HSP Projektbericht

Philipp Eidenschink, Florian Laufenböck, Tobias Schwindl
Matrikelnummern : 3080919, 2894759, 3080498

31. März 2017

Inhaltsverzeichnis

[1] Field Programmable Gate Array
[2] Intellectual Property
[3] Hard Processor System

Todo list

1 Einleitung

Dieser Projektbericht beschreibt die Tätigkeiten der Autoren im Laufe des HSP[4] im Wintersemester 2016/2017. Diese beinhalten im wesentlichen Folgendes: Die Ersetzung der kompletten Hardwarearchitektur des ALF und dessen Raspberry Pi auf eine neuere, verbesserste Hardwarearchitektur, um mehr Leistungsreserven zu besitzen.

Motivation für das Ersetzen des Raspberry Pi

- Leistungsreserven
- Flexibilität
- weils geht und cool ist!

1.1 Lesehinweise

Dieses Dokument ist in mehrere Kapitel gegliedert. Bevor dieses Dokument gelesen und verstanden werden kann folgen hier einige Lesehinweise:

- Coderepository - Der gesamte Code und alle relevante Dokumenation zu dem Projekt befindet sich aktuell auf Github. Der Link zum aktuellen Stand ist https://github.com/Alabamajack/Garfield.
- Weitere Dateien - Leider beschränkt Github die maximale Dateigröße auf 100MB. Aus diesem Grund liegen Dateien, die größer als 100MB sind, auf dem Laborlaufwerk unter dem Verzeichnis . Diese Daten werden im Dokument extra gennant.

Pfad zum Laborlaufwerk

- Pfade - Alle Dateipfade, die im Dokument genannt werden und für die keine weiteren Informationen angegeben sind, beziehen sich auf das root-Verzeichnis des Coderepositories. Weitere Pfade die verwendet werden sind:
  - Pfad im Linux Kernel: Diese Pfade sind absolute Pfade innerhalb einer bestimmten Version des Linux Kernels (nur Releases, keine Pre-Releases etc.). Solche Pfade haben den Prefix *LINUX_VX.X* wobei *X.X* die Version des Linux Kernel bezeichnet, der verwendet wurde.
  - Pfade auf dem HPS System - Dies sind Linux Distributionspfade. Alle verwendeten Pfade haben als root-Verzeichnis das *home*-Verzeichnis des Standardbenutzers *ubuntu*. Als Prefix dafür wird *HPS* genannt. Sollte ein übergeordneter Pfad zum *home*-Verzeichnis bezeichnet werden ist der Prefix *HPS_boot*.

[4] Hauptseminar Projektstudium

1.2 Eingesetzte Tools

Im folgenden Abschnitt soll ein kurzer Überblick über die verschiedenen eingesetzten Tools gegeben werden. Diese Beschreibung ist nicht vollständig da Grundkentnisse (wie kompilieren eines Linux Kernels aus den Sourcen) vorausgesetzt werden.

1.2.1 FPGA Entwicklung

1.2.1.1 Quartus

Das Tool Quartus bildet die Grundlage um für Altera (bzw. inzwischen Intel) FPGAs entwickeln zu können. Im Prinzip ist es eine Sammlung von verschiedenen Tools, die über eine GUI gesteuert werden. Alle relevanten Prozesse (Building, Generieren von IP-Cores, Systemanalyze etc.) sind auch (u.U. sogar mächtiger) als Kommandozeilentools verfügbar. Das vollständige Handbuch ist unter Quartus Handbuch zu finden (Achtung: 1939 Seiten!, und das ist nur der erste Teil). Als Überblick und um mit dem Garfield Projekt zu starten gibt es einige nützliche Links und Tutorials wie z.B. Altera University Programm - Start. Eine weitere, sehr empfehlenswerte Anlaufstelle bei Problemen oder auf der Suche nach Application Notes, Tutorials, HowTos und auch Vorträgen ist rocketboards.org. Dabei handelt es sich um die offizielle Open-Source Sammlung rund um Intel FPGAs.

Die OTH hat eine Reihe von Lizenzen für die gesamte Altera Toolchain (Quartus, SoC EDS, IP-Cores etc.). Wird das Garfield Projekt mit der unlenzierten Version syntethisiert, kompiliert Quartus automatisch einen “Ablaufzeitstempel”mit ins Design. Der NIOS2 Prozessor und einige IP-Cores sind damit nur ca. 30 Minuten lauffähig! Die Lizenzen verwaltet Herr Altmann. Dieser hat auch mind. einen WLAN-USB Stick an dessen MAC-Addresse die Lizenz gebunden ist. Es ist nicht empfehlenswert die Lizenz an eine feste MAC-Addresse eines privaten PCs zu binden, da diese Lizenz dann für andere Studenten verbraucht wäre. Die Installation von Quartus sollte im Hochschulnetz erfolgen, da die Downloadgröße ca. 20GB beträgt.

1.2.1.2 Qsys

Bei QSYS handelt es sich um Intels System Integrations Tool. Es ist eine sehr abstrakte Variante sich ein komplettes FPGA System zusammenzuklicken und automatisch jegliche Hardwarebeschreibungen, evtl. notwendige Treiber für den NIOS2 etc. zu generieren. Nach der generierung entsteht ein großer IP-Core, der abschließend in die eigene Pinbeschreibung eingebettett werden muss. Auch für dieses Tool kann am besten wieder auf ein Tutorial QSYS Tutorial oder auf rocketboards.org hingewiesen werden. Da das Garfield Projekt auch zwei selbstgeschriebenen Hardwarekomponenten beinhaltet, müssen die Pfade für diese QSYS bekanntgemacht werden. Dies ist nicht nur notwendig für die Person, die das Hardwaredesign anpasst/syntethisiert, sondern auch für Beteiligte, die

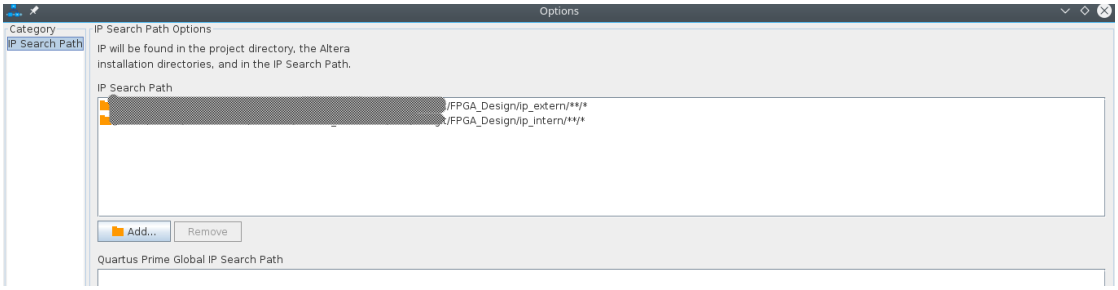

Abbildung 1.1: Einstellungen für die selbstgeschriebenen IP-Cores

Code für den NIOS2 schreiben wollen und auf die HAL[5]-Generierung des Tools angewiesen sind. Dazu müssen in QSYS die Einstellungen wie in Abbildung 1.1 dargestellt. Bei geöffnetem QSYS muss unter `Tools->Options->IP Search Path`.

1.2.1.3 Quartus Programmer

Der Quartus Programmer dient dazu, entweder das FPGA direkt mit der entsprechenden Image Datei (`*.sof`-Endung) oder das angeschlossene Flash mit einer `*.jic` Datei zu flashen. In den entsprechenden Quartus Tutorials sind auch kleine HowTos enthalten, wie mit dem Programmer umzugehen ist. Auf der für das Board zugeschnittene System-CD befindet sich auch eine `DE0-Nano-SoC_User_manual.pdf`. In diesem ist auch beschrieben, wie man eine Datei für den Flash erstellt und dies auf den Flash lädt.

[5]Hardware Abstraction Layer

2 Architektur

Hier wird die Architektur beschrieben, die dem ganzen Projekt zugrunde liegt.

2.1 Hardware

2.1.1 FPGA Design

Die Beschreibung des FPGA wird, soweit möglich, mit dem Systemintegrationstool QSYS, das Teil der Quartus Toolchain ist, durchgeführt. Das Mapping zwischen QSYS-System und Pins wird klassisch in VHDL beschrieben. Das Top-Level-File des Systems ist `FPGA_Design/Garfield_Design/Garfield.vhdl`. Dort wird das von QSYS erzeugt System und einige kleine IP-Cores zusammengeführt und auf definierte Aus-/Eingänge geführt. Diese Ein-/Ausgänge werden dann über den *Pin-Planner* auf die physikalischen Pins geführt.
Im Projektverzeichnis befinden sich alle Dateien, die für den FPGA Teil relevant sind, unter `FPGA_Design`. Die Struktur ab diesem Ordner ist wie folgt aufgebaut:

- `Datasheets` - Einie Datenblätter und Application Notes zu dem FPGA Teilprojekt
- `Garfield_Design` - In diesem Ordner befinden sich die Quartus Projektdateien, Konfigurationsdateien und das QSYS Projekt.
- `ip_extern` - Eine Sammlung von externen IP-Cores, die im Projekt verwendet wurden. Es befinden sich dort nur die IP-Cores, die nicht von Altera stammen oder nicht direkt in QSYS verfügbar sind.
- `ip_intern` - Alle IP-Cores, die für dieses Projekt entwickelt wurden.
- `output_files` - In jedem Unterordner innerhalb dieses Ordners befinden sich FPGA Images und die entsprechenden Konfigurationsdateien um ein Softwareprojekt dafür zu bauen.

Im folgenden werden die alle Systemkomponenten, die für das Garfield-Projekt erzeugt wurden, beschrieben.

2.1.1.1 IP-Cores

Abbildung 2.1 zeigt eine Übersicht der eingesetzten IP-Cores und deren Verbindung zur Außenwelt. Ausgenommen sind die IP-Cores, die für die Kommunikation mit dem HPS benötigt werden. Die nachfolgende Tabelle beschreibt die Funktion der einzelnen IP-Cores im Detail und Besonderheiten dazu.

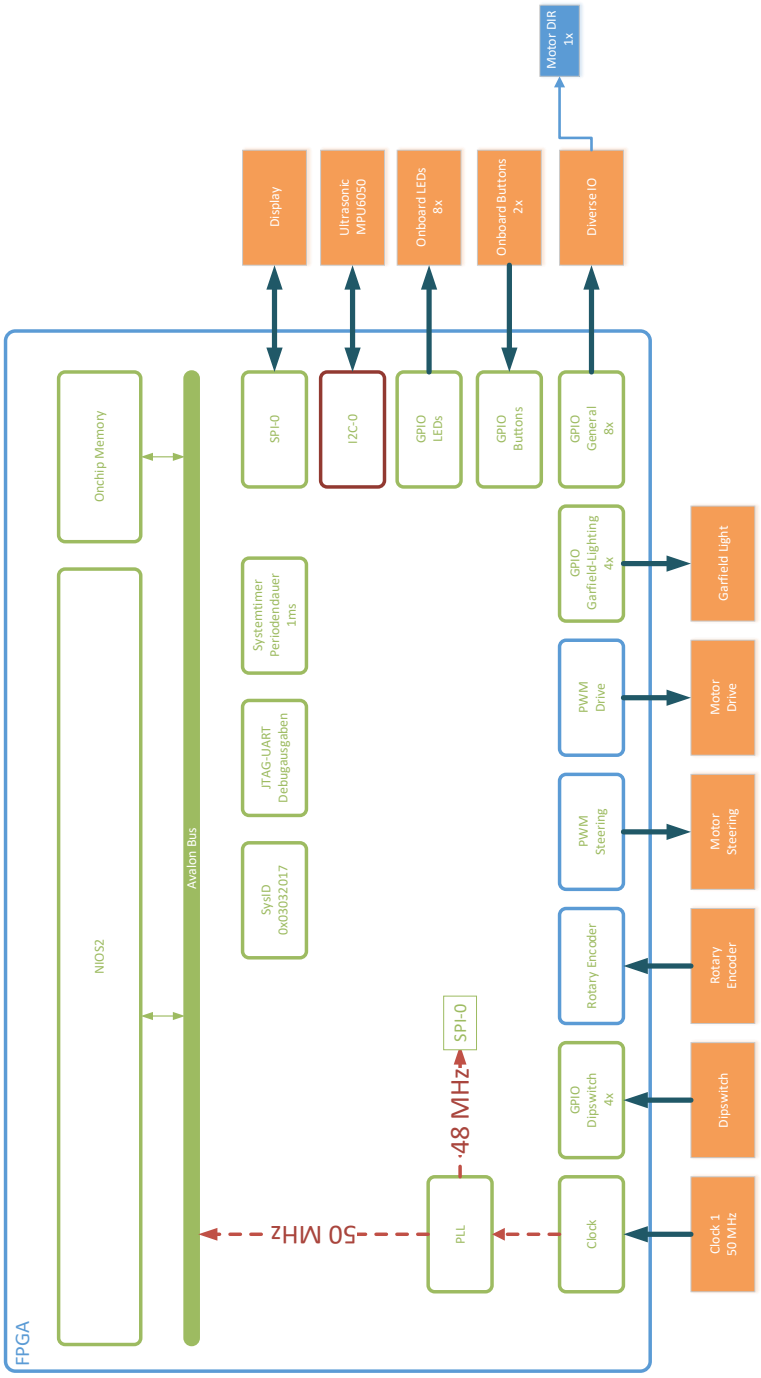

Abbildung 2.1: Übersicht der (meisten) eingesetzten IP-Cores. Die Abbildung verzichtet auf die Darstellung der Teile, die für die Kommunikation mit dem HPS zuständig sind. Blau umrandet sind Cores, die im Rahmen des Projekts selbst progammiert wurden, grün diejenigen, die Teil der Altera Toolchain sind und rot externe IP-Cores von opencores.org

Name	Beschreibung
SPI-0	Stellt einen SPI-Datenbus mit 24MHz Clock-Frequenz zur Verfügung. Es werden insgesamt 3 Chipselect Signale zur Verfügung gestellt, wobei aktuell nur eines für das Display benutzt wird. In der aktuellen Ausbaustufe wird nur das Display auf dem Arduino-Header auf dem FPGA angesteuert.
I2C-0	Stellt einen I2C Datenbus zur Verfügung. Der IP-Core stammt von opencores.org und wurde manuell integriert. Er stellt u.a. eine eine in Software änderbare Clock-Frequenz zur Verfügung und bindet die Ultraschallsensoren und die MPU-6050 an das System an.
GPIO-X	Die verschiedenen GPIO Cores dienen dazu einfache Peripherie anzubinden. Dazu gehören die LEDs, die Dip-Switches, die Buttons und generische IOs, die im Projekt benötigt werden um z.B. die Drehrichtung des Motors einzustellen.
PWM X	Die beiden PWM IP-Cores erzeugen Signale zur Geschwindigkeitssteuerung und für den Lenkmotor.
Rotary-Encoder	Der Rotary Encoder zählt die steigenden Flanken der NAME. Durch Abfragen des Ergebnisregisters in regelmäßigen festen Zeitinervallen kann die aktuelle Geschwindigkeit, die an den Rädern anliegt, gemessen werden. Leider funktioniert der NAME aktuell nicht mehr. Um das Signal zu nutzen müsste man die Hardware neu aufbauen bzw. ersetzen.
Clock & PLL[6]	Die externe Referenzclock taktet mit 50 MHz. Dieses Signal wird über eine PLL allen beteiligten IP-Cores bereitgestellt. Auch die FPGA-HPS Bridges werden mit dem 50MHz Signal gespeist. Einzige Ausnahme bildet der SPI-0 Core. Um eine Frequenz von 24MHz zu erreichen (die maximale Frequenz mit der das Display angesprochen werden darf) wird ein vielfaches dieser Frequenz benötigt. Das nächsthöhere verfügbare vielfache der 24MHz sind 48MHz. Die selbst geschriebenen IP-Cores sind von der Frequenz der PLL abhängig. Erhöht man die Frequenz der PLL auf z.B. 100MHz um mehr Laufzeit für einzelne Funktionen zur Verfügung zu haben, muss man die Frequenz in den IP-Cores manuell anpassen!
SysID	Mit der System ID (in Kombination mit einem Zeitstempel) kann man das Hardware Design eindeutig identifizieren. Dies ist hilfreich wenn mehrere Hardware- und Softwareversionen existieren, die parallel entwickelt werden. Um Zugriffsfehler auf Register oder ähnliches zu vermeiden, kann die Software die System-ID nutzen um Funktionen ab- bzw. zuzuschalten.
JTAG-UART	Mit Hilfe dieses Cores kann man printf ähnliche Ausgaben für Debug-Ausgaben an einen angesteckten PC schicken.
Systemtimer	Der Systemtimer ist ein kontinuierlich laufender Timer, der sich alle 1ms automatisch erhöht. Außerdem erzeugt er ein Interrupt, das FreeRTOS zur internen Zeitbestimmung nutzt.

[6] Phase-locked loop

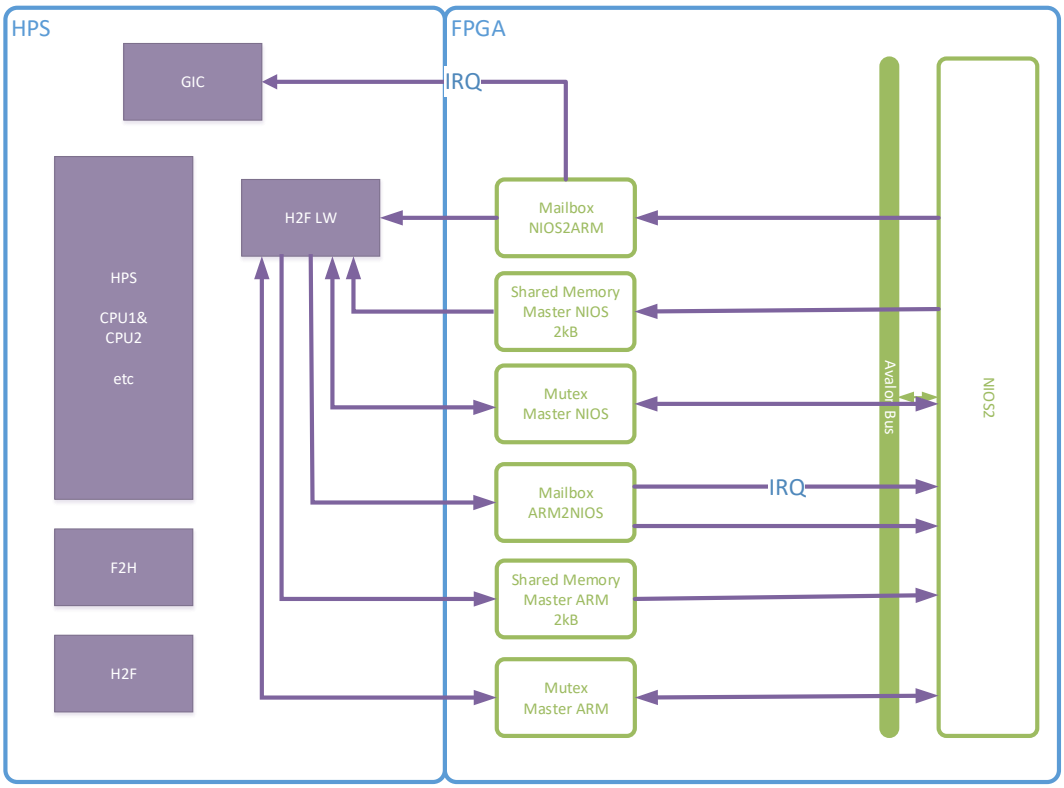

Abbildung 2.2: IP-Cores und deren Kontrollfluss, die an der Kommunikation zwischen NIOS2 und HPS beteiligt sind.

NIOS2	Hierbei handelt es sich um eine Softcore-CPU. Diese wird von Altera zur Verfügung gestellt (inkl. Toolchain) und kann unbegrenzt benutzt werden (mit entsprechender Lizenz). Es handelt sich um eine 32-bit **RISC!**[7] Architektur die durchaus eine weite Verbreitung im industriellen Umfeld genießt. Weiter Informationen dazu findet man unter https://www.altera.com/products/processors/overview.html
Onchip Memory	Ein einfacher IP-Core, der Speicherbausteine auf dem FPGA nutzt um RAM(hier genutzt) oder ROM (nicht genutzt) zu erzeugen. Dieser Speicher kann dann von einem Prozessor (hier der NIOS2) als Instruktions- und Datenspeicher genutzt werden. In der aktuellen Ausbaustufe ist die Speichergröße mit 128kB angegeben.

NAME

Abbildung 2.2 zeigt die IP-Cores, die für die Kommunikation zwischen FPGA und

[7] **RISC!**

HPS benötigt/eingesetzt werden. Auf der linken Seite der Abbildung ist das HPS System illustriert. Auf dieser Seite sind im wesentlichen drei Hardwareeinheiten an der Kommunikation beteiligt:

- GIC - Der ARM *General Interrupt Controller* : Dieser Controller ist ein sehr mächtiger Interrupt Controller, der u.a. die Interruptverarbeitung an die einzelnen CPUs verteilt. Insgesamt stehen 64 Interrupts zur Verfügung, die aus dem FPGA heraus ausgelöst werden können. Auf dem eingesetzten Cyclone V beginnen diese mit der Interrupt ID 72 vom GIC. Genauere Informationen zum GIC kann man entweder auf der Homepage von ARM oder [1] erhalten.

- H2F LW - Die HPS2FPGA Leightweight Bridge : Dies ist eine der drei Bridges, mit denen zwischen FPGA und HPS kommuniziert werden kann. Dies ist keine High-Performance Bridge, es ist aber keine großen Änderungen notwendig, das System auf eine der anderen Bridges umzubauen. Diese Bridge **muss** aktiviert werden, bevor über sie kommuniziert werden kann. Ist die Bridge nicht aktiviert, treten *Segmentation Faults* auf (kein gültiger Speicherbereich). Im Prinzip befindet sich "hinter"der Bridge ein Speicherbereich, der durchgehend addressiert werden kann um direkt in Register zu schreiben. Auch lesende Zugriffe daraus können erfolgen [2].

- CPUs - Die ARM A9 Applikationsprozessoren dienen zur Verarbeitung der Interrupts bzw. zum triggern der einzelnen IP-Cores.

Es folgt eine Beschreibung der IP-Cores, die für die Kommunikation gebraucht werden. Da die Kommunikationseinheiten in beide Richtungen analog aufgebaut sind, beschränkt sich die Beschreibung auf einen Teil:

- Mailbox X2Y : Die Mailbox ist ein einfacher IP-Core der Nachrichten von einem Buspartner (X, z.B. NIOS2) einem anderen Buspartner (Y, z.B. ARM) zur Verfügung stellt. Es gibt also einen Transmitter und einen Receiver. Beide sind über eigene Interfaces (und damit über ihren eigenen Addressbereich) an die Mailbox angeschlossen. Die Nachrichtenübermittlung erfolgt mit Hilfe von zwei Registern:
  - Command Register - Dieses Register kann vom Empfänger nur gelesen werden. Es dient dazu, ein Kommando oder Nachricht an den Empfänger zu senden. Ein schreibender Zugriff auf dieses Register vom Sender löst das zugehörige Interrupt aus, dass vom Empfänger verarbeitet werden muss.
  - Pointer Register - In diesem Register wird die Addresse, in der die eigentliche Nachricht im Speicher steht, übertragen. Sollen nur ganz kleine Nachrichten (4 oder 8 Bytes) übertragen werden, kann man das Pointer und Command Register dazu benutzen, die Nachricht zu übertragen. In diesem Projekt wird aber die eigentliche Nachricht im Shared Memory übertragen, in der Mailbox nur die Addresse im Shared Memory und ein Kommando im Command Register

  Eine detailierte Beschreibung des Cores findet sich unter [5, 470ff]

- Shared Memory Master X - Dieser Speicher, der wie der Arbeitsspeicher des NIOS2 direkt im FPGA synthetisiert wird, dient der Nachrichtenübermittlung. Dort werden die Nutzdaten einer Nachricht von X reingeschrieben und können zu einem späteren Zeitpunkt vom Empfänger Y ausgelesen werden. Es wurde sich bewusst dazu entschieden zwei Shared Memory zu benutzen um eine jegliche Kollision zu vermeiden bzw. zu vereinfachen. Die Größe beider Speicherbereiche beträgt jeweils 2kB. Dies reicht für die aktuellen Nachrichten leicht aus. Zu einem späteren Zeitpunkt können die Bereiche auch noch vergrößert werden, sollte der Speicher nicht groß genug sein Nachrichten zu übertragen.

- Mutex Master X - Dies ist ein spezieller IP-Core, der auch als Teil des Altera IP-Core Katalogs zur Verfügung stellt. Dieser hat nur ein Register, das hier betrachtet werden soll und erlaubt einen atomaren Mutex Zugriff auf geteilte Resourcen. Die geteilte Resource, die über diesen Mutex gesperrt wird ist der zugehöriger Shared-Memory. Die Referenz für diesen Core ist ebenfalls [5, 319ff]. Das Register besteht aus zwei Teilen: die oberen 16 Bit werden als Speicherplatz für die CPU-ID (der NIOS2 hat die ID 0x03, der ARM immer 0x01) benutzt. In die unteren 16 Bit kann ein beliebiger Wert gespeichert werden. Ein lesender Zugriff auf den Mutex ist immer möglich. Ein schreibender Zugrif ist nur möglich wenn
  - Die CPU-ID mit der CPU-ID übereinstimmt, dessen Wert man in das Register schreiben will.
  - (oder) Der Wert (untere 16-Bit) Null ist.

  Man kann also nur schreibend auf den Mutex zugreifen, wenn einem der Mutex bereits gehört oder der Mutex frei (=0) ist. Nach einem schreibenden Zugriff muss der Registerwert mit dem Wert der geschrieben wurde verglichen werden. Stimmen beide Werte überein, hat der Schreiber den Mutex gelockt, andernfalls ist der atomare Lock fehlgeschlagen.

2.1.1.2 Address-Map

Abbildung 2.3 zeigt die Addressen die im System benutzt werden und den zugehörigen Addressbereich der verfügbar ist. Diese Addressmap ist auch im QSYS-Projekt des Projektes verfügbar.

2.1.1.3 Eigenentwickelte IP-Cores

Der PWM-Generator generiert ein PWM Signal auf die Ausgabeleitung. Der Registerzugriff ist in Tabelle 2.3 dargestellt

Der Rotary Encoder IP-Core kann steigende Flanken von einer externen Flanke zählen, hat ein auslesbares Ergebnisregister (siehe 2.7)und ein Controlregister (siehe 2.5).

System: Garfield_system **Path:** mailbox_arm2nios_0

	fpga_only_master.master	...	hps_0.h2f_lw_axi_master	hps_only_master.master	nios2_gen2_0.data_mas...	nios2_gen2_0.instruction_...
timer_0_nios2.s1					0x0000_0000 - 0x0000_001f	
spi_0.spi_control_port					0x0000_0020 - 0x0000_003f	
Garfield_lighting.s1					0x0000_0060 - 0x0000_006f	
Garfield_GPIO.s1					0x0000_0070 - 0x0000_007f	
Drive_PWM.avalon_slave_0					0x0000_0080 - 0x0000_0087	
Steering_PWM.avalon_slave_0					0x0000_0090 - 0x0000_0097	
Rotary_Encoder_0.avalon_slav...					0x0000_00a0 - 0x0000_00a7	
mailbox_arm2nios_0.avmm_m...					0x0000_00c0 - 0x0000_00cf	
i2c_opencores_0.avalon_slav...					0x0000_8000 - 0x0000_801f	
sysid_fpga.control_slave	0x0001_0000 - 0x0001_0007		0x0001_0000 - 0x0001_0007		0x0001_0000 - 0x0001_0007	
mailbox_nios2arm_0.avmm_m...					0x0001_0030 - 0x0001_003f	
Onboard_LED.s1					0x0001_0050 - 0x0001_005f	
Onboard_DipSW.s1					0x0001_0080 - 0x0001_008f	
Onboard_Button.s1					0x0001_00c0 - 0x0001_00cf	
onchip_memory2_nios2.s1					0x0002_0000 - 0x0003_ffff	0x0002_0000 - 0x0003_ffff
nios2_gen2_0.debug_mem_sl...					0x0004_0800 - 0x0004_0fff	0x0004_0800 - 0x0004_0fff
jtag_uart_nios2.avalon_jtag_sl...					0x0004_1000 - 0x0004_1007	
shared_memory_mutex_mast...			0x0005_0000 - 0x0005_0007		0x0005_0000 - 0x0005_0007	
shared_memory_master_hps_...			0x0006_0000 - 0x0006_07ff		0x0006_0000 - 0x0006_07ff	
shared_memory_mutex_mast...			0x0008_0000 - 0x0008_0007		0x0008_0000 - 0x0008_0007	
shared_memory_master_nios_...			0x0009_0000 - 0x0009_07ff		0x0009_0000 - 0x0009_07ff	
hps_0.f2h_axi_slave				0x0000_0000 - 0xffff_ffff		
mailbox_arm2nios_0.avmm_m...			0x0002_0000 - 0x0002_000f			
mailbox_nios2arm_0.avmm_m...			0x0007_0000 - 0x0007_000f			

Abbildung 2.3: Übersicht über die Addressen und Addressbereiche im System

Bit	Name	Access	Reset Value	Description
7 ... 0	control	RW	0	sets the dutycylce of the PWM signal generator
31 ... 7	-	R	0	not used

Tabelle 2.3: Registermap des PWM Cores

2.2 Software

Die Software unterteilt sich insgesamt in 3 Teile.

- Headquarter (HQ): Linux System das mit dem Fahrzeug über WLAN kommuniziert
- ARM: Linux ARM System, dass die Netzwerkaufgaben, sprich Kommunikation, mit dem HQ übernimmt
- NIOS: VHDL Core, der sonstige Periphere anspricht auf dem ein Echtzeitbestriebsystem (FreeRTOS) läuft. Die Software hiervon setzt sich zusammen aus /Software/common/ARM_NIOS_HQ/ und Software/Software_NIOS2/*.

2.2.1 ARM

Comm Gateway beschreiben

2.2.1.1 Mailbox Kommunikation ARM ↔ NIOS2

Name, Pfad?

Die Kommunikation über die in das FPGA programmierte Mailbox (siehe 2.1.1.1) wird über eine abstrakte Klassenimplementierung in C++ dargestellt. Die Klasse stellt einige *Write* und *Read* Funktionen zur Verfügung, die mit verschiedenen Objekten umgehen können. Durch dieses Vorgehen ist sichergestellt, das der Aufrufer als Übergabeparameter nur bestimmte Objekte (die sauber definiert sind) übergegeben werden können,

Bit	Name	Access	Reset Value	Description
0	enable	RW	0	Enable bit for the core
1	clear	W	0	Clear bit. clears the result register and set it to 0; Must not be manually set to 0 after clearing. With the next rising edge of the clock it goes down on itself.
2	reset	W	0	Resets the whole core and set all values to default. At a read operation, it is always 0
15 ... 3	not accessable	-	0	-
16	error	R	0	Indicates an error within the counting process. You should reset the core!
31 ... 17	not accessable	-	0	-

Tabelle 2.5: Registermap des Controlregisters des Rotary Encoder

Bit	Name	Access	Reset Value	Description
31 ... 0	result	R	0	Result of the counting process

Tabelle 2.7: Registermap des Ergebnisregisters des Rotary Encoder

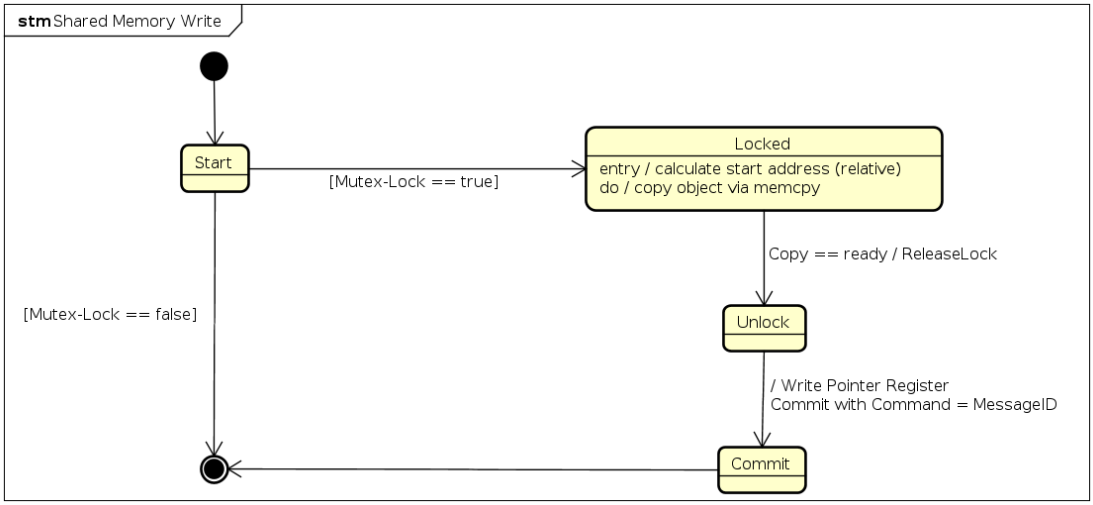

Abbildung 2.4: Schreibvorgang in den Shared Memory

die auch verarbeitet werden können. Die Kommunikation erfolgt asynchron und ist intern gepuffert. Die Klasse dient als Abstraktionsschicht in beide Richtungen. Sowohl die Empfangsrichtung als auch die Senderichtung werden über die Klasse abgebildet, sodass der Aufrufer keinerlei interne Informationen über die Hardware und die Implementierung haben muss. Nachfolgend werden die beiden wesentlichen Operationen (*Write* und *Read*) dargestellt und beschrieben.

Write Der schreibende Zugriff auf den Shared Memory ist in Abbildung 2.4 illustriert und hat einen einfachen Ablauf. Nachdem sich der Schreiber den Lock auf den Shared Memory über den Mutex Core geholt hat kann er Daten in diesen Bereich schreiben. Der Speicher wird dabei linear durchlaufen. Der Speicher wird dabei wie ein Ringspeicher behandelt. Ist am oberen Ende des Speichers nicht mehr genug Platz für die neue Nachricht wird wieder bei der relativen Addresse Null angefangen zu schreiben. Sind alle Daten in den Speicher geschrieben wird der Mutex wieder freigelassen. Im Speicher steht jetzt ein Abbild des Objekts, das übertragen werden soll. Es werden also nur Nutzdaten in den Speicher geschrieben. Anschließend wird die Startaddresse der Daten in das Pointer Register der Mailbox geschrieben. In das Command Register wird die eindeutige Nachrichten ID, die innerhalb des Garfield Projekts definiert ist, geschrieben. Dieser Schreibvorgang ist die letzte Instruktion zum Nachrichtenaustausch aus Sendersicht.

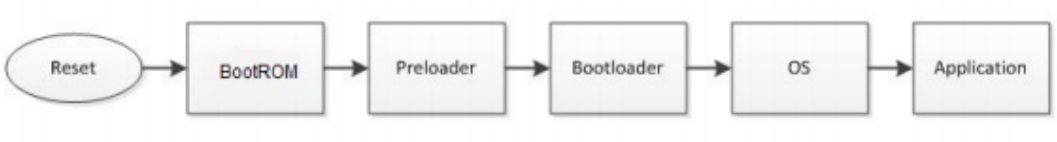

Abbildung 2.5: Typsicher Bootvorgang des ARM A9 Dualcore Prozessors [4]

Read Der lesende Zugriff auf den Shared Memory ist komplizierter und unterteilt sich in zwei Abschnitte.

Der erste Teil der lesenden Kommunikation besteht aus dem **Interrupt**. Von dem Mailbox IP-Core wird bei einem schreibenden Zugriff auf das Command Register automatisch ein Interrupt erzeugt. Innerhalb des *ReadInterruptHandler* wird nicht der Shared Memory ausgelesen, sondern nur die Nachricht aus der Mailbox gespeichert. Die Klasse hält zusätzlich zu jedem Objekt, das verschickt bzw. empfangen werden soll einen kleinen Ringpuffer. Dieser dient dazu, die Addressen der Objekte im Shared Memory zu puffern. Tritt nun das Interrupt auf, wird zuerst das Pointer Register zwischengespeichert. Anschließend wird das Command Register, in dem der Nachrichtentyp übertragen wird, ausgelesen und anhand des Nachrichtentyps die Addresse des Pointer Registers in den passenden Puffer geschrieben. Dieses Vorgehen sorgt dafür, dass diese Funktion, die in einem Interrupthandler ausgeführt wird, sehr schnell wieder beendet ist.

Der zweite Teil besteht aus einem **Hintergrundtask** der zyklisch durchlaufen wird. Innerhalb dieses Task werden die Werte eines Objekts, die ausgelesen werden sollen, über eine *Read* Funktion bei Bedarf überschrieben. Bei Bedarf deswegen, weil die Werte nur überschrieben werden, wenn aktuellere Werte im Shared Memory stehen. Sind aktuelle Werte verfügbar, ist der Klasseninterne Ringpuffer, der die Addressen für die Nutzdaten enthält, nicht leer. Es wird über die Addresse aus dem Shared Memory gelesen und anschließend diese Nachricht (also die Addresse) aus dem Ringpuffer entfernt.

2.2.1.2 HPS Startvorgang und Software

Der folgende Abschnitt beschreibt die Bestandteile die für den Betrieb des ARM Dualcore notwendig sind und wie diese konfiguriert und kompiliert werden.

Eine sehr gute und übersichtliche Beschreibung des Bootvorgangs der A9 Kerne ist in [4] beschrieben. Der im GarfieldProjekt verwendeter Bootvorgang ist in Abbildung 2.5 dargestellt. Die ersten beiden Schritte (BootRom, Preloader) sollen hier nicht weiter beschrieben werden da keine manuelle Anpassung daran notwendig ist. Als Referenz für den Buildvorgang des gesamten Systems diente https://eewiki.net/display/linuxonarm/DE0-Nano-SoC+Kit. Eine komplette Kopie des Tutorials befindet sich auch im Projektverzeichnis unter . Wenn Änderungen an der im Tutorial beschriebenden Vorgehensweise notwendig sind werden diese erwähnt.

Pfad

Als Bootloader kommt der beliebte U-Boot [8] in einer leicht angepassten Variante zum Einsatz. Wie im Tutorial beschrieben werden einige Startvariablen(unter anderem das zu ladende Linux Device Tree Binary) hinzugefügt um von der SD Karte zu booten. U-Boot lädt nach dem Start dann automatisch zunächst den

Linux Device Tree Der Linux Device Tree ist ein Bestandteil des Linux Kernels um eine Abstraktionsschicht zwischen Hardware (Pinouts, Speicher, Interrupts) und dem Linux Kernel zu schaffen. Durch Einsatz des Device Tree kann ein gleiches Kernel Binary auf verschiedenen Hardwareversionen und sogar komplett verschiedenen Boards benutzt werden. Der Device Tree ist eine textuelle Beschreibung der Hardware die kompiliert wird und dem Kernel beim Startvorgang übergeben wird. Ein Auszug aus einem solchen Device Tree zeigt der Code 2.1. Die Spezifikation des Device Trees kann man unter http://www.devicetree.org/specifications/ einsehen.

Listing 2.1: Auszug aus socfpga.dtsi [6, Version 4.7, `arch/arm/boot/dts/socfpga.dtsi`]

```
cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        enable-method = "altr,socfpga-smp";

        cpu@0 {
                compatible = "arm,cortex-a9";
                device_type = "cpu";
                reg = <0>;
                next-level-cache = <&L2>;
        };
        cpu@1 {
                compatible = "arm,cortex-a9";
                device_type = "cpu";
                reg = <1>;
                next-level-cache = <&L2>;
        };
};
```

Listing 2.2: Notwendige Änderungen am Device Tree

```
&fpga_bridge0 {
        bridge-enable = <1>;
};

&fpga_bridge1 {
        bridge-enable = <1>;
};
```

[8] https://www.denx.de/wiki/U-Boot

```
&fpga_bridge2 {
        bridge-enable = <1>;
};

/* we are extending the soc device with a specific interrupt! */
/{
        soc {
                mbox_rx: mailbox@0x00070000 {
                        compatible = "altr,mailbox-1.0";
                        reg = <0x70000 0x8>;
                        interrupt-parent = < &intc >;
                        interrupts = <GIC_SPI 60 4>;
                        #mbox-cells = <1>;
                };
        };

};
```

Die Änderungen, die für den Device Tree im GarfieldProjekt nötig sind, sind im Codeausschnitt 2.2 gezeigt. Es existiert außerdem eine patch-Datei, mit der der geänderte Device Tree direkt in die heruntergeladenen Kernelsourcen gepatcht und kompiliert werden kann. Die vorgenommen Änderungen ergeben sich wie folgt

- fpga_brigdes - Durch die hinzugefügten Einträge `fpga_bridgeX` werden die verschieden verfügbaren (Achtung: Ist nur für Kernel Version 4.7 so gültig, ältere Kernel haben u.U. weniger verfügbare Bridges) Bridges beim Laden des Device Tree aktiviert.

- mbox_rx - Mit diesem Eintrag wird das Interrupt, ausgelöst durch den Mailbox IP-Core (siehe 2.1.1.1) als verfügbare Hardwareeinheit dem Kernel bekanntgemacht. Interessante Attribute sind
  - `@` - Die Zahl nach dem @ entspricht der Physikalischen Addresse der Mailbox. Da die Adresse im weiteren Verlauf nicht verwendet wird ist der exakte Wert nicht von Bedeutung.
  - `compatible` - Mit diesem Attribut wird dem Interrupt ein Name gegeben. Wäre im Kernel ein generischer Hardwaretreiber für diese Mailbox vorhanden könnte dieser zur Verarbeitung genutzt werden. Wichtig für das Projekt ist der Name trotzdem, da damit das Zuordnen der Interrupt-Service-Routine zu dem Interrupt geschieht.
  - `interrupt-parent` - Damit wird der Interrupt Controller (in unserem Fall der ARM GIC, der in einer inkludierten Beschreibungsdatei beschrieben wird) identfiziert, der das Interrupt aufnimmt und die weitere Abarbeitung in die Wege leitet.

  - `interrupts` - Damit wird das Interrupt, das am GIC ausgelöst wird (Nummer 60) bekannt gemacht. Die Nummer ergibt sich aus:
    * Das 60te Interrupt der *Shared Peripheral Interrupt* am GIC entspricht der 92ten Interruptleitung am GIC.
    * Die 92ten Interruptleitung entspricht der 20ten Interruptleitung, die vom FPGA verfügbar ist.

    [3].

Die Änderungen werden direkt in die Datei arch/arm/boot/dts/socfpga_cyclone5_de0_sockit.dts geschrieben.

Linux wird als Betriebssystem von U-Boot geladen. Im Projekt kommt eine leicht angepasste Version der im Tutorial beschriebenen Linux Version vor. Die Änderungen sind marginal und betreffen:

- Das USB-Subsystem - Zum Betrieb des Lidar[9] ist es notwendig einige Punkte des USB Subsystem während des Linux Kompiliervorgang (make menuconfig) auszuwählen und zu kompilieren. Im groben handelt es sich um die USB-OTG Unterstützung, das ACM Subsystem und noch einige kleinere Änderungen. Auch für diesen Schritt existiert eine patch Datei im Verzeichnis . Pfad
- Das `uio` Modul - Mit diesem Modul ist es möglich mit Interrupts im User-Space von Linux zu arbeiten. Dazu wird das Modul mit dem Namen, der im Device Tree angegeben wurde altr ,mailbox−1.0 geladen. Dieses Modul erzeugt anschließend ein Gerät unter `/dev`, dass dann im User-Space genutzt werden kann. Auch diese Änderung ist in der patch Datei mit angegeben.

Als Distribution kommt eine minimale Version von Ubuntu zum Einsatz (Alternativ kann auch Debian verwendet werden). Diese Distribution wurde gewählt um einen kleine Footprint (im Leerlauf insgesammt nur ca. 35MB Arbeitsspeicherverbrauch) mit dem Komfort einer “normalen”Linux Distribution inklusiver einem Softwarerepository zur einfachen Installation von Software zu verbinden. Sobald Linux gestartet ist kann die (hier) entscheidene Applikation

Comm-Gateway gestartet werden. Das Kommunikationsgateway ist die Schnittstelle um zwischen einem Host-PC und dem NIOS2 kommunizieren zu können. Die Sourcedateien befinden sich unter `Software/Software_ARM/Gateway`. Dieses Programm wird später auch ausgeführt um Daten austauschen zu können. Neben der Initialisierung der Hardware, dem Öffnen eines Serverports, um sich mit dem Client zu verbinden und Daten auszutauschen, besteht dieses Programm aus insgesamt drei Threads:

- readData - Dieser Thread behandelt den Nachrichtenaustausch in Richtung Host PC → NIOS2. Der Thread wartet auf eingehenden Nachrichten (im Moment nur

[9] Light detection and ranging

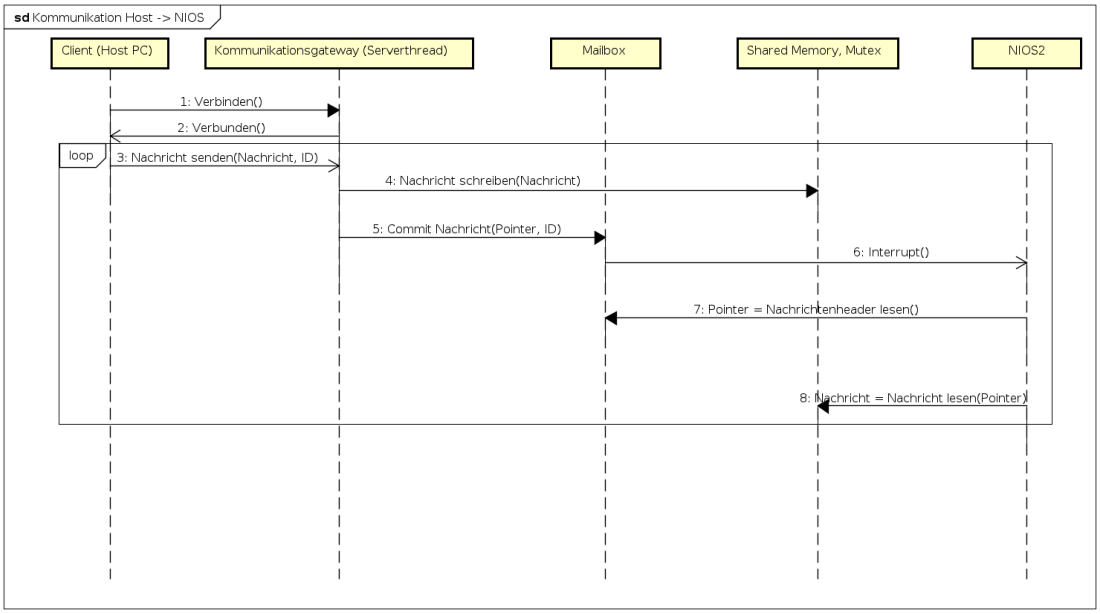

Abbildung 2.6: Kommunikationsablauf zwischen Host PC und NIOS

Fahrkommandos), parst und castet diese in das entsprechende Objekt und schreibt das Objekt dann über die Klasse in den Shared Memory mit dem NIOS2. Dieser Kommunikationsweg ist auch in Abbildung 2.6 dargestellt.

KLass

- writeData -
- HardwareReadHandler -

2.2.1.3 Empfohlener Buildvorgang und Abweichungen zum Tutorial

Der schnellste und unkomplizierteste Weg zu einem funktionierenden Linux Image auf SD-Karte ist folgender

1. Das Tutorial unter https://eewiki.net/display/linuxonarm/DE0-Nano-SoC+Kit mit einer beliebigen (Mindestgröße 4GB) SD Karte durchführen. Als Kernelversion sollte unbedingt die Version 4.7 gewählt werden, da sich die im Projektrepository befindlichen Dateien darauf beziehen. Nach diesem Schritt kann man den Bootvorgang von U-Boot und Linux testen und versuchen sich im Ubuntu einzuloggen. Anschließend werden die Änderungen durchgeführt.

2. Den Linux Devcie Tree Patchen. Dazu in einem Linux-Terminal

```
patch socfpga-kernel-dev/KERNEL/arch/arm/boot/dts/
    socfpga_cyclone5_de0_sockit.dts
    socfpga_cyclone5_de0_sockit_garfield.patch #dieses
    Verzeichnis ist nach dem Tutorial verfuegbar
```

3. Das Linux Konfigurationsfile patchen

   ```
   patch socfpga-kernel-dev/KERNEL/.config Garfield_Kernel.patch)
   ```

4. Anschließend über das Skript `socfpga-kernel-dev/tools/rebuild.sh` (oder manuell über den entsprechenden make-Befehl) den Kernel inkl. Device Tree neu kompilieren

5. Den Linux Kernel und die Device Trees wie im Tutorial (die Punkte `Copy Kernel Image` und `Copy Kernel Device Tree Binaries`) auf die SD Karte kopieren.

Anschließend kann man die SD-Karte wieder entnehmen und das FPGA wieder einschalten. Das System sollte wie gewohnt starten. Um zu überprüfen, ob die Operationen funktioniert haben sollten folgenden Befehle die gleiche Ausgabe wie in Listing haben.

Verweis

Listing einfügen, module uio und enable bridge enable

2.2.1.4 FPGA Programmierung

Die Programmierung des FPGA Subsystems erfolgt unmittelbar während des Systemstarts und ist aktuell vom Softwarestart abgekoppelt. Das Image für das FPGA wird dabei aus dem auf dem Board integrierten Flash-Speicher geladen. Die möglichen alternativen Konfigurationen sind übersichtlich im User-Manual zu dem `DE0-Nano` Board aufgelistet. Das Manual dazu findet man auf der System CD, die unter Link verfügbar ist.

Auf dem Flash befindet sich sowohl das Image für das FPGA als auch das executable für den nach dem flashen im FPGA verfügbaren NIOS2.

Kopie?

2.2.1.5 Applikationsstart

Für die Kommunikation zwischen Host PC und NIOS2 ist das starten des Kommunikationsgateway erforderlich. Dies befindet sich unter `HPS/bin/Comm_Gateway` und ist das Kompilat des oben beschriebenen . Damit das Programm Ordnungsgemäß funktionieren kann müssen noch einige Schritte durchgeführt werden, die hier kurz erklärt werden.

Name

Listing 2.3: Listing

```
sudo -s # superuserrechte erlangen, Passwort = temppwd
rmmod uio_pdrv_genirq
rmmod uio
modprobe
ls /dev/
bin/Comm_Gateway
```

Der Codeausschnit 2.3 zeigt alle durchzuführenden Schritte, die notwendig sind um das Kommunikationsgatway ordnungsgemäß zu starten. Die ersten beide Befehle entfernen evtl. geladenene Module, die beim startup mit falschen Parametern geladen werden. Der dritte Befehl lädt das Linux-Modul um mit Interrupts im User-Space umgehen zu

können und erstellt dafür ein Gerät mit dem Namen `HSP_boot/dev/uio0`, dass im Kommunikationsgatway verwendet werden. Hier ist auch der Name, der im Linux Device Tree angegeben wurde, von Bedeutung, da über diesen das richtige Interrupt ausgewählt wird. Mit dem vierten Befehl sollte vor Ausführung des Gateways überprüft werden ob das Gerät `HSP_boot/dev/uio0` auch wirklich verfügbar ist. Ist dies der Fall, kann die Applikation mit Superuserrechten gestartet werden. Die Reihenfolge ist dabei

1. Starten des FreeRTOS auf dem NIOS2 (wird i.d.R. automatisch bei Systemstart durch Laden aus dem externen Flash durchgeführt).
2. Konfigurieren des Linux und starten der Applikation (eine Serververbindung wird angeboten).
3. Anschließendes Verbinden der Applikation auf dem Host-PC mit dem Server. Daten werden ab diesem Zeitpunkt ausgetauscht.

2.2.1.6 Beenden der Applikation

Aktuell ist keinerlei Fehlerbehandlung für das Beenden der Client-Server Verbindung (Host-PC $\leftrightarrow$ HPS) vorhanden. Beendet der Client die Kommunikation hängt sich das gesamte System auf (Grund: Interrupts werden nicht mehr angenommen und der interne Bus wird durch die Mailbox blockiert!). Es ist also unbedingt notwendig die Applikation auf dem laufenden Linux mittels eines SIGINT (STRG+C) zu beenden. Alternativ kann man dem Programm auch auf der Kommandozeile oder durch z.B. htop das Signal schicken. Mit dem Signal sendet Linux einen speziellen Befehl an den NIOS2 wodurch dieser die Kommunikation auch beendet. Werden wieder Befehle geschickt, wird die Kommunikation von beiden Seiten wieder aufgenommen.

3 Operating System - FreeRTOS

Dieses Kapitel beschreibt wie das verwendete Echtzeitbetriebsystem zu benutzen ist. Ein FreeRTOS Port für den NIOS2 in der Version 9.0.0 wird derzeit eingesetzt. Dabei wurden nur minimale Änderungen an der Standardkonfiguration vorgenommen, die allerdings jederzeit erweiterbar ist, sofern das durch die Applikation nötig ist. Die Website von FreeRTOS mit Download und kompletter Spezifikation ist unter http://www.freertos.org/ zu finden.

3.1 Tasks

Die verwendeten Tasks auf der NIOS2 Plattform sind unter /Software/Software_NIOS2/tasks/ in den tasks_nios Dateien zu finden. Folgende Tasks sind dabei im Moment vorhanden:

- readUltraSonic: Dieser Task ist dafür zuständig die vier verwendeten und über IIC angeschlossenen (vorne zwei, hinten zwei) SRF08 Ultraschallsensoren auszulesen. Im Moment werden nur wenige der möglichen Ultraschalldaten verwendet, die dann dafür sorgen, dass im setMotor_and_Steering Task langsamer (wenn im Nahbereich eines Hindernisses) gefahren bzw. komplett stehen geblieben wird (wenn direkt vor dem Hinderniss). Die Parameter für das Anpassen der Entfernungen (Nahbereich bzw. Notstop) können durch Anpassung der Konfigurationsparameter in der tasks_nios.cpp angepasst werden (auch in der Doxygen zu finden).

- readMPU: Dieser Task hat die Aufgabe den mpu6050 Sensor auszulesen. Dabei werden die ausgelesenen Werte im Moment nur grafisch angezeigt. An anderer Stelle werden diese Informationen nicht benutzt. Können eventuell für eine spätere Positionsbestimmung bzw. Aufzeichung einer Wegstrecke verwendet werden.

- readRotary: Dieser Task liest den RotaryEncoder aus, der auf dem Fahrzeug mit drauf ist. Auch diese Daten werden im Moment nur für die grafische Oberfläche benutzt. Auch diese Daten können später dafür verwendet werden, um eine Wegstrecke zu ermitteln und dann damit eine genaue Position auszurechnen.

- setMotor_and_Steering: Dieser Task ist für das Setzen des Lenkwinkels und des Motor PWM Signals zuständig. Dabei werden die Informationen aus den Ultraschallmodulen verwendet, um auf Hindernisse reagieren zu können. Ansonsten werden alle Daten, die aus dem HQ auf den ARM und damit in den Shared Memory Bereich kommen ausgelesen und als Geschwindigkeit und Lenkwinkel gesetzt. Dabei wird auch ein Timeout Mechanismus verwendet, der dafür sorgt, falls keine

Tabelle 3.1: Taskkonfiguration

Taskname	*Priorität*	*Zykluszeit*	*verwendete Variablen*
readUltraSonic	3	75 ms	write: global_us_front_left_data write: global_us_front_right_data write: global_us_rear_left_data write: global_us_rear_right_data
readMPU	2	50 ms	write: global_acc_data write: global_gyro_data write: global_temp_data write: global_drive_info
readRotary	2	50 ms	write: global_drive_info
setMotor_and_Steering	3	20 ms	read: sharedMem(Alf_Drive_Command)
setDriveInfo	1	200 ms	write: sharedMem(global_drive_info)

neuen Daten mehr kommen, weil die z. B. die Verbindung zwischen HQ und Fahrzeug abgebrochen ist, das Fahrzeug langsamer wird und schlussendlich auch stehen bleibt. Sobald die Verbindung wiederhergestellt ist kann sofort ohne Verzögerung weitergefahren werden. Ein manueller Reset oder ähnliches ist nicht notwendig.

- setDriveInfo: Dieser Task schreibt alle gesammelten Fahrinfos (mpu6050 Daten + Geschwindigkeit) in den Shared Memory Bereich zwischen ARM und NIOS, die dann vom ARM Linux System ausgelesen werden können.

Die komplette Taskkonfiguration ist in Tabelle 3.1 zu sehen. Dabei ist die Priorität im FreeRTOS von 0 zur höchsten (konfigurierbaren) Priorität festgelegt. Das bedeutet das 0 die kleinste Priorität ist, die z. B. auch der Idle Task vom FreeRTOS Kernel besitzt, der aufgerufen wird, wenn kein anderer Task im ready Status ist. Die Zykluszeit gibt an, nach wie vielen 1/1000 Sekunden der Task wieder aufgerufen wird. Dabei ist das ganze nur mit geringer Abweichung fest konfigurierbar. Das heißt ein eingestellter Wert von 20ms wird meist nie genau getroffen - diese geringen Abweichungen stellen im Moment allerdings kein Problem dar. Die verwendeten Variablen geben an auf welche in der tasks_nios.cpp definierten Variablen der Task schreibend und lesend zugreift. Die komplette Task Kommunikation, die sich auf ein Minimum beschränkt wird aktuell über lokale bzw. teilweise über globale Variablen erledigt. Das Fahren und die Ultraschallsensoren sind aktuell am wichtigsten da auf diese Ereignisse sofort reagiert werden soll. Alle anderen Bereiche haben eine geringere Priorität. Die Ultraschallmodule können in der Standardeinstellung (6 Meter Reichweite) alle 65ms ein neues Ergebnis liefern. Da allerdings die Genauigkeit des FreeRTOS Zyklus nicht so genau war, wurde hier ein leicht größerer Wert gewählt. Würde ein geringeren Wert gewählt müssen zuerst die Ultraschallsensoren abgefragt werden, ob diese bereits mit der Messung fertig sind. Da dieses Abfragen auf keinen Fall passieren soll, der IIC sollte so wenig wie möglich am Stück benutzt werden, da nur ein IIC Modul vorhanden ist und auch der mpu6050 über den selben Bus angeschlossen ist. Somit könnten verschiedene IIC Transmissionen kollidieren, wenn ein Task länger auf den Bus

zugreift.

Ein Task kann mittels xTaskCreate() erzeugt werden. Der Scheduler wird dann mit dem Befehl vTaskStartScheduler() gestartet. Von diesem Zeitpunkt an werden alle vorher erzeugten Task zyklich aufgerufen. Eine genauere Dokumentation zu den jeweiligen APIs des FreeRTOS ist im Repository unter /Datasheets im Reference Manual zu finden. Das zyklische Aufrufen eines Tasks ist mit verschiedenen Methoden möglich. Die in diesem Projekt verwendete Methode wird nun kurz in 3.1 aufgezeigt.

Listing 3.1: Taskzyklus erzeugen

```
TickType_t xLastWakeTime;
// bestimmt die Zyklusfrequenz des Tasks in
const TickType_t xFrequency = 20;

while(1)
{
  // hier wird gewartet bis der Zyklus vollstaendig ist
  vTaskDelayUntil( &xLastWakeTime, xFrequency );
  // alle Taskoperationen koennen hier ausgefuehrt werden
}
```

3.2 RTOS Config

Hier wird die genaue RTOS Konfiguration beschrieben, die für das OS auf dem NIOS2 benutzt wurde. Die Konfiguration ist in /Software/Software_NIOS2/os/ in der FreeRTOSConfig.h zu finden. Die einzelnen Parameter werden im Reference Manual genauer erläutert. Hier werden nur die wichtigsten verwendeten Parameter erklärt.

- configUSE_PREEMPTION: wenn auf 1 gesetzt, ermöglicht das dem Scheduler das unterbrechen der Tasks wenn zu einem FreeRTOS Tick ein höherpriorer Task als ready markiert ist, als der aktuelle. Wenn auf 0 gesetzt kann der Task nur durch sich selbst beendet/unterbochen werden (z.B. ein Zyklus fertig, warten auf den nächsten ready Status). Da das Fahren und der Ultraschall jeweils sehr wichtig sind, wurde dieses Verfahren gewählt.
- configTICK_RATE_HZ: definiert die FreeRTOS Tick Rate. Ein Wert von 200Hz bedeutet das 200mal in der Sekunde ein FreeRTOS Tick auftritt an denen der Scheduler aktiv wird und einen Kontextwechsel einleitet bzw. den nächsten Task aufruft, der bereit ist.
- configCPU_CLOCK_HZ: Hier wird die Rate angegeben, mit der der interne Systemtimer (aktuell: TIMER_0_NIOS2_BASE) arbeitet den das FreeRTOS benutzt. Muss nicht der CPU Frequenz entsprechen.
- configIDLE_SHOULD_YIELD: wenn auf 1, wird der Idle Task, den das FreeRTOS automatisch implementiert sofort wieder durch den Scheduler unterbrochen und der

nächste Task, der ready ist kann sofort aufgerufen werden. Andernfalls bleibt der Idle Task mindestens einen kompletten FreeRTOS Tick lang aktiv, was dazu führt das alle Tasks insgesamt eine gerechtere Aktivzeit erhalten. Da dies in unserem Fall nicht nötig ist, wurde dieser Parameter auf 1 gesetzt.

Um das FreeRTOS auf dem NIOS2 Core zu benutzen sind weiterhin zwei Dinge zu beachten. Da bei jedem Erstellen eines BSPs in der system.h das Makro
ALT_ENHANCED_INTERRUPT_API_PRESENT
definiert wird, der NIOS2 Port damit aber nicht umgehen kann, muss dieses Makro durch
ALT_LEGACY_INTERRUPT_API_PRESENT
ersetzt werden, ansonsten meldet der Linker undefinierte Verweise. Des Weiteren ist unter /Software/Software_NIOS2/os/Source/portable/ die port.c zu finden. In dieser wird der Timer definiert, der dafür zuständig ist die FreeRTOS ticks zu erzeugen, die z. B. auch der Scheduler benutzt. Sollte dieser Systemtimer geändert werden, aus welchen Gründen auch immer, muss dieses File mit angepasst werden und alle vorkommenden TIMER_0_NIOS2_BASE Einträge auf den neuen Timer angepasst werden.

4 Eclipse NIOS2

4.1 C++ - Beschränkungen, besondere Einstellungen

Das in der 16.1 verwendeten Quartus mit dem mitgeliefertem GCC Compiler hat einige Einschränkungen bezüglich der Verwendung von einigen C++ Features. Alle in den C++ Standardbibliotheken vorhandenen STL Container, z. B. std::vector, std::stack, std::map usw., sind nicht benutzbar. Außerdem ist es nicht möglich die std::string Klasse zu benutzen. Die Benutzung solcher Features führt dazu, dass der Speicher nicht mehr ausreicht. Für die Verwendung dieser Klassen sind auf dem NIOS2 mit der aktuellen Toolchain ca 700KB RAM nötig, allerdings sind nur 128KB RAM vorhanden. Dies wurde vom ALTERA Support Team direkt bestätigt mit der Angabe, dass die Benutzung von C++ im Moment nicht effizient möglich ist und externer Speicher für die Verwendung von C++ angeraten ist. Alle anderen Features hingegen sind, soweit bekannt, ohne Einschränkungen benutzbar.

Für die Unterstützung von c++11 ist eine kleine manuelle Anpassung des Makefiles nötig, welches die Toolchain mit Erstellung eines BSP automatisch generiert. In der Sektion

- # Arguments only for the C++ compiler.

ist die Ergänzung folgenden Flags nötig: -std=c++11; da diese Einstellung über die GUI nicht erhalten bleibt. Die Sektion sollte dann folgendermaßen aussehen:

- # Arguments only for the C++ compiler.
  APP_CXXFLAGS := $(ALT_CXXFLAGS) $(CXXFLAGS)
  -std=c++11

Diese Einstellung ist auch unbedingt nötig, da einige c++11 Features verwendet wurden und demzufolge ohne diesem Flag die Applikation nicht erfolgreich kompiliert.

4.2 Die externen/internen ips

Qsys bemühen, die mitzunehmen, sonst fehlt da was; siehe git doku?

5 NIOS2 - Treiber / Hardware Abstraction

Hier wird kurz der Aufbau der HAL bzw. der Aufbau der jetzt zur Verfügung stehenden Treiber des NIOS2 Cores erläutert. Genauere Dokumentation ist in der Doxygen Dokumentation zu finden, die im Anhang mitgeliefert wird.

5.1 Display

Das Display kann sogar Zahlen anzeigen

5.2 Motor

Die Klasse Drive ist für die Ansteuerung des Motors zuständig. Dabei ist es möglich die maximale Geschwindigkeit im Bereich zwischen 0% und 100% zu begrenzen. Eine weitere Funktion ermöglicht das Setzen der Richtung und der Geschwindigkeit, das dann in ein PWM Signal für den Motor umgerechnet wird.

5.3 Lenkung

Die Lenkung wird mittels der Klasse Steering ermöglicht. Eine Init() Funktion setzt denn maximalen Lenkwinkel, der für die Lenkung benutzt wird. Eine weitere Funktion Set() setzt dann den tatsächlichen Winkel.

5.4 MPU6050

Das mpu6050 Modul ist über den IIC Bus angeschlossen und kann die 3 Beschleunigungsachsen, 3 Drehachsen und die Temperatur auslesen. Das Modul muss vor der Verwendung einmalig initalisiert werden. Dabei ist zu beachten, dass der AD0 Pin die IIC Adresse des mpu6050 Bausteines hardwaremäßig verändert.

5.5 Ultraschall

Die vier zur Verfügung stehenden Ultraschallsensoren sind über den gleichen IIC Bus, wie das MPU6050 Modul angebunden. Diese Geräte benötigen keine Initialisierung, d.h. sind sofort nach dem Anschließen an den Bus einsatzbereit. Allerdings ist zu beachten,

dass nur Geräte an den Bus angeschlossen werden, die unterschiedliche IIC Adressen haben (Funktion zur Änderung der IIC Adresse ist vorhanden), da es sonst zu undefiniertem Verhalten auf dem Bus kommt. Für das ändern der Address sollte nur ein Gerät angeschlossen sein.

6 Hilfreiche Knowledge Bases

Link/Artikelsa

7 Zusammenfassung

Das Projekt konnte erfolgreich abgeschlossen werden.

8 Offene Probleme und zukünftige Arbeitspakete

8.1 Offene Probleme

Zum Zeitpunkt der Ab-/Übergabe des Garfield Projekts bestehen noch einige kleine und umfangreichere Probleme. Manche dieser Probleme sollten mit einer hohen Priorität behandelt werden, andere Probleme stören nur die Bedienbarkeit und haben deswegen eine niedrigere Priorität.

8.1.1 Spannungs- und Stromversorgung

Dies ist ein Problem, dass mit höherer Prioriät, bevorzugt am Anfang des nächsten Projekts, behandelt werden sollte. Im wesentlichen gibt es zwei größere Probleme:

- Immer wieder kehrende Spannungseinbrüche nach Starten des Motors.
- Eine nicht ausreichende Spannungsversorgung durch kleine RC-Akkus mit einer Sollspannung von $7,2$ Volt.

Für die Spannungseinbrüche konnte bis jetzt wieder eine sinnvolle Erklärung noch ein Lösungsvorschlag erreicht werden. Die Einbrüche treten sowohl an der Versorgung durch ein Labornetzteil auf(wobei nicht jedes Labornetzteil die gleichen Symptome zeigt) als auch bei Speisung durch einen großen RC Akku mit 12V Sollspannung auf. An einem Labornetzteil treten die Einbrüche zwar wesentlich weniger oft auf, aber auch dort treten sie nicht-deterministisch auf. Die Fehler treten auch wesentlich öfter auf als dies im Vorgängerprojekt mit einem Rasperry Pi der Fall war. Die aktuelle Lösung braucht zwar etwas mehr Leistung (zwischen 0.5 und 1 Watt), aber sowohl der RC Akku als auch ein Labornetzteil können normalerweise mit solchen Problemen umgehen

Das Problem mit den kleinen RC Akkus (für die sich eine Akkuhalterung auf dem Auto befindet) kann sich ebenfalls nicht erklärt werden. Die $7,2V$ werden nur zur Versorung des Motors benötigt, ansonsten sind nur 5V und $3.3V$ notwendig. Diese beiden Spannungen werden aber über Konstantspannungskonverter erzeugt, die beide relativ unempfindlich gegen höhere, auch schwankende Spannungen sind.

Als erster Ansatzpunkt kann nur ein Hinweise/Lösungsstrategie empfohlen werden: Die Platine, die immer wieder wechselnden Bedürfnissen angepasst wurde, sollte überprüft werden. Hier ist die Unvoreingenommenheit einer anderen Bearbeitungsgruppe mit

Sicherheit von großem Vorteil. Die Nachfolgegruppe kann alle Verbindungen überprüfen und, obwohl ein Schaltplan vorhanden ist, über jede Verbindung nachdenken und überprüfen ob die Beschaltung so Sinn machen wie sie aktuell gelöst sind. An diesem Punkt kann man auch nochmal über Stützkondensatoren an relevanten Stellen nachdenken. Unter Umständen ist es dann auch hilfreich eine neue Platine zu erstellen.

8.1.2 Motortreiber

Der Motortreiber hat während der Tests immer wieder sporadische Aussetzer gehabt und funktionierte nach einem Neustart manchmal nicht richtig. Als Übergangslösung wurde an dem Motortreiber ein Reset-Button angebracht, der, sollte der Motortreiber nicht mehr funktionieren, gedrückt werden kann und alle Fehlerzustände des Motortreibers löscht und ihn wieder funktionieren lässt. Die Ursache für diese Fehler ist nicht offensichtlich, aber eventuell ist ein Zusammenhang mit den in 8.1.1 beschriebenen Fehlern zu beobachten.
Eine mögliche Lösung wäre die Fehlersignale und das Resetsignal mit dem FPGA zu verbinden und im NIOS2 zu verarbeiten. Dann kann eine saubere Fehlerbehandlung eine Resetauslösung durch einen μController erfolgen.

Foto einfügen

8.1.3 MPU6050

Tobi: Probleme beschreiben

8.1.4 Sporadische Fehler in der Nachrichtenübertragung

In der Nachrichtenübertragung zwischen Host-PC und NIOS2 (mit dem Nachrichtenweg wie in Abbildung) treten sporadische Fehler auf. Diese äußern sich in Nachrichtenpaketen (einzelne Werte, i.d.R. sind nicht alle Werte falsch), die den jeweiligen Maximalwert annehmen(z.B. uint8_t x = 255 anstatt uint8_t x = 0). Für diesen Fehlerfall konnten noch keine weiteren Beobachtungen gemacht werden, insbesondere nicht woher die falschen Nachrichten kommen (Entweder vom Host PC als Initiator der Nachricht oder vom Kommunikationsgateway das für das kopieren von Nachrichten in den Speicher verantwortlich ist). Die falschen Werte treten in sehr unregelmäßigen Abständen auf, es ist auch möglich das über mehrere Sitzungen hinweg kein einziger solcher Fehler auftritt. An anderen Tagen tritt dieser Fehler wiederrum regelmäßig im 10-Sekundentakt auf.

irgendwie Nachrichtenweg noch herzaubern?

8.2 Mögliche Arbeitspakete

Die nächsten Schritte

Abbildungsverzeichnis

Abkürzungsverzeichnis

ALF	Autonomes Laser Fahrzeug
HAL	Hardware Abstraction Layer
HSP	Hauptseminar Projektstudium
Lidar	Light detection and ranging
ROS	Robot Operating System
RVIZ	ROS Visualization
SoC	System-on-a-Chip
FPGA	Field Programmable Gate Array
IP	Intellectual Property
HPS	Hard Processor System
PLL	Phase-locked loop

Literaturverzeichnis

[1] Altera. *Using the ARM Generic Interrupt Controller*. Application Note. Intel. URL: ftp://ftp.altera.com/up/pub/Altera_Material/14.0/Tutorials/Using_GIC.pdf.

[2] Altera/Intel. *SW Development for Altera SoC Devices Workshop*. 2016. URL: https://rocketboards.org/foswiki/pub/Documentation/WS1IntroToAlteraSoCDevices/WS_1_Intro_To_SoC_SW_Workshop.pdf.

[3] Philemon Favrod. "How to configure Linux to receive IRQs from the FPGA?" HowTo. Eidgenössische Technische Hochschule Lausanne. URL: https://wiki.epfl.ch/prsoc/documents/Cyclone_V_SoC_Linux_Interrupt-2.pdf.

[4] Altera Inc. *HPS SoC Boot Guide - Cyclone V SoC Development Kit*. 2016. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an709.pdf.

[5] Intel. *Embedded Peripherals IP User Guide*. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf.

[6] Linus Torvalds und the Kernel Team. *Linux*. URL: kernel.org.