



**Hochschule für Technik  
und Wirtschaft Berlin**

**University of Applied Sciences**

# **Anpassung eines Emulators zur Durchführung von Hardwaretreiber-Unit-Tests am Beispiel des F-DPU Prozessors der ESA Raumfahrtmission PLATO**

**Abschlussarbeit**

zur Erlangung des akademischen Grades

**Bachelor of Engineering**

an der

Hochschule für Technik und Wirtschaft Berlin

Fachbereich 1: Ingenieurwissenschaften - Energie und Information

Studiengang Computer Engineering

**Erstprüfer** Prof. Dr.-Ing. habil. Carsten Gremzow

**Zweitprüferin** Ulrike Witteck M.Sc.

Eingereicht von: Pauline Hergersberg

Matrikelnummer: 568413

Abgabe: 25. August 2021

## **Vorwort**

Diese Arbeit ist am Institut für Optische Sensorsysteme im Deutschen Zentrum für Luft- und Raumfahrt im Rahmen der European Space Agency (ESA) Raumfahrtmission PLANetary Transits and Oscillations of stars (PLATO) entstanden.

Das DLR ist das Forschungszentrum für Luft- und Raumfahrt der Bundesrepublik Deutschland. Es ist von der Bundesregierung mit der Planung und Umsetzung der deutschen Raumfahrtaktivitäten beauftragt. Das Institut für Optische Sensorsysteme ist Teil des Forschungsbereiches Weltraum im DLR und beschäftigt sich mit der Erforschung und Entwicklung von passiven und aktiven optischen Sensorsystemen, die in der Raumfahrt Einsatz zum Beispiel auf fliegenden Plattformen und Robotern finden.

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und weitere Geschlechteridentitäten sind dabei ausdrücklich eingeschlossen, soweit es für die Aussage erforderlich ist.

## **Kurzbeschreibung**

Softwareentwicklung in Raumfahrtprojekten erfordert von Entwicklern, den implementierten Code besonders ausführlich zu testen. Dabei werden sie vor die Herausforderung gestellt, dass die Zielhardware noch nicht oder nur in geringen Mengen zur Verfügung steht und dass bestimmte Fehlerfälle darauf nicht ausgelöst werden können. Eine mögliche Lösung für diese Probleme ist die Verwendung von Emulatoren.

Diese Arbeit untersucht den quelloffenen Emulator Quick Emulator (QEMU) und den kommerziellen Emulator Terma Emulator (TEMU) von Terma jeweils auf ihre Eignung für die Durchführung von Hardwaretreiber Unittests der PLATO Fast Data Processing Unit Anwendungssoftware.

Hierfür werden Kriterien definiert, auf die die ausgewählten Emulatoren untersucht werden. Diese Kriterien umfassen auf der einen Seite die Anpassung der Emulatoren auf das AHB Status Register der PLATO F-DPU. Ziel ist es auf den angepassten Emulatoren Unittests für die AHB Status Treiber der PLATO ASW auszuführen. Außerdem werden die Emulatoren auf die Möglichkeiten zum Einfügen von Fehlern, Einbettung in kontinuierliche Integration und Generierung von Daten für Testüberdeckung untersucht.

Die Untersuchung ergibt, dass TEMU alle Kriterien vollständig oder teilweise erfüllt, für die Nutzung und den entsprechenden Support allerdings Gebühren gezahlt werden müssen. Die Anpassbarkeit beschränkt sich außerdem auf selbst implementierte Geräte.

QEMU kann ebenfalls auf die F-DPU Hardware angepasst werden. Werkzeuge für die Fehlerinjektion und das Erfassen der Testüberdeckung sind in QEMU standardmäßig nicht implementiert. Da das Projekt quelloffen ist, können fehlende Funktionen selbst implementiert werden. Die Nutzung ist außerdem kostenfrei.

Die Auswertung der Untersuchung führt zu dem Ergebnis, dass beide Emulatoren prinzipiell für die Durchführung von Tests für die PLATO F-DPU ASW geeignet sind. Die Entscheidung für einen der Emulatoren hängt von der Gewichtung der Kriterien Kosten, Verfügbarkeit und Anpassbarkeit ab.

# Inhaltsverzeichnis

1.	Einleitung . . . . .	2
2.	Hintergrund . . . . .	4
2.1.	PLATO Mission . . . . .	4
2.2.	Softwareentwicklung in Raumfahrtmissionen . . . . .	7
3.	Theoretische Grundlagen von Emulatoren . . . . .	10
3.1.	Gegenüberstellung von Emulatoren und Simulatoren . . . . .	10
3.2.	Allgemeine Funktionsweise von Emulatoren . . . . .	11
3.3.	Binäre Übersetzung von Programmen in Emulatoren . . . . .	11
3.4.	Auswahl von Emulatoren für die Untersuchung . . . . .	13
3.5.	Verwandte Arbeiten . . . . .	14
4.	Konzept zur Untersuchung von Emulatoren . . . . .	16
4.1.	Unterstützte Prozessorarchitekturen und -kerne . . . . .	16
4.2.	Implementierte Geräte und Erweiterbarkeit . . . . .	16
4.3.	Kontinuierliche Integration und Testüberdeckung . . . . .	17
4.4.	Fehlerinjektion . . . . .	17
4.5.	Kosten und Support . . . . .	17
5.	Untersuchung und Anpassung von Emulatoren auf die PLATO Hardware . . . . .	18
5.1.	QEMU . . . . .	19
5.2.	TEMU . . . . .	21
5.3.	Überwundene Schwierigkeiten bei der Umsetzung . . . . .	23
6.	Auswertung . . . . .	26
6.1.	Unterstützte Prozessorarchitekturen und -kerne . . . . .	26
6.2.	Implementierte Geräte und Erweiterbarkeit . . . . .	26
6.3.	Kontinuierliche Integration und Testüberdeckung . . . . .	26
6.4.	Fehlerinjektion . . . . .	27
6.5.	Kosten und Support . . . . .	27
6.6.	Zusammenfassung der Untersuchungsergebnisse . . . . .	28
7.	Fazit und Ausblick . . . . .	30
	Eidesstattliche Erklärung . . . . .	32
	Anhang . . . . .	36
A.	TEMU . . . . .	37
B.	QEMU . . . . .	41

# Abbildungsverzeichnis

1.	Beispielhafte Transitlichtkurve . . . . .	5
2.	Mögliche Beobachtungsfelder von PLATO . . . . .	6
3.	3D Modell des PLATO Satelliten . . . . .	6
4.	Ausführen der AHB Status Treiber Unittests in QEMU . . . . .	20
5.	Integration von QEMU in Jenkins . . . . .	20
6.	Ausführen der AHB Status Unittests in TEMU . . . . .	22
7.	Ändern von Registerwerten während der Emulation in TEMU . . . . .	23
8.	PSR Register eines SPARC V8 Prozessors . . . . .	24
9.	Trap 0x06 in QEMU . . . . .	25

## Quellcodeverzeichnis

1.	Event-Loop eines CPU-Emulators in Pseudocode [26, p. 71] . . . . .	11
2.	Registrierung des AHB Status Gerätes in QEMU . . . . .	19
3.	Einfügen des AHB Status Gerätes in das LEON3 Default Board . . . . .	22
4.	Aktivieren von Interrupts im PSR in QEMU . . . . .	24
5.	Implementierung des AHB Status Registers in TEMU . . . . .	37
6.	Implementierung des AHB Status Registers in QEMU . . . . .	41
7.	Kontinuierliche Integration von QEMU mit Jenkins . . . . .	44

# Akronyme

<b>AHB</b>	Advanced High-performance Bus
<b>APB</b>	Advanced Peripheral Bus
<b>ASW</b>	Anwendungssoftware
<b>CAN</b>	Controller Area Network
<b>DLR</b>	Deutsches Zentrum für Luft- und Raumfahrt
<b>ECSS</b>	European Cooperation for Space Standardization
<b>ESA</b>	European Space Agency
<b>F-DPU</b>	Fast Data Processing Unit
<b>GRTIMER</b>	General purpose Timer Unit
<b>IRQMP</b>	Interrupt Controller for Multiple Processors
<b>ISO</b>	Internationale Organisation für Normung
<b>MBUs</b>	Multiple Bit Upsets
<b>NASA</b>	National Aeronautics and Space Administration
<b>OBSW</b>	Onboard Software
<b>PLATO</b>	PLANetary Transits and Oscillations of stars
<b>PSR</b>	Prozessor Status Register
<b>QEMU</b>	Quick Emulator
<b>RAM</b>	Random Access Memory
<b>SEUs</b>	Single Event Upsets
<b>SPARC</b>	Scalable Processor Architecture
<b>TEMU</b>	Terma Emulator

### 1. Einleitung

Raumfahrtsoftware muss im Orbit jahrelang funktionieren und wartbar sein. Deswegen unterliegt diese extrem hohen Anforderungen. Um die korrekte Funktionsweise der Software zu garantieren, muss ausführlich und frühzeitig getestet werden.

Entwickler in Raumfahrtprojekten der ESA sind nach dem Entwurf und der Implementierung der Software dazu verpflichtet frühzeitig Unit-, Integrations- und Systemtests durchzuführen [24, p. 135]. Während mit Unittests implementierte Funktionen ausschließlich atomar betrachtet und getestet werden und somit keine vollständige Emulation der Zielhardware notwendig ist, liegt der Fokus von Integrationsstests auf der Interaktion zwischen zu testender Software und Zielhardware und stellt damit einen ganzheitlicheren Ansatz dar. Das frühe Ermitteln von Softwaremängeln ermöglicht es, Fehler zeitnah zu beseitigen und Kosten durch zu spätes Handeln zu vermeiden. Dabei ergeben sich für Entwickler und Tester vor allem zwei organisatorische Problemfelder:

- die Zielhardware steht noch nicht oder lediglich in geringen Mengen zur Verfügung und
- bestimmte Fehlerfälle können auf der Zielhardware nicht ausgelöst werden.

Aufgrund des zeitlichen Verlaufs eines Raumfahrtprojektes werden Softwaremodule zu einem Stand des Projektes entwickelt, zu dem die Hardwarespezifikationen zwar bereits feststehen, Labormodelle für die Durchführung von Tests aber noch nicht verfügbar sind.

Hinzu kommt, dass speziell entwickelte und produzierte Hardware für den Einsatz im Weltraum außerordentlich kostspielig ist. Daher wird für die Softwareverifikation nur eine geringe Menge von Labormodellen oder Evaluierungsboards bereitgestellt, mithilfe derer grundlegende Funktionalitäten getestet werden können. Testläufe auf der Zielhardware müssen aufgrund der begrenzten Menge verfügbarer Hardware unter Teammitgliedern koordiniert werden. Wird in Nachfolgeprojekten außerdem ein anderes Prozessmodell oder veränderte Hardware eingesetzt, können Evaluierungsboards nicht weiterverwendet werden. Als weiterer Aspekt kommt hinzu, dass das Testen von Fehlern mithilfe der Zielhardware teils nur umständlich, zum Beispiel durch das händische Erzeugen von Kurzschlüssen mittels eines Drahtes, durchgeführt werden kann.

Solche Fehler, wie sie vor allem im Weltall auftreten, können beispielsweise einzelne oder mehrere Bitflips, also durch Strahlung veränderte Bitwerte, oder auch Kurzschlüsse sein.

Eine mögliche Lösung für die oben beschriebenen Problemfälle stellt die Emulation des Zielsystems dar. Emulatoren werden eingesetzt, um Umgebungen oder Hardware abzubilden. Sie bieten die Möglichkeit, bestimmte Fehler- und Ausnahmefälle nachzustellen und ermöglichen es Entwicklern und Testern, das Verhalten von Software in zahlreichen Szenarien zu testen.

Emulatoren werden bereits in verschiedenen Forschungs- und Entwicklungsgebieten eingesetzt, so zum Beispiel in der Emulation von Photovoltaik-Generatoren für das Durchführen



von Laboruntersuchungen und -tests [9], in Netzwerkemulationen für die Performanzermittlung von Netzwerkprotokollen [19] sowie in der Emulation von Umgebungen für das Testen von Systemschnittstellen in der Luftfahrt [31].

Bereits bei Festlegung der Spezifikationen der Zielhardware kann ein Emulator entsprechend angepasst und erweitert werden. Unit- und Integrationstests können frühzeitig, ausführlich und in einer automatisierten Testumgebung integriert ausgeführt werden. Darüber hinaus kann ein Emulator für unterschiedliche Projekte auf die variierende Hardware angepasst werden. Somit werden Kosten und Zeit gespart.

In dieser Arbeit werden Kriterien definiert, die zur Untersuchung der Anpassbarkeit der Emulatoren Quick Emulator (QEMU) und Terma Emulator (TEMU) zum Testen von Raumfahrtsoftware genutzt werden. Dazu werden die Emulatoren an die Anwendungssoftware (ASW) der Fast Data Processing Unit (F-DPU) der ESA Raumfahrtmission PLATO angepasst, um Unittests für den Treiber eines Registers des PLATO F-DPU Prozessors auszuführen. Im Anschluss wird die Eignung anhand der ausgesuchten Kriterien bewertet.

Die Arbeit beginnt zunächst mit der Vorstellung der PLATO Mission sowie der Softwareentwicklung in Raumfahrtmissionen als Kontext zu Kapitel 2.

Die Theorie wird im nachfolgenden Kapitel 3 mit der Abgrenzung von Emulation zu Simulation, der Erläuterung grundlegender Funktionsweisen von CPU-Emulatoren, der Vorstellung der Emulatoren QEMU und TEMU und verwandter Arbeiten dargestellt.

Die definierten Kriterien zur Untersuchung der Emulatoren werden in Kapitel 4 beschrieben. In Kapitel 5 werden die Emulatoren in Hinblick auf die in Kapitel 4 erläuterten Kriterien untersucht, an die PLATO F-DPU ASW angepasst und mithilfe der angepassten Emulatoren Unittests ausgeführt.

Die Ergebnisse der Untersuchung werden in Kapitel 6 ausgewertet. Die Arbeit schließt mit einer Zusammenfassung der gewonnenen Erkenntnisse und einem Ausblick auf offene Arbeiten in Kapitel 7.

## 2. Hintergrund

In diesem Kapitel wird der Hintergrund der Arbeit erläutert. Dafür wird auf das Ziel und die grundlegenden Funktionsweisen der PLATO Mission und Softwareentwicklung in Raumfahrtmissionen eingegangen.

### 2.1. PLATO Mission

Die PLATO Mission ist eine von drei Medium Klasse Missionen des *Cosmic Vision* Programms der ESA und National Aeronautics and Space Administration (NASA) zur Entdeckung und Charakterisierung von Exoplaneten. Der Begriff *Exoplanet* bezeichnet einen Planeten, dessen Umlaufbahn nicht um unseren Heimatstern Sonne liegt [5].

Exoplaneten wurden Anfang der 1990er Jahre erstmals durch Bodenstationen entdeckt und beobachtet [5].

Bekannte Vertreter der Exoplaneten-Missionen sind die COROT und Kepler Missionen, wobei Ziel dieser Missionen vor allem das Entdecken möglichst vieler Exoplaneten war.

Ziel der PLATO Mission ist hingegen nicht nur das Entdecken von Planeten in der habitablen Zone ihres Sterns, sondern auch die Charakterisierung dieser Planeten und ihrer Sterne sowie die Erforschung der Entstehung solcher Planetensysteme.

Die Gesichtspunkte für die Charakterisierung [22] der Exoplaneten sind

- Masse und Radius für die Berechnung der mittleren Dichte,
- Atmosphäre und Orbit sowie
- Alter

und für deren Sterne

- Masse und Radius,
- Typ, Leuchtkraft und Aktivität sowie
- Alter.

Um diese Parameter zu erfassen, werden verschiedene Methoden der Datenerfassung verwendet.

Eine dieser Methoden ist die Astroseismologie oder Oszillation von Sternen, bei der sogenannte Sonnenbeben durch das Messen der Helligkeit eines Sterns über Zeit beobachtet werden können. Durch Fourier-Transformation der Frequenzen können Parameter des Sterns wie Zusammensetzung, Alter und Masse festgestellt werden [22].

Ein weiteres verwendetes Verfahren ist die Transitmethode, bei der die Lichtkurve eines Sterns aufgezeichnet wird. Wie in Abbildung 1 dargestellt, fällt die Lichtkurve, wenn sich

## 2. HINTERGRUND

---

ein Planet zwischen Beobachter und observierten Stern bewegt. Anhand des gemessenen Lichtabfalls zweier Planetendurchgänge kann der Radius und die Umlaufgeschwindigkeit des Himmelskörpers berechnet werden [22].

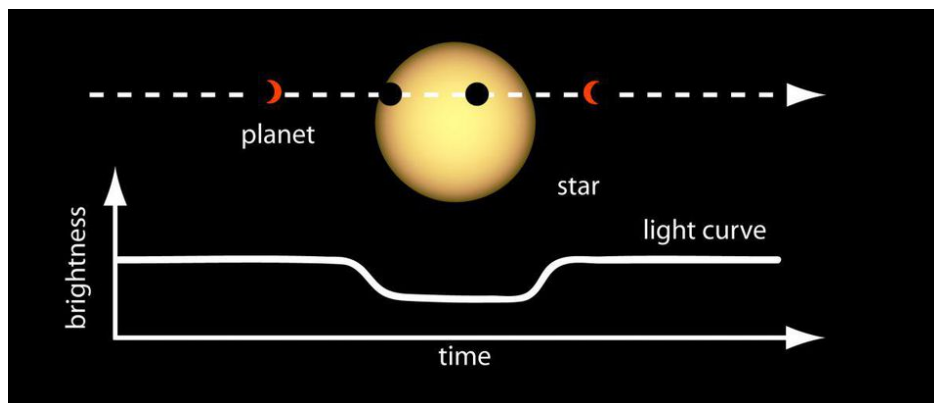


Abbildung 1: Beispielhafte Transitlichtkurve [25]

Durch Analyse des Lichtspektrums kann außerdem die Zusammensetzung des Planeten untersucht und so festgestellt werden, um was für eine Art von Planet es sich handelt. So wird bei der Atmosphäre von Gasplaneten ein anderes Lichtspektrum gemessen als bei der Atmosphäre von Gesteinsplaneten wie der Erde [22].

Unterstützend zu den bereits genannten Methoden werden von der Erde aus Beobachtungen unter Verwendung der Radialgeschwindigkeitsmethode oder Doppler-Spektroskopie durchgeführt. Um die Masse eines Exoplaneten zu ermitteln, wird das Lichtspektrum des Sterns analysiert und anhand der Rot- und Blauverschiebung die Geschwindigkeit bestimmt, mit der sich der Stern um das Massezentrum des Zwei-Körper-Systems Planet-Stern bewegt [36].

Um ein besonders großes Beobachtungsfeld zu ermöglichen, wird sich PLATO auf einem Orbit um den Lagrange-Punkt 2 (L2)<sup>1</sup> befinden. Dabei befinden sich Erde und Sonne stets auf der Rückseite des Satelliten, wodurch die ununterbrochene Stromversorgung durch Solarenergie gewährleistet ist [22].

Die in Abbildung 2 sichtbaren Beobachtungsfelder zeigen einen möglichen Beobachtungsverlauf von PLATO, wobei deren genaue Positionen noch nicht festgelegt wurden. Für Vergleiche eingezeichnet sind die Beobachtungsfelder von Kepler und COROT. Zwei Bereiche, in Abbildung 2 rot dargestellt, werden über einen langen Zeitraum von zwei bis drei Jahren, die sogenannte *Long-Duration Observation* Phase beobachtet. In dieser Zeit sollen im beobachteten Bereich Planeten in der habitablen Zone ihres Sterns gesucht werden. Da für die Transitmethode zwei Planetendurchgänge benötigt werden, dauert diese Phase mindestens zwei Jahre [1, p. 62]. Alle weiteren Beobachtungsfelder werden für einen Zeit-

---

<sup>1</sup>L2 ist einer jener Punkte, in denen die Gravitationskräfte von Sonne und Erde im rotierenden Bezugssystem gerade durch die Zentrifugalkraft kompensiert werden, sodass der Satellit in diesem Bezugssystem ruht.

## 2. HINTERGRUND

---

raum von zwei bis fünf Monaten beobachtet in der sogenannten *Step-and-Stare Observation* Phase. In dieser Phase sollen weitere Transits der in der Long-Duration Observation Phase entdeckten Planeten beobachtet werden. Außerdem werden dabei Informationen über die entsprechenden Sternensysteme und ihre Entstehung gesammelt [1, p. 62].

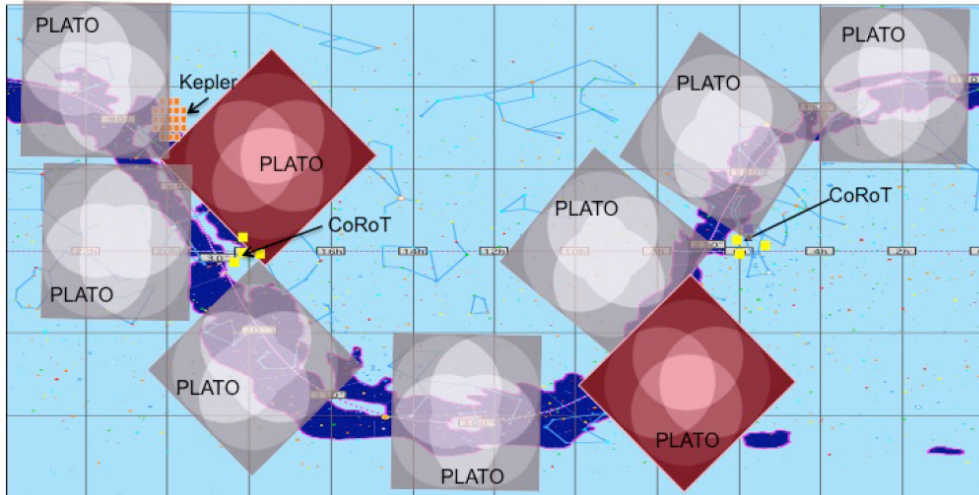


Abbildung 2: Mögliche Beobachtungsfelder von PLATO [28]

PLATO wird für möglichst präzise Kategorisierungen besonders helle Sterne beobachten. Der Satellit besteht daher, anders als bei bisherigen Missionen, nicht aus einem einzigen großen Teleskop sondern aus 24 normalen Kameras und zwei sogenannten *Fast-Kameras*. Abbildung 3 zeigt ein Modell des PLATO Satelliten. Darauf ist zu erkennen, dass die normalen Kameras in vier Gruppen à sechs Kameras angeordnet sind. Sie erfassen Bilder mit einer Ausleseverzögerung von 25 Sekunden.

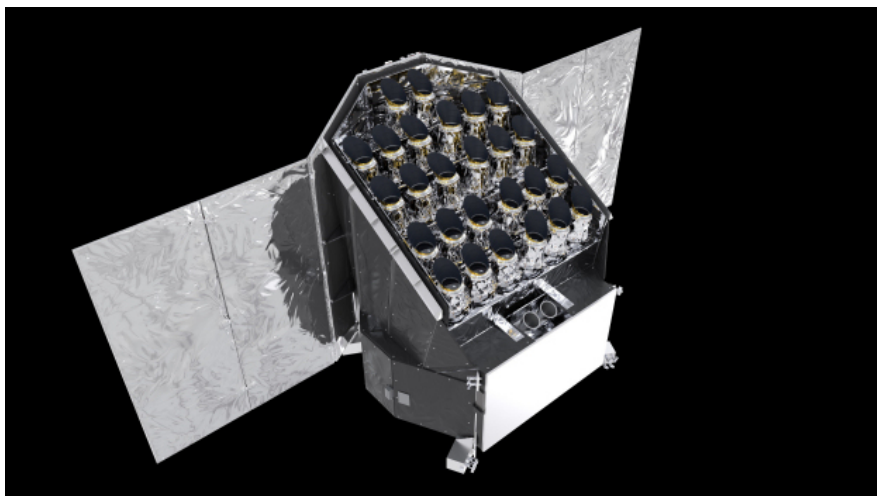


Abbildung 3: 3D Modell des PLATO Satelliten [13]

Im Gegensatz zu den normalen Kameras liefern die Fast-Kameras Bilddaten mit einer Ausleseverzögerung von 2,5 Sekunden [1, p. 69]. Aufgabe der Fast-Kameras ist die Betrachtung hellerer Sterne zur Feinausrichtung des Satelliten. Dabei ist eine Kamera mit einem Blaufilter und die andere mit einem Rotfilter ausgestattet, um Beobachtungen von Sternen in zwei Spektralbändern zu ermöglichen [1, p. 73].

Zu jeder Fast-Kamera gibt es eine F-DPU, die mittels eines Feinausrichtungsalgorithmus anhand der aufgenommenen Bilddaten die Ausrichtung des Teleskopes mit einer Genauigkeit von Millibogensekunden berechnet. Der Prozessor der F-DPU und die ASW werden in dieser Arbeit als Fallbeispiel betrachtet.

### 2.2. Softwareentwicklung in Raumfahrtmissionen

Die Entwicklung von Software für Raumfahrtmissionen folgt einem strengen Ablaufplan und unterliegt verschiedenen Standards. Diese Regelungen werden bedingt durch den organisatorischen Verlauf eines Raumfahrtprojektes.

Die Entwicklung verschiedener Software-Komponenten wird häufig von unterschiedlichen Institutionen durchgeführt. Die PLATO Mission beispielsweise wird durch die ESA geplant und administriert. Die Onboard Software (OSW) und Hardware wird dabei nicht nur durch die ESA, sondern viele unterschiedliche Institutionen, wie dem Deutschen Zentrum für Luft- und Raumfahrt (DLR), europaweit entwickelt [29]. Die Verteilung von Projektabschnitten auf unterschiedliche ausführende Institutionen erfordert präzises Planen und gemeinsame Projekt- und Programmierrichtlinien.

Die Entwicklung von OSW wird daher in einzelne Projektstufen eingeteilt.

Der erste Schritt in der Entwicklung der OSW ist die *Functional Analysis*, die funktionale Softwareanalyse. Die Aufgaben der OSW beinhalten das Verarbeiten von Messdaten, das Steuern der Missions-Hardware und Behandeln von Fehlern. Die Anforderungen an die OSW müssen daher während der ersten Missionsstufen von Wissenschaftlern gemeinsam mit Entwicklern erarbeitet und in einem sogenannten *Funktionsbaum* festgehalten werden [12, p. 130].

Dieser beinhaltet die einzelnen Funktionen geordnet nach Funktionsmodi des Satelliten [12, p. 131].

Auf Grundlage des Funktionsbaums werden in der *Software Requirements Definition* Phase detaillierte Softwareanforderungen definiert. Sie beschreiben in kurzen Sätzen die Funktionsweise der Software und allgemeine Anforderungen wie Aufbau, Performanz, Programmierrichtlinien und Schritte für die Verifizierung [12, p. 132].

Die definierten Anforderungen werden in nachfolgenden Schritten herangezogen und für die Implementierung und Verifikation verwendet [12, p. 135].

In der folgenden *Software Design* Phase werden die einzelnen zu implementierenden Funktionen beschrieben. Dabei hängt die verwendete Methode für die Beschreibung von dem Level der Implementierung und der verwendeten Programmiersprache ab. Häufig verwendete Programmiersprachen in der Raumfahrt sind Assembler, High Level Assembler (HAL), JOVIAL, Ada, C und C++ [12, p. 135]. Eine bekannte Methode für das Beschreiben des Softwaredesigns ist die Unified Modeling Language (UML).

Die vorletzte Phase *Software Implementation and Coding* beinhaltet die Umsetzung des in den vorherigen Phasen beschriebenen Entwurfs in Code. Unterschiedliche Coding-Konventionen und die Abstraktionsebene des Entwurfs bestimmen dabei den Aufwand der Implementierung [12, p. 148].

Die *Software Verification and Testing* Phase stellt den letzten Abschnitt dar. Sie beinhaltet sowohl das Testen der entwickelten Software, als auch die Überprüfung, ob die definierten Softwareanforderungen erfüllt werden.

Die in Testverfahren verwendeten Nachbildungen der Hardware reichen dabei von reiner Software oder Prüfständen bis zu vollständig funktionsfähigen Nachbauten der Hardware [12, p. 149].

Die in den oberen Phasen beschriebenen Entwurfs-, Implementierungs- und Testschritte erfolgen nach Standards, die die Richtlinien für selbige vorgeben. Dabei existieren unterschiedliche Standards nach denen vorgegangen wird.

Für die europäische Raumfahrt werden die ECSS Standards durch die European Cooperation for Space Standardization (ECSS) herausgegeben. Die ECSS ist eine Kommission innerhalb der ESA bestehend aus Vertretern verschiedener internationaler Institute und Firmen [12, p. 167].

Ein weiterer international anerkannter Standard sind die *Aeronautical Software Standards DO178B*, herausgegeben durch die Radio Technical Commission for Aeronautics. Der DO178B Standard soll „[...] sicherstellen, dass die Software in der Luft die für den Einsatz in einer sicherheitsrelevanten Anwendung erforderliche Integrität aufweist“ [30, p. 100].

Der durch das Einhalten von Softwarestandards entstehende Mehraufwand in der Entwicklung wird vor allem auch durch die Kategorisierung in Gefährdungsstufen beeinflusst.

Software wird dabei, abhängig davon wie kritisch ihr Ausfall für die Mission wäre, in eine Gefährdungsstufe eingeordnet [12, p. 168]. In unterschiedlichen Standards existieren verschiedene Aufstellungen dieser Stufen. Die Gefährdungsstufen des ECSS Standards ECSS-Q-ST-80C grenzen Software wie folgt ab [12, p. 169]:

## 2. HINTERGRUND

---

### **Stufe A**

Mit Stufe A wird Software bezeichnet, deren Ausfall oder nicht korrekte Ausführung katastrophale Folgen hat, z.B. einen Verlust der Mission.

### **Stufe B**

Mit Stufe B wird Software bezeichnet, deren Ausfall oder nicht korrekte Ausführung kritische Folgen hat, z.B. die Gefährdung der Mission.

### **Stufe C**

Mit Stufe C wird Software bezeichnet, deren Ausfall oder nicht korrekte Ausführung wesentliche Folgen hat.

### **Stufe D**

Mit Stufe D wird Software bezeichnet, deren Ausfall oder nicht korrekte Ausführung geringfügige oder vernachlässigbare Folgen hat.

## 3. Theoretische Grundlagen von Emulatoren

In diesem Abschnitt werden der Begriff Emulator und die Grundlagen von CPU-Emulatoren erläutert. Außerdem werden die Emulatoren QEMU und TEMU und zwei verwandte Arbeiten vorgestellt.

### 3.1. Gegenüberstellung von Emulatoren und Simulatoren

Bevor in diesem Abschnitt näher auf die Funktionsweise von CPU-Emulatoren eingegangen werden kann, werden der Begriff Emulator und seine Abgrenzung zum Begriff Simulator näher erläutert.

Es kommt vor, dass beide Begriffe in der technischen Informatik synonym verwendet werden. Jedoch können sie anhand unterschiedlicher Kriterien voneinander differenziert werden. In Anlehnung an die Arbeit von Jens-Martin Loebel [26] wird die Betrachtung der Etymologie und der Zielsetzung beider Herangehensweisen als Ansatz verwendet.

Die Wortherkunft des Begriffs Simulation liegt im lateinischen *simulare* [11], das mit *nachbilden* oder *ähnlich machen* übersetzt werden kann. Die Definition der Internationalen Organisation für Normung (ISO) zu Simulator beschreibt diesen als „Gerät, Computerprogramm oder System, das sich wie ein bestimmtes System verhält oder arbeitet, wenn es eine Reihe von kontrollierten Eingaben erhält“ [23]. Das bedeutet, dass ein Simulator nur unter bestimmten Bedingungen in einer kontrollierten Umgebung das Original ersetzen kann. Dies kann also auch bedeuten, dass nicht das gesamte System, sondern nur ein bestimmter, für die Anwendung relevanter Teil des ursprünglichen Systems nachgebildet wird.

Auf der anderen Seite findet sich die Herkunft des Wortes Emulation laut Duden [10] im lateinischen Wort *aemulari*, das soviel wie *wetteifern* bedeutet. Daneben lässt sich die Definition der ISO betrachten, die Emulatoren als „Gerät, Computerprogramm oder System, das dieselben Eingaben annimmt und dieselben Ausgaben erzeugt wie ein bestimmtes System“ [23], bezeichnet.

Werden beide Aussagen kombiniert, ergibt sich daraus, dass ein Emulator versucht, so gut wie möglich sein nachzubildendes Original zu imitieren. Im Zusammenhang mit Hardware heißt das, dass, je nach Emulationsgrad, nicht nur Schnittstellen, sondern beispielsweise auch Speicherstrukturen und systeminterne Kommunikationswege nachgebildet werden [35, p. 2]. Emulatoren können in Form von Software, aber auch Hardware, das Verhalten anderer Hardware imitieren, diese Form der Emulatoren wird als *In Circuit Emulator* (ICE) bezeichnet [23]. In dieser Arbeit wird ausschließlich auf Software-Emulatoren eingegangen, daher wird der Begriff Emulator synonym für Software-Emulator verwendet.

Für diese Arbeit werden ausschließlich Emulatoren betrachtet, da die ausgewählte Software für Unittests und Integrationstests geeignet sein soll.

Da für Integrationstests das Verhalten der gesamten Software auf der Hardware getestet wird,



ist ein Simulator nach der oben beschriebenen Definition dafür nicht geeignet.

Die allgemeine Funktionsweise von CPU-Emulatoren lässt sich anhand des Aufbaus und der binären Übersetzung erläutern.

#### 3.2. Allgemeine Funktionsweise von Emulatoren

Jens-Martin Loebel beschreibt in seiner Arbeit [26] die allgemeine Funktionsweise von Emulatoren.

Loebel stellt fest, dass Aufbau und Arbeitsweise eines CPU-Emulators sich an der generellen Struktur von Computersystemen, die nach der Von-Neumann-Architektur modelliert sind [26, p. 68], orientieren. In deren Zentrum steht neben dem herkömmlichen physischen Aufbau eines Computersystems auch der logische Arbeitsablauf innerhalb einer CPU, der *Fetch-Execute-Zyklus*.

Wie Quellcode 1 zeigt, werden in einem CPU-Emulator innerhalb des Event-Loops zunächst nacheinander anliegende, zeitlich gesteuerte Ereignisse, Interrupts und CPU-Events abgearbeitet. Anschließend beginnt die eigentliche Abarbeitung des Fetch-Execute-Zyklus, in dem zunächst der Programmzähler erhöht und der nächste Steuercode geladen und ausgeführt wird. Dieser Vorgang wird wiederholt, bis das Ende des geladenen Programms erreicht wird.

```
1   BOOL finished = false;
2 REPEAT
3   IF ( TIMED_EVENT ) ExecuteTimedEvents();
4   FOR ( interrupt IN openInterrupts[] )
5     ExecuteInterrupt(interrupt);
6   IF ( PROCESSOR_EVENT ) ExecuteProcessorEvent();
7
8   IP = IP++;
9   command = FetchCommand(IP);
10  opcode = command[0];
11  operands[] = DecodeAddresses(command[]);
12  SWITCH (opcode)
13    CASE 01: ExecuteCommand01(operands);
14    [...]
15  END SWITCH
16 UNTIL (finished == true)
```

Quellcode 1: Event-Loop eines CPU-Emulators in Pseudocode [26, p. 71]

#### 3.3. Binäre Übersetzung von Programmen in Emulatoren

Überträgt man die am Anfang des Kapitels erläuterte Definition von Emulatoren auf die Anwendung im Prozessorbereich, bedeutet das, dass ein CPU-Emulator in der Lage ist, ein Programm, das für eine bestimmte Prozessorarchitektur gebaut wurde, auf einem Prozessor einer anderen Prozessorarchitektur auszuführen. Diese Art der Übersetzung wird als binäre Übersetzung bezeichnet [35, p. 2].

Dabei besteht bei der CPU-Emulation das Problem, dass durch die binäre Übersetzung selbst mehr Rechenleistung vom Host-System, dem emulierenden System, erforderlich ist, als das Ziel-System, also das emulierte System, besitzt. Das Host-System muss also leistungsstärker sein als das Ziel-System. Die Komplexität des Ziel-Systems kann dabei eine zehnfach bis über hundertfach höhere Performanz des Host-Systems erfordern [27, p. 60].

Die Emulation eines Super Nintendo Entertainment Videospiele-Systems mit 3Mhz benötigt beispielsweise mindestens einen Pentium Prozessor mit 550Mhz für das Host-System, was ca. einem Faktor von 183 entspricht [35, p. 3].

Neben der Komplexität des Ziel-Systems spielt auch die Genauigkeit der binären Übersetzung eine Rolle für die erforderliche Rechenleistung des Ziel-Systems, die sich in folgende Abstufungen aufteilen lässt [27, p. 80]:

#### **Datenbus- und Pin-Genauigkeit**

Der Übersetzungsgrad der *Datenbus- und Pin-Genauigkeit* stellt die Abbildung der internen Hardwarestrukturen des Ziel-Systems dar und ist damit die aufwändigste Form der Emulation. Diese Abbildungspräzision ist nur in seltenen Fällen erforderlich, zum Beispiel wenn die für das Ziel-System auszuführende Software alle Hardwareeigenschaften bis auf ihre Grenzen ausnutzt [35, p. 3].

#### **Zyklengenauigkeit**

Bei der *Zyklengenauigkeit* werden Instruktionen auf dem Host-System mit einer gleichen oder annähernd gleichen Geschwindigkeit wie auf dem Ziel-System ausgeführt. Dadurch treten bei der Ausführung von Originalcode keine oder wenig Fehler auf, die auf Geschwindigkeitsunterschiede bei der Emulation zurückzuführen sind [35, p. 3].

#### **Instruktions- oder Taktgenauigkeit**

Die *Instruktions- oder Taktgenauigkeit* vernachlässigt den Aspekt der Geschwindigkeitspräzision und bildet lediglich die Instruktionen des Ziel-Systems ab. Unterschiedlich komplexe Funktionen können dabei in der gleichen Zeit abgeschlossen werden [35, p. 3].

#### **Blockgenauigkeit mittels dynamischer Rekompilierung**

Die schnellste Variante der bitweisen Übersetzung stellt die *Blockübersetzung* dar, wobei die Methode der dynamischen Rekompilierung angewendet wird.

Bei dieser werden Blöcke des für das Ziel-System gebauten Codes interpretiert, während der Ausführung wiederholend auftretende Codeblöcke gespeichert und wenn möglich optimiert. Statt diese Codeblöcke nun jedes Mal neu zu übersetzen, wird der ein Mal übersetzte und optimierte Block ausgeführt. Sollte innerhalb des optimierten Codes ein Fehler auftreten, wird die Maschine mithilfe des nicht optimierten und übersetzten Codes in den Zustand innerhalb des Blocks versetzt [35, p. 10].

Die Datenbus- und Pin-Genauigkeit stellt dabei die präziseste und damit rechenintensivste also langsamste und die Blockgenauigkeit die am wenigsten präzise, dafür aber effizienteste Form der Emulation dar.

#### 3.4. Auswahl von Emulatoren für die Untersuchung

In dieser Arbeit werden die Emulatoren QEMU und TEMU betrachtet.

Einer der ausgewählten Emulatoren ist die quelloffene Software QEMU.

QEMU wurde ursprünglich durch den Entwickler Fabrice Bellard entwickelt, veröffentlicht und seitdem durch ihn und weitere Entwickler erweitert und optimiert. QEMU ist unter der GNU Public License Version 2<sup>2</sup> lizenziert.

Mit QEMU setzte Bellard sich das Ziel einen Emulator zu programmieren, der nicht nur die Virtualisierung vielseitiger Ziel-Systeme, sondern dies auch auf unterschiedlichen Host-Systemen ermöglicht. Dabei kann die CPU-Architektur des Host- und des Ziel-Systems variieren. Neben unterschiedlichen Debugging Optionen für virtuelle Maschinen, bietet QEMU auch die Möglichkeit, eingebettete Systeme abzubilden und durch Implementierung eigener Geräte anzupassen [3, p. 41]. QEMU wurde für die Untersuchung ausgewählt, da dieser Emulator das prominenteste Beispiel für quelloffene Emulatoren ist. Außerdem war QEMU bereits Grundlage wissenschaftlicher Arbeiten für die CPU-Emulation in Raumfahrtprojekten. Zwei Beispiele für solche Arbeiten werden in Abschnitt 3.5 betrachtet.

Ein weiterer betrachteter Emulator ist TEMU. TEMU ist ein kommerzieller Emulator, der von der Firma Terma in Kooperation mit der NASA entwickelt wurde. TEMU ist auf die Emulation von in der europäischen Raumfahrt verwendeten Prozessorarchitekturen ausgerichtet. Der Fokus liegt dabei besonders auf der Vereinbarkeit vom präzisen Abbilden von Prozessorinstruktionen und der Performanz aktueller Prozessoren, inklusive Multiprozessor-systemen [20, p. 1]. Die Herausforderung der Echtzeitdarstellung wird dabei durch die höheren Taktfrequenzen modernerer LEON Mikroprozessoren verursacht. Bei einem LEON4 Prozessor liegt die Taktfrequenz beispielsweise bis 400MHz [2, p. 1] im Vergleich zum LEON3 FT mit einer Taktfrequenz von bis zu 100MHz [17]. TEMU realisiert dies über die Verwendung von LLVM, einer Sammlung modular verwendbarer Werkzeugketten [20, p. 2].

Neben TEMU sind auch andere CPU-Emulatoren von Scalable Processor Architecture (SPARC) Prozessoren verfügbar, wie beispielsweise TSIM von der Firma Gaisler oder laysim-leon3 vom Korea Aerospace Research Institute.

Die Evaluationslizenz von TSIM und laysim ermöglicht allerdings nur die Nutzung einiger Funktionen. Die Modellierung und das Einfügen eigener Geräte in Boardkonfigurationen ist nicht möglich. So kann nicht getestet werden, ob die Emulatoren auf die PLATO F-DPU angepasst werden können. Die Evaluationslizenz von TEMU schränkt die Funktionalität des Emulators nicht ein. Daher wird als weiterer Emulator TEMU betrachtet.

---

<sup>2</sup><https://www.gnu.org/licenses/old-licenses/gpl-2.0.txt>

#### 3.5. Verwandte Arbeiten

Wie in der Einleitung bereits erwähnt, werden Emulatoren bereits in unterschiedlichen Forschungsgebieten für die Durchführung von Tests eingesetzt. Auch in Raumfahrtprojekten finden Emulatoren Verwendung.

Die Arbeit von B. Carvalho et al. [6] beschreibt die Entwicklung von QERx, eines Emulators für LEON2 und ERC32 Prozessoren auf Basis von QEMU. Arbeitsschwerpunkt ist die akkurate Abbildung der Performanz im Jahr 2012 aktueller Prozessoren, die flexible Anpassung der simulierten Systemgeschwindigkeit sowie die Integration in Simulationsumgebungen.

QERx übernimmt dabei Prozessormodelle für SPARC, PROM und Random Access Memory (RAM) aus QEMU. Da QEMU auch andere Architekturen als die SPARC Architektur unterstützt, wurden nicht relevante Codebestandteile entfernt und auf das für QERx Wesentliche reduziert. Das Implementieren und Einfügen eigener Geräte in den Speicher ist, wie bei QEMU, möglich.

Außerdem wurde ein API entwickelt, das der des TSIM von Gaisler ähnelt. Auch für Performancevergleiche wird TSIM herangezogen. Die durchgeführten Vergleiche zeigen, dass QERx ähnlich leistungsstark ist wie TSIM. Deutliche Unterschiede zeigen sich bei den Floating-Point-Operationen, deren Nachbesserung zum Zeitpunkt der Veröffentlichung der wissenschaftlichen Arbeit noch ausstand.

Seit 2014 wurden keine weiteren Informationen zum aktuellen Arbeitsstand veröffentlicht und auf der Website des Herstellers Scisys<sup>3</sup> sind keine Informationen zu QERx verfügbar. Daher ist davon auszugehen, dass das Projekt eingestellt wurde. QERx kann in dieser Arbeit daher nicht berücksichtigt werden.

Jong-Wook Choi und Byeong-Gyu Nam [7] stellen *laysim* vor, einen Emulator für die zyklengenaue Emulation von LEON3 und ERC32 Prozessoren mit grafischer Benutzeroberfläche. Die Software soll für das Korea Aerospace Research Institute als Ersatz für die kostenpflichtige Software TSIM von Gaisler dienen.

Laysim baut, wie QERx, auf QEMU auf. Zum Zeitpunkt der Entstehung des Papers 2012 stand in QEMU bereits eine LEON3 Maschine zur Verfügung. Der Support für den AMBA Bus und die zugehörige Plug and Play Funktion fehlte allerdings noch.

Die Autoren fokussieren sich grundlegend auf die Implementierung des AMBA Busses und weiterer Systemkomponenten aus der Gaisler IP Core Bibliothek [16]. Die Modellierung des Systemverhaltens orientiert sich an der Funktionalität von TSIM.

Verglichen wurde die Performanz des Emulators mit TSIM von Gaisler. Dabei zeigte sich, dass der entwickelte Emulator bis zu sieben Mal schneller ist als TSIM. Die Modellierung eines Field Programmable Gate Array, Multiprozessor-Unterstützung und die Implementierung weiterer Gaisler IP Core Systemkomponenten werden als Ziele für Erweiterungen erwähnt.

---

<sup>3</sup><https://www.cgi.com/de/de>

### 3. THEORETISCHE GRUNDLAGEN VON EMULATOREN

---

Für diese Arbeit ist die Eignung der vorgestellten Software für die Durchführung von Unit-tests möglich. Die verfügbare Evaluierungslizenz unterstützt jedoch nicht das Entwerfen und Einfügen eigener Modelle, daher kann laysim in dieser Arbeit nicht berücksichtigt werden.

### 4. Konzept zur Untersuchung von Emulatoren

Zur Auswahl eines Emulators, der zum Testen von Raumfahrtsoftware eingesetzt werden soll, werden in diesem Kapitel Untersuchungskriterien definiert. Nach den Kriterien werden die Emulatoren auf ihre Eignung untersucht, um das Testen der Software in Raumfahrtprojekten zu erleichtern.

#### 4.1. Unterstützte Prozessorarchitekturen und -kerne

Für aktuelle Raumfahrtprojekte werden im Deutsches Zentrum für Luft- und Raumfahrt (DLR) unter anderem Prozessoren der SPARC Architektur eingesetzt, im speziellen Anwendungsfall PLATO ein LEON2 FT Prozessor. Für die Ausführung von Unittests kann ein LEON3 Kern verwendet werden, da die gravierendsten Unterschiede zu einem LEON2 Kern im AMBA Plug and Play Support und der Fähigkeit zu Multiprocessing zu finden sind [14]. Für zukünftige Missionen ist die ARM Prozessorarchitektur als Alternative möglich. Die Unterstützung derselben ist daher wünschenswert, stellt aber kein Ausschlusskriterium dar. Für Integrationstests der PLATO Mission sollte ein LEON2 Kern vorhanden oder implementierbar sein.

Für die in dieser Arbeit betrachteten Emulatoren ist die Unterstützung der SPARC Architektur Voraussetzung. Daher wird in der Bewertung dieser Kategorie lediglich berücksichtigt, ob die Kerne für LEON2 und LEON3 vorhanden oder nicht vorhanden sind und ob weitere Kerne implementiert werden können.

#### 4.2. Implementierte Geräte und Erweiterbarkeit

Damit ein Emulator für Unit- und Integrationstests verwendet werden kann, muss er die Hardware abbilden, für die die Treiber und ASW geschrieben wurden.

Übliche Bussysteme, die im Zusammenhang mit der Prozessorarchitektur SPARC verwendet werden, sind beispielsweise der Controller Area Network (CAN) Bus, der Advanced Peripheral Bus (APB) und Advanced High-performance Bus (AHB) sowie SpaceWire.

Besonders im Bereich der LEON Prozessoren finden die System-on-Chip Geräte der Gaisler IP Bibliothek[16] Verwendung. Dazu gehören unter anderem der Interrupt Controller for Multiple Processors (IRQMP), die General purpose Timer Unit (GRTIMER), der APBUART, Gaisler SpaceWire 1/2 und weitere. Um initiale Entwicklungszeiten gering zu halten, ist eine breite Unterstützung der oben erwähnten Geräte für einen geeigneten Emulator wünschenswert.

Fehlende oder projektspezifisch individualisierte Geräte müssen implementierbar und in Boardkonfigurationen einbindbar sein.

In dieser Kategorie wird bewertet, ob die Emulatoren erfolgreich auf die PLATO F-DPU Hardware angepasst werden können. Ausschlaggebend ist dafür, ob Unittests für die ASW Treiber erfolgreich ausgeführt werden können. Eine weitere Rolle für die Bewertung spielt der

Programmieraufwand für die Bereitstellung häufig verwendeter Geräte. Daher wird außerdem berücksichtigt, wie viele der oben genannten Geräte in den Emulatoren bereits implementiert sind.

### 4.3. Kontinuierliche Integration und Testüberdeckung

Unit- und Integrationstests sollen in einen Entwicklungsfluss, auch *kontinuierliche Integration* genannt, eingebunden werden, um sie so zeiteffizient wie möglich ausführen zu können. Dabei wird eine Reihe von Arbeitsschritten automatisiert nacheinander ausgeführt.

Als Werkzeug für die kontinuierliche Integration wird die quelloffene Software Jenkins<sup>4</sup> verwendet. Für diese Arbeit wird ein Jenkins Server installiert. Mithilfe von Aufgaben, die automatisch gestartet werden können, kann eine Reihe von Befehlen auf dem Jenkins Server ausgeführt werden.

Im betrachteten Anwendungsfall sollen das Bauen einer Binärdatei mittels SPARC Compiler und das Ausführen der Binärdatei durch einen Emulator automatisiert werden.

Neben der kontinuierlichen Integration soll der Emulator Möglichkeiten für die Ermittlung der Testüberdeckung bieten. Dabei wird während der Ausführung des Codes überprüft, welche Funktionen und Abzweigungen während der Ausführung durchlaufen wurden und in einem Bericht verfügbar gestellt.

### 4.4. Fehlerinjektion

Hardware, die im All eingesetzt wird, ist besonderen Umgebungseinflüssen wie kosmischer Strahlung ausgesetzt, die unerwartetes Verhalten in der Hardware auslösen können [18, p. 1]. Auftretende Fehler können dabei sogenannte Bitflips sein, das Kippen eines Bits auf den umgekehrten Wert, oder Latch-Ups, Kurzschlüsse in Halbleiterbauteilen. Eine Ursache für diese Fehler sind sogenannte Single Event Upsets (SEUs), Zustandsänderungen von Registern oder Flip-Flops. Betroffene Geräte können unbestimmtes Verhalten zeigen oder falsche Werte produzieren [18, p. 3]. Ein geeigneter Emulator soll daher die Möglichkeit bieten, während der Emulation Fehler in den Speicher des Prozessors einzufügen. Im Idealfall sind solche Funktionen bereits implementiert, andernfalls soll die Implementierung durch den Entwickler oder den Hersteller durchführbar sein.

### 4.5. Kosten und Support

Ein Faktor, der für die Bewertung der Emulatoren ebenfalls in Betracht gezogen wird, sind auftretende Kosten. Dabei spielen einmalige und laufende Kosten für Lizenzen und Support eine Rolle. Ebenfalls berücksichtigt wird, ob es für die Software einen offiziellen Support gibt, an den sich Entwickler und Tester bei Fragen oder auftretenden Problemen wenden können. Als offizieller Support werden dabei Kontaktstellen berücksichtigt, über die der Hersteller garantiert Hilfe leistet.

---

<sup>4</sup><https://www.jenkins.io/>

## 5. Untersuchung und Anpassung von Emulatoren auf die PLATO Hardware

In diesem Kapitel wird die Einsatzfähigkeit der Emulatoren QEMU und TEMU zum Testen von Raumfahrtsoftware anhand der in Kapitel 4 ausgewählten Kriterien untersucht. Im Speziellen wird dabei die Implementierung des AHB Status Registers sowie die Ausführung von Unittests für die Treiber des AHB Status Registers aus der PLATO F-DPU ASW in beiden angepassten Emulatoren durchgeführt.

Das AHB Status Register dient mit dem AHB Failing Address Register der Identifikation fehlerhafter Zugriffe auf den AHB des F-DPU-Prozessors. Wird auf eine nicht zugewiesene Adresse des AHB zugegriffen, wird der AHB Standard-Slave aufgerufen, welcher wiederum einen Fehler im AHB Status Register auslöst.

Das 32-Bit AHB Status Register speichert nicht nur ab, dass ein fehlerhafter Zugriff stattfand. Je nach Adresse des Fehlers wird auch erfasst, ob der Zugriff auf eine Adresse innerhalb des RAM stattfand und somit korrigierbar ist.

Das AHB Failing Address Register speichert jeweils die dazugehörige Adresse ab, auf die der fehlerhafte Zugriff stattfand. Schließlich wird der Interrupt 1 des AHB Interruptcontrollers ausgelöst.

Da das AHB Status Register das AHB Failing Address Register für seine Funktionalität benötigt, werden beide Register gemeinsam implementiert. Der Einfachheit halber wird im folgenden Verlauf nur vom AHB Status Register oder AHB Status Gerät gesprochen. Der Begriff AHB Status Gerät wird im Zusammenhang mit implementierter Hardware in Emulatoren verwendet.

Für das Nachbilden des AHB Controller Verhaltens bei Auslösen eines Speicherfehlers wird ein zusätzliches Register, das AHB Debug Register, im Emulator implementiert. Dieses setzt die entsprechenden Flags für die Identifikation der Fehlerart im AHB Status Register und speichert die Fehleradresse im AHB Failing Address Register.

Die Treiber in der ASW für das AHB Status und das AHB Failing Address Register implementieren neben einigen Get- und Set-Funktionen eine Interruptroutine und die Installation derselben.

In der Routine wird überprüft, um welche Art von Fehler es sich handelt, und die Fehleradresse in einem Stack gespeichert. Außerdem werden, abhängig davon ob es sich um einen korrigierbaren Fehler handelt oder nicht, die entsprechenden Zählvariablen in der Funktion inkrementiert.

Für die AHB Status Treiber der PLATO ASW werden Unittests geschrieben und jeweils mit den zu testenden Emulatoren ausgeführt. Dafür wird das Testframework Unity<sup>5</sup> verwendet. Dieses bietet in C unter anderem eine Reihe von assert-Funktionen, mit deren

---

<sup>5</sup><https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html>



## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

---

Hilfe zwei Werte miteinander verglichen werden können. Nach Ausführung der implementierten Tests wird eine kurze Zusammenfassung ausgegeben, die anzeigt, wie viele Tests erfolgreich ausgeführt wurden und wie viele Tests fehlerhaft verlaufen sind.

### 5.1. QEMU

Der in Abschnitt 3.4 bereits vorgestellte CPU-Emulator QEMU ist frei verfügbar. Die Software kann als vorgebaute Binärdatei für Windows, Mac und Linux heruntergeladen werden<sup>6</sup>. Für Entwicklungen und Anpassungen kann das Git Repository<sup>7</sup> geklont werden. Da QEMU ein auf die Virtualisierung vielseitiger Zielsysteme ausgerichteter Emulator ist [3, p. 45], werden viele Prozessorarchitekturen unterstützt, darunter unter anderem ARM, SPARC, PowerPC und i386/x86-64 [8].

In QEMU wurde für die Emulation von LEON3 Prozessoren durch Entwickler der Firma AdaCore<sup>8</sup> eine LEON3 Konfiguration, sowie der IRQMP, GPTIMER und der APBUART aus der Gaisler IP Bibliothek implementiert. Die AMBA Plug and Play Funktionalität wurde nachträglich durch Jiri Gaisler eingefügt.

Die Implementierung eigener Geräte erfolgte in QEMU mithilfe des QEMU Object Models [17]. Die benötigten Register und Verweise auf Schnittstellen wie Interrupts werden innerhalb eines Structs zugewiesen. Wird auf den Speicherbereich des Gerätes zugegriffen, wird der Offset der Adresse innerhalb einer Switch-Routine behandelt und die hinterlegten Funktionen ausgeführt.

Die vollständige Implementierung des AHB Status Registers in QEMU kann in Anhang 6 nachvollzogen werden.

Für die Unittests wurde die in QEMU existierende LEON3 Boardkonfiguration dupliziert und verwendet. Für das Einfügen des AHB Status Registers in die Memory Map des Boards wurde die Prozessor-Initialisierungsroutine um das Anlegen des AHB Status Gerätes erweitert.

```
1 /* Allocate Ahbstat device */
2   dev = qdev_new(TYPE_AHBSTAT);
3   sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
4   sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, LEON3_AHBSTAT_OFFSET);
5   sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0,
6                       qdev_get_gpio_in(irqmpdev, LEON3_AHBSTAT_IRQ));
```

Quellcode 2: Registrierung des AHB Status Gerätes in QEMU

---

<sup>6</sup><https://www.qemu.org/download/>

<sup>7</sup><https://gitlab.com/qemu-project/qemu.git>

<sup>8</sup><https://www.adacore.com/>

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

Wie in Codeausschnitt 2 zu sehen ist, wird das AHB Status Register angelegt und als System Bus im QEMU Object System registriert, in den Arbeitsspeicherbereich des Boards gelegt und mit Interrupt 1 des Interruptcontrollers verbunden.

Damit das angepasste Board in QEMU zur Verfügung steht, mussten mehrere Dateien des Build Systems angepasst und um die neue Boarddatei erweitert werden. Im Anschluss wurde QEMU mit dem Befehl *make* gebaut.

Wie in Abbildung 4 erkennbar ist, wurden die in Kapitel 4 beschriebenen Unittests nach den Anpassungen erfolgreich durchgeführt.

```
herg_pa@rmc-070sim2vl:~/qemu/build$ ./qemu-system-sparc -nographic -no-reboot -M leon3_ahbstat
Release-4.4.2-leon3/fdpu_unittest_ut_01
../src/main.c:55:test_drv_ahbstat:PASS

-----
1 Tests 0 Failures 0 Ignored
OK
```

Abbildung 4: Ausführen der AHB Status Treiber Unittests in QEMU

Für die SPARC Architektur unter QEMU existieren keine Werkzeuge für die Fehlerinjektion. Die Implementierung solcher Werkzeuge wurde für die ARM Architektur allerdings bereits durchgeführt [15]. Es ist daher wahrscheinlich, dass fehlende Funktionen für Fehlerinjektion für SPARC ebenfalls implementierbar sind. Die Durchführung war allerdings im zeitlichen Rahmen dieser Arbeit nicht möglich.

Für die kontinuierliche Integration mithilfe von Jenkins wurde ein sogenanntes Pipeline-Projekt angelegt, das eine Reihe von Befehlen ausführt.

QEMU kann in Jenkins integriert ausgeführt werden. In Abbildung 5 ist sichtbar, dass die Unittests mit QEMU innerhalb der Jenkins Pipeline erfolgreich durchgeführt wurden. Das vollständige Pipeline-Script ist im Anhang unter Codeabschnitt 7 abgebildet.

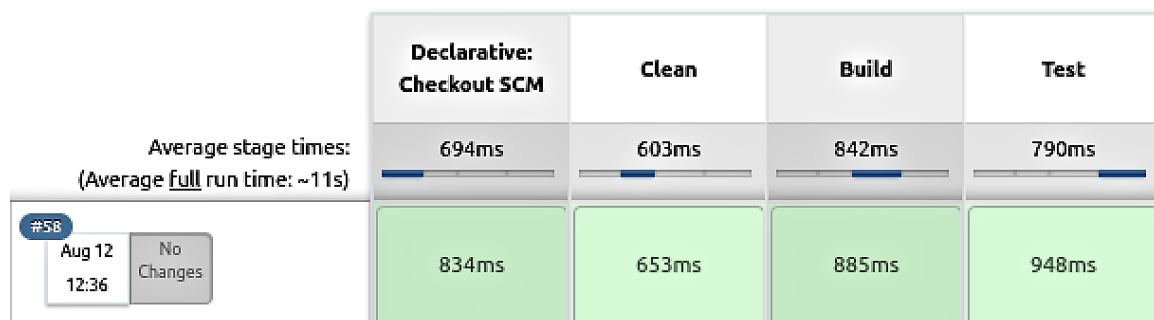


Abbildung 5: Integration von QEMU in Jenkins

Für die Ermittlung der Testüberdeckung werden in QEMU keine Daten generiert. In dem Projekt Couverture [4] wurde die Erweiterung von QEMU um die Generierung von Daten für die Testüberdeckung behandelt. Das Programm ist allerdings nicht mehr über die

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

---

Quellen verfügbar. Eine Untersuchung des Programms war damit nicht möglich. Die Implementierung eines eigenen Werkzeugs für die Generierung von Daten für die Testüberdeckung konnte innerhalb dieser Arbeit aus zeitlichen Gründen nicht durchgeführt werden.

### 5.2. TEMU

Für eine Einzellizenz des in Kapitel 3.4 vorgestellten Emulators TEMU erhebt Terma Kosten im fünfstelligen Euro Bereich zuzüglich jährlicher anteiliger Kosten für Updates und Support. Für verschiedene Prozessorarchitekturen müssen dabei Einzellizenzen erworben werden.

Die aktuelle Stable Version 2 des Programms wird für Linux Distributionen mit C++11 Unterstützung bereitgestellt.

Durch seine Ausrichtung auf Softwareprogrammierung im europäischen Raumfahrtsektor unterstützt TEMU die PowerPC, SPARC und ARM Prozessorarchitekturen. Dabei werden durch Terma für SPARC die Prozessorkerne ERC32, LEON2, LEON3 und LEON4 bereitgestellt [34].

Weiterhin sind von Terma bereits folgende Geräte und Busse aus der Gaisler Bibliothek implementiert [34]:

- AHBCTRL
- AHBSTAT
- APBCTRL
- APBUART
- CAN\_OpenCores
- FTMCTRL
- GPTIMER
- GRSPW2
- IRQMP
- AMBA Plug and Play

Die Firma Terma weist auf ihrer Website<sup>9</sup> außerdem darauf hin, dass auf Anfrage weitere Prozessorkerne implementiert werden können.

Da für die Verwendung von TEMU eine Lizenz benötigt wird, wurde für den Zeitraum der Bachelorarbeit eine Testlizenz mit vollem Funktionsumfang durch Terma zur Verfügung gestellt. Genaue Hinweise zur Installation und Einrichtung von TEMU sind auf der Website der Firma beschrieben<sup>10</sup>, daher wird im Folgenden nicht näher auf die grundlegende Bedienung der Software eingegangen.

TEMU bietet ein Framework für das Erstellen und Einbinden eigener Geräte in Boardkonfigurationen als Plugins. Entsprechende Anleitungen finden sich auf der Website von Terma<sup>11</sup>. Ähnlich wie bereits für QEMU wurden auch in TEMU über eine Struktur die

---

<sup>9</sup><https://temu.terma.com/features.html>

<sup>10</sup><https://temu.terma.com/docs/manuals/v2.2.5/sum/temu-user-manual.pdf>

<sup>11</sup><https://temu.terma.com/docs/manuals/nightly/public/manual/latest/cli.html>

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

---

Geräteigenschaften modelliert und die Referenz auf den Interruptcontroller sowie die TEMU-Klasse *Super* für das Vererben allgemeiner Objekteigenschaften angelegt. In TEMU wird das Verhalten bei Zugriff auf den zugewiesenen Speicherbereich mithilfe einer Switch-Routine realisiert, allerdings werden hier eigens definierte Get- und Set-Methoden referenziert, da diese für die Fehlerinjektion und das Debugging benötigt werden.

Die Initialisierung des AHB Status Gerätes erfolgt innerhalb einer einzelnen Funktion, die die Register mit ihren Get- und Set-Funktionen, sowie die Referenz auf den Interruptcontroller und das Memory Interface entgegennimmt.

Die vollständige Implementierung des AHB Status Gerätes kann im Anhang in Codeausschnitt 5 nachvollzogen werden.

Nach der Kompilierung der Quelldatei wurde diese als Bibliothek in die Standard-Boardkonfiguration für LEON3 eingebunden. Dies erfolgte in TEMU über eine Reihe von Kommandos, die in Codeausschnitt 3 aufgeführt sind. Dort wurde zunächst ein Objekt der Klasse AHB Status erstellt und anschließend in den Bereich des RAM gelegt.

```
1 import AhbStat
2 echo "mapping leon3 ahbstat device"
3 object-create class=AHBSTAT name=ahbstat0
4 memory-map memspace=mem0 addr=0x8000000C length=0xC object=ahbstat0
```

Quellcode 3: Einfügen des AHB Status Gerätes in das LEON3 Default Board

Nach dem Erweitern der LEON3 Konfiguration wurden die in Kapitel 4 beschriebenen Unittests ausgeführt. In Abbildung 6 ist zu sehen, dass die Ausführung der Unittests fehlerfrei war.

```
temu> run obj=cpu0
../src/main.c:55:test_drv_ahbstat:PASS

-----
1 Tests 0 Failures 0 Ignored
00.001481: info: cpu0 : error mode due to 'trap_instruction' trap (tt = 0x80) @ 0x40000800
```

Abbildung 6: Ausführen der AHB Status Unittests in TEMU

Für die Injektion von Fehlern stehen in TEMU unterschiedliche Werkzeuge zur Verfügung. Im Zentrum steht jeweils die Möglichkeit die Emulation in TEMU mit einem Zeitparameter zu starten, sodass sie entweder nach einer bestimmten Anzahl von Zyklen oder Sekunden pausiert wird. Der Benutzer hat dann die Möglichkeit, Methoden seines eigenen Gerätes oder durch Terma implementierte Methoden aufzurufen.

Für die Änderung von Registerwerten während der Emulation können beispielsweise Get- und Set-Methoden aufgerufen werden, wie sie in Quellcode 5 für das AHB Status Register implementiert wurden.

In Abbildung 7 ist die Anpassung eines Registerwertes während der Emulation zu sehen. In

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

---

einem while-Loop wurde das AHB Status Register auf den Wert 0x200 überprüft. Nach dem Schreiben des Wertes in das Register, wurde das Programm beendet.

```
temu> run obj=cpu0 time=0.5
temu> object-prop-write prop=ahbstat0.ahb_sr val=0x200
:info: ahbstat0 : updating ahb_sr from 0x0 to 0x200
temu> run obj=cpu0 time=0.5
CE bit is set
0.500051: info: cpu0 : error mode due to 'trap_instruction' trap (tt = 0x80) @ 0x40000800
```

Abbildung 7: Ändern von Registerwerten während der Emulation in TEMU

Außerdem können SEUs und Multiple Bit Upsets (MBUs) als Attribute von Adressen im Speicher gesetzt werden. Mithilfe des gesetzten Attributs kann ein Fehlerhandler verbunden werden, der bei Zugriff auf den angegebenen Speicherbereich ausgelöst wird. Für das Manipulieren einzelner Speicherwerte kann während der Emulation ein Wert in den Speicher geschrieben werden. Diese Funktionalität wurde nicht getestet, wird aber vom Hersteller zugesichert.

Da die durch Terma ausgestellte Evaluierungslizenz für TEMU zum Zeitpunkt des Testens der kontinuierlichen Integration nicht mehr gültig war, konnte diese in Jenkins für TEMU nicht getestet werden.

Es ist davon auszugehen, dass die Einbindung prinzipiell möglich ist. Das genaue Verhalten, beispielsweise mit Pausierung der Emulation und Einfügen von Fehlern, konnte aber nicht getestet werden.

Daten für die Testüberdeckung können in der, für die Arbeit nicht vorliegenden, Beta Version 3 von TEMU generiert werden [33]. Dabei wird erfasst, welche Code-Abzweigungen wie oft betreten wurden. Die Daten können dann in eine Datei exportiert werden.

Die vollständige Auswertung ist nicht implementiert und muss mithilfe der Binärdatei durch den Anwender vorgenommen werden. Hierfür steht die Entwicklung einer vollständigen Integration durch Terma noch aus.

### 5.3. Überwundene Schwierigkeiten bei der Umsetzung

Während der Arbeit mit den Emulatoren QEMU und TEMU sind Probleme aufgetreten, auf die in diesem Abschnitt mit der entsprechenden Lösung eingegangen wird.

#### Deaktivierte Traps im Prozessor Status Register (PSR)

In LEON3 Prozessoren sind Traps im PSR standardmäßig deaktiviert. Die Aktivierung erfolgt im Normalfall im Startcode, der den Prozessor und andere Geräte initialisiert.

Die Problematik der deaktivierten Traps äußerte sich dadurch, dass Interrupthandler nach dem Auslösen eines Interrupts nicht betreten wurden.

Um die Traps zu aktivieren, musste, wie in Abbildung 8 zu erkennen ist, das Enable Traps

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

(ET) Bit des PSR gesetzt werden. Dies wurde durch eine UND-Verknüpfung des PSR mit dem Wert 0x2 realisiert.

Figure 4-3 PSR Fields

<i>impl</i>	<i>ver</i>	<i>icc</i>	reserved	EC	EF	PIL	S	PS	ET	CWP
31:28	27:24	23:20	19:14	13	12	11:8	7	6	5	4:0

Abbildung 8: PSR Register eines SPARC V8 Prozessors [21, p. 28]

In QEMU kann die Aktivierung im Usercode oder in der Boardkonfiguration vorgenommen werden. Da die Software in der Boardkonfiguration lediglich Maschinencode entgegennimmt, mussten die entsprechenden Befehle für das Anpassen des PSR in Maschinencode übergeben werden. In Codeausschnitt 4 ist die Erweiterung der Initialisierungsroutine um die Aktivierung der Traps in Maschinencode abgebildet.

```
1 /* activate Interrupts in psr */
2 stl_p(p++, 0x85480000); /* rd %psr, %g2 */
3 stl_p(p++, 0x8410a020); /* or %g2, 0x20, %g2 */
4 stl_p(p++, 0x8188a000); /* mov %g2, %psr */
5 stl_p(p++, 0x01000000); /* nop */
6 stl_p(p++, 0x01000000); /* nop */
7 stl_p(p++, 0x01000000); /* nop */
```

Quellcode 4: Aktivieren von Interrupts im PSR in QEMU

In TEMU müssen die Traps im Usercode aktiviert werden, da die Boardkonfiguration nicht einsehbar ist. Eine weitere Möglichkeit ist, vor Auslösen eines Interrupts während der Emulation den Wert des PSR über TEMU Kommandos zu ändern.

### Fehlerhafte AMBA Plug and Play Implementierung in QEMU

In QEMU wurde durch Jiri Gaisler die AMBA Plug and Play Funktionalität implementiert. Wird direkt nach dem Klonen des Git Repositories mit dem LEON3 Board ein Programm ausgeführt das den APBUART verwendet, werden keine Daten über diesen ausgegeben. Grund dafür ist ein Fehler in der Implementierung der AMBA Plug and Play Funktion.

Für die Beseitigung wurde ein Patch von der Gaisler Website<sup>12</sup> nachinstalliert.

### AMBA Plug and Play Scan Funktion in QEMU

In Bare Metal Compilern der Firma Gaisler wird ab Version 1.0.41 automatisch eine Scan Routine des AMBA Busses ausgeführt. Diese sucht den AMBA Bus nach UARTs, Timern und Interruptcontrollern ab. So können diese Geräte an andere als die jeweilige Standardadresse im Speicher gelegt werden. In QEMU löste diese Funktion einen Trap 0x06 Window Underflow aus.

<sup>12</sup><https://gaisler.se/qemu/>

## 5. UNTERSUCHUNG UND ANPASSUNG VON EMULATOREN AUF DIE PLATO HARDWARE

---

```
herg_pa@rmc-070sim2vl:~/qemu/build$ ./qemu-system-sparc -nographic -no-reboot -M l
qemu: fatal: Trap 0x06 (Window Underflow) while interrupts disabled, Error state
pc: 400019b4 npc: 40006a98
%g0-7: 00000000 40008800 80000300 80000000 00000003 00000003 40006a88 00000000
%o0-7: 00000001 0000000d 43fffff4 00000000 00000000 00000000 43ffff90 40001994
%l0-7: 43fffff4 00000000 00000000 00000000 00000000 00000000 00000000 00000000
%i0-7: 00000001 00000000 00000000 00000000 00000000 00000000 44000000 40006a90
%f00: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
%f08: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
%f16: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
%f24: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
psr: f34000c7 (icc: -Z-- SPE: SP-) wim: 00000001
fsr: 00000000 y: 00000000
```

Aborted

Abbildung 9: Trap 0x06 in QEMU

Die Scan Funktion konnte als Auslöser über die Analyse der Assemblerdatei und die Ausgabe des Programmzählers in QEMU identifiziert werden.

Um das Problem zu umgehen, musste bei der Kompilierung die Option *-qnoambapp* an den Compiler übergeben werden. Außerdem mussten UART, Timer und Interruptcontroller an die jeweilige Standardadresse im Speicher gelegt werden.

Dieses Problem spielt vor allem in Anwendungen eine Rolle, die keinen Startcode für die Zuordnung abweichender Adressen für die oben genannten Geräte ausführen. Für Anwendungen von QEMU in solchen Fällen muss die genaue Ursache des Problems ermittelt und beseitigt werden.

### 6. Auswertung

Die Emulatoren QEMU und TEMU wurden in Kapitel 5 auf ihre Eignung zum Testen von Raumfahrtsoftware untersucht. Die Ergebnisse dieser Untersuchung werden in diesem Kapitel erläutert und in einer Tabelle am Ende des Kapitels zusammengefasst.

#### 6.1. Unterstützte Prozessorarchitekturen und -kerne

Sowohl QEMU als auch TEMU unterstützen neben SPARC die ARM Architektur. Damit können beide Emulatoren für PLATO und andere Projekte mit SPARC oder ARM Prozessoren in Betracht gezogen werden.

In QEMU liegt lediglich eine LEON3 Implementierung vor. Da dieser Emulator jedoch quelloffen und damit frei erweiterbar ist, ist die Implementierung weiterer LEON Kerne möglich.

In TEMU werden die Kerne der LEON2, LEON3 und LEON4 Prozessoren bereitgestellt, weitere Implementierungen können lediglich durch Terma vorgenommen werden.

Die Unterscheidung beider Emulatoren besteht hier demnach in der direkten Verfügbarkeit mehrerer SPARC Prozessorkerne auf Seiten von TEMU und der flexiblen Erweiterbarkeit auf Seiten von QEMU.

#### 6.2. Implementierte Geräte und Erweiterbarkeit

Beide untersuchte Emulatoren unterstützen Gaisler Hardware, wobei TEMU, wie in Kapitel 5 beschrieben, mehr Geräte unterstützt als QEMU. Dafür können vorhandene Implementierungen in TEMU lediglich durch Terma angepasst werden, während in QEMU die Anpassung aller Komponenten möglich ist.

In QEMU enthält die AMBA Plug and Play Funktion Fehler. Diese mussten durch die Nachinstallation eines Patches beseitigt werden. Dies zeigte, dass in QEMU Fremdimplementierungen auf korrekte Funktionalität hin überprüft werden müssen.

In Abschnitt 5 wurde in den Emulatoren QEMU und TEMU das AHB Status Register nachgebildet. Dabei wurde die Funktionalität des Registers in beiden Emulatoren erfolgreich emuliert, sodass die mit Unity geschriebenen Unittests fehlerfrei durchgeführt wurden.

Die Implementierung neuer Geräte ist in beiden Emulatoren möglich.

#### 6.3. Kontinuierliche Integration und Testüberdeckung

Die Einbettung in kontinuierliche Integration mithilfe von Jenkins wurde lediglich für QEMU durchgeführt, da die Probelizenz für TEMU zu dem Zeitpunkt der Untersuchung bereits abgelaufen war.

Die Untersuchung von QEMU zeigte aber, dass die prinzipielle Ausführung eines Emulators



in Jenkins möglich ist. Das genaue Verhalten von TEMU wurde zwar nicht betrachtet, es ist aber davon auszugehen, dass die Integration möglich ist.

Daten für die Testüberdeckung werden in QEMU nicht generiert. Das Projekt Couverture [4], das diese fehlende Funktion behandelt, ist auf der entsprechenden Website nicht mehr verfügbar. Couverture zeigt, dass es möglich ist, Werkzeuge für die Fehlerinjektion selbst zu implementieren und wie bei einer Implementierung vorgegangen werden könnte.

Obwohl TEMU keine direkten Berichte zur Testüberdeckung produziert, stellt die Software in der aktuellsten Version 3 die nötigen Rohdaten zur Verfügung, um entsprechende Berichte selbst zu generieren. Die Zusammenführung mit der Binärdatei für eine vollständige Auswertung muss durch den Anwender selbst durchgeführt werden. Damit kann dieser Aspekt für TEMU als teilweise erfüllt betrachtet werden.

### 6.4. Fehlerinjektion

Auch für die Fehlerinjektion stehen in QEMU keine Werkzeuge zur Verfügung, zumindest nicht für die Prozessorarchitektur SPARC. Dass Fehlerinjektionen prinzipiell möglich sind, wurde in Kapitel 5 anhand eines Beispiels gezeigt. Eine Funktion für SPARC muss unter QEMU aber erst implementiert werden.

In TEMU wurden Fehler durch eigene Get- und Set-Funktionen eingefügt. Außerdem stellt TEMU prinzipiell Werkzeuge für das Auslösen von SEUs und MBUs im Speicher zur Verfügung.

### 6.5. Kosten und Support

Wie in Abschnitt 5.2 beschrieben, werden für die Nutzung von TEMU durch Terma Kosten im fünfstelligen Euro Bereich erhoben. Für Implementierungen in TEMU wurden durch Terma bereitgestellte Benutzeranleitungen verwendet [32]. Außerdem wurde für Fragen der Produktsupport kontaktiert, der zeitnah und ausführlich antwortete.

Die Nutzung von QEMU ist kostenfrei. Die Erläuterung in den vorangegangenen Punkten legt dar, dass viele Funktionen, die TEMU bietet, in QEMU fehlen. Dies beinhaltet auch das Fehlen eines Supports. Es muss entsprechend Zeit investiert werden, um QEMU auf einen funktionalen Stand zu bringen, auf dem TEMU sich bereits befindet.

Für QEMU stehen ebenfalls Dokumentationen für die Implementierung eigener Geräte bereit [17]. Für auftretende Fragen während der Implementierung wurde im Internet nach Lösungen der QEMU Community recherchiert. Zusätzlich besteht die Möglichkeit über eine entsprechende Mailing-Liste QEMU Entwickler zu kontaktieren. Dabei besteht jedoch keine Garantie auf Erfolg beziehungsweise eine zeitnahe Beantwortung der Fragestellung.

### 6.6. Zusammenfassung der Untersuchungsergebnisse

Die Untersuchungsergebnisse zeigen, dass beide Emulatoren Stärken und Schwächen aufweisen.

	QEMU	TEMU
<b>Implementierte SPARC Prozessorkerne</b>	LEON3, frei erweiterbar	LEON2, LEON3, LEON4, kostenpflichtig erweiterbar
<b>Implementierte Geräte</b>	IRQMP, GPTIMER, APBUART, AMBA Plug and Play  alle Geräte anpassbar	AHBCTRL, APBCTRL, IRQMP, GPTIMER, APBUART, AMBA Plug and Play FTMCTRL, CAN Bus alle Geräte nur durch Terma anpassbar
<b>Erweiterbarkeit</b>	frei erweiterbar	kostenpflichtig erweiterbar
<b>Kontinuierliche Integration</b>	integrierbar	voraussichtlich integrierbar
<b>Testüberdeckung</b>	nicht vorhanden, implementierbar	grundlegend vorhanden
<b>Fehlerinjektion</b>	nicht vorhanden, implementierbar	vorhanden
<b>Kosten</b>	kostenfrei	fixe und jährliche Lizenzkosten
<b>Support</b>	kein Support	kostenpflichtiger Support

Tabelle 1: Tabellarische Zusammenfassung der Untersuchungsergebnisse

Für den kurzfristigen Einsatz ist TEMU besser geeignet, da in diesem mehr LEON Kerne und Gaisler Geräte implementiert sind, als es bei QEMU der Fall ist. TEMU bietet außerdem einen offiziellen Support und Werkzeuge für Fehlerinjektion. Grundlegende Daten für die Ermittlung der Testüberdeckung werden generiert.

Da es sich um kommerzielle Software handelt, müssen jedoch Lizenzkosten und Supportgebühren gezahlt werden. Außerdem können vorhandene Implementierungen nur durch den Hersteller Terma angepasst werden. Neue Prozessorarchitekturen und -kerne müssen ebenfalls durch Terma hinzugefügt werden.

In QEMU können alle Dateien des Emulators angepasst werden, da der Quellcode offen ist. Dadurch können sowohl das Gesamtverhalten des Emulators als auch bereits implementierte Geräte und Kerne verändert werden. Außerdem ist QEMU kostenfrei erhältlich. Auf langfristige Sicht kann QEMU daher nicht nur aus finanziellen Gründen attraktiver sein. Wird in einem Projekt statt SPARC eine andere Prozessorarchitektur verwendet, kann dies in QEMU umgesetzt werden.

Die initiale Entwicklungsarbeit ist in QEMU allerdings höher, da nur wenige Gaisler Geräte und nur ein LEON3 Kern implementiert sind. Werkzeuge für Fehlerinjektionen oder Generierung von Daten für die Testüberdeckung sind in QEMU nicht standardmäßig enthalten. In dieser Arbeit getestete Implementierungen anderer Entwickler, wie die AMBA Plug and Play Funktion, haben sich teilweise als fehlerhaft herausgestellt.

## 6. AUSWERTUNG

---

Daher müssen Fremdimplementierungen ausführlich getestet oder Werkzeuge selbst entwickelt werden.

Die Wahl eines Emulators hängt also von der Gewichtung der Faktoren kurzfristige Verfügbarkeit, Kosten und Anpassbarkeit ab.

### 7. Fazit und Ausblick

In dieser Arbeit wurde die Notwendigkeit erläutert, Software in Raumfahrtprojekten früh und ausführlich zu testen. Dabei wurden Problemfelder präsentiert, denen Softwareentwickler in Raumfahrtprojekten beim Testen ihrer Implementierungen begegnen. Dies sind noch nicht oder in geringen Mengen zur Verfügung stehende Zielhardware und fehlende Möglichkeiten zum Auslösen bestimmter Fehlerfälle.

Als mögliche Lösung wurde der Einsatz von Emulatoren für die Durchführung von Tests aufgezeigt und erläutert.

Um die Eignung von Emulatoren für die Durchführung solcher Tests festzustellen, wurden Kriterien für die Untersuchung definiert:

- Unterstützte Prozessorarchitekturen und -kerne,
- Implementierte Geräte und Erweiterbarkeit,
- Kontinuierliche Integration und Testüberdeckung,
- Fehlerinjektion sowie
- Kosten und Support.

Ziel der Arbeit war es, die Emulatoren QEMU und TEMU mithilfe der ausgewählten Kriterien auf die Eignung zur Durchführung von Unittests für Hardware Treiber einer Raumfahrtsoftware zu untersuchen. Dazu wurden am Beispiel des AHB Status Registers der PLATO F-DPU ASW Unittests entwickelt. Im Anschluss wurde dieses Register mit beiden Emulatoren nachgebildet.

Grundsätzlich konnten beide Emulatoren auf die PLATO F-DPU Hardware angepasst werden. Die Unittests konnten auf beiden Emulatoren erfolgreich ausgeführt werden. Die Untersuchung hat dabei die Stärken und Schwächen beider Emulatoren aufgezeigt. Die Stärken von QEMU liegen in der Anpassbarkeit des gesamten Quellcodes und kostenfreien Nutzung. Die Schwächen der Software sind in der geringen Anzahl von implementierten LEON Prozessorkernen, Geräten der Gaisler IP Bibliothek und weiteren in der Raumfahrt genutzten Geräten zu finden. Bereits implementierte Lösungen wiesen außerdem teilweise Fehler auf. TEMU wird mit einer größeren Anzahl von LEON Kernen, Gaisler IP und weiteren relevanten Geräten geliefert. Außerdem besteht durch die Herstellerfirma ein Support. Die Nutzung und Erweiterung der Software ist jedoch an hohe Lizenzgebühren gebunden.

## 7. FAZIT UND AUSBLICK

---

Sowohl QEMU als auch TEMU sind für die Durchführung von Unittests für die Treiber der PLATO ASW geeignet. Die Entscheidung für einen der beiden Emulatoren orientiert sich letztendlich an der Gewichtung der Aspekte kurzfristige Verfügbarkeit, Kosten und Anpassbarkeit.

In QEMU sind offen gebliebene Arbeiten das Testen oder Entwickeln geeigneter Werkzeuge für die Fehlerinjektion und Generierung von Daten für die Testüberdeckung. Zudem ist die AMBA Plug and Play Funktion noch fehlerhaft.


In TEMU muss eine Lösung gefunden werden, um die generierten Daten für die Testüberdeckung zusammen mit der Binärdatei in einen Bericht zu verarbeiten.

## Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass

- ich die vorliegende wissenschaftliche Arbeit selbstständig und ohne unerlaubte Hilfe angefertigt habe,
- ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe,
- ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe,
- die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfbehörde vorgelegen hat.

Berlin, den 25. August 2021

  
Pauline Hergersberg

# Literaturverzeichnis

## Websites

- [5] Pat Brennan. *What in the World is an 'Exoplanet?'* <https://www.nasa.gov/feature/jpl/what-in-the-world-is-an-exoplanet>. Zugegriffen am 19.03.2021. 2018 (siehe Seite 4).
- [8] QEMU Community. *Documentation/Platforms*. <https://wiki.qemu.org/Documentation/Platforms>. Zugegriffen am 25.07.2021. 2020 (siehe Seite 19).
- [10] Duden. *Emulation*. <https://www.duden.de/rechtschreibung/Emulation>. Zugegriffen am 17.07.2021. 2021 (siehe Seite 10).
- [11] Duden. *Simulation*. <https://www.duden.de/rechtschreibung/Simulation>. Zugegriffen am 17.07.2021. 2021 (siehe Seite 10).
- [13] ESA. *Artist's impression of PLATO*. <https://sci.esa.int/web/plato/-/artist-s-impression-of-plato-1>. Zugegriffen am 13.05.2021. 2019 (siehe Seite 6).
- [14] ESA. *Microprocessors*. [https://www.esa.int/Enabling\\_Support/Space\\_Engineering\\_Technology/Onboard\\_Computers\\_and\\_Data\\_Handling/Microprocessors](https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Microprocessors). Zugegriffen am 08.08.2021. 2021 (siehe Seite 16).
- [17] Aeroflex Gaisler. *GR712RC Dual-Core LEON3FT SPARC V8 Processor*. <https://www.gaisler.com/index.php/products/processors/leon3ft?task=view&id=364>. Zugegriffen am 19.07.2021. 2021 (siehe Seiten 13, 19, 27).
- [21] SPARC International Inc. *The SPARC Architecture Manual Version 8*. <https://www.gaisler.com/doc/sparcv8.pdf>. 1992 (siehe Seite 24).
- [22] International Space Science Institute-Beijing. *ESA's PLATO mission with Heike Rauer*. <https://www.youtube.com/watch?v=vRMbpXm5Vr4>. Zugegriffen am 10.05.2021. 2020 (siehe Seiten 4, 5).
- [23] ISO. *Systems and software engineering — Vocabulary*. <https://www.iso.org/standard/>. 2017 (siehe Seite 10).
- [25] Brian Koberlein. *transit-1200x511.jpg*. <https://www.forbes.com/sites/briankoberlein/2017/04/07/weve-found-an-atmosphere-around-an-earth-sized-world/>. Zugegriffen am 10.05.2021. 2017 (siehe Seite 5).

- [27] H Neuroth u. a. *Eine kleine Enzyklopädie der digitalen Langzeitarchivierung. Version 2.3, Projekt: nestor-Kompetenznetzwerk Langzeitarchivierung und Langzeitverfügbarkeit digitaler Ressourcen für Deutschland.* [http://nestor.sub.uni-goettingen.de/handbuch/nestor-handbuch\\_23.pdf](http://nestor.sub.uni-goettingen.de/handbuch/nestor-handbuch_23.pdf). 2015 (siehe Seite 12).
- [28] Hugh Osborn. *What can PLATO do for exoplanet astronomy?* <http://www.hughosborn.co.uk/2014/01/30/what-can-plato-do-for-exoplanet-astronomy/>. Zugegriffen am 13.05.2021. 2014 (siehe Seite 6).
- [29] Dr. Isabella Pagano. *Involved Institutes*. <https://platomission.com/involved-institutes/>. Zugegriffen am 21.08.2021. 2018 (siehe Seite 7).
- [30] RCTA. *List of Available Documents*. <https://www.rtca.org/wp-content/uploads/2020/12/LIST-OF-AVAILABLE-DOCS-AS-OF-SEPTEMBER-2020.pdf>. Zugegriffen am 21.08.2021. 2020 (siehe Seite 8).
- [32] Terma. *TEMU - Documentation*. <https://temu.terma.com/documentation.php>. Zugegriffen am 16.08.2021. 2021 (siehe Seite 27).
- [33] Terma. *TEMU - Profiling and Coverage*. <https://temu.terma.com/docs/manuals/nightly/public/manual/latest/profiling-and-coverage.html>. Zugegriffen am 16.08.2021. 2021 (siehe Seite 23).
- [34] Terma. *TEMU Features*. <https://temu.terma.com/features.html>. Zugegriffen am 21.07.2021. 2018 (siehe Seite 21).
- [36] Matt Williams. *What is the Radial Velocity Method?* <https://www.universetoday.com/138014/radial-velocity-method/>. Zugegriffen am 13.05.2021. 2017 (siehe Seite 5).

## Bücher

- [1] Conny Aerts u. a. *PLATO Revealing habitable worlds around solar-like stars*. ESA, 2017 (siehe Seiten 5–7).
- [2] Jan Andersson, Jiri Gaisler und Roland Weigand. “Next Generation MultiPurpose Microprocessor”. In: *DASIA 2010-Data Systems In Aerospace* 682 (2010), Seite 8 (siehe Seite 13).
- [3] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Band 41. California, USA. 2005, Seite 46 (siehe Seiten 13, 19).
- [4] Matteo Bordin u. a. “Object and Source Coverage for Critical Applications with the C OUVERTURE Open Analysis Framework”. In: *ERTS2 2010, Embedded Real Time Software & Systems*. 2010 (siehe Seiten 20, 27).



- [6] B. Carvalho, A. Pidgeon und P. Robinson. “QERx- A Faster than Real-Time Emulator for Space Processors”. In: *DASIA 2012 - Data Systems In Aerospace*. Band 701. ESA Special Publication. Aug. 2012, Seite 55 (siehe Seite 14).
- [7] Jong-Wook Choi und Byeong-Gyu Nam. “Development of high performance space processor emulator based on QEMU—Open source dynamic translator”. In: *2012 12th International Conference on Control, Automation and Systems*. IEEE. 2012, Seiten 300–304 (siehe Seite 14).
- [9] Maria Carmela Di Piazza und G. Vitale. *Photovoltaic sources: Modeling and emulation*. Band 53. Jan. 2013. ISBN: 978-1-4471-4377-2. DOI: 10.1007/978-1-4471-4378-9 (siehe Seite 3).
- [12] Jens Eickhoff. *Onboard computers, onboard software and satellite operations: an introduction*. Springer-Verlag Berlin Heidelberg, 2012 (siehe Seiten 7, 8).
- [15] Davide Ferraretto und Graziano Pravadelli. “Efficient fault injection in QEMU”. In: *2015 16th Latin-American Test Symposium (LATS)*. IEEE. 2015, Seiten 1–6 (siehe Seite 20).
- [16] Aeroflex Gaisler. *Core User’s Manual, Cobham Gaisler AB*. 2020. URL: <https://www.gaisler.com/products/grlib/grip.pdf> (siehe Seiten 14, 16).
- [18] C. S. Guenzer, E. A. Wolicki und R. G. Allas. “Single Event Upset of Dynamic Rams by Neutrons and Protons”. In: *IEEE Transactions on Nuclear Science* 26.6 (1979), Seiten 5048–5052. DOI: 10.1109/TNS.1979.4330270 (siehe Seite 17).
- [19] Stephen Hemminger u. a. “Network emulation with NetEm”. In: *Linux conf au*. Band 5. Citeseer. 2005, Seite 2005 (siehe Seite 3).
- [20] Mattias Holm. “The Terma Emulator Evolution”. In: *Simulation & EGSE Facilities for Space Programmes* (2015) (siehe Seite 13).
- [24] M Jones u. a. “Introducing ECSS software-engineering standards within ESA”. In: *ESA bulletin* (2002), Seiten 132–139 (siehe Seite 2).
- [26] Jens-Martin Loebel. *Lost in Translation: Leistungsfähigkeit, Einsatz und Grenzen von Emulatoren bei der Langzeitbewahrung digitaler multimedialer Objekte*. vwh Verlag, 2014 (siehe Seiten III, 10, 11).
- [31] Yi Sun und Li Sun. “The Design of Avionics System Interfaces Emulation and Verification Platform Based on QAR Data”. In: *Applied Mechanics and Materials* 668-669 (Okt. 2014), Seiten 879–883. DOI: 10.4028/www.scientific.net/AMM.668-669.879 (siehe Seite 3).
- [35] Arjan Tijms. “Binary translation: Classification of emulators”. In: *Leiden Institute for Advanced Computer Science* (2000) (siehe Seiten 10–12).

# Anhang

## A. TEMU

```
1 #include <stdint.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include "temu-c/Support/Objsys.h"
6 #include "temu-c/Support/Logging.h"
7 #include "temu-c/Models/IrqController.h"
8 #include "temu-c/Memory/Memory.h"
9
10 #define AHB_ADDR_RAM 0x40000000
11 #define AHB_SRAM_SIZE 0x00800000
12
13 #define AHB_STATUS_CE (1<<9)
14 #define AHB_STATUS_NE (1<<8)
15
16
17 // AHBSTAT Definition
18 typedef struct {
19     temu_Object super;
20     temu_IrqCtrlIfaceRef irq;
21     uint32_t ahb_sr;
22     uint32_t ahb_failr;
23     uint32_t ahb_debug;
24 } AHBSTAT;
25
26 // Konstruktor and Destruktor
27 // Funktion erstellt ein AHBSTAT Objekt und initialisiert die Register jeweils
    mit Null
28 static void* create(const char *name, int argc, const temu_CreateArg argv[])
29 {
30     AHBSTAT *stat = malloc(sizeof(AHBSTAT));
31     memset(stat, 0, sizeof(AHBSTAT));
32     stat->ahb_sr=0x0;
33     stat->ahb_failr=0x0;
34     stat->ahb_debug=0x0;
35     return stat;
36 }
37
38 // Setzt den zugeordneten Speicher frei
39 static void dispose(void *obj)
40 {
41     AHBSTAT *stat = (AHBSTAT*)obj;
42     free(stat);
43 }
44
45 // Setter
46 static void setSTAT(void *obj, temu_Propval pv, int idx)
```

```

47 {
48     AHBSTAT *stat = (AHBSTAT*)obj;
49     temu_logInfo(stat, "updating ahb_sr from 0x%x to 0x%x", stat->ahb_sr, stat->
50         ahb_sr|pv.u32);
51     stat->ahb_sr = pv.u32;
52 }
53 static void setFAILR(void *obj, temu_Propval pv, int idx)
54 {
55     AHBSTAT *stat = (AHBSTAT*)obj;
56     stat->ahb_failr = 0x0;
57     temu_logInfo(stat, "updating ahb_failr from 0x%x to 0x%x", stat->ahb_failr,
58         stat->ahb_failr|pv.u32);
59     stat->ahb_failr = pv.u32;
60 }
61 static void setDEBUG(void *obj, temu_Propval pv, int idx)
62 {
63     AHBSTAT *stat = (AHBSTAT*)obj;
64     temu_logInfo(stat, "updating ahb_debug from 0x%x to 0x%x", stat->ahb_debug,
65         stat->ahb_debug|pv.u32);
66     stat->ahb_debug = pv.u32;
67     stat->ahb_failr = stat->ahb_debug;
68     stat->ahb_sr |= AHB_STATUS_NE;
69     if((value >= AHB_ADDR_RAM) && (value < (AHB_ADDR_RAM + AHB_SRAM_SIZE))){
70         stat->ahb_sr |= AHBSTAT_CE_BIT;
71     }
72 }
73
74     stat->irq.IfaceraiseInterrupt(stat->irq.Obj, 1);
75 }
76
77 // Getter
78 static temu_Propval getSTAT(void *obj, int idx)
79 {
80     AHBSTAT *stat = (AHBSTAT*)obj;
81     temu_logInfo(stat, "reading AHB_STAT as 0x%x", stat->ahb_sr);
82     return temu_makePropU32(stat->ahb_sr);
83     stat->ahb_failr = 0x0;
84 }
85
86 static temu_Propval getFAILR(void *obj, int idx)
87 {
88     AHBSTAT *stat = (AHBSTAT*)obj;
89     temu_logInfo(stat, "reading AHB_FAILR as 0x%x", stat->ahb_failr);
90     return temu_makePropU32(stat->ahb_failr);
91 }
92

```

## A. TEMU

---

```
93 static temu_Propval getDEBUG(void *obj, int idx)
94 {
95     AHBSTAT *stat = (AHBSTAT*)obj;
96     temu_logInfo(stat, "reading AHBDEBUG as 0x%x", stat->ahb_debug);
97     return temu_makePropU32(stat->ahb_debug);
98 }
99
100 // Memory Interface
101 // Memory Read Switch
102 static void memRead(void *obj, temu_MemTransaction *mt)
103 {
104     temu_Propval pv;
105     mt->Value = 0;
106     switch(mt->Offset){
107     case 0x4:
108         pv = getSTAT(obj, 0);
109         break;
110     case 0x0:
111         pv = getFAILR(obj, 0);
112         break;
113     case 0x8:
114         pv = getDEBUG(obj, 0);
115         break;
116     default:
117         temu_logWarning(obj, "register at offset %d not implemented for reads", (int)
118             mt->Offset);
119     }
120     mt->Value = temu_propValueU32(pv);
121     mt->Cycles = 100;
122 }
123
124 // Memory Write Switch
125 static void memWrite(void *obj, temu_MemTransaction *mt)
126 {
127     temu_Propval pv = temu_makePropU32(mt->Value);
128     switch(mt->Offset){
129     case 0x4:
130         setSTAT(obj, pv, 0);
131         break;
132     case 0x0:
133         setFAILR(obj, pv, 0);
134         break;
135     case 0x8:
136         setDEBUG(obj, pv, 0);
137         break;
138     default:
139         temu_logWarning(obj, "register at offset %d not implemented for writes", (int)
140             mt->Offset);
```

```
140     return ;
141 }
142 mt->Cycles = 100;
143 }
144
145 // Zuordnung der Memory Interface Funktionen
146 temu_MemAccessIface MemAccessIface = {
147     NULL,
148     memRead,
149     memWrite,
150     NULL
151 };
152
153 // Initialisierung der Zugriffsfunktionen und Interfaces
154 TEMU_PLUGIN_INIT
155 {
156     temu_Class *c = temu_registerClass("AHBSTAT", create, dispose);
157     temu_addProperty(c, "ahb_failr", 0x0, teTY_U32, 1, setFAILR, getFAILR, "AHB
        failing address register");
158     temu_addProperty(c, "ahb_sr", 0x4, teTY_U32, 1, setSTAT, getSTAT, "AHB Status
        register");
159     temu_addProperty(c, "ahb_debug", 0x8, teTY_U32, 1, setDEBUG, getDEBUG, "debug
        register, for simulating hw behaviour");
160     temu_addProperty(c, "irq", offsetof(AHBSTAT, irq), teTY_IfaceRef, 1, NULL,
        NULL, "interrupt controller");
161     temu_addInterface(c, "MemAccessIface", TEMU_MEMACCESS_IFACE_TYPE, &
        MemAccessIface, 0, "memory access interface");
162 }
```

Quellcode 5: Implementierung des AHB Status Registers in TEMU

## B. QEMU

```
1 #include <stdio.h>
2
3 #include "qemu/osdep.h"
4 #include "hw/sysbus.h"
5 #include "hw/misc/ahbstat.h"
6 #include "qemu/module.h"
7 #include "qom/object.h"
8 #include "hw/irq.h"
9
10 #define AHBSTAT_CE_BIT (1<<9)
11 #define AHBSTAT_NE_BIT (1<<8)
12
13 #define AHB_ADDR_RAM 0x40000000
14 #define AHB_SRAM_SIZE 0x00800000
15
16 // AHBSTAT Definition
17 OBJECT_DECLARE_SIMPLE_TYPE(Ahbstat_plato, AHBSTAT)
18
19 struct Ahbstat_plato {
20     SysBusDevice parent_obj;
21     MemoryRegion iomem;
22     qemu_irq irq;
23
24     uint32_t ahb_sr;
25     uint32_t ahb_failr;
26     uint32_t ahb_debug;
27 };
28
29 // Memory Interface
30 // Memory Read Switch
31 static uint64_t ahbstat_plato_read(void *opaque, hwaddr offset, unsigned size)
32 {
33     uint64_t val = 0;
34     Ahbstat_plato *stat = opaque;
35
36     switch (offset) {
37
38     case 0x0:
39         val = stat->ahb_failr;
40         break;
41
42     case 0x4:
43         val = stat->ahb_sr;
44         break;
45
46     case 0x8:
47         val = stat->ahb_debug;
```

```

48         break;
49
50     default:
51         printf("Reads not implemented at offset %d", (uint32_t)offset);
52     }
53     return val;
54 }
55
56 // Memory Write Switch
57 static void ahbstat_plato_write(void *opaque, hwaddr offset, uint64_t value,
58     unsigned size)
59 {
60     Ahbstat_plato *stat = opaque;
61
62     switch (offset) {
63     case 0x0:
64         stat->ahb_failr = 0;
65         stat->ahb_failr = value;
66         break;
67         case 0x4:
68             stat->ahb_sr = 0;
69             stat->ahb_sr = value;
70             break;
71         case 0x8:
72             stat->ahb_debug = 0;
73             stat->ahb_debug = value;
74             stat->ahb_failr = stat->ahb_debug;
75
76             stat->ahb_sr |= AHBSTAT_NE_BIT;
77
78             if((value >= AHB_ADDR_RAM) && (value < (AHB_ADDR_RAM + AHB_SRAM_SIZE))){
79                 stat->ahb_sr |= AHBSTAT_CE_BIT;
80             }
81
82             qemu_irq_raise(stat->irq);
83
84             break;
85     default:
86         printf("Writes not implemented at offset %d \n\r", (uint32_t)offset);
87     }
88 }
89 }
90
91 // Zuordnung der Memory Interface Funktionen
92 static const MemoryRegionOps ahbstat_plato_ops = {
93     .read = ahbstat_plato_read,
94     .write = ahbstat_plato_write,
95     .endianness = DEVICE_NATIVE_ENDIAN,

```



```
96 };
97
98 // Initialisierung von AHBSTAT mit Zuordnung von Parent Objekten und Interfaces
99 static void ahbstat_plato_init(Object *obj)
100 {
101     SysBusDevice *sd = SYS_BUS_DEVICE(obj);
102     Ahbstat_plato *s = AHBSTAT(obj);
103
104     memory_region_init_io(&s->iomem, obj, &ahbstat_plato_ops,
105                          s, "ahbstat_plato", 0x50);
106     sysbus_init_mmio(sd, &s->iomem);
107     sysbus_init_irq(sd, &s->irq);
108 }
109
110 static const TypeInfo ahbstat_info = {
111     .name          = TYPE_AHBSTAT,
112     .parent        = TYPE_SYS_BUS_DEVICE,
113     .instance_size = sizeof(Ahbstat_plato),
114     .instance_init = ahbstat_plato_init,
115 };
116
117 // Registrierung des neuen Typs in QEMU
118 static void ahbstat_register_types(void)
119 {
120     type_register_static(&ahbstat_info);
121 }
122
123 type_init(ahbstat_register_types)
```

Quellcode 6: Implementierung des AHB Status Registers in QEMU

```
1 pipeline {
2   // Hinzufuegen der Pfade des SPARC Compilers und von QEMU zur
   Benutzerumgebung
3   environment{
4     PATH = "/opt/sparc-elf-4.4.2/bin:/home/herg_pa/qemu/build:${env.PATH}"
5   }
6   agent any
7   stages {
8     //Bereinigung des Verzeichnisses der Unittests
9     stage('Clean'){
10      steps{
11        sh 'make -C UT.01/Release-4.4.2-leon3 clean'
12      }
13    }
14    // Bauen der Unittests
15    stage('Build'){
16      steps{
17        sh 'make -C UT.01/Release-4.4.2-leon3 all'
18      }
19    }
20    // Ausfuehrung der Unittests mit QEMU
21    stage('Test'){
22      steps{
23        dir('UT_01/Release-4.4.2-leon3'){
24          sh 'qemu-system-sparc -nographic -no-reboot -M
leon3_ahbstat -m 64M -kernel /home/herg_pa/qemu/build/fdpu_unittest_ut_01'
25        }
26      }
27    }
28  }
29 }
```

Quellcode 7: Kontinuierliche Integration von QEMU mit Jenkins