauben es zusätzlich, eine Aussage über die Korrektheit und Vollständigkeit der Portierung zu treffen.

4.1 Testanwendungen

Die Testanwendungen wurden fortlaufend während der Portierung erstellt. Ziel dieser Anwendungen ist das Testen der in einem Portierungsschritt umgesetzten Funktionalität. Sofern die Testanwendungen Fehler in der Portierung angezeigt haben, wurden diese korrigiert und die Testanwendung erneut ausgeführt. Die Testanwendungen sind im Verzeichnis *erika/apps* zu finden.

4.2 OSEK-Konformitätstestfälle

Das Projekt **Methods and tools for the validation of OSEK/VDX - based Distributed Arcitectures (MODISTARC)** definiert standardisierte Konformitätstestfälle für OSEK. Die Konformitätstestfälle erlauben es nachzuweisen, dass eine konkrete Implementierung den OSEK-Standard korrekt umsetzt. Werden die Testfälle erfolgreich ausgeführt, so kann die Implementierung als vollständig und korrekt betrachtet werden.

Die aktuell verfügbare MODISTARC-Spezifikation [Osea] gilt für die OSEK/OS Spezifikation [gro97]. Diese ist älter als der aktuell gültige OSEK/OS-Standard [Oseb]. Die Konformität zu diesem kann somit über MODISTARC nicht nachgewiesen werden.

Eine Untermenge der Konformitätstestfälle betrifft OSEK/OS und ist damit für diese Arbeit relevant. Diese Testfälle sind in die folgenden sechs Testgruppen eingeteilt [gro99].

1. Taskmanagement
2. Interruptbehandlung
3. Eventmechanismen

4.2 OSEK-Konformitätstestfälle

4. Ressourcenmanagement
5. Alarme
6. Fehlerbehandlung, Hookfunktionen und OS-Kontrolle

Die Anzahl der auszuführenden Testfälle variiert, je nachdem für welche Konformitätsklasse eine Implementierung getestet werden soll. So sind zum Beispiel einige Testfälle nur für die Konformitätsklasse *ECC2* relevant, während andere Testfälle für alle Klassen relevant sind. Ebenso hängt die Anzahl auszuführender Testfälle von den Scheduler-Strategien ab, die eine Implementierung unterstützt. Zuletzt hängt die Anzahl der Testfälle noch davon ab, ob eine Implementierung lediglich Standard-Rückgabewerte oder erweiterte Rückgabewerte verwendet [gro99].

4.3 Umsetzung

Im Rahmen dieser Arbeit wurden sämtliche MODISTARC Testfälle für OSEK/OS mit den nachfolgend genannten Einschränkungen implementiert.

Einschränkung 1

MODISTARC definiert auch Testfälle zu verschachtelten Interrupts. Verschachtelte Interrupts sind optional in OSEK/OS und werden von der Portierung nicht unterstützt. Daher wurden diesen Testfälle nicht realisiert. Weiterhin sind auch Testfälle zu den Systemaufrufen `EnableInterrupt`, `DisableInterrupt` und `GetInterruptDescriptor` spezifiziert. ErikaOS unterstützt diese nicht, daher wurde auch diese Testfälle nicht realisiert.

Einschränkung 2

Für alle Testfälle wurde nur die Variante mit erweitertem Rückgabewert realisiert. Dieses Vorgehen lässt sich damit begründen, dass die Rückgabewerte vom ErikaOS-Kern ermittelt werden. Je nach Konfiguration in der OIL-Datei gibt der ErikaOS-Kern einen Standard-Rückgabewert oder einen erweiterten Rückgabewert (mit Debuginformationen) an die Anwendung zurück. Ist der erweiterte Rückgabewert korrekt, kann davon ausgegangen werden, dass auch der Standard-Rückgabewert korrekt ist. Das Wiederholen der Testfälle mit Standard-Rückgabewerten würde in diesem Fall lediglich den ErikaOS-Kern also den von der Portierung unabhängigen Code von ErikaOS testen.

Einschränkung 3

Für Testfälle, bei denen die Scheduler-Strategie offensichtlich keine Rolle spielt, wurde nur die nicht-präemptive Variante realisiert. Zum Beispiel der Testfall `ActivateTask` mit ungültiger *Task-ID*. Dieses Vorgehen lässt sich genauso wie oben damit begründen, dass das Wiederholen der Testfälle für andere Scheduler-Strategien lediglich den ErikaOS-Kern - also den von der Portierung unabhängigen Code von ErikaOS - testen würde.

Die Implementierung der Testfälle ist im Verzeichnis *erika/testcases* zu finden. Jeder Testfall stellt eine eigene ErikaOS-Anwendung dar. Die Testfälle der Testgruppen 1-5 können vollständig automatisiert ablaufen. Dazu wird nach Abschluss des Testfalls ein Reset über den Watchdog des AM1808

ausgelöst. Nach dem Neustart wird der nächste Testfall automatisch über U-Boot geladen und ausgeführt. Die Testfälle sind nach Konformitätsklassen sortiert in entsprechenden Unterverzeichnissen `testcases/bcc1`, `testcases/bcc2`, `testcases/ecc1` und `testcases/ecc2` abgelegt.

Die Testfälle der Testgruppe 6 müssen manuell ausgeführt werden. Dies liegt daran, dass einige Testfälle dieser Gruppe nicht terminieren und absichtlich in einer Endlosschleife enden. Daher kann nach Abschluss des Testfalls kein Reset über den Watchdog erfolgen. Diese Testfälle befinden sich ebenfalls nach Konformitätsklassen sortiert im Verzeichnis `erika/testcases/error_hook_os`.

4.4 Ergebnis

Alle umgesetzten OSEK-Konformitätstestfälle sind erfolgreich. Die Ergebnisse sind in der Datei `erika/modistarc_results` beschrieben. Diese ErikaOS Portierung ist damit in Bezug auf die Spezifikation [gro97] ein OSEK konformes Betriebssystem, mit Ausnahme von Einschränkung 1.

Um die vollständige Konformität zum alten OSEK-Standard [gro97] zu erreichen, müssen die fehlenden Systemaufrufe aus Einschränkung 1 ergänzt werden.

Um die vollständige Konformität zum aktuell gültigen OSEK-Standard [Oseb] zu erreichen, sind zu diesem Standard passende Konformitätstestfälle erforderlich. In diesem Fall müssen die fehlenden Systemaufrufe aus Einschränkung 1 nicht realisiert werden, weil diese im aktuellen Standard [Oseb] durch neue Systemaufrufe ersetzt worden sind. Diese neuen Systemaufrufe werden sowohl von ErikaOS als auch der Portierung unterstützt. Die Portierung stellt dazu die Funktionen `irq_enable` und `irq_disable` bereit. Diese wurden über eigene Testanwendungen validiert. Somit kann davon ausgegangen werden, dass die Portierung auch konform zum aktuellen Standard [Oseb] ist.

5

ZUSAMMENFASSUNG

Die vorliegende Arbeit beschreibt die Portierung des OSEK/AUTOSAR konformen Echtzeitbetriebssystems ErikaOS auf den ARM9 basierten EV3-Baustein des LEGO EV3 MINDSTORMS.

In Kapitel 2 erfolgte eine Einführung in OSEK/OS, eine Beschreibung des Aufbaus von ErikaOS sowie der Hardware des EV3-Bausteins.

In Kapitel 3 wurde die Entwicklungsumgebung vorgestellt. Hervorzuheben sind hier insbesonders die beiden folgenden Punkte. Erstens der im Rahmen dieser Arbeit entwickelte Deployment-Prozess. Dieser erlaubt es Software auf dem EV3-Baustein ablaufen zu lassen, in dem diese automatisiert über die serielle Schnittstelle geladen wird. Zweitens die zusätzliche Portierung von ErikaOS auf die von QEMU unterstützte Versatilepb-Hardware. Diese ermöglicht ARM9-Code über GDB und QEMU zu debuggen.

Weiterhin wurden in Kapitel 3 die Details der Implementierung beschrieben. Im Rahmen dieser Arbeit wurden alle von OSEK spezifizierten Konformitätsklassen implementiert. Die Implementierung lässt sich lediglich hinsichtlich folgender Punkte erweitern:

1. Es werden keine verschachtelten Interrupts unterstützt.
2. Es wird nur ein Hardware-Timer unterstützt.
3. Als Interruptquelle kann nur der Hardware-Timer verwendet werden.
4. Die Implementierung ist nicht hinsichtlich Performance optimiert .
5. Zur Ansteuerung von Sensoren muss direkt auf die dazugehörigen Hardwareregister zugegriffen werden.

Sollten die ersten drei Aspekte für ErikaOS-Anwendungen relevant sein, so muss die Portierung entsprechend erweitert beziehungsweise optimiert werden. Die fünfte Einschränkung stellt insofern kein Problem dar, weil es existierende Software mit Treibern für Sensoren gibt. Diese Treiber können direkt aus der ErikaOS-Anwendung aufgerufen werden.

In Kapitel 4 erfolgt die Evaluation der Portierung. Dazu wurden die standardisierten Testfällen MODISTARC implementiert und ausgeführt. Mit diesen Testfällen konnte die Konformität der ErikaOS-Portierung zur OSEK/OS Spezifikation [gro97] für alle Konformitätsklassen nachgewiesen

5 Zusammenfassung

werden. Ausgenommen sind die von ErikaOS nicht unterstützen Systemaufrufen EnableInterrupt, DisableInterrupt und GetInterruptDescriptor.

Das Ziel, ErikaOS für den LEGO EV3 MINDSTORMS zu portieren, wurde erreicht. Damit steht für LEGO EV3 MINDSTORMS ein OSEK-Echtzeitbetriebssystem zur Verfügung.

6

ANHANG

Kompileroptionen

- -marm: Es wird im *ARM*- und nicht *Thumbmodus* Code erzeugt
- -mabi: ABI (Application Binary Interface) wird festgelegt
- -march: Zielarchitektur wird festgelegt (hier für ARM9 die ARMv5te)
- -mno-thumb-interwork: Kein Moduswechsel zwischen *ARM*- zu *Thumbmodus*
- -isystem erlaubt die Nutzung libgcc [Lib], was wichtige Funktionen wie zum Beispiel Division bereitstellt
- -fno-common: Nicht initialisierte, globale Variablen in die BSS Sektion legen
- -msoft-float: Keine Instruktionen für die Fließkommazahleneinheit
- -fno-builtins: Keine „builtin-Funktionen“ von GCC nutzen, welche nicht explizit den Prefix „__builtin“ besitzen
- -ffreestanding: Anwendung läuft in freistehender Umgebung, wo möglicherweise keine Standardbibliotheken zur Verfügung stehen
- -nostdinc: Nicht nach Headerdateien in den Standardverzeichnissen suchen
- -fno-stack-protector: Den Stack nicht schützen. Normalerweise werden „Guardvariablen“ auf den Stack gelegt, welche vor dem Rücksprung aus einer Funktion geprüft werden

Linkeraufruf

```
1 arm-none-eabi-ld -o hello_world -e hello_world hello_world.o -g \
   -Ttext 0xC1000000 -L$(LIBGCCDIR) -lgcc
2 arm-none-eabi-objcopy -O binary hello_world hello_world.bin
```

Listing 6.1 – EV3-Baustein: Linkeraufruf für Hallo-Welt-Anwendung

Erstellen der „boot.scr“ Datei

Listing 6.2 zeigt den für das Erstellen der „boot.scr“ Datei relevanten Ausschnitt des Makefiles. Um die Adresse der Eintrittsfunktion herauszufinden, wird aus der ELF-Datei der Anwendung ein Assembler-Listing erzeugt (Zeile 1) und die Adresse der Funktion `hello_world` herausgesucht. Das Suchen dieser Funktion fällt im Makefile zusammen mit dem Erzeugen des `go` Befehls (Zeile 4). Die „boot.scr“ Datei muss mittels `mkimage` [ID10] mit einem U-Boot spezifischen Header versehen werden (Zeile 6).

```
1 arm-none-eabiobjdump -d hello_world > hello_world.asm
2
3 echo "fatload mmc 0 0xC1000000 hello_world.bin" > boot.cmd
4 echo "go 0x$(shell arm-none-eabi-objdump -d hello_world | grep \
5     'hello_world' | sed -n 1p | cut -d' ' -f1)" >> boot.cmd
6 mkimage -C none -A arm -T script -d boot.cmd boot.scr
```

Listing 6.2 – EV3-Baustein: Erzeugen von „boot.scr“ für Hallo-Welt-Anwendung

QEMU Debugging

Das Debuggen erfolgt, indem der von QEMU bereitgestellte GDB-Server angesprochen wird. QEMU muss so gestartet werden, dass die Anwendung nach dem Laden nicht ausgeführt wird. So kann sich GDB vor dem Start der Anwendung mit dieser verbinden. Der dafür genutzte QEMU-Aufruf ist in Listing 6.3 gezeigt. Das Argument „-S“ verhindert die Ausführung der Anwendung. Das Argumente „-s“ startet den GDB-Server der TCP-Port 1234 öffnet.

```
1 qemu-system-arm -M versatilepb -m 128M -nographic -kernel \
    application.bin -s -S
```

Listing 6.3 – Versatilepb: Starten des QEMU

Zum Starten des GDB wird der in Kapitel 6 gezeigte Aufruf genutzt. Dabei wird sich mit dem QEMU verbunden, indem ein initiales GDB-Kommando übergeben wird. Dieses kann auch manuell nach dem Starten des GDB eingegeben werden.

Nach dem Starten, sowie dem Verbinden vom QEMU und GDB, ist es möglich, die Anwendung

```
1 gdb-none-eabi application.elf --eval-command="target remote \
    localhost:1234"
```

wie gewohnt mit GDB zu debuggen. Lediglich das Starten der Anwendung mit dem GDB-Befehl `run` entfällt, da die Anwendung bereits im laufenden Zustand ist. Sie wird lediglich vom QEMU angehalten. Somit startet die Anwendung durch den GDB-Befehl `continue`.

RT-Druid Workaround

RT-Druid nutzt das Attribut, um in der generierten Headerdatei das Präprozessorsymbol __CPU_NAME__ zu hinterlegen. Darüber wird gesteuert, welche Headerdateien beim Übersetzen eingebunden werden sollen. So enthält zum Beispiel „ee.h“ Abfragen auf mögliche Werte von der __CPU_NAME__ (siehe Listing 6.4).

```

1 #ifdef __X86__
2 #include "cpu/x86/inc/ee_cpu.h"
3 #endif
4
5 #ifdef __AVR8__
6 #include "cpu/avr8/inc/ee_avr8_cpu.h"
7 #endif
8
9 #ifdef __CORTEX_MX__
10 #include "cpu/cortex_mx/inc/ee_cpu.h"
11 #endif

```

Listing 6.4 – ErikaOS: CPU Includes

Das Problem RT-Druid ohne Anpassungen für ARM9 nutzen zu können, wurde wie folgt gelöst. In der OIL-Datei wird für das Attribut „CPU_DATA“ stets der Cortex-Mx eingetragen. Zusätzlich wird in der OIL-Datei das ErikaOS spezifische Attribut EE_OPT = "__ARM9__" gesetzt. Dies führt dazu, dass RT-Druid in der generierten Headerdatei das Präprozessorsymbol "__ARM9__" hinterlegt. Damit bei Generierung keine Cortex-Mx-Header, sondern nur ARM9-Header eingebunden werden, wurden die Header wie „ee.h“ angepasst. Alle ARM9 spezifischen Dateien wurden über ein „#ifdef __ARM9__“ ergänzt und alle Cortex-Mx spezifischen Dateien wurde über ein „#ifndef __ARM9__“ ausgeschlossen (siehe Listing 6.5).

```

1 ...
2
3 #ifdef __ARM9__
4 #include "cpu/arm9/inc/ee_cpu.h"
5 #endif
6
7 #ifdef __CORTEX_MX__
8 #ifndef __ARM9__
9 #include "cpu/cortex_mx/inc/ee_cpu.h"
10 #endif
11 #endif

```

Listing 6.5 – ErikaOS: CPU angepasste Includes

Zusätzlich kann die ARM9 Portierung über die folgenden Parameter durch das Attribut „EE_OPT“ konfiguriert werden.

EE_OPT = "__AM1808__" wählt die MCU AM1808

VERZEICHNISSE

EE_OPT = "__VERSATILEPB__" wählt die MCU Versatilepb

EE_OPT = "__Mindstorms__" fügt Unterstützung für EV3-Baustein Hardware hinzu

Für jeden dieser Parameter wird von RT-Druid ein entsprechendes Präprozessorsymbol in der generierten Headerdatei eingefügt, welche von der Portierung berücksichtigt werden.

ABKÜRZUNGSVERZEICHNIS

AINTC	Arm Interrupt Controller
CPU	Central Processing Unit
GPIO	General Purpose Input/Output
MCU	Microcontroller Unit
SPI	Serial Peripheral Interface
OIL	OSEK Implementation Language
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
UART	Universal Asynchronous Receiver / Transmitter
MODISTARC	Methods and tools for the validation of OSEK/VDX - based Distributed Architectures
QEMU	Quick Emulator

ABBILDUNGSVERZEICHNIS

2.1 ErikaOS: Verzeichnisstruktur	16
2.2 ErikaOS: Übersetzungsprozess	17
3.1 EV3-Baustein: UART-Anschluss	22
3.2 EV3-Baustein: JTAG Pins	26
3.3 EV3-Baustein: Reset-Taster	27
3.4 ErikaOS Portierung: Verzeichnisstruktur	28

TABELLENVERZEICHNIS

3.1 EV3-Baustein: JTAG Pin Zuordnung	26
--	----

CODEVERZEICHNIS

2.1 OSEK/OS Beispielanwendung Minitask	6
2.2 OSEK/OS Beispielanwendung Interrupt	8
2.3 OSEK/OS Beispielanwendung Event	9
2.4 OSEK/OS Beispielanwendung Ressource	11
2.5 OSEK/OS Beispielanwendung Alarm	12
2.6 OSEK/OS Beispielanwendung Hookfunktionen	13
2.7 OIL Beispiel: Objektreferenzen	14
 example_sources/hello_world/hello_world.c	22
3.1 ARM9 bare-metal Hello World Code	22
3.2 EV3-Baustein: Übersetzeraufruf für Hallo-Welt-Anwendung	23
 6.1 EV3-Baustein: Linkeraufruf für Hallo-Welt-Anwendung	43
6.2 EV3-Baustein: Erzeugen von „boot.scr“ für Hallo-Welt-Anwendung	44
6.3 Versatilepb: Starten des QEMU	44
6.4 ErikaOS: CPU Includes	45
6.5 ErikaOS: CPU angepasste Includes	45

LITERATUR

- [Am1] *AM1808 JTAG-Dongle.* <https://www.ti.com/tool/TMDSEMU200-U>, <https://www.ti.com/tool/TMDSEMU560V2STM-U> and <https://www.ti.com/tool/TMDSEMU560V2STM-UE>. (Besucht am 09.09.2022).
- [ARM] ARM. *PrimeCell UART (PL011) Technical Reference Manual*. URL: <https://documentation-service.arm.com/static/5e8e3655fd977155116a9042> (besucht am 09.09.2022).
- [ARM02] ARM. *ARM9EJ-S ATechnical Reference Manual*. 2002. URL: <https://documentation-service.arm.com/static/5e8e476f88295d1e18d3aecb> (besucht am 09.09.2022).
- [ARM04a] ARM. *ARM Architecture Reference Manual*. 2004. URL: <https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a> (besucht am 09.09.2022).
- [ARM04b] ARM. *ARM Dual-Timer Module (SP804) Revision: r1p0 Technical Reference Manual*. 2004. URL: <https://documentation-service.arm.com/static/5e8e2102fd977155116a4aef> (besucht am 09.09.2022).
- [ARM04c] ARM. *PrimeCell Vectored Interrupt Controller (PL190) Revision: r1p2 Technical Reference Manual*. 2004. URL: <https://documentation-service.arm.com/static/5e8e3604fd977155116a8e89> (besucht am 09.09.2022).
- [ARM08] ARM. *ARM926EJ-S Technical Reference Manual*. 2008. URL: <https://documentation-service.arm.com/static/5e8e3d1088295d1e18d3a9b2> (besucht am 09.09.2022).
- [ARM16] ARM. *ARM Compiler Version 5.06 armasm User Guide*. 2016. URL: <https://documentation-service.arm.com/static/5ea074249931941038de698a> (besucht am 09.09.2022).
- [ARM22] ARM. *Procedure Call Standard for the Arm® Architecture*. 2022. URL: <https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs32.pdf> (besucht am 09.09.2022).
- [ATU] ATUOSAR. *Autosar Homepage*. URL: <https://www.autosar.org/> (besucht am 09.09.2022).
- [Ava+15] Arianna Avanzini u. a. „Integrating Linux and the real-time ERIKA OS through the Xen hypervisor“. In: *10th IEEE International Symposium on Industrial Embedded Systems SIES*. IEEE; 2015, S. 1–7. ISBN: 978-1-4673-7711-9. DOI: 10.1109/SIES.2015.7185063. URL: <https://www.tib.eu/de/suchen/id/ieee%3Aef00f5c5fa6fa47158e395755630a407921c>
- [Cke] C-Kermit 9.0. URL: <http://www.columbia.edu/kermit/ck90.html> (besucht am 09.09.2022).
- [CW19] Dietrich Christian Werner. „Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System“. PhD dissertation. Gottfried Wilhelm Leibniz Universität Hannover, 2019. URL: www.sra.uni-hannover.de/Publications/2019/dietrich_19_phd.pdf.

- [Dil21] Diligent. *JTAG HS3 Reference Manual*. 2021. URL: <https://digilent.com/reference/programmers/jtag-hs3/reference-manual> (besucht am 09.09.2022).
- [Eria] Erika Educational related links. URL: http://erika.tuxfamily.org/old_erika_website/links.html (besucht am 09.09.2022).
- [Erib] ErikaOS Dateien. URL: https://svn.tuxfamily.org/viewvc.cgi/erika_erikae/repos/ee/trunk/ee/ (besucht am 09.09.2022).
- [Ev3] ev3dev is your EV3 re-imagined. URL: <https://www.ev3dev.org/> (besucht am 09.09.2022).
- [Evi14] Evidence. Porting ERIKA Enterprise and RT-Druid to a new microcontroller. 2014. URL: http://erika.tuxfamily.org/wiki/index.php?title=Porting_ERIKA_Enterprise_and_RT-Druid_to_a_new_microcontroller (besucht am 09.09.2022).
- [Evi21] Evidence. Erika Enterprise. 2021. URL: <https://www.evidence.eu.com/products/erika-enterprise.html> (besucht am 09.09.2022).
- [Fou] Eclipse Foundation. Eclipse Entwicklungsumgebung. URL: <https://www.eclipse.org/> (besucht am 09.09.2022).
- [Gai] Paolo Gai. Erika3 Is a real-time kernel certified OSEK/VDX compliant! URL: <https://www.erika-enterprise.com> (besucht am 09.09.2022).
- [gro04a] OSEK group. OSEK/VDX Binding Specification Version 1.4.2. 2004. URL: <https://www.osek-vdx.org/mirror/Binding142.pdf> (besucht am 09.09.2022).
- [gro04b] OSEK group. OSEK/VDX System Generation OIL: OSEK Implementation Language Version 2.5. 2004. URL: <https://www.osek-vdx.org/mirror/oil25.pdf>.
- [gro05] OSEK group. OSEK/VDX Operating System Version 2.2.3. 2005. URL: <https://www.osek-vdx.org/mirror/os223.pdf> (besucht am 09.09.2022).
- [gro97] OSEK group. OSEK/VDX Operating System - Version 2.0 Revision 1. 1997.
- [gro99] OSEK group. OSEK/VDX OS Test Plan. 1999. URL: https://www.osek-vdx.org/mod_20/ostestplan20.pdf (besucht am 09.09.2022).
- [HRP14] Uwe Hentschel, Daniel Richter und Andreas Polze. Embedded operating system projects, Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Potsdam, 2014. URL: <https://www.tib.eu/de/suchen/id/TIBKAT%3A818344652>.
- [Hö+18] Nils Hölscher u. a. „Examining and Supporting Multi-Tasking in EV3OSEK“. In: 2018, S. 25–30. URL: <https://ospert18.ittc.ku.edu/proceedings-ospert2018.pdf>.
- [ID10] Nobuhiro Iwamatsu und Wolfgang Denk. mkimage Manpage. 2010. URL: <https://manpages.ubuntu.com/manpages/bionic/man1/mkimage.1.html> (besucht am 09.09.2022).
- [Ins11] Texas Instruments. STARTERWARE-SITARA StarterWare for ARM based TI Sitara-Processors. 2011. URL: <https://www.ti.com/tool/STARTERWARE-SITARA> (besucht am 09.09.2022).
- [Ins14] Texas Instruments. AM1808 ARM Microprocessor. 2014. URL: <https://www.ti.com/lit/ds/symlink/am1808.pdf> (besucht am 09.09.2022).
- [Ins16] Texas Instruments. AM1808/AM1810 Sitara ARM Microprocessor Technical Reference Manual. 2016. URL: <https://www.ti.com/lit/ug/spruh82c/spruh82c.pdf> (besucht am 09.09.2022).
- [LEG] LEGO. LEGO Mindstorms EV3. URL: <https://www.lego.com/de-de/product/lego-Mindstormss-ev3-31313> (besucht am 09.09.2022).
- [LEG13] LEGO. LEGO Mindstorms EV3 Hardware Developer Kit. 2013. URL: <https://education.lego.com/de-de/product-resources/Mindstormss-ev3/downloads/developer-kits> (besucht am 09.09.2022).

- [Lej] *nxtOSEK/JSP*. 2013. URL: <http://lejos-osek.sourceforge.net/> (besucht am 09.09.2022).
- [Lib] *The GCC low-level runtime library*. URL: <https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html> (besucht am 09.09.2022).
- [Ope] OpenOCD. *Open On-Chip Debugger*. URL: <https://openocd.org/> (besucht am 09.09.2022).
- [Osea] *OSEK/VDX MODISTARC*. URL: https://www.osek-vdx.org/whats_modistarc.html (besucht am 09.09.2022).
- [Oseb] *Road vehicles - Open interface for embedded automotive application - Part3: OSEK/VDX Operating System (OS)*. Standard. International Organization for Standardization, 2005.
- [QEM] QEMU. *qemu Homepage*. URL: <https://www.qemu.org/> (besucht am 09.09.2022).
- [Sol13] Xander Soldaat. *EV3: Creating a Console Cable*. 2013. URL: <http://botbench.com/blog/2013/08/15/ev3-creating-console-cable/> (besucht am 09.09.2022).
- [TBP16] Andrew S. Tanenbaum, Herbert Bos und Katharina Pieper. *Moderne Betriebssysteme, IT-Informatik*. Hallbergmoos/Germany: Pearson; 2016. ISBN: 386894270X, 9783868942705. URL: <https://www.tib.eu/de/suchen/id/TIBKAT%3A855592095>.
- [Uar] *UART Dongle*. URL: https://www.amazon.de/gp/product/B01N9RZK6I/ref=ppx_yo_dt_b_asin_title_004_s00?ie=UTF8&psc=1 (besucht am 09.09.2022).
- [Uboa] *U-Boot Github*. URL: <https://github.com/u-boot/u-boot> (besucht am 09.09.2022).
- [Ubob] *U-Boot » Command-line Parsing*. URL: <https://u-boot.readthedocs.io/en/latest/usage/cmdline.html> (besucht am 09.09.2022).
- [Ver] *Arm Versatile boards (versatileab, versatilepb*. URL: <https://qemu.readthedocs.io/en/latest/system/arm/versatile.html> (besucht am 09.09.2022).