



Hochschule
Augsburg University of
Applied Sciences

Bachelorarbeit

Fakultät für
Informatik

Studienrichtung
Technische Informatik

Konfiguration und Optimierung des Embedded-Linux-Betriebssystem für Automotive Image Processing Unit

Betreuer: Mladen Kovacev

Verfasser:
Hugues landry Nseupi Nono
Salomon-Idler-Str 25
86159 Augsburg
+49 157 79552970
landrynono60@yahoo.de
Matrikelnr.: 2022666

in Kooperation mit der Firma: EDAG Engineering GmbH

Prüfer: Prof.Dr.-Ing Hubert Högl

Hochschule für angewandte
Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

© 2022 Hugues landry Nseupi Nono

Diese Arbeit mit dem Titel

»Konfiguration und Optimierung des Embedded-Linux-Betriebssystem für
Automotive Image Processing Unit - Betreuer: Mladen Kovacev«

von Hugues landry Nseupi Nono steht unter einer

*Creative Commons Namensnennung-Nicht-kommerziell-Weitergabe unter gleichen
Bedingungen 3.0 Deutschland Lizenz (CC BY-NC-SA).*
<http://creativecommons.org/licenses/by-nc-sa/3.0/de/>



Sämtliche, in der Arbeit beschriebene und auf dem beigelegten Datenträger vorhandene, Ergebnisse dieser Arbeit in Form von Quelltexten, Software und Konzeptentwürfen stehen unter einer GNU General Public License Version 3.

<http://www.gnu.de/documents/gpl.de.html>

Die nachfolgende Arbeit enthält vertrauliche Informationen und Daten der Firma EDAG Engineering GmbH. Veröffentlichungen oder Vervielfältigungen - auch nur auszugsweise oder in elektronischer Form sind ohne ausdrückliche schriftliche Genehmigung der Firma EDAG Engineering GmbH nicht gestattet.

Zusammenfassung

Abstract auf Deutsch. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Abstract

Abstract in English. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Inhaltsverzeichnis

Inhaltsverzeichnis	IV
Abkürzungsverzeichnis	VI
Abbildungsverzeichnis	VII
Verzeichnis der Listings	IX
1 Einleitung	1
1.1 Motivation	1
1.2 Ziel der Arbeit	1
1.3 Überblick über den Aufbau der Arbeit	2
2 Technische Grundlagen	3
2.1 Technische Ausgangssituation	3
2.2 Can Bus Systeme	4
2.2.1 Can Message Frame	4
2.2.1.1 Data Frame	5
2.2.1.2 Remote Frame	6
2.2.1.3 Error Frame	7
2.2.1.4 Overload Frame	8
2.2.2 Can Physical Layer	8
2.3 SPI Interface	9
2.4 Embedded Linux für Xilinx ZynqMP Ultrascale+MPSoC	11
2.5 Komponente des Embedded Linux Betriebssystems für ARM Prozes-	
soren	12
2.5.1 Der Bootloader	12
2.5.2 Device-tree	13
2.5.3 Der Linux Kernel	14
2.5.4 Das Linux Root files System (Rootfs)	15
2.5.5 Der Init Prozess	15
2.5.6 Der Zynq UltraScale+MPSoC Boot-Prozess	15
2.5.6.1 Die Boot-Setup-Phase	16
2.5.6.2 Die Bootloader-Phase	17

2.5.6.3	Der Boot Flow	18
2.5.7	Die Kernel-Boot-Phase	19
2.5.8	Die init-Phase	19
2.6	Petalinux Tool Flow	19
2.6.1	Petalinux Installation	20
2.6.2	Wichtige Petalinux Kommando	21
2.6.3	Petalinux Projekt Strukture	23
3	Umsetzung	25
3.1	Allgemein über das Projekt	25
3.2	Hardware Platform	27
3.2.1	Xilinx ZCU106 Evaluation Board	27
3.2.1.1	Ultrasale + MPSoC Architektur	28
3.2.1.2	Allgemeine Ansicht des Zynq Ultrasale+ MPSoC . .	29
3.2.1.3	Processing System (PS)	32
3.2.1.4	Programmable Logik (PL)	33
3.2.2	MCP251XFD CAN Controller + Transceiver	34
3.2.2.1	CAN FD Controller Modul	36
3.2.2.2	TLE6251 CAN Transceiver	37
3.3	Konfiguration und Bauen des Systems	38
3.3.1	Das Yocto Projekt	38
3.3.1.1	Yocto Layer	38
3.3.1.2	Poky	39
3.3.1.3	Bitbake Engine	40
3.3.1.4	Bitbake Recipes(Bitbake Rezept)	40
3.3.2	Konfigurieren des PetaLinux-Projekts	41
3.3.3	Device-Tree Eintrag für den MCP251XFD CAN-Controller .	43
3.3.4	Rezepte zur Kompilierung der Software Modulen	45
3.3.4.1	Hinzufügen einer neue Layer zu BBLAYERS	46
3.3.4.2	Rezept zur Kompilierung der Software Modulen . .	47
3.3.5	Bauen des Systems	50
3.3.6	Booten des eingebetteten Linux-Images und Test des MCP251xfd CAN-Controller	52
4	Fazit und Ausblick	56
4.1	Fazit	56
4.2	Ausblick	56
Literaturverzeichnis		57

Abkürzungsverzeichnis

APU	Application processing units
ATF	ARM Trusted Firmware
BSP	board support packages
CAN	Control Area Network
CRC	Cyclic Redundancy Check
CSU	Configuration Security Unit
DLC	Data Length Code
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSBL	First Stage Bootloader Codes
IP	Intellectual property
IPU	Image Processing Unit
OCM	On-Chip RAM
PL	Programmable Logic
PLL	Phase Locked Loops
PMU	Platform Management Unit
PS	Processing System
RPU	Real-time processing units
RTR	Remote Transmission Request
SOF	Start of Frame
XSDB	Xilinx System Debugger

Abbildungsverzeichnis

2.1	CAN System Diagram	3
2.2	CAN-Data Frame Architektur	5
2.3	CAN-Remote Frame Architektur	7
2.4	Can-Error-Frame	7
2.5	Can-Overload-Frame	8
2.6	Can-Bus Connexion	8
2.7	CAN_H and CAN_L	9
2.8	SPI Bus	10
2.9	ZCU106-Gerätbaum gpio-keys and leds	13
2.10	der Bootvorgang bei zynq+MPSoCs	18
2.11	PetaLinux-Werkzeugfluss	20
2.12	petalinux Projektstruktur	23
3.1	Versuchsaufbau ZynqMP Ultralcale + mcp251xfd	26
3.2	Xilinx ZCU102 Evaluation Board	27
3.3	Hard-(ARM Cortex-A9/Cortex-A53) und Soft-Prozessoren (Micro-Blaze)	29
3.4	Zynq UltraScale+ MPSoC EV Block Diagram	30
3.5	Zynq UltraScale+ MPSoC Top-Level Blockdiagramm	31
3.6	Detailliertes APU-Blockdiagramm	33
3.7	CAN-Transceiver- und Controller-Modul	34
3.8	CAN Controller Modul	35
3.9	CAN Transceiver Modul	37
3.10	Menuconfig Startbildschirm des PetaLinux-Projekts	39
3.11	Menuconfig Startbildschirm des PetaLinux-Projekts	41
3.12	Pfard zur mcp251xfd Treiber	42
3.13	Konfig Fenster mcp251xfd Treiber	42
3.14	Device-tree Projekt Struktur	43
3.15	Xilinx SPI-Controller	44
3.16	Neue Yocto-Ebene für IPU-NG Software Module	47
3.17	Aktivierung Software Pakete in Rootfs	50
3.18	Formatierung der SD-Karte Partitionen	51
3.19	Laden des mcp251xfd Treibers	53

Abbildungsverzeichnis

3.20 Inbetriebnahme des CAN-Mikrochips	53
3.21 Gesendete CAN-Nachrichten	54
3.22 Empfangene CAN-Nachrichten	54
3.23 Test Statistiken	55

Verzeichnis der Listings

3.1	Inhalt der system-top.dts Datei	43
3.2	Device-tree Eintrag für den mcp251xfd CAN-Controller	44
3.3	Hinzufügen neue Yocto-ebene	46
3.4	Rezept für das sw-main-app Modul	48
3.5	Rezept für das sw-system-module Modul	48
3.6	Rezept für das sw-system-module Modul	49
3.7	Einbindung Software in Rootfs	49

1 Einleitung

1.1 Motivation

Aufgrund der Einsätze von immer mehr Geräten, deren Funktionen uns das Leben erreichten. Egal, ob die Waschmaschine zu Hause, der Drucker in unseren Büros, oder die Kaffeemaschine in der Kantine, werden in alle diese Geräte kleine Computer gebaut, damit sie ihre Aufgabe bequem erledigen. Aber durch die gestiegene Rechenleistung und die erweiterten Kapazitäten von Mikroprozessoren werden die Aufgaben von solche kleinen Computer immer komplexer. Es besteht dann die Möglichkeit, ein vollwertiges Betriebssystem im diesen einzusetzen. Hier hat sich Linux durch die vielseitige Anwendbarkeit und das offene Ökosystem für Embedded Devices besonders bewährt. Am EDAG Engineering GmbH, wurde im Rahmen des internen Projekts, ein Image Processing Unit (IPU) Hardware Plattform auf Basis des Kria KV260 FPGA(Field Programmable Gate Array) entwickelt. Auf dieser Plattform wird dann aufgrund der Komplexität des Projekts ein Linux Betriebssystem eingesetzt, mit dem die 8 wesentlichen Anwendungen des Projekts konfiguriert, kompiliert, und zum User zur Verfügung gestellt wird.

1.2 Ziel der Arbeit

Angesichts der weltweiten Krise auf dem Halbleitermarkt in den letzten Monaten, wurde es immer schwieriger, hochwertige Komponenten, wie die für das Projekt verwendeten Kria KV260 Board zu finden. Statt auf der einzigen Platine des Unternehmens, musste ich meine Arbeit auf einer alternativen Platine durchführen. Also meine Arbeit in den letzten Monaten bei EDAG Engineering GmbH wurde in zwei Aufgaben aufgeteilt. Das erste Ziel dieser Arbeit war es, ein in der Firma entwickeltes CAN FD Controller (mcp251xfd), der über SPI mit einem Zynq UltraScale + MP-SoC ZCU106 Board verbunden ist, in Betrieb zu nehmen, damit verschiedenen Can Node vom Linux angesprochen wird.

Im Anschluss musste ich 3 von den in der Firma entwickelten Applikationen, im Linux bauen, damit das System automatisch mit den Anwendungen bootet. Dafür

müsste ich Rezepte schreiben, die sich darum kümmern werden, die Applikationen zu konfigurieren, zu kompilieren und zu installieren.

1.3 Überblick über den Aufbau der Arbeit

Diese Arbeit lässt sich in 4 Hauptkapitel aufteilen:

- **Die Einleitung:** In der Einleitung werden, die Motivation, das Ziel der Arbeit und ein gesamter Überblick auf dem Ablauf der Arbeit behandeln.
- **In den technischen Grundlagen** wird zuerst erklärt, wie das System (CAN Controller und die Zynq Mp Plattform) gebaut und funktionieren soll. Des Weiteren werden, der CAN Bus System und die SPI Interface erklärt. Dann wird dem Grundprinzip von Embedded Linux Systemen und deren Komponenten erläutert. Zum Schluss erfolgt, die Beschreibung der Petalinux Tools Flow, welches der Build System, der verwendet wird, um Linux Distribution für Xilinx Bausteinen zu kompilieren.
- **Im Versuch Aufbau** wird das Projekt, in dem ich gearbeitet habe dargestellt, dann folgt eine tiefe beschreibung der Hardware. Anschließen wird detailliert auf verschiedenen Schritte für das Bauen des System eingegangen.
- **Im Kapitel Fazit und Ausblick** werden aufgetretene Probleme und Herausforderungen erläutert, es wird analysiert, wie weit das Ergebnis von dem Ziel entfernt ist. Und anschließend wird ein Ausblick auf die möglichen Verbesserungen gegeben.

2 Technische Grundlagen

In einem ersten Schritt wird es darum gehen, die Eigenschaften eines solchen Systems zu beschreiben, das aus einem MCP251XFD CAN Controller und einem ZynqMP besteht. Das dient dazu, die Anforderungen an die Hardware und die Konfiguration des Systems verständlicher zu machen. Und dann werden die CAN Bus Systeme und die SPI Schnittstelle tiefer vorgestellt. Danach folgt eine Beschreibung von allgemeine Embedded Linux System. Im letzten Abschnitt wird das verwendete Build System präsentiert.

2.1 Technische Ausgangssituation

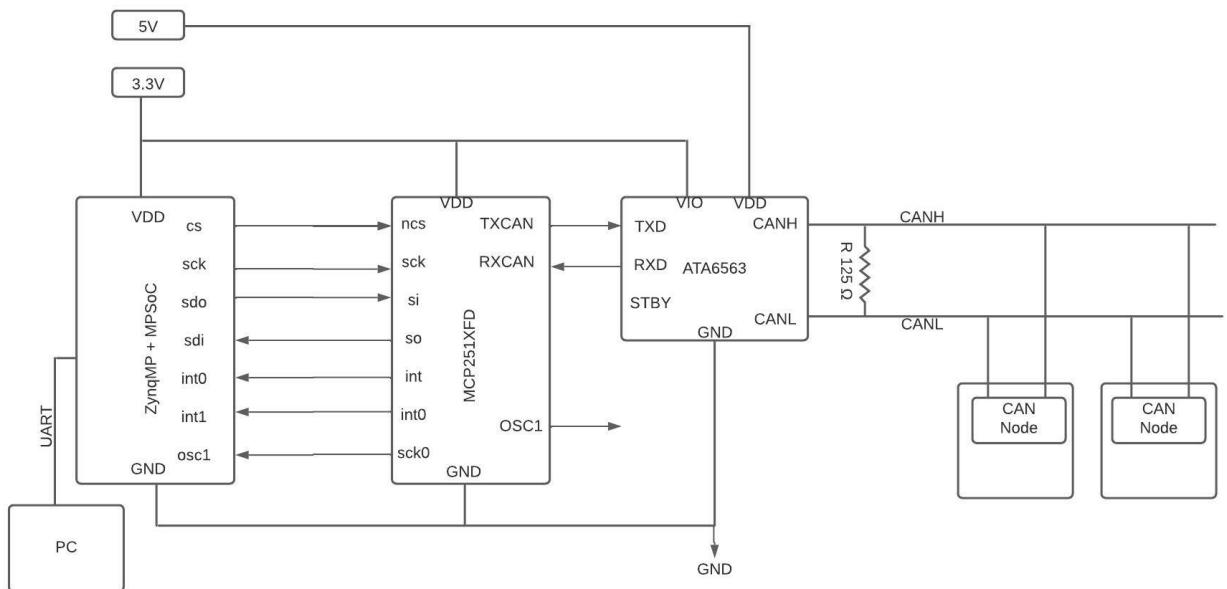


Abbildung 2.1: CAN System Diagram

Das Bild auf der Abbildung 2.1 stellt die Verschaltung zwischen der MCP251xFD CAN Controller und der ZynqMP Plattform dar. Aufgrund der hohen Lizenzgebühren für den internen Ultrascale CAN-Bus-Controller wird ein externer CAN-Controller mit Serial Peripheral Interface (SPI) als Ersatz verwendet, um die Gesamtkosten des Systems zu reduzieren. Das SPI Interface wird also verwendet, um

sowohl der CAN-Controller zu konfigurieren, als auch die CAN-Bus-Daten bis zum FPGA zu übertragen. Auf dem Xilinx FPGA Board werden außer viele andere Peripheriegeräte auch ein SPI-Controller implementiert, mit dem Zweck, die Daten, die vom externen Peripheriegeräte kommen, zu kontrollieren. Zwischen dem MCP251XFD und dem physikalischen Zweidraht-CAN-Bus ist ein CAN-FD Transceiver (ATA6563 von Microchip) zu sehen, der als Schnittstelle zwischen beiden dient. Der bietet unterschiedlicher Empfangs- und Sendefähigkeiten mit einer Hochgeschwindigkeiten von bis 5Mbit/s. In den folgenden Abschnitt gebe ich die Vorteile, ein CAN-Bus zu verwenden.

2.2 Can Bus Systeme

Ein typischer Bereich, in dem die Nutzung von CAN-Bussen unumgänglich ist, wäre die Automobilindustrie. Moderne Auto verfügen Heutzutage über eine Vielzahl an elektronischen Systemen, die miteinander kommunizieren müssen. Und die übliche Verkabelungen wäre mit dem Vielzahl an Steuergeräten kaum mehr möglich. Der CAN-Bus ist in der CAN-Spezifikation von [Bosch \(1991\)](#) als ein Multicast-Kommunikationsprotokoll definiert, das folgende Vorteile aufweist

- CAN ist ein Multi-Master-Broadcast-System. Das heißtt, dass jeder Knoten auf dem Bus mit jedem anderen Knoten kommunizieren kann.
- Der CAN-Bus hat eine Datenübertragungsgeschwindigkeit von bis zu 1 Mbit/s.
- Jeder neue Knoten kann in den Bus eingefügt werden, ohne die ursprüngliche Hardware zu verändern.
- Es bietet eine Fehlerprüfung zur Vermeidung von Busfehlern.
- Das differentielle CAN-Signal bietet eine hohe Rauschunterdrückung.

Da dieses Protokoll sehr viele Vorteile mitbringt, wurde es in den letzten Jahren in der Industrie sehr viel verbreitet. In viele Mikrocontroller werde auf diesem Grund bei der Herstellung ein CAN-Bus eingebaut.

2.2.1 Can Message Frame

In der Sprache des CAN-Standards werden alle Nachrichten als Frames bezeichnet; es gibt Daten-Frames, Remote-Frames, Error-Frames und Overload-Frames. Die an den CAN-Bus gesendeten Informationen müssen definierten Frame-Formaten von

unterschiedlicher, aber begrenzter Länge entsprechen. CAN verfügt über vier verschiedene Arten von Message Frames:

- **Data Frame (Sendet Daten):** Die Daten werden von einem Sendeknoten zu einem oder mehreren Empfangsknoten übertragen.
- **Remote Frame (Fordert Daten an):** Jeder Knoten kann Daten von einem Quellknoten anfordern. Auf einen Remote-Frame folgt somit ein Daten-Frame, der die angeforderten Daten enthält
- **Error Frame (Meldet einen Fehlerzustand):** Jeder Busteilnehmer, egal ob Sender oder Empfänger, kann zu jeder Zeit während einer Daten- oder Remote-Frame-Übertragung einen Fehlerzustand melden.
- **Overload-Frame (Meldet Knotenüberlastung):** Ein Knoten kann zwischen zwei Daten- oder Remote-Frames eine Verzögerung anfordern, das heißt, dass der Overload-Frame nur zwischen Daten- oder Remote-Frame-Übertragungen auftreten kann.

Im Nachfolgenden gehen wir auf der Architektur von den jeweiligen CAN Frame Typen ein.

2.2.1.1 Data Frame

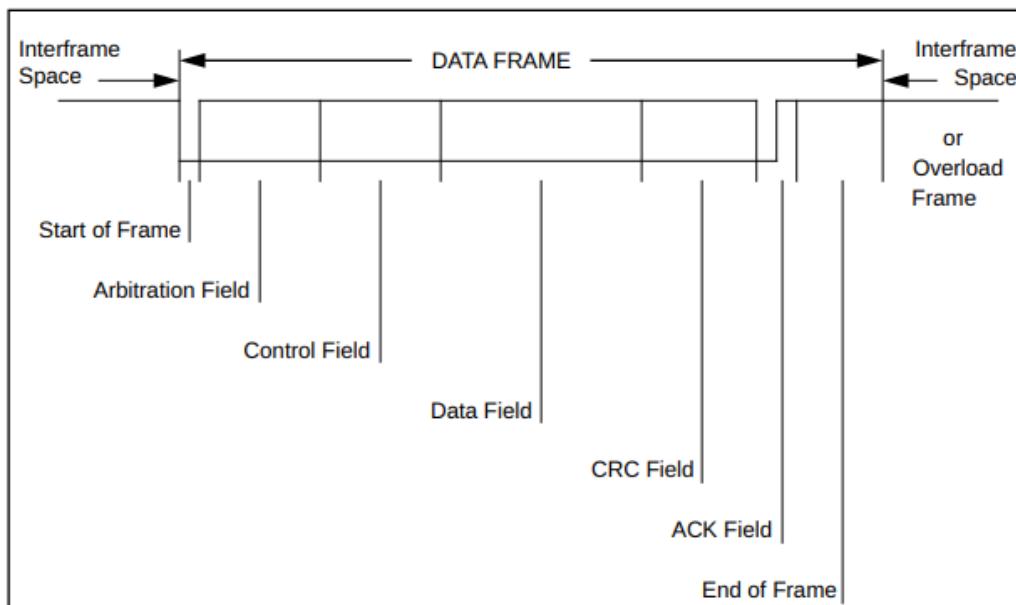


Abbildung 2.2: CAN-Data Frame Architektur Bosch (1991)[p. 12]

Die Abbildung 2.2 beschreibt die 7 Bestandteile, aus denen ein Data Frame besteht, nämlich:

- **SOF(Start of Frame):** Zeigt den Beginn von Daten und Remote Frames an.
- **Arbitration Field:** der besteht auf
 - Identifikator: Die Basis-ID besteht aus 11 Bits und die erweiterte ID aus 29 Bits.
 - RTR(Remote Transmission Request)-Bit: Im Data Frame ist das RTR-Bit "0". Im RTR-Frame hingegen ist es "1".
- **Control Field:** Dient zur Bestimmung der Datengröße und der Länge der Nachrichten-ID. der besteht auf 6 Bits.
 - IDE (Identifikator-Erweiterung): Dieses Bit bestimmt den Identifikator als Basis-ID oder Erweiterte ID.
 - R0,R1: reservierte Bits.
 - DLD (Data Length Code): Er wird zur Bestimmung der Datenlänge verwendet.
- **Data Field:** bis zu 8 Byte Datenfeld.
- **CRC-Field (Cyclic Redundancy Check):** zur Überprüfung der Datenkorrektur.
- **ACK Field (Acknowledgement Field):** um zu bestimmen, ob die Nachricht empfangen wurde oder nicht. Bei Empfang von Daten wird dieses Bit auf High gezogen.
- **EOF (End of Frame):** Zeigt das Ende von Daten- und Remote-Frames an.

2.2.1.2 Remote Frame

Die Abbildung 2.3 beschreibt die Bestandteile eines Remote-Frame. Data-Frame und Remote-Frame sind sich sehr ähnlich. Im Prinzip ist der Remote Frame ein Data Frame ohne das Datenfeld. Dieser besteht in der Regel aus den gleichen Bestandteilen wie der Data Frame.

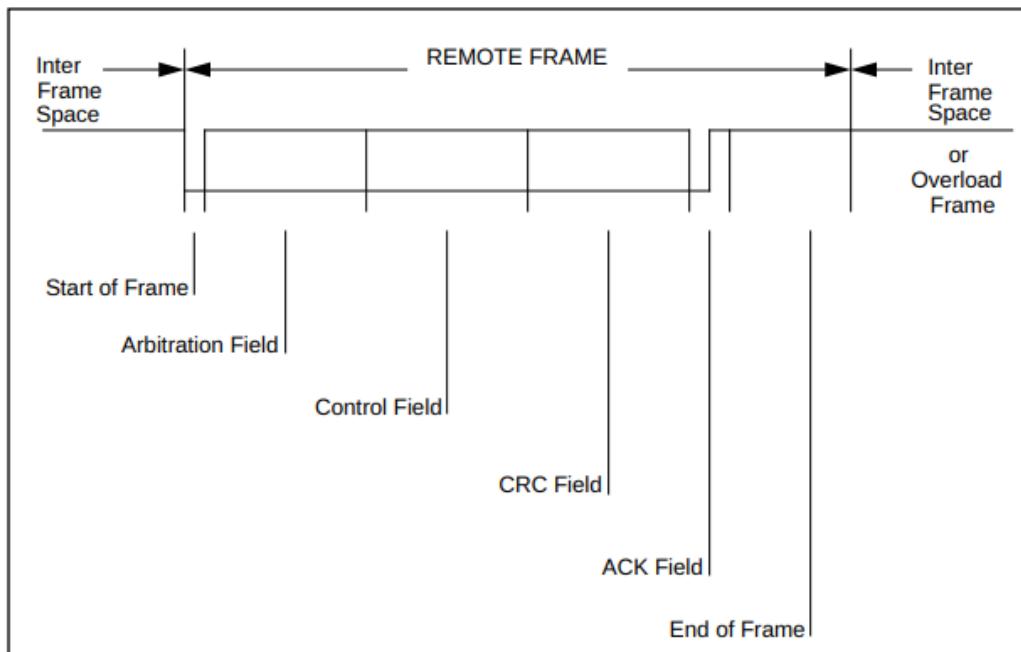


Abbildung 2.3: CAN-Remote Frame Architektur Bosch (1991)[p. 17]

2.2.1.3 Error Frame

Abbildung 2.4 zeigt die Struktur der Error Frame an. Der Error Frame besteht aus zwei Teilen:

- Error Flag: stellt ein Knoten einen Fehlerzustand fest, erzeugt er bis zu 12 Bits "0" für das Fehlerflag.
- Error Delimiter: 8 Bits "1" beenden den Error Frame.

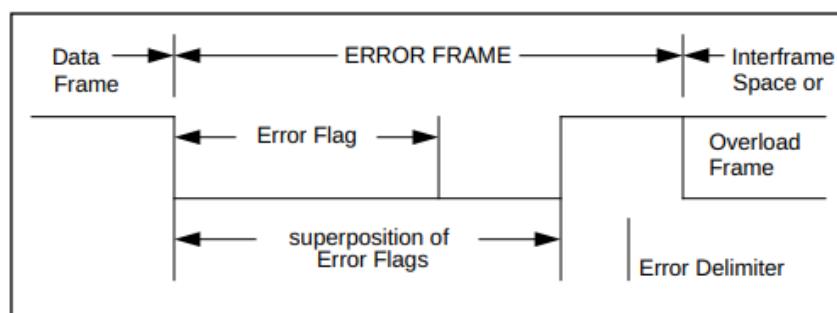


Abbildung 2.4: Can-Error-Frame Bosch (1991)[p. 18]

2.2.1.4 Overload Frame

Abbildung 2.5 ist der Überlastrahmen. Er wird von dem Empfängerknoten erzeugt, um mehr Verzögerung zwischen den Datenrahmen zu erzwingen.

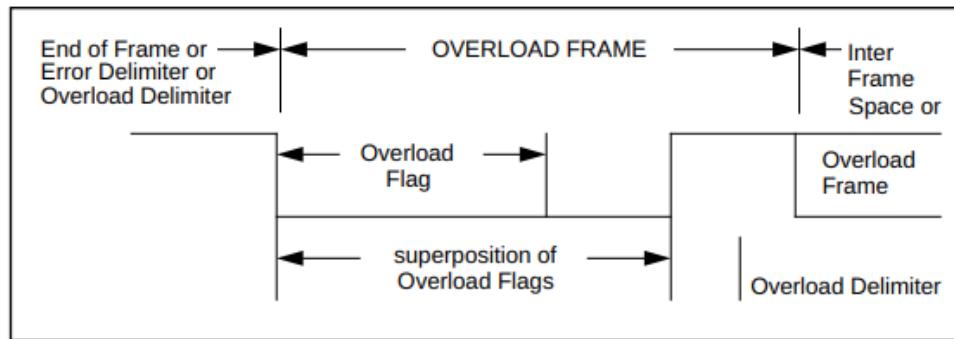


Abbildung 2.5: Can-Overload-Frame Bosch (1991)[p. 19]

2.2.2 Can Physical Layer

Der CAN FD Protokoll, der während dieser Arbeit verwendet wird, ist in ISO 1189-1:2015 definiert. Dieses Protokoll beschreibt nicht die mechanischen, Drähte, und Anschlüsse, aber fordert allerdings, dass die Drähte und Anschlüsse den elektrischen Spezifikationen entsprechen müssen. Abbildung 2.6 zeigt eine CAN-Verbindung mit

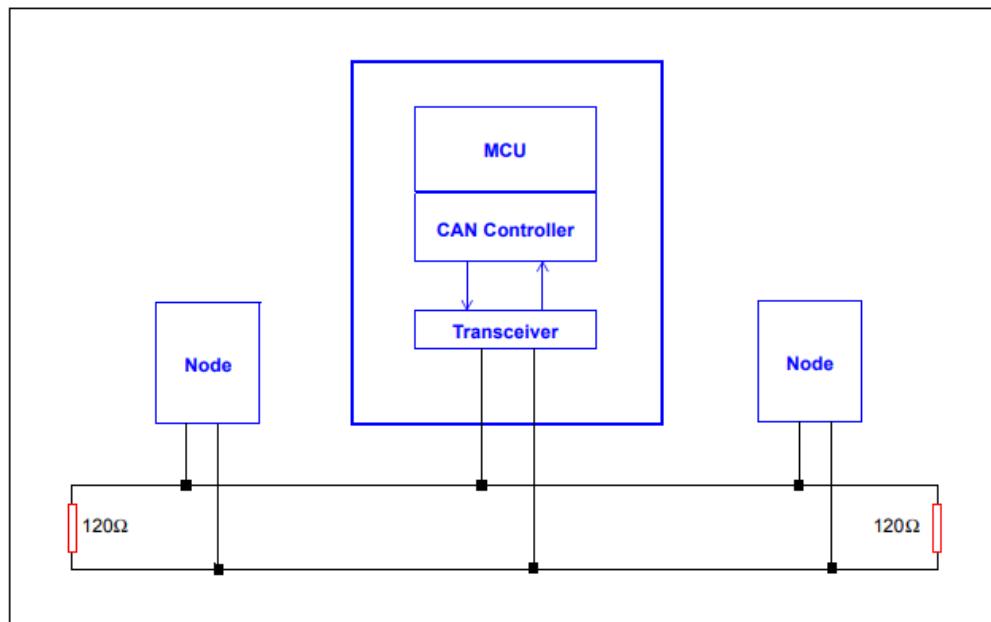


Abbildung 2.6: Can-Bus Connexion Richards (2002)[p. 2]

zwei CAN-Node gemäß der ISO-11898-1 CAN-Spezifikation. CAN High(CAN_H) und CAN Low(CAN_L) verlangen zwei 120Ω -Abschlusswiderstände. Der Transceiver wandelt die von CAN-Knoten kommenden CAN-Signale in ein digitales Rx- und Tx-Signal für den Node Controller um. Des Weiteren handelt es sich bei CAN_H und CAN_L um Differenzsignale. wie auf der Abbildung 2.7 zu sehen ist, wenn die zwei Signale bei 2,5 V liegen, ist dies ein rezessives Signal, also eine logische 0. Wenn CAN_H auf 3,5 V und CAN_L auf 1,5 V, dann handelt es sich um ein dominantes Signal, also eine logische 1.

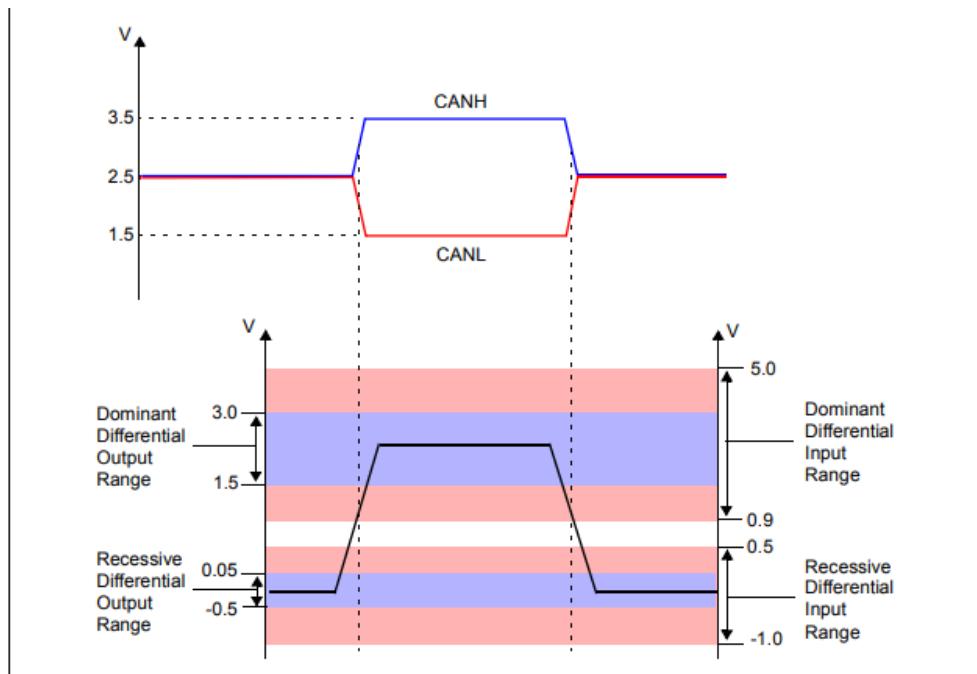


Abbildung 2.7: CAN_H und CAN_L Richards (2002)[p. 3]

2.3 SPI Interface

Die *Serial Peripheral Interface* (SPI) ist eine der am häufigsten verwendeten Schnittstellen zwischen Mikrocontrollern und Peripherie-ICs wie Sensoren, ADCs, DACs, Schieberegistern, SRAM und anderen. Die Schnittstelle SPI ist eine synchrone, auf Voll-Duplex basierte Master-Slave-Schnittstelle. Die Daten vom Master oder Slave werden mit der aufsteigenden oder abfallenden Taktflanke synchronisiert. Dabei können sowohl Master als auch Slave gleichzeitig Daten übertragen.

Das SPI arbeitet aber nach dem Single-Master-Prinzip. Das bedeutet, dass ein zentrales Gerät die gesamte Kommunikation mit den Slaves initiiert. Der Master sendet Daten auf der MOSI-Signalleitung und empfängt Daten auf der MISO-Signalleitung,

so dass der Busmaster gleichzeitig Daten senden und empfangen kann. wie auf dem Bild A der Abbildung 2.8 zu sehen ist. Alle Datenübertragungen müssen zwischen dem Bus-Master und den Slaves stattfinden. Datenübertragungen die direkt zwischen zwei Slave-Geräten stattfinden sind nicht erlaubt.

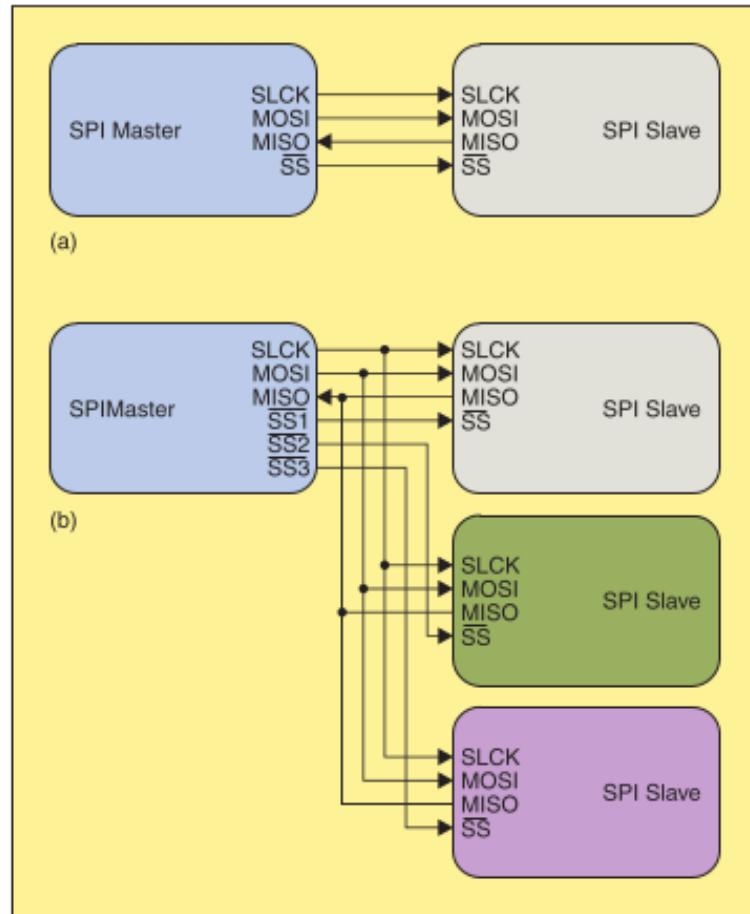


Abbildung 2.8: SPI Bus [Leens \(2009\)](#)[p. 9]

Möchte der SPI-Master Daten an einen Slave senden und/oder von ihm Informationen anfordern, dann wählt er einen Slave aus, und zwar durch Ziehen der entsprechenden SS-Leitung nach unten, während er das Taktsignal mit einer für den Master und den Slave nutzbaren Taktfrequenz aktiviert. SPI ist eine Protokoll mit vier Signalleitungen, wie auf [Leens \(2009\)](#)[p. 9] zu lesen ist.

- **Ein Clock-Signal (SCLK)**, welches vom Bus-Master an alle Slaves gesendet wird; alle SPI-Signale sind mit diesem Clock-Signal synchronisiert.
- **Der Slave Select Signal**: der zur Auswahl des Slaves dient, mit dem der Master kommuniziert.

- **Eine Datenleitung vom Master zu den Slaves**, bezeichnet als Master Out-Slave In (MOSI).
- **Eine Datenleitung von den Slaves zum Master**, bezeichnet als Master In-Slave Out (MISO).

Im kommenden Abschnitt möchte ich das Thema Embedded Systems ansprechen und die Vorteile solcher Systeme erläutern.

2.4 Embedded Linux für Xilinx ZynqMP Ultrascale+MPSoC

Bevor ich auf *Embedded Linux* eingehe, was eigentlich der englische Begriff von *eingebettetes System* ist, möchte ich zunächst klarstellen, dass es keine spezielle Version des Linux-Kernels für Embedded-Systeme gibt. Das Wort *Linux* in embedded Linux bezeichnet hier den Mainline-Linux-Kernel, der auf einem embedded System läuft. Aus Sprachmissbrauchsgründen wird es anstelle von "Linux auf einem eingebetteten System" verwendet.

Im Bereich der eingebetteten Softwareentwicklung wird entschieden, ob man auf Basis von Baremetal oder auf Basis eines Betriebssystems programmiert. Ein Betriebssystem bringt gegenüber der direkten Systemprogrammierung Vorteile mit sich, die es zu berücksichtigen gilt. Bare Metal heißt, dass ein Programm oder eine Software ohne Unterstützung eines Betriebssystems direkt auf der Hardwareebene ausgeführt wird. Anders ausgedrückt, programmiert man einen Mikrocontroller direkt mit ein paar Zeilen C- oder Assembler-Code. Bei embedded Linux im Gegenteil werden Anwendungen über dem Kernel ausgeführt oder von diesem unterstützt und arbeiten so als Betriebssystem (OS). Jede Kommunikation zwischen Hardware und Software läuft also über den Kernel, was tatsächlich viele Vorteile mit sich bringt.

- Treiber-Unterstützung für viele Geräte
- Prozess- und Speicherverwaltung
- Bestehende Anwendungen und Netzwerkprotokolle
- Skalierbarkeit und Echtzeitfähigkeit
- Große Entwickler-Community

Man spart nicht nur Zeit, sondern trägt auch zur Wartbarkeit der Software bei, wenn man vorhandene Software verwendet. Wenn man solche Komponenten von Null an entwickelt, dann hat man eine Quelle für eventuelle Fehler, die bei betriebssystembasierter Software wegen der hohen Verbreitung und Unterstützung durch die

Gemeinschaft und die Entwickler in der Regel minimiert werden. Außerdem haben Betriebssysteme den Vorteil, dass die Software leichter auf Nachfolgeplattformen und mithilfe von Standards wie POSIX auf andere Betriebssysteme übertragen werden kann.

Im Rahmen dieser Arbeit wird ein Linux-Kernel auf Basis der Kernel Version 5.10 verwendet [[Linux-kernel](#)], der um einige Zynq-spezifische Features in Form von Treibern erweitert wurde. Eine Liste der von Xilinx zur Verfügung gestellten Treiber ist im Official Xilinx Wiki zu finden. Eine Liste der von Xilinx bereitgestellten Treiber kann man

2.5 Komponente des Embedded Linux Betriebssystems für ARM Prozessoren

In diesem Abschnitt möchte ich auf die wesentlichen Komponenten von eingebetteten Linux-Betriebssystemen im Detail eingehen. Im Anschluss daran wird der typische Boot-Prozess solcher Systeme beschrieben. Ein auf Linux basierendes Betriebssystem wird in Form einer Linux-Distribution angeboten. Es handelt sich dabei um eine Sammlung von Softwarepaketen, Bibliotheken und Dienstprogrammen zusammen mit einer eigenen Linux-Kernel-Variante, die für eine bestimmte Prozessorarchitektur angepasst oder verändert wird. Zum Booten eines Linux-Betriebssystems auf einem ARM-Prozessor werden die folgenden Komponenten benötigt:

- **der Bootloader**
- **Der Gerätebaum(Device-tree):**
- **Der Linux Kernel**
- **Root filesystem:** beinhaltet die Bibliotheken und Programme, die ausgeführt werden, sobald der Kernel seine Initialisierung abgeschlossen hat.
- **Der “Init“ Prozess:**

2.5.1 Der Bootloader

Derviş (2013)Der Bootloader ist die Software, die beim Einschalten des Systems ausgeführt wird und für das Laden eines Betriebssystems für die Hardware verantwortlich ist. Beim Einschalten des Rechners, auf denen Linux als Betriebssystem

installiert ist, wird nach der ersten Einrichtung der Bootloader, der für die Initialisierung der Hardware-Peripherie und das Laden des Bitstreams im FPGA verantwortlich ist, in den Speicher geladen und der Code ausgeführt.

Der Bootloader muss zunächst von der Festplatte in den Prozessorspeicher geladen werden, bevor er ausgeführt werden kann. Beim Zynq UltraScale+MPSoC wird UBoot zum Laden des Linux-Kernels verwendet, und die Bootload-Phase ist hier in zwei Stufen unterteilt: die FSBL-Phase und die U-Boot-Phase

2.5.2 Device-tree

```
/*
 * dts file for Xilinx ZynqMP ZCU106
 *
 * (C) Copyright 2016 - 2020, Xilinx, Inc.
 *
 * Michal Simek <michal.simek@xilinx.com>
 */

#include "include/dt-bindings/input/input.h"
#include "include/dt-bindings/gpio/gpio.h"
#include "include/dt-bindings/pinctrl/pinctrl-zynqmp.h"
#include "include/dt-bindings/phy/phy.h"

{
    model = "ZynqMP_ZCU106_RevA";
    compatible = "xlnx,zynqmp-zcu106-revA", "xlnx,zynqmp-zcu106", "xlnx,zynqmp";

    gpio-keys {
        compatible = "gpio-keys";
        autorepeat;
        sw19 {
            label = "sw19";
            gpios = <@gpio 22 GPIO_ACTIVE_HIGH>;
            linux,code = <KEY_DOWN>; /* down */
            wakeup-source;
            autorepeat;
        };
    };

    leds {
        compatible = "gpio-leds";
        heartbeat-led {
            label = "heartbeat";
            gpios = <@gpio 23 GPIO_ACTIVE_HIGH>;
            linux,default-trigger = "heartbeat";
        };
    };
}
```

Abbildung 2.9: Ein Screenshot des ZCU106-Gerätebaums, der den Inhalt des gpio-keys und leds auf dem Board zeigt

Die Abbildung 2.9 zeigt Informationen zu den Schalter- und led-Knoten im Device Tree des Xilinx ZCU106 Boards. Hier wird z.B. angegeben, an welchem gpio-Pin der Schalter SW9 auf der Platine angeschlossen ist.

Derviş (2013) Der Linux-Kernel benötigt Informationen über den Prozessor, auf dem er ausgeführt wird, die Peripheriegeräte, mit denen der Prozessor verbunden ist, ihre Schnittstellen zum Prozessor und ihre physikalischen Adressen. Der Kernel muss zur Initialisierung der Treiber und der mit diesen Peripheriegeräten verbundenen

Dienste auch überprüfen, dass die Funktionen, die in seiner Konfiguration aktiviert wurden, tatsächlich von der Hardware unterstützt werden, die er steuert. Dabei kann es sich um Informationen über die Taktgeber und Register der Hardware oder über die mit der Hardware verbundenen Peripheriegeräte wie den externen Speicher, SPI handeln.

Der Zynq UltraScale+MPSoC verwendet daher den Device-Tree, um die Geräte- und Peripherie-Informationen wie physikalische Geräteadressen, E/A-Registeradressen, Speicheradressraum und Interrupt-Informationen während des Bootvorgangs an den Kernel weiterzugeben.

Der Gerätebaum wird im Textformat in einer Datei mit der Erweiterung ".dts" dargestellt. Hierbei handelt es sich um eine Quelltextdatei, die Informationen über Geräte und Verbindungsbusse beschreibt, die mit einer Computer-Hardware verbunden sind. Sie ist in Form von "Knoten" organisiert, deren Stammverzeichnis durch "/" dargestellt wird, genau wie im Linux-Stammdatesystem. Jeder Knoten hat einen Namen, der ein mit dem Prozessor verbundenes Gerät oder einen Bus darstellt, und der Knoten besteht aus Eigenschaften. Jeder übergeordnete Knoten für ein bestimmtes Peripheriegerät oder einen Bus kann "Kind"-Knoten für Geräte enthalten, die mit diesem Peripheriegerät oder Bus verbunden sind. Die Werte der Eigenschaften können Zeichenketten oder Listen von Zeichenketten sein, oder sie können leer sein, wenn das Vorhandensein oder Nichtvorhandensein des Wertes eine boolesche Logik an den Kernel übermittelt. Die Device-Tree-Quelldatei wird mit Hilfe des "dtc -Compilers" zu einem Device-Tree-Blob (.dtb) kompiliert.

2.5.3 Der Linux Kernel

Das ist das Herzstück des Systems, das die Systemressourcen und Schnittstelle zur Hardware verwaltet. Die Hauptfunktionen des Linux-Kernels sind die folgenden [Daniel P. Bovet and Marco Cesati \(2006\)](#):

- Planung von Prozessen und Einrichten einer Umgebung für ihre Ausführung.
- Zuteilung von Speicher an einen Prozess und Schutz des von einem Prozess verwendeten Speichers vor anderen Prozessen.
- Verwaltung der Kommunikation zwischen den Prozessen, um eine effiziente Ausführung der Prozesse zu gewährleisten
- Sicherstellung der Integrität des Systems, wenn das Computersystem mehrere Benutzer hat, die berechtigt sind, Änderungen am Root-Dateisystem und an Softwarepaketen vorzunehmen.

- Verwalten der Computerressourcen und des Zugriffs auf diese Ressourcen

Nachdem der Bootloader den Gerätebaum und den Kernel in den DRAM des Prozessors geladen hat, teilt er dem Kernel die Adresse des Gerätebaums mit, bevor der Kernel mit der Ausführung beginnt. Der Kernel prüft dann der Reihe nach alle Hardware-Peripheriegeräte, Clock-Generator und Speicher, die vom FSBL initialisiert wurden, und aktiviert dann alle mit der zugrunde liegenden Hardware verbundenen Dienste und Funktionen, die in der Kernelkonfiguration aktiviert wurden. Sobald der Kernel mit diesen Aufgaben fertig ist, hängt er das Root-Dateisystem ein und führt den `init`Prozess aus, der es dem Benutzer ermöglicht, in den Benutzerraum einzutreten.

2.5.4 Das Linux Root files System (Rootfs)

Das Linux-Root-Dateisystem [Derviş (2013)] enthält alle binären ausführbaren Dateien, Geräteinformationen, Prozessprotokolle, Softwarepakete und Bibliotheken, die vom Benutzer in seinem Benutzerbereich benötigt werden, um das Linux-Betriebssystem effizient zu nutzen. Das Dateisystem auf der Festplatte ist in Verzeichnissen organisiert. Das Root-Dateisystem wird in das Verzeichnis“/“ des Dateisystems eingehängt, welches die Spitze der Hierarchie des Root-Dateisystems markiert.

2.5.5 Der Init Prozess

Der `init`Prozess [Derviş (2013)] wird als erster Prozess im Benutzerbereich vom Kernel initiiert und ist für die Initialisierung der Systemverwaltungsdienste zuständig, bevor die Benutzer sich anmelden können. Alle Software-Dienste, die im Root-Dateisystem installiert und im Benutzerraum aktiviert wurden, werden vom `init`Prozess ausgeführt, bevor sich die Benutzer sich am System anmelden

Als Nächstes wird den Boot-Prozess bei Systemen, die auf Zynq UltraScale+ MP-SoCs basiert sind, erläutert, da dies genau die Plattform ist, die wir für unsere Arbeit verwenden werden.

2.5.6 Der Zynq UltraScale+MPSoC Boot-Prozess

Zum Booten von Linux auf dem Zynq UltraScale+MPSoC muss das Boot-Image (BOOT.BIN) auf dem Boot-Medium entsprechend dem vom Benutzer gewählten Boot-Modus vorhanden sein [UG1137 (2017)[p. 14]]. Das ZCU106 Board unterstützt das Booten über JTAG, Quad-SPI Flash, SD Card und NAND Flash Drive. In dieser Arbeit booten wir Linux von der SD-Karte[Petalinux (2020)[p. 61]], daher muss

die Datei BOOT.BIN auf der FAT32-Partition der SD-Karte vorhanden sein. Die BOOT.BIN-Datei wurde so konfiguriert, dass sie die Platform Management Unit Firmware (PMUFW), die FSBL und die ausführbaren U-Boot-Dateien beinhaltet, welche die empfohlene Konfiguration für den Zynq UltraScale+ ist. Die BOOT.BIN enthält in dieser Arbeit auch den FPGA-Bitstream zur Programmierung der programmierbaren Logik (PL). Für ein vollständiges Booten über eine SD-Karte sollten das Kernel-Image (Image) und der Device-Tree-Blob ("Projektnname.dtb") ebenfalls auf der FAT32-Partition vorhanden sein. Das Root-Dateisystem muss auf der EXT4-Partition der SD-Karte vorhanden sein.

Der Bootvorgang von Linux auf dem Xilinx Zynq UltraScale+MPSoC lässt sich in vier Phasen unterteilen

- Die Boot-Setup-Phase
- Die Bootloader-Phase
- Die Kernel-Boot-Phase
- und Die “init“-Phase

2.5.6.1 Die Boot-Setup-Phase

Die Platform Management Unit (PMU) und die Configuration Security Unit (CSU) sind für das Einrichten des Zynq UltraScale+ MPSoC verantwortlich, bevor Linux auf dem PS gebootet werden kann. Die Boot-Setup-Phase besteht aus drei Phasen, die wie folgt unterteilt werden können[[Xilinx Inc. \(2019\)](#)[p. 27]]:

- **Pre-configuration stage (Oder Vor-Konfigurationsphase):** Dieser Phase wird von der PMU angesteuert, die den PMU-ROM-Code(PBR). Nachdem der PBR-Code ausgeführt wurde, übergibt er die Systemkontrolle an die Configuration Security Unit (CSU). Die wichtigsten Schritte der Vor-Konfigurationsphase sind im Folgenden aufgeführt[[Xilinx Inc. \(2019\)](#)[p. 57]]:
 - Initialisieren des Systemmonitors
 - Initialisierung der Phase Locked Loops (PLL) für Takte
 - Leeren des PMU-RAMs.
 - Initialisieren Sie des Dynamic Random Access Memory (DRAM).
 - Freigabe der CSU oder Eintritt in den Fehlerzustand.

- **Configuration stage oder (Konfiguration der Phase):** Diese Phase übernimmt das Laden des First-Stage-Bootloader-Codes (FSBL) für den PS in das On-Chip-RAM (OCM), und zwar sowohl im sicheren als auch im unsicheren Boot-Modus. Während des Bootvorgangs lädt die CSU auch die PMU-Benutzerfirmware (PMU FW) in das PMU-RAM, um in Verbindung mit dem PMU-ROM Plattform-Management-Dienste bereitzustellen.
- **Post-configuration stage oder (Post-Konfigurationsphase):** Nach dem Start der FSBL-Ausführung geht der CSU-ROM-Code in die Post-Konfigurationsphase über, die für die Reaktion auf Systemmanipulationen verantwortlich ist

2.5.6.2 Die Bootloader-Phase

First Stage Bootloader (FSBL)

Die FSBL ist der sekundäre Programmable in der Terminologie des generischen ARM-Prozessor-Boot-Prozesses. Der FSBL führt die folgenden Aufgaben aus. FSBL führt die folgenden Aufgaben aus [Xilinx Inc. (2019)]

- Suchen in der FAT32-Bootpartition der SD-Karte nach dem PL-Bitstream und dem second stage Bootloader (das U-boot).
- Initialisierung der PS-Hardware, der E/A-Geräte, des Speichers und der Takte entsprechend der Konfiguration, die durch das in der Xilinx Vivado Design Suite spezifizierte Hardware-Design definiert ist
- Programmieren des PL mit dem FPGA-Bitstream
- Laden der ARM Trusted Firmware (ATF) und das U-Boot in den APU-Prozessorspeicher.

Das U-boot

Wenn der Prozessor eingeschaltet wird, enthält der Prozessorspeicher kein Betriebssystem, sodass ein Bootloader erforderlich ist, um den Linux-Kernel und den Gerätebaum aus dem Speicher in den Prozessorspeicher zu laden, diese Aufgabe wird also vom U-boot übernommen. Der U-Boot ist der Second Stage Bootloader (SSBL) für den Xilinx Zynq UltraScale+ MPSoC oder der Tertiary Program Loader (TPL) in der Terminologie des generischen ARM-Bootprozesses. U-Boot kann den Kernel und den Device-Tree über das Netzwerk mittels Ethernet, über JTAG, von der SD-Karte, Quad-SPI-Flash und vom NAND-Flash-Laufwerk laden.

Die Abbildung 2.10 Es zeigt die Rolle der verschiedenen Einheiten des Zynq UltraScale+MPSoC beim Booten des Linux-Betriebssystems auf der Hardware. Man sieht, dass die PMU die CSU freigibt, die wiederum die FSBL lädt. Die FSBL lädt

dann das ATF und das U-Boot. Das U-Boot ist dann für das Booten von Linux auf der Hardware verantwortlich.

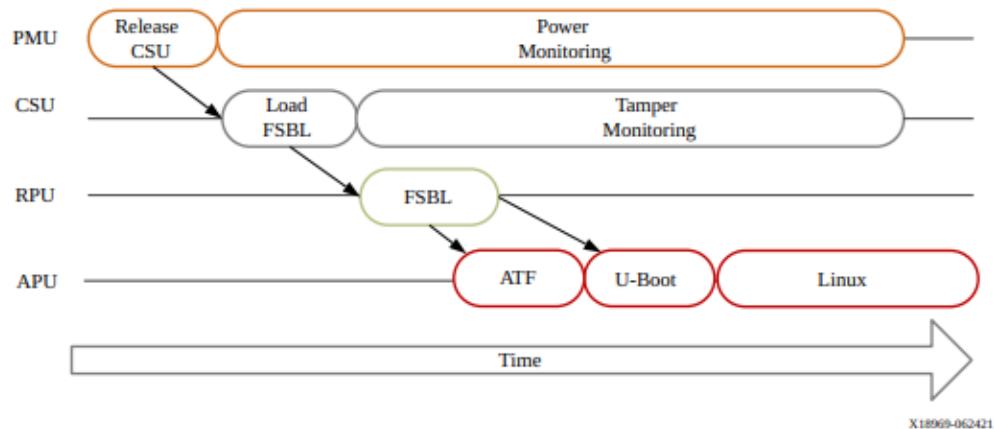


Abbildung 2.10: Überblick über den Bootvorgang [Xilinx Inc. \(2019\)](#)[p. 58]

2.5.6.3 Der Boot Flow

Der Zynq UltraScale+ MPSoC kann in zwei Modi gebootet werden: im sicheren und im unsicheren Modus. Diese werden im Folgenden beschrieben [\[Xilinx Inc. \(2019\)\]](#)[p. 58]

Nicht gesicherter Bootvorgang

In diesem Boot-Modus gibt die PMU die CSU frei und wechselt in den Servicemodus, in dem sie die Plattform überwacht. Die CSU lädt die FSBL in den On-Chip-Speicher (OCM) der APU und die PMUFW in das RAM der PMU. Die PMUFW wird parallel zur Ausführung der FSBL ausgeführt und läuft, bis Linux auf dem Zynq UltraScale+ MPSoC gebootet ist. Der FSBL initialisiert die Peripheriegeräte, E/A-Geräte, den Speicher und die Takte und übergibt sie an den ATF, der sie dann an das U-Boot weitergibt. Das U-Boot lädt dann den Linux-Kernel in das APU-Prozessor-Memo

gesicherter Bootvorgang

Der sichere Bootvorgang unterscheidet sich vom unsicheren Bootvorgang nur durch einige zusätzliche Authentifizierungs- und Entschlüsselungsschritte, die von der CSU ausgeführt werden. Beim sicheren Bootvorgang gibt die PMU den Reset der Configuration Security Unit (CSU) frei und geht in den PMU-Servermodus über, in dem sie die Stromversorgung überwacht. Nachdem die PMU das Zurücksetzen der CSU aufgehoben hat, prüft die CSU, ob eine Authentifizierung durch die FSBL oder die Benutzeranwendung erforderlich ist

2.5.7 Die Kernel-Boot-Phase

Das U-Boot sorgt für das Laden des Kernel-Images und des Gerätebaums in den Speicher des APU-Prozessors und die Übergabe der Kontrolle an den Kernel. Der Kernel erhält auch die Adresse des Gerätebaums, der den Kernel über die Hardware-Peripherie, E/A-Geräte, Speicher und Taktgeber informiert. Basierend auf diesen Informationen startet der Kernel den Bootvorgang und aktiviert die Treiber und Dienste, für die der Kernel konfiguriert wurde. Nachdem der Kernel gebootet hat, mountet er das Root-Dateisystem entsprechend den an den Kernel übergebenen Boot-Argumenten. Je nach dem an den Kernel übergebenen Boot-Argument hängt der Kernel das Root-Dateisystem über NFS oder von der SD-Karte oder einem anderen permanenten Speicherort ein.

2.5.8 Die init-Phase

Nach Einhängen des Root-Dateisystems sucht der Kernel nach der ausführbaren Datei des für die Linux-Distribution angegebenen init-Prozesses. Sobald der Kernel den init-Prozess gefunden hat, führt er ihn aus und übergibt die Kontrolle an den init-Prozess. Der init-Prozess aktiviert dann alle Systemverwaltungs- und andere Hintergrunddienste, die im Root-Dateisystem installiert und im Benutzerbereich aktiviert wurden. Nach der Aktivierung dieser Dienste startet der init-Prozess den Benutzeranmeldungsprozess, der es dem Benutzer ermöglicht, den Benutzerraum zu betreten

Alle bisher genannten Linux-Komponenten, ob der Bootloader, der Kernel oder das Root-Dateisystem können mit einem Build-System wie dem Yocto-Projekt erstellt werden. Da unser FPGA aber einen MPsoc enthält, der einen Bootloader, ATF-Firmware, pmufw, den Bitstream und u-boot benötigt, ist ein Build System zu verwenden, das automatisch alle diese Komponenten erzeugen kann.. Hierfür ist Petalinux am besten geeignet. Im nächsten Abschnitt werde ich das Petalinux Build System vorstellen.

2.6 Petalinux Tool Flow

PetaLinux ist ein Embedded Linux Software Development Kit (SDK), das auf FPGA-basierte System-on-a-Chip (SoC)-Designs abzielt [[Petalinux \(2020\)](#)]. Es setzt sich praktisch auf Yocto auf, so wird das Root-Dateisystem unter Verwendung von Yocto erstellt. Unter PetaLinux versteht man eine Reihe von High-Level-Befehlen, die auf der Yocto-Linux-Distribution aufbauen. Die PetaLinux-Werkzeuge können zur

Anpassung, Erstellung und Bereitstellung von Embedded Linux-Lösungen/Linux-Images für Xilinx-Prozessorsysteme verwendet werden. Ein wesentlicher Vorteil von petalinux ist, dass es eine Reihe von vereinfachten Befehlen enthält, die für das Booten und die Integration von HW- und SW-Projekten sehr nützlich sind. In Abbildung 2.11 sieht man einen Überblick über den PetaLinux-Werkzeugfluss auf oberster Ebene.

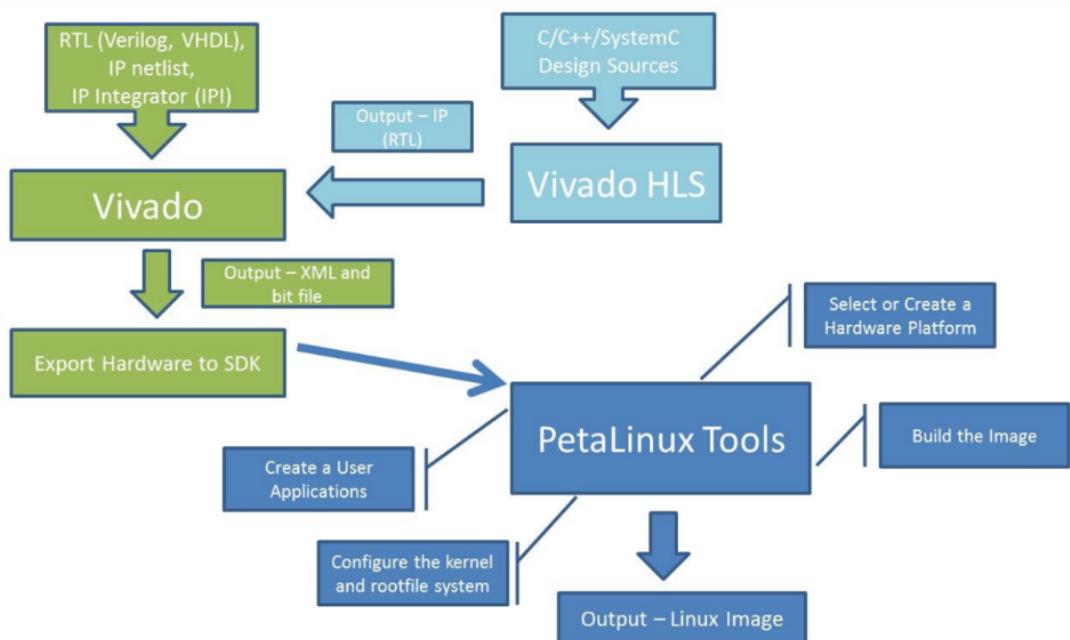


Abbildung 2.11: Überblick über den PetaLinux-Werkzeugfluss [Support (2022)]

Wie man in der Abbildung 2.11 sehen kann, ist es möglich, mit Vivado erstellte Hardware-Designs in petalinux zu importieren, einige Anwendungen in petalinux einzubinden und ein Linux-Image zu erstellen.

2.6.1 Petalinux Installation

Wie jedes Build-System benötigt petalinux viele Ressourcen auf dem PC. Um die Komplizierzeit deutlich zu reduzieren, und Zeit zu sparen, ist es daher sinnvoll, einen Computer mit folgenden Eigenschaften zu verwenden. [Petalinux \(2020\)](#)

- 8 GB RAM
- 2 GHz CPU-Takt oder gleichwertig
- 100 GB freier HDD-Platz
- Petalinux unterstützt nur auf Linux Kernel basierte Betriebssysteme.

- Petalinux-Tools erfordern, dass das Host-System /bin/sh **bash** ist. Der folgende Befehl kann verwendet werden, um die Bash als Terminal einzurichten, wobei der Befehl als root ausgeführt werden muss.

```
1 $ sudo dpkg-reconfigure dash
```

Bevor das Installationsprogramm ausgeführt wird, müssen eine Reihe von Abhängigkeiten installiert werden. Die folgenden Befehle installieren alle erforderlichen Pakete auf einmal:

```
1 $ sudo apt upgrade
2 $ sudo apt install -y python3 tofrodos iproute2 gawk xvfb gcc git
   make net-tools libncurses5-dev tftpd zlib1g-dev ibssl-dev flex
   bison libselinux1 gnupg wget diffstat chrpath socat xterm
   autoconf libtool tar unzip texinfo zlib1g-dev gcc-multilib build
   -essential libsdl1.2-dev libglib2.0-dev screen pax gzip
```

Einmal die vorherigen Voraussetzungen erfüllt, kann man also die Installationsdatei von Petalinux unter diesem Link <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html> herunterladen.

mit dem **mkdir** Kommando in Linux kann man ein Petalinux Installation Ornder erstellen, in dem die Installationsdatei dann kopiert wird. mit dem -p Schalter kann den Ordner in einem spezifischen Ordner erstellen.

```
1 $ mkdir -p /home/<user>/petalinux/<petalinux-version>
```

Mit den folgenden Befehlen kann man die Datei ausführbar machen und der Instalationsprozess starten.

```
1 $ chmod 755 ./petalinux-v<petalinux-version>-final-installer.run
2 $ ./petalinux-v<petalinux-version>-final-installer.run
```

2.6.2 Wichtige Petalinux Kommando

- **petalinux-create**: Erstellt ein neue Petalinux Projekt. man kann dem Befehl verschiedenen Optionen zuweisen,
 - **type**: definiert den Projekt Type
 - **template**: Bei der Erstellung des Projekts kann man eine Vorlage definieren. Für das Projekt wurde zynqMP verwendet.

- ***srcuri***: Hier wird der Pfad zu einem Board Support Package (BSP) angegeben, das zur Erstellung des Projekts verwendet wird.
- ***name***: definiert den Name des Projekts.
- **petalinux-config**: dieser Befehl wird verwendet zur Initialisierung oder Aktualisierung der Hardwarekonfiguration des Projekts oder Konfiguration der Kernel- und/oder Dateisystemeinstellungen. Je nach Anwendung stehen hier auch uns eine Reihe von Konfiguration-Optionen zur Verfügung. Einige davon sind:
 - ***get-hw-description***: Initialisiert den Petalinux-Projekt mit einem vom Vivado Hardware-description-file(HDF). PetaLinux verwendet HSI-Dienstprogramme, um Informationen über die Hardware aus dieser Datei zu extrahieren, sowie Informationen wie Intellectual property Cores (IP-Cores), Netze, Ports und Schnittstellen, die in anderen Tools wie dem Devicetree-Generator verwendet werden .
 - ***-c rootfs***: Startet das Konfigurations-Menü des Root-Dateisystems.
 - ***-c kernel***: Startet das Konfigurations-Menü des Kernel.
- **petalinux-build**: Das Tool Erstellt bestimmter Komponenten oder eines ganzen Linux-Systems für das PetaLinux-Projekt (einschließlich FSBL, uboot, Gerätebaum usw.). Genau so wie mit ***petalinux-config*** Befehl, können auch Besonderheiten mit den Zeichen ***-c*** und ***-x*** festgelegt werden.
 - ***-c oder -component***: Baut die angegebene Komponente(kernel, u-boot, rootfs, device-tree ...). Es handelt sich hierbei um die Standard Komponente, die unterstützt werden. Es können aber auch eigenes Objekt erstellen werden (z. B. eigene Anwendung oder Modul).
 - ***-x oder execute*** : Führt den angegebenen Build-Schritt aus. Es können alle Yocto-Tasks über diese Option übergeben werden(build, clean, cleansstate, distclean ...).
- **petalinux-boot**: Das Werkzeug Bootet ein angegebenes Linux-Image entweder über JTAG auf die Hardware oder den QEMU-Softwareemulator.
 - ***-jtag***: Die jtag-Tools sind sehr hilfreich, wenn man genau sehen möchte, wie der Boot-Vorgang im Einzelnen abläuft.

- **petalinux-package:** Das Werkzeug packt ein gebauter PetaLinux-Projekt in einem für die Bereitstellung geeigneten Format. je nach Zielpaketformat bietet es mehrere Arbeitsabläufe, deren Operationen abweichen. Für das Projekt verwenden wir ***petalinux-package -boot***, der hat die folgenden Optionen:
 - **-format:** Das zu erzeugendes Bilddateiformat(BIN, MCS, DOWNLOAD.BIT)
 - **-fsbl:** Damit definiert man den Pfad zum First Stage Bootloader(FSBL) .elf-Binäre Datei.
 - **-fpga BITSTREAM:** Den Pfad zur Bitstream-Datei.
 - **-force:** Existierende Dateien auf der Festplatte überschreiben.
- **petalinux-devtool:** Das petalinux-devtool ist das letzte auf der Liste der petalinux-Tools, die ich beschreiben wollte und die ich für meine Arbeit benötigen werde. Das ist ein Dienstprogramm, das mit Hilfe des Yocto-Devtools Software erstellt, getestet und verpackt werden können. In den kommenden Abschnitten werde ich auf jeweilige Optionen, die ich verwendet habe eingehen.

2.6.3 Petalinux Projekt Strukture

In diesem Abschnitt möchte ich über die Petalinux-Projektstruktur sprechen. Es ist wichtig, dies zu erwähnen, damit klar ist, wie und wo Komponenten, Module oder Software geändert werden können.

```
project-spec
    hw-description
    configs
    meta-user
pre-built
    linux
        implementation
        images
        xen
hardware
    <project-name>
components
    plnx_workspace
    device-tree
config.project
README
```

Abbildung 2.12: typische petalinux projektstruktur [Support (2022)]

In Abbildung 2.12 ist eine typische Petalinux Projektstruktur dargestellt.

- **project-spec:** In diesem Verzeichnis werden alle Änderungen an dem Projekt durchgeführt. Hier können z. B. neue Projekt Layers erstellen, den Gerätebaum(Device-tree) geändert oder sogar Rezepte für Software, die vom Kernel kompiliert werden soll, erstellt werden.
- **pre-built:** Dieses Verzeichnis beinhaltet alle Board-spezifischen Design- und Konfigurationsdateien, vorgefertigte und getestete Hardware und Software-Images, die Sie auf dem Board direkt heruntergeladen werden können. Der Ordner ist jedoch nur sichtbar, wenn man das Projekt auf der Basis des für das Board spezifischen Board Support Package (BSP) erstellt hat.
- **hardware:**

3 Umsetzung

In diesem Kapitel möchte ich mich am allerersten noch mal über das Ziel dieser Arbeit äußern, dabei geben ich eine tiefe Eindruck auf meine tatsächliche Arbeit. Dann möchte ich die verwendete Hardware, folgen mit einer grobe Beschreibung, wie sie Verschalten sind, beschreiben. Anschließend möchte ich auf die Schritte zum Bauen und Konfigurieren des Gesamt Betriebssystem eingehen.

3.1 Allgemein über das Projekt

Ähnlich wie ein Desktop-PC, der ein Betriebssystem wie Windows oder Linux benötigt, um seine gesamte Software auszuführen oder die angeschlossene Hardware zu steuern, benötigt embedded Geräte auch ein Betriebssystem, um ihre Anwendungen einfacher zu verwalten. Das Betriebssystem, das auf unserem Gerät laufen wird, soll also die folgenden 8 Softwaremodule im Hintergrund ausführen, die dann sämtliche Funktionen der IPU-NG Projekt beschreiben. Wie im Kapitel 1.2 beschrieben, sollen im Rahmen dieser Arbeit 3 der 8 unten beschriebenen Softwaremodule in das Betriebssystem eingebaut werden.

- **System Watchdog:** der zuständig ist das System zu überwachen.
- **Power Manager:** der bedient die Stromversorgung der einzelnen Teile des Systems.
- **System Updater:** der aktualisiert das gesamte System, in falls, dass es Neuerung gibt.
- **License Manager:**
- **Web Backend:** der dient als Brücke zwischen der Web Anwendung und dem Rest des Systems
- **Web Anwendung / WEB Frontend:** Die Webanwendung ermöglicht es dem Benutzer, den Status des Geräts abzurufen, es zu aktualisieren, die Debug-Protokolldateien herunterzuladen

- **System Logger:** die von Linux erzeugten Log Files verfolgen, parsen und entsprechend der Liste von oben die notwendigen Informationen von den entsprechenden Log Files extrahieren und in den von der System Logger Anwendung verwalteten zirkularen Buffer schreiben.
- **Hauptanwendung:** die stellt die Kern-Funktionalität des IPU NG Gerätes zur Verfügung

Weiterhin sollte im Rahmen dieser Arbeit das Betriebssystem so konfiguriert werden, dass der MCP251xFD CAN-Controller über SPI sowohl konfiguriert werden als auch Daten an den Mainline-Linux-Kernel senden kann.

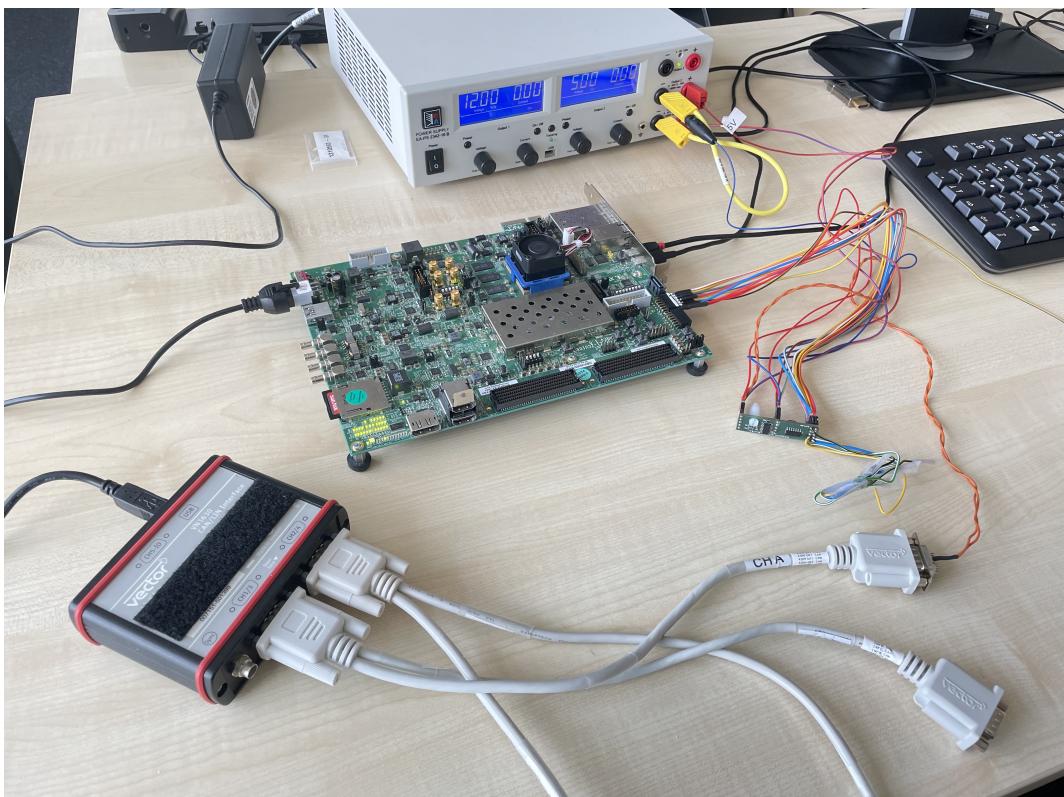


Abbildung 3.1: Versuchsaufbau ZynqMP Ultralcale + mcp251xfd

Die Abbildung 3.1 zeigt die Verschaltung zwischen der Ersatz Board und der MCP251xfd CAN Controller und der Ersatzt Board Zynq Ultrascale. Im folgenden Abschnitt würde ich eine ausführliche Beschreibung der verwendeten Hardware vorstellen.

3.2 Hardware Platform

Aus Kostengründen wurde anstelle des FPGA-internen CAN-Controllers ein externer CAN-Controller verwendet. Und viel besser: Dieser CAN-Controller wurde im Unternehmen entwickelt. Außerdem wurde das von Xilinx entwickelte Board ZCU106 verwendet, das auf dem Zynq UltraScale + MPSoC basiert. Das enthaltene ZU7EV-Gerät (Zynq UltraScale + MPSoC) integriert ein Quad-Core Arm Cortex-A53 Processing System (PS) und einen Dual-Core Arm Cortex-R5F Echtzeit-Prozessor. Das Board verfügt auch über viele programmierbare digitale Komponenten, mit denen wiederum eine Vielzahl von Schaltungen realisiert werden können. Wie bereits erwähnt, ist das FPGA Bestandteil des ZCU106 Boards, das im nächsten Kapitel besprochen wird.

3.2.1 Xilinx ZCU106 Evaluation Board

Dieses Kapitel beschreibt die Xilinx ZCU106 Evaluation Board, insbesondere die Bereiche, die diese Plattform einzigartig machen. Dazu gehören die Zynq UltraScale + MPSoC Architektur, das Processing System (PS) und die Programmable Logik (PL). Die Peripheriegeräte, die für die Interaktion mit dem CAN-Controller erforderlich sind, werden dabei erwähnt. Genauere Informationen dazu findet man im technischen Referenzhandbuch [Xilinx Inc. (2019)] des Zynq-Plattforms.

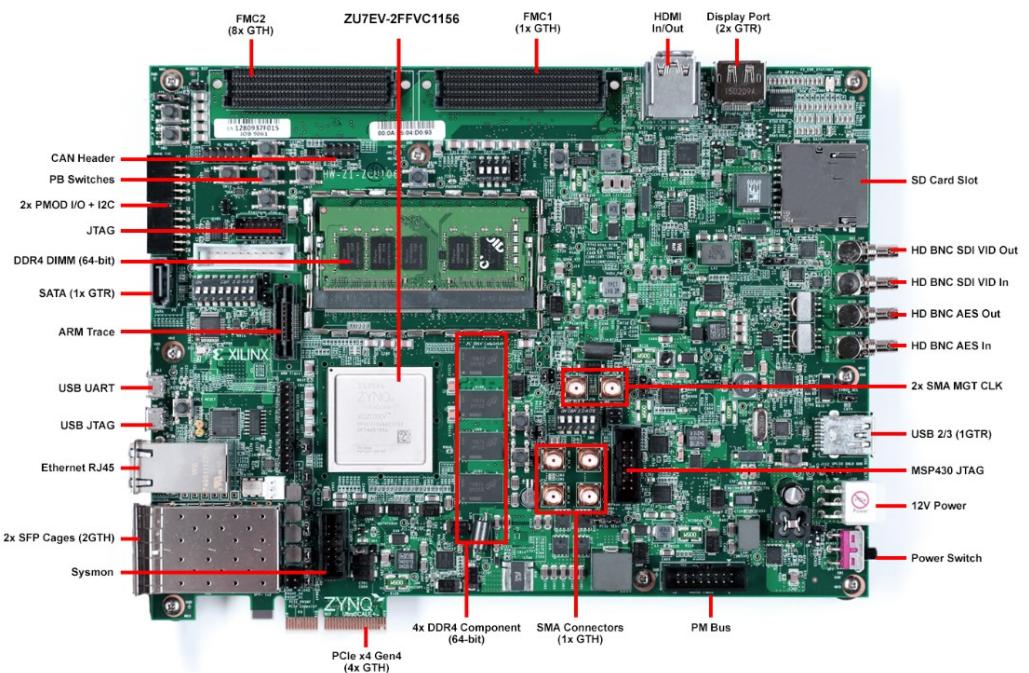


Abbildung 3.2: Xilinx ZCU102 Evaluation Board [Xilinx Inc. (a)]

3.2.1.1 Ultrasale + MPSoC Architektur

Lassen wir uns, bevor wir uns mit der Zynq Ultrascale+ MPSoC-Architektur beschäftigen, ein wenig über die Zynq-Architektur im Allgemeinen sprechen. Es handelt sich bei Zynq um eine neue Generation von System-on-Chip (SoC), die eine CPU mit einem programmierbaren Logik-FPGA auf demselben Chip kombiniert. FPGAs sind für ihre besondere Flexibilität beim Entwurf digitaler Schaltungen bekannt. Dennoch ist für viele Anwendungen der Entwurf einer riesigen Zustandsmaschine in Very High Speed Hardware Description Language (VHDL) oder Verilog nicht ausreichend. Stattdessen ziehen wir eine softwareprogrammierbare CPU-Architektur vor, die mit einfacheren FPGA-Blöcken zusammenarbeitet. Im Grunde genommen kann eine CPU jeden Algorithmus ausführen, so lange genug Speicher für ihre Bedürfnisse vorhanden ist. Softwarecode kann schnell geändert, neu kompiliert, gepatcht und debuggt werden. Die Rechenkapazität eines CPU-Kerns allein reicht jedoch oft nicht aus, um eine große Datenmenge zu verarbeiten. weshalb tendieren wir zu parallelen Architekturen, wie Multicore-CPUs, FPGAs und GPUs, um den Rechendurchsatz zu erhöhen.

Bislang gab es für jemanden, der eine CPU in Kombination mit einem FPGA benötigte, zwei Möglichkeiten: eine diskrete CPU und ein diskretes FPGA, die über einen Bus miteinander kommunizieren (was zu Bandbreitenbeschränkungen führte), oder eine Soft-Core-CPU. Diese wird direkt in den FPGA programmiert (z.B. 32-Bit Microblaze, 64-Bit Ultrascale, usw.). Die Entscheidung hängt von den Einschränkungen und Anforderungen der Anwendung ab, wie z. B. dem Systempreis, dem Stromverbrauch, der Komplexität und selbstverständlich der Leistung. Xilinx bietet mit der Zynq-Serie ein höheres Maß an Integration mit einer System-on-Chip-Hardcore-ARM-CPU, einem Xilinx-FPGA und Bussen für den effizienten Datentransfer zwischen beiden. Darüber hinaus umfassen die Zynq-Bausteine verschiedene Arten von Input/Output (I/O)-Controllern, Speicherschnittstellen und Hochgeschwindigkeits-Transceivern

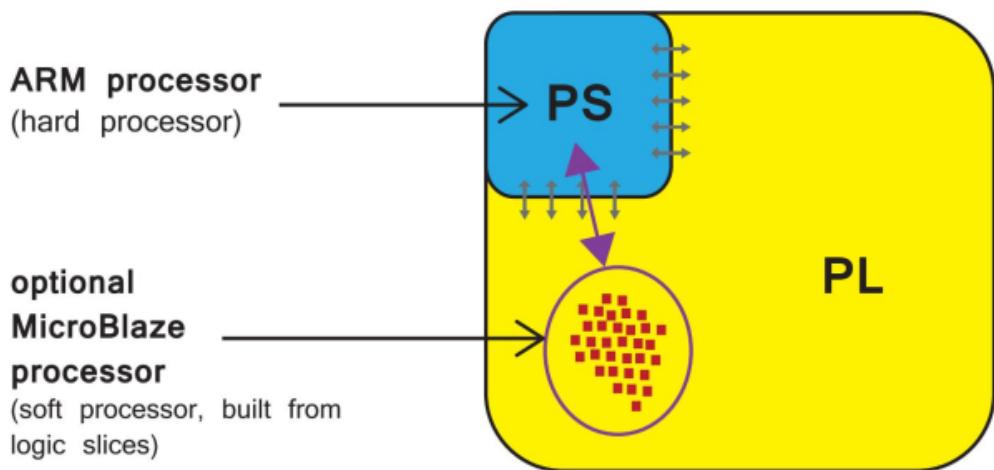


Abbildung 3.3: Die Anordnung von Hard- (ARM Cortex-A9/Cortex-A53) und Soft-Prozessoren (MicroBlaze) auf einem Zynq/ZynqMP-Baustein [Crockett, Louise H and Elliot, Ross A and Enderwitz, Martin A and Stewart (2014)]

Die Abbildung 3.3 zeigt die Trennung zwischen dem festen Logik-Hardware-Prozessor und der programmierbaren Logik, die einen oder mehrere Soft-Prozessoren enthalten kann.

3.2.1.2 Allgemeine Ansicht des Zynq Ultrascale+ MPSoC

Die Zynq Ultrascale+ MPSoC-Serie ist eine im Jahr 2015 von Xilinx eingeführte moderne SoC-Architektur [CNX Software]. Abbildung 3.4 zeigt das Blockdiagramm des Zynq Ultrascale+ EG, der verfügt über mehrere Verarbeitungseinheiten wie die ARM Cortex A53 Application Processing Unit (APU) mit 4 Kernen, die Real-Time Processing Unit (RPU) und die Platform Management Unit (PMU). Die Abbildung zeigt die Schnittstelle zwischen dem Verarbeitungssystem (PS) und der programmierbaren Logik (PL). Die PL verfügt über mehrere Blöcke wie GPIO, Block-RAM und High-Connectivity-Block für die Implementierung von Designs zur Kommunikation mit Peripheriegeräten wie Ethernet oder SPI.

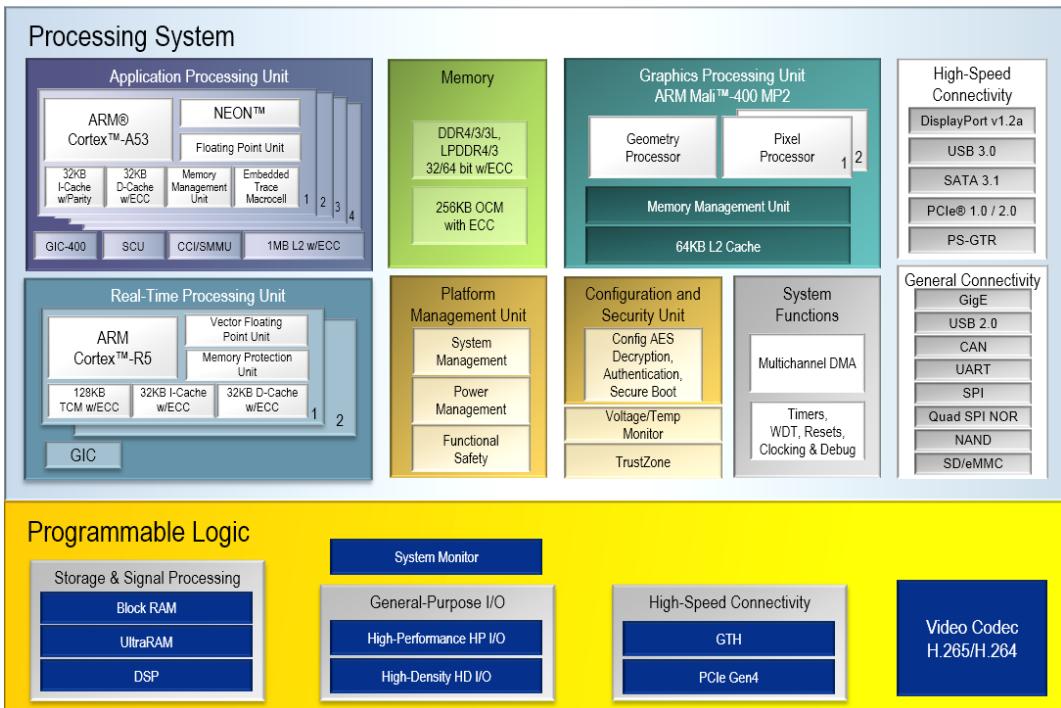


Abbildung 3.4: Zynq UltraScale+ MPSoC EV Block Diagram [Xilinx Inc. (b)]

Im Folgenden findet sich eine Liste der besonderen Komponenten dieser Familie:

- **APU**
 - 64-bit Quad-core ARM Cortex-A53 1.5 GHz
 - NEON Media Processing Engine + Floating Point Unit (FPU)
 - Unterstützung für 32/64-Bit-Betriebsmodi
 - 32 KB Level-1 cache
 - 1 MB Level-2 cache
- **RPU**
 - 32-bit Dual-core ARM Cortex-R5 600 MHz
- **graphics processing uni(GPU)**
 - ARM Mali-400 MP2
- **On-Chip Memory (OCM):** 256KB
- **Zwei 8-Channel Direct Memory Access (DMA) controllers**
- **System Memory Management Unit (SMMU)**

- Platform Management Unit (PMU)

Der PS unterstützt auch viele Eingangs-/Ausgangsschnittstellen wie SRAM-Schnittstellen (SD, QSPI, NAND), GPIOs, UART usw.

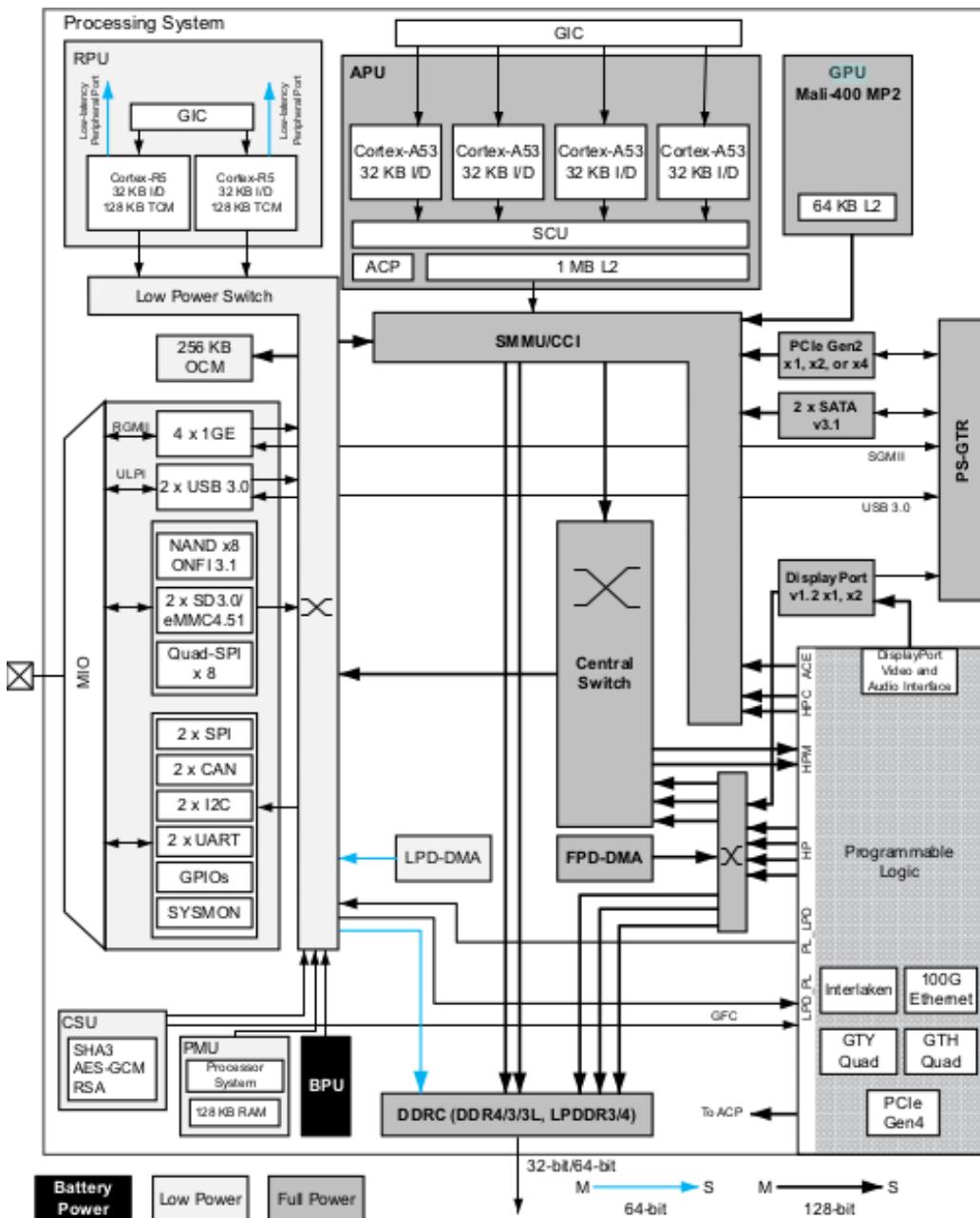


Abbildung 3.5: Zynq UltraScale+ MPSoC Top-Level Blockdiagramm [Xilinx Inc. (2019)]

Im Top-Level-Blockdiagramm der Zynq MPSoC-Architektur 3.5 ist die Verbindung zwischen den verschiedenen Blöcken markiert. Da kann die verschiedenen Verarbeitungseinheiten (APU, RPU, GPU) sowie viele Verbindungen zwischen den Blöcken

finden. Zu beachten ist, dass die Richtung der Pfeile die Prioritätsreihenfolge zwischen Master und Slave festlegt. Sie haben vielleicht bemerkt, dass in Abbildung 3.4 erwähnt wird, dass es sich um das Blockdiagramm für die EV-Variante handelt. Tatsächlich gibt es drei Varianten des Zynq MPSoC, die wie folgt gekennzeichnet sind:

- **CG**: Mittelklasse-Gerät, mit Dual Cortex-A53 und Dual Cortex-R5.
- **EG**: High-End-Gerät, mit Quad-Cortex-A53, Dual-Core-Cortex-R5 und Mali GPU
- **EV**: High-End-Gerät, mit Quad Cortex-A53, Dual-Core Cortex-R5, Mali GPU und Video-Codec (H.264, H.265).

Die beiden Hauptkomponenten des ZynqMP, das Verarbeitungssystem und die programmierbare Logik, werden im Folgenden ausführlicher beschrieben.

3.2.1.3 Processing System (PS)

Das ZynqMP-Verarbeitungssystem umfasst einerseits den ARM-Prozessor, anderseits aber auch mehrere zugehörige Verarbeitungsmodule, die eine Application Processing Unit (APU) bilden, sowie weitere Schnittstellen für Peripheriegeräte, Cache-Speicher, Speicherschnittstellen, Verbindungs- und Takterzeugungsschaltungen. Die APU besteht aus einem speziellen ARM Cortex-A53 MPCore (Quad oder Dual). In der ARM-Gerätefamilie gelten diese als relativ stromsparend, sind aber in der Lage, ein vollwertiges Betriebssystem wie Linux, Android oder ähnliches auszuführen

Abbildung 3.6 fasst einige wichtige Merkmale der APU zusammen. So verfügt sie beispielsweise über einen separaten 32-KB-Cache der Ebene 1 für Befehle und Daten und einen gemeinsamen 1-MB-Cache der Ebene 2. Eine Snoop Control Unit (SCU) sorgt für Cache-Kohärenz zwischen den Kernen und zeigt an, wenn die Daten ungültig sind.

Der L2-Cache-Controller APU kommuniziert mit dem Rest des SoC über eine 128-Bit AXI Coherency Extension (ACE) Master-Schnittstelle zur Cache Coherent Interconnect

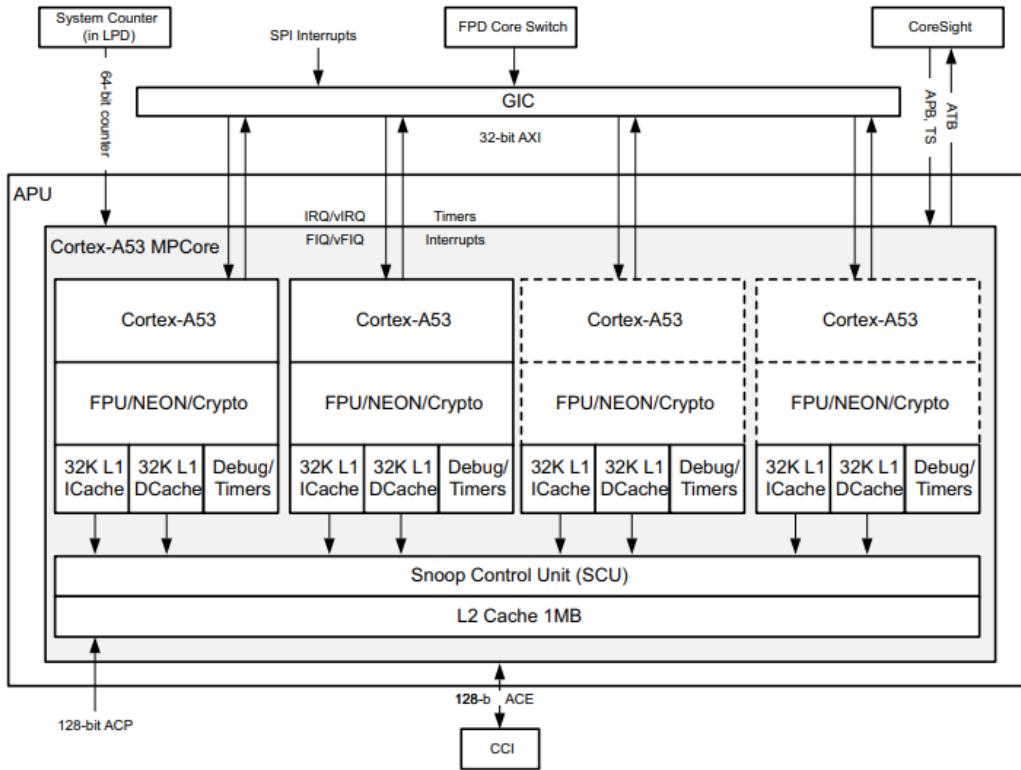


Abbildung 3.6: Detailliertes APU-Blockdiagramm [UG1137 (2017)[p. 57]]

Andererseits kann ein 128-Bit-Accelerator Coherency Port (ACP) Slave-Controller verwendet werden, wenn ein anderer Block mit Master-Zugriff auf Daten im L2-Cache zugreifen möchte. In der Praxis bedeutet dies, dass der PL auf den L2-Cache über den Port

3.2.1.4 Programmable Logik (PL)

Dieser Abschnitt bietet einen Überblick über die verfügbaren Funktionen der Zynq Ultrascale+ MPSoC Programmable Logic (PL). Tabelle 3.1 ist ein Vergleich einiger EG-Bausteine. Es gibt tatsächlich mehr Varianten als die dort aufgeführten. Die FPGA-Fabric besteht hauptsächlich aus Logikzellen und konfigurierbaren Logikblöcken (CLB). Ein CLB kann Flipflops und eine Look-up-Table (LUT) enthalten, muss es aber nicht.

Gerät Name	ZU4EV	ZU5EV	ZU7EV
System Logic Cells (K)	192	256	504
Speicher (Mb)	18.5	23.1	38.0
DSP-Schnitte	728	1,248	1,728
Video-Code-Einheit (VCU)	1	1	1
Maximale E/A-Pins	252	252	464

Tabelle 3.1: Vergleich einiger Zynq Ultrascale+ MPSoC EG [Xilinx Inc. (b)]

Der Speicher eines FPGAs besteht entweder aus dedizierten Speicherblöcken wie BlockRAM und UltraRAM oder aus CLB, die als RAM-Zellen verwendet werden (Distributed RAM). Schließlich enthält das FPGA eine Reihe von DSP48E2 IP-Blöcken, die Multiplikationen und andere arithmetische/logische Aufgaben effizient durchführen können

3.2.2 MCP251xFD CAN Controller + Transceiver

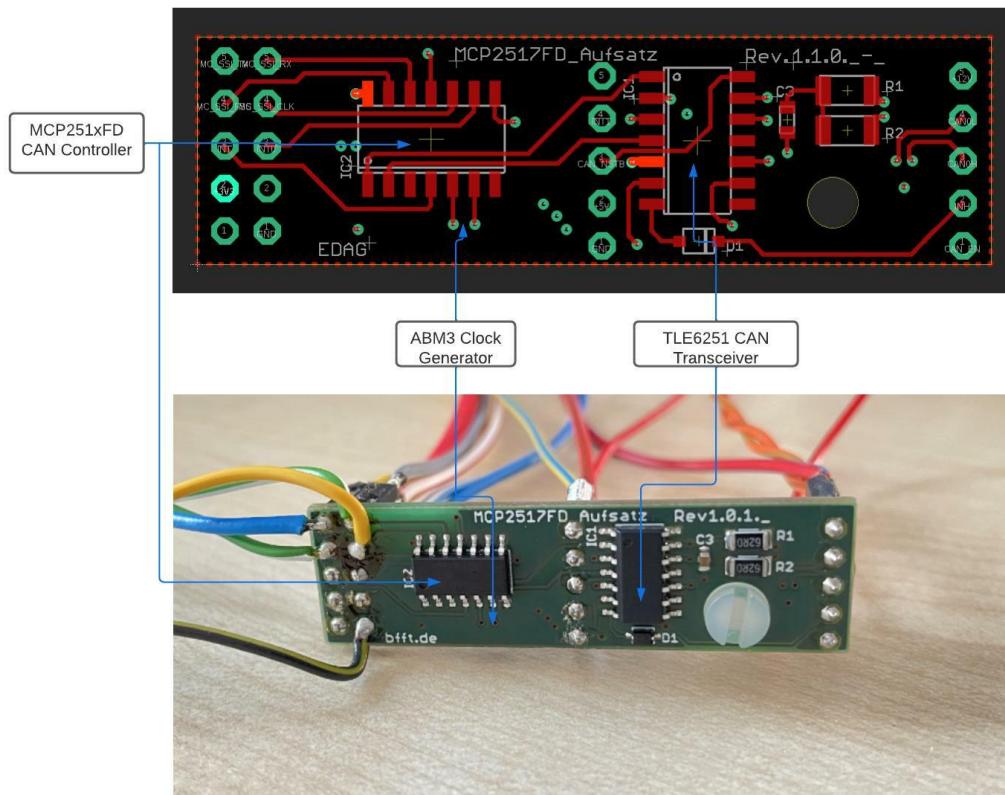


Abbildung 3.7: CAN-Transceiver- und Controller-Modul

3 Umsetzung

Bevor wir uns näher mit diesem CAN-Controller befassen, sollte erklärt werden, dass das “x“ im Namen des Chips entweder “7“ oder “8“ werden kann. MCP251XFD unterstützt also MCP2517FD und MCP2518FD.

Die Firma entwickelte im Rahmen dieses Projekts, aus kostengünstigen Gründen den MCP2517FD. Zweck war es, zu prüfen, wie schnell Daten mit diesem CAN-Controller verarbeitet werden können. Auf die Eigenschaften des Chips wird nun im Folgenden eingegangen.

Der MCP2517FD-Chip ist ein kompaktes Board, das eine komplette CAN-Lösung bietet und als Steuerknoten in einem CAN-Netzwerk verwendet wird. Wie in Abbildung 1 zu sehen ist, besteht das Board hauptsächlich aus einem CAN-FD-Controller mit SPI-Schnittstelle (MCP2517FD) und einem Hochgeschwindigkeits-CAN-Transceiver (TLE6251). Letzterer stellt eine physikalische Verbindung mit dem CAN-Bus selbst her, während der CAN-Controller MCP2517FD eine Schnittstelle zwischen der MCU und dem PHY darstellt. Die Aufgabe des CAN-Controllers seinerseits ist es, die Arbitrierung, das Nachrichtenframing, die Nachrichtenvalidierung, die Fehlererkennung, die Nachrichtenfilterung und vieles mehr zu übernehmen. Weiterhin unterstützt er sowohl CAN-Frames im klassischen Format (CAN2.0B) als auch im CAN-Flexible-Data-Rate-Format (CAN FD), wie in ISO 11898- 1:2015.

Der ISO 11898- 1:2015 hier, der von der International Organization for Standardization (ISO) rausgegeben wurde, spezifiziert das klassische CAN-Rahmenformat und das neu eingeführte CAN Flexible Data Rate Frame Format. Das klassische CAN-Frame-Format erlaubt Bitraten bis zu 1 Mbit/s und Nutzdaten bis zu 8 Byte pro Frame.

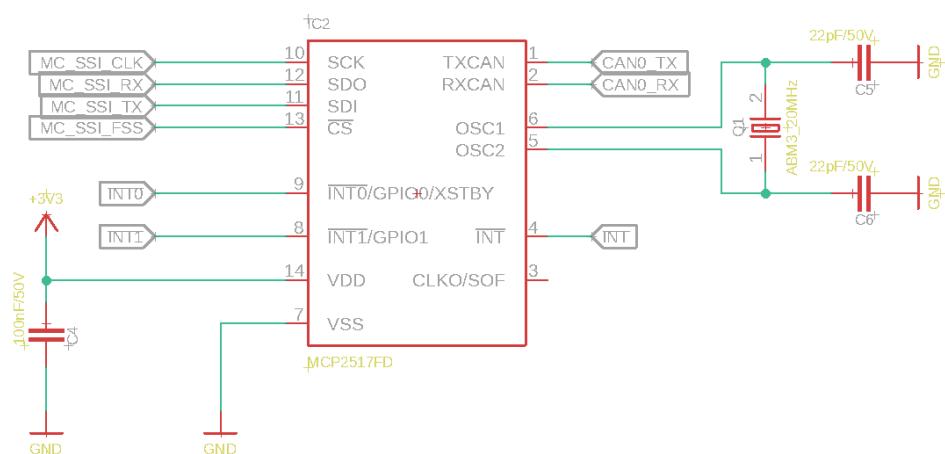


Abbildung 3.8: CAN Controller Modul

Der MCP2517FD enthält die folgenden Hauptblöcke [Transmission u. a. (2018)]:

- Das CAN FD Controller-Modul implementiert das CAN FD-Protokoll und enthält die FIFOs und Filter.
- Die SPI-Schnittstelle, die zur Steuerung des Geräts durch Zugriff auf SFRs und RAM verwendet wird
- Der RAM-Controller arbitriert die RAM-Zugriffe zwischen dem SPI- und dem CAN FD Controller-Modul
- Der Nachrichten-RAM, der zur Speicherung der Daten der Nachrichtenobjekte verwendet wird
- Der Oszillator: Das MCP251XFD CAN Controller verwendet den ABM3 Clock Generator, als Standardtaktquelle für den Chip. Der ABM3 auf diesem Board wurde so programmiert, dass er eine Ausgangsfrequenz von 20 MHz erzeugt.

Pin	Name	Beschreibung
1	CAN0_TX	TX Interrupt
2	CAN0_RX	RX Interrupt
3	CLKO/SOF	
4	INT	Interrupt
5	OSC1	Clock Output 1
6	OSC2	Clock Output 2
7	VSS	Ground
8	INT0	
9	INT1	
10	SCK	SPI Clock
11	SDI	SPI Data IN
12	SDO	SPI Data OUT
13	CS	Chip Select zur Auswahl des Microchips
14	VDD	Versorgungsspannung des CAN Module. Die Spannung liegt zwischen 3.3V und 5V

Tabelle 3.2: mcp251xfd Pins

3.2.2.1 CAN FD Controller Modul

Der Controller kann in verschiedenen Modi eingestellt werden, nämlich

- Configuration

3 Umsetzung

- Normal CAN FD
- Normal CAN 2.0
- Sleep
- Listen Only
- Restricted Operation: Also Eingeschränkter Betrieb
- und Internal and External Loop back modes (Interner und externer Loopback-Modus)

3.2.2.2 TLE6251 CAN Transceiver

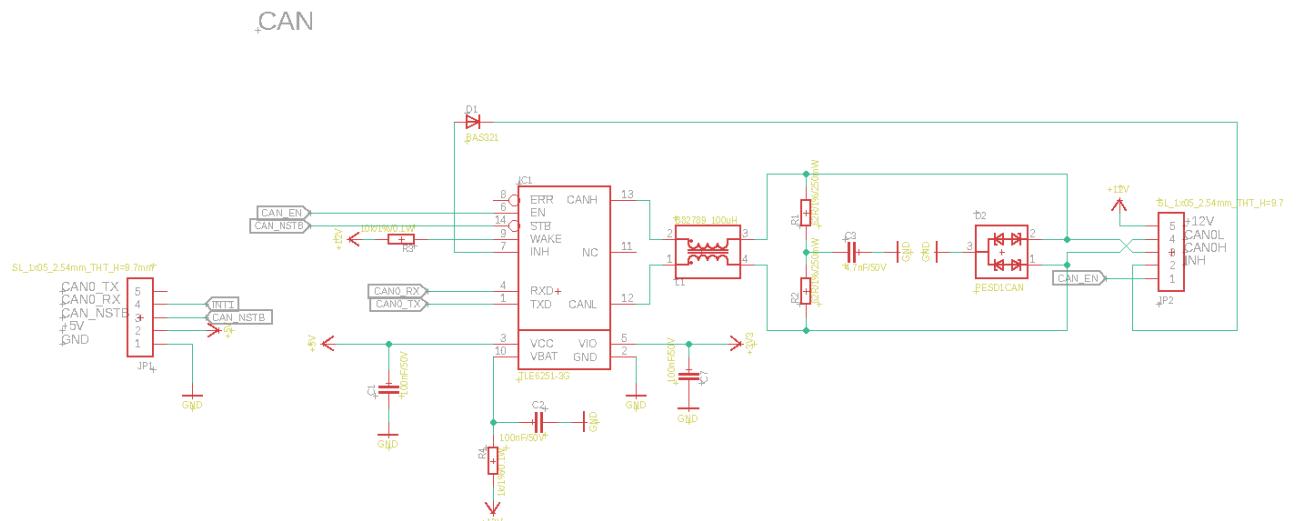


Abbildung 3.9: CAN Transceiver Modul

Der Hochgeschwindigkeits-CAN-Transceiver, der TLE6251, stellt eine physische Verbindung mit dem CAN-Bus her. Dieser ermöglicht eine Kommunikationsgeschwindigkeit von bis zu 5Mbps und unterstützt die Betriebsmodi Normal und Standby. Der Normalmodus ist eingeschaltet, wenn der STBY-Pin, der auf den AN-Pin der mikroBUS geführt wird, auf einem logischen Low-Pegel liegt, während der TXD-Pin auf einem hohen logischen Pegel gehalten wird. Im Normalmodus können die Daten über die CAN H/L-Busleitungen gesendet und empfangen werden.

3.3 Konfiguration und Bauen des Systems

In diesem Abschnitt werden vor allem die Werkzeuge erläutert, die zur Erstellung einer angepassten Linux-Distribution benötigt werden, sowie die Methode, mit der eine solche Distribution erstellt wird. Dabei wird ein Überblick über das Yocto-Projekt als Grundlage für die PetaLinux-Werkzeuge gegeben. Der Prozess der Erstellung einer Linux-Distribution mit den PetaLinux-Tools wird beschrieben. Darüber hinaus wird der Prozess der Konfiguration und Erstellung des Kernels und des Root-Dateisystems für den Zynq UltraScale+ MPSoC im Detail erklärt.

Um den PetaLinux-Build Prozess zu verstehen, ist es wichtig, bestimmte Schlüsselkonzepte im Zusammenhang mit dem Yocto-Projekt zu verstehen, darum wird in dem nächsten Abschnitt erst das Yocto-Projekt vorgestellt.

3.3.1 Das Yocto Projekt

Die Arbeit mit dem Yocto-Projekt ist die Basis für die Erstellung einer Linux-Distribution mit den PetaLinux-Tools. Die PetaLinux-Tools nutzen den Build-Prozess des Yocto-Projekts zur Erstellung der Linux-Distribution und verbergen damit die Komplexität, die mit der direkten Verwendung des Yocto-Projekts einhergeht. Sie bieten eine einfach zu bedienende Kommandozeilenschnittstelle, die es Entwicklern erleichtert, das Betriebssystem zu erstellen.

Das Yocto-Projekt jedoch ist ein Open-Source-Projekt, das auch Entwicklern hilft, angepasste Linux-Distributionen für ihre eingebetteten Systeme zu erstellen, die verschiedene Prozessorarchitekturen unterstützen. Um die Distribution zu erstellen, verfügt das Projekt über eine Sammlung von Werkzeugen und Mechanismen zur Erstellung von angepassten Komponenten wie FSBL, U-Boot, Device-Tree und Kernel-Image zum Booten von Linux auf Embedded-Systemen.

Das Yocto-Projekt verwendet ein Schichtenmodell(Layer Model), das den Benutzern die Flexibilität bietet, schichtspezifische Änderungen vorzunehmen.

3.3.1.1 Yocto Layer

Yocto-Layer oder Meta-Daten-Layer sind Repositories von Konfigurations- und Build-Skripten zusammen mit Build-Spezifikationen (BitBake-Rezepten), die dem Yocto-Build-System (OpenEmbedded Build System) mitteilen, wie eine angepasste Linux-Distribution zu bauen ist. Sie können hardwarespezifisch sein, oder so flexibel sein, dass sie auch für andere Architekturen angepasst werden können. Je nachdem, wie komplex der Entwickler eine bestimmte Schicht gestalten möchte, können Layer verwendet werden, um bestimmte Build-Aufgaben zu trennen oder zu kombinieren. Die

Erhöhung der Build-Aufgaben, die mit bestimmten Layers verbunden sind, erhöht die Komplexität des Projekts und erschwert den Entwicklern die Anpassung, Wartung und Wiederverwendung von diesen Layers. Abbildung 3.10 zeigt die Ausgabe einer typische bblayers.conf Datei.

```
# WARNING: this configuration has been automatically generated and in
# most cases should not be edited. If you need more flexibility than
# this configuration provides, it is strongly suggested that you set
# up a proper instance of the full build system and use that instead.

LCONF_VERSION = "7"

BBPATH = "${TOPDIR}"
SDKBASEMETAPATH = "/home/landry/mein_Abschluss/IPU-NG/components/yocto"
BBLAYERS := " \
    ${SDKBASEMETAPATH}/layers/core/meta \
    ${SDKBASEMETAPATH}/layers/core/meta-poky \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-perl \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-python \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-filesystems \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-gnome \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-multimedia \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-networking \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-webserver \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-xfce \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-initramfs \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-oe \
    ${SDKBASEMETAPATH}/layers/meta-clang \
    ${SDKBASEMETAPATH}/layers/meta-browser/meta-chromium \
    ${SDKBASEMETAPATH}/layers/meta-qt5 \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-microblaze \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-bsp \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-pynq \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-contrib \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-standalone \
    ${SDKBASEMETAPATH}/layers/meta-xilinx-tools \
    ${SDKBASEMETAPATH}/layers/meta-petalinux \
    ${SDKBASEMETAPATH}/layers/meta-virtualization \
    ${SDKBASEMETAPATH}/layers/meta-openamp \
    ${SDKBASEMETAPATH}/layers/meta-jupyter \
    ${SDKBASEMETAPATH}/layers/meta-python2 \
    ${SDKBASEMETAPATH}/layers/meta-som \
    ${SDKBASEMETAPATH}/layers/meta-security \
    ${SDKBASEMETAPATH}/layers/meta-security/meta-tpm \
    /home/landry/mein_Abschluss/IPU-NG/project-spec/meta-user \
    /home/landry/mein_Abschluss/IPU-NG/project-spec/meta-ipu_ng \
    /home/landry/mein_Abschluss/IPU-NG/components/yocto/workspace \
    "
```

Abbildung 3.10: Menuconfig Startbildschirm des PetaLinux-Projekts

3.3.1.2 Poky

Bei Poky handelt es sich um eine Referenzdistribution, die vom Yocto-Projekt verwendet wird, um eigene Linux-Distributionen zu erstellen. Sie besteht aus dem OpenEmbedded Build-System und den Metadaten, die dem Build-System helfen, eine minimale Linux-Distribution zu erstellen. Die minimale Linux-Distribution, die mit Poky erstellt wurde, wird mit weiteren Meta-Layern wie meta-xilinx und meta-petalinux angepasst, um das Linux-Image für Xilinx-Prozessoren anzupassen. Bei den Metadaten handelt es sich im Wesentlichen um Konfigurationsskripte und BitBake-Rezeptdateien.

3.3.1.3 Bitbake Engine

Wie in der Bitbake-Dokumentation beschrieben, handelt es sich bei Bitbake um eine Build-Engine, die vom Yocto-Projekt zur Erstellung von Linux-Distributionen verwendet wird. Genau wie der Linux-Kernel ist sie in der Lage, die in den OpenEmbedded-Build-Skripten beschriebenen Aufgaben parallel auszuführen, zu verwalten und zu koordinieren. Darüber hinaus stellt die Bitbake-Engine sicher, dass jede Aufgabe die erforderlichen Ressourcen für die Ausführung hat, indem sie BitBake-Rezepte verwendet. Dies sind Dateien die die Endung ".bb" tragen.

3.3.1.4 Bitbake Recipes(Bitbake Rezept)

In diesem Abschnitt möchte ich die Bitbake-Rezepte beschreiben, da im nächsten Paragraphen ein paar Rezepte geschrieben werden. Rezepte sind die von BitBake verwendeten grundlegenden Metadaten-Dateien, die mit der Dateinamenerweiterung .bb gekennzeichnet sind. Das Ziel wäre es, ein oder mehrere Ausgabepakete mit diesen Rezepten zu erhalten, die dann in das rootfs integriert werden können. folgenden wichtige Informationen sollten Datei vorhanden sein:

- Wichtige Informationen über das Rezept wie z. B. die Version, den Autor, die Homepage und die Lizenz
- Build- und Laufzeit-Abhängigkeiten
- Pfad zur Quellcode und wie man ihn abruft
- Mögliche Patches für den Quellcode und wie sie angewendet werden können
- Konfiguration und Kompilierung des Quellcodes
- wie und wo die generierten Build-Produkte zu installieren sind

Die wesentlichen Aufgaben, die vom Build-System beim Parsen eines Rezepts in der angegebenen Reihenfolge ausgeführt werden, sind die folgenden[Projekt]

do_fetch: Diese Funktion lädt den Quellcode von einer angegebene Pfad Variable SRC_URI herunter.

do_unpack: entpackt die heruntergeladenen Daten

do_patch: fügt Patches auf den Quellcode ein

do_configure: konfiguriert den gesamten Quellcodebaum.

do_compile: kompiliert den vorbereiteten Quellcode

do_stage: legt die Kompilierungsergebnisse im Verfügungsbereich ab.

do_install: sorgt dann für die Einrichtung des Pakets im Paketbereich.

do_package: erstellt ein Paket, das die gewünschte Ausgabe enthält.

Die resultierende Pakete oder Module werden dann im Rootfs eingefügt

3.3.2 Konfigurieren des PetaLinux-Projekts

Wie im Abschnitt 2.6.2 gut beschrieben, wird das Petalinux Kommando **petalinux-config** zur konfigurieren des Projekts verwendet. Bei der Konfiguration des Projekts ist es möglich die Board-Support-Pakete für ein Design in der Vivado Design-Suite zu erstellen, um einen Pfad zur Hardwar-Design-Datei während der Konfigurationsphase anzugeben, verwendet man die Option **-get-hw-description** wie im 2.6.2 erklärt.

Mit diesem Befehl kann Petalinux die Linux-Distribution entsprechend der bereitgestellten Hardwarebeschreibungsdatei (HDF) konfigurieren. Die Übergabe des Pfades zum BSP hilft bei der Generierung der Konfiguration für die Erstellung des korrekten Device-tree, FSBL, U-Boot und der Kernel-Treiber, die zum Booten von Linux auf der Xilinx-Plattform erforderlich sind.

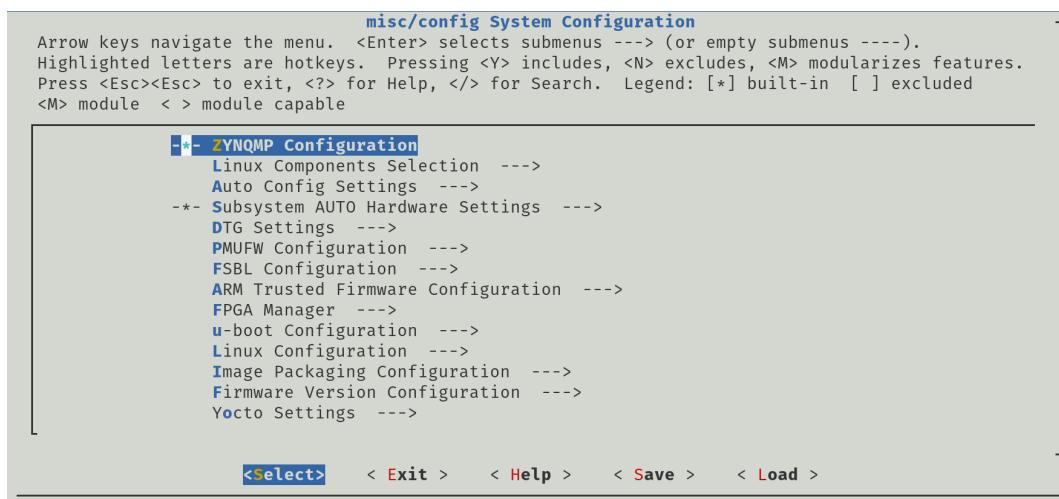


Abbildung 3.11: Menuconfig Startbildschirm des PetaLinux-Projekts

Die Abbildung 3.11 zeigt der Menuconfig Startbildschirm, der erscheint, nachdem der Befehl **petalinux-config** die nach dem Befehl **petalinux-create** erzeugten Kconfig-Dateien analysiert hat. Diese Menuconfig ist eigentlich ausreichend, um das gesamte Projekt zu konfigurieren, aber man kann jedoch auch individuelle Konfigurationen für Boot-Komponenten mit einer ähnlichen Menuconfig-Schnittstelle

3 Umsetzung

vornehmen. Diese Individuelle Komponenten Konfigurationen sind nämlich wichtig wenn man komplexe Konfigurationen vornehmen soll. So gibt es eine Menuconfig für den Kernel, das U-boot, das Rootfs ...

Damit unser MCP251xfd CAN-Controller erkannt und in Betrieb genommen werden kann, sollte man z.B. das Kernel-Menüocnfig verwenden, um den Treiber zu integrieren. Absichtlich wurde Petalinux 2021 verwendet, da für den Chip ein Treiber vorhanden ist.

```
Linux Kernel Configuration
└-> Networking support
└-> CAN bus subsystem support
└-> CAN Device Drivers
└-> CAN SPI interfaces
└-> Microchip MCP251xFD SPI CAN controllers
```

Abbildung 3.12: Pfad zur mcp251xfd Treiber

Der oben gezeigte Pfad führt uns zum Kernel-Fenster, wo man den Treiber für den MCP251xFD aktivieren oder deaktivieren kann. Die Abbildung 3.13 zeigt alle CAN-Controller mit SPI-Schnittstelle, die uns der Kernel 5.10 bietet. Hier können sie auch als Modul aktiviert werden, so dass der Treiber nach dem Bootvorgang geladen oder entladen werden kann. Dies wurde in dieser Arbeit genutzt, um zu überprüfen, ob der Treiber den Mikrochip korrekt initialisiert hat.

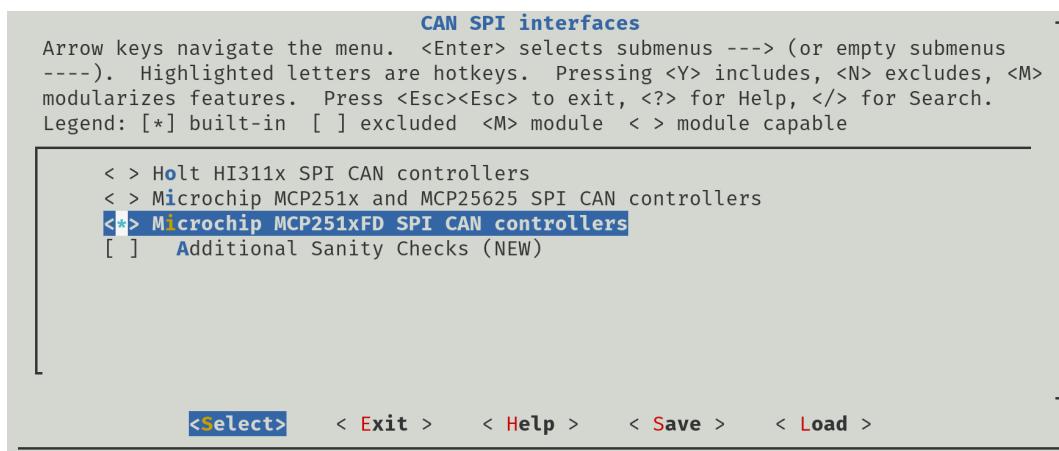


Abbildung 3.13: Konfig Fenster mcp251xfd Treiber

3.3.3 Device-Tree Eintrag für den MCP251XFD CAN-Controller

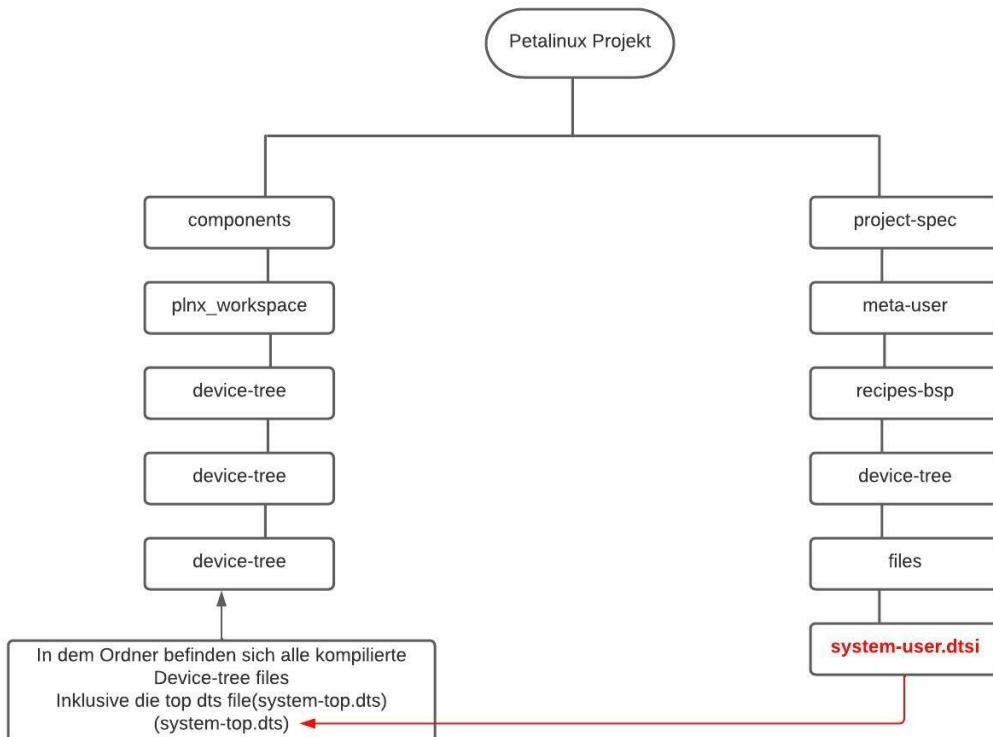


Abbildung 3.14: Device-tree Projekt Struktur

Damit unser Mikrochip nach dem Bootvorgang in Betrieb genommen werden kann, müssen dem Kernel Informationen (wie z.B. der Name des Hardwaretreibers, die Taktfrequenz oder der Interrupt-Pin...) übergeben werden, damit die Hardware während des Bootvorgangs initialisiert wird. Daher wird in Petalinux empfohlen, die Datei **system-user.dtsi** zum Hinzufügen, Löschen oder Ändern eines Gerätebaumknotens zu verwenden. Diese Datei befindet sich in `<project>/project spec/meta-user/recipes-bsp/device-tree/files/`. Sie wird an das Ende der Datei system-top.dts gestellt wie im Listing 3.1 dargestellt, so dass sie eine höhere Priorität erhält, und wird somit zuerst kompiliert.

Listing 3.1: Inhalt der system-top.dts Datei

```

1 /dts-v1/;
2 #include "zynqmp.dtsi"
3 #include "zynqmp-clk-ccf.dtsi"
4 #include "zcu106-reva.dtsi"
5 #include "pcw.dtsi"
6 /
7 chosen {
8     bootargs = "earlycon";
9     stdout-path = "serial0:115200n8";
  
```

3 Umsetzung

```
10    };
11    aliases {
12        ethernet0 = &gem3;
13        i2c0 = &i2c0;
14        i2c1 = &i2c1;
15        serial0 = &uart0;
16        serial1 = &uart1;
17        spi0 = &qspi;
18        spi1 = &spi0;
19    };
20    memory {
21        device_type = "memory";
22        reg = <0x0 0x0 0x0 0x7ff00000>, <0x00000008 0x00000000 0x0 0x80000000>;
23    };
24 };
25 #include "system-user.dtsi"
```

Das Petalinux-Projekt wurde mit einer von Vivado exportierten Hardwarebeschreibungsdatei (HDF-Datei) konfiguriert, in der bestimmte Vorkonfigurationen bereits vorgenommen wurden. Beispielsweise die Konfiguration der Pins, des gpio-Controllers, des SPI-Controllers usw.

```
spi1: spi@ff050000 {
    compatible = "cdns,spi-ripl6";
    status = "disabled";
    interrupt-parent = <&gic>;
    interrupts = <0 20 4>;
    reg = <0x0 0xff050000 0x0 0x1000>;
    clock-names = "ref_clk", "pclk";
    #address-cells = <1>;
    #size-cells = <0>;
    power-domains = <&zynqmp_firmware PD_SPI_1>;
};
```

Abbildung 3.15: Xilinx SPI-Controller

Der Code, der in Abbildung 3.15 zu sehen ist, beschreibt den im Vivado implementierten Xilinx-SPI-Controller, an den unserer CAN-Controller angeschlossen werden soll. Im **reg** ist die Basisadresse und der Adressbereich, in dem der SPI-Controller betrieben wird.

Der Code, der in den aktuellen Device-tree für den CAN-Controller eingefügt werden muss, ist der unten stehende.

Listing 3.2: Device-tree Eintrag für den mcp251xfd CAN-Controller

```
1
2 /include/ "system-conf.dtsi"
3 / {
4     can0_osc: can0_osc {
5         compatible = "fixed-clock";
6         #clock-cells = <0>;
7         clock-frequency = <20000000>;
8     };
}
```

```

9  };
10
11 &spi0 {
12     #address-cells = <1>;
13     #size-cells = <0>;
14     can00 {
15         compatible = "microchip,mcp251xfd";
16         reg = <0>;
17         clocks = <&can0_osc>;
18         pinctrl-names = "default";
19         //pinctrl-0 = <&can0_pins>;
20         interrupts-extended = <&gpio 78 8>;
21         spi-max-frequency = <8333333>;
22         microchip,rx-int-gpios = <&gpio 79 1>;
23     };
24 };

```

wichtige Information dazu wurde in den folgende Tabelle zusammengefasst.

Funktion	Beschreibung
<i>compatible</i>	wird verwendet, um zu entscheiden, wie das Gerät ausgeführt werden soll und das Gerät mit dem Treiber zu verbinden. Er enthält eine Zeichenkette in der Form <Hersteller>,<Modell>. In unseren Fall handelt es sich um ein microchip herstellte mcp251xfd.
<i>reg = <0></i>	dem CAN-Controller wird eine eindeutige ID zugewiesen, der hat also keine Adresse Bereich.
<i>clocks</i>	Der interne Taktgenerator im Controller wird hier beschrieben. Der kann 40, 20 oder 4 MHz generieren. Hier wurde er aber auf 20 MHz eingestellt.
<i>interrupts-extended</i>	Hier werden die vom Gerät erzeugten Interrupts aufgelistet. Es wird auch spezifiziert an welchem GPIO-Pin diese Interrupts empfangen werden.
<i>spi-max-frequency</i>	steht für die maximale SPI-Taktfrequenz des Geräts. Diese wurde auf 8,33 MHz gesetzt, da für eine gute Kommunikation mit dem SPI-Master die SPI-Taktfrequenz die Hälfte oder weniger des Takts betragen sollte.

3.3.4 Rezepte zur Kompilierung der Software Modulen

Für die Erstellung von Rezepte wurde im Rahmen dieser Arbeit das Petalinux-devtool verwendet. Das ist ein Werkzeug, welches das Yocto devtool benutzt, um Software zu bauen, zu testen und einzubinden. Mit dem "petalinux-devtool" Kommando kann man zum Beispiel:

- neue Rezepte einfügen
- Quelle Code von existierenden Rezepte ändern
- Umbenennen einer Rezeptdatei im Arbeitsbereich
- Oder Rezepte kompilieren

Um die neuen Änderungen vom Rest des Projekts zu isolieren, wurde eine neue Ebene erstellt, in die alle Änderungen geschrieben wurden.

3.3.4.1 Hinzufügen einer neue Layer zu BBLAYERS

Es wurde zuerst ein neue Ebene-Verzeichnis und dessen Konfigurationsverzeichnis erstellt. dafür wurde ein bestehende Layers konfigurationsdatei in das conf-Verzeichnis der meta-Layers kopiert und wie folgt geändert. Diese Datei enthält die Informationen, die von BitBake benötigt werden, um die Ebene und die darin enthaltenen Metadaten zu erkennen. Insbesondere enthält diese Datei die Variable **BBPATH**, die das Stammverzeichnis des Layers angibt, in dem BitBake alle Dateien finden soll, die von Rezepten geerbt werden. Außerdem gibt die Variable **BBFILES** die Pfade an, in denen BitBake erwartet, Rezepte zu finden. Abhängigkeiten zwischen Schichten müssen stattdessen durch die Variable **LAYERDEPENDS_*** ausgedrückt werden.

Listing 3.3: Hinzufügen neue Yocto-ebene

```
1
2  # We have a conf and classes directory, add to BBPATH
3  BBPATH .= ":${LAYERDIR}"
4
5  # We have recipes-* directories, add to BBFILES
6  BBFILES += "${LAYERDIR}/recipes-*//*/*.bb \
7  ${LAYERDIR}/append/*/*.bbappend"
8
9  BBFILE_COLLECTIONS += "meta-ipu_ng"
10 BBFILE_PATTERN_meta-ipu_ng = "^${LAYERDIR}/*"
11 BBFILE_PRIORITY_meta-ipu_ng = "8"
12 LAYERSERIES_COMPAT_meta-ipu_ng = "gatesgarth"
13
14 # Set a variable to get to the top of the meta-layer location
15 HAB_BASE := '${LAYERDIR}'
```

Anschliessend sollte die neue Yocto-Layer mit Hilfe des Systemkonfiguration Menü zu der hauptkonfiguration Layers (BBLAYER) hinzugefügt werden. Die Datei enthält alle Verweise auf die Ebenen, die von BitBake während des Analyseprozesses der Rezepte gescannt werden.

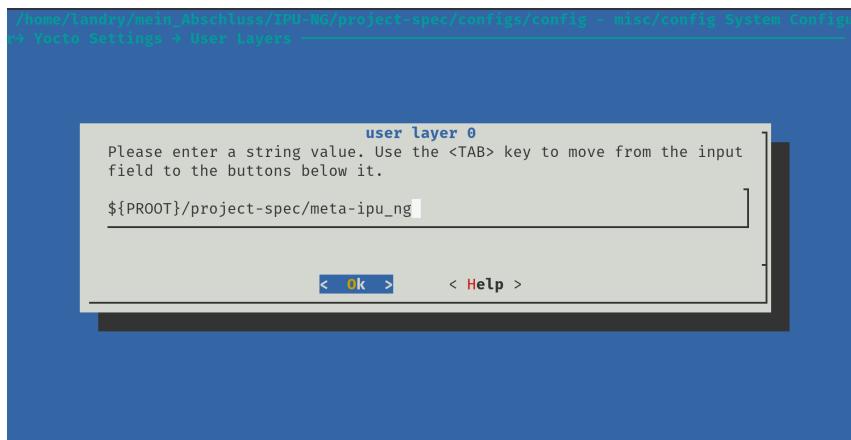


Abbildung 3.16: Neue Yocto-Layer für IPU-NG Software Module

Das Abbildung 3.10 ist beispielsweise die Ausgabe der Datei **bblayers.conf** des Projekts. Am Ende können Sie sehen, dass die neue Layer am Ende der Datei eingefügt wird, nachdem das Projekt kompiliert wurde.

3.3.4.2 Rezept zur Kompilierung der Software Modulen

Die folgenden Variable wurden notwendig für die Erstellung der Rezepte.

SUMMARY : Diese Variable enthält einen String-Wert, den der Benutzer hilft zu wissen, worum es in dem Rezept geht.

LICENSE : In dieser Variable ist die Art der Lizenz anzugeben, die man für das Rezept verwenden möchten. In unsere Fall ist es CLOSE, da man die in Firma entwickelte Software verwenden.

LIC_FILES_CHKSUM : Diese Datei wird von dem OpenEmbedded-Build-System verwendet, um sicherzustellen, dass sich der Lizenztext nicht geändert hat. Diese Variable ist bei uns aber Leer, da wir die Liezens "CLOSE"verwenden.

PR : Es geben wir das Revision Paket an. Der wurde in unsere Fall auf r0 gesetzt.

PV : definiert die Version des Rezepts

SRC_URI : Hier wird der Pfad zum Quellcode der Software angegeben. In unserem Fall werden die Quellcodes vom Git-Server des Unternehmens abgerufen.

DEPENDS : Hier werden Abhängigkeiten, die zur Kompilierungszeit gebraucht werden, angegeben.

S : Hier gibt man das Quellverzeichnis an, in dem man die gesamte Erstellung durchführen kann.

3 Umsetzung

Das vom Yocto-Projekt unterstützte CMake Autotools wird im Rezept zum Konfigurieren, Kompilieren und Installieren der Software verwendet. Der Zweck dieser Werkzeuge ist es, die Arbeit des Programmierers zu erleichtern, indem mehrere Schritte des Erstellungsprozesses automatisiert werden. Nachfolgend sind die Rezepte zur Kompilierung der Software beschrieben.

Listing 3.4: Rezept für das sw-main-app Modul

```
1
2  #
3  # This file is the sw-main-app recipe.
4  #
5
6  SUMMARY = "sw-main-app applications"
7  LICENSE = "CLOSED"
8  #LIC_FILES_CHKSUM = " "
9
10 #PR = "r0"
11 #PV = "1.0.5+git${SRCPV}"
12
13 SRC_URI = "https://csp.edag.de/bitbucket/scm/ee21i00805/sw-main-app.git"
14 PV = "1.0+git${SRCPV}"
15
16 SRCREV = "${AUTOREV}"
17 S = "${WORKDIR}/git"
18
19 DEPENDS = "gstreamer1.0 gstreamer1.0-plugins-base glib-2.0 sw-system-module
20           sw-system-logger"
21
22 inherit cmake pkgconfig
```

Listing 3.5: Rezept für das sw-system-module Modul

```
1 #
2 ## This file is the sw-system-module recipe.
3 ##
4 SUMMARY = "sw-system-module applications"
5 LICENSE = "CLOSED"
6 LIC_FILES_CHKSUM = ""
7
8 SRC_URI = "https://csp.edag.de/bitbucket/scm/ee21i00805/sw-system-module.git"
9
10 PV = "1.0+git${SRCPV}"
11 SRCREV = "${AUTOREV}"
12
13 S = "${WORKDIR}/git"
14 DEPENDS = "gstreamer1.0 gstreamer1.0-plugins-base glib-2.0"
15
16 inherit cmake pkgconfig
```

Listing 3.6: Rezept für das sw-system-module Modul

```
1 #
2 #
3 ## This file ist the sw-system-logger recipe
4 #
5 SUMMARY = "sw-system-logger applications"
6 LICENSE = "CLOSED"
7 LIC_FILES_CHKSUM =
8
9 SRC_URI = "https://csp.edag.de/bitbucket/scm/ee21i00805/sw-system-logger.git"
10
11
12 PV = "1.0+git${SRCPV}"
13 SRCREV = "515609c99db34769320e54c5535cbfe58031839b"
14
15 S = "${WORKDIR}/git"
16 DEPENDS = "gstreamer1.0 gstreamer1.0-plugins-base glib-2.0 sw-system-module"
17
18 inherit cmake pkgconfig
```

Jede dieser Rezepte lassen sich mit dem folgenden Kommando kompilieren.

```
1 $ petalinux-devtool build <recipe_name>
```

Die resultierenden Softwarepakete wurden dann in rootfs aufgenommen, indem ein Eintrag in der Datei (\$PROOT/project-spec/project-spec/meta-user/conf/user-rootfsconfig) erstellt wurde, wie in Listing 3.7 dargestellt.

Listing 3.7: Einbindung Software in Rootfs

```
1 #Note: Mention Each package in individual line
2 #These packages will get added into rootfs menu entry
3
4 CONFIG_gpio-demo
5 CONFIG_peekpok
6
7 CONFIG_sw-main-app
8 CONFIG_sw-system-module
9 CONFIG_sw-system-logger
10
11 CONFIG_gstreamer-vcu-examples
12 CONFIG_packagegroup-petalinux-v4lutils
13 CONFIG_packagegroup-petalinux-audio
```

Bevor die Software Pakete(sw-main-app, sw-system-logger, sw-system-module) schließlich in Rootfs integriert werden können müssen sie noch in dem Rootfs Konfiguration Menü aktiviert werden.

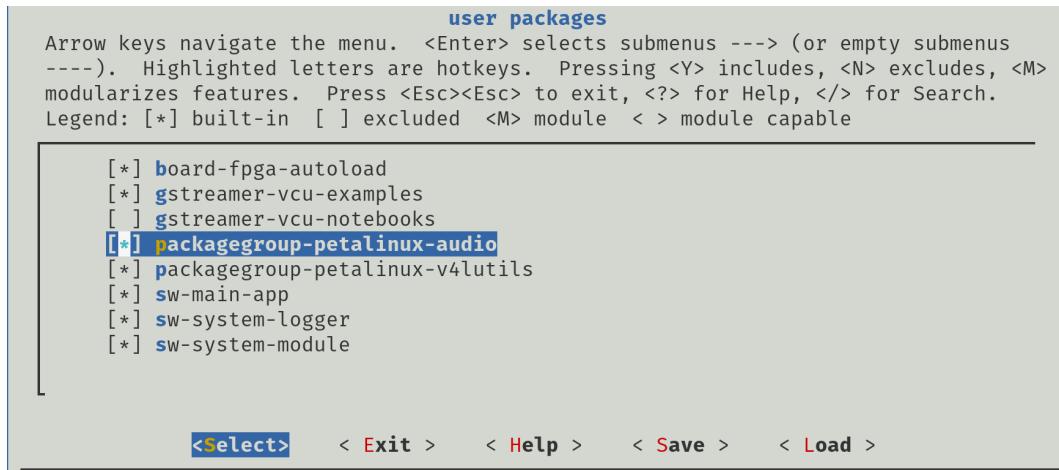


Abbildung 3.17: Aktivierung Software Pakete in Rootfs

3.3.5 Bauen des Systems

Nachdem alle Konfiguration gemacht wurden, könnte das Petalinux Projekt anschließend mit dem Kommando **petalinux-build** gebaut werden. Das petalinux-build Skript baut dann den Gerätebaum gemäß der HDF und der Treiberkonfiguration des Kernels auf, so dass der Treiber für den mcp251xfd CAN-Controller die Informationen über die Hardware am entsprechenden Knoten im Gerätebaum finden und die Hardware physisch lokalisieren kann.

Das **petalinux-build** erstellt auch das Root-Dateisystem für die PetaLinux-Distribution entsprechend der Kernelkonfiguration und der Metadaten in den erstellten Schicht. Schließlich sollten nur noch die Boot Daten mit dem unten stehenden Befehl generiert werden.

```
1 $ petalinux-package --boot --fsbl images/linux/zynqmp_fsbl.elf --
  fpga ../bitstream/bd1_wrapper.bit --pmufw images/linux/pmufw.elf
  --u-boot --force
```

Sobald der Erstellungsprozess des PetaLinux-Projekts abgeschlossen ist, steht eine Reihe von bootfähigen Images zur Verfügung, die in der folgenden Tabelle beschrieben werden.

Erzeugte Datei	Beschreibung
BOOT.BIN	Diese Datei ist eine Sammlung von mehreren Images(Boot-Header, Partition-Header, und Image-Header).
system.bit	Dies ist die Bitstream Datei. Sie enthält die Informationen, die für das FPGA notwendig sind, um die programmierbare Logik gemäß ihrer Konzeption zu konfigurieren.
pmufw.elf	Dies ist die PMU-Firmware wie im Abschnitt 2.5.6.1 beschrieben, läuft auf der PMU und wird nach dem Ausführen des PMU Boot-ROM geladen.
zynqmp_fsbl.elf	Der FSBL ist der erste Bootloader, der auf der APU läuft. Nähere Informationen findet man im Abschnitt 2.5.6.2
u-boot.elf	U-Boot ist die zweite Stufe des Bootloaders. Und wird im Paragraf 2.5.6.2 beschrieben
system.dtb	Das ist die kompilierte Version des Gerätebaums(Device-tree).
Image	Dies ist das generische binäre Linux-Kernel-Image. der zusammen mit dem DTB file können zur Ausführung des Linux-Betriebssystems verwendet werden.
image.ub	Es handelt sich um ein FIT-Image (Flattened Image Tree), das aus dem Kernel-Image und dem Gerätebaum in einem einzigen Image besteht und von U-Boot verwendet werden kann.
bl31.elf	Dad ist die ARM Trusted Firmware, die verwendet wird verwendet, um Übergänge zwischen der sicheren und der unsicheren Boot Modus zu behandeln.

Nach Abschluss aller dieser Schritte haben wir nun alle notwendigen Elemente, um diese auf eine SD-Karte zu laden. In dem nächsten Schritt wird die SD-Karte so formatiert werden, dass sie für das eingebettete Linux-Betriebssystem geeignet ist. Hierfür sollten 2 Partitionen auf der SD-Karte erstellt und formatiert werden. So dass die erste die Boot-Partition und die zweite die Root-Partition ist.

```
landry@pop-os:~$ sudo mkfs.vfat /dev/sdb1
mkfs.fat 4.2 (2021-01-31)
landry@pop-os:~$ sudo mkfs.ext4 /dev/sdb2
mke2fs 1.45.7 (28-Jan-2021)
/dev/sdb2 hat ein ext4-Dateisystem
auf Thu Apr 21 16:57:32 2022
erzeugtTrotzdem fortfahren? (j,n) j
Ein Dateisystem mit 1766971 (4K) Blöcken und 442368 Inodes wird erzeugt.
UUID des Dateisystems: f3db00f3-279c-4142-92c2-29ad46f0cc6d
Superblock-Sicherungskopien gespeichert in den Blöcken:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

beim Anfordern von Speicher für die Gruppentabellen: erledigt
Inode-Tabellen werden geschrieben: erledigt
Das Journal (16384 Blöcke) wird angelegt: fertig
Die Superblöcke und die Informationen über die Dateisystemnutzung werden
geschrieben: erledigt
```

Abbildung 3.18: Formatierung der SD-Karte Partitionen

Nach diesem Schritt können die generierte Dateien auf die SD-Karte geschrieben werden.

Die Dateien BOOT.BIN boot.scr und image.ub sollten aus <plnx-proj-root>/pre built/linux/images/ in der Boot-Partition, die in FAT32-Format formatiert wurde, kopiert werden. der folgende Befehl wurde vom Petalinux Projekt angegeben.

```
1 $ sudo cp -v BOOT.BIN boot.scr image.ub /media/landry/0F4B-72BB
```

Um das Root-Dateisystem auf die Root-Partition zu schreiben, wurde den folgenden Befehl eingegeben.

```
1 $ sudo tar xvf ./images/linux/rootfs.tar.gz -C /media/landry/
f3db00f3-279c-4142-92c2-29ad46f0cc6d
```

3.3.6 Booten des eingebetteten Linux-Images und Test des MCP251xfd CAN-Controller

Damit ich die Möglichkeit habe, den Bootvorgang zu debuggen und eventuelle Fehlermeldungen herauszufinden, habe ich beschlossen, die Hardware-Plattform im JTAG Modus zu booten. Der JTAG-Boot kommuniziert mit dem Xilinx ilinx System Debugger(XSDB), das wiederum mit dem hw_server über den TCP-Port 3121 kommuniziert. ein Tool command language (TCL) Script wird über das **petalinux-boot** Kommando ausgeführt, um die Zynqmp Plattform über JTAG zu starten. Das Skript sendet Befehle und Binärdateien über das Netzwerk an den Remote-Hardware-Server, die die Boot Schritte repräsentieren. der Terminal Emulator Programm Picocom wird hier verwendet, um die Bootmeldungen anzuzeigen.

Die folgenden schritte wurden durchgeführt, um Linux zu booten.

- Mit Hilfe einer JTAG Kabel, ist der JTAG Anschluss des Boards mit dem Hostrechner zu verbinden
 - das gleiche gilt auch mit der serielle Schnittstelle
 - Der Modus-Schalter der Plattform sollte auf den JTAG-Modus umgestellt werden, indem die 4 Schalter auf **ON** geschaltet werden.
 - Die Baudrate zur Kommunikation mit Picocom sollte eingestellt werden.
-

```
1 $ sudo picocom /dev/ttyUSB0 -b 115200
```

- Erstellung einer Verbindung zu einem entfernten GDB-Server mit Hilfe des XSDBs
- Mit dem folgenden Befehl lässt sich Linux Booten.

3 Umsetzung

```
1  petalinux-boot --jtag --fpga --bitstream images/linux/system.  
     bit --kernel
```

Nachdem das Betriebssystem vollständig gebootet hat, kann man den Treiber mit den folgenden Befehlen entladen bzw. laden und überprüfen, ob der Treiber die Hardware korrekt initialisiert hat.

```
1  $ rmmod mcp251xfd  
2  $ modprobe mcp251xfd
```

Abbildung 3.19 zeigt die Ausgabe des **modprobe** Kommandos aus. Da kann sehen dass die Hardware mit den richtigen Taktfrequenz initialisiert wurde.

```
root@xilinx-zcu106-2021_1:~# modprobe mcp251xfd  
[ 672.707562] mcp251xfd spi1.0 can1: MCP2517FD rev0.0 (+RX_INT +MAB_NO_WARN +CRC_REG +CRC_RX +CRC_TX +ECC -HD c:20.00MHz m:8.33MHz r:8.33MHz e:0.00MHz) successfully initialized.  
root@xilinx-zcu106-2021_1:~#
```

Abbildung 3.19: Laden des mcp251xfd Treibers

Ab jetzt kann die CAN Schnittstelle konfiguriert und getestet werden. Die wichtigsten Konfigurationen sind hier die Bit- und Datenrate, die über den IP-Kommando eingestellt werden können. Außerdem muss die Schnittstelle mit dem Befehl **UP** in Betrieb genommen werden. Das Bild 3.20 zeigt das Ergebnis der Ausführung aller diese Befehle an.

```
root@xilinx-zcu106-2021_1:~# ip link set can1 type can bitrate 1000000 dbitrate 2000000 fd 0  
n  
root@xilinx-zcu106-2021_1:~# ip link set can1 up  
[ 54.907474] IPv6: ADDRCONF(NETDEV_CHANGE): can1: link becomes ready  
root@xilinx-zcu106-2021_1:~# ip -details link show can1  
5: can1: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UP mode DEFAULT group default qlen 10  
    link/can promiscuity 0 minmtu 0 maxmtu 0  
    can <FD> state ERROR-ACTIVE (beerr-counter tx 0 rx 0) restart-ms 0  
        bitrate 1000000 sample-point 0.750  
        tq 25 prop-seg 14 phase-seg1 15 phase-seg2 10 sjw 1  
        mcp251xfd: tseg1 2..256 tseg2 1..128 sjw 1..128 brp 1..256 brp-inc 1  
        dbitrate 2000000 dsample-point 0.750  
        dtq 25 dprop-seg 7 dphase-seg1 7 dphase-seg2 5 dsjw 1  
        mcp251xfd: dtseg1 1..32 dtseg2 1..16 dsjw 1..16 dbrp 1..256 dbrp-inc 1  
        clock 40000000 numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
```

Abbildung 3.20: Inbetriebnahme des CAN-Mikrochips

Um den CAN-Controller zu testen, wurde ein Host-Computer an einen CAN-Knoten angeschlossen. Der Computer sollte dann unter der Kontrolle des CAN-Controllers 64-Bit-Daten über den CAN-Bus an den Linux-Kernel senden. Dabei stehen die ersten 21 Bits für die Prüfsumme und die restlichen Bits für die gesendeten Daten. Diese Daten werden dann auf dem Linux-Terminal mit dem folgenden Linux Befehl angezeigt.

3 Umsetzung

1 \$ candum can1

Folgenden Daten wurden an dem Linux Kernel gesendet.

Abbildung 3.21: Gesendete CAN-Nachrichten

die folgenden CAN-Nachrichten wurden dann auf der Konsole empfangen.

Abbildung 3.22: Empfangene CAN-Nachrichten

3 Umsetzung

Bis zu einer Buslast von etwa 70% konnten Daten gesendet und empfangen werden, was ein hervorragendes Ergebnis ist. Im Normalbetrieb liegt die Buslast in der Regel bei etwa 25%.

In Abbildung 3.23 ist die Statistik des Tests zu sehen

CAN Channel: CAN 1 - CAN				
Statistic	Current / ...	Min	Max	Avg
Busload [%]	0.00	0.00	68.31	45.24
Min. Send Dist. [ms]	0.000	n/a	n/a	n/a
Bursts [total]	1325	n/a	n/a	n/a
Burst Time [ms]	6.830	6.830	6.830	6.830
Frames per Burst	0	5	5	5
Std. Data [fr/s]	0	0	500	331
Std. Data [total]	6625	n/a	n/a	n/a
Ext. Data [fr/s]	0	0	0	0
Ext. Data [total]	0	n/a	n/a	n/a
Std. Remote [fr/s]	0	0	0	0
Std. Remote [total]	0	n/a	n/a	n/a
Ext. Remote [fr/s]	0	0	0	0
Ext. Remote [total]	0	n/a	n/a	n/a
Errorframes [fr/s]	0	0	0	0
Errorframes [total]	0	n/a	n/a	n/a
Chip State	Active	n/a	n/a	n/a
Transmit Error Co...	0	n/a	0	n/a
Receive Error Co...	0	n/a	0	n/a
Transceiver Errors	0	n/a	n/a	n/a
Transceiver Delay [n...	162	0	162	127

Abbildung 3.23: Test Statistiken

4 Fazit und Ausblick

4.1 Fazit

Im Rahmen dieser Arbeit **Konfiguration und Optimierung des Embedded-Linux-Betriebssystem für Automotive Image Processing Unit** wurde den Prozess der Erstellung einer Linux-Distribution für den Zynq UltraScale+ MPSoC bzw. der von Xilinx ZCU106 Baord beschrieben, und illustriert, wie ein

4.2 Ausblick

Literaturverzeichnis

[Bosch 1991]

BOSCH, Robert: CAN Specification Version 2.0. In: *Rober Bousch GmbH, Postfach 300240* (1991), 72. <http://esd.cs.ucr.edu/webres/can20.pdf> 2.2, 2.2, 2.3, 2.4, 2.5

[CNX Software]

CNX SOFTWARE: CNXSoft. Xilinx Introduces Zynq Ultrascale+ MPSoC with Cortex A53 and R5 Cores, Ultrascale FPGA. <https://www.cnx-software.com/2015/03/05/xilinx-introduces-zynq-ultrascale-mpsoc-with-cortex-a53-r5-cores-ultrascale-fpga/> 3.2.1.2

[Crockett, Louise H and Elliot, Ross A and Enderwitz, Martin A and Stewart 2014]

CROCKETT, LOUISE H AND ELLIOT, ROSS A AND ENDERWITZ, MARTIN A AND STEWART, Robert W.: *The ZYNQ book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC*. Strathclyde Academic Media, 2014 <https://cds.cern.ch/record/2001018> 3.3

[Daniel P. Bovet and Marco Cesati 2006]

DANIEL P. BOVET AND MARCO CESATI: *Understanding the Linux Kernel*. 2006. – ISBN 9780596005658 2.5.3

[Derviş 2013]

DERVIŞ, Barış: *Mastering Embedded Linux Programming*. 2013. – 1689–1699 S. – ISBN 9788578110796 2.5.1, 2.5.2, 2.5.4, 2.5.5

[Leens 2009]

LEENS, Frédéric: An introduction to I2C and SPI protocols. In: *IEEE Instrumentation and Measurement Magazine* 12 (2009), Nr. 1, S. 8–13. <http://dx.doi.org/10.1109/MIM.2009.4762946>. – DOI 10.1109/MIM.2009.4762946. – ISSN 10946969 2.8, 2.3

[Linux-kernel]

LINUX-KERNEL: *ChangeLog-5 @ mirrors.edge.kernel.org*. <https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.10> 2.4

[**Petalinux 2020**]

PETALINUX: PetaLinux Tools Documentation Reference Guide. In: *Ug1144* 1144 (2020), 1–144. https://www.xilinx.com/support/documentation/sw-manuals/xilinx2019_1/ug1144-petalinux-tools-reference-guide.pdf 2.5.6, 2.6, 2.6.1

[**Projekt**]

PROJEKT, Yocto: *Yocto Project Documentation*. <https://www.yoctoproject.org/docs/latest/dev-manual/dev-manual.html> 3.3.1.4

[**Richards 2002**]

RICHARDS, Pat: A CAN Physical Layer Discussion. In: *Technology* (2002), S. 1–12 2.6, 2.7

[**Support 2022**]

SUPPORT, Xilinx: *Xilinx Support*. https://support.xilinx.com/s/article/1066813?language=en_US. Version: 2022 2.11, 2.12

[**Transmission u. a. 2018**]

TRANSMISSION, Message ; RECEPTION, Message ; OBJECTS, Mask ; FEATURES, Special: Mcp2517Fd Mcp2517Fd. (2018) 3.2.2

[**UG1137 2017**]

UG1137: Zynq UltraScale. In: *User guide 1137* (2017), 1–268. https://www.xilinx.com/support/documentation/user_guides/ug1137-zynq-ultrascale-mpsoc-swdev.pdf 2.5.6, 3.6

[**Xilinx Inc. a**]

XILINX INC.: *Xilinx zcu106 Evaluation Board*. <https://www.xilinx.com/products/boards-and-kits/zcu106.html> 3.2

[**Xilinx Inc. b**]

XILINX INC.: *zynq-ultrascale-mpsoc*. <https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html> 3.4, 3.1

[**Xilinx Inc. 2019**]

XILINX INC.: ZCU106 Evaluation Board. In: *Xilinx Technical Documentation* 1244 (2019), Nr. v1.4, S. 1–134 2.5.6.1, 2.5.6.2, 2.10, 2.5.6.3, 3.2.1, 3.5

Literaturverzeichnis

Ich, Hugues landry Nseupi Nono, Matrikel-Nr. 2022666, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

*Konfiguration und Optimierung des Embedded-Linux-Betriebssystem für
Automotive Image Processing Unit - Betreuer: Mladen Kovacev*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Augsburg, den 26. April 2022

HUGUES LANDRY NSEUPI NONO