Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Mathis Garrandt

Erstellung einer Funktionstestumgebung für ein Medizingerät mit einem Plattform Prozessormodul unter Linux

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Mathis Garrandt

Erstellung einer Funktionstestumgebung für ein Medizingerät mit einem Plattform Prozessormodul unter Linux

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Elektrotechnik- und Informationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Rasmus Rettig
Zweitgutachter : Andreas Rix

Abgegeben am 4. Oktober 2017

Mathis Garrandt

Thema der Bachelorarbeit

Erstellung einer Funktionstestumgebung für ein Medizingerät mit einem Plattform Prozessormodul unter Linux

Stichworte

Embedded Linux, Kernel, Linux Distribution, Entwicklungsumgebung, Funktionstest, ARM, i.MX6 SoC, Cyclone V SoC, Bootloader U-Boot

Kurzzusammenfassung

Es soll ein vorhandenes Plattform Prozessormodul, welches in mehreren Medizingeräten zum Einsatz kommt, für Funktionstests eingesetzt werden. Hierzu soll von beiden Prozessoren über ein Linux-Betriebssystem auf die verschiedenen Schnittstellen zugegriffen werden können. Diese Arbeit beschreibt das Aufsetzen eines Linux Betriebssystems auf einem ARM Prozessor und die Einrichtung einer Test- und Entwicklungsumgebung.

Mathis Garrandt

Title of the paper

Development of a function testing environment for a medical device with a platform processor module running Linux

Keywords

Embedded Linux, Kernel, Linux distribution, Development Environment, Function test ARM, i.MX6 SoC, Cyclone V SoC, Bootloader U-Boot

Abstract

An existing platform processor module, used in several medical devices, shall be used for function testing. For this purpose, the different interfaces shall be accessed, using a Linux operating system. This paper describes the installation of a Linux OS on an ARM processor and the set up of a testing and development environment.

Danksagung

Ich möchte mich bei denjenigen bedanken, die mich bei der Erstellung dieser Arbeit sowohl mit fachlichen, als auch mit taktischen Tipps unterstützt haben.
Mein Dank gilt Nathan-Jedidja Hirschauer, der mich mit seinen Linux Kenntnissen unterstützt hat und mir mehr als einmal bei scheinbar undurchdringbaren Problemen den entscheidenden Tipp, oder die passende Hilfestellung gegeben hat.
Arno Morbach hatte auf alle hardwarebezogenen Fragen eine Antwort parat und hat mir die Einarbeitung in die Hardware des M48 durch seine umfangreiche Dokumentation deutlich erleichtert.
Die engagierte Betreuung durch Prof. Dr. rer. nat. Rasmus Rettig und Andreas Rix wusste ich beim Erstellen dieser Arbeit sehr zu schätzen.

Inhaltsverzeichnis

Tabellenverzeichnis **7**

Abbildungsverzeichnis **8**

1. Einführung **10**
1.1. Motivation 10
1.2. Zielsetzung 11
1.3. Aufbau der Arbeit 12

2. Grundlagen **13**
2.1. Hardware 13
2.1.1. Architektur 13
2.1.2. i.MX6 15
2.1.3. Cyclone V 15
2.1.4. Aufbau des M48 15
2.1.5. Testboard 16
2.2. Software 17
2.2.1. Linux 17
2.2.2. Device Trees 21
2.2.3. Bootvorgang 22
2.3. Funktionstest 23

3. Anforderungsanalyse **25**
3.1. Anforderungen an das System 25
3.2. Anforderungen an die Testroutinen 26

4. Systementwurf **28**
4.1. Linux System 28
4.2. Entwicklungsumgebung 29
4.3. Funktionstest 30

5. Realisierung **32**
5.1. Linux System 32
5.1.1. Build-Prozess 33

5.1.2. Anpassen des Bootloaders . . . 36
5.1.3. Anpassen des Device Trees . . . 37
5.1.4. Konfiguration des Kernels . . . 42
5.2. Root Filesystem und Desktopumgebung . . . 43
5.3. Funktionstest . . . 43
5.3.1. Gemeinsame Log-Funktionen . . . 44
5.3.2. i.MX6 I2C On-Module EEPROM . . . 45
5.3.3. i.MX6 I2C Testboard EEPROMs . . . 48
5.3.4. i.MX6 SPI Testboard EEPROMs . . . 50
5.3.5. i.MX6 UARTs . . . 51
5.3.6. i.MX6 GPIOs . . . 53
5.3.7. Safety Logik . . . 58
5.3.8. Konfigurierbare Testsequenz . . . 62

6. Test und Bewertung **65**
6.1. Linux System . . . 65
6.2. Funktionstest . . . 67
6.2.1. Testroutinen . . . 67
6.2.2. Testsequenz . . . 73

7. Ausblick **74**

Literaturverzeichnis **75**

A. DVD **78**
A.1. Bachelorarbeit . . . 78
A.2. Quellcode des Funktionstests . . . 78
A.3. Quellcode des Device Trees . . . 78
A.4. Kernel Konfiguration . . . 78
A.5. Dokumentation . . . 78

Tabellenverzeichnis

3.1. Anforderungen an die Testroutinen für Cyclone V Komponenten 26
3.2. Anforderungen an die Testroutinen für i.MX6 Komponenten 27

4.1. Vergleich von Linux Build Systemen 29

6.1. Zugriff auf i.MX6 Komponenten und Abdeckung durch Testroutinen 66
6.2. Zugriff auf Cyclone V Komponenten und Abdeckung durch Testroutinen . . . 67

Abbildungsverzeichnis

1.1. Beatmungsgerät Dräger Evita® Infinity® V500 [9] 11

2.1. Prozessormodul M48 auf Testboard 14
2.2. Schematische Darstellung des Prozessormoduls M48 auf einem PI-Board, Quelle: Eigene Darstellung 17
2.3. Aufbau eines Linux Systems, Quelle: Eigene Darstellung 19
2.4. Bootvorgang i.MX6 23
2.5. Bootvorgang Cyclone V 23

4.1. Flussdiagramm des Funktionstests 31

5.1. Übersicht: Linux Build-Prozess für den i.MX6, Quelle: Eigene Darstellung . . 35
5.2. SD Karte als Bootmedium, Quelle: Eigene Darstellung 36
5.3. Zusammensetzung des Device Trees am Beispiel des i.MX6 SPI Bus, Quelle: Eigene Darstellung 38
5.4. Untermenü für USB Treiber im Kernel Configuration tool 43
5.5. Debian mit installierter LXDE Desktopumgebung auf dem M48 44
5.6. On-ModuleEEPROM Test, Flussdiagramm 46
5.7. I2C Testboard EEPROM Test, Flussdiagramm 48
5.8. i.MX6 SPI Testboard EEPROM Test, Flussdiagramm 50
5.9. i.MX6 UART Test, Flussdiagramm 52
5.10. i.MX6 GPIO Test, Flussdiagramm 54
5.11. M48 Safety Logik, vereinfachter Schaltplan 59
5.12. i.MX6 Safety Logik Test, Flussdiagramm 60

Abkürzungsverzeichnis

ADC	Analog-to-Digital-Converter
BGA	Ball Grid Array
BLOB	Binary Large Object
CAN	Controller Area Network
DMA	Direct Memory Access
DTB	Device Tree Blob
EDM	Embedded Device Module
FDT	Flattened Device Tree
FPGA	Field Programmable Gate Array
GPIO	General-purpose input/output
HDMI	High Definition Multimedia Interface
HPS	Hard Processor System
IOMUX	Input/Output Multiplexer
IOMUXC	IOMUX Controller
JTAG	Joint Test Action Group
LVDS	Low Voltage Differential Signaling
OS	Operating System
PCB	Printed Circuit Board
PMIC	Power Management Integrated Circuit
RAM	Random-Access Memory
ROM	Read Only Memory
RTC	Real-Time Clock
SoC	System-on-a-Chip
SPL	Second Program Loader
UART	Universal Asynchronous Receiver Transmitter

1. Einführung

1.1. Motivation

Diese Arbeit wurde in der Drägerwerk AG & Co. KGaA durchgeführt. Dräger ist einer der weltweit führenden Hersteller medizintechnischer Geräte. Zu Drägers Produktportfolio gehören unter anderem Beatmungsgeräte wie das in Abbildung 1.1 dargestellte Dräger Evita® Infinity® V500.

In verschiedenen Therapiegeräten von Dräger soll als zentrale Recheneinheit das eigens entwickelte Prozessormodul „M48“ eingesetzt werden. Über einen standardisierten Connector kann das Prozessormodul in die unterschiedlichen Produktintegrations-Boards (PI-Boards) der Geräte eingebunden werden und übernimmt dort die produktspezifischen Regelungsaufgaben. Auf dem M48 läuft ein Echtzeitbetriebssystem, das neben der Regelung die Eingaben des Anwenders verarbeitet und die Bildsignale für angeschlossene Monitore erzeugt.

Die Serienentwicklung der Software für das eingesetzte Echtzeitbetriebssystem vxWorks erfolgt erst später im Entwicklungszyklus. Während der Entwicklung der Elektronik besteht jedoch schon zu einem früheren Zeitpunkt die Notwendigkeit, einzelne Hardwarekomponenten mit geeigneter Software zu testen. Als Alternative zum Betriebssystem vxWorks, soll ein leicht zugängliches Linux System erstellt werden, mit dem schon zu Beginn des Entwicklungszyklusses einfache Testanwendungen für das M48 entwickelt werden können.

Gleichzeitig würde damit auch die Möglichkeit geschaffen, das Prozessormodul unabhängig vom Betriebssystem zu testen, so dass die Ursache für auftretende Fehler schnell auf eines der Betriebssysteme oder die Hardware eingegrenzt werden kann.

Auch ein Einsatz des M48 als universeller Einplatinencomputer in Testanwendungen ist denkbar. Trotz eines signifikanten Preis- und Leistungsunterschiedes bietet sich ein Vergleich mit dem Raspberry Pi an. In Situationen, in denen bisher auf diesen und ähnliche Einplatinencomputer zurückgegriffen wurde, könnte in Zukunft das M48 eingesetzt werden. Das M48 ist in vielen Bereichen der Entwicklungsabteilungen präsent, und das leichter zugängliche Linux System würde es erlauben, das Potential der Hardware auch außerhalb von Dräger-Geräten auszuschöpfen.

Abbildung 1.1.: Beatmungsgerät Dräger Evita® Infinity® V500 [9]

1.2. Zielsetzung

Ziel dieser Arbeit ist es, auf dem M48 eine Linuxbasierte Funktionstestumgebung zu implementieren. Mit dieser soll dann die Funktionsfähigkeit der Schnittstellen getestet werden. Im Rahmen einer Anforderungsanalyse sind zunächst die Anforderungen an das Linux System und die Funktionstestumgebung zu formulieren. Auf Basis der Anforderungen wird dann eine Linux Distribution installiert, die den Zugriff auf alle erforderlichen Schnittstellen ermöglicht. Das M48 soll dann in ein vorhandenes Testboard eingesetzt werden, und mittels eines Shell-Skriptes soll eine automatische Testsequenz ausgeführt werden. Über das Shell-Skript werden Softwarekomponenten aufgerufen, deren Sourcecode, je nach Eignung für die jeweilige Aufgabe, in den Sprachen C oder Python geschrieben ist. Diese Softwarekomponenten sollen dabei so aussagekräftig dokumentiert sein, dass sie von anderen Personen als Ausgangspunkt für die Entwicklung weitergehender Anwendungen genutzt werden können.

1.3. Aufbau der Arbeit

In den *Grundlagen* werden zunächst die elementaren Bestandteile eines Linux Systems vorgestellt. Hier wird auch auf den Bootloader U-Boot und den Ablauf des Bootvorgangs eingegangen. Der Abschnitt *Hardware* gibt einen Überblick über die verwendeten Prozessoren und den Aufbau des Prozessormoduls M48. Weiterhin wird der Begriff des Funktionstests erläutert.

In einer *Anforderungsanalyse* werden die Anforderungen an das System und an die einzelnen Komponenten definiert.

Im Kapitel Systementwurf wird dann die Entscheidungsfindung bei der Auswahl eines Linuxsystems dargestellt. Außerdem wird ein Konzept für den Aufbau der Testsoftware erstellt.

Das Kapitel *Realisierung* beschreibt zunächst die Phase des „Board Bring-Up“. Diese umfasst die Programmierung des Bootloaders, das Anpassen der Device-Trees und die Konfiguration des Kernels. Anschließend werden die Installation eines Debian Root-Filesystem und die Einrichtung einer grafischen Benutzeroberfläche auf dem i.MX6 beschrieben. Die Entwicklung des Funktionstests folgt dem V-Modell [26, S. 29]. Zunächst werden die einzelnen Testroutinen entwickelt und dann in eine Testsequenz eingebunden.

Im Kapitel *Test und Bewertung* werden zunächst die Testroutinen und dann die Testsequenz im Gesamtkontext getestet. Hierbei erfolgt eine Bewertung der Ergebnisse im Hinblick auf die aufgestellten Anforderungen.

2. Grundlagen

In diesem Kapitel werden die Grundlagen der verwendeten Hardware und Software beschrieben. Zunächst wird das Prozessormodul M48 und das dazugehörige Testboard vorgestellt. Dann werden die elementaren Bestandteile eines Linux Systems erläutert und der Ablauf eines Bootvorgangs beschrieben. Außerdem wird der Begriff des Funktionstests in Bezug auf elektronische Baugruppen definiert.

2.1. Hardware

Die Hardware-Plattform für die aufzusetzende Funktionstestumgebung bildet ein von Dräger entwickeltes Prozessormodul mit der Bezeichnung „M48“. Abbildung 2.1 zeigt das M48 in Verbindung mit dem Testboard, das während der Entwicklung des M48 zum Testen des Prozessormoduls genutzt wird.

2.1.1. Architektur

Die Architektur des M48 fällt in die Klasse der eingebetteten Systeme (embedded systems), die sich vor allem dadurch auszeichnen, dass Hardware und Software an eine bestimmte Funktionalität angepasst sind und dabei oft mit geringen Ressourcen auskommen.

Auf dem Prozessormodul befinden sich zwei Ein-Chip-Systeme (engl. System-on-a-Chip, SoC), welche die zentrale Rechenplattform für Anwendungen und Signalverarbeitung des entsprechenden Dräger-Gerätes darstellen. Ein SoC ist ein Bauteil, das die Funktionalität verschiedener integrierter Schaltkreise (IC, Integrated Circuit) auf einem Chip vereint. Neben einem Mikroprozessor sind auf einem SoC oft auch Controller für Schnittstellen und Speicher, Taktgeber und Sensoren wie z. B. ein Temperatur- oder Beschleunigungssensor integriert. Nachfolgend werden die Bezeichnungen „Prozessor“ und „SoC“ synonym verwendet.

In beiden SoCs wird als CPU (Central Processing Unit) ein ARM Cortex-A9 eingesetzt. ARM steht für „Advanced RISC Machine“, eine Prozessorarchitektur, die hohe Leistung bei vergleichsweise niedrigem Energieverbrauch bietet [18, S.12]. Dies wird unter anderem durch

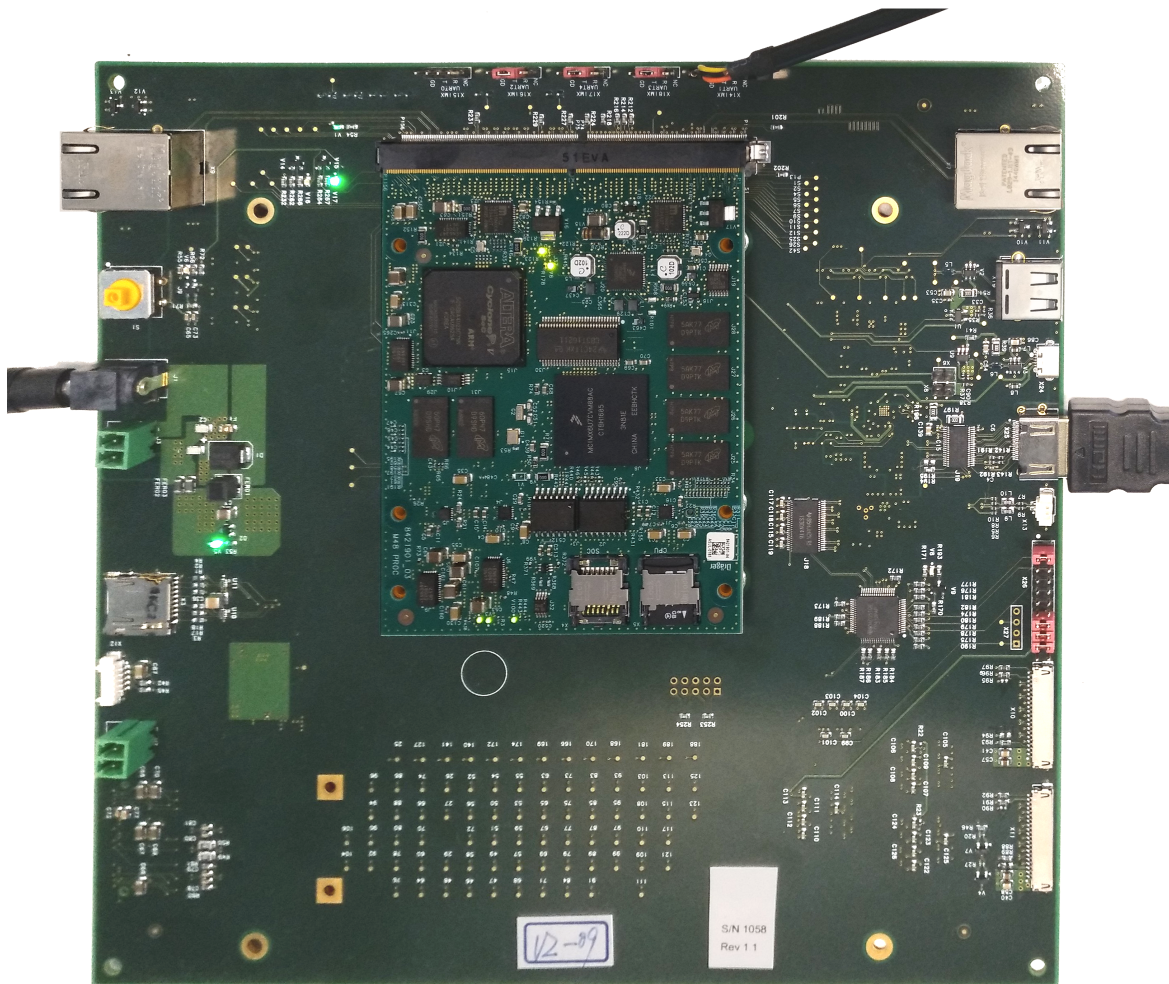

Abbildung 2.1.: Prozessormodul M48 auf Testboard

einen reduzierten Befehlssatz (RISC, Reduced Instruction Set Computer) und eine, im Vergleich zu x86-Prozessoren, geringere Taktfrequenz erreicht. Die ARM Architektur wird vom Unternehmen ARM Limited entwickelt und als Lizenz an Halbleiterhersteller wie z. B. Intel oder NXP verkauft.

Das Prozessordesign Cortex-A9 wird durch die 32-Bit Architektur ARMv7 realisiert. Zur Einordnung der Prozessorleistung kann der Raspberry Pi 2 Model B [20] herangezogen werden. Dort wird ebenfalls die ARMv7 Architektur eingesetzt, allerdings im neueren Prozessordesign Cortex-A7. Der Cortex-A7 ist der Nachfolger des Cortex-A9 und bietet bei vergleichbarer Leistung eine um den Faktor 1,6 höhere Energieeffizienz [2].

2.1.2. i.MX6

Eines der SoC ist ein „i.MX6“ Mikroprozessor von NXP, das vor allem die grafische Benutzeroberfläche (GUI) zur Verfügung stellt [16]. Verwendet wird der i.MX6 Dual Lite, der zwei ARM Cortex-A9 Kerne besitzt, die jeweils mit bis zu 1 GHz getaktet werden können. Auch verfügt dieses SoC über einen 3D-Grafikbeschleuniger, ein Gigabit Ethernet Interface und verschiedene Busse wie z.B. USB, I^2C, SPI und CAN. Das SoC verfügt zudem über einen HDMI Ausgang und zwei LVDS (Low Voltage Differential Signaling) Schnittstellen über die mittels eines Transceivers ebenfalls Monitore mit HDMI Interface angesteuert werden können.

2.1.3. Cyclone V

Das zweite SoC ist ein „Cylone V“ von Altera. Den Hauptbestandteil des SoC bildet ein FPGA (Field Programmable Gate Array), dem ein ARM Cortex-A9 Kern als Hard-Processor-Core (HPS) zur Seite gestellt wurde. Dieses SoC übernimmt gerätespezifische Regelungsaufgaben, wie z.B. die Regelung der Beatmung. Das FPGA dient dabei vornehmlich zum Einlesen der Sensordaten, die von ADCs (Analog to Digital Converter) zur Verfügung gestellt werden. Der Cylone V verfügt neben einem Gigabit Ethernet Interface, einer UART Schnittstelle und einem I^2C Bus über zahlreiche GPIOs (general purpose input/output) die zum Großteil direkt mit dem FPGA verbunden sind.

2.1.4. Aufbau des M48

Abbildung 2.2 zeigt eine schematische Darstellung des Prozessorboards M48 in Verbindung mit einem PI-Board (Produktintegrations-Board), über das die Schnittstellen zur Verfügung gestellt werden. Die Verbindung des M48 mit dem PI-Board wird über zwei standardisierte EDM-Connectoren (Embedded Device Module) hergestellt. Hierdurch kann das Prozessorboard auf verschiedenen produktspezifischen PI-Boards eingesetzt werden.

Jedem der SoC werden 1 GB DDR3 RAM (Random Access Memory) als Hauptspeicher zur Verfügung gestellt. Zudem verfügt jedes der SoC über mehrere MMC[1]- (Multimedia Card) bzw. SDIO- (Secure Digital Input Output) Schnittstellen. An jeweils einem der SDIO Schnittstellen ist ein microSD Karten Slot angebunden. Auf den microSD Karten werden der Bootloader, das Image des Betriebssystems, und das Linux Dateisystem (rootfs) abgelegt. Die microSD-Karte des Cyclone V enthält zusätzlich noch einen Preloader und ein FPGA-Image.

[1] Der SD Standard ist eine Weiterentwicklung des MMC Standards

Jeweils eine weitere SDIO-Schnittstelle dient zur Kommunikation der Prozessoren untereinander. Neben der seriellen-SDIO Schnittstelle wurde auch eine parallele EIM-Schnittstelle implementiert, da sie der Datenübertragung via SDIO bei kleinen Datenmengen überlegen ist. Der vom SD-Protokoll definierte Datenframe enthält neben der zu übertragenden Information immer auch einen Befehl und eine Prüfsumme, und ist dadurch mindestens 6 Byte lang [21]. Dieser „Overhead“ wirkt sich vor allem dann auf die Übertragungsgeschwindigkeit aus, wenn die eigentlich zu sendende Information sehr kurz ist. Die EIM-Schnittstelle erlaubt es aufgrund der parallelen Datenübertragung auch einzelne Bytes schnell zwischen den Prozessoren auszutauschen.

Zusätzlich überwachen sich die Prozessoren gegenseitig durch eine diskret aufgebaute Überwachungslogik (Safety Logik), über die ein Watchdog Verhalten implementiert werden kann. Fällt einer der Prozessoren aus, kann dies erkannt werden und das System wird kontrolliert in einen sicheren Zustand versetzt und gegebenenfalls neugestartet.

Die Spannungsversorgung des i.MX6 erfolgt durch ein dediziertes Power Management IC (PMIC). Dieser PMIC kann über einen I^2C Bus des i.MX6 angesteuert werden, so dass der Energieverbrauch des i.MX 6 mittels Dynamic Voltage Scaling [28] optimiert werden kann. Ebenfalls via I^2C angeschlossen sind eine Echtzeituhr (real-time clock, rtc) sowie ein EEPROM, in dem die MAC-Adresse und weitere Informationen zur Identifikation des Boards abgelegt sind.

Für den Cyclone V steht auf dem M48 ein QSPI Flash Speicher zur Verfügung, auf dem unter anderem das FPGA-Image abgelegt werden kann.

2.1.5. Testboard

Das Testboard (Abbildung 2.1) stellt eine möglichst große Anzahl an Schnittstellen zur Verfügung. In der frühen Implementierungsphase des Linux Systems sind vor allem die UART-Schnittstellen (Universal Asynchronous Receiver Transmitter) von Bedeutung, da an diesen standardmäßig ein tty-Terminal eingerichtet ist, über die das System bedient werden kann. Später werden Maus und Tastatur über die USB-Schnittstelle, sowie ein HDMI-Monitor angeschlossen. Auch die LVDS Signale können durch einen Multiplexer auf den HDMI-Ausgang geschaltet werden. Der Multiplexer kann über einen GPIO Pin angesteuert werden, so dass die Bildquelle in einem Testprogramm automatisch gewechselt werden kann.

Die freien I^2C-Schnittstellen von i.MX6 und Cyclone V, sowie der SPI-Bus des i.MX6 sind auf dem Testboard an EEPROMs angeschlossen, so dass die Schnittstellen ohne Anschluss von externen Komponenten getestet werden können. Die GPIOs und andere frei konfigurierbare Pins von beiden SoCs sind auf dem Testboard paarweise mit Widerständen verbunden. Auch wurden die JTAG Interfaces der Prozessoren herausgeführt, um das Debuggen

zur Laufzeit zu ermöglichen. Die Bootquelle der Prozessoren kann über eine Reihe von Bootselect-Jumpern bestimmt werden.

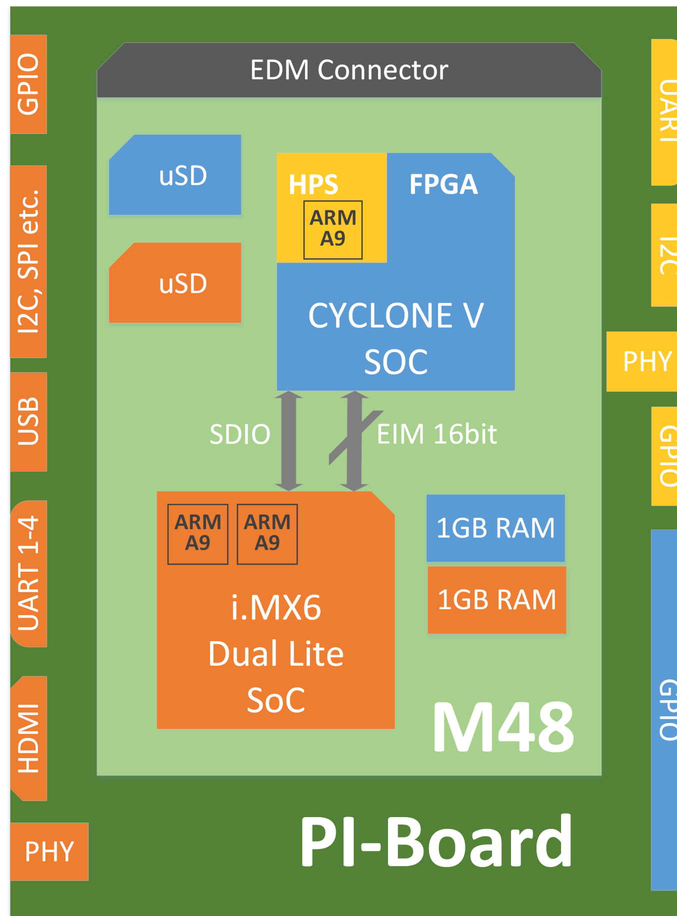

Abbildung 2.2.: Schematische Darstellung des Prozessormoduls M48 auf einem PI-Board, Quelle: Eigene Darstellung

2.2. Software

2.2.1. Linux

Linux ist die Bezeichnung für ein freies Betriebssystem, das auf dem Linux Kernel und GNU-Software basiert. Hinter der Bezeichnung GNU steckt eine Sammlung vollständig freier Software, die unter der GNU General Public License (GPL) [11] steht. Der Linux Kernel wird

ebenfalls unter der GPL veröffentlicht [15], ist aber kein Bestandteil des GNU Projekts. Linux zählt zu den „unixoiden“ oder unix-ähnlichen Betriebssystemen [30], da es Konzepte des 1969 von Bell Laboratories entwickelten Betriebssystems Unix implementiert, dabei die Single UNIX Specification [13] aber nur teilweise umsetzt. Zur GNU Software gehören Entwicklerwerkzeuge wie z. B. die BASH Shell, die Gnu Compiler Collection (GCC) und diverse Bibliotheken, aber auch Anwendersoftware wie das Bildbearbeitungsprogramm GIMP.

Als Distribution wird eine Sammlung von GNU-Software und distributionsspezifischer Software bezeichnet, die in Verbindung mit dem Linux Kernel veröffentlicht wird. Zur distributionsspezifischen Software gehören meist ein Sitzungsmanager und ein Anzeige-Server in Verbindung mit einer Desktopumgebung. Die Verzeichnisstruktur, die die Gesamtheit aller Dienste, Konfigurationsdateien und Anwenderprogramme enthält, wird als Root-Filesystem (rootfs) bezeichnet. Als Abgrenzung zum hardwarenahen Kernel, werden die Verzeichnisse des Root-Filesystems auch Userland genannt. Abbildung 2.3 zeigt den Aufbau eines Linux Systems als Schichtenmodell, das sich aus den beiden Hauptbestandteilen des Betriebssystems und der Hardware zusammensetzt.

Kernel

Der Kernel stellt als Betriebssystemkern die wichtigste Komponente eines Linux Systems dar. Der Kernel bildet die Schnittstelle zwischen Anwenderprogrammen und Hardware und verwaltet alle Ressourcen. Die Aufgaben des Kernels lassen sich im wesentlichen fünf Funktionseinheiten zuordnen: Gerätetreiber (Device Driver Layer), Steuerung von Ein-und Ausgabe (IO-Management), Verwaltung der Prozesse (Process Management), Verwaltung des Speichers (Memory Management) und die Schnittstelle für Systemaufrufe (Systemcall-Interface). Diese Gliederung der Kernel Bestandteile orientiert sich an der Darstellung in [18, S. 16].

Der in Abbildung 2.3 ebenfalls als Bestandteil des Kernels dargestellte Device Tree ist keine aktive Komponente des Kernels. Der Device Tree wird während des Bootvorgangs in den Kernel eingebunden und stellt dort eine virtuelle Repräsentation aller Hardwarekomponenten dar. Erst anhand des eingebundenen Device Trees weiß der Kernel, welche Hardware Ressourcen ihm zur Verfügung stehen.

Die Gerätetreiber stellen dem Kernel eine einheitliche Schnittstelle zum Zugriff auf verschiedene Hardwarekomponenten zur Verfügung. Der Entwickler einer Hardwarekomponente muss entweder sicherstellen, dass die Hardware, z. B. ein Drucker, mit den im System vorhandenen Standardtreibern kompatibel ist, oder zusammen mit dem Gerät einen gerätespezifischen Treiber ausliefern. Der Treiber übersetzt standardisierte Lese- oder Schreibbefehle des Kernels in die zur Ansprache der Hardware notwendigen, proprietären Befehle.

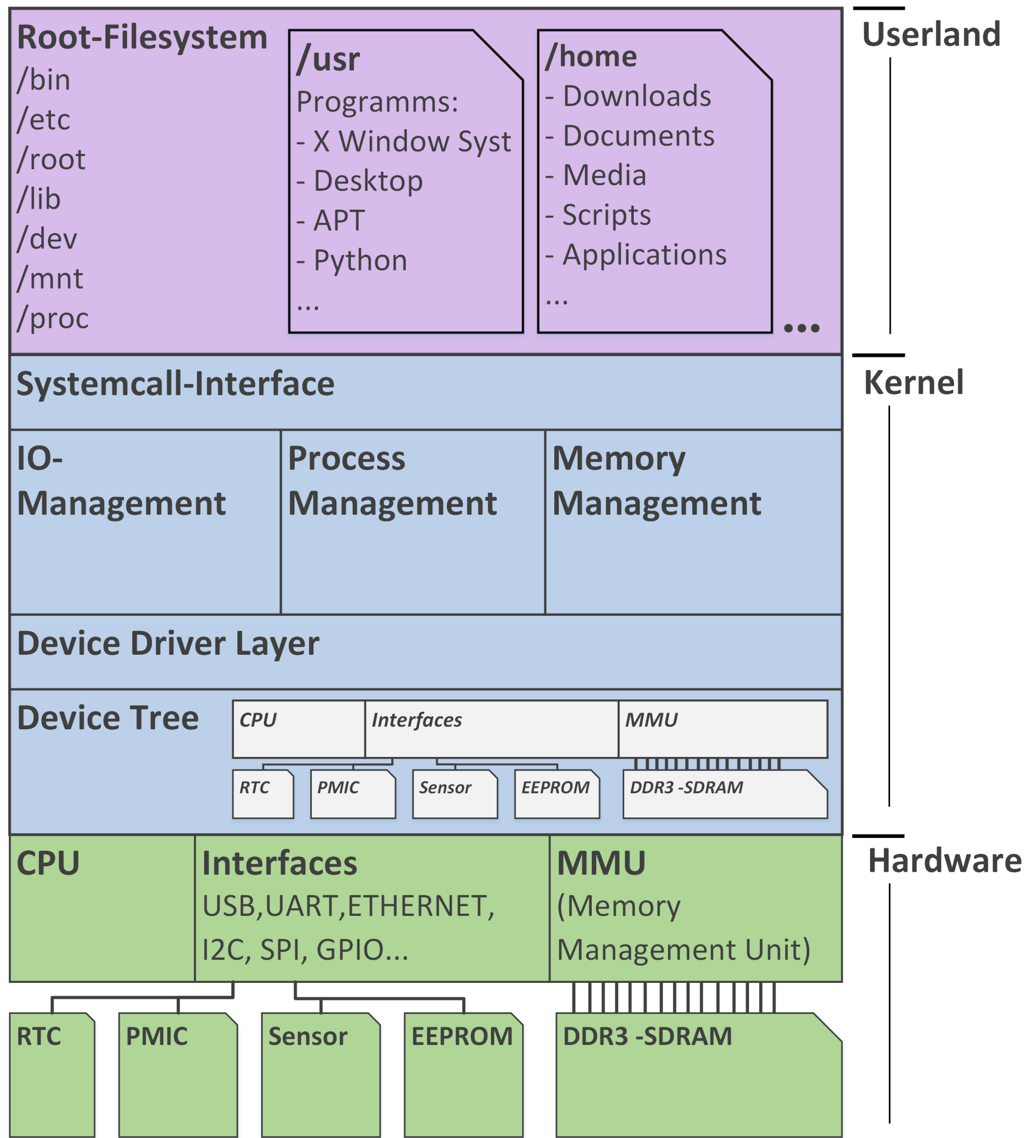

Abbildung 2.3.: Aufbau eines Linux Systems, Quelle: Eigene Darstellung

Das Systemcall-Interface stellt die Schnittstelle des Kernels zum Userland dar. Anwenderprogramme nutzen Systemaufrufe, um auf Funktionen zuzugreifen, die vom Betriebssystem zur Verfügung gestellt werden. Bei Verwendung der Funktion *fopen()* zum Öffnen einer Datei beispielsweise, greift die in der C-Bibliothek implementierte Funktion auf das Systemcall-Interface zu und fordert dort den Dateizugriff an.

Das IO-Management ist für Zugriffe auf die Peripherie und die Verwaltung der Dateisysteme zuständig. Das IO-Management verwaltet auf der einen Seite Zugriffsanforderungen von Seiten des Systemcall-Interface und stellt auf der anderen Seite das Treiberinterface zur Verfügung, mit dem die Gerätetreiber in den Kernel eingebunden werden [18, S. 18].

Das Prozess-Management verwaltet die parallele Abarbeitung der Prozesse und Threads, in denen die aktiven Programme laufen. Diese quasi-parallele Abarbeitung der Aufgaben wird als Task-Scheduling bezeichnet Es können dabei verschiedene Scheduling-Verfahren angewendet werden. Im Bereich eingebetteter Systeme ist vor allem das prioritätengesteuerte Scheduling relevant, mit dem Realzeiteigenschaften realisiert werden können [18, S. 16].

Das Memory Management übernimmt die Verwaltung der Speicherressourcen des Systems. Bei einem Prozessor-System mit separatem Hauptspeicher (RAM) sind logischer und physischer Adressraum zu unterscheiden. Programme greifen über logische Adressen auf den Hauptspeicher zu. Der für Anwendungen zur Verfügung stehende logische Adressraum auf einem 32-Bit System hat für gewöhnlich eine Größe von 3 GByte. Diesem logischen Adressraum steht ein physischer Adressraum von, im Fall des M48, 1 GByte gegenüber. Die Umsetzung des physischen in den logischen Adressraum kann vom Memory Management vorgenommen werden.

Bei dem in Abbildung 2.3 dargestellten System wird diese Umsetzung der Adressräume von einer in Hardware angelegten Memory Management Unit (MMU) vorgenommen. Solch eine MMU ist auch in dem auf dem M48 verwendeten i.MX6 vorhanden. Das im Kernel integrierte Memory Management ist außerdem dafür zuständig, virtuellen Speicher zur Verfügung zu stellen und verschiedene Speicherbereiche voneinander abzugrenzen. Jedem Prozess wird ein virtueller Speicherbereich zur Verfügung gestellt, den das dort laufende Programm zur freien Verfügung hat. Gleichzeitig stellt das Memory Management sicher, dass kein Prozess Zugriff auf den Speicherbereich eines anderen Prozesses hat.

Userland

Das Userland stellt den Bereich des Linux Systems dar, den der Anwender durch das Installieren und Deinstallieren von Anwendungen und Paketen nach seinen Belieben anpassen kann. Das Starten von Diensten oder die Netzwerkkonfiguration werden durch einen dort aktiven Sitzungsmanager vorgenommen. Im Userland können ein Fenstermanager und eine

Desktopumgebung installiert werden, um das System über eine grafische Benutzeroberfläche zu bedienen.

Das zugrunde liegende Root-Filesystem ist ausgehend von einem Root-Verzeichnis aufgebaut. In dem Root-Verzeichnis liegen verschiedene systemrelevante Verzeichnisse. Einige der Verzeichnisse sind nicht als reale Verzeichnisse auf einem Datenträger angelegt, sondern vom System zur Laufzeit erzeugte, virtuelle Verzeichnisse. Eine der definierenden Unix-Eigenschaften die auch bei Linux Anwendung findet, lautet: „Everything is a file!“. Viele Bestandteile des Systems, wie z.B. Prozesse (Verzeichnis /proc) oder Hardwarekomponenten (Verzeichnis /dev) werden als virtuelle Dateien im Userland zur Verfügung gestellt. Der Zugriff auf diese virtuellen Dateien erfolgt, wie bei herkömmlichen Textdateien, über Schreib- und Lesebefehle.

Der Zugriff auf Hardwarekomponenten kann dabei sowohl über das Verzeichnis */dev* als auch das Verzeichnis */sys* (sysfs) erfolgen. Im Verzeichnis */dev* liegen meist die Dateien, über die direkt auf die Komponenten zugegriffen werden kann (UARTs, I^{2}C, Memory), während unter */sys* Statusinformationen der Komponenten abgerufen werden können. Einige Treiber, wie die im Rahmen dieser Arbeit verwendeten EEPROM-Treiber, hinterlegen die virtuelle Datei (eeprom) nur im */sys*-Verzeichnisbaum. Der Pfad dieses Verzeichnisbaums ist dabei äquivalent zum Zweig der Komponente im Device Tree.

2.2.2. Device Trees

Ein Device Tree ist eine Datenstruktur, die die Hardware eines Systems beschreibt.
Device Trees sind unabhängig von einem bestimmten Betriebssystem konzipiert und werden neben Linux, unter anderem auch von den Betriebssystemen vxWorks und QNX verwendet. Sie bestehen aus Verzweigungspunkten (Nodes), die die einzelnen Hardwarekomponenten und Schnittstellen über deren Eigenschaften (Properties) definieren. Die Properties für einen bestimmten Gerätetyp (z.B. „gpio“) müssen einer bestimmten Struktur entsprechen, welche als „Binding“ bezeichnet wird. Anhand dieser Bindings kann der Kernel die Hardwarekomponente korrekt ansprechen. So existieren beispielsweise Bindings für die Prozessorkerne, Busse, serielle Schnittstellen, GPIO-Controller, aber auch für einzelne Komponenten, wie den Temperatursensor der CPU.

Lange Zeit war die Hardwarebeschreibung als „Open Firmware“ fest in den Linux Kernel integriert. Damit war der Kernel dann aber nur auf einem bestimmten Prozessordesign lauffähig. Mit der fortschreitenden Verbreitung der Arm-Architektur und der damit einhergehenden Diversifizierung der Prozessoren, stellten die Device Trees die Lösung dar, um universelle Kernels zu erstellen [19]. Ein Kernel, der für die ARM-Architektur kompiliert wird, ist dabei in verschiedenen Systemen lauffähig, da ihm der Device Tree erst während des Bootvorgangs zur Seite gestellt wird.

Die wichtigste Eigenschaft eines Device Trees ist die *compatible* Eigenschaft. Im *root node* definiert sie das Board, sowie den SoC, mit dem der Device Tree kompatibel ist. In den übrigen Nodes stellt *compatible* die Beziehung zu einer Kernelkomponente her - in den meisten Fällen ein Treiber. Weitere Eigenschaften sind unter anderem die Adresse und Breite des entsprechenden input/output (I/O)-Registers, der zugeordnete Interrupt und der *status*, über den die Komponente aktiviert oder deaktiviert werden kann.

Der Quellcode eines Device-Tree wird vom Device Tree Compiler (dtc) übersetzt. Das Ergebnis ist ein Device Tree Blob[2] (DTB), der vom Kernel beim Booten interpretiert werden kann. Gleichbedeutend mit DTB wird auch die Bezeichnung Flattened-Device-Tree (FDT) verwendet.

2.2.3. Bootvorgang

Das Booten des Linux Systems geschieht bei beiden SoC in mehreren Stufen, wobei jede Stufe dafür zuständig ist, die nächste Stufe zu laden.

Abbildung 2.4 zeigt den Ablauf des Bootvorgangs auf dem i.MX6 SoC und Abbildung 2.5 den Bootvorgang auf dem Cyclone V. Dargestellt sind die Software-Komponenten, die in den einzelnen Stufen geladen werden. Die Abbildungen orientieren sich an der Darstellung des Bootvorgangs in der „Booting and Configuration Introduction“ zum Cyclone V [1, S. 3].

Die erste Stufe bildet das in beiden Prozessoren integrierte Boot ROM, das die „Firmware“ des SoC enthält. Die Bezeichnung „Boot ROM“ wird dabei meist synonym sowohl für den Speicherort, als auch für die dort abgelegte „Firmware“ verwendet [5, S.13; 1, S.3]. Beim Reset des Systems führt der dort liegende Code erste Initialisierungen durch. Caches werden aktiviert, Taktgeber (PLLs) werden konfiguriert, der Flash-Controller wird initialisiert und die Boot-Quelle wird anhand eines Bootselect-Registers bestimmt.

In der nächsten Stufe lädt der i.MX6 dann den Bootloader, der als Image auf einem Flash Speicher - wie einer microSD-Karte - abgelegt sein kann. Hierbei wird zunächst der sehr kompakte „Second Programm Loader“ (SPL), der Bestandteil des U-Boot Images ist, in den on-chip RAM geladen. Der SPL lädt dann den eigentlichen Bootloader in den Hauptspeicher (RAM). Verwendet wird der in Embedded Systemen weit verbreitete Bootloader U-Boot (offizieller Name „Das U-Boot“)[7]. Der Bootloader muss speziell für das vorliegende Board kompiliert worden sein, da er schon diverse Hardwareinitialisierungen vornimmt, um z. B. auf den Ethernet Controller oder einen SPI Bus zuzugreifen.

Im nächsten Schritt lädt der Bootloader dann das Kernel Image und den Device-Tree-Blob aus einem Dateisystem, das sich entweder auf einem Server im Netzwerk oder lokal, auf

[2]BLOB = Binary Large Object

einem Flashspeicher, befindet. Die beiden Binärdateien werden im RAM abgelegt und dann die Kontrolle an den Kernel abgegeben. Der Kernel dekomprimiert sich, initialisiert die Hardware entsprechend des Device Trees und bindet schließlich das Root Filesystem ein, aus dem heraus diverse Dienste gestartet werden.

Der Bootvorgang des Cyclone V verläuft in großen Teilen identisch, beinhaltet aber aufgrund der engen Verknüpfung des HPS mit dem FPGA noch eine zusätzliche Stufe. Während beim i.MX6 der im Device Tree konfigurierte IOMUX-Controller das Routing der Signale zu den entsprechenden Pins vornimmt, übernimmt diese Aufgabe im Cyclone V der Preloader.
Der Preloader wird direkt vom Boot-ROM geladen und nimmt neben dem Pin-Multiplexing auch die Konfiguration der Clocks und die Initialisierung der einzelnen Speicher-Controller vor. Erst dann lädt der Preloader den Bootloader. Neben dem Kernel-Image lädt U-Boot auf dem Cyclone V auch das Image für das FPGA. Das FPGA Image wird dem im HPS integrierten FPGA-Manager übergeben, welcher dann den FPGA konfiguriert.

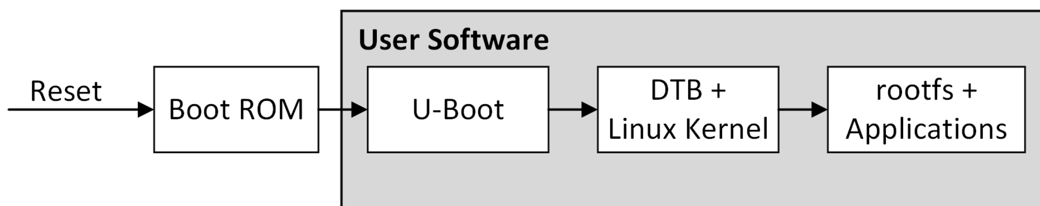

Abbildung 2.4.: Bootvorgang i.MX6

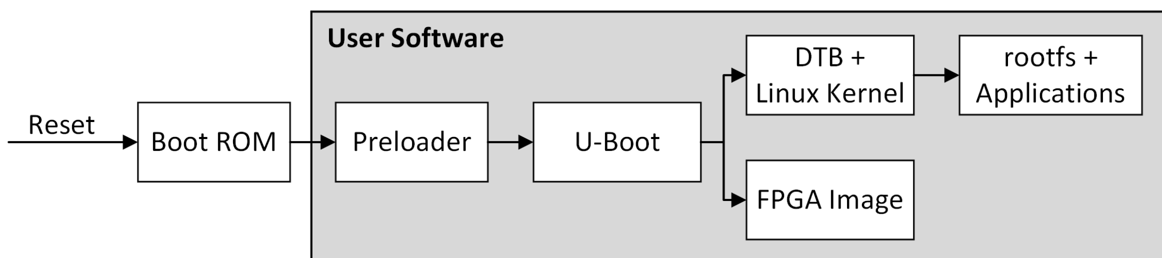

Abbildung 2.5.: Bootvorgang Cyclone V

2.3. Funktionstest

Als Funktionstest bezeichnet man die Prüfung einer Funktionseinheit gegen deren funktionale Anforderungen [29]. Funktionstests können zu verschiedenen Zeitpunkten während der

Lebenszeit des Produkts erfolgen. In Form einer Endprüfung nach Fertigung der Funktionseinheit (Band-Ende-Test), nach dem Einbau in das System, zu dem die Funktionseinheit gehört (Integrationstest), oder als regelmäßiger Routinetest während der Produktlebenszeit.

Bestandteil des Band-Ende-Tests bei elektronischen Baugruppen ist meist ein In-Circuit-Test (ICT), bei der die Baugruppe, im allgemeinen eine Leiterkarte, auf Fehler in der Leiterbahnführung, sowie Bestückungs- und Lötfehler geprüft wird. Hierbei können in erster Linie die analogen Parameter von Bauteilen getestet werden [10].

Soll auch die Funktion der elektronischen Baugruppe im Kontext des Gesamtsystems, d.h. im Endprodukt, getestet werden, muss eine Form des Integrationstests durchgeführt werden. Das Endprodukt kann allerdings unter Umständen nicht dazu geeignet sein, die Baugruppe hinreichend umfangreich zu testen. Es kann z. B. die Anforderung bestehen, Fehlerfälle zu testen, die sich am Endprodukt nicht kontrolliert produzieren lassen, der Funktionstest soll standardisiert und automatisch protokolliert werden, oder das Gesamtsystem ist aufgrund seiner Größe nicht dazu geeignet, beim Fertiger der Baugruppe vorgehalten zu werden.

An Stelle eines Integrationstests kann vor der Endmontage dann ein Elektro-Emulations-Test durchgeführt werden. Der Test simuliert die reale Umgebung und erzwingt so Schaltvorgänge, die sonst nur im eingebauten Zustand überprüft werden können. Dabei werden Spannungen, Ströme und andere Messgrößen so eingestellt und Schaltfolgen so abgearbeitet, wie sie in der Realität ablaufen werden [27]. Die Folgen der einzelnen Testsequenzen können aufgezeichnet und in Form eines standardisierten Testprotokolls abgelegt werden.

3. Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an das Linux System und die Funktionstestumgebung definiert. Die hier definierten Anforderungen bilden im weiteren Verlauf die Basis für Entscheidungen beim Entwurf des Systems und werden später als Referenz bei der Bewertung des fertigen Systems herangezogen.

3.1. Anforderungen an das System

Erstellt werden soll eine Testumgebung, die die Funktion aller aus dem Prozessormodul M48 herausgeführten Schnittstellen verifizieren kann. Ziel der in dieser Arbeit entwickelten Softwarekomponenten ist dabei eine Überprüfung des Boards auf Hardwarefehler, die durch das Testen der Grundfunktionalitäten der einzelnen Schnittstellen realisiert wird. Diese Überprüfung soll durch eine automatische Sequenz von Funktionstests erfolgen, an deren Ende eine „pass" oder „fail" Entscheidung steht.

Das System soll dafür einen Zugriff auf alle nach außen geführten Schnittstellen ermöglichen. Zwischen i.MX6 und dem Cyclone V soll zudem eine Kommunikation über die EIM- sowie über die SDIO-Schnittstelle möglich sein.

Bei der Auswahl der Linux Distribution liegt der Fokus mehr auf der Flexibilität denn auf der Performance. Auf dem laufenden System sollen zu jedem Zeitpunkt Softwarepakete nachinstalliert werden können, um für eine Aufgabe das am besten geeignete Werkzeug wählen zu können. Das Linux System und die darauf eingerichtete Entwicklungsumgebung sollen für neue Anwender schnell zugänglich sein, und Anwendungen sollen sich in der vorhandenen Entwicklungsumgebung ohne umfangreiche Anpassungen erstellen lassen. Auch neue Hardware soll mit geringem Aufwand eingebunden werden können.

Eine flüssige Bedienung des Systems soll durch eine Distribution gewährleistet werden, die für die begrenzten Ressourcen eines Embedded Systems ausgelegt wurde und sich im Auslieferungszustand auf einen grundlegenden Funktionsumfang beschränkt. Die Bedienung des Systems soll dabei sowohl über Maus und Tastatur, als auch über eine Remote-Verbindung via Netzwerk oder serieller Schnittstelle möglich sein.

3.2. Anforderungen an die Testroutinen

Die Tabellen 3.2 und 3.1 listen die Anforderungen an Testroutinen für die Komponenten des i.MX6, respektive Cyclone V. Alle auf dem M48 herausgeführten, bzw. zwischen den SoC implementierten Schnittstellen, werden durch die gelisteten Komponenten abgedeckt. Die Testroutinen sollen, je nach Eignung für die jeweilige Aufgabe, als Shellskript oder in den Sprachen C oder Python implementiert werden und die Grundfunktionalität der jeweiligen Komponente testen. Dabei sollen sie so aussagekräftig dokumentiert sein, dass sie von anderen Personen als Ausgangspunkt für die Entwicklung weitergehender Anwendungen genutzt werden können.

Tabelle 3.1.: Anforderungen an die Testroutinen für Cyclone V Komponenten

No.	Komponente	geprüfte Eigenschaft	Parameter
1	I2C_0 Testboard EEPROM Addr: 0x50	Besteht Lese- und Schreibzugriff?	Übertragungsrate 100 kbit/s, Zugriff auf virtuelle Datei
2	UART 0	Können Daten über Loopback geschrieben und gelesen werden?	115200 Baud, 8n1
3	Safety Logik	Logik triggerbar?	50 Hz Triggersignal
4	GPIOs	Wird der Zustand eines Ausgangspins am verbundenen Eingangspin erkannt?	GPIOs paarweise mit $1k\Omega$ verbunden Zustände '0' und '1' testen
5	SDIO	Datenaustausch mit i.MX6 möglich?	Zu spezifizieren
6	EIM Parallel Interface	Datenaustausch mit i.MX6 möglich?	Zu spezifizieren
7	QSPI Flash	Besteht Lesezugriff?	50 MHz Takt (Read), Übertragungsrate 6.25 Mbyte/s

Tabelle 3.2.: Anforderungen an die Testroutinen für i.MX6 Komponenten

No.	Komponente	geprüfte Eigenschaft	Parameter
1	I2C_2 On-Module-EEPROM Addr: 0x50	Besteht Lesezugriff? Ist die richtige Partnumber hinterlegt?	Übertragungsrate 100 kbit/s, Zugriff auf virtuelle Datei
2	I2C_3 Testboard EEPROM Addr: 0x50	Kann ein Datenmuster geschrieben, und korrekt ausgelesen werden?	Übertragungsrate 100 kbit/s, Zugriff auf virtuelle Datei
3	I2C_4 Testboard EEPROM Addr: 0x50	Kann ein Datenmuster geschrieben, und korrekt ausgelesen werden?	Übertragungsrate 100 kbit/s, Zugriff auf virtuelle Datei
4	SPI_3 EEPROMs an SS1, SS2, SS3, SS4	Kann ein Datenmuster geschrieben, und korrekt ausgelesen werden?	Clock: 5 MHz SPI Mode 0: CPOL=0, CPHA=0
5	SPI_3 Generisches SPI-Device an SS4	Können einzelne Bytes geschrieben, und korrekt ausgelesen werden?	Clock: 5 MHz SPI Mode 0: CPOL=0, CPHA=0 Direkter Zugriff auf /dev/spidev
6	USB Host	Besteht Lese- und Schreibzugriff auf einen Flashspeicher?	USB 2.0 Spezifikation min. 500 mA I_{BUS}
7	USB OTG	Besteht Lese- und Schreibzugriff auf einen Flashspeicher?	USB 2.0 Spezifikation min. 500 mA I_{BUS}
8	UART 1-4	Können Daten über Loopback geschrieben und gelesen werden?	115200 Baud, 8n1
9	GPIOs	Wird der Zustand eines Ausgangspins am verbundenen Eingangspin erkannt?	GPIOs paarweise mit $1k\Omega$ verbunden Zustände ’0’ und ’1’ testen
10	Safety Logik	Logik triggerbar?	50 Hz Triggersignal
11	HDMI	wird ein Bild ausgegeben?	1280x720p
12	LVDS1 LVDS2	wird ein Bild ausgegeben?	1280x720p Jeweils ein Interface via MUX an HDMI Ausgang geschaltet
13	Ethernet Interface	Netzwerkverbindung?	Datenrate > 100 Mbit/s
14	SDIO	Datenaustausch mit Cyclone V möglich?	Zu spezifizieren
15	EIM Parallel Interface	Datenaustausch mit Cyclone V möglich?	Zu spezifizieren
16	PCI Express	PCIE Karte ansteuerbar?	PCI Express 2.0 Standard

4. Systementwurf

In diesem Kapitel erfolgt der Entwurf des Linux Systems und der Funktionstestumgebung. Auf Basis der aufgestellten Anforderungen werden verschiedene Build Systeme verglichen und eine Linux Distribution wird ausgewählt. Zudem werden Konzepte für eine Entwicklungsumgebung und den Funktionstest entwickelt.

4.1. Linux System

Die Basis der Funktionstestumgebung soll eine Linux Distribution bilden. Es muss entschieden werden, ob dafür auf eine der gängigen Distributionen wie z. B. Debian oder Ubuntu zurückgegriffen wird, oder ob die Distribution mittels eines Build Systems individuell erstellt wird. Die hauptsächlich genutzten Systeme für letztere Option sind Yocto [17] und Buildroot [6]. Beide Systeme basieren auf einem Layer/Recipe-Prinzip, bei dem die Komponenten des Linux Systems individuell zusammengestellt werden. Tabelle 4.1 zeigt die Unterschiede der beiden Systeme [4], und den Vergleich mit einer Debian Distribution.

Bei beiden Build Systemen ist das Board Support Package für die Zielhardware integriert, so dass neben dem Root Filesystem auch ein Bootloader und der Device Tree erzeugt werden. Das Einpflegen der M48 Board-Support Dateien in eines der beiden Systeme stellt aber zunächst keine Option dar, da bereits ein funktionierendes Image des Bootloaders vorhanden ist. Das Erstellen eines Root-Filesystems mit Yocto würde eine zu große Einarbeitung erfordern, kann aber für eine spätere Weiterentwicklung des Linux Systems in Betracht gezogen werden.

Mit Buildroot soll sich ein Root-Filesystem ohne hohe Einstiegshürden erstellen lassen. Buildroot kann dennoch nicht verwendet werden, da es die Anforderung der flexiblen Nachinstallation von Software (kein Package Management) nicht erfüllt und mangels Toolchain keine On-Target Entwicklung erlaubt.

Anhand des Vergleichs konnte schnell die Entscheidung gefällt werden, dass für das zu entwickelnde System auf die Distribution Debian zurückgegriffen wird. Ein Debian Root-Filesystem kann mit dem Tool Debootstrap [22] erstellt und angepasst werden. Das so erstelle Debian kann eine Größe von einigen hundert Megabyte annehmen und enthält dabei

auch nicht benötigte Software. Dennoch genügt es den Anforderungen, da Größe und Effizienz des Systems gegenüber der Flexibilität niedrig priorisiert sind.

Als Bootloader wird die für das M48 bereits vorhandene U-Boot Version genutzt. Der Linux Kernel und der Device Tree werden separat konfiguriert und kompiliert.

Tabelle 4.1.: Vergleich von Linux Build Systemen

Build System	**Größe**	**Komplexität**	**Konfigurierbar**	**Package Management**	**BSP integriert**	**On-Target Toolchain**	**Output**
Yocto	Variabel	Hoch	Ja	Ja	Ja	Ja	Distribution
Buildroot	Klein	Gering	Ja	Nein	Ja	Nein	rootfs
Debian + BSP	Größer	Gering	Ja	Ja	Nein	Ja	rootfs

4.2. Entwicklungsumgebung

Ein entscheidender Unterschied bei der Entwicklung von Anwendungen für einen PC oder für ein eingebettetes System ist die Leistungsfähigkeit der Zielhardware. Da dem eingebetteten System in der Regel weniger Ressourcen zur Verfügung stehen, werden Anwendungen oft mit einem Host-Target Transfer entwickelt. Dabei wird die Anwendung auf einem Host-System geschrieben und kompiliert und erst die Ausführbare Datei auf Das Zielsystem übertragen.

Unterscheidet sich die Architektur des Targets zudem noch von der Architektur des Hosts, wird von Cross-Entwicklung gesprochen. Um beispielsweise auf einem Rechner mit x86 Architektur Anwendungen für einen ARM Prozessor zu kompilieren, muss auf eine Cross-Compiling-Toolchain zurückgegriffen werden.

Für die Entwicklung der Testroutinen für das M48 kann eine Host-Target Cross-Entwicklung in Betracht gezogen werden, da ein separater Entwicklungsrechner eine größere Entwicklungsumgebung mit bereits installierter Versionsverwaltung bietet.

Die für das On-Target-Kompilieren benötigte Zeit fällt bei dem geringen Umfang der Testroutinen jedoch nicht ins Gewicht. Die Entwicklung der Testroutinen soll daher direkt auf dem M48 erfolgen. Durch das Einrichten der Entwicklungsumgebung auf dem M48 wird die Anforderung einer portablen und schnell einsetzbaren Entwicklungsumgebung erfüllt. Der Entwickler

greift von seinem Host-Rechner über eine Remote-Schnittstelle auf das M48 zu und erstellt dort die Programme, ohne eine Entwicklungsumgebung auf dem eigenen Rechner einrichten zu müssen.

Die Gnu Compiler Collection, sowie eine Python-Version sind unter Debian bereits installiert. Zusätzlich soll die Paketverwaltung pip installiert werden, um Python-Module unkompliziert nachzuinstallieren. In einem Verzeichnis der Entwicklungsumgebung wird ein Makefile angelegt, das die einzelnen Komponenten eines C-Programms automatisch kompiliert und linkt. Das für die Testroutinen angelegte Makefile kann angepasst werden um damit weitergehende Programme zu erstellen.

4.3. Funktionstest

Der Funktionstest soll in Form einer automatisch Testsequenz realisiert werden. Abbildung 4.1 zeigt den Ablauf des Funktionstests als Flussdiagramm.

Der Ablauf der Testsequenz wird in einem Shell-Skript beschrieben, das die einzelnen Testroutinen aufruft und deren Rückgabewerte auswertet. Der Ablauf des Tests wird in einem Log-File dokumentiert, auf das sowohl die Shellskripte als auch die in C und Python geschriebenen Testroutinen Zugriff haben.

Für alle C-, bzw. Python-Programme existiert jeweils eine gemeinsame Log-Funktion *Put_log()*, die zentral definiert ist und von den Programmen eingebunden wird. In diesen Log-Funktionen kann die Ausgabe der Meldungen zentral angepasst werden. Meldungen können zusätzlich auf die Konsole umgeleitet werden, und auch das nachträgliche Hinzufügen eines Zeitstempels ist denkbar.

Die Konfiguration des Tests erfolgt in einer separaten Konfigurationsdatei, die zu Beginn der Testsequenz eingelesen wird. Diese Konfigurationsdatei kann mit einem Texteditor bearbeitet werden, um Testroutinen anhand von zuvor definierten Variablen an- oder abzuwählen.

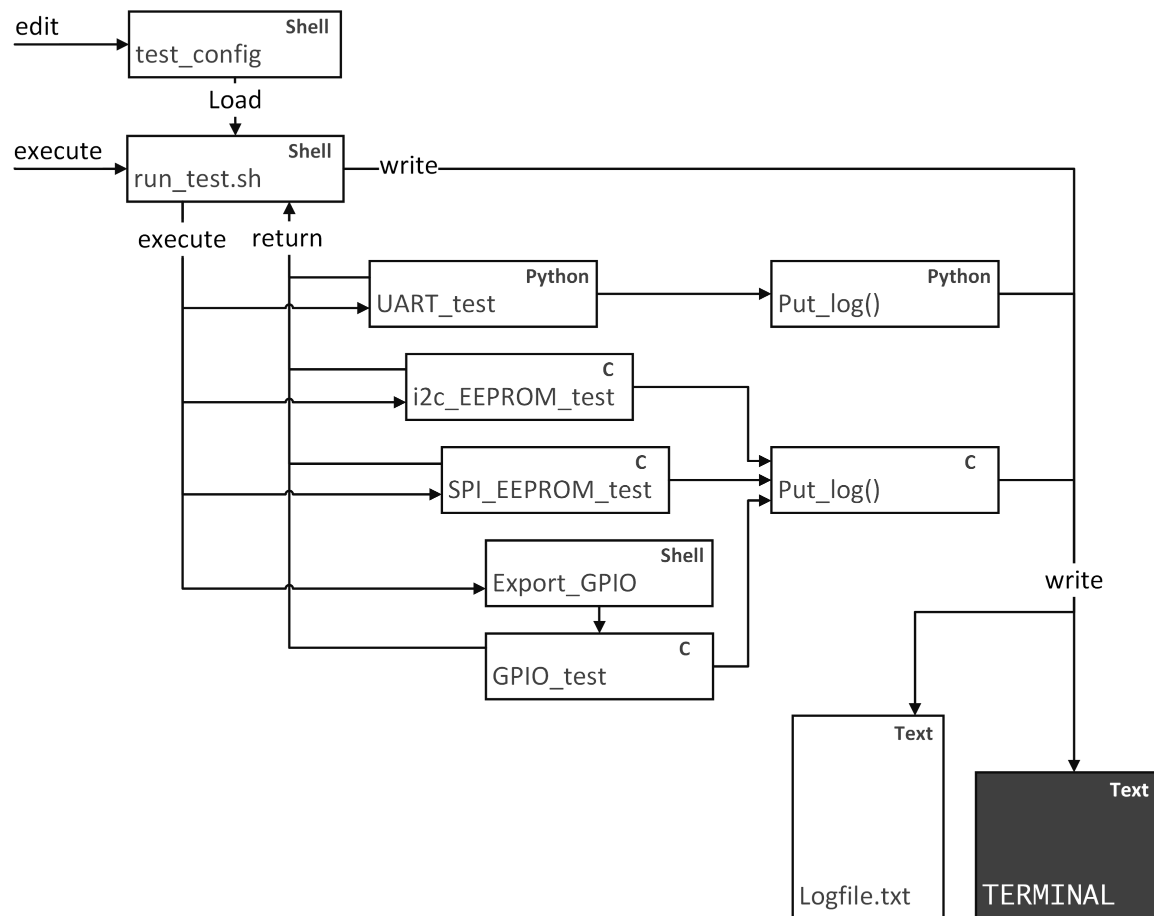

Abbildung 4.1.: Flussdiagramm des Funktionstests

5. Realisierung

Im Folgenden werden die Installation des Linux Systems und die Erstellung des Funktionstests beschrieben. Im Hinblick auf die Linux-Installation wird vor allem der Board-Bring-Up-Prozess beschrieben, der das Einrichten des Bootloaders, die Konfiguration des Kernels und das Anpassen des Device Trees umfasst. Im Abschnitt Funktionstest werden die Struktur der Testsequenz und die Implementierung der Testroutinen beschrieben.

5.1. Linux System

Voraussetzung für einen umfassenden Hardwaretest ist die korrekte Pin-Zuordnung aller Komponenten und die Einbindung der passenden Gerätetreiber in das Linux System. Das geschieht vornehmlich durch die Beschreibung der gesamten Hardware im Device Tree und das Setzen der passenden „compatible-properties“ in den einzelnen Nodes. Bei der Konfiguration des Kernels werden dann alle benötigten Treiber ausgewählt. Zudem muss auch der Bootloader durch diverse Source- und Konfigurationsdateien an das Board angepasst werden. Dieser Vorgang der erstmaligen Hardwarebeschreibung eines Boards wird auch als Board-Bring-Up bezeichnet.

Die Basis des Systems ist der von Linus Torvalds, in dessen github Repository [24] veröffentlichte Linux „Mainline“ Kernel. Neben der eigentlichen Kernel Source enthält das Repository unter anderem auch die Device Tree Dateien aller offiziell unterstützten Boards und eine umfangreiche Dokumentation.

Der Hersteller des i.MX6 SoC, NXP, hat zusammen mit der Firma Boundary Devices ein Prozessormodul mit dem Namen Sabre Lite [8] (SABRE, Smart Application Blueprint for Rapid Engineering) als Referenzhardware für den i.MX6 erstellt. Das Layout des i.MX6 auf dem von Dräger entwickelten M48 entspricht in großen Teilen dem Sabre Lite Referenzdesign.

Daher kann der Device Tree des Sabre Lite als Ausgangspunkt für den Device Tree des M48 genutzt werden. Die Unterschiede des M48 gegenüber dem Sabre Lite liegen vor allem in zahlreichen Änderungen des Pinouts für einzelne Komponenten, die durch den IOMUX Controller (Abschnitt 5.1.3) frei belegt werden können. Zudem besitzt das M48 gegenüber dem Sabre Lite zusätzliche Komponenten, wie z. B. das Power-Management-IC und ei-

ne Realtime-Clock oder abweichende Komponenten, z. B. einen anderen Ethernet-Phy und einen anderen microSD-Karten-Slot.

Der für das M48 vorhandene Device Tree, der von vxWorks genutzt wird, konnte nur als Referenz für die Konfiguration des IOMUX Controllers genutzt werden. Obwohl Device Trees unabhängig von Betriebssystemen konzipiert sind, unterscheiden sich die für vxWorks angelegten Bindings und Treiberbezeichnungen deutlich von denen des Linux Kernels.

5.1.1. Build-Prozess

Der Linux Build Prozess wird mit einer Host-Target Entwicklung realisiert. Auf dem Host-Rechner, ein Notebook mit x86-Architektur, wird der Kernel für das Target, das M48 mit ARM-Architektur, kompiliert. Damit die erzeugten Binärdateien auf dem Target lauffähig sind, müssen die Quelldateien unter Verwendung einer Cross-Compiling Toolchain kompiliert worden sein. Kernel und Bootloader werden dabei mit einer GCC-Toolchain (GNU Compiler Collection) kompiliert.

Abbildung 5.1 zeigt eine Übersicht der Schritte beim Erstellen eines Linux Systems. Ergänzend dazu veranschaulicht Abbildung 5.2 die Partitionierung der microSD-Karte, die als Bootmedium des i.MX6 SoC genutzt wird.

Als Dateisystem für die Hauptpartition, die das rootfs enthält, wurde BTRFS, sprich „Butter FS“ gewählt. Das Dateisystem befindet sich noch in der Entwicklung, gilt aber als stabil. BTRFS eignet sich vor allem für Embedded Systeme, die auf Flash-Speichern installiert sind. Durch ein „Copy-On-Write“ Verfahren werden Kopien von Dateien erst dann angelegt, wenn die Kopie dem Original gegenüber auch tatsächlich verändert wird. BTRFS führt dadurch möglichst wenige Schreibvorgänge aus und besitzt damit das Potential, die Lebensdauer des Flash-Speichers zu verlängern.

U-Boot Build

Der Bootloader kann über Änderungen am Quellcode angepasst werden. Die Quellcode- und Header-Dateien, die U-Boot speziell für den i.MX6 auf dem M48 beschreiben, wurden in einer Config-Datei definiert. Durch Aufruf der Option *make mx6dlm48_config* wird die Konfiguration für den nächsten Build ausgewählt. Da auf dem M48 eine ältere U-Boot-Version eingesetzt wird, muss eine kompatible Version der GCC-Toolchain parallel zur aktuellen GCC-Version installiert werden. Hierfür wird auf die GCC-Toolchains von Linaro zurückgegriffen [14]. Das Kompilieren des Quellcodes erzeugt verschiedene Images.

Dem Image des Bootloaders muss je nach Auswahl des Image-Typs noch der Secondary Program Loader (SPL) vorangestellt werden. Der SPL und die Datei „u-boot.img“ werden direkt hintereinander im unpartitionierten Bereich am Anfang der microSD-Karte abgelegt. Alternativ kann auch das „u-boot.imx“ Image, das bereits beide Komponenten enthält, dort abgelegt werden.

Zum Kopieren der Images in den nicht partitionierten Bereich der Speicherkarte wird das Programm „dd“ genutzt, das Bestandteil jeder Linux-Installation ist. Mit dem Programm dd können Binärdaten Bit-genau, ohne Berücksichtigung eines Dateisystems, zwischen Datenträgern kopiert werden. Neben dem U-Boot-Image als Quelle und dem Datenträger als Ziel werden zum Kopieren mit dd auch die Blockgröße, Startadresse auf dem Zieldatenträger und gegebenenfalls die Anzahl der maximal zu übertragenden Blöcke angegeben.

Device Tree Build

Auch der Device-Tree wird über direkte Änderungen an den Quelldateien angepasst. Der Device Tree wird dann mit dem Device Tree Compiler (dtc) kompiliert, der Bestandteil der Linux-Quelldateien ist, aber auch separat über einen Paketmanager unter Ubuntu installiert werden kann. Der so erhaltene Device Tree Blob (auch Device Tree Binary) kann direkt auf der Boot-Partition der microSD Karte abgelegt werden.

Linux Kernel Build

Der Kernel wird zunächst menübasiert konfiguriert und dann mit einer GCC-Toolchain kompiliert. Dafür kann z. B. die unter Ubuntu vorinstallierte GCC-Version genutzt werden. Das zImage ist eine komprimierte Version des Linux Kernels und die am häufigsten verwendete Form des Kernel Images. Wenn der Bootloader die Kontrolle an den Kernel übergibt, dekomprimiert sich dieser zunächst. Erst beim Dekomprimieren kopiert sich der Kernel an die Startadresse im Hauptspeicher, die den Beginn des virtuellen Speichers darstellt. Diese Startadresse, auch Loadadresse genannt, muss dem Kernel vom Bootloader bekannt gegeben werden. Ist die Startadresse im Bootloader hinterlegt, kann dieser das zImage direkt laden.

Meistens wird anstelle des zImages aber ein uzImage verwendet. Das uzImage ist ein zImage, dem durch das zu U-Boot gehörende Programm mkImage ein U-Boot Header vorangestellt wurde. Dieser Header enthält neben der Startadresse auch weitere Informationen, wie z.B. eine Bezeichnung des Kernels oder dessen Version. Listing 5.1 zeigt den Header des Kernel Images für den i.MX6. Beim Erstellen der Images ist zu beachten, dass die Startadresse des i.MX6 (0x10100000) verschieden von der Startadresse des Cyclone V

(0x00100000) ist. Der Offset des Adressraumes beim i.MX6 kann Kapitel 2 „Memory Maps“ der Reference Manual [12, S.213] entnommen werden.

```
mathis@t440s:~/linux/arch/arm/boot$ mkimage -l uzImage
Image Name:   Linux kernel i.MX6QDL
Created:      Thu Sep 28 18:56:24 2017
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4098816 Bytes = 4002.75 kB = 3.91 MB
Load Address: 10100000
Entry Point:  10100000
```

Listing 5.1: U-Boot Header des i.MX6 Linux Kernel Image

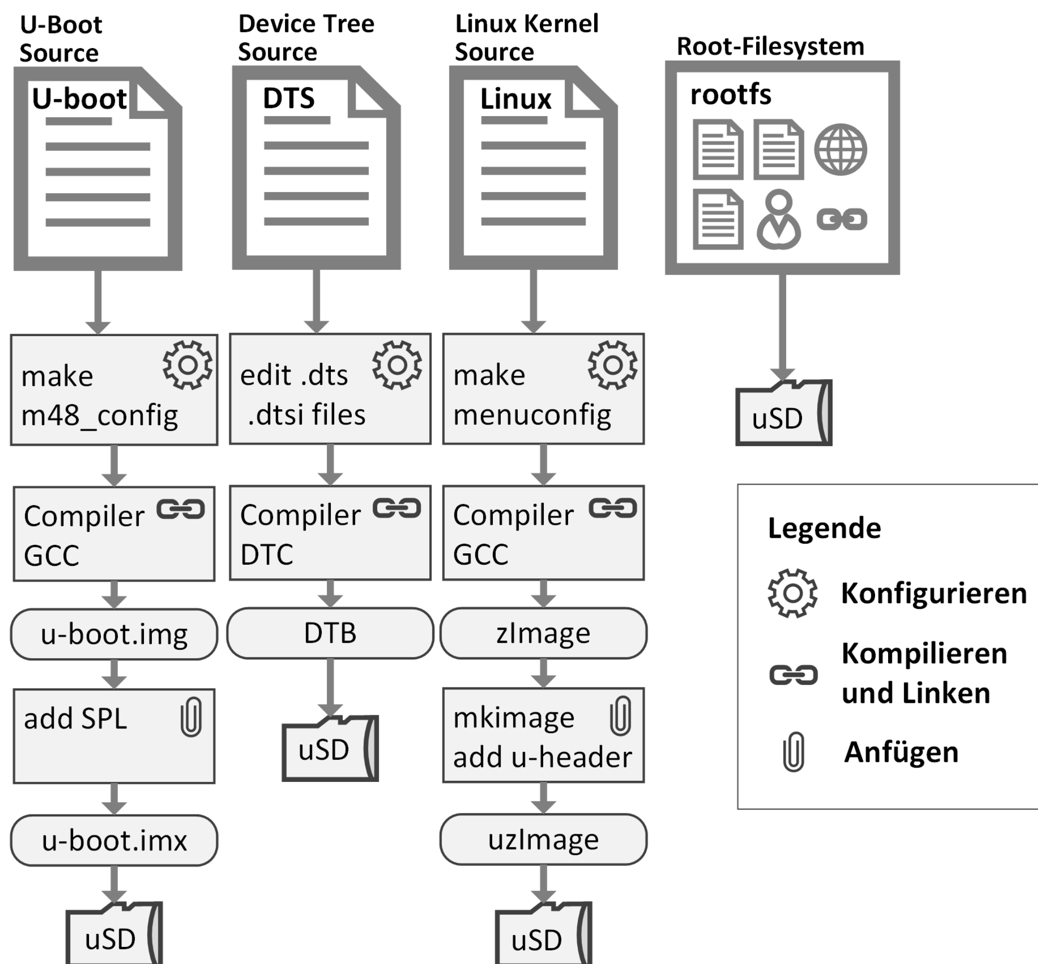

Abbildung 5.1.: Übersicht: Linux Build-Prozess für den i.MX6, Quelle: Eigene Darstellung

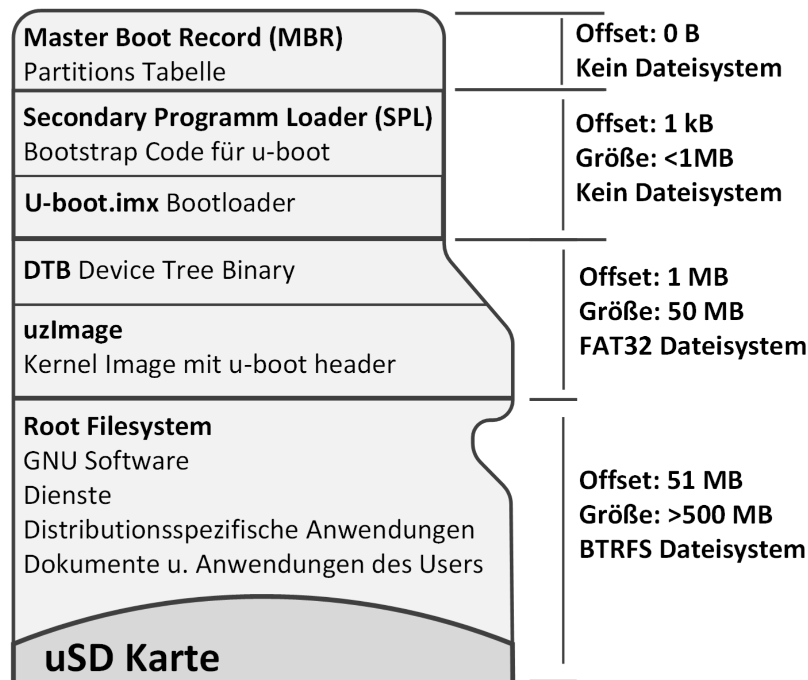

Abbildung 5.2.: SD Karte als Bootmedium, Quelle: Eigene Darstellung

5.1.2. Anpassen des Bootloaders

Der Bootloader U-Boot kann in großen Teilen so übernommen werden, wie er für den Betrieb des M48 unter vxWorks genutzt wird. Der Device Tree Blob und das Image des Betriebssystems werden dabei auf einer FAT32 Partition abgelegt. Für die Verwendung mit Linux sollte es allerdings auch möglich sein, Dateien von dem Linux-spezifischen EXT4-Dateisystem zu laden. Damit könnten das Kernel Image und der Device Tree Blob in einem Verzeichnis des Root Filesystems abgelegt werden. Um das Laden von EXT4Partitionen zu unterstützen, wird ein Makro im U-Boot-Quellcode aktiviert und U-Boot daraufhin neu kompiliert.

Die Schritte, die U-Boot beim Booten ausführt, und die Dateien, die dabei geladen werden, sind in Umgebungsvariablen hinterlegt. Die Umgebungsvariablen können bereits im U-Boot-Quellcode definiert werden, über Skripte während des Bootvorgangs geladen werden oder in einem interaktiven Modus vom Nutzer eingegeben werden. Wird der Bootvorgang kurz nach dem Reset mit der Tastenkombination *STRG + C* abgebrochen, gelangt der Nutzer in eine U-Boot Konsole. Hier können Umgebungsvariablen neu gesetzt, und Run-Befehle definiert werden. Auch die MAC-Adresse des Boards wird in der U-Boot-Konsole definiert und mit dem Befehl *setMacAddr* im EEPROM abgelegt.

Ein Run-Befehl kann aus einer Reihe von U-Boot-Kommandos bestehen, die in einem Funk-

tionsnamen abgelegt werden. So kann eine Sequenz von U-Boot-Kommandos erstellt werden, die Kernel Image und Device Tree Blob von der Speicherkarte in den Hauptspeicher laden und anschließend den Kernel starten. Diese Sequenz wird der Umgebungsvariable *autoboot* zugewiesen, so dass sie nach einem Reset automatisch ausgeführt wird.

U-Boot hat auch die wichtige Aufgabe, dem Kernel beim Booten grundlegende Variablen, die sogenannten *Bootargs* mitzugeben. Neben der MAC Adresse des Boards enthalten die Bootargs auch die Baudrate und den Gerätenamen der seriellen Schnittstelle, über die der Anwender das Linux System bedient. Werden diese Variablen falsch gesetzt, bootet der Kernel womöglich im Hintergrund, ohne dass der Anwender Meldungen des Kernels sieht.

5.1.3. Anpassen des Device Trees

Im Kernel-Source Tree liegt der Device Tree eines bestimmten Boards in Form einer Device-Tree-Source Datei (.dts) und mehreren Device-Tree-Include Dateien (.dtsi) vor. Abbildung 5.3 veranschaulicht die Zusammensetzung des Device Tree für den i.MX6 am Beispiel des SPI Bus. Die Properties in den einzelnen Nodes sind hierbei stark gekürzt. Der komplette Device Tree für den i.MX6 auf dem M48 beispielsweise, hat einen Umfang von ca. 1700 Zeilen.

Die Include-Dateien sind meist über mehrere Verzeichnisebenen verteilt und erlauben eine Beschreibung der Hardware durch gestaffelte Abhängigkeiten. So gibt es meist eine Ebene für die Prozessorfamilie, eine Ebene für die spezielle Ausführung des Prozessors, sowie eine darüber liegende Ebene für die boardspezifische Konfiguration. Jüngere Device-Tree-Einträge können dabei ältere Einträge überschreiben. Hierbei ist die Reihenfolge der Einträge ausschlaggebend, die sich durch die Reihenfolge der Include-Anweisungen ergibt. In den SoC-spezifischen Include-Dateien sind Komponenten oft standardmäßig deaktiviert und werden erst in der Board-spezifischen dts-Datei durch Überschreiben der status-property aktiviert.

Auch externe Komponenten können im Device Tree hinterlegt werden. Im Device Tree des i.MX6 sind der am I2C Bus angeschlossene PMIC (Power Management IC) und die Realtime-Clock (RTC) definiert. Auch die auf dem Testboard vorhandenen EEPROMs an I2C und SPI Bus sind in den entsprechenden Nodes des jeweiligen Busses angelegt.

Die wichtigste Referenz bei der Arbeit mit Device Trees ist die Dokumentation der Device Tree Bindings [23] im Verzeichnis des Mainline Kernels. Für jede Klasse von Komponenten und oft auch für bestimmte Chips eines Herstellers gibt es eine Textdatei, die definiert, welche Properties das Binding unterstützt und ob diese verpflichtend oder optional sind. Oft muss auch auf die Treiber [25] im Linux Kernel zurückgegriffen werden. Dort ist der „compatible-string“ definiert, der im Device Tree Node hinterlegt werden muss, damit der Kernel das Gerät mit dem passenden Treiber anspricht.

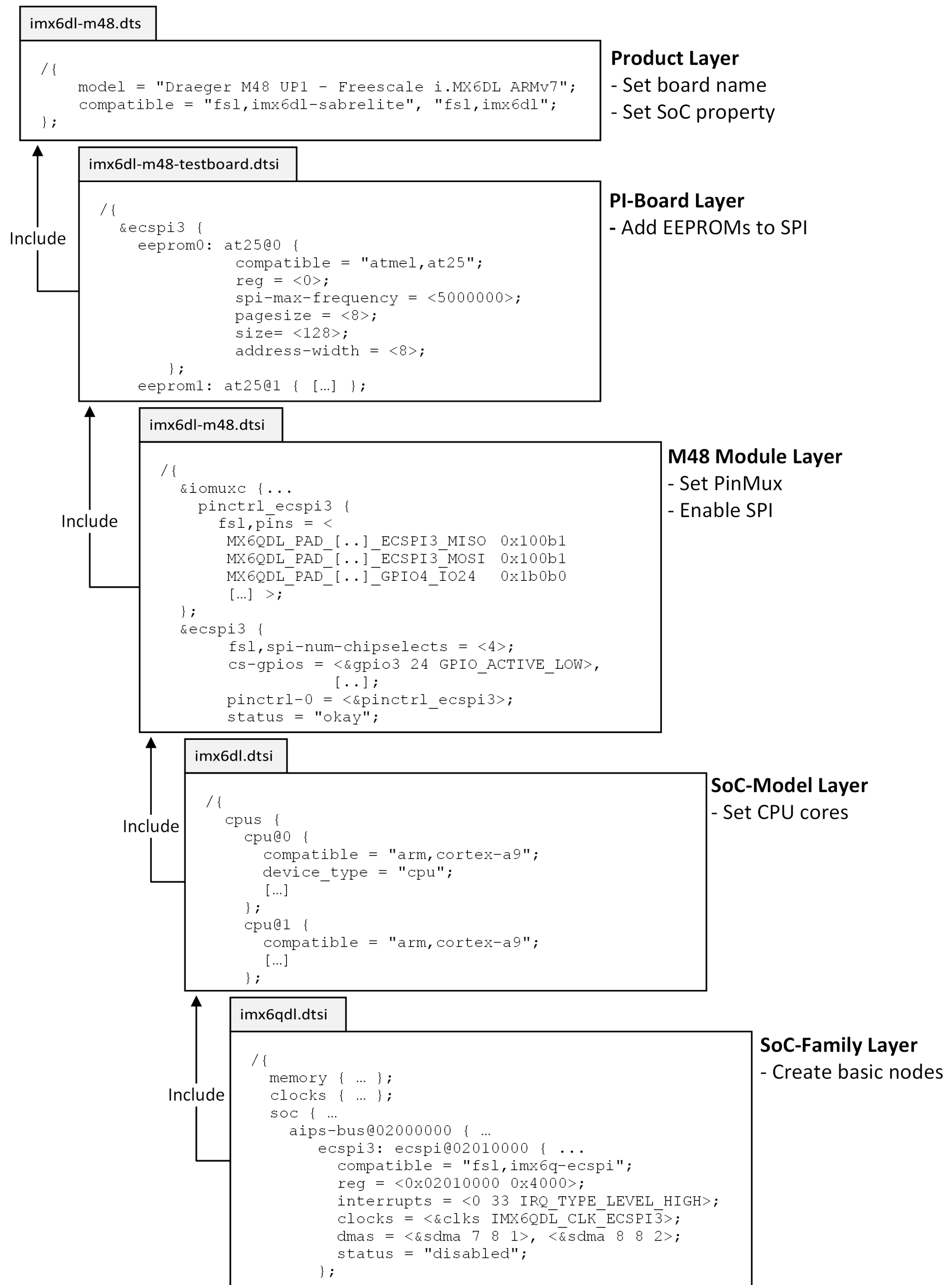

Abbildung 5.3.: Zusammensetzung des Device Trees am Beispiel des i.MX6 SPI Bus, Quelle: Eigene Darstellung

Input/Output Multiplex Controller

Die Anzahl der Pins eines SoC ist begrenzt durch die Größe des Gehäuses (Package) sowie die Strukturbreite (Pitch) der Anschluss-Pins, bzw. Pads. Oft werden nicht alle Schnittstellen eines SoC genutzt, einzelne Pads sollen als GPIOs genutzt werden und bestimmte Schnittstellen können aufgrund des PCB-Layouts nur in einem bestimmten Bereich aus dem Chip hinausgeführt werden. Daher wird jedem Pad ein Input/Output Multiplexer (IOMUX) vorgeschaltet. Mit dem IOMUX Controller (IOMUXC) lassen sich neben der Funktion auch die elektrischen Eigenschaften des Pads konfigurieren. So können z.B. die slew-rate (Flankensteilheit), die Ein- oder Ausgangsimpedanz sowie der maximale Strom und die Art des Treibers - Pull-up, Pull-down oder open drain - eingestellt werden.

Der Pin-Controller-Node im Device Tree wird dem i.MX-IOMUX Treiber durch die SoC-spezifische compatible-property *fsl,imx6dl-iomuxc* zugeordnet. Für jedes Gerät bzw. jede Funktionsgruppe wird im Pin-Controller-Node ein Pin-Configuration-Node angelegt. In einem solchen Node werden anhand von Makros die Pins der Funktionsgruppe konfiguriert. Jedem Pin lässt sich dabei eine von bis zu acht verschiedenen Funktionen zuordnen. Die Eigenschaften eines Pins werden durch ein Tupel von sechs Integer-Werten bestimmt. Fünf der Werte sind in einem Makro hinterlegt, das die durch Multiplexing ausgewählte Funktion bestimmt. Diese Werte werden im „Pad Mux Register“ des entsprechenden Pins abgelegt. Das sechste Element konfiguriert die elektrischen Eigenschaften des Ausgangstreibers. Dieser Wert wird in das „Pad Control Register“ geschrieben. Die Funktion der einzelnen Bits der Register kann Kapitel 37 „IOMUX Controller“ der i.MX6 Reference Manual entnommen werden.[12, S.2001-S.2813]

Konfiguration der SPI Schnittstelle

Exemplarisch für den an der SPI Schnittstelle angeschlossenen EEPROM AT25010B [3], soll das Einpflegen einer neuen Komponente in den Device Tree beschrieben werden.

SPI ist ein synchroner, serieller Datenbus, bei der mehrere Bus-Teilnehmer nach dem Master-Slave miteinander kommunizieren. Die Datenleitungen vom Master zum Slave, respektive vom Slave zum Master werden MOSI und MISO genannt. Die Datenübertragung wird über eine separate Clock-Leitung synchronisiert. Alle Slaves sind an der gleichen Datenleitung angeschlossen, werden aber nur aktiv, wenn die Ihnen zugeordnete Chip-Select-Leitung vom Master gegen Masse gezogen wird.

Listing 5.2 zeigt die Konfiguration des IOMUX Controllers. Die Pads für die Datenübertragung und die Clock werden dort dem SPI-Block des i.MX6 zugeordnet. Die Pads der Chipselect-Leitungen müssen explizit als GPIOs definiert werden, da die native Chip-Select-Funktion vom Linux SPI-Treiber nicht unterstützt wird.

```
pinctrl_ecspi3: ecspi3grp {
fsl,pins = <
MX6QDL_PAD_DISP0_DAT2__ECSPI3_MISO  0x100b1 /*100k Pull Down*/
MX6QDL_PAD_DISP0_DAT1__ECSPI3_MOSI  0x100b1
MX6QDL_PAD_DISP0_DAT0__ECSPI3_SCLK  0x100b1
MX6QDL_PAD_DISP0_DAT3__GPIO4_IO24 0x1b0b0 /*SS0 -> 100k Pull Up*/
/*MX6QDL_PAD_DISP0_DAT3__ECSPI3_SS0 0x1b0b0   SS0-3 does not work!*/
MX6QDL_PAD_DISP0_DAT4__GPIO4_IO25 0x1b0b0 /*SS1*/
MX6QDL_PAD_DISP0_DAT5__GPIO4_IO26 0x1b0b0 /*SS2*/
MX6QDL_PAD_DISP0_DAT6__GPIO4_IO27   0x1b0b0 /*SS3*/
>;
};
```

Listing 5.2: IOMUX Einstellungen für die SPI Schnittstelle, imx6qdl-m48.dtsi

Im SPI Device Tree Node (Listing 5.3) wird die Anzahl der Chip-Select-Leitungen spezifiziert. Als GPIOs werden die vorher im IOMUX Controller definierten GPIOs ausgewählt. Der Treiber wird außerdem darüber informiert, dass die SPI-Slaves bei einem Low-Zustand der Chip-Select-Leitung aktiv sind. Die IOMUX Konfiguration für den SPI wird erst dadurch aktiv, dass sie vom SPI Node referenziert wird. Erst durch Setzen der *status-property* zu „okay“ wird der SPI Bus aktiviert.

Der EEPROM wird als ein „Sub-Node des SPI Node angelegt. Der *compatible* String muss dem *.compatible Member* der Device-ID Struktur in einem Linux Treiber (Listing 5.4) entsprechen, damit der entsprechende Treiber beim Booten geladen, und der Komponente zugeordnet wird. Parameter wie Adress-Breite und Größe des EEPROMs müssen dem Datenblatt [3] entnommen, und in den EEPROM-Node eingepflegt werden.

Wurde der EEPROM erfolgreich in den Device Tree eingepflegt und der entsprechende Treiber in der Kernel Konfiguration ausgewählt, steht der EEPROM als virtuelle Datei „eeprom“ im sysfs zur Verfügung. So kann der EEPROM mit einem Texteditor beschrieben und ausgelesen werden.

```
&ecspi3 {
fsl,spi-num-chipselects = <4>;
cs-gpios =  <&gpio4 24 GPIO_ACTIVE_LOW>,
      <&gpio4 25 GPIO_ACTIVE_LOW>,
      <&gpio4 26 GPIO_ACTIVE_LOW>,
      <&gpio4 27 GPIO_ACTIVE_LOW>;
pinctrl-names = "default";
pinctrl-0 = <&pinctrl_ecspi3>;
status = "okay";
  eeprom0: at25@0 {
  compatible = "atmel,at25";
  reg = <0>;
  spi-max-frequency = <5000000>;  /* 5MHz SCK*/
  pagesize = <8>;
  size= <128>;
  address-width = <8>;
  };
  eeprom1: at25@1 {
  [...]
}
```

Listing 5.3: Peripherie, eingebunden in die SPI Schnittstelle

```
387 static const struct of_device_id at25_of_match[] = {
388 { .compatible = "atmel,at25", },
389 { }
390 };
391 MODULE_DEVICE_TABLE(of, at25_of_match);
392
393 static struct spi_driver at25_driver = {
394 .driver = {
395 .name   = "at25",
396 .of_match_table = at25_of_match,
397 },
398 .probe    = at25_probe,
399 .remove   = at25_remove,
400 };
401
```

Listing 5.4: SPI EEPROM Treiber, at25.c, Zeilen 387-400

5.1.4. Konfiguration des Kernels

Der Kernel muss vor dem Kompilieren an die eigene Hardware angepasst werden. Dafür müssen die Prozessorfamilie sowie Treiber für die einzelnen Schnittstellen ausgewählt werden. Um den Kernel dabei möglichst klein zu halten, sollten nur die Funktionen aktiviert werden, die tatsächlich für das eigene Board benötigt werden. Ein kleiner Kernel belegt weniger Speicherplatz und bootet auch schneller.

Eine Kernel-Konfiguration ist in einer Datei mit der Erweiterung .config abgelegt. Bei der Arbeit mit dem Linux Source Code ist zu beachten, dass Linux versteckte Dateien über einen dem Dateinamen vorangestellten Punkt identifiziert. Diese Dateien, wie z.B. die gerade erwähnte .config-Datei werden für den Nutzer erst sichtbar, wenn er die Anzeige versteckter Dateien im Dateiexplorer mit der Tastenkombination *Strg + H* aktiviert, bzw. in der Shell das tool *ls* mit der Option *-a* (list all) aufruft.

Die .config-Datei sollte nicht manuell verändert werden, da viele Einträge gegenseitige Abhängigkeiten besitzen. Zur Konfiguration des Kernels wird das *Kernel Configuration* tool (Abbildung 5.4) genutzt, das über den Befehl *make menuconfig* aufgerufen wird. Beim Aufruf muss die Variable *ARCH=arm* angegeben werden, um die ARM-spezifische Kernel Konfiguration zu öffnen.

Ein Treiber, der mit dem Kernel kompiliert wird, ist durch das <*> vor dem jeweiligen Eintrag gekennzeichnet. Mit <M> gekennzeichnete Treiber werden auch mit dem Kernel kompiliert, allerdings als separate Module angelegt. Diese Kernelmodule müssen im Root-File System abgelegt werden, um zur Laufzeit nachgeladen zu werden. Je mehr Treiber als Module angelegt sind, desto flexibler ist der Kernel einsetzbar, und desto schneller bootet das System.

Die speziell für den i.MX6 auszuwählenden Treiber umfassen u.a.

- i.MX Pincontrol System
- USB-Host und -OTG Funktionalität
- PF100 PMIC
- i.MX on-chip Spannungsregler
- Freescale Ethernet Controller
- Micrel Ethernet Phy
- Tempon Temperatursensor für den Prozessorkern

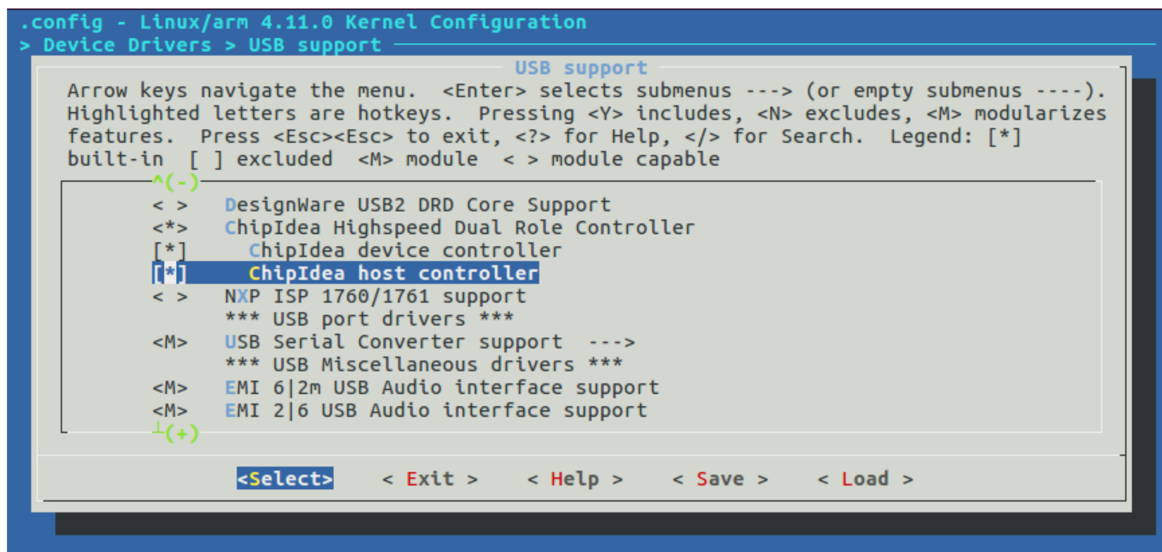

Abbildung 5.4.: Untermenü für USB Treiber im Kernel Configuration tool

5.2. Root Filesystem und Desktopumgebung

Das verwendete Debian Root-Filesystem wurde von dem Kollegen Nathan-Jedidja Hirschauer übernommen, der damit erste Versuche einer Linux-Installation auf dem M48 unternommen hat. Das Root-Filesystem wurde mit dem von Debian zur Verfügung gestellten Tool Debootstrap [22] erstellt. Debootstrap installiert Debian in ein Verzeichnis, das dann einfach auf die Hauptpartition der Speicherkarte kopiert wird.

Der Window Server XORG, der für die Darstellung von Fenstern notwendig ist, ist bereits Teil der Debian Installation. Mit dem Package Management System APT (Advanced Packaging Tool) kann die effiziente, aber minimalistische Desktopumgebung LXDE im laufenden System nachinstalliert werden. Abbildung 5.5 zeigt das M48 und die LXDE Desktopumgebung.

5.3. Funktionstest

Im Folgenden wird der Funktionstest realisiert. Zunächst werden die Testroutinen für den I^2C Bus und den SPI Bus, die vier UART Schnittstellen, die GPIOs und die Safety-Logik implementiert. Dann wird in einem Shellskript eine Testsequenz geschrieben, die die einzelnen Testroutinen einbindet.

Abbildung 5.5.: Debian mit installierter LXDE Desktopumgebung auf dem M48

5.3.1. Gemeinsame Log-Funktionen

Für die in C und Python geschriebenen Testroutinen wurde jeweils eine gemeinsame Log-Funktion implementiert. Diese Log-Funktion wird von den Testroutinen eingebunden und erlaubt es, das Format der Log-Meldungen an zentraler Stelle anzupassen. Die Testroutinen wurden so implementiert, dass diese keine Ausgaben in die Standardausgabe schreiben, sondern alle auftretenden Fehlermeldungen oder Testergebnisse in die Log-Funktion umleiten. Die Log-Funktion schreibt die Meldung dann in die Log-Datei und gibt diese gegebenenfalls auch gleichzeitig auf der Standardausgabe aus.

Listing 5.5 zeigt die Log-Funktion für die in C geschriebenen Testroutinen, und Listing 5.6 zeigt die entsprechende Funktion in Python. Die Log-Datei wird jeweils nur für den Schreibzugriff geöffnet und nach erfolgreichem Schreibvorgang wieder geschlossen, so dass die Datei anderen Testroutinen zur Verfügung steht. Tritt beim Zugriff auf die Log-Datei ein Fehler auf, wird dieser direkt auf die Standardausgabe ausgegeben.

```
9  int put_log(char* logstr){
10   char logbuf[LOG_BUFLEN];
11   FILE *fp=NULL;
12   // open logfile
13   if ((fp=fopen(LOGFILE_PATH, "a")) == NULL){
```

```
14      snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't open Logfile: %s\n",
       strerror(errno));
15      printf(logbuf);
16      return EXIT_FAILURE;
17    }
18    // write line to logfile
19    if (fputs(logstr, fp) < 0){
20      snprintf(logbuf, LOG_BUFLEN, "FAILED: Error in fputs: %s\n",strerror(
       errno));
21      printf(logbuf);
22      return EXIT_FAILURE;
23    }
24    fclose(fp);
25    if(PRINT_LOG){
26      printf(logstr);
27    }
28    return EXIT_SUCCESS;
29  }
```

Listing 5.5: Gemeinsame Log-Funktion, m48test.c, Zeilen 9-29

```
8   import sys
9   PRINT_LOG = True
10  PROJECT_PATH = '/root/m48test/'
11  LOG_FILE_PATH = PROJECT_PATH+'log/logfile.txt'
12
13  def put_log(log_string):
14      if(PRINT_LOG):
15          sys.stdout.write(log_string)
16      try:
17          # open logfile in 'append' mode
18          logfile = open(LOG_FILE_PATH, 'a')
19          logfile.write(log_string)
20          logfile.close
21      except IOError:
22          sys.stdout.write("IOError: python put_log() cannot open logfile\n")
23          return 1
24      return 0
```

Listing 5.6: Log Funktion für Python Skripte, m48test.py, Zeilen 8-24

5.3.2. i.MX6 I2C On-Module EEPROM

An der I^2C 1 Schnittstelle des i.MX6 ist ein EEPROM angebunden, der die MAC Adresse des i.MX6 und andere Informationen zur eindeutigen Identifikation des Boards speichert. Mit der Testroutine für den EEPROM soll überprüft werden, ob auf diesen zugegriffen werden kann, und ob die Teilnummer (Partnumber) des M48 hinterlegt ist. In der eingebundenen Header-Datei *imx_i2c_eeprom.h* werden die Adressen der im EEPROM abgelegten Strings definiert,

so dass bei weiterführenden Anwendungen auch andere Informationen ausgelesen werden können.

Abbildung 5.6 zeigt ein Flussdiagramm der Testroutine 5.7. Die rot hinterlegten Felder im Flussdiagramm stellen alle im Fehlerfall möglichen Rückgabewerte dar. Da der EEPROM nach Einbindung des Treibers im virtuellen Dateisystem hinterlegt ist, kann dieser wie eine Datei angesprochen werden. Der Lesezeiger springt innerhalb der Datei an die Position, an der die Partnumber hinterlegt sein sollte, und liest die dort stehende Information aus. Stimmt der gelesene String nicht mit der erwarteten Partnumber überein, liefert der Test den Rückgabewert *EXIT_FAILURE*, der als Makro für den Wert '1' definiert ist. Wurde die richtige Partnumber gelesen, wird *EXIT_SUCCESS* zurückgegeben, was dem Wert '0' entspricht.

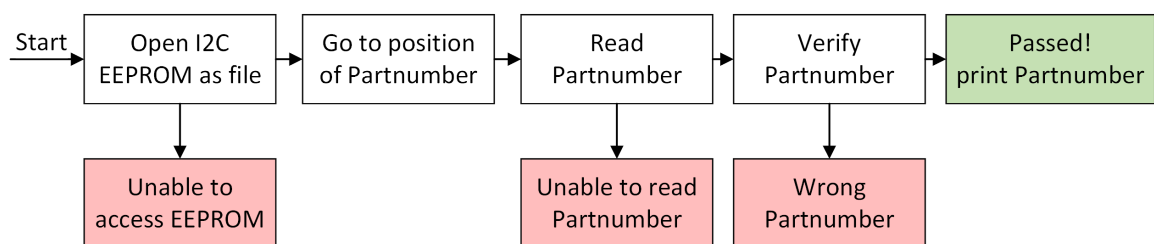

Abbildung 5.6.: On-ModuleEEPROM Test, Flussdiagramm

```c
#include "imx_i2c_eeprom.h"
#include "m48test.h"

#define FILE_PATH "/sys/bus/i2c/devices/1-0050/eeprom"
#define READ_BUFLEN 1024


int main(void){
  char readbuf[READ_BUFLEN];
  char logbuf[LOG_BUFLEN];
  FILE *fp=NULL;
  // open EEPROM Device through driver
  if ((fp=fopen(FILE_PATH, "r+")) == NULL){
    snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't open EEPROM: %s\n",
    strerror(errno));
    put_log(logbuf);
    return EXIT_FAILURE;
  }
  // go to address that holds partnumber
  if (fseek(fp, PARTNO_ADDR, SEEK_SET) > 0){
    snprintf(logbuf, LOG_BUFLEN,"FAILED: Error in fseek %s\n",strerror(errno
    ));
    put_log(logbuf);
    return EXIT_FAILURE;
  }
  // read partnumber
  if (fgets(readbuf, (PARTNO_LEN+1), fp) == NULL){
    snprintf(logbuf, LOG_BUFLEN,"FAILED: Error in fgets %s\n",strerror(errno
    ));
    put_log(logbuf);
    return EXIT_FAILURE;
  }
  fclose(fp);
  // compare obtained partnumber with expected partnumber
  if (strcmp(PARTNO,readbuf)==0){
    snprintf(logbuf, LOG_BUFLEN,"OK:     imx i2c eeprom partnumber: %s\n",
    readbuf);
    put_log(logbuf);
    return EXIT_SUCCESS;
  }
  else{
    snprintf(logbuf, LOG_BUFLEN,"FAILED: imx i2c eeprom partnumber: %s\n",
    readbuf);
    put_log(logbuf);
    return EXIT_FAILURE;
  }
}
```

Listing 5.7: i.MX6 On-Module EEPROM test, imx_i2c_eeprom_test.c, Zeilen 6-44

5.3.3. i.MX6 I2C Testboard EEPROMs

Der i.MX6 verfügt über zwei weitere I^2C Schnittstellen, die auf das Testboard herausgeführt sind. An jeden der Busse ist ein EEPROM des Typs angebunden, der auch auf dem M48 verwendet wird. Die beiden EEPROMs sind nicht mit Daten vorbelegt und können daher während des Tests beschrieben und vollständig gelöscht werden. Für jeden der I^2C Busse wurde diese Testroutine einmal erstellt.

Mit diesem Test sollen die EEPROMs auch auf Bitfehler getestet werden. Verbleibt ein Bit dauerhaft im Zustand '1' oder '0', soll das von diesem Test detektiert werden. Abbildung 5.7 stellt den Ablauf der Testroutine 5.8 dar.

Nach dem Öffnen wird der ganze Speicherbereich des EEPROMs mit dem Wert '0x00' beschrieben um alle Bits zu Null zu setzen. Der Dateizeiger wird daraufhin zurück auf den Anfang gesetzt und der EEPROM Byteweise ausgelesen. Hat ein Byte nicht den Zustand '0x00', liegt mindestens ein Bitfehler vor, und ein Fehler-Zähler wird inkrementiert. Zusätzlich wird der Fehler, unter Angabe der Position des Bytes, gelogged. Der Vorgang wird mit dem Wert '0xff' wiederholt, um auch fehlerhafte Bits zu identifizieren, die dauerhaft im Zustand '0' verbleiben. Wurden einer oder mehrere Bytefehler gefunden, liefert die Testroutine *EXIT_FAILURE* als Rückgabe und loggt die Gesamtzahl der aufgetretenen Fehler.

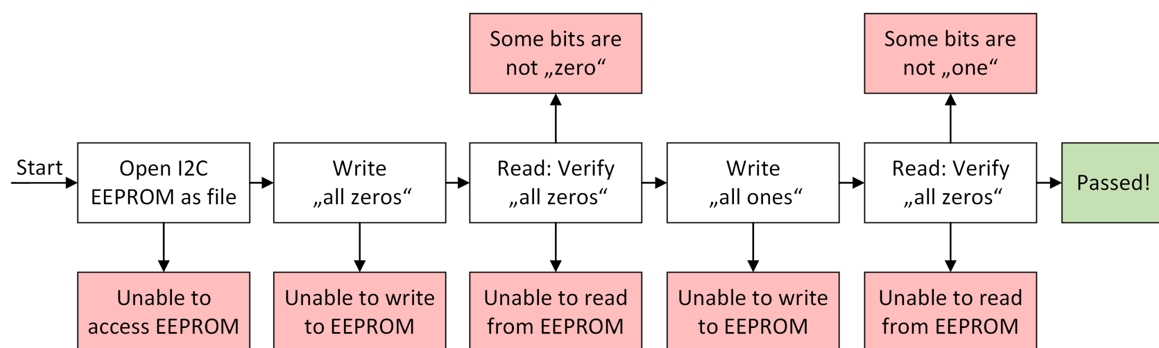

Abbildung 5.7.: I2C Testboard EEPROM Test, Flussdiagramm

```
9  #include "m48test.h"
10
11 #define EEPROM_PATH "/sys/bus/i2c/devices/2-0050/eeprom"
12 #define READ_BUFLEN 1024
13 #define EEPROM_SIZE 1024
14
15
16 int main(void){
17   char logbuf[LOG_BUFLEN];
18   FILE *fp=NULL;
19   int error_count=0;
20
```

```
21  // open EEPROM Device through driver
22  if((fp=fopen(EEPROM_PATH, "r+")) == NULL){
23    snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't open I2C2 Testboard
      EEPROM: %s\n",strerror(errno));
24    put_log(logbuf);
25    return EXIT_FAILURE;
26  }
27  // write all zeros (1024 Byte) to EEPROM
28  for(int i=0; i<EEPROM_SIZE; i++){
29    if (fputc(0x00, fp) < 0){
30      printf("ERROR in fputc %s\n",strerror(errno));
31      return EXIT_FAILURE;
32    }
33  }
34  // go back to start of Memory
35  rewind(fp);
36  // read entire EEPROM  (1024 Byte) and check for "all zeros"
37  for(int i=0; i<EEPROM_SIZE; i++){
38    if( fgetc(fp)!= 0x00 ){
39      snprintf(logbuf, LOG_BUFLEN, "FAILED: Bit error on I2C2 Testboard
      EEPROM. Expected 0x00 at Byte %d\n",i);
40      put_log(logbuf);
41      error_count++;
42    }
43  }
44  rewind(fp);
45  // write all ones (1024 Byte)to EEPROM
46  // This is equivalent to an erased EEPROM.
47  for(int i=0; i<EEPROM_SIZE; i++){
48    if (fputc(0xff, fp) < 0){
49      printf("ERROR in fputc %s\n",strerror(errno));
50      return EXIT_FAILURE;
51    }
52  }
53  rewind(fp);
54  // read entire EEPROM  (1024 Byte) and check for "all ones"
55  for(int i=0; i<EEPROM_SIZE; i++){
56    if( fgetc(fp)!= 0xff ){
57      snprintf(logbuf, LOG_BUFLEN, "FAILED: Bit error on I2C2 Testboard
      EEPROM. Expected 0xff at Byte %d\n",i);
58      put_log(logbuf);
59      error_count++;
60    }
61  }
62  fclose(fp);
63  if(error_count == 0){
64  // Test succeded if no error has been detected
65  snprintf(logbuf, LOG_BUFLEN,"PASSED: I2C2 Testboard EEPROM is working\n");
66  put_log(logbuf);
67  }
68  else{
69  snprintf(logbuf, LOG_BUFLEN,"FAILED: %d Bit Errors on I2C2 Testboard
      EEPROM\n", error_count);
70  put_log(logbuf);
71  }
```

```
72    return EXIT_SUCCESS;
73 }
```

Listing 5.8: i.MX6 I2C Testboard EEPROM test, imx_i2c2_testboard_eeprom_test.c, Zeilen 12-28

5.3.4. i.MX6 SPI Testboard EEPROMs

Vom i.MX6 wird ein SPI Bus genutzt und auf das Testboard herausgeführt. An jede der vier Chip-Select Leitungen ist ein EEPROM angeschlossen. Die SPI EEPROMS auf dem Testboard sind mit 128 MB kleiner als die 1024 Byte großen I2C EEPROMS. Da die SPI EEPROMS auch über Treiber im sysfs hinterlegt sind, können sie, abgesehen von einer Änderung der Speichergröße, mit der gleichen Routine getestet werden, wie die I2C EEPROMs. Für die EEPROMs an den vier Chip-Select-Leitungen wurde eine gemeinsame Testroutine geschrieben, die ausschnittsweise in Listing 5.9 dargestellt ist. Abbildung 5.8 zeigt ein Flussdiagramm des Testablaufs.

Anhand des Arguments, das der Funktion beim Aufruf übergeben wird, wird entschieden, welcher EEPROM getestet wird. Die Chip-Select Nummer des getesteten EEPROMs wird in die erzeugten Log-Meldungen eingebunden, um einen fehlerhaften EEPROM eindeutig identifizieren zu können.

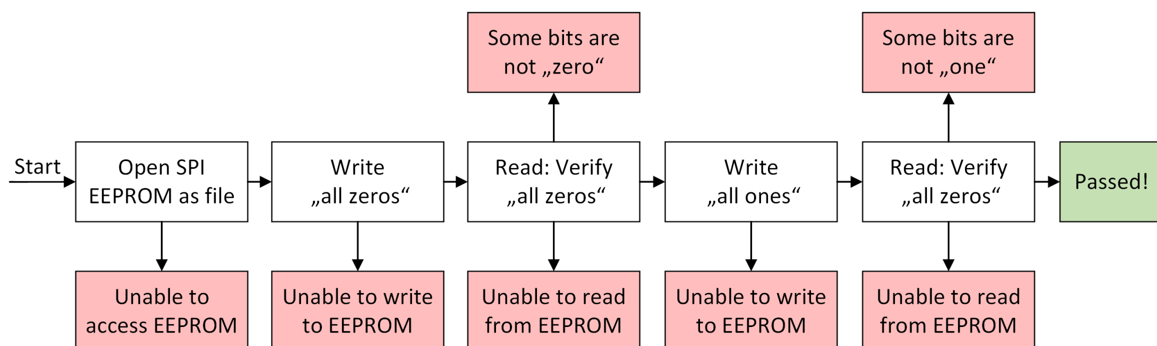

Abbildung 5.8.: i.MX6 SPI Testboard EEPROM Test, Flussdiagramm

```
12 #define EEPROM_SIZE 128 //Byte
13 #define PATH_BUFLEN 36
14 #define TEST_THE_TEST TRUE
15
16 int main(int argc, char *argv[]){
17
18   char logbuf[LOG_BUFLEN];
19   char eeprom_path[PATH_BUFLEN];
20   FILE *fp=NULL;
21   int error_count=0;
22   signal(SIGUSR1, sigh); // register handler for signal SIGUSR1
23   if(argc == 2){
24     if(strcmp(argv[1],"spi2_0")==0){
25       strncpy(eeprom_path,"/sys/bus/spi/devices/spi2.0/eeprom", PATH_BUFLEN)
     ;
26     }
27     else if(strcmp(argv[1],"spi2_1")==0){
28       strncpy(eeprom_path,"/sys/bus/spi/devices/spi2.1/eeprom", PATH_BUFLEN)
     ;
```

Listing 5.9: i.MX6 SPI Testboard EEPROM test, imx_spi_eeprom_test.c, Zeilen 9-100

5.3.5. i.MX6 UARTs

Die Testroutine für die UART Schnittstellen wurde in Python implementiert. Abbildung 5.9 zeigt den Programmablauf und Listing 5.10 den Quellcode der Testroutine. Für einen erfolgreichen Test müssen die RX- und TX- Leitungen der UART Schnittstellen auf dem Testboard jeweils mit einem Jumper gebrückt werden.

Die zu testende UART Schnittstelle wird der Testroutine beim Aufruf übergeben. Zunächst wird versucht, die Schnittstelle mit der in den Anforderungen festgelegten Datenrate von 115200 Baud zu öffnen. Für den Fall, dass auf die Schnittstelle nicht zugegriffen werden kann, wurde ein Timeout von 500 Millisekunden festgelegt. Nach Ablauf der Zeit liefert die Funktion als Rückgabewert den Fehlerzustand '1'. Gelingt es, die Schnittstelle zu öffnen, wird ein Test String gesendet und gleichzeitig versucht, diesen wieder zu lesen. Kann der String nicht wieder empfangen werden, wird eine Fehlermeldung erzeugt.

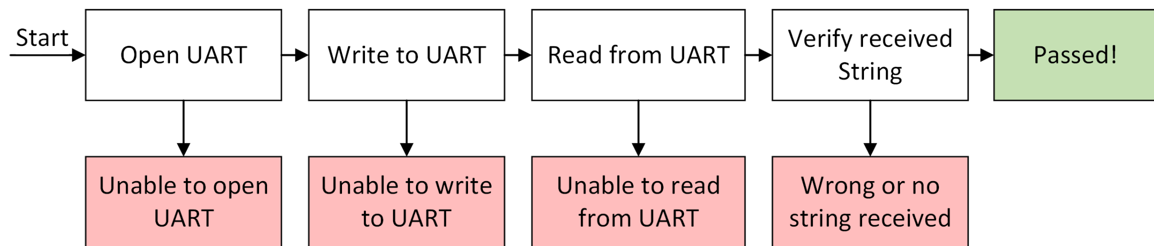

Abbildung 5.9.: i.MX6 UART Test, Flussdiagramm

```
import serial
import sys
from m48test import put_log

BAUDRATE = 115200
TEST_STRING = 'M48 UART test'

# uart number 'uartX' should be passed when calling script
try:
    if (sys.argv[1] == 'uart0'):
        UART ='UART0'        # name that goes into logfile
        port ='/dev/ttymxc0'# actual UART device
    elif (sys.argv[1] == 'uart1'):
        UART ='UART1'
        port ='/dev/ttymxc1'
    elif (sys.argv[1] == 'uart2'):
        UART ='UART2'
        port ='/dev/ttymxc2'
    elif (sys.argv[1] == 'uart3'):
        UART ='UART3'
        port ='/dev/ttymxc3'
    else:
        put_log('FAILED: imx UART test: invalid device specified\n')
        sys.exit(1)
except IndexError:
        put_log('FAILED: imx UART test: no device specified\n')
        sys.exit(1)

# Open a UART device. Linux /dev/-device and baudrate are passed.
# Other parameters can be changed in function body.
# If the device doesn't exist or cannot be accessed, an exception is caught.
def open_port(port, baudrate):
    try:
        uart = serial.Serial(port)
    except serial.SerialException:
        return None
    uart.port = port
    uart.baudrate = baudrate
    uart.parity = serial.PARITY_NONE
    uart.bytesize = serial.EIGHTBITS
    uart.rtscts = False
```

```
47      uart.dsrdtr = False
48      uart.stopbits = serial.STOPBITS_ONE
49      uart.timeout = 0.5
50      uart.write_timeout = 0.5
51      return uart
52
53  # open specified device "port"
54  uart = open_port(port, BAUDRATE)
55  if(uart is None):
56      put_log('FAILED: imx '+UART+' could not be opened\n')
57      sys.exit(1)
58  else:
59      # write test string and append linebreak + carriage return
60      uart.write(TEST_STRING + '\n\r')
61      # read test-string and remove any trailing whitespace characters
62      read_string = uart.readline().rstrip()
63      # close device before exiting
64      uart.close()
65      # compare send and received string
66      if (read_string == TEST_STRING):
67          put_log('OK:     imx '+UART+' loopback is working\n')
68          sys.exit(0)
69      else:
70          put_log('FAILED: imx '+UART+' loopback is not working\n')
71          sys.exit(1)
```

Listing 5.10: UART Test in Python, imx_uart_test.py, Zeilen 6-27

5.3.6. i.MX6 GPIOs

Eine relativ große Anzahl von GPIOs des i.MX6 ist auf das Testboard herausgeführt und mit Widerständen in Zweierpaaren verbunden. Mit einer Testroutine soll die Verbindung der GPIOs geprüft werden. Für den Test der GPIOs muss entschieden werden, ob diese über das sysfs oder durch direkten Zugriff auf die GPIO Register, mittels Memory Mapping, angesprochen werden. Ausschlaggebend ist die bei der Ansteuerung der GPIOs benötigte Geschwindigkeit. Bei direktem Zugriff über die Register können die Pins mit einer Frequenz in der Größenordnung von 1MHz umgeschaltet werden. Bei dem Zugriff über das sysfs lassen sich Frequenzen im Bereich einiger kHz realisieren.

Der direkte Zugriff auf den Speicher über /dev/mem ist in der vorliegenden Kernel Konfiguration durch die aktivierte Option *STRICT_DEVMEM* gesperrt. Der direkte Zugriff auf Speicher kann schnell zu fatalen Fehlern führen, so dass dabei sehr bedacht vorgegangen werden sollte. Der direktem Zugriff über die Register ist bei der Entwicklung von Treibern notwendig, die Gebrauch von GPIOs machen, wird aber für den Funktionstest nicht benötigt. Für die Ansteuerung der GPIOs im Funktionstest soll der Zugriff daher über das sysfs erfolgen.

Vor der Implementierung der Testroutine müssen die Pads der aus dem M48 herausgeführten GPIOs als solche konfiguriert werden. Listing 5.11 zeigt die Konfiguration des IOMUX Controllers im Device Tree. Im sogenannten „Pincontrol-Hog“ wird eine große Anzahl an GPIOs in Anspruch genommen („gehogged“), ohne einem bestimmten Treiber zugeordnet zu werden.

```
&iomuxc {
  pinctrl-names = "default";
  pinctrl-0 = <&pinctrl_hog>;

  imx6q-sabrelite {
    pinctrl_hog: hoggrp {
      fsl,pins = <
      /* SGTL5000 sys_mclk */
      MX6QDL_PAD_GPIO_0__CCM_CLKO1     0x030b0
      MX6QDL_PAD_DISP0_DAT17__GPIO5_IO11  0x1b0b0
      MX6QDL_PAD_SD4_DAT4__GPIO2_IO12   0x1b0b0
      MX6QDL_PAD_SD4_DAT5__GPIO2_IO13   0x1b0b0
      MX6QDL_PAD_SD4_DAT6__GPIO2_IO14   0x1b0b0
      MX6QDL_PAD_SD4_DAT7__GPIO2_IO15   0x1b0b0
      MX6QDL_PAD_EIM_A22__GPIO2_IO16    0x1b0b0
      [...]
```

Listing 5.11: IOMUX Einstellungen für GPIOs, imx6qdl-m48.dtsi

Abbildung 5.10 zeigt den Ablauf der GPIO-Testroutine. Da die Testroutine aus mehreren Komponenten besteht und der Quellcode der C-Datei relativ umfangreich ist, werden die Bestandteile hier jeweils einzeln erklärt. Die zusammenhängende Testroutine kann auf der DVD in Anhang A eingesehen werden.

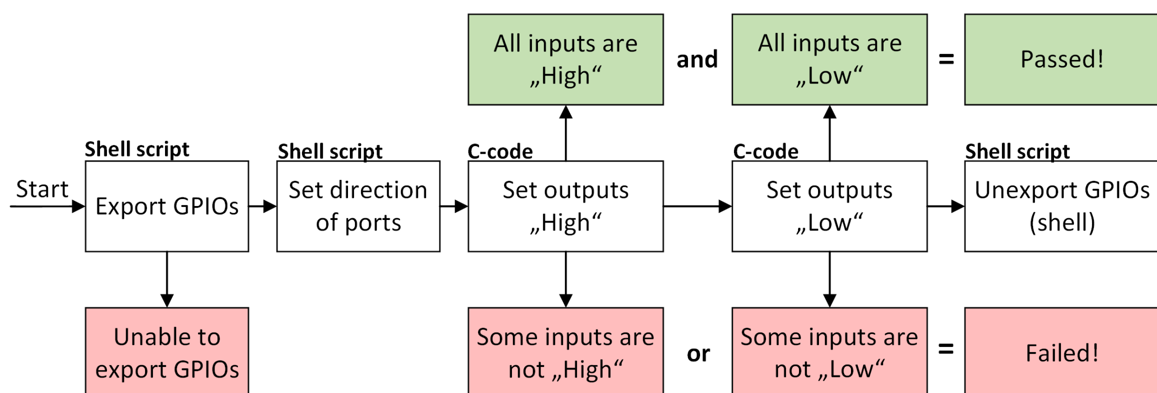

Abbildung 5.10.: i.MX6 GPIO Test, Flussdiagramm

Die GPIOs werden zunächst mit dem in Listing 5.12 dargestellten Skript exportiert, d.h. der GPIO Treiber reserviert die entsprechenden GPIOs und stellt sie über das sysfs zur Verfügung. Jeder der GPIOs wird von einem Verzeichnis repräsentiert, das unter anderem die

virtuellen Dateien *value* und *direction* enthält. Diese Dateien können beschrieben werden, um den Wert oder die Richtung eines GPIOs festzulegen. Die Hälfte der Pins wird als Ausgang, und die andere Hälfte als Eingang definiert.

```
7  # export gpios to provide access from userspace
8  # gpio number is calculated as follows:
9  # linux gpio number = (gpio_bank - 1) * 32 + gpio_bit
10 # source:
11 # https://www.kosagi.com/w/index.php?title=Definitive_GPIO_guide
12
13 # gpios are grouped according to connections on M48 Testboard
14
15 echo "139" >/sys/class/gpio/export    #GPIO5_IO11
16 echo "44" >/sys/class/gpio/export   #GPIO2_IO12
17
18 echo "45" >/sys/class/gpio/export   #GPIO2_IO13
19 echo "46" >/sys/class/gpio/export   #GPIO2_IO14
20
21 echo "47" >/sys/class/gpio/export   #GPIO2_IO15
22 echo "48" >/sys/class/gpio/export   #GPIO2_IO16
```

Listing 5.12: Export der GPIOs ,imx_gpio_export.sh, Ausschnitt

Mit den drei Funktionen *new_gpio()*, *set_value()* und *get_value()* kann auf das GPIO-Struct zugegriffen werden (siehe Listing 5.14). Da die GPIOs selbst, und auch deren Wert über Dateipfade angesprochen werden, wurde eine Struktur erstellt, um die Pfade eines GPIOs zu bündeln (siehe Listing 5.13).

```
15
16 /* Gpio struct defines properties of a single gpio */
17 typedef struct gpio{
18   char name[NAME_LEN];
19   char direction[PATH_LEN];
20   char value[PATH_LEN];
```

Listing 5.13: GPIO Struct,imx_gpio_test.c

Mit den drei Funktionen *new_gpio()*, *set_value()* und *get_value()* kann auf das GPIO-Struct zugegriffen werden (siehe Listing 5.14).

```
22
23 /* 'Getter' and 'Setter' functions encapsulate access to gpio */
24 int new_gpio(Gpio*, char*);
25 int set_value(Gpio*, int);
26 /* A new gpio is created by assigning a name and
27 *  concatenating the properties to obtain paths for
28 *  value and direction */
29 int new_gpio(Gpio* gpio, char* name){
30   char path[PATH_LEN];
31   strcpy(path, GPIO_PATH);
32   strcat(path, name);
33   strcat(path, "/");
```

```
34   strcpy(gpio->value, path);
35   strcpy(gpio->direction, path);
36   strcat(gpio->value, "value");
37   strcat(gpio->direction, "direction");
38   strcpy(gpio->name, name);
39   return EXIT_SUCCESS;
40 }
41
42 /* Value of gpio is read as either '0' or '1' ASCII-character
43 *  from virtual 'value' file.
44 *  The value is obtained as the integer value of the ASCII character.
45 *  Subtract ASCII-Value of '0' to obtain integer 0 or 1 */
46 int get_value(Gpio* gpio){
47   char logbuf[LOG_BUFLEN];
48   int value = -1;
49   FILE* fp=NULL;
50   if ((fp=fopen(gpio->value, "r")) != NULL){
51     value=fgetc(fp)-ASCII_ZERO;
52     fclose(fp);
53   }
54   else{
55     snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't read value: %s %s\n",gpio
       ->name ,strerror(errno));
56     put_log(logbuf);
57   }
58   return value;
59 }
60
61
62 /* ADD ASCII-Value of '0' to obtain integer value of
63 *  ASCII character '0' or '1' */
64 int set_value(Gpio* gpio, int value){
65   char logbuf[LOG_BUFLEN];
66   FILE* fp=NULL;
67   if ((fp=fopen(gpio->value, "w")) != NULL){
68       fputc(value + ASCII_ZERO,fp);
69       fclose(fp);
70   }
71   else{
72     snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't read value: %s %s\n",gpio
       ->name ,strerror(errno));
73     put_log(logbuf);
74     return EXIT_FAILURE;
75   }
76   return EXIT_SUCCESS;
77 }
```

Listing 5.14: Getter- und Setter Funktionen, imx_gpio_test.c

Die Namen der exportierten GPIOs sind in einem Array fest hinterlegt (siehe Listing 5.15).

```
35   Gpio out_gpios[NUM_GPIO_PAIR];
36   Gpio in_gpios[NUM_GPIO_PAIR];
37   // set names of gpios
```

```
38  char* out_gpio_names[NUM_GPIO_PAIR] = {
39  "gpio139",
40  "gpio45",
41  "gpio47",
42  "gpio49",
43  "gpio50",
44  "gpio51",
45  "gpio52",
46  "gpio53",
47  "gpio54",
48  "gpio132",
49  "gpio125",
50  "gpio95",
51  "gpio93",
52  "gpio30",
53  "gpio26",
54  "gpio4",
55  };
```

Listing 5.15: Array mit den als Ausgang festgelegten GPIOs, imx_gpio_test.c

In der eigentlichen Testroutine werden die Ausgangspins zunächst auf ’0’ und später auf ’1’ gesetzt, und jeweils am verbundenen Eingangspin überprüft, ob der erwartete Pegel anliegt. Wenn mindestens einer der beiden Zustände bei einem GPIO-Paar nicht detektiert werden kann, wird ein Fehlerzähler inkrementiert und die Nummern der beiden GPIOs werden in einer Fehlermeldung gelogged.

Ist der Fehlerzähler am Ende des Tests ungleich Null, gilt der Test als nicht bestanden, und das Makro *EXIT_FAILURE* wird zurückgegeben. Dieser Testvorgang ist in Listing 5.16 dargestellt.

```
76  for(int i=0; i<num_gpio_pair; i++){
77    new_gpio(&out_gpios[i],out_gpio_names[i]);
78    new_gpio(&in_gpios[i],in_gpio_names[i]);
79  }
80  // set all outputs to zero
81  for(int i=0; i<num_gpio_pair; i++){
82    set_value(&out_gpios[i],0);
83  }
84  // compare inputs to corresponding outputs
85  for(int i=0; i<num_gpio_pair; i++){
86    int out = get_value(&out_gpios[i]);
87    int in = get_value(&in_gpios[i]);
88    cmp_result[i]=(out == in);
89    //printf("%s, %s out:%d in:%d \n", out_gpios[i].name, in_gpios[i].name,
       out,in);
90  }
91  // set all outputs to one
92  for(int i=0; i<num_gpio_pair; i++){
93    set_value(&out_gpios[i],1);
94  }
95  // compare inputs to corresponding outputs
```

```
96    for(int i=0; i<num_gpio_pair; i++){
97      int out = get_value(&out_gpios[i]);
98      int in = get_value(&in_gpios[i]);
99      cmp_result[i]=((out == in)&&cmp_result[i]);
100     //printf("%s, %s out:%d in:%d \n", out_gpios[i].name, in_gpios[i].name,
      out,in);
101   }
102   for(int i=0; i<num_gpio_pair; i++){
103     if(cmp_result[i] == 0){
104       num_broken_gpio ++;
105       snprintf(logbuf, LOG_BUFLEN,"FAILED: %s and %s connection test failed\
      n",out_gpios[i].name, in_gpios[i].name);
106       put_log(logbuf);
107     }
108   }
109   if(num_broken_gpio == 0){
110     snprintf(logbuf, LOG_BUFLEN,"OK:     GPIO test Succesfull\n");
111     put_log(logbuf);
112     return EXIT_SUCCESS;
113   }
114   else{
115     snprintf(logbuf, LOG_BUFLEN,"FAILED: GPIO test failed, %d of %d gpio
      pairs not connected\n",num_broken_gpio, num_gpio_pair);
116     put_log(logbuf);
117     return EXIT_FAILURE;
118   }
```

Listing 5.16: Test der Ausgangs- und Eingangs Pins, imx_gpio_test.c

5.3.7. Safety Logik

Die Watchdog-Funktion auf dem M48 ist durch eine diskrete Überwachungslogik realisiert. Abbildung 5.11 zeigt die Schaltung in vereinfachter Form.

Jeder der Prozessoren muss in regelmäßigen Abständen ein Triggersignal senden, das die Überwachungslogik zurücksetzt. Durch einen Hochpass am Eingang der Schaltung wird sichergestellt, dass nur ein regelmäßiger Flankenwechsel die Logik triggern kann. Das Triggersignal schaltet einen MOSFET, der den Kondensator eines RC-Glieds entlädt. Die Spannung des RC-Glieds wird durch einen Komparator überwacht, dessen Schaltschwelle bei $2/3V_{DD}$ liegt. Diese Schaltung ist für jeden der Prozessoren einmal vorhanden. Die Ausgänge der beiden Komparatoren sind durch einen gemeinsamen Pull-Up-Widerstand als ein „wired-and“ verbunden. Sobald das Triggersignal eines Prozessors ausfällt, zieht der Komparator den Ausgang SAFE_STATE_L auf Masse. Die Reaktion auf dieses Signal wird produktspezifisch implementiert. Ein Beatmungsgerät beispielsweise könnte seine Funktion einstellen und in einen als sicher definierten Zustand versetzt werden.

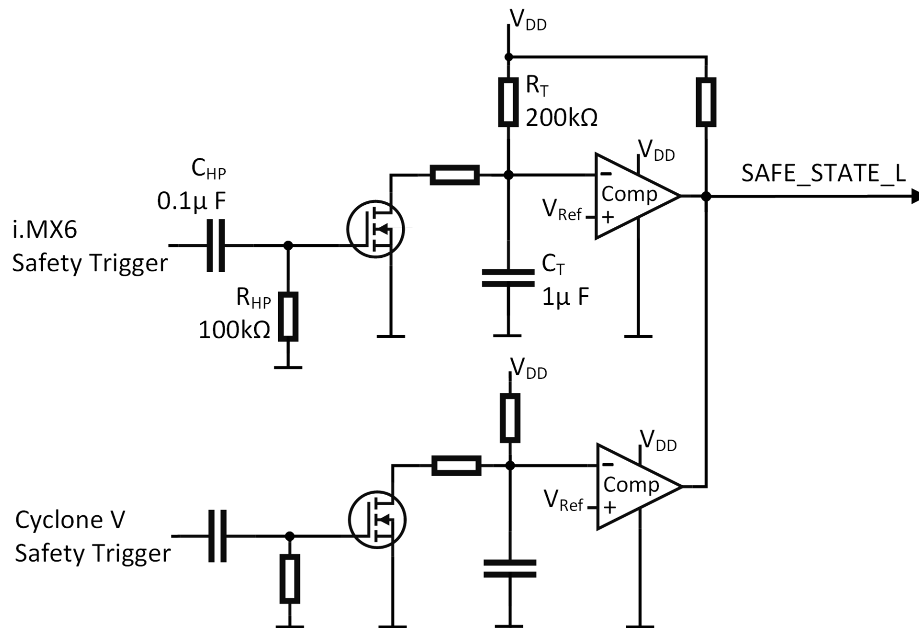

Abbildung 5.11.: M48 Safety Logik, vereinfachter Schaltplan

Das Triggersignal wird über einen GPIO ausgegeben und muss auf jedem Prozessor in der produktspezifischen Software erzeugt werden. Die Frequenz des zu erzeugenden Signals kann anhand der Komponenten in den RC-Gliedern bestimmt werden. Aus der Zeitkonstante $T = R_T \cdot C_T = 200\,k\Omega \cdot 1\mu F = 200\,ms$ ergibt sich eine Mindestfrequenz von 5 Hz, da sich der Kondensator in der Zeit T auf $0,63\,V_{DD}$ auflädt. Erreicht werden muss auch die Grenzfrequenz des Hochpasses, die sich zu $\frac{1}{2\pi R_{HP} C_{HP}} = \frac{1}{2\pi \cdot 100\,k\Omega \cdot 0.1\,\mu F} \approx 16\,Hz$ berechnet.

Als Bestandteil der Testroutine wurde jeweils ein Shellscript für den i.MX6 (Listing 5.17) und den Cyclone V (Listing 5.18) erstellt, das den als SAFETY_TRIGGER definierten GPIO mit einer Frequenz von 50 Hz toggelt, d.h. den Ausgang abwechselnd zwischen „high“ und „low“ umschaltet. Der Zustand des Ausgangs SAFE_STATE_L wird auf dem Testboard durch eine LED angezeigt und zudem auf einen GPIO des i.MX6 zurückgekoppelt.

Soll der Funktionstest für die Safety Logik auf dem i.MX6 erfolgreich verlaufen, muss zuvor Script 5.18 auf dem Cyclone V ausgeführt werden, um das Triggern der Safety Logik von dessen Seite zu starten. Zu Beginn des Funktionstests auf dem i.MX6 wird das Script 5.17 als ein Hintergrundprozess gestartet, so dass auch der i.MX6 die Safety Logik kontinuierlich triggert. Die Prozess-ID dieses Hintergrundprozesses wird dabei in einer Variable gespeichert, um später wieder auf den Prozess zugreifen zu können. Nach einer kurzen Verzögerung wird durch die Funktion *test_imx_safety_toggle* (Listing 5.19) überprüft, ob das Signal SAFE_STATE_L am dazugehörigen GPIO anliegt. Hat das Signal den Zustand „high“, wur-

de die Safety Logik nicht ausgelöst, d.h. beide Prozessoren erzeugen einen regelmäßigen Flankenwechsel an deren SAFETY_TRIGGER Ausgängen.

Vor Abschluss des Funktionstests wird dem Anwender die Möglichkeit gegeben, die korrekte Funktion der Safety-Logik, anhand der dazugehörigen LED, auch visuell zu verifizieren. Erst nach Bestätigung einer Eingabeaufforderung wird das Triggern des SAFETY_TRIGGER Ausgangs gestoppt. Hierzu wird dem Hintergrundprozess über den Linux Befehl *kill* ein Signal gesendet, das diesen veranlasst, sich selbst zu beenden. Der Prozess kann hierbei über die vorher gespeicherte Prozess-ID eindeutig angesprochen werden.

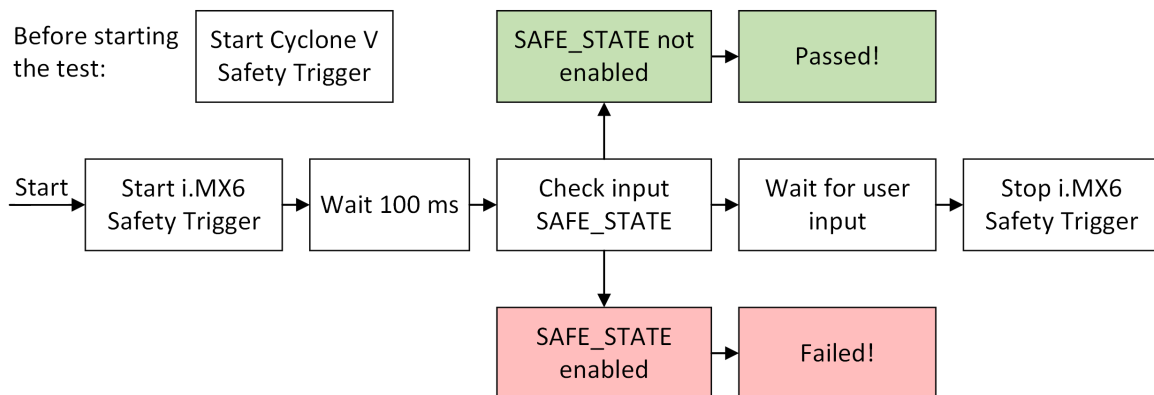

Abbildung 5.12.: i.MX6 Safety Logik Test, Flussdiagramm

```
#!/bin/bash
function cleanup {
  echo "83" >/sys/class/gpio/unexport
  echo "124" >/sys/class/gpio/unexport
}
set -e # stop script if a command fails
#gpio is unexported when skript is killed with CTRL+C
trap cleanup EXIT

# export SAFETY_TRIGGER imx GPIO3_IO19
echo "83" >/sys/class/gpio/export
echo "out" >/sys/class/gpio/gpio83/direction
# gpio124 is connected to SAFE_STATE_L on testboard
echo "124" >/sys/class/gpio/export
echo "in" >/sys/class/gpio/gpio124/direction

# toggle output with 50Hz (20ms per loop)
while true
do
  echo "1" >/sys/class/gpio/gpio83/value
  sleep 0.01
  echo "0" >/sys/class/gpio/gpio83/value
  sleep 0.01
done
```

Listing 5.17: Triggern der Safety Logik auf dem i.MX 6, imx_safety_toggle.sh

```
6 # export SAFETY_TRIGGER hps GPIO62
7 echo "431" >/sys/class/gpio/export
8 echo "out" >/sys/class/gpio/gpio431/direction
```

Listing 5.18: Safety Logic GPIOs auf dem Cyclone V, altsoc_safety_toggle.sh, Ausschnitt

```
110 function start_imx_safety_toggle () {
111   if [[ $test_imx_safety = "y" ]]
112   then
113     # "./some_executable &" runs process in background
114     ./shell_source/imx_safety_toggle.sh &
115     # $! holds process ID of last process executed in background
116     IMX_SAFETY_PID=$!
117     echo "If Testboard LED V16 is on, both safety lines are toggling"
118   fi
119 }
120 function test_imx_safety_toggle () {
121   if [[ $test_imx_safety = "y" ]]
122   then
123     sleep 0.1
124     SAFE_STATE=$(</sys/class/gpio/gpio124/value)
125     if [[ $SAFE_STATE = "1" ]]
126     then
127       echo "PASSED: Active safety logic detected"| tee -a ${logfile}
128     else
129       echo "FAILED: Active safety logic not detected"| tee -a ${logfile}
130     fi
131   fi
132 }
133 function stop_imx_safety_toggle () {
134   if [[ $test_imx_safety = "y" ]]
135   then
136     wait_for_finish_test_input
137     kill $IMX_SAFETY_PID
138   fi
139 }
```

Listing 5.19: Shell-Funktionen für Safety Logik Test, test_functions.sh, Zeilen 110-139

5.3.8. Konfigurierbare Testsequenz

Die einzelnen Testroutinen werden im Rahmen einer Testsequenz aufgerufen (Listing 5.21). Die Testsequenz wird durch Variablen konfiguriert, die in der Datei *test_config.txt* (Listing 5.20 definiert werden.

In der Testsequenz werden Shell-Funktionen aufgerufen, die die eigentliche Testroutine ausführen und auch den Rückgabewert auswerten. Listing 5.22 zeigt die Funktionen *update_succes_rate()* und *test_imx_eeprom_partnumber()*. Der Aufbau der meisten Funktionen in *test_functions.sh*, die eine Testroutine aufrufen, entspricht dem Aufbau von *test_imx_eeprom_partnumber()*.

Wenn die Testroutine durch Setzen der dazugehörigen Konfigurationsvariablen auf 'yes', bzw. 'y' aktiviert wurde, wird die Testroutine über die Shell aufgerufen und ausgeführt. Zur Testlaufzeit werden Logmeldungen von der Testroutine in die Log-Datei und gegebenenfalls in die Standardausgabe geschrieben. Wenn die aufgerufene Testroutine sich beendet hat, wird der erhaltene Rückgabewert von der Funktion *update_succes_rate()* ausgewertet. Ist die Testroutine mit einem Fehler zurückgekehrt, wird ein Zähler inkrementiert.

Listing 5.23 zeigt die Funktion zur abschließenden Ausgabe des Testergebnisses. Ist der Fehler-Zähler *total_result* ungleich Null, gilt der Test als nicht bestanden, und die Anzahl der fehlgeschlagenen Tests wird ausgegeben.

```
# M48 function test config
#************************************************
# either <soc=altsoc> or <soc=imx> must be specified
soc=imx
# Tests for i.MX6DL (uP1)
test_imx_eeprom_partnumber=y
test_imx_i2c2=y
test_imx_i2c3=y
test_imx_uart0=n
test_imx_uart1=y
test_imx_uart2=y
test_imx_uart3=y
test_imx_gpios=y
test_imx_safety=y
test_imx_spi_ss0=y
test_imx_spi_ss1=y
test_imx_spi_ss2=y
test_imx_spi_ss3=y

# Tests for Cyclone V (uP2)
test_altsoc_i2c3=n
test_altsoc_safety=y
```

Listing 5.20: Test-Config Datei, test_config.txt

```
logfile="./log/logfile.txt"
# overwrite old logfile
echo "*********************************************"| tee ${logfile}
echo "* M48 Function Test"              | tee -a ${logfile}
echo "*********************************************"| tee -a ${logfile}
date >> ${logfile}
echo "Loading test config..."
# import shell scripts
source test_config
source test_functions.sh
# total_result holds number of failed tests
total_result=0
if [[ $soc = "imx" ]]
then
  echo "iMX.6________________________________________"| tee -a ${logfile}
  start_imx_safety_toggle
  test_imx_safety_toggle
  test_imx_eeprom_partnumber
  test_imx_i2c2
  test_imx_i2c3
  test_imx_spi_ss0
  test_imx_spi_ss1
  test_imx_spi_ss2
  test_imx_spi_ss3
  test_imx_uart0
  test_imx_uart1
  test_imx_uart2
  test_imx_uart3
  test_imx_gpios
  stop_imx_safety_toggle
elif [[ $soc = "altsoc" ]]
then
  echo "Cyclone V________________________________________"| tee -a ${logfile
    }
  start_altsoc_safety_toggle
  wait_for_finish_test_input
  stop_altsoc_safety_toggle
else
    echo "no soc specified"
fi
print_result
cp ${logfile} /boot/log
echo "a copy of the logfile has been written to /boot/log"
```

Listing 5.21: Konfigurierbare Testsequenz, run_test.sh

```
#!/bin/bash
#****************************************************
# M48 test routine calls and common functions
#****************************************************

# Count total number of failed tests
function update_success_rate () {
  # $? holds the exit status of the most recently run process
```

```
9     # Succesfull tests return '0', Failed Tests return '1'
10    let "total_result = total_result + $?"
11  }
12  # EEPROM Partnumber
13  function test_imx_eeprom_partnumber () {
14    if [[ $test_imx_eeprom_partnumber = "y" ]]
15    then
16      ./c_source/imx/i2c/i2c1/imx_i2c_eeprom_test
17      update_success_rate
18    fi
19  }
```

Listing 5.22: Test Funktionen, Ausschnitt I, test_functions.sh

```
164 function print_result () {
165   if [[ $total_result = 0 ]]
166   then
167     echo "Test Passed!"| tee -a ${logfile}
168   else
169     echo "Test Failed!"| tee -a ${logfile}
170     echo "Number of failed tests:" ${total_result} | tee -a ${logfile}
171   fi
172 }
```

Listing 5.23: Test Funktionen, Ausschnitt II, test_functions.sh

6. Test und Bewertung

In diesem Kapitel wird die Funktionstestumgebung im Hinblick auf die aufgestellten Anforderungen bewertet. Dafür werden zunächst der Funktionsumfang und die Stabilität des Linuxsystems beschrieben und eingeordnet. Dann werden die Testroutinen durch individuell zu entwickelnde Methoden zunächst einzeln getestet, bevor ein zusammenhängender Test aller Routinen im Rahmen der Testsequenz vorgenommen wird.

6.1. Linux System

Es wurde erfolgreich ein Linux System auf dem M48 installiert. Auf beiden SoCs können Anwendungen erstellt werden, so dass das M48 auf Basis der Testroutinen und des dazugehörigen Makefiles als Entwicklungsumgebung genutzt werden kann.

Auf dem i.MX6 wurde eine Desktopumgebung installiert, die mit Tastatur und Maus flüssig bedient werden kann.

Für beide SoCs wurde erfolgreich eine Netzwerkverbindung eingerichtet. Dadurch können neue Programme über einen Paket-Manager nachinstalliert werden, so dass die Anforderung eines flexiblen Systems als erfüllt betrachtet werden kann.

Eine Verbindung zur Datenübertragung zwischen i.MX6 und Cyclone V wurde nicht implementiert, da dafür zeitliche Ressourcen fehlten.

Die Verwendung von BTRFS als Dateisystem hat sich als instabil herausgestellt. Gegen Ende der Implementierungsphase hat ein Reset des Boards das BTRFS-Dateisystem beschädigt, so dass kein Zugriff mehr auf das rootfs bestand. Das System konnte nicht mehr wieder hergestellt werden, und es musste auf ein Backup des Betriebssystems zurückgegriffen werden, auf dem die Testumgebung dann neu eingerichtet wurde. Die Wahrscheinlichkeit eines Dateisystemfehlers könnte stark reduziert werden, wenn das Board konsequent erst nach einem Herunterfahren des Systems ausgeschaltet würde. Diese Anforderung ist in einem Embedded System jedoch schwer zu realisieren.

Die Tabellen 6.2 und 6.1 geben einen Überblick über die Einbindung der Schnittstellen und Hardware-Komponenten in das Linux System.

Tabelle 6.1.: Zugriff auf i.MX6 Komponenten und Abdeckung durch Testroutinen

No.	Komponente	Zugriff durch System	Testroutine
1	I2C_2 On-Module-EEPROM	möglich	implementiert
2	I2C_3 Testboard EEPROM	möglich	implementiert
3	I2C_4 Testboard EEPROM	möglich	implementiert
4	SPI_3 EEPROMs	möglich	implementiert
5	SPI_3 Generisches SPI-Device an SS4	möglich	nicht implementiert
6	USB Host	möglich	nicht implementiert
7	USB OTG	möglich	nicht implementiert
8	UART 1-4	möglich	implementiert
9	GPIOs	möglich	implementiert
10	Safety Logik	möglich	implementiert
11	HDMI	möglich	nicht implementiert
12	LVDS1 LVDS2	vorerst nicht möglich	nicht implementiert
13	Ethernet Interface	möglich	nicht implementiert
14	SDIO	vorerst nicht möglich	nicht implementiert
15	EIM Parallel Interface	vorerst nicht möglich	nicht implementiert
16	PCI Express	möglich Hardwaremodifikation nötig	nicht implementiert

Tabelle 6.2.: Zugriff auf Cyclone V Komponenten und Abdeckung durch Testroutinen

No.	Komponente	Zugriff durch System	Testroutine
1	I2C_0 Testboard EEPROM	möglich	nicht implementiert
2	UART 0	möglich	nicht implementiert
3	Safety Logik	möglich	Triggern der Logik implementiert
4	GPIOs	möglich	nicht implementiert
5	SDIO	vorerst nicht möglich	nicht implementiert
6	EIM Parallel Interface	vorerst nicht möglich	nicht implementiert
7	QSPI Flash	vorerst nicht möglich	nicht implementiert

6.2. Funktionstest

Im Folgenden soll der Funktionstest auf das Verhalten der Testroutinen in verschiedenen Fehlerfällen getestet werden.

6.2.1. Testroutinen

Die Testroutinen können nicht explizit auf das Fehlen von Hardwarekomponenten getestet werden, da z. B. das Entfernen der EEPROMS vom Testboard nur schwer wieder rückgängig zu machen ist. Das „Entfernen von Hardware-Komponenten aus dem Linux System durch Deaktivieren des entsprechenden Nodes im Device Tree würde keinen in der Realität zu erwartenden Fehlerfall widerspiegeln. Bei einigen Tests wird sich daher in Teilen auf die Softwareseitige Manipulation von Komponenten beschränkt.

i.MX6 I2C On-Module-EEPROM

Für den Test des On-Board EEPROMs, der unter anderem die Partnumber enthält, werden zwei zusätzliche Funktionen zum Löschen und Setzen der Partnumber erstellt. Die Konsolenausgabe 6.1 zeigt den Ablauf des Tests, bei dem die Partnumber zunächst gelöscht und nach der Fehlermeldung der Testroutine wieder zurückgeschrieben wird. Die vorhandene Partnumber wird dann erfolgreich identifiziert.

```
root@M48_uP1:~/# ./i2c1_erase_eeprom
erasing EEPROM...
succesfully erased EEPROM
root@M48_uP1:~/# ./imx_i2c_eeprom_test
FAILED: imx i2c eeprom partnumber: ???????
root@M48_uP1:~/# ./i2c1_eeprom_set_partno
written partno: 8421901
root@M48_uP1:~/# ./imx_i2c_eeprom_test
PASSED: imx i2c eeprom partnumber: 8421901
```

Konsolenausgabe 6.1: i.MX6 I2C On-Module-EEPROM Testroutine

i.MX6 I2C Testboard EEPROMs

Für den Test der EEPROMs an SPI und I^2C muss die Testroutine ergänzt werden, so dass der Test zur Laufzeit pausiert werden kann, um dann einzelne Bits des EEPROMs zu manipulieren.

Hierzu wird die Testroutine um eine Warteschleife und einen Signal-Handler erweitert. In Listing 6.2 sind die Ergänzungen an der ursprünglichen Testroutine zusammengefasst. Nach dem Beschreiben des EEPROMs mit Nullen wird der Test pausiert und der Dateihandler des EEPROMs geschlossen. Dann wird manuell mit dem Editor Nano auf den EEPROM zugegriffen, um drei Zeichen zu löschen und zwei neue Zeichen am Anfang des Speicherbereichs hinzuzufügen. Anschließend wird das Signal *SIGUSR1* mit dem Programm kill an die Testroutine geschickt, die sich in einer Warteschleife befindet. Der Signalhandler reagiert auf das Signal mit dem Setzen eines *continue_flag*. Dies löst die Testroutine aus der Warteschleife, und der Test wird fortgesetzt. Dabei werden die Manipulationen am EEPROM von der Testroutine entdeckt, und der Test schlägt fehl.

```
1  #define TEST_THE_TEST FALSE
2
3  // signal-handler for "continue!"-signal
4  void sigh(int);
5  volatile sig_atomic_t continue_flag = FALSE;
6
7  int main(void){
8    [...]
9    // write all zeros (1024 Byte) to EEPROM
10   for(int i=0; i<EEPROM_SIZE; i++){
11     if (fputc(0x00, fp) < 0){
12       printf("ERROR in fputc %s\n",strerror(errno));
13       return EXIT_FAILURE;
14     }
15   }
16   // go back to start of Memory
17   rewind(fp);
```

```c
18   // Halt the test at this point when testing this testroutine
19   if(TEST_THE_TEST){
20     fclose(fp);
21     printf("test mode, all zeros written\n");
22     printf("execute: kill -10 <pid> to continue\n");
23     // wait for interrupt through signal handler
24     while(!continue_flag){}
25     // open EEPROM Device through driver
26     if((fp=fopen(EEPROM_PATH, "r+")) == NULL){
27       snprintf(logbuf, LOG_BUFLEN, "FAILED: Couldn't open I2C2 Testboard
       EEPROM: %s\n",strerror(errno));
28       put_log(logbuf);
29       return EXIT_FAILURE;
30     }
31   }
32   // End of testing-the-test section
33   // read entire EEPROM  (1024 Byte) and check for "all zeros"
34   [...]
35 }
36
37 void sigh(int signum){
38   // Handle signal SIGUSR_1
39   if(signum == SIGUSR1){
40     continue_flag = TRUE;
41   }
42 }
```

Listing 6.2: Quellcode Ergänzung zum Pausieren des I²C EEPROM Tests, imx_i2c2_testboard_eeprom_test_the_test.c

```
root@M48_uP1:~/# ./imx_i2c2_testboard_eeprom_test &
[5] 3829
test mode, all zeros written
execute: kill -10 3829 to continue
root@M48_uP1:~/# nano /sys/bus/i2c/devices/2-0050/eeprom
# Im Texteditor nano eingeben: <ENTF>,<ENTF>,<ENTF>,'0','0'
root@M48_uP1:~/# kill -10 3829
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte 0
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte 3
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte
 1023
FAILED: 3 Bit Errors on I2C2 Testboard EEPROM
```

Konsolenausgabe 6.3: i.MX6 I2C Testboard EEPROM Testroutine, Fehlgeschlagen

```
root@M48_uP1:~/# ./imx_i2c2_testboard_eeprom_test
PASSED: I2C2 Testboard EEPROM is working
```

Konsolenausgabe 6.4: i.MX6 I2C Testboard EEPROM Testroutine, Bestanden

i.MX6 SPI Testboard EEPROMs

```
root@M48_uP1:~/# ./imx_i2c2_testboard_eeprom_test &
[5] 3829
test mode, all zeros written
execute: kill -10 3829 to continue
root@M48_uP1:~/# nano /sys/bus/i2c/devices/2-0050/eeprom
# Im Texteditor nano eingeben: <ENTF>,<ENTF>,<ENTF>,'0','0'
root@M48_uP1:~/# kill -10 3829
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte 0
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte 3
FAILED: Bit error on I2C2 Testboard EEPROM. Expected 0x00 at Byte
1023
FAILED: 3 Bit Errors on I2C2 Testboard EEPROM
```

Konsolenausgabe 6.5: i.MX6 I2C Testboard EEPROM Testroutine, Fehlgeschlagen

Die Testroutine für die SPI-EEPROMs wird ebenso wie die Testroutine des I^{2}C Bus so ergänzt, dass der Test während des Ablaufs pausiert werden kann. Die Manipulation einzelner Bytes auf dem EEPROM wird auch hier erkannt.

```
root@M48_uP1:~/# ./imx_spi_eeprom_test &
[5] 3829
test mode, all zeros written
execute: kill -10 4465 to continue
root@M48_uP1:~/# nano /sys/bus/spi/devices/spi2.3/eeprom
# Im Texteditor nano eingeben: <ENTF>,<ENTF>,<ENTF>,'0','0'
root@M48_uP1:~/# kill -10 3829
FAILED: Bit error on spi2_3 Testboard EEPROM. Expected 0x00 at Byte 0
FAILED: Bit error on spi2_3 Testboard EEPROM. Expected 0x00 at Byte 3
FAILED: Bit error on spi2_3 Testboard EEPROM. Expected 0x00 at Byte
 1023
FAILED: 3 Bit Errors on spi2_3 Testboard EEPROM
```

Konsolenausgabe 6.6: i.MX6 SPI Testboard EEPROM Testroutine, Fehlgeschlagen

```
root@M48_uP1:~/# ./imx_spi_eeprom_test spi2_3 &
PASSED: spi2_3 Testboard EEPROM is working
```

Konsolenausgabe 6.7: i.MX6 I2C Testboard EEPROM Testroutine, Bestanden

i.MX6 UARTs

In Konsolenausgabe 6.8 fällt auf, dass der Test von UART 0 fehlschlägt, da der zum Test gesendete String in das aktuelle Konsolenfenster geschrieben wird. Dies liegt daran, dass zum

Zeitpunkt des Tests über eine tty-Schnittstelle an UART 0 auf das System zugegriffen wurde. Somit erscheinen nicht nur die in die Standardausgabe geschriebenen Ausgaben im Konsolenfenster des Anwenders, sondern auch die direkt über die Schnittstelle gesendeten Daten. Ein Fehlschlagen der Tests für UART1, UART2 und UART3 konnte durch das Entfernen der Jumper zwischen den TX- und RX-Pins provoziert werden.

```
root@M48_uP1:~/# ./imx_uart_test.py uart0
M48 UART test
FAILED: imx UART0 loopback is not working
root@M48_uP1:~/# ./imx_uart_test.py uart1
FAILED: imx UART1 loopback is not working
root@M48_uP1:~/# ./imx_uart_test.py uart2
FAILED: imx UART2 loopback is not working
root@M48_uP1:~/# ./imx_uart_test.py uart3
FAILED: imx UART3 loopback is not working
```

Konsolenausgabe 6.8: i.MX6 UART 0-3 Testroutinen, Fehlgeschlagen

```
root@M48_uP1:~/# ./imx_uart_test.py uart1
PASSED: imx UART1 loopback is working
root@M48_uP1:~/# ./imx_uart_test.py uart2
PASSED: imx UART2 loopback is working
root@M48_uP1:~/# ./imx_uart_test.py uart3
PASSED: imx UART3 loopback is working
```

Konsolenausgabe 6.9: i.MX6 UART 1-3 Testroutinen, Bestanden

i.MX6 GPIOs

Für den Test der GPIOs werden keine Manipulationen an der Hardware vorgenommen. Die erfolgreich getestete Verbindung einiger GPIOs kann auch durch Messung der 'high'-und 'low'-Pegel an den Widerständen mit einem Multimeter bestätigt werden. Die Ursache für das Fehlschlagen der Tests zweier GPIO-Paare konnte noch nicht gefunden werden.

```
root@M48_uP1:~/# ./imx_export_gpios.sh
root@M48_uP1:~/# ./gpio_test
FAILED: gpio95 and gpio94 connection test failed
FAILED: gpio30 and gpio27 connection test failed
FAILED: GPIO test failed, 2 of 16  gpio pairs not connected
root@M48_uP1:~/# ./imx_unexport_gpios.sh
```

Konsolenausgabe 6.10: i.MX6 GPIOs Testroutine, Fehlgeschlagen

Safety Logik

Für den Test der Safety Logik wird das Triggersignal von Seiten des Cyclone V zunächst deaktiviert und nach Fehlschlagen der Testroutine wieder aktiviert, so dass die Funktion der Safety Logik erfolgreich detektiert werden kann.

```
# Triggersignal auf dem Cyclone V inaktiv
# Testroutine auf dem i.MX6 starten
root@M48_uP1:~/# source test_functions.sh
root@M48_uP1:~/# source test_config
root@M48_uP1:~/# start_imx_safety_toggle
[1] 573
If Testboard LED V16 is on, both safety lines are toggling
root@M48_uP1:~/# test_imx_safety_toggle
FAILED: Active safety logic not detected
# Triggersignal auf dem Cyclone V aktivieren:
root@M48_uP2:~/# altsoc_safety_toggle.sh
# Testroutine auf dem i.MX6 erneut starten
root@M48_uP1:~/# test_imx_safety_toggle
PASSED: Active safety logic detected
```

Konsolenausgabe 6.11: i.MX6 Safety Logik Testroutine, Fehlgeschlagen

6.2.2. Testsequenz

Für den abschließenden Test der gesamten Testsequenz werden alle Testroutinen, mit Ausnahme der Testroutine für den UART 0 des i.MX6, aktiviert. Das System wird so eingerichtet, dass alle Testroutinen den Test bestehen sollten. Dafür werden die Jumper zwischen den TX-und RX-Verbindungen der UARTs gesetzt und die richtige Partnumber im On-Board EEPROM hinterlegt. Das Triggern der Safety Logik auf dem Cyclone V wirde gestartet und die Änderungen an den Testroutinen der Testboard-EEPROMs an I^{2}C und SPI werden rückgängig gemacht.

Die Testsequenz wird ohne Unterbrechung durchlaufen, und alle aktivierten Testroutinen werden ausgeführt. Bis auf die GPIOs, bei denen das Fehlen zweier Verbindungen noch untersucht werden muss, bestehen alle Komponenten den Test.

```
root@M48_uP1:~/m48test# ./run_test.sh
*********************************************
* M48 Function Test
*********************************************
Loading test config...
iMX.6________________________________________
If Testboard LED V16 is on, both safety lines are toggling
PASSED: Active safety logic detected
PASSED: imx i2c eeprom partnumber: 8421901
PASSED: I2C2 Testboard EEPROM is working
PASSED: I2C3 Testboard EEPROM is working
PASSED: spi2_0 Testboard EEPROM is working
PASSED: spi2_1 Testboard EEPROM is working
PASSED: spi2_2 Testboard EEPROM is working
PASSED: spi2_3 Testboard EEPROM is working
PASSED: imx UART1 loopback is working
PASSED: imx UART2 loopback is working
PASSED: imx UART3 loopback is working
FAILED: gpio95 and gpio94 connection test failed
FAILED: gpio30 and gpio27 connection test failed
FAILED: GPIO test failed, 2 of 16  gpio pairs not connected
Press any key to finish test
Test Failed!
test_functions.sh: line 164: 3342 Terminated
Number of failed tests: 1
A copy of the logfile has been written to /boot/log
```

Konsolenausgabe 6.12: Ausführen der Testsequenz

7. Ausblick

Mit dem installierten Linux System als Basis, kann das M48 in diversen darauf aufbauenden Projekten eingesetzt werden. Das Einbinden von spezieller Peripherie, wie z. B. Sensoren oder Eingabegeräten kann anhand der Erläuterungen in dieser Arbeit gegebenenfalls schneller realisiert werden.

Noch während der Bearbeitungszeit dieser Arbeit ist eine frühe Version des Systems in Umlauf gebracht worden. Dieses System soll bei Untersuchungen zur Leistungsaufnahme und Temperaturentwicklung des M48 genutzt werden. Interessant ist hierbei vor allem die Effizienz und Ressourcennutzung des Linux Betriebssystems im Vergleich mit dem in den Seriengeräten eingesetzten vxWorks.

Es wurde auch schon Bedarf für einen Schreibtest für USB-Flashspeicher angemeldet. Dabei sollen unter vxWorks gehäuft auftretende Fehler des Dateisystems auf externen Datenträgern untersucht werden. Ein paralleler Test auf dem M48 unter Linux würde dabei helfen, die Ursache auf einen Hardwarefehler des M48 oder einen Fehler in den vxWorks-Treibern einzugrenzen.

Es wird darüber nachgedacht, auf Basis des M48 ein linuxbasiertes, standardisiertes System für Funktionstests von Leiterkarten in zukünftigen Gerätegenerationen zu entwickeln. Die Funktionstests dienen zur Qualitätsicherung während der Fertigung und werden für neue Produkte meist von Grund auf neu entwickelt. Dabei werden die peripheren Komponenten eines Gerätes mit unterschiedlichster Hardware, von einzelnen Mikrokontrollern bis zu Einplatinenrechnern wie dem Raspberry Pi, nachgebildet, um die Elektronik unter kontrollierten Bedingungen zu testen (siehe Kapitel 2.3, Elektro-Emulations-Test).

Für das neue Konzept würde das M48 in ein Rack-System eingebunden, bei dem sämtliche Schnittstellen über Pfostenstecker oder ähnliche Konnektoren nach außen geführt sind. Die bei den Tests benötigte, produktspezifische Peripherie, würde in Form von standardisierten Karten entwickelt, die in dieses Rack-System eingesetzt werden. Die Flexibilität des Linux Systems erlaubt es dabei auch, eine intuitiv zu erfassende, grafische Benutzerschnittstelle zu implementieren und dadurch den Testablauf und die Auswertung der Ergebnisse zu vereinfachen.

Literaturverzeichnis

[1] ALTERA CORPORATION: *Cyclone V - Booting and Configuration Introduction*, 2014. – URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5400a.pdf

[2] ARM LIMITED: *Cortex-A7*. – URL https://developer.arm.com/products/processors/cortex-a/cortex-a7. – Zugriffsdatum: 30.09.2017

[3] ATMEL: *AT2510B Datasheet*. – URL http://ww1.microchip.com/downloads/en/devicedoc/atmel-8707-seeprom-at25010b-020b-040b-datasheet.pdf

[4] BELLONI, Alexandre ; PETAZZONI, Thomas: *Buildroot vs. OpenEmbedded/Yocto Project: A Four Hands Discussion*. 2016. – URL https://events.linuxfoundation.org/sites/events/files/slides/belloni-petazzoni-buildroot-oe_0.pdf. – Zugriffsdatum: 29.09.2017

[5] BISSON, Gary: *Quickboot on i.MX6*. – URL http://cache.freescale.com/files/training/doc/ftf/2014/FTF-AUT-F0401.pdf. – Zugriffsdatum: 04.10.2014

[6] COMMUNITY, Buildroot: *Buildroot, Making Embedded Linux Easy*. – URL https://buildroot.org/. – Zugriffsdatum: 02.10.2017

[7] DENX SOFTWARE ENGINEERING: *Das U-Boot – the Universal Boot Loader*. 2017. – URL https://www.denx.de/wiki/U-Boot/WebHome. – Zugriffsdatum: 30.09.2017

[8] DEVICES, Boundary: *BD-SL-I.MX6*. – URL https://boundarydevices.com/product/sabre-lite-imx6-sbc/. – Zugriffsdatum: 02.10.2017

[9] DRÄGERWERK AG & CO. KGAA: *Dräger Evita Infinity V500*. – URL https://www.draeger.com/de_de/Hospital/Products/Ventilation-and-Respiratory-Monitoring/ICU-Ventilation-and-Respiratory-Monitoring/Evita-Infinity-V500-ventilator. – Zugriffsdatum: 30.09.2017

[10] ELECTRONICS NOTES: *ICT, In Circuit Test Tutorial*. – URL https://www.electronics-notes.com/articles/test-methods/automatic-automated-test-ate/ict-in-circuit-test-what-is-primer.php. – Zugriffsdatum: 01.10.2017

[11] FREE SOFTWARE FOUNDATION, Inc.: *GNU Lizenzen*. – URL https://www.gnu.org/licenses/. – Zugriffsdatum: 02.10.2017

[12] FREESCALE SEMICONDUCTOR, INC.: *i.MX 6Solo/6DualLite Applications Processor Reference Manual*, 2015

[13] GROUP, The O.: *The Single UNIX Specification, Version 3*. – URL http://www.unix.org/unix/version3/overview.html. – Zugriffsdatum: 02.10.2017

[14] LINARO: *arm-linux-gnueabihf toolchain v.4.9.4*. 2017. – URL https://releases.linaro.org/components/toolchain/binaries/4.9-2017.01/arm-linux-gnueabihf/. – Zugriffsdatum: 25.09.2017

[15] LINUX KERNEL ORGANIZATION, Inc.: *The Linux Kernel Archives*. – URL https://www.kernel.org/category/faq.html. – Zugriffsdatum: 02.10.2017

[16] NXP: *i.MX 6 Series Applications Processors*. 2017. – URL https://www.nxp.com/products/microcontrollers-and-processors/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-6-processors:IMX6X_SERIES. – Zugriffsdatum: 20.09.2017

[17] PROJECT, Yocto: *yocto project*. – URL https://www.yoctoproject.org/. – Zugriffsdatum: 02.10.2017

[18] QUADE, Jürgen: *Embedded Linux lernen mit dem Raspberry Pi*. dpunt.verlag, 2014. – ISBN 978-3-86490-143-0

[19] QUADE, Jürgen ; KUNST, Eva-Katharina: *Kernel- und Treiberprogrammierung mit dem Linux-Kernel - Folge 68*. Linux-Magazin 06/2013. 2013. – URL http://www.linux-magazin.de/Ausgaben/2013/06/Kern-Technik

[20] RASPBERRY PI FOUNDATION: *Raspberry Pi 2 Model B*. – URL https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. – Zugriffsdatum: 30.09.2017

[21] SEABRIGHT TECHNOLOGY: *SD Card Protocol*. 2009. – URL http://wiki.seabright.co.nz/wiki/SdCardProtocol.html. – Zugriffsdatum: 30.09.2017

[22] TEAM, Debian W.: *Debootstrap*. – URL https://wiki.debian.org/de/Debootstrap. – Zugriffsdatum: 02.10.2017

[23] TORVALDS, Linus: *Device Tree Bindings, Kernel Doku*. – URL https://github.com/torvalds/linux/tree/master/Documentation/devicetree/bindings. – Zugriffsdatum: 02.10.2017

[24] TORVALDS, Linus: *linux*. – URL https://github.com/torvalds/linux. – Zugriffsdatum: 02.10.2017

[25] TORVALDS, Linus: *Linux Drivers*. – URL https://github.com/torvalds/linux/blob/master/drivers

[26] VEREIN DEUTSCHER INGENIEURE: *VDI 2206*

[27] WEETECH GMBH: *Funktionstests*. – URL http://www.weetech.de/news-infos/tester-abc/funktionstest/. – Zugriffsdatum: 01.10.2017

[28] WIKIPEDIA: *Dynamic voltage scaling — Wikipedia, The Free Encyclopedia*. 2017. – URL https://en.wikipedia.org/w/index.php?title=Dynamic_voltage_scaling&oldid=796276867. – Zugriffsdatum: 02.10.2017

[29] WIKIPEDIA: *Funktionstest — Wikipedia, Die freie Enzyklopädie*. 2017. – URL https://de.wikipedia.org/w/index.php?title=Funktionstest&oldid=162380695. – Zugriffsdatum: 02.10.2017

[30] WIKIPEDIA: *Unixoides System — Wikipedia, Die freie Enzyklopädie*. 2017. – URL https://de.wikipedia.org/w/index.php?title=Unixoides_System&oldid=167074058. – Zugriffsdatum: 02.10.2017

[31] YAGHMOUR, Karim ; MASTERS, Jon ; BEN-YOSSEF, Gilad ; GERUM, Philippe: *Building Embedded Linux Systems*. O'Reilly and Associates, 2008. – ISBN 978-0596529680

A. DVD

A.1. Bachelorarbeit

- Bachelorarbeit im PDF-Format

A.2. Quellcode des Funktionstests

- C-Quellcode und Makefile
- Python-Quellcode
- Shell-Skripte

A.3. Quellcode des Device Trees

- Device Tree des M48

A.4. Kernel Konfiguration

- Konfigurationsdatei für den Kernel

A.5. Dokumentation

- Device Tree Bindings für Komponenten des M48
- Dokumentation des Build Prozess für den i.MX6 als Textdatei
- Dokumentation des Build Prozess für den Cyclone V als Textdatei

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 4. Oktober 2017	
Ort, Datum	Unterschrift