HSP Projektbericht

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

26. März 2017

Inhaltsverzeichnis

[1]Field Programmable Gate Array

Todo list

1 Einleitung

Dieser Projektbericht beschreibt die Tätigkeiten der Autoren im Laufe des HSP[2] 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!

[2] Hauptseminar Projektstudium

1.1 Eingesetzte Tools

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 Garfield.vhdl. Dort wird das von QSYS erzeugt System und einige kleine IP[3]-Cores zusammengeführt und auf definierte Aus-/Eingänge geführt. Diese Ein-/Ausgänge werden dann über den *Pin-Planner* auf die physikalischen Pins geführt. Im folgenden werden die alle Systemkomponenten, die für das Garfield-Projekt erzeugt wurden, beschrieben.

Pfade?

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.

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.

[3]Intellectual Property

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[5]	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.
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!**[6] 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 HPS benötigt/eingesetzt werden. Auf der linken Seite der Abbildung ist das HPS System

[5] Phase-locked loop
[6] **RISC!**

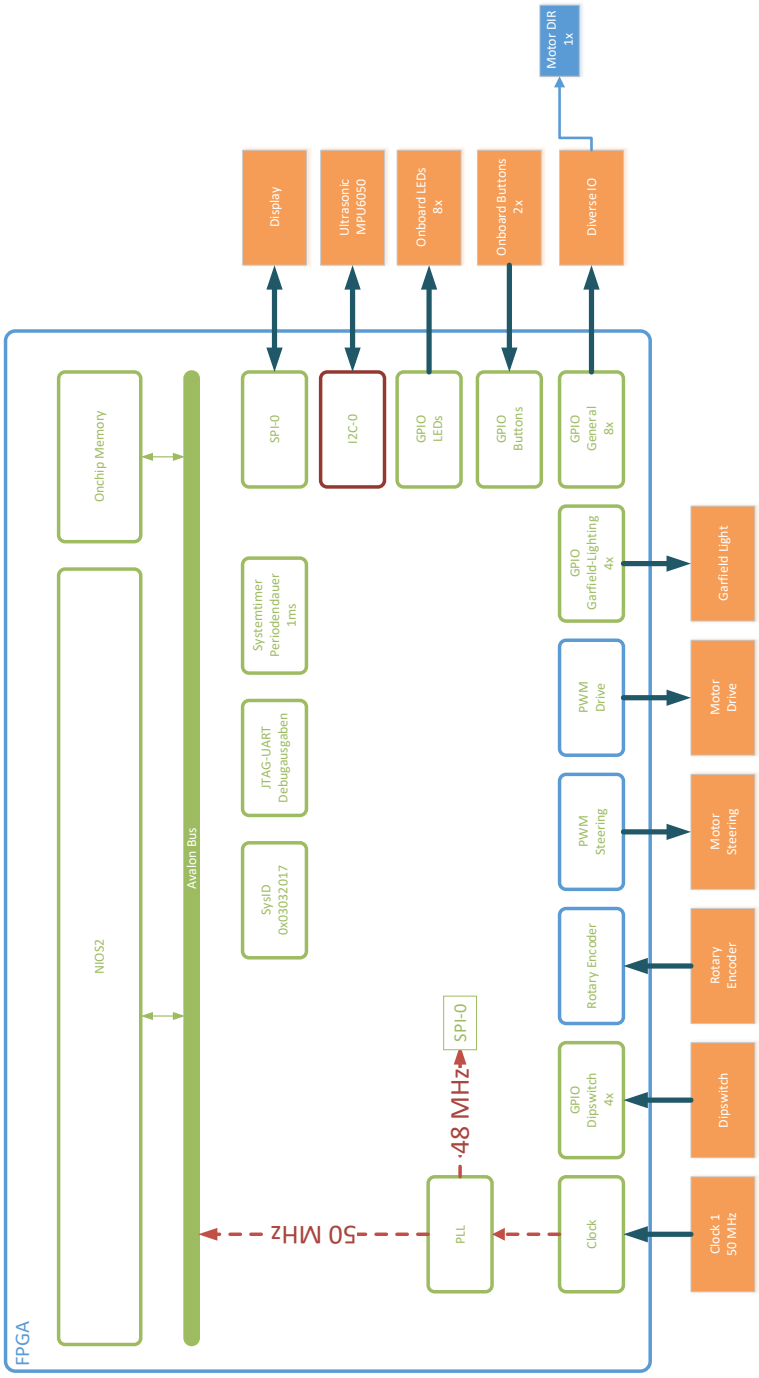

Abbildung 2.1: Übersicht der (meisten) eingesetzten IP-Cores. Die Abbildung verzichtet auf die Darstellung der Teile, die für die Kommunikation mit dem HPS[4] zuständig sind. Blau umrandet sind Cores, die im Rahmen des Projekts selbst progammiert wurden, grün diejenigen, die Teil der Altera Toolchain sind und rot externe IP-Cores von opencores.org

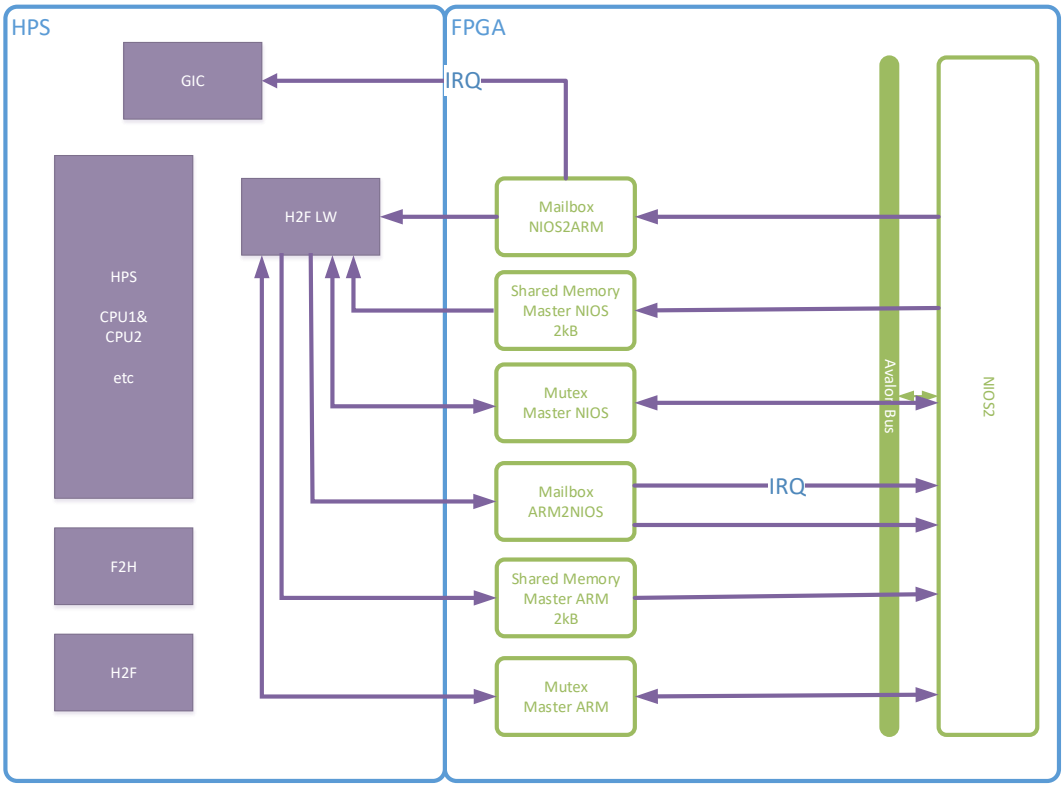

Abbildung 2.2: IP-Cores und deren Kontrollfluss, die an der Kommunikation zwischen NIOS2 und HPS beteiligt sind.

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 [**Using_GIC**] 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 [**FPGA_Workshop**].

- 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 [**embedded_guide**]

- 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 [**embedded_guide**]. 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.

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.

2.2 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

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

- 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

Mailbox Kommunikation ARM ↔ NIOS2

Name, Pfad?

Die Kommunikation über die in das FPGA programmierte Mailbox (siehe 2.1.1) 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, 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

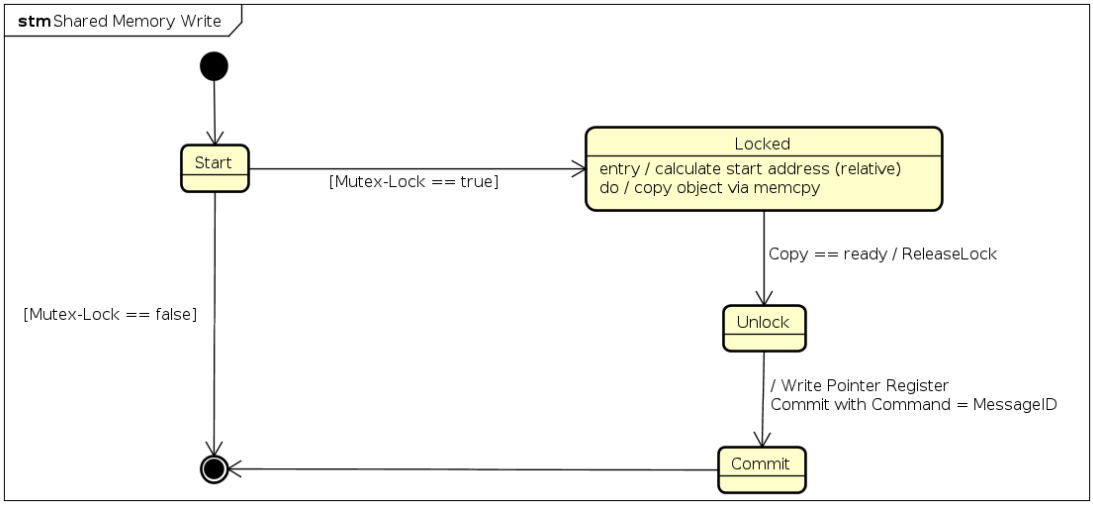

Abbildung 2.4: Schreibvorgang in den Shared Memory

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.

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.

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 Zusammenfassung

Das Projekt konnte erfolgreich abgeschlossen werden.

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