NTNU
Norwegian University of
Science and Technology

Development Tools for the Xilinx Zynq-7000 SoC

Design of a Development Framework and System Interaction Method for Embedded Systems on the Zynq-7000 SoC

Roland Peterer
Christian Schmid

Master of Energy and Environmental Engineering
Submission date: July 2016
Supervisor: Lars Einar Norum, ELKRAFT

Norwegian University of Science and Technology
Department of Electric Power Engineering

NTNU
Norwegian University of
Science and Technology

Master Thesis

Development Tools for the Xilinx Zynq-7000 SoC

Design of a Development Framework and System Interaction Method for Embedded Systems on the Zynq-7000 SoC

Authors:
Christian Schmid and
Roland Peterer

Supervisor:
Prof. Dr. Lars Norum

Department of Electric Power Engineering

July 1, 2016

Declaration of Authorship

We, Christian Schmid and Roland Peterer, declare that this thesis titled "Development Tools for the Xilinx Zynq-7000 SoC" and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.
- Where the thesis is based on work done by ourself jointly with others, we have made clear exactly what was done by others and what we have contributed ourself.

Trondheim, July 1, 2016

"*Any sufficiently advanced technology is indistinguishable from magic.*"

Clarke's Third Law

Abstract

Development Tools for the Xilinx Zynq-7000 SoC

Design of a Development Framework and System Interaction Method for Embedded Systems on the Zynq-7000 SoC

by Christian Schmid and
Roland Peterer

The evolution of processing system hardware provides powerful and versatile implementation platforms for system engineers such as the Xilinx Zynq-7000 SoC. Due to this new capable hardware, advanced and more demanding control systems become feasible. However, the complexity of such targets has risen enormously. They rely on special features and enhancements to enable the necessary power for complex tasks. In order to use those, a profound knowledge of the target platform is needed.

In order to ease the usage of a hardware platform, a comprehensive development framework was set up. It allows the usage of the commonly known approach to design and outline control algorithms with Matlab/Simulink. The developed control method can then be tested in combination with the desired hardware. The framework guides the system engineer in generating the necessary code, the setup of the target hardware and the deployment of the realized control design. Methods to test and interact with the running system on the target in realtime are introduced.

A simple way of interacting with the completed system was realised. It allows the change of parameters and the monitoring of signals in the system during run time. Furthermore, recording transient step responses is possible. By utilizing standard Ethernet protocols as a base, the interaction can take part on a remote system in a GUI. Other means of interaction can be implemented due to the open data interface. This enables the supervision and manipulation of the running target from afar.

Acknowledgements

We would like to thank our Thesis Advisor Prof. Dr. Lars Norum for his support and time. Also, our gratitude goes to our Master's Study Advisor Prof. Dr. Benno Bucher who supported us throughout our studies.

Further we thank Anirudh Budnar Acharya and Eirik Haustveit for the collaboration and exchange of ideas.

Additionally we thank Tomas Richter for the interesting discussions and support during the MathWorks course "Programming Xilinx Zynq SoCs with Matlab and Simulink"

Roland would like to express his sincere gratitude to the *Hirschmann Stiftung* for the financial support. It would not have been possible to enter this exchange adventure without its support.

Contents

List of Figures

List of Tables

Glossary and Abbreviations

Asymmetric Multi Processing (AMP)
On the cores of a processor, different software systems (bare-metal and/or operating systems) are running. See also: SMP.

Analog to Digital Converter (ADC)
Unit which measures an analog signal and passes the discretised value in digital form.

Advanced eXtensible Interface (AXI)
High performance data connection both in terms of frequency and transmitting capabilities. In the Zynq mainly used for communication between the PS and PL.

Bare-metal system
An embedded system software which implements the functionality directly without any operating system or similar setup.

Bitfile
Binary file containing a FPGA hardware description, similar to an executable software for processing systems.

Bourne-again Shell (Bash)
Standard shell for Linux.

Constraint File
Is used to set IO standards and chip pin mapping for Vivado projects.

Cross-Compiler
In contrast to normal compiling, cross-compiling prepares executable code for a different system architecture as the one it is compiled on.

Direct Memory Access (DMA)
Method of allowing the direct access of memory by peripherals without causing additional processor load.

Digital Signal Processing (DSP)
Special FPGA structures for complex digital signal operations.

Dynamic Reconfiguration Port (DRP)
IP core interface to interact with the hardwired XADC.

Field Programmable Gate Array (FPGA)
Digital integrated circuit with programmable gate arrays. The functionality is programmed and can be modified in the field in a hardware description language (HDL). Features vary heavily between FPGAs and are distinctly different from software-based, sequential processing systems.

First Stage Boot Loader (FSBL)
Program which runs directly on the underlying hardware, without an intermediate system software layer.

Framework
Set of tools and libraries as an environment. Used to ease and guide development of a given system and functionality.

General Purpose Input/Output (GPIO)
Pin which can be used as a simple input or output by the user. Usually, also combining several GPIOs and connecting them to special peripheral modules to implement communication buses is possible.

Graphical User Interface (GUI)
Program, website or other mean of a graphical manipulation interface to oversee and manipulate another program or system.

Graphical User Interface Design Environment (GUIDE)
Graphical user interface development environment of Matlab.

Hardware Description Language (HDL)
Language to describe the hardware to be implemented in an FPGA.

High Level Synthesis (HLS)
Process of generating a hardware description from an abstract algorithm design or model.

Integrated Development Environment (IDE)
Software which combines all needed tools and further simplifying mechanisms to allow the development and testing of code for a certain system.

Inter Integrated Cirquit ($\mathbf{I^2C}$)
Serial peripheral bus system for low-speed communication between several master and slave participants.

Joint Test Action Group (JTAG)
Industry Colaboration which defined a standard for in-cirquit programming and debugging. The interface allows low-overhead serial communication with a stopped or running system.

MicroBlaze
Separate processor realised within a programmable logic structure (Softcore).

Multicore Communication API (MCAPI)
Standard of communication methods between systems in an AMP scheme.

Multiplexed Input/Output (MIO)
Interface which connects multiple peripheral pins to processor signals. Configured via the PS [58, page 54]

NEON
The NEON engine (as an addition to the standard ARM functionality) provides Single Instruction Multiple Data (SIMD) facilities.

Open Asymmetric Multi Processing (OpenAMP)
Framework for using standardised AMP interaction methods between the involved software systems.

Operating System (OS)
System software which offers basic functionality like hardware drivers, file system support or network capabilities. Most often provides multitasking support and the use of standardised library functions for simpler development, as well as standard tools for working on a system.

Operator
Person who runs, oversees and manipulates a given system.

PMOD
Standard pin array interface using GPIOs.

Processing System (PS)
System which runs software code, including the processor and associated elements.

Programmable Logic (PL)
System which consists of programmable logic blocks and its connections. Hardware structures described in HDL run on this part of a device.

Register Transfer Level (RTL)
Description of data flow and operations in an abstract design.

Remote System
Computer or other device connected to the target. Used for interaction and control of the target by means of a GUI or a superordinated control system.

Second Stage Boot Loader (SSBL)
A more sophisticated, but still minimal system software. It is started by the FSBL and is responsible for initialising further hardware and start the bare-metal software or operating system.

Serial Peripheral Interface (SPI)
Serial bus system for communication between processor parts or peripherals. Less flexible than I^2C, it is mainly used for more local usage.

Simulink PL Canvas
Canvas to implement functionality of HDL part.

Symmetric Multi Processing (SMP)
On all cores of a processor the same system software is run.

System Engineer
Person who develops and modifies a system, e.g. a control system for a given appliance.

Target System
Platform on which an intended function is realised.

Tool Command Language (TCL)
Language to describe and script the initialisation of a Vivado project.

User Datagram Protocol (UDP)
UDP is a simple connectionless transmission protocol for computer networks.

Universal Asynchronous Reciever Transmitter (UART)
A very common serial communication protocol. Physical interfaces include RS-232 or EIA-485.

Universal Serial Bus (USB)
: Standard serial connection for computer peripherals.

Watchdog
: Monitoring system to determine if a system is still sane. If it isn't assured of propper functionality by a keep alive signal periodically, the system is put into a safe state by the watchdog.

XADC
: Analog to digital converter integrated in the Xilinx Zynq-7000 SoC.

Additionally, a good list of common terms and abbreviations can be found in [9].

Chapter 1

Introduction

This chapter gives an overview over this document. It introduces the project context and its purpose. There is also a brief introduction into embedded systems in general and the used platform in particular to get the reader started.

1.1 Overview over the Document

The introductory chapter gives the reader an overview over the whole document, as well as some general information about the project and the context it stands in. Next, the underlying problem and the resulting functional specification of the project are outlined. The *Approach and Elaborated Solution* chapter explains the approaches which were used to implement the desired functionality, while the *Working with the Development Framework and Remote System Interaction* chapter focusses on the usage of the resulting framework and tools. Chapter *Further Background Information* gives a deeper insight into surrounding or underlying topics, while *Demonstration and Tutorials* shows the implemented functionality. Also it helps to get readers started in setting up and using the provided tools. Finally, a conclusion over the reached topics rounds up the project.

In the Glossary, to be found on page xv, some less commonly known expressions and abbreviations are explained. Appendix A lists some additional resources for interested readers, while Appendix B gives some hints about convenient commands and tools for both Linux and Matlab. In Appendix C the structure and contents of the Digital Appendix are described for easier navigation and comprehension.

1.2 Project Environment

The Department of Electric Power Engineeringof the Norwegian University of Science and Technology in Trondheim strives to introduce a new, universal hardware platform for control applications. It is bound to make use of the Xilinx Zynq-7000 SoC, which offers great flexibility and versatility as a hardware base. While the control algorithms and other software-based parts of future projects should reside in a Zynq, the effective purpose of a system and therefore its surrounding hardware vary significantly. Some projects already made use of this platform. However, there is no consistent way of designing such systems and most of the gained insights into the common base is lost after each project. Therefore a framework which defines a handy workflow would be helpful to ease the getting started with the Zynq SoC.

1.3 Purpose of the Project

The evolution of processing system hardware provides powerful and versatile implementation platforms to system engineers such as the Xilinx Zynq-7000 SoC. Due to this new capable hardware, advanced and more demanding control systems become feasible. However, the complexity of such targets has risen enormously. They rely on special features and enhancements to enable the necessary power for complex tasks. In order to use those, a profound knowledge of the target platform is needed.

While traditional commercial projects can make use of experienced specialists for all aspects of such an implementation, especially applications in education and/or research can't always rely on such expertise. Time and knowledge are limited. Moreover, also modern design approaches suggest the development of algorithms and functions on a higher abstraction level. Therefore, it would be beneficial to allow a system engineer to concentrate on the design of the control aspects for a certain application, without the need of experience in special hardware interaction or even low-level programming skills.

As outlined in chapter 2, the aim of this project is to help system engineers to get started with their work on a complicated and versatile hardware platform. Also it should diminish the need for extensive research about the underlying hardware base by summarising several aspects of the Xilinx Zynq-7000 SoC environment and use.

1.4 Introduction to Embedded Systems

Modern power electronic designs heavily rely on advanced control algorithms. Be it multilevel converters, motor drive systems, energy storage or power station systems. For an efficient, safe and reliable operation, they all depend on highly sophisticated and demanding control schemes. Usually such control schemes are implemented in an embedded system. These embedded systems are electronic structures which serve a distinct purpose. They are highly optimised for performance and reliability. Some applications even demand real-time and fully deterministic behaviour, which can only be achieved on a system with delimited purposes.

When designing algorithms for an embedded system, the main concern must be reliability and performance. Unnecessary operations should not interfere with the immediate purpose of the system. Therefore the special limitations and demands should already be taken into account in the design process. For further information about code optimisation also see [25, 31]. While working on an embedded system, the design principle should be:

KISS

Keep **i**t **s**imple, **s**tupid. [45]

This means that the complexity and size of a system should be kept as small as possible. This helps to keep it comprehensible and reduces the risk of unwanted interferences. It also rises the manageability and eases maintenance significantly.

1.5 Short Overview over the Xilinx Zynq-7000 SOC and the Digilent ZedBoard

The Xilinx Zynq-7000 SoC is a modern, high performance system on chip. It is equipped with both a software processing system and a programmable logic part (FPGA). It features a dual-core ARM Cortex-A9 MPCore with a maximum frequency of 1 GHz, supports several external memory types, features a NEON engine, provides UART, USB2.0, Gigabit Ethernet and SD/SDIO support as well as many more. The FPGA, originating from the Artix-7 or Kintex-7 family, consists of and features a specialized DSP, an XADC and Block RAMs. The PS and PL part are interfaced with a high performance AXI interface. [59]

The combination of both a processing system and a programmable logic offers a huge advantage in control applications. Highly critical and demanding tasks can be outsourced to the FPGA. There they can meet hard realtime requirements, run in parallel and in a deterministic environment. More complex tasks like interfacing to other components or the user can be implemented in the processing system, where a lot of peripherals can be used. MicroBlaze softcores or other complex and customised structures can be implemented and undertake special tasks, due to the vast size of the FPGA. All this adds up to a high performing foundation for all kinds of control systems.

The Digilent ZedBoard is a development kit based on the Xilinx Zynq Z-7020 SoC, which features the most powerful Artix-7 FPGA of the Zynq family. 512 MB of DDR3 memory offer both fast and vast storage. The ZedBoard allows the user to concentrate on the software and FPGA design on a proven hardware base. A broad range of interfaces offers flexibility for more complex projects, such as an USB host, audio inputs and outputs (connected via I^2S), Gigabit Ethernet, HDMI and VGA output, an 128x32 OLED display, a total of 7 push buttons, 8 switches, 9 LEDs, several PMODs and a FMC port for more complex interfacing. An SD card slot offers easy and flexible boot and data storage possibilities. To make the developer's life easier, there is also an USB to UART bridge and the possibility of daisy-chained JTAG programming. Booting from JTAG is also possible, as well as from the internal 256 MB Quad-SPI flash. [5]

For further reference on the Xilinx Zynq-7000 SoC, its applications and the ZedBoard, also see [9].

Chapter 2

Problem Definition and Functional Specification

The aim of this project is to provide future system engineers for power electronics with the necessary environment to easily conceive, implement and test their control design on a target hardware. In addition it should enable an operator to remote interact with the developed system. This shall be done by providing two sets of tools; on one hand a framework for the development phase, on the other hand a GUI for remote system interaction.

2.1 Development Framework

In order to ease the usage of a hardware platform, a comprehensive development framework shall be set up. It allows to benefit from a commonly known approach to design and outline control algorithms in combination with the desired hardware. It also guides the system engineer in generating the necessary code, the setup of the target hardware and the deployment of the realized control design. Methods to test and interact with the running system are introduced. Figure 2.1 shows the framework, which offers the containers of the custom user functions for the PS and PL parts. It also provides the connections to the surrounding components.

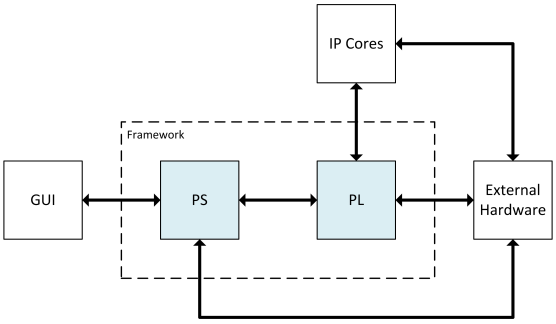

Figure 2.1: Development Framework in which the User Function Resides in the PS and PL Frames

2.1.1 Functional Requirements for the Development Framework

In descending priority, the functional requirements are as follows:

- Must have criteria:
  - Canvas for implementing the user function into the framework
  - Methods for code generation and synthesis of the user function, as well as the deployment onto the target
  - Methods to deploy user functions to the target persistently and run them standalone
  - Interface hooks for the user functions to communicate with hardware and a GUI (see section 2.2)
- Nice to have criteria:
  - Analysation of performance boosting measures, the need of them and their implementation (priority and affinity, threading, parallelism, AMP etc.)

2.2 Remote System Interaction

A simple way of interacting with the system shall be realised. It targets mainly for the usage of the finished system by an operator without the need of all development tools and skills. But such an interaction can also come in handy during testing and setting up a system. It allows the change of parameters and the monitoring of signals in the system during run time. By utilizing standard Ethernet protocols as a base, the interaction can take part on a remote system in a GUI, as depicted in Figure 2.2. This enables the supervision and manipulation of the running target from afar. In this part, the ease of use and setup in the operating phase is emphasized.

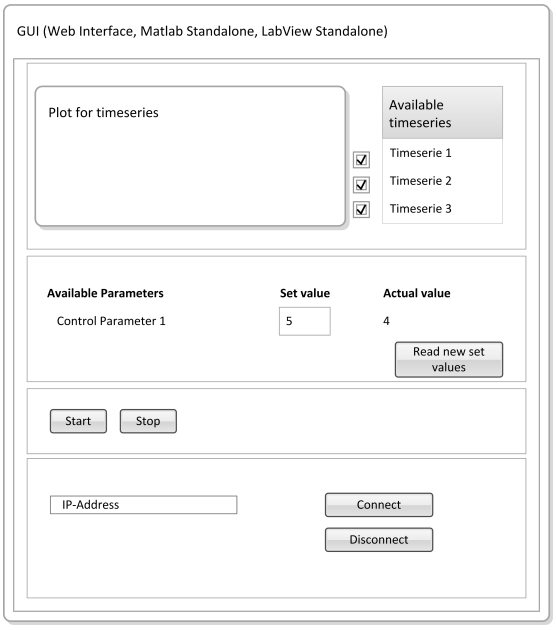

FIGURE 2.2: Interaction GUI on a Remote Machine

2.2.1 Functional Requirements for the Remote System Interaction

In descending priority, the functional requirements are as follows:

- Must have criteria:
  - Enable the user to get and set parameters
  - Logging of time series data (slower, continuously monitored signals)
  - Analysis of a step response with parameter step setup, triggering of the step and the response measurement
  - Methods for starting and stopping of the system
  - Enable the user to export acquired data
- Nice to have criteria:
  - Dynamic creation of the GUI, depending on the needs of the user function

2.3 Accompanying Documentation

The documentation of the framework should guide the user to ease the first steps upon usage. Where the provided tools fall short, a comprehensive description of all parts should allow the adaption to new needs. Furthermore, for the remote system interaction part the usage and setup is described in detail. The method of communication with the running system is disclosed in detail. This enables the development of other tools using the same interface for further enhancements and use scenarios.

2.4 Guidelines and other Specifications

Besides the overall goal of the project, resulting in the functional specification above, some additional guidelines are to be followed.

- As a target hardware platform, the Digilent ZedBoard with a Xilinx Zynq-7000 SoC is to be used.
- For a reference implementation, the motor drive system developed by Eirik Haustveit is to be used as a first use case.
- Only free or already licensed software available at the NTNU are to be used.

Chapter 3

Approach and Elaborated Solution

This chapter describes the functionality of the developed Framework and the Remote System Interaction. Those parts should accomplish the specified functional requirements. The chapter outlines the methods used to implement the needed features and gives a general overview over the possibilities. Detailed information about the usage and modification is described in chapter 4. The source code of all developed parts can be found in Appendix C.

3.1 Prosecuted Approach

To ease the development of embedded systems, several tools exist in the market. For almost all platforms, at least one Integrated Development Environment (IDE) exists. For the Zynq in particular, one of the most powerful tools is Vivado, which is released by the manufacturer of the Zynq, Xilinx. But this targets mostly the development by using traditional coding style, with program code for the PS and hardware descriptive language for the PL part. This requires the system engineer to have considerable knowledge in writing program and HDL code. The use of Matlab/Simulink offers a much easier entry into the world of embedded systems for students. Since most students are familiar with it and the graphic way of defining algorithms with Simulink offers a flat learning curve. Even in more advanced development teams it is an often used tool. More information on the workflow with Matlab/Simulink can be found in section 5.1. Due to its advantages and support for the Zynq, it was decided to use Matlab/Simulink as a base to realise both the Development Framework and the Remote System Interaction. This allowed the use of tools, scripts and predefined function provided by MathWorks to setup the solution presented in this document.

3.2 Development Framework

The Framework consists of various parts which are explained in the following subsection. It is organised in a folder structure, which can be found in Appendix C.

3.2.1 Custom Reference Design

To be able to deploy a system onto a target, the hardware layout and other information must be defined. This is done by implementing a Reference Design, which itself is announced to Matlab for use. The process of customizing such a Custom Reference

Design is described in section 6.3 and in [33]. The design used for the Development Framework is described in subsection 4.2.2

3.2.2 Main Simulink Project

The main part of the Development Framework is set up by using a Simulink project. This allows for a comprehensive automation of tasks and management of files (see [34] for further information).

The Simulink project contains a startup script (*scripts/project_ startup.m*) which sets up the needed paths for the Reference Design, adds the Simulink Library and changes the current Matlab path to a work folder. The main Simulink model (*models/framework_ main.slx*) contains the functionality of the Framework and the User Canvases. It is described in the following sections.

Overview over the Model

The Development Framework model is built highly hierarchical. On the topmost level, the main software part is realised. It contains all necessary sections for the framework functionality, especially for the interfacing between the PS and PL part as can be seen in Figure 3.1.

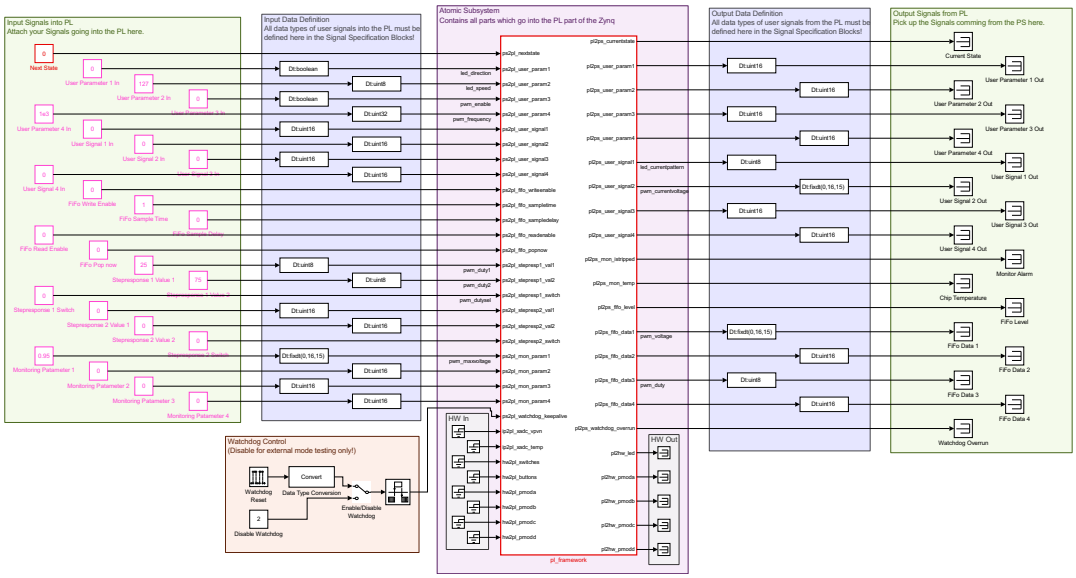

FIGURE 3.1: Model of the PS Part in the Framework

Within the main PS part, the first user canvas for the user software is placed. Also, the atomic subsystem *pl_framework* which contains all PL parts is placed here. This contains the main control state machine, the different states and all other PL functions such as monitoring, the FiFo buffer and the Watchdog. Within the states and the monitoring block, the respective user canvases can be found while the other parts need not to be changed for using the provided functionality.

Control State Machine

To be able to control the system, a basic control state machine is realised within the PL. It consists of four states: Init state, Stop state, Run state and Failsafe state. The goal is to allow the system engineer to prepare different structures for each state he wants to run his system in. Basically, all states are provided with a set of identical inputs and outputs to interact with the surrounding system. Between those, the engineer can realise the desired functionality without having to care for the transition from one state to another.

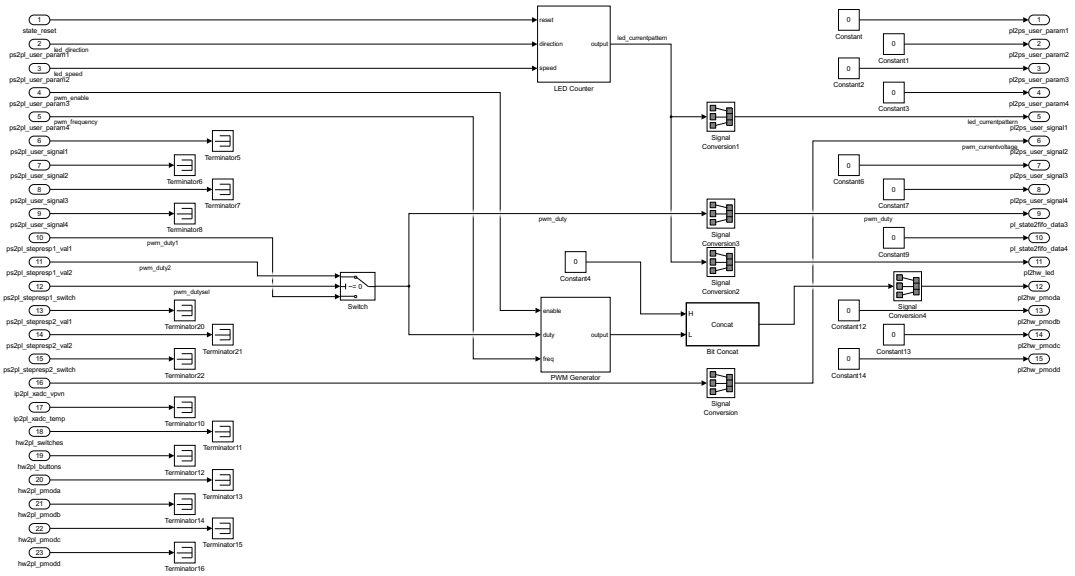

FIGURE 3.2: Canvas of the User Run State, here with the Contents of the Demo Project

The state machine is realised entirely within the PL. This allows for a secure operation even when an error occurs in the PS (time constrains not met, software or system misbehaviour etc.). The next state to be executed is defined by a next state signal from the PS. However, if either the watchdog or the measurement system detect a faulty behaviour, the state machine switches into the Failsafe state, which is locking within the PL. It stays in Failsafe until all alarms from both the watchdog and the monitoring are revoked and the PS initiates a switch to the Init state.

An LED indicator shows which state is active. It is realised in the PL and drives the pin PMODA JA1.

State	LED Pattern
Init State	off
Stop State	on
Run State	blinking slow (1 Hz)
Failsave State	blinking fast (5 Hz)

TABLE 3.1: Blinking Patterns of State LED

Communication with GUI and Hardware

The Framework provides several prepared signals between the various system parts such as the GUI, the PS and PL parts as well as hardware interfaces (be it hardware pins or IP cores). As long as the system uses only parts which are integrated in the underlying reference design, all communication is already implemented in the framework. Communication with the GUI is done in the topmost model level, while the interface to the hardware and IP cores lies on the edge of the atomic subsystem and is accessed from the main PL part.

User Function Canvases

The main part for the system engineer to work in are the User Function Canvases. One canvas is provided within the topmost model level and contains all user software functions. For each of the four states, a canvas for the respective functionality in the PL is provided, as well as one canvas for the Monitoring System. Within these canvases, the system engineer places all his functions. All of them are connected by a set of prepared signals and parameters, which can be used as needed. While the PS and the monitoring canvases are executed all the time, the User State Canvases are only active during the associated state. While all input signals to these four canvases are parallel, the outputs are multiplexed in accordance to the state machine.

Monitoring

The Monitoring System, which resides in the PL, is always active and is to be used to detect unwanted conditions within the system. Several inputs provide data to the user to check the operation parameters and an alarm output allows to set the state machine to the Failsafe mode, in which appropriate measures against the fault can be taken.

FiFo Buffer for Fast Signal Analysis

To enable the analysis of fast signals and step responses on parameter changes, a FiFo buffer is implemented into the PL. It records the data provided by the user into block ram structures and can be read out at a slower pace afterwards. Within the Framework, the triggering and pacing mechanisms for recording data are provided. The FiFo capture is activated by enabling it and changing one of the parameter switches. The FiFo will now run until it is full, even if a state transition occurs. This allows for debugging transitions into Failsafe as well. The signals to be recorded into the FiFo must be provided by the user states.

The pace at which the FiFo buffer records data can be set by the system engineer from the PS side, as well as the delay for the parameter switch. Since this requires a complex logical structure, a certain delay always occurs when the FiFo is triggered. Table 3.2 gives an overview over the timeouts and delays.

The readout of the FiFo is handled by the PS, in conjunction with the Remote GUI. For the readout, the PS supplies the FiFo with a base clock, to which on each edge a new set of data is transmitted.

Sample Time	1	1	1	2	2	2
Sample Delay	0	1	10	0	1	10
1st FiFo Push	2	2	2	2	2	2
Switch Transition	2	3	12	2	4	22

TABLE 3.2: Sample Delays on the FiFo Trigger (1 sample corresponds to 10 ns for a PL clock frequency of 100 MHz)

Watchdog

The PL state machine is equipped with a watchdog. It makes sure that the independent running PL part of the Zynq notices when the collaborating software in the PS doesn't work as expected anymore.

The PL software needs to send a keep alive signal periodically. If this doesn't happen for a defined time, the watchdog sets the PL state machine into Failsafe mode, which in itself is locking (see section 3.2.2). Therefore, a simple return of the keep alive signal isn't sufficient for recovery, but a transition from any state to the Init state must occur to free the system again. This ensures that the transition into the Init state can not happen if the PL system receives the order to change to it next by mistake, e.g. from a corrupt parameter storage.

The overrun condition of the watchdog is announced to the PS software in order for proper detection and to initiate the releasing procedures if necessary.

3.2.3 Simulink Library

In the provided Simulink Library in Appendix C, some practical Simulink Blocks are available. Most of them offer functionality for the Remote System Interaction described in section 3.3, but some ease the development of synthesisable Simulink code for the PL part. For example, some Flipflops or Edge Detectors are implemented.

3.3 Remote System Interaction

During the development and implementation phase of the control algorithm, direct testing methods can be applied such as external mode or PIL simulation. To enable an operator to interact with the developed system, some enhancement is needed. For remote interaction with the system, a communication between the target (Zynq) and a remote system (PC) has to be established. This communication is realised as a UDP interface which is already provided by MathWorks. Advantages of a UDP interface are that it is simple, fast and has a small overhead. However, it is a connectionless protocol which means there is per se no guarantee that the sent data reach their target. Since all safety relevant functions are implemented within the PS and PL standalone, a connectionless UDP communication interface is a good starting point for remote system interaction.

The Remote System Interaction part provides various Simulink blocks for different tasks. They are described in detail in the following subsections. This tasks go from simple "Get and Set parameters" to complex tasks like "FiFo control". All described blocks can

be found in the PS part of the provided Simulink library in Appendix C. An overview of the available block is depicted in Figure 3.3.

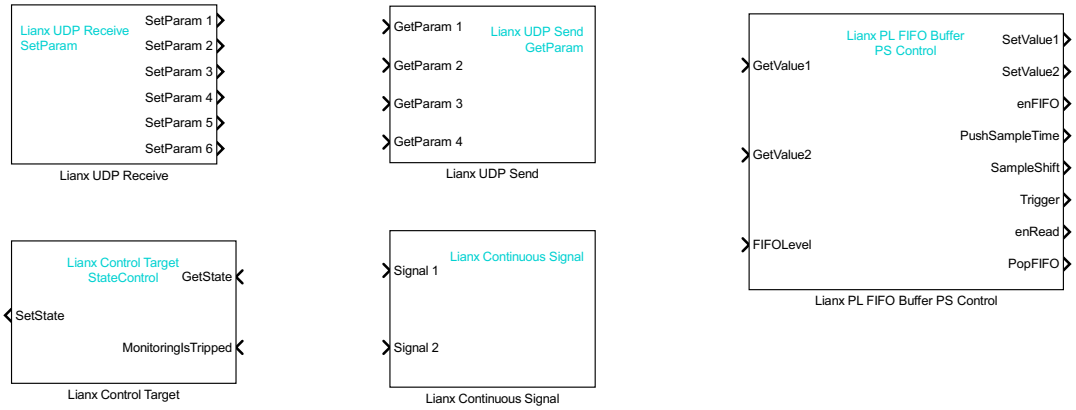

FIGURE 3.3: Remote System Interaction Simulink Communication Blocks

For enhancement of such remote system interaction blocks there are the following three main design tasks.

- Creation and assembly of Simulink Library blocks and their masking.
- Writing of a script to read out fundamental parameters of created library block which are needed for correct GUI interaction afterwards.
- Creation of a GUI module which enables remote system interaction with the designed block.

This is also the order how the blocks developed in this work are presented in the next subsection.

3.3.1 Getting and Setting of Parameters

Lianx UDP Send GetParam

Simulink Block: The aim of the Block *Lianx UDP Send GetParam* is to send parameters from the target to the GUI. The designed block is illustrated in Figure 3.4. To fulfil its task the block needs at least a UDP Socket to send data. This socket block is already realised by MathWorks as an s-function and can easily be deployed to the target. It makes sense to pack some signal together to bring down the network load. This is done by a byte packing block. Since this library should be generic and does not know in advance which data type the signal has, a data type conversion is necessary because only integer data can be transmitted. The width of the signal is limited to 32bit. Since this block is used to get parameters, a sample time of 2 seconds[1] should be sufficient even though the input data can have a smaller sample time. To slow down the transmission, a rate transmission block is implemented.

[1] There is a timing issue with the used table object in Matlab, which has grave implications on the performance of data updates. Therefore its wise to use a sample time > 1 second [3, p. 571].

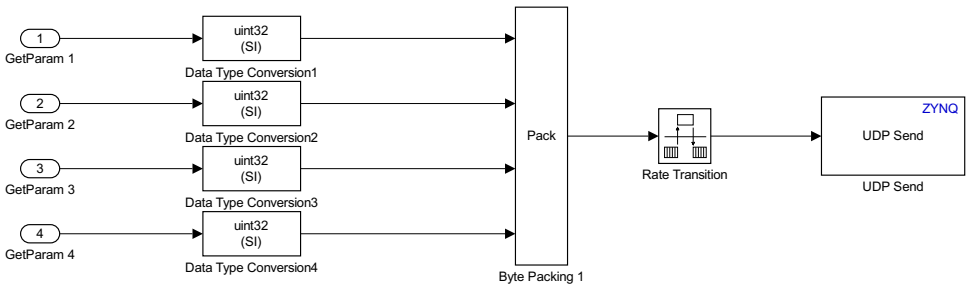

FIGURE 3.4: Get Parameters Block

Simulink blocks can be masked for multiple reasons. One reason is to implement a small documentation of such a designed block that other users know how to use it. Another reason is to hide complexity and present only important parameters to the user which he can set on his own. One can define default values for chosen mask parameters such that the user does not need to set a value by himself. As long as the dependencies are not known the mask should not be changed. Some scripts are relying on these mask parameters which would not work anymore.

API: To interact with the target later on some important information is needed to setup the GUI. Since the library block stays constant except the set parameters of it, all important parameters which need to be known can be read out via a script. All Simulink block information and parameters can be gathered with the following instructions:

```
find_system(<sys>, '<Constraint1>', <Constraintvalue1>);
get_param(<Object>, <Parameters>);
```

For further information about the capability of these functions, the MathWorks documentation is a good reference.

Some parameters like the data type can only be determined when the model is in compile mode. Therefore the model has to be set in it. Then the parameters can be read out and the compile mode terminated afterwards. Interaction with the model is impossible during compile mode.

```
eval([<sys>, '([],[],[],''compile'')']);
get_param(...
eval([<sys>, '([],[],[],''term'')']);
```

All values of the determined parameters are stored in a Matlab variable of class type struct. The structure of the variable is as follows:

```
API (struct)
├── GetParam (struct)
│   ├── BlockName (char)
│   ├── RemoteURL (char), default:  '255.255.255.255', (broadcast)
│   ├── RemotePort (double), default:  25003
│   ├── SampleTime (double)
│   ├── DataType (4x1 cell)
│   └── ParamName (4x1 cell)
└── ...
```

The DataType cell array can only be determined during compilation of the model. It is important to know the datatype for correct reinterpretation of the value in the GUI after it was changed to uint32 and transferred over UDP. The ParamName contains the signal name which defines an appropriate label for the parameter. The definition of a signal name is explained in subsection 4.3.1.

GUI: In the GUI there is a panel for each interaction library block. Each panel is only filled when the corresponding block is used. An init function determines if such a block is available in the API list and if so loads and prepares the specific module. For the UDP transmission from the GUI to the target, the instrument control toolbox of MathWorks is used. To initialise such an UDP object the IP address of the target and the local port of the GUI need to be known. The local port of the UDP object corresponds to the remote port to which the target sends its data. The UDP object listens on the allocated port and creates an interrupt when a specified amount of data is received. This interrupt is known as a callback function in Matlab. This particular callback function reads the received data from the input buffer as uint32 values. For correct display of the received values, the data type needs to be known and a conversion has to be made. When the values are transformed to their correct type, they are written into the prepared table and the user can see them.

Lianx UDP Receive SetParam

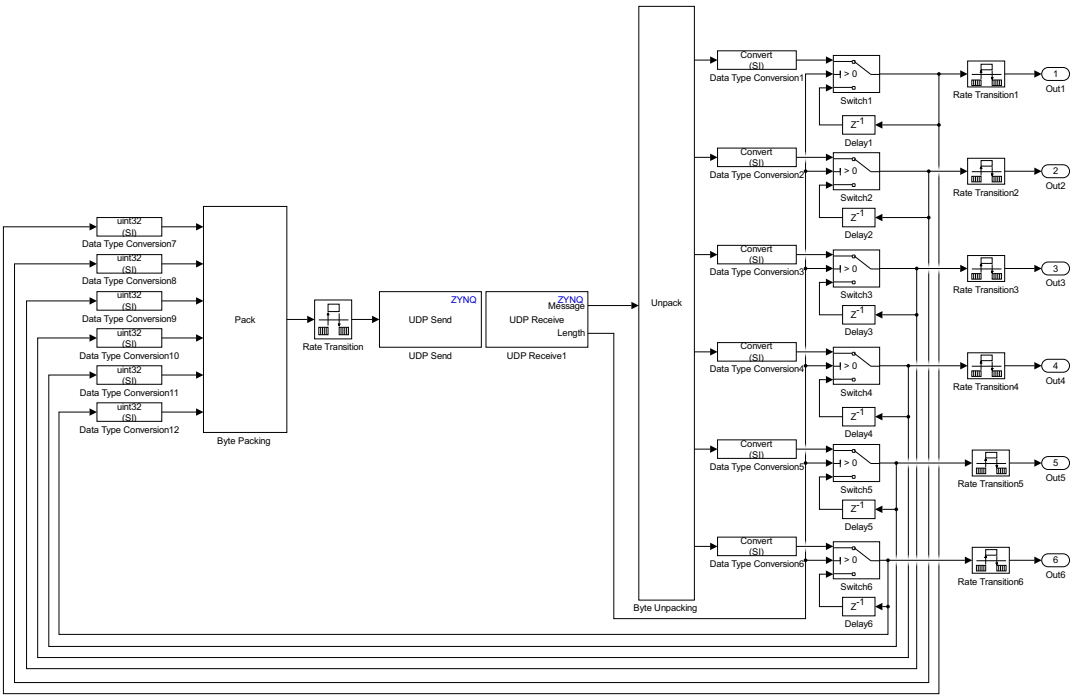

FIGURE 3.5: Set Parameters Block

Simulink Block: The aim of the Block *Lianx UDP Receive SetParam* is to receive parameters from the GUI and send back the current value for control purposes. The designed block is illustrated in Figure 3.5. The read back part of the current values is very similar to the get parameter block. On the other hand there is a UDP Receive block which implements an UDP socket to listen on a specified port. Each time a UDP datagram is received it is unpacked and distributed to the specific outport. Since it is possible that for some signal an initial value is wanted, there is a delayed feedback

of the current value. This is updated by the new set value each time a datagram is received. To fulfil the requirements of the receiving block regarding the sample time a rate transition block transforms this via back propagation.

Hint: There is a parameter called blocking time inside the UDP Receive block, which should be handled with care. It should be kept as short as possible (default: 10 µs) because it blocks the process before it returns control to the scheduler.

API: For the API the same procedure as for the get parameter block is valid. There just are some additional parameters which need to be determined because the set parameter block has a read back capability and can show the current values. Therefore it needs a send and receive block with properties.

```
API (struct)
├── SetParam (struct)
│   ├── BlockName (char)
│   ├── ReceiveRemoteURL (char), default:  '0.0.0.0', (all)
│   ├── localPort (double), default:  25000
│   ├── SendRemoteURL (char), default:  '255.255.255.255', (broadcast)
│   ├── remotePort (double), default:  25001
│   ├── ReceiveSampleTime (double), default:  0.5
│   ├── SendSampleTime (double), default:  2
│   ├── DataType (6x1 cell)
│   ├── ParamName (6x1 cell)
│   ├── Min (6x1 cell)
│   └── Max (6x1 cell)
└── ...
```

GUI: For the GUI part of the set parameter block its almost the same as before. There is some additional functionality which validates if the new set value fits the requirements. These are set by the user via min and max values. Additionally it is checked if the new set values fulfil the data type requirements. If they do not, they are either set back to the value before or rounded to an appropriate fixpoint value if the data type requires so.

3.3.2 Logging of Slow Time Series

Simulink Block: The aim of the block *Lianx Continuous Signal* is to continuously send slower signal to the GUI. The block is depicted in Figure 3.6. A rate transition block, which can be set to a specific value, assures a periodic data flow. A buffer captures a certain amount of data which are packed later on with a byte packing block. The reason for the buffering and packing is to reduce the UDP overhead and send bigger datagrams. Another benefit is on the GUI side; processing a few bigger data packets is faster then a lot of small ones.

API: For the API there is a SigBufferSize parameter which sets the size of the signal buffer. For a good signal buffer size there are some things to consider. Since the transferred data are of datatype uint32 there are 4 bytes for each sent value. Due to the two signal lines there are 8 bytes for each sample. It is good practice that a UDP package should not exceed 512 bytes. Therefore each buffer has 64 values to capture before sending a datagram. The Signal buffer slows down the update rate in the GUI. If the sample time is 0.01 s then each 0.64 s a datagram package is sent.

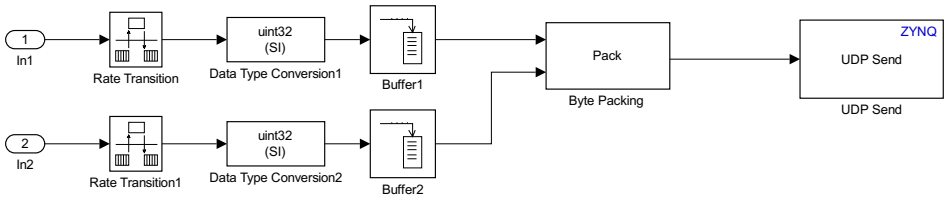

FIGURE 3.6: Continuous Signal Block

```
API (struct)
└─ ContSig (struct)
   ├─ BlockName (char)
   ├─ RemoteURL (char), default:  '255.255.255.255', (broadcast)
   ├─ RemotePort (double), default:  27001
   ├─ SampleTime (double), default:  0.01
   ├─ SigBufferSize (double), default:  64
   ├─ DataType (2x1 cell)
   ├─ ParamName (2x1 cell)
   └─ ...
```

GUI: The GUI part of the continuous signal is designed such that the x-axis is set to a range of 10 seconds per default. From 0 to 10 seconds its static and starts rolling when the time exceeds 10 seconds. The callback function of the UDP block considers the signal buffer size to split the received data correctly into signal 1 and signal 2. Each time the callback function of the UDP class object is called, the axes in the GUI are updated. All received data is stored in a ring buffer. Before the ring buffer overflows all so far captured data points are stored as a mat-file with date and time as file name. When the continuous signal module is turned off, all remaining values in the ring buffer are also stored. That way the user has access to all data which was acquired while the module was on.

3.3.3 Step Response Analysis of Fast Signals

Simulink Block: The aim of the block *Lianx PL FIFO Buffer PS Control* is to control the FiFo buffer which resides in the PL from the GUI. Additionally it also controls the read back of the captured values into the GUI. The FiFo buffer has some parameters which can be set. The upper part of the block depicted in Figure 3.7 is responsible to set those parameters. The trigger value and the two set values are fed back in order to display them correctly in the GUI. The trigger signal is responsible to switch from set value 1 to set value 2 and vice versa. For the latter there is also the possibility to set an initial value if needed. The lower part of the block is responsible for controlling the read out of the FiFo buffer. When the FiFo is triggered, it starts to capture data and the FiFo level will rise to its maximum (the FiFO buffer size). When it reaches this value, the read out process starts. A pulse generator is needed to pop the FiFo buffer. All popped values are packed to a appropriate size and sent over UDP to the GUI. The packed size is calculated using the following algorithm:

```
maxUDPSize = 512; % byte
nrSignal = 2; % 2 Channel
SignalWidth = 4; % uint32
% determine max SignalBufferSize as factor of FIFOBuffer Size
```

```
% constraint by maxUDPSize
if round(PL_FIFO_Size) == PL_FIFO_Size
    maxFrameSize = maxUDPSize/nrSignal/SignalWidth;
    fact = factor(PL_FIFO_Size);
    BufferSize = 1;
    for n = 0:numel(fact)-1
        if BufferSize <= maxFrameSize
            NewValue = BufferSize*fact(end-n);
            if NewValue <= maxFrameSize
                BufferSize = NewValue;
            else
                BufferSize = BufferSize;
            end
        else
        end
    end
else
    error('Input datatype not valid, must be an integer.');
end
```

A UDP send block is enabled during the readout process. This sends data packages with the pre defined size. When the FiFo level reaches its end (at level 0), it is disabled.

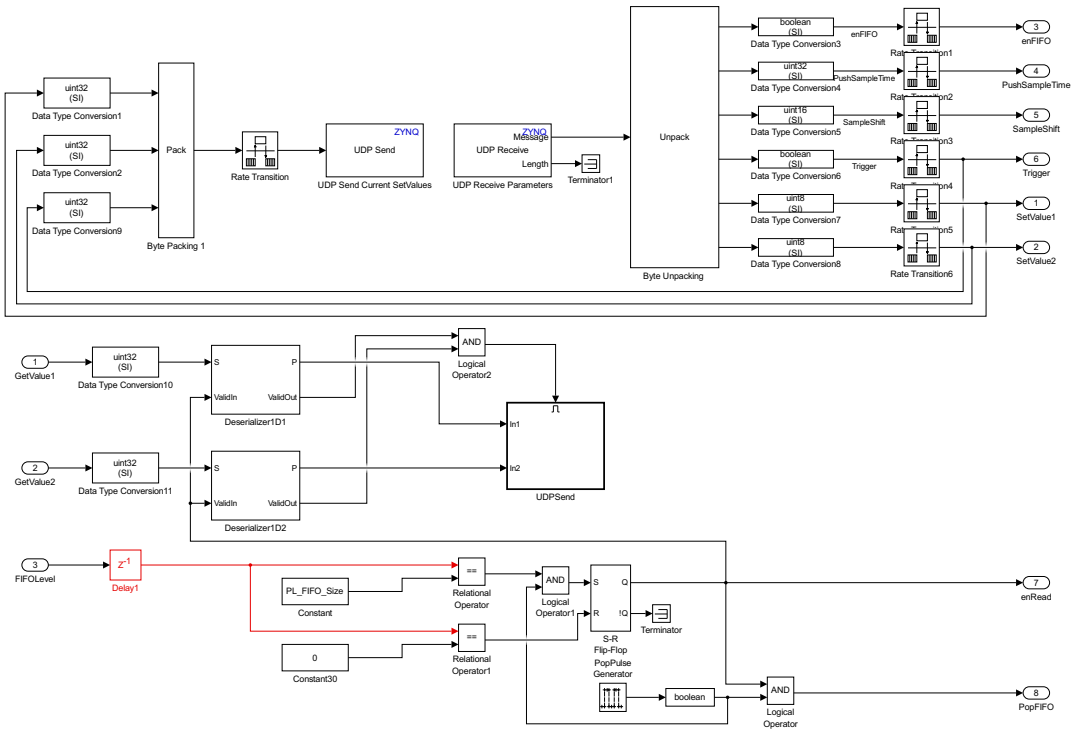

FIGURE 3.7: Step Response, FiFo Control Block

API: The API structure of this block contains some important information about the signal buffer size which the GUI part needs to know. The default pop rate of the FiFO

buffer is 1 kHz which means a FiFo of size 10000 needs 10 seconds to read out. The FiFo buffer channels are read out in parallel.

```
API (struct)
└─ PLFIFOBuf (struct)
   ├─ BlockName (char)
   ├─ FIFORemoteURL (char), default:  '255.255.255.255', (broadcast)
   ├─ FIFORemotePort (double), default:  28003
   ├─ SigBufferSize (double)
   ├─ RemoteURL (char), default:  '0.0.0.0', (all)
   ├─ LocalPort (double), default:  28000
   ├─ SendRemoteURL (char), default:  '255.255.255.255', (broadcast)
   ├─ RemotePort (double), default:  28001
   ├─ SampleTime (double), default:  2
   ├─ FIFOPopSampleTime (double), default:  0.001
   ├─ PLFIFOSize (double)
   ├─ InportDataType (2x1 cell)
   ├─ OutportDataType (6x1 cell)
   ├─ GetParamName (2x1 cell)
   ├─ SetParamName (2x1 cell)
   ├─ Min (2x1 cell)
   ├─ Max (2x1 cell)
   └─ ...
```

GUI: On the GUI side, the FiFo buffer control block needs two UDP objects, one for the set and read back signal and the other to receive the readout values stored in the FiFo. For the readout process the UDP object has an input buffer size of

$$\text{(input buffer size)} = \text{(FiFo buffer size)} \cdot \text{(number of signals)} \cdot \text{(size of datatype)}$$

When all values are received, the UDP object initiates a callback function which updates the trigger data axes and stores the values as a mat-file.

3.3.4 System Control

Simulink Block: The aim of the block *Lianx Control Target* is to remotely control the target. The block is depicted in Figure 3.8. This block is responsible to send next state signals to the target and receives current state and state of the monitoring system. Its very handy to have the state information of the guarded system available. This helps to see if there is a monitoring problem present in the PL part which can't be reset from the PS.

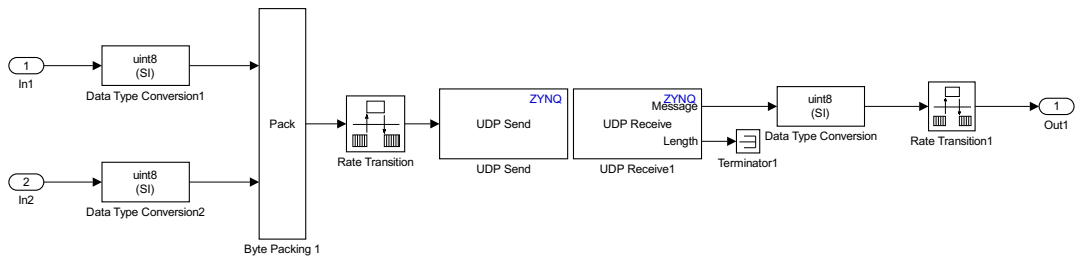

FIGURE 3.8: Control Target Block

API: The most important parameter for the API list in this case is the port informations.

```
API (struct)
└── CtrlTarget (struct)
    ├── BlockName (char)
    ├── ReceiveRemoteURL (char), default:  '0.0.0.0', (all)
    ├── LocalPort (double), default:  26000
    ├── SendRemoteURL (char), default:  '255.255.255.255', (broadcast)
    ├── RemotePort (double), default:  26001
    └── SampleTime (double), default:  1
└── ...
```

GUI: The GUI part of this block consists of three buttons. One to set the system into the Run state, one to set the system into the Stop state and one to reset / initialise the system. The current state is depicted as text and the monitoring state is shown as a coloured field, red for tripped and green for no alarm in condition.

3.3.5 User Interface

The base structure of the user interface is designed using GUIDE. This is a development environment to create GUI's, provided by MathWorks. The GUI could also have been designed using another programming language such as C, C++, Java, ... The reason why it was realised using Matlab is because it was best known to the authors and provides some handy typecast and UDP libraries. A disadvantage of using Matlab is, that it is an interpreter programming language and is therefore not as fast as C or C++ which are compiled languages. The bottleneck of the developed application lies on the GUI side. Some callback function of UDP objects to update incoming data are too time consuming. It seems that the problem is less the amount of data to process, but rather the rate of callbacks. All callback functions can be interrupted by another which then are queued until there is enough CPU time available. There are some hints on the internet to fasten Matlab application using their profiling tools [36] to find the time consuming functions. When these functions are determined one can look for faster solutions and improve performance. The approach during the GUI development phase was:

make it work
make it right
make it fast [17]

Unfortunately there was not enough time to focus on the optimisation of performance.

When the GUI development is finished, there is a possibility of creating a standalone application. Such an application can run without a Matlab license. The only thing which is needed is the Matlab Runtime in the version as the application was created with. In the Matlab version R2015b there is a bug regarding the Matlab application compiler toolbox. The compiled GUI has a different look than during the design phase. To correct this bug there are some lines of code to add in a startup script [21].

Overview of UDP Ports and Signals

The following list gives an overview of the default UDP port specification with its transfer datatype and the data structure itself.

- **Lianx UDP Receive SetParam**

  destination port: 25000
  transfer datatype: uint32
  data: [SetParam1|SetParam2|SetParam3|SetParam4|SetParam5|SetParam6]

  local port: 25001
  transfer datatype: uint32
  data: [CurrentParam1|CurrentParam2|CurrentParam3|CurrentParam4|
  CurrentParam5|CurrentParam6]

- **Lianx UDP Send GetParam**

  local port: 25003
  transfer datatype: uint32
  data: [GetParam1|GetParam2|GetParam3|GetParam4]

- **Lianx Continuous Signal**

  local port: 27001
  transfer datatype: uint32
  data: [Signal1|Signal2]

- **Lianx PL FIFO Buffer PS Control**

  destination port: 28000
  transfer datatype: uint32
  data: [SetValue1|SetValue2|enFIFO|PushSampleTime|SampleShift|Trigger]

  local port: 28001
  transfer datatype: uint32
  data: [Current Value1|CurrentValue2|CurrentTrigger]

  local port: 28003
  transfer datatype: uint32
  data: [GetValue1, GetValue2]

- **Lianx Control Target StateControl**

  destination port: 26000
  transfer datatype: uint8
  data: SetState

  local port: 26001
  transfer datatype: uint8
  data: [GetState|MonitoringIsTripped]

Dynamic Creation

The task of a dynamic creation of the GUI could not be fulfilled due to a shortage of time. But the developed GUI modules are only setup if in the generated API list such a block is referenced, otherwise the module will not be loaded. The imperfection about this solution is that the area stays empty.

Chapter 4

Working with the Development Framework and Remote System Interaction

This chapter describes how to work with the Development Framework and the Remote System Interaction. In section 6.2 a sample implementation using the here presented methods can be found.

4.1 Prerequisites

To be able to use the Development Framework and Remote System Interaction designed in this project, some prerequisites must be met. Since some of the tools change rapidly due to their ongoing development, backwards compatibility might be unreliable. Due to the highly interlinked usage of tools, functionality is only provided if using the given release versions.

- **Matlab/Simulink R2015b**

  HDL Coder Support Package for Xilinx Zynq-7000 Platform 15.2.0

  Embedded Coder Support Package for Xilinx Zynq-7000 Platform 15.2.2

  Embedded Coder Support Package for ARM Cortex-A Processors 15.2.0

  Embedded Coder 6.9

  HDL Coder 3.7

  Simulink Coder 8.9

  Instrument Control Toolbox 3.8

  Fixed-Point Designer 5.1

  Matlab Coder 3.0

  DSP System Toolbox 9.1

- **Xilinx Vivado 2014.4** with SDK (WebPack Edition is sufficient[1])
- **Lianx System Software** running on the ZedBoard (see section 6.1 for setup instructions)

[1] for device limitations see [54].

- **Development Framework**
- **Lianx Libraries**

Also, it is crucial that the system engineer has a basic understanding of embedded systems and the challenges of their development. Although the Development Framework tries to ease the development process, not all obligations can be taken over. For example, the engineer needs to know about the advantages and disadvantages of implementing certain functions in software or programmable logic. Failure to understand the basic principles and executing an intensive examination of the framework will have serious impacts on the resulting system. Not all these steps can be covered in this Thesis, but the reader is highly encouraged to dive into the explanatory chapter 5 and the additional literature in Appendix A, as well as other sources of information about embedded system engineering.

4.2 Development Framework

This section outlines the usage of the Development Framework and its adaption to the system engineer's needs. To help the understanding of the framework and to demonstrate the basic functionality, a sample implementation was developed. It is described in section 6.2, as well as the steps needed to set it up.

The framework as is provides some basic functionality and tools. A fully generic framework which covers all needs is not feasible. But in its basic setup, it should be able to handle most scenarios with only minor modifications. Hints on modifying the framework are given in subsection 4.2.11.

The basic workflow is as follows:

- The system engineer designs his system and informs himself about the provided functions and canvases of the framework (subsection 4.2.1 to subsection 4.2.5).
- Then the User functions are implemented into the framework. This process is described in subsection 4.2.6.
- Subsection 4.2.7 describes the simulation of the whole system.
- After the functionality of the system was tested in simulation, the PL bitfile is created and programmed onto the Zynq FPGA (subsection 4.2.8).
- Now the system can be tested on the target in realtime as described in subsection 4.2.9.
- Some hints on the performance analysis of the PS part are given in subsection 4.2.10.

4.2.1 A Note on Signals and the PL Clock Frequency

Simulink is a data flow based modelling technique. Therefore, to transport data between the different blocks of the framework, signals are used. All signals the user has to deal with are described in the section about their respective functionality. For a better overview, the signal names are strictly hierarchical, beginning with the notation of their origin and target, followed by its functionality and finally a distinctive name. *ps2pl_user_param3* for example denotes a signal which transports the third parameter

from the PS to the PL user state which the system engineer can use for whatever he desires. *pl_ mon2fifo_ data2* is the second data source for the FiFo block, comming from the Monitoring system within the PL.

While the PS can execute its code at a predefined rate, the PL clock is set in the Reference Design. Also, the transfer rates of the signals transmitted between the PS and PL over the AXI interface need to be defined. Both those rates are set with Simulink workspace variables:

- **pl_axi_freq** (default: 1000 ≙ 1000 Hz)
  Defines the transfer rate of the signals between the PS and PL part.
- **pl_clock_freq** (default: 100'000'000 ≙ 100 MHz)
  Indicates the PL clock frequency set in the Reference Design. Used to calculate time steps within PL blocks and control the rate transitions for simulation.

Since the Zynq uses 32bit ARM processors for the PS, operations with a higher word width are strongly discouraged. While such are possible, they have a severe impact on performance. Also note that Simulink supports 32bit wide data over the *AXI*-Interface only, so staying within the 32bit limit is recommended. The use of floating point signals is not possible on the PL and *AXI* interface, so only integer and fixed point data can be used. For more information about fixed point design see also [26].

4.2.2 Prepared Peripherals

To interact with the hardware, the following peripherals are prepared for use:

- **ZedBoard LEDs, Switches and Buttons**
  The eight LED on the ZedBoard are available as an output signal, as well as the eight switches and fife buttons are as inputs.
- **Zynq XADC**
  The Zynq XADC, as well as the needed data readout algorithms, are implemented into IP cores in the reference design. The differential voltage measurement between the pins *Vp* and *Vn* of the XADC header are provided, corresponding directly to measurements between 0 V and 1 V. Also, the temperature of the chip is available as a value in °C.
- **ZedBoard PMODs**
  PMODA to *PMODD* are available as both input and output signals. See subsection 4.2.8 for the mode selection.

The signals to and from the hardware and IP cores are as follows:

- **ip2pl_xadc_vpvn** (ufix16_en15: 0 ≙ 0 V, 1 ≙ 1 V)
  Supplies the measured Voltage between Vp and Vn on the XADC header.
- **ip2pl_xadc_temp** (sfix16_en6: value in °C)
  Supplies the measured chip temperature.
- **hw2pl_switches** (uint8)
  Supplies the value of the switches on the ZedBoard.
- **hw2pl_buttons** (ufix5)
  Supplies the value of the buttons on the ZedBoard.

- **hw2pl_pmod<a-d>** (uint8, except for PMODA which is ufix7[2])
  Depending on the configuration, these signals provide or accept signals to or from the PMOD headers PMODA to PMODD. If used as an output, the signal direction reverses to *pl2hw_pmod<a-d>*.
- **pl2hw_led** (uint8)
  Output to the ZedBoard LEDs.

Note that the use and connection of the PMODs is done while setting up the synthesis of the PL hardware description. See subsection 4.2.8 for further information about that step. The addition of further IP cores and other peripherals can be achieved by modifying the underlying Reference Design of the framework. This process is described in section 6.3. Some hints on the adaption of the framework are given in subsection 4.2.11.

4.2.3 User Function Canvases

First of all, the system engineer needs to outline which parts of his system shall reside in which part of the framework. For slower processes (e.g. 1 kHz), the PS canvas is the right choice, while critical and fast functions (e.g. 100 MHz) should reside in the PL whenever possible. Then, on the PL side, he has to consider what functions need to be implemented where. What is needed to reach a secure state of the system and how can this be implemented in the Failsafe state? What signals are needed? Which parameters should be analysed and recorded for a closer inspection with the FiFo buffer? Only when all these design steps are dealt with, the implementation can begin.

User PS Canvas

In the User PS canvas, the system engineer can place the processes which should reside in the PS. The canvas is provided with several signals to and from the PL. Most importantly, the signal *ps2pl_nextstate* controls the systems state machine. Note that once the state machine is in Failsafe mode, only a transition to the Init state can release it again.

The sample rate of the PS functions is set in the model blocks. If they inherit it from the model, it is defined in the Model Configuration Parameters which per default is set to 1 kHz. Simulink tries to match those rates, but this may not always be possible. Please refer to [25] for further information. Note that a higher frequency, combined with more complex tasks, leads to a considerably higher system load (see also subsection 4.2.10). The sample rate of the AXI communication however is fixed to the value it was set in subsection 4.2.1.

Involved signals for the exchange of data between the User PS canvas and the PL User State canvases, as well as other purposes are as follows:

- **ps2pl_nextstate** (uint8: 0≙Init, 1≙Stop, 2≙Run, 3≙Failsafe)
  Defines which state should be entered next.
- **pl2ps_currentstate** (uint8: 0≙Init, 1≙Stop, 2≙Run, 3≙Failsafe)
  Indicates in which state the PL state machine is currently in.

[2] By default PMODA is configured as an output and JA1 is used for the status LED.

- **ps2pl_user_param<1-4>** and **ps2pl_user_signal<1-4>** (user defined[3])
  These two times four signals can be used to exchange data between the User PS canvas and the User state canvases.
- **pl2ps_user_param<1-4>** and **pl2ps_user_signal<1-4>** (user defined[3])
  These two times four signals can be used to exchange data between the User state canvases and the User PS canvas. The signals are multiplexed according to the current state.

It is considered to be a good idea to define all signals with their data types and ranges at the beginning and to enter them into the signal definition blocks in the User PS canvas. This ensures a consistent signal flow throughout the framework.

User State Canvases

For each of the four states, a canvas for the respective functionality in the PL is provided. Here, the system engineer places all his functions to be residing in the FPGA. Note that whenever a signal shall simply be passed through a User State, a signal conversion block with the signal copy option must be inserted. This is due to a Simulink feature and yields a warning if not considered.

Besides all signals described already under the User PS canvas and other dedicated functions, the User state canvases are provided with one special signal:

- **state_reset** (bool, high during one cycle when state was entered)
  This signal indicates when a state was entered. It can be used to reset counters or other systems like control loops, estimations etc.

Monitoring Canvas

The Monitoring canvas offers a way of checking the system's status and reacting if something goes wrong. It resides in the PL and is executed constantly. As input signals, four parameters can be passed from the PS canvas and all hardware signals are available as well. An alarm output puts the system into the Failsafe state if activated. Also, two signals can be passed on to the FiFo buffer for closer inspection.

Using those signals , as an example a current measurement by the XADC can be compared to a defined maximal value. If it is exceeded, the alarm output can be set and the Failsafe state is entered. To analyse the current measurement in detail, it can be passed on to the FiFo buffer.

- **ps2pl_mon_param<1-4>** (user defined[3])
  These four signals can be used to supply the Monitoring functions with parameters.
- **alarm** (bool: 0≙ok, 1≙alarm)
  Output of the monitoring block to signal an alarm condition.
- **ps2pl_mon_istripped** (bool: 0≙ok, 1≙alarm)

[3] The data types of these signals can be defined by the user. This is done within the PS canvas or the PL canvases and will be propagated throughout the framework. Non-matching definitions will yield a warning from Simulink. Note that for the used *AXI*-interface, the maximal width of the data type supported by Simulink is 32 bit and that floating point data is not supported in the PL.

4.2.4 Usage of the FiFo Buffer

The FiFo buffer is meant for capturing fast events in the system for closer inspection using the Remote GUI. The main application is analysing the step response upon a parameter change. Therefore, it is set up to work best for exactly that. From the PS User canvas (and/or the GUI), two sets of step response parameters are available. Each consists of two values and a selector to determine which value is currently active. Is the FiFo buffer write enabled and a transition occurs on one of the selectors, the FiFo is triggered and the passed data is recorded until the buffer is full. This even keeps on going if a state transition occurs.

As an example, the step response on a target speed value should be evaluated. Two speed values can be set from the PS side and within the User Run state, a switch controls which value is active. The switch is controlled by the selector signal. The FiFo buffer is fed with the selected target speed value from within the User Run state and the measured effective speed from within the user Monitoring canvas. Switching from one value to the other triggers the FiFo buffer and the reaction of the system on the parameter step can be analysed. To indicate any switch to another state during capturing, the target speed value fed to the FiFo buffer by other states can be set to an unreasonable value which then will be recorded.

Several signals are needed to use the FiFo buffer:

- **ps2pl_fifo_writeenable** (bool: 0≙disabled, 1≙enabled)
  This signal controls if the FiFo should record data on the next parameter switch.
- **ps2pl_fifo_sampletime** (uint32: 1≙10ns, 100'000'000≙1s)
  Defines at which rate the FiFo records samples. The number is given in steps of 10 ns. Consider that large numbers lead to very long recording times.
- **ps2pl_fifo_sampledelay** (uint16: 0≙no delay, 10'000≙10'000 samples)
  Defines for how long the parameter switch is delayed for pre-switch recording. The number is given in samples.
- **ps2pl_fifo_readenable** (bool: 0≙disabled, 1≙enabled)
  This signal controls if the FiFo should output recorded data upon the next edge of the *ps2pl_fifo_popnow* signal.
- **ps2pl_fifo_popnow** (bool: both edges triggered)
  If reading is enabled, the FiFo outputs a new set of samples on each edge.
- **ps2pl_stepresp<1-2>_val<1-2>** (user defined[3])
  These values are used to supply the user function with a parameter which should be switched. For each stepresponse block, two values are provided between which can be switched.
- **ps2pl_stepresp<1-2>_switch** (bool: 0≙val1, 1≙val2)
  Defines which of the two parameter values is active. A switch triggers the FiFo if enabled.
- **pl_state2fifo_data<1-2>** (user defined[3])
  Actual data sources for two of the FiFo buffer channels from the user states.
- **pl_mon2fifo_data<3-4>** (user defined[3])
  Actual data sources for two of the FiFo buffer channels from the monitoring system.

- **pl2ps_fifo_level** (uint16, length depending on the FiFo buffer size)
  Indicates the current FiFo buffer level.
- **pl2ps_fifo_data<1-4>** (user defined[3])
  Output of recorded data.

Also, one Simulink workspace variable defines parts of the FiFo buffer for its synthesis:

- **pl_fifo_size** (default: 10'000)
  Defines how many samples each channel of the FiFo buffer can hold. Consider that the number of block rams available is limited. Four FiFo buffer channels with 10'000 samples each, use 25 % of the available block rams.

4.2.5 Usage of the Watchdog

The Watchdog makes sure that the PL notices when something goes wrong on the PS side of the system. For example, when shutting down the system software, the PL keeps on running without any means to stop it. The Watchdog needs to be reset periodically from the PS, or it will put the system into the Failsafe state after a timeout. In the PS, a pulse source is set up which sends a positive edge to reset every second base cycle (of 1 ms). Therefore, the minimal value which works is 3ms due to the delays in signal transports.

- **ps2pl_watchdog_keepalive** (uint8)
  This signal causes the watchdog to reset upon each transition from 0 to 1. The value 2 disables the watchdog altogether.
- **pl2ps_watchdog_overrun** (bool, 0≙ok, 1≙ Watchdog is currently triggered)
  This signal flags the current state of the Watchdog.

One Simulink workspace variable defines the timeout of the Watchdog for its synthesis:

- **pl_watchdog_timeout** (default: 10, minimum is 3)
  Defines the watchdog timeout in milliseconds.

4.2.6 Implementation of the User Functions

Now that the system engineer knows about the framework functions and canvases, it is time to implement the system into it. Start by making a copy of the supplied folder structure to work in. The source files are provided in Appendix C. Folders 2 to 3 and 5 are needed and referenced within each other, so keep the structure as it is. Connect the ZedBoard, running the Lianx System Software (setup see section 6.1). Now open the Development Framework Simulink project (*DevelopmentFramework.prj*), which initialises the Matlab environment and the connection to the ZedBoard. Open the User Canvases with the first Simulink Project shortcut which can be found in the upper left corner of the Projects Shortcuts tab in Matlab. The custom functions can now be implemented into the six canvases.

4.2.7 Simulation of the whole Model

When all functions are implemented and compiling the model (by pressing [Ctrl] + [D]) yelds no errors, the whole model can be simulated. To make sure all PL parts are

simulated correctly, the step size is set to the *pl_ clock_ freq*. Since this is automatically changed in the next workflow steps, always enter the model with the first project shortcut. Note that simulations of the whole system can take a long time due to the small step size.

When everything works as expected, safe the model and close it. It is now ready to be implemented onto the Zynq.

4.2.8 Synthesis, Code Generation and Deployment

The next step is to generate the HDL code and bitfile for the FPGA part of the Zynq. Make sure all models are saved and closed, then use the second Simulink Project shortcut to start the HDL Workflow advisor. Follow it until the last step. Note that the script also sets the frequency with which data is transmitted over the AXI bus as defined in subsection 4.2.1.

Step 1: General setup and connection of the signals

In step 1.2 of the Workflow Advisor, the signals to and from the PL are set up. All signals to and from the PS are set to use the AXI4 Lite bus and hardware is connected to its dedicated signals. Each PMOD on the ZedBoard can be either used as an input or an output. The effective direction of each PMOD is determined by its assignment in step 1.2. For example, if PMODA should be used as an output, select "No Interface Specified" in its "Input" column and "Pmod Connector JA1 [0:7]" in its "Output" column as shown in Figure 4.1 and Figure 4.2.

hw2pl_pmoda	Inport	uint8	No Interface Specified
hw2pl_pmodb	Inport	uint8	No Interface Specified
hw2pl_pmodc	Inport	uint8	No Interface Specified
hw2pl_pmodd	Inport	uint8	No Interface Specified

FIGURE 4.1: Connection setup for PMODs as Inputs

pl2hw_pmoda	Outport	uint8	Pmod Connector JA1 [0:7]	[0:7]
pl2hw_pmodb	Outport	uint8	No Interface Specified	
pl2hw_pmodc	Outport	uint8	No Interface Specified	
pl2hw_pmodd	Outport	uint8	No Interface Specified	

FIGURE 4.2: Connection setup for PMODs as Outputs

Step 2: Preparation of the Model

In this step, the model is checked for its compatibility for HDL code generation. Carefully observe the error messages and correct the reported issues.

Step 3: HDL Code Generation

When the model passed all checks of step 2, the HDL code is generated.

Step 4: Integration into the Embedded System

In the last section, the code generation is finalised. Save the inferface model generated in step 4.2 in the work folder of the Development Framework with the proposed file name, it will be used later. In step 4.3, the HDL code is used to generate the bitfile for the FPGA. Note that this process is done in an external console window. Wait until it signals that it is done and can be closed. Also watch out for error reports there. In the last step, the generated bitfile is programmed onto the Zynq.

4.2.9 Realtime On-Target Testing

Now that the PL functions run on the Zynq, the PS software can be created and loaded onto the target. To do so, use the third Simulink shortcut which opens the interface model created before.

In this model, the whole PL subsystem is now swapped for an AXI interface block which enables communication between the PS and PL. The complete system can now be run in the External Mode, meaning that interaction with constants, switches, scopes etc. in Simulink is still possible. But the system actually runs on the Zynq target and communicates with Simulink over the Ethernet interface. This allows for realtime system testing with all functions and states, hence providing a comprehensive verification of the desired functionality.

Due to an error in Simulink[4], the Watchdog must be turned off when using the External Mode. Set the corresponding switch in the PS accordingly. However, it is recommended to enable the watchdog after using the external mode and moving on to using the Remote System Interaction.

4.2.10 Performance of the User PS Functions

After the implementation, the system engineer should make sure that the PS can handle the User functions residing in it. If they are too complex, the time constraints can't be met and the system's performance is compromised. To do so, the system engineer should check the load of the complete system. (Note that if the functions of the Remote System Interaction are used, these also add a small load to the system and should be implemented as well when checking the load.) Basically, this is done by logging on to the running system (see section 6.1) and run the *top* command. This lists the average system load in three values: the first one for the last minute, the second and third one for the last 5 and 15 minutes respectively. A value of 2 (because of the two cores of the Zynq) means that the PS can perform its tasks just barely, but still in time. Lower values are better and indicate sufficient performance. If the load is higher, the PS can't handle its assigned tasks and some functionality should be moved into the PL part of the Zynq.

[4] While all PS functions are executed on the target while running in External Mode, the change of a parameter from Simulink can cause a short pause in the PS function execution. Such a delay can inhibit the timely reset of the watchdog and cause it to overrun. The pause might be due to the transmission of the parameters from Simulink to the target via TCP/IP, which can cause a delay in the millisecond range with its acknowledge procedures and network timeouts. The actual reason for this sporadic behaviour could not be identified, but is limited to parameter changes via the external mode only. The change of parameters via the developed System Interaction method is not affected, even when running in parallel to the External Mode.

For a more detailed explanation of the Linux system load indication, see [43]. Also see section 5.2 which lists possible performance tweaks and for information about the task scheduling. Consider that Simulink already increases the priority of the tasks performing the model functionality to values in the realtime range.

4.2.11 Notes on Modification outside the User Canvases

The Development Framework was set up with the intention to demonstrate the capabilities and to provide a basic functionality. Since every specific project has different demands and challenges, the framework most likely needs to be adapted to fit all needs. Modifications can range from the simple addition of signals and change of parameters, over more complex adaptions like adding more or different states up to complete rebuilds of extensive parts. To describe all possible improvements, extensions and modifications would certainly go beyond the scope of this Thesis. However, it might be a good idea to start by familiarising with the concepts and solutions implemented in the framework and the Demo Project to get started. Many simple adaptions can be done by simply look "how things are done" and imitating them.

4.3 Remote System Interaction

This section describes the preparation of the framework for control by the system engineer. Further it explains the usage of the Remote System Interaction software by the operator.

The basic workflow is as follows:

- Implementation of the GUI parts in the PS, mainly with provided library blocks
- Generation of standalone application for target and readout of the API parameters
- Temporary or persistent deployment of PS and PL code
- Execution of the GUI

The first three items address the system engineer while the last one address the operator.

4.3.1 Usage of the Lianx Simulink Blocks

Once the Development Framework tasks are done, the system engineer can start to prepare the model for remote control. First a decision which functionality should be available to the operator has to be made. There are different library blocks in the PS part of the provided Lianx library. Mainly the following tasks can be made available to an operator with the provided blocks:

- Set and get parameter
- Monitor slow continuous signal (up to 100 Hz)
- Control Target (Run, Stop, Reset)
- Trigger step response of fast PL signal (FiFo buffer)

A brief explanation follows in the next subsection. The provided functionality can of course be extended. A good starting point for an extension can be found in section 3.3. Note that each block can only be once in a model so far.

Lianx UDP Receive SetParam

The set parameter block can be used like a normal constant block. The block enables the operator to change chosen parameters remotely. Within the provided block there are six set parameters available. These are also read back into the GUI, so that the user knows the new set parameters reached the target. For ease of use, the block is masked and only relevant parameters are shown to the user. The mask is depicted in Figure 4.3. There are two tabs of different settings. It is important not to use floating point datatypes (single, double), because those are not yet supported by the GUI. Another important point is to set a range (min, max) for the values. If no range is set, it is assumed that the value is not used. An initial value can be set as well. To provide the operator with the name of the parameter, it is important to assign a signal name to each port. This can be done by double clicking on the connection line and entering a name. Regarding the IP port configuration the user is advised to use the default settings. So far there is no possibility on the GUI side to set local IP addresses of the PC. However the destination IP address for packages sent from the target can be set here. Another important point is the sample time. Since there are some update

time issues [3, p. 571] with the update procedure of the used table in the GUI, it is wise to use a sample time > 1 second.

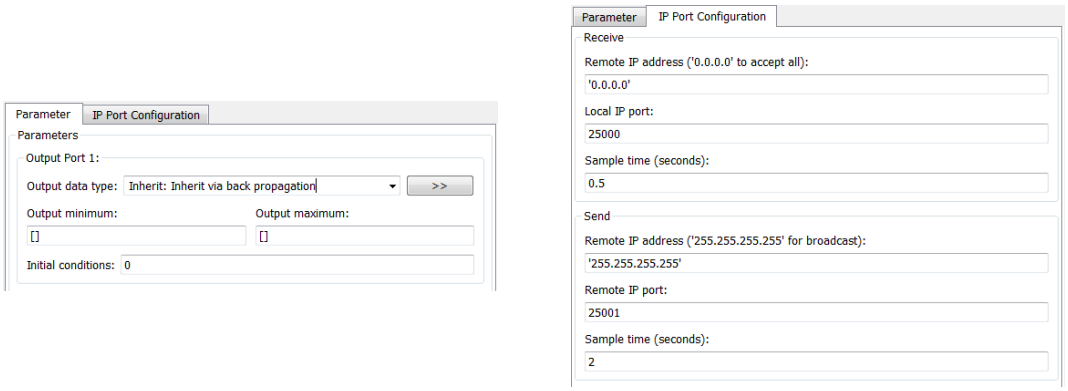

FIGURE 4.3: Mask of Lianx UDP Receive SetParam Block

Lianx UDP Send GetParam

The get parameter block can be used like a Simulink display block. The system engineer can decide which parameters he wants to make available for the operator. Regarding the datatypes the same rules apply as before. The bitwidth of the datatype is limited to 32bit. To ease the use of the block it is masked as depicted in Figure 4.4. To provide the operator with the parameter name, it is advised to label the connection line. Regarding the IP port configuration the default settings should be used.

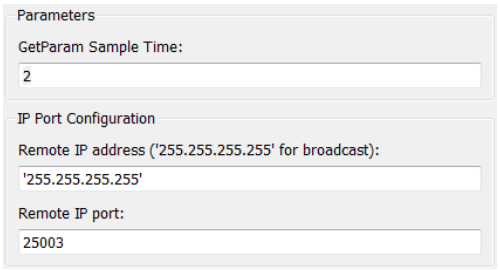

FIGURE 4.4: Mask of Lianx UDP Send GetParam Block

Lianx Continuous Signal

The continuous signal block can be used to monitor continuous signals with a sample rate of about 100 Hz. The system engineer can decide which signals to connect. The datatypes are limited as before. The block mask is depicted in Figure 4.5. To provide the operator with the signal name it is wise to label the connection line. The buffer size is another important parameter to set. The system engineer has to consider that the GUI update rate should be less then twice per second. Since there are two signals with a width of 32bit this adds up to 8 byte per sample. A UDP datagram should not exceed the size of 512 byte. Considering this, a buffer size per signal of about 64 samples at a rate of 100 Hz is a good choice. That means the update time of the GUI is every 0.64 s with this configuration.

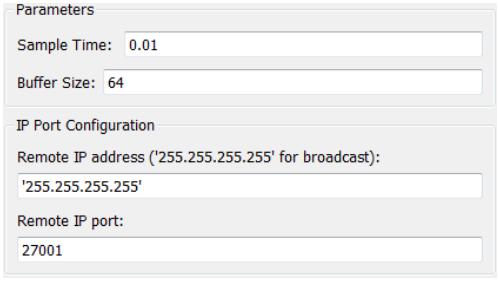

FIGURE 4.5: Mask of Lianx Continuous Signal Block

Lianx Control Target StateControl

The control target block can be used to set the next state. Additionally this block reads back the current state and the status of the monitoring system. The block is masked as depicted in Figure 4.6. It enables an operator to start, stop and reset the system.

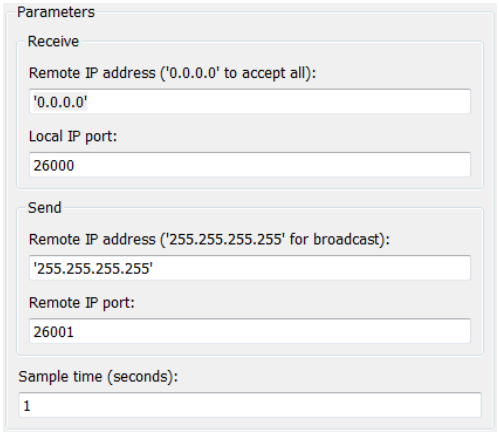

FIGURE 4.6: Mask of Lianx Control Target StateControl Block

Lianx PL FIFO Buffer PS Control

The FiFo buffer control block can be used to control the FiFo buffer which resides in the PL. With this block it is possible to control and readout two FiFo buffers implemented in the Development Framework. To ease the use of this block it is masked as depicted in Figure 4.7. To provide the operator with the signal and param name its wise to label the connection lines of the set and get ports. Another important parameter is the FIFO Size, it has to be set to exactly the same value as the FiFo. Normally this should be done via the model workspace variable *pl_fifo_size*. In the Set Parameter tab the system engineer can choose a specific datatpye for the set values and also an initial condition. It is very important to set a range (min, max) for the set values. Since the set values are illustrated using table object in the GUI, there are the same timing constraint as mentioned before with the set and get blocks.

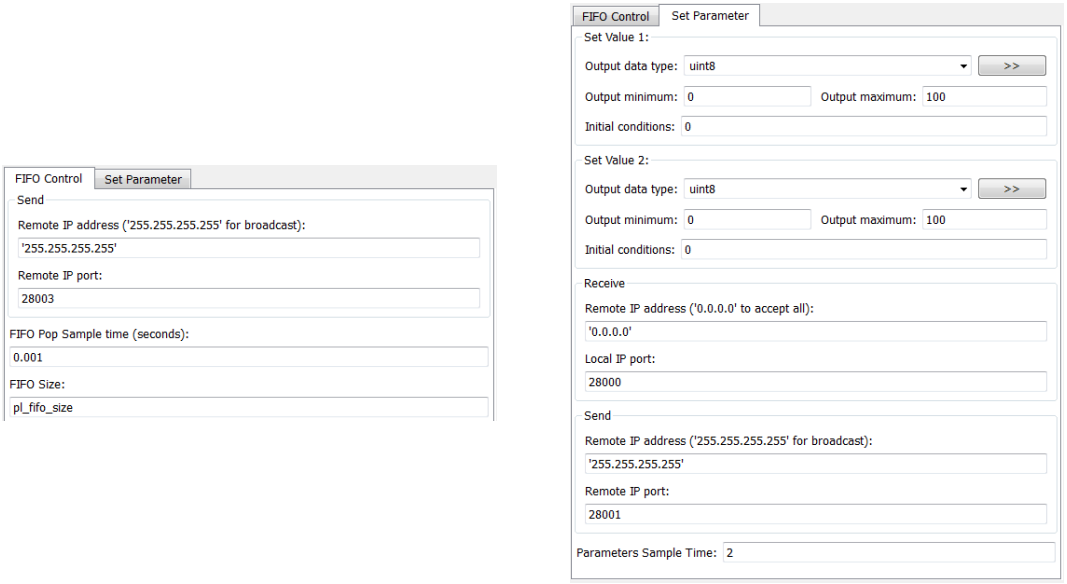

FIGURE 4.7: Mask of Lianx PL FIFO Buffer PS Control Block

4.3.2 Generation of the PS Executable and Readout of API Parameters

The system engineer can go ahead with code generation after extending the developed system with the provided GUI blocks where needed. The process of code generation can be executed by the shortcut *04 - Generate PS Software and GUI API*.

The script produces an output of the following two files which are stored in the work folder of the Development Framework:

- Executable: <modelName>
- API file: API_Data_<modelName>.mat

4.3.3 Deployment of the Generated Code

To deploy the generated code temporarily the *05 - Deploy PL and PS code temporarily* shortcut can be used. The user is prompted to choose the related bit-file which is also programmed on the target. The executable and bit-file will be put in a temporary folder on the target and hence are not available after reboot. For persistent deployment see subsection 4.3.4.

4.3.4 Persistent User Function Deployment

To deploy the generated code persistently the shortcut *06 - Deploy PL and PS code persistently* can be used. The user is prompted to choose an executable and a related bit-file for the target. Both files will be deployed in the persistent folder *Model* on the SD card. They also will be executed automatically upon boot. To revoke the persistent deployment the shortcut in the Development Framework can be used.

4.3.5 Usage of the GUI

This subsection addresses the operator of the system which controls the target using the GUI. The following two files are needed to successfully run the GUI:

- Executable GUI: RemoteSystemInteraction.exe
- API list: API_Data_<modelName>.mat

After starting the GUI by double clicking on the executable, the screen displayed in Figure 4.8 appears.

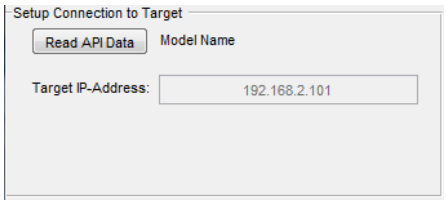

FIGURE 4.8: GUI Start Screen

When the operator clicks on the button *Read API Data* he is asked to choose a new API Data file. This file is provided by the system engineer after code generation. When such a file is chosen there is an initialisation process and the GUI shows all modules which are available in the API Data file. Figure 4.9 shows an example. The operator can check if the IP address of the target is set correctly, the default value is *192.168.2.101*. He has to be aware to change the IP address only when all modules are disabled.

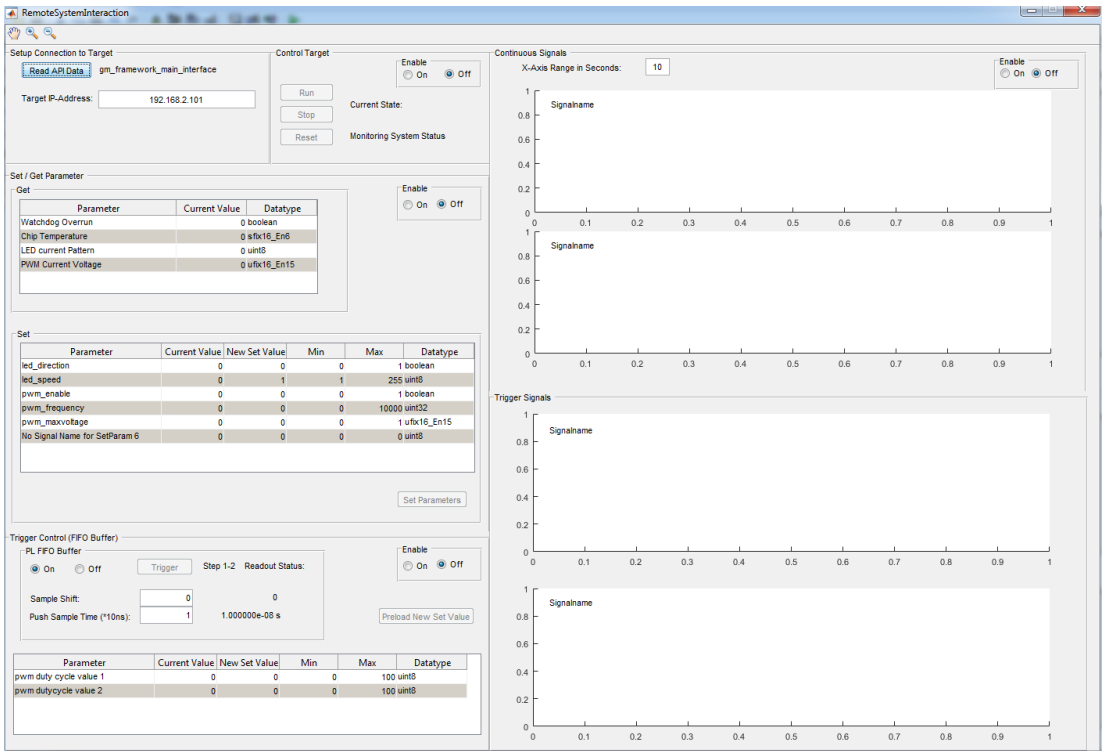

FIGURE 4.9: GUI with loaded Modules

The following subsections explain how to interact with the specific modules. All the modules can be enabled by clicking on the *On* radio button inside the module.

Control Target

The current state of the target is shown when the module is enabled. The Monitoring system status is also indicated by a coloured field where red indicates a problem and green normal operation. With the push buttons the user can change the current state. When pushing the *Run* button, the target changes into the Run state if the Monitoring system is not alarming. When the system goes into Failsafe it can be set back by pushing on the *Reset* button. To stop the system the operator has to push the *Stop* button.

Set / Get Parameter

The current get values are displayed when the module is enabled and updated in the time interval specified by the system engineer. To set new set values the user can click in the table field and enter a new value. It is checked to assure that the new set value is within the defined range and of the correct datatype. After pushing the button *Set Parameters*, the new set values are sent to the target. When they reach the target they will be displayed in the row *Current Value*.

Continuous Signals

The continuous signal module displays a signal changing over time. The default x-axis range is set to 10 s. The axis starts rolling after the time has reached the x-axis range. To set the range of the y-axis, the operator can right click on the plot area itself and set it manually. When the module is enabled, it creates a subfolder called *StoredData* in the path of the application file. There, all data acquired during runtime is stored as mat-files. The file name contains the date and time of the file creation. After a certain amount of data a new file is created. The stored mat-files can easily be loaded back into Matlab for post processing.

To interact or analyse some of the data more detailed within the GUI, the following functions are implemented:

- Pan:
  The user can move the axes when clicking on it if this function is activated.
- Zoom In:
  The user can zoom into the acquired data to get a better detailed view.
- Zoom Out:
  The user can zoom out of the acquired data to get a better overview.

The symbols of these three functions are depicted in Figure 4.10

FIGURE 4.10: Additional Plot Control

Trigger Control / Trigger Signals

With the trigger module the user can record step responses. When enabling this module, a subfolder called *TriggerData* is created in the path of the application file. The following settings can be changed:

- PL FIFO Buffer On/Off
  This radio button enables or disables the FiFo buffer itself. When its turned off the new set values can be used like a normal set value.
- Push Sample Time:
  This is the sample time of the FiFo buffer. The data are acquired with the set push sample time. The smallest possible value is 10 ns, while the largest one is 1 s. The value is set in multiples of 10 ns.
- Sample Shift:
  This is a delay of the switch event. With this feature the user can see what happens with the value right before switching.
- SetValue <1-2>:
  The active set value can be identified by the field right of the *Trigger* push button. It either changes from 1 to 2 or from 2 to 1, where the first value indicates the currently active value.

When all parameters are set, the user has to preload the trigger by pushing on the button *Preload New Set Value*. Once the values are ready, the trigger button is enabled and the user can trigger the preloaded event. When the acquiring of data reached its end, the values are read back automatically into the GUI. Be aware that this process takes at least the following time:

$$\text{ReadBackTime} = \text{FIFOBufferSize} \cdot (\text{PushSampleTime} + \text{PopSampleTime})$$

After the acquired data is read back into the GUI, it is stored as mat-files in the folder *TriggerData*. This way the operator can post process the data afterwards and analyse precisely.

Chapter 5

Further Background Information

This chapter contains some more detailed information which was gathered during writing this Thesis. It should help the reader to get a deeper insight about the general topics Matlab/Simulink, Embedded Systems and Linux as an Embedded Operating System.

5.1 Matlab/Simulink Workflow

The following sections describe the MathWorks workflow to develop embedded system software for sophisticated SoCs. It shows how a system engineer can be supported during the development process of such a system, using the well known tools from MathWorks. It goes even further and shows how to adapt prepared templates and extend the usability of the provided tools.

The main idea behind this workflow is the model based design approach, which will be discussed in the next section. Since the capabilities of the before mentioned tools have grown it became a widespread utilisation due to the many advantages.

5.1.1 Model Based Design

Model based design is a model-centric approach to develop a system. The model includes all system relevant behaviour such as control logic, algorithms, physical components and intellectual property. Figure 5.1 visualises the model based design approach. After research and requirement study, starting by modelling and simulating of the system. With this approach it is possible to rapid prototype the system in a very early state on a development kit. Automatic code generation makes it easy to test and verify the model on a real target. The model based design approach is a highly iterative process which always relies on a working model. [1]

The following list shows some advantages of the model based design approach:

- Facilitates general communication between development groups
- Early error correction (during design process)
- Reuse of design (create reusable block libraries)
- Automated code generation for target deployment
- Acceleration of development process (cost reduction)
- Quick exploration of new ideas (rapid prototyping)

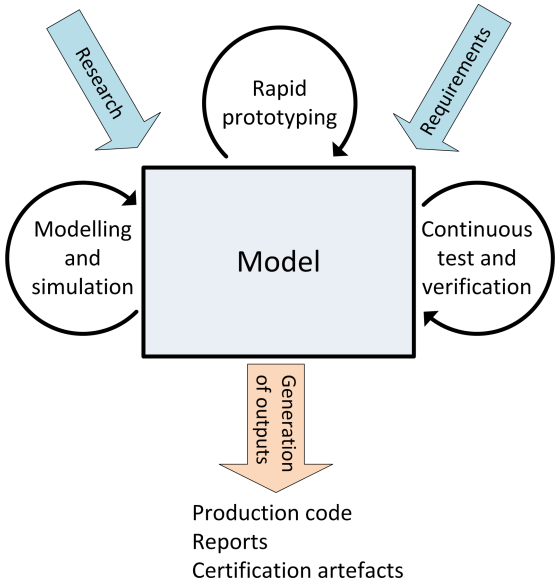

FIGURE 5.1: Model Based Design Workflow

5.1.2 Matlab/Simulink as a Code Development Environment

The following section discusses MathWorks products as code development environment. A lot of system engineers use Matlab products to develop their control algorithms, due to the comprehensible and handy workflow. A lot of useful toolboxes such as the Control System Toolbox, Model Predictive Control Toolbox, Robust Control Toolbox and other support system engineers in their development process. After the control algorithm is developed on a simulation base, the system engineer has to convert it to c-code to implement the control algorithm onto an embedded system and start with testing. To further support this development process, it is beneficial to directly generate code from existing models. MathWorks provides toolboxes to generate c-code and HDL code out of Matlab code or Simulink models. This is a very handy way, since a system engineer can go on working with its familiar tools.

It is important to be aware that there are some limitations using these toolboxes. Not all functionality can be achieved by using the provided standard blocks. Some features need to be programmed by hand.

Synthesis with HDL Workflow Advisor

The HDL Workflow Advisor is a tool provided by MathWorks within the HDL Coder toolbox. It supports the user with HDL code generation and hardware synthesis. There are different checks which the user can run on a certain Simulink model. Also some analysis on the code performance can be done. For further information [31, chapter 22] is a good reference.

Embedded Code Generation

The Embedded Code Generation toolbox from MathWorks has a huge functionality. There is a very powerful support for embedded targets. The Xilinx Zynq-7000 Platform is one of the already supported targets with a predefined code generation toolchain.

This makes it very handy for a user to create embedded code by designing Simulink models. For further information [25, chapter 37] is a good reference.

Prototype Testing Methods

Simulink provides a very handy testing method, the so called External Mode. In it one can generate embedded code from a model, deploy it and run it on the target in realtime. After compiling and deploying the code, Simulink establishes a TCP/IP connection to the target and communicates with it. This allows to get back parameters and signals while the target runs in real time. The fed back data can be displayed on the remote PC using scope and display blocks in the model. It further allows parameter changes on the target for certain Simulink blocks. This testing method can be a real benefit to test an algorithm. For further information consult the Simulink help.

5.1.3 Creation of a Reusable IP Core for Vivado

The following section describes how to create a reusable IP core out of a Simulink model. It is assumed that the functionality of the IP core already exists and has a defined interface. It is important to have an atomic subsystem which contains the whole functionality. The interface to this block (IP core) has to be designed with Simulink In- and/or Outport blocks. To generate the reusable IP core the HDL Workflow Advisor can be applied on the atomic subsystem block (right click on atomic subsystem → HDL Code → HDL Workflow Advisor). The Workflow Advisor will start and can be followed straightforward. If an AXI4 interface to the IP core is needed, IP core generation needs to be chosen as target workflow and ZedBoard as target platform. If no AXI4 interface is needed, generic ASIC/FPGA as target workflow can be selected. In task 1.1 the proper values for the target platform need to be set. These values are filled in automatic if ZedBoard as target platform is chosen. In Task 4.1 the provided tcl-script *package_IP.tcl* (Appendix C folder: 08_tutorials/02_reusableIPCore) has to be added under additional project creation Tcl file. [33] This Tcl script will create and wrap an IP core.

5.1.4 Creation of a Custom Reference Design

To extend the possibility of using MathWorks products for SoC hardware and software development, a custom reference design can be created. There already are some limited Reference Designs available for defined tool chains. These Reference Designs can be used as a template. They are sufficient for a lot of basic projects. If specific hardware is needed or even one would like to register a custom board, there is a possibility to implement them into the MathWorks workflow. The next few sections explain one way how to create and register a custom Reference Design. This enhances the advantage of the handy design workflow a lot.

Figure 5.2 shows the idea behind such a Reference Design. The term *Reference Design* is used for the Vivado hardware design, including the interface from and to the Simulink PL Canvas, except the Simulink PL Canvas itself. If the Reference Design is registered with all its IP cores and the interface between them, Simulink is able to build a working hardware design using Vivado in the background. In Simulink itself, one can build the missing IP core and define the final interface using the predefined interface. Afterwards,

using the HDL Workflow Advisor, Simulink is able to run a synthesis using Vivado in the background and to generate the bit-file.

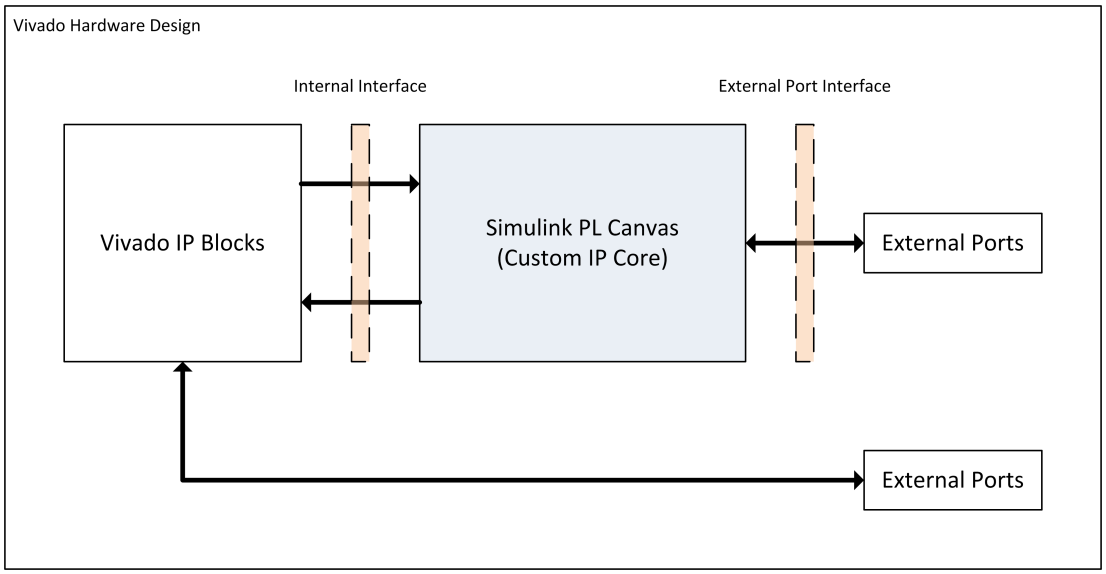

FIGURE 5.2: Vivado Hardware Design with Simulink PL Canvas

A step by step guide to setup a custom Reference Design can be found in section 6.3, while the following subsections outline the general procedure.

Create Vivado Project

To create a custom reference design one can start with a new block design, in Vivado. It is important to have all additional (custom) IP cores available which are used in the Reference Design. Custom or third-party IP cores have to be added in the Vivado IP Repository (IP Settings → IP → Add Repository...) to have them available. In the end all connection are done except the ones which connect the reference design with the Simulink PL Canvas. Those connection have to be left open because they will be defined in the Simulink model.

It can be helpful to write down all interface connection which will be included in the Simulink PL Canvas. Simulink needs to now where to connect the left over connection points and therefore this interface has to be registered in a specific folder structure.

To finish this step it is necessary to export the tcl-script of the just created Reference Design. This tcl-script is the base for all further modelling using this reference design in the background. Vivado provides a function to export it easily (File → Export → Export Block Design). The tcl-script has to be saved as system_top.tcl.

Register created custom Hardware Reference Design

The custom reference design has to follow a certain folder structure. This folder structure and its content is explained in this subsection. Figure 5.3 shows all necessary folders and files which need to be provided to ensure the compatibility with the MathWorks workflow.

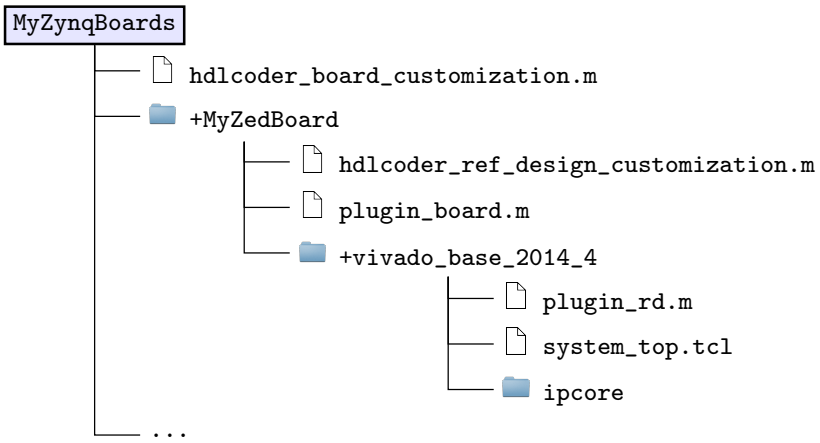

FIGURE 5.3: Claimed Folder Structure for Custom Reference Design

For a proper registration, the following files and folders need to be adapted:

MyZynqBoards
The main folder *MyZynqBoards* and its content need to be in the Matlab search path to ensure that the custom reference design will be available in the HDL Workflow Advisor. This folder can contain several board plug-ins.

hdlcoder_board_customization.m
The file *hdlcoder_board_customization.m* contains the path to all board plug-ins *plugin_board*. Matlab picks up any registration files like this which are in the Matlab search path.

+MyZedBoard This is the board folder. It is important, that the board folder (in this case *+MyZedBoard*) is a package folder. Package folder always begin with the + character. To find out more about package folders in Matlab, one can type *doc package folder* in the Matlab command line.

hdlcoder_ref_design_customization.m
This file points to the Reference Design plug-ins and also returns its associated board name. It is important that the Reference Design plug-ins are located in a package folder.

plugin_board.m
The file *plugin_board.m* contains the construction of a MathWorks board object. A board object contains all relevant information such as board name, FPGA device information, tool information, JTAG chain position and the interface to the external ports of the Simulink PL canvas.

+vivado_base_2014_4
This folder contains the Vivado hardware design (*system_top.tcl*) and all necessary files to create it, such as custom IP cores and constraint files. Constraint files contain pin properties (pin mapping) and IO standards.

plugin_rd.m
The file *plugin_rd.m* contains the construction of a MathWorks Reference Design Object. A Reference Design Object contains all relevant information such as on which Reference Design it is based on (system_top.tcl), custom constraint files, clock interfaces, AXI4-Lite slave interface and all internal IO-interfaces.

system_top.tcl
This tcl script describes the Vivado Reference Design created in section 5.1.4 and has to be placed in the mentioned folder.

ipcore
The folder *ipcore* contains all custom IP cores employed in the Reference Design as zip file container.

5.1.5 Writing a Custom Device Driver using Matlab/Simulink

For certain programming tasks there are no Simulink blocks available. As an example, if someone needs to write to a Linux console for code debugging there are so called Matlab System Blocks. With those blocks one can reference plain c-code. They are also capable of defining what should be done during simulation. In addition there are other ways of writing device drivers such as s-function, legacy code tool, etc. which can be used in a Simulink model. There is a step by step guide available in [8] which explains the different approaches.

5.2 Embedded System Software

Embedded systems usually serve a distinct function. Due to that, they are tailored to suit their purpose both on a hardware and software level. To accomplish a simple task, there is no need to use a fully equipped and powerful system. A simple micro controller with sufficient capabilities is often cheaper, less power hungry, less complex, more durable and easier to maintain. The same can be noted for the software part as well. With every additional function and feature which is implemented, the probability of unwanted interactions and even security risks rises.

5.2.1 Embedded Operating Systems

The Benefit of using an Operating System

However, some microcontrollers specifically targeted to embedded systems such as the *Xilinx Zynq-7000 SoC* are extremely powerful and offer a high variety of special features (for further information also see section 1.5). These advanced capabilities allow for better performance and a wider field of application for a given controller. However, to be able to make use of such advanced capabilities, a profound knowledge of all technical aspects is needed. A high level of complexity arises if the potential performance should be exploited.

As an example, the *Xilinx Zynq-7000 SoC* with its two *ARM Cortex-A9* processors offers a great variety of storage possibilities. Each core has a seperate L1 cache, furthermore there is a combined L2 cache, on chip memory and DDR memory. The correct usage of caches and command queues is critical for a decent performance. An advanced cache and queue strategy is necessary to make use of the potential capabilities of the chip. See also [58] for an overview over the vast amount of features the *Xilinx Zynq-7000 SoC* offers.

Furthermore, the SOC offers a great variety of dedicated peripheral modules like I^2C, *SPI*, *UART*, Ethernet and many more. These need to be initalised and interfaced correctly. Depending on their complexity, this implies a more demanding implementation. For example, to be able to use a network connection, besides the pure hardware layer one must also implement the needed Ethernet protocols for proper communication. Failure to correctly implement all aspects and layers can affect performance, security and even lead to disturbance in the whole network.

To a certain extent, a well configured compiler can help to make use of advanced architecture features. Also, for a lot of peripherals, libraries can be found and used. When working with an operating system, the software developer can make use of the combined, in-depth knowledge of all contributors to the operating system code. Many advanced features and support for peripherals are already built-in and ready to ease the development process. The necessary software to deal and exploit the complex optimizations is written by experts. Due to a wider audience, the code is maintained and bugs are fixed. The level of complexity is cut drastically and the level of abstraction can be raised for an easier development. Also, most operating systems offer multitasking and priority management for processes, which eases the development of complex systems. The same can be said for the use of standard APIs and common functions. Examples include the use of a file system to store data or Ethernet capabilities, which can be used

straightforward instead. The need for implementing custom solutions is lowered, which leads to shorter development times.

An operating system enables the parallel execution of tasks (multitasking). While this seems contradictory to real-time demands, almost every system needs multi threading abilities. For example, the exchange of data with other systems must be possible while a control application is running. An operating system can incorporate sophisticated multi tasking schemes, which allow parallel execution of applications while minimizing the impact on performance. If a system is equipped with several processors, the operating system can allocate resources based on demand (see also subsection 5.2.2). [13, part A and B]

Drawbacks of Operating Systems

While using an operating system offers an easier environment, some flexibility and control is lost. Basic system functions are handled by the operating system, so the users influence on them is limited. Also, restrictions in accessing peripherals can be an issue if an operating system doesn't allow certain operations. If the interfaces to a system core aren't publicly available, writing code or drivers for hardware not supported out of the box gets tricky. With this in mind, also resources from an active development community can be a great advantage.

Operating systems usually are designed to serve several purposes, which leads to potentially exuberant functionality and unnecessary overhead. Not all system software packages can be tailored to the same extent. Unnecessary or badly timed system functions can have impacts on the performance and latency of user processes. These factors need to be considered carefully when deciding whether to use a bare-metal system or selecting an appropriate operating system.

Complex software packages like operating systems can prove to be hard to get to know. While a dedicated standalone application might still be overseeable, a complete system software with multitasking capabilities and other advanced features can become confusing. Changing, adding or even using parts can be tricky, especially if a decent documentation and/or active and open community is missing. [18]

Comparison of Embedded Operating Systems

There are several operating systems specifically targeted to embedded systems available.

- **FreeRTOS**
  FreeRTOS is a lightweight operating system which targets embedded systems specifically. It aims to deliver real time performance while offering comprehensive standard operating system functionality. FreeRTOS even complies to a high level of safety certification (IEC61508). [14] It comes with both a free (open source) and commercial license options. [2]

- **QNX**
  QNX had a turbulent history and is currently owned by Blackberry. The source code of QNX was released and it is free to use for non-commercial applications. It is widely used in critical applications such as the medical, safety and the automotive industry. The system offers real time support and aims for embedded systems.

- **Linux**
  Linux is a highly versatile operating system which targets all sorts of systems and platforms. It is open source, released under the GPL and can be adapted well for every purpose. For embedded systems, the modular nature of Linux systems offers a high flexibility and tailored performance. Real time performance can be achieved with special kernel patching and configuring during the build process.

- **Windows Embedded**
  Windows Embedded is a series of operating systems by Microsoft, targeting embedded platforms. At the time of writing, the current version of Windows Embedded (Windows 10 IoT) was still under development. The current Windows Embedded 8.1 Industry poses high hardware requirements compared to other embedded systems and uses a full graphical desktop [47]. Windows Embedded Automotive 7 is mainly targeted for multimedia systems [37]. Older editions of still base on Windows CE and Windows XP. Only commercial licensing is possible.

5.2.2 Performance and Feature Considerations

Embedded control systems normally implement critical algorithms. These must meet certain requirements in respect of rate and reliability. These realtime tasks must be treated with absolute priority and care. On the other hand, modern systems also consist of less critical interfacing and other support functions. For example, the exchange of parameters and other data with superordinated systems or measurement systems are a fairly common task. The Xilinx Zynq-7000 SoC offers two ARM cores (PS), as well as an FPGA area (PL). This offers unique possibilities to the system engineer related to the overall performance.

Performance for critical Control Processes

In a programmable logic, all tasks can be executed in parallel and will run deterministic. No overhead of other tasks will interrupt execution in the PL. Also, the Xilinx Zynq-7000 SoC offers DSP and flexible storage capabilities, which allow almost all kind of functionality to be realized. Given the vast capabilities of today's FPGAs, as many of the critical functions should be implemented into the PL where they do not need to share resources and are executed deterministically in realtime.

While it is possible to build a deterministic and hard-realtime capable bare-metal system, this is a lot trickier when using a larger system software. However, there are ways to extend realtime capabilities of operating systems like Linux. Using the *preempt_rt* patches is the most common way to go, but there are lots of others. [53] gives an insight into the possible approaches to achieve better performance.

Soft-Realtime and non-critical Processes

In almost every control system, also some non-critical tasks need to be executed. Especially when dealing with complex peripherals or protocols, it is often simpler to implement the needed functionality in classical software. As an example, the processing, storage and visualisation of measurements doesn't need to be dealt with in realtime. Such tasks can normally be executed sequential and therefore be deployed to the processing system without significant drawbacks. Also, they can profit from vast memory

resources and processing power. Multitasking allows the execution of several tasks in parallel.

Naturally some critical functions will reside in the PS. Examples include control loop parts or critical interaction with external hardware and other systems. Due to the sequential nature of processing systems, a good priority scheme and the intelligent use of the available resources are essential for a good performance. The Xilinx Zynq-7000 SoC is equipped with two ARM Cortex-A9 processors. This enables the PS part to execute two tasks in parallel, which enhances realtime capabilities to a certain extent.

The main limiting factors for PS performance is the overall load of the system. Whenever there is a lot of overhead due to basic system tasks or other programs running on the same platform, the scheduling of critical events gets tricky. Normally, a decent operating system will be able to appoint resources in an appropriate manner with its queue scheduling algorithm. System engineers can also try to raise the priority of vital functions manually. But if the overall system load is to high, not all tasks can be executed any more and the performance will degrade. Also, when messing with priorities to much, one can do more harm than good. Only really critical tasks should be given elevated privileges and even then, excessive context switching can decrease overall performance. Also, comprehensive tasks like webservers can compromise the system's overall performance significantly.

Also see subsection 5.3.5 for more information on process priority. For further reading, [39] provides some hints on how to deal with realtime demands in embedded systems.

Symmetric Multi Processing

Due to the two ARM cores present in the Xilinx Zynq-7000 SoC, normal operating systems can spread and divert the workload. A scheme where several cores are used by the same system software is called Symmetric Multi Processing, since it deals with the hardware in an equivalent manner.

Some operating systems allow the user to influence the allocation of tasks to resources. This is done by manually changing the affinity of a process, which binds it explicitly to a CPU. If this can also be done for basic functions like interrupts, a manual division of system and user functions between cores can be achieved, while all interfacing between them is still dealt with by a common system software. See subsection 5.3.5 for more details on how this can be achieved when using Linux.

Asymmetric Multi Processing

Different cores in a system can also execute varying software. While Symmetrical Multi Processing allows to make use of the combined capabilities of a system in a simpler manner, the asymmetric scenario allows for a more distinct partitioning of tasks and processing power.

Several AMP scenarios are feasible, a good overview with further information is provided in [51]. Some of the most common and interesting combinations are listed as follows:

- **Bare-metal with bare-metal**
  The combination of two bare-metal systems allows for a maximal customisation. The division of resources and jobs is completely open and can be tailored to specific needs.

- **Linux with bare-metal**
  When combining a Linux with a bare-metal system, normally Linux is responsible for communication and the handling of other resources. It alows the usage of the available tools and functions of an operating system, while the bare-metal part takes over the critical tasks, which can be executed in a deterministic environment.
- **Linux with FreeRTOS**
  In this scenario, FreeRTOS takes over the critical tasks. Compared to a bare-metal solution, it provides some comfort and basic functionality while keeping the system lean.
- **Linux with Xenomai (Dual Kernel Setup)**
  Xenomai is a realtime development framework, which provides hard real-time support for Linux systems. Different approaches with a single or dual-kernel scheme exist.

The combination of any of the above schemes with one or more additional soft cores in the PL (MicroBlaze) can be of interest. These soft-cores can be used to deal with tasks in a deterministic environment or even customized to deliver higher performance on special tasks than the normal processing system is capable of.

To connect two systems, most often a part of the total memory is accessed by both systems. (Of course, each system also needs some reserved memory space which is hidden from the other one.) This memory can be used to exchange flags, semaphores and data. One system loads its data into the designated area and signals that some data is ready. Upon this notification, the second system can read the data and mark the used area as free again. This exchange scheme needs to be defined and implemented with great care, otherwise functionality can suffer and even significant security risks can arise.

It should be noted that AMP makes less sense the more cores a system offers. This is due to the fact, that modern operating systems have multitasking and paralleling mechanisms which usually deliver a better performance as a manual division, especially when there are enough resources to distribute the load evenly.

For further reference on AMP systems, consider [15] which provides a deep insight into the challenges and benefits of AMP schemes. [11] offers some performance comparisons of Linux and Xenomai setups for realtime applications.

Traditionally, every time a AMP scheme is implemented, the interface between the involved software systems are defined anew. This makes collaboration between teams or code-reuse costly and impractical. To deal with this, there are some efforts to define a standardized AMP interaction scheme called Open Asymmetric Multi Processing (OpenAMP). It specifies the interface methods to exchange signals and data between participating software systems based on the Multicore Communications API (MCAPI) and offers inter processor communication and essential libraries. Xilinx is currently working on an OpenAMP implementation, but the project is still in an early stage (late 2015/early 2016). It mainly targets ARM architectures and implementations with Linux as one of the involved systems, but other configurations are also possible. It will most definitely target the Zynq family as well and therefore offer a good base for further work in this area.

There are other implementations of AMP systems as well, for example the RSoC AMP framework. This particular solution offers support for Linux, bare-metal systems and FreeRTOS and targets the Zynq as well. But due to its proprietary license, it was not looked into further.

If AMP is to be used, using the OpenAMP approach might be the most promising way to go. Due to the high complexity ongoing work of similar projects, the introduction of an AMP system is not pursued in this project any further.

Sources: [9, 11, 15, 40, 38, 41, 52, 57, 44]

5.3 Linux as an Embedded Operating System

Linux is already widely used on the Zedboard and supported by Matlab/Simulink as the standard target operating system. Community support is vast, a lot of topics are already documented. Libraries and device drivers are abundant. Linux was ported to almost any platform and hardware imaginable. Also, it is lightweight enough to run on the Zynq with minimal base load. Due to those factors, it was decided to use Linux as a system software base and not to look into alternatives further.

Also, it was decided to use a SMP strategy to reduce complexity. Depending on the requirements, the Zynq in combination with Linux should be capable of running most systems. Only if specific requirements of a project call for additional performance, the benefits justify the effort of AMP. However, further investigation into AMP schemes is highly encouraged if the situation calls for it.

5.3.1 Linux Variants for the ZedBoard

For the Zynq and the ZedBoard in particular, several Linux flavours are available. Most projects or demonstrations use a particular version and not all serve the same purpose or offer the same functionality out of the box. A short list and comparison should outline the key differences and their impacts of the most obvious variants.

- **MathWorks Linux**
  MathWorks uses a small Linux distribution, which is equipped with just enough functionality to be used as a system software for the Matlab/Simulink workflow. Basically it consists of the absolute minimum of tools for some basic interaction with Linux, network capabilities and file system access to the SD card. An update method is not provided and the image is deployed directly to the SD card when setting up the toolboxes in Matlab. It is published under the GPL license.

- **Xilinx PetaLinux**
  PetaLinux is Xilinx's own prepared Linux distribution for their controllers. A full SDK is provided for configuration and application development and the usage is highly interlinked with the Vivado workflow. Board support packages are available and Licensing for commercial purposes is possible. The Setup process is described in detail in [13].

- **Xilinux (Ubuntu)**
  Xilinux is basically an Ubuntu desktop installation ported to ARM and the Zynq specifically. Due to the performance of the Zynq, running the full desktop environment is no problem at all. It features all Ubuntu functionality like *apt-get*

for updating and installing new applications. However, this can compromise and even completely break the installation due to missing ports.

- **Avnet Linaro**
  Linaro is a industry collaboration with open source community for embedded development. It publishes a specialised Linux distribution which is mainly released under the EPL license. Used for the Avnet FOC Demo Model [7], it comes with a graphical user interface and even an embedded scope to view system data.

PetaLinux, Xilinux and Linaro offer all a lot of functionality which will not be needed. The latter two offer a graphical desktop per out of the box, which is interesting for displaying system data and operating on the Zynq directly. However, such applications are not intended by now. The Mathworks Linux offers only the bare minimum required, but a far simpler system environment with less overhead. It was decided to customise the latter so a individually adaptable system software for the project can be used.

5.3.2 Custom Linux Lianx

As described in [9], Linux uses a two-stage boot process on the Zynq. Most embedded Linux operating systems boot from a packed image rather than a normal, persistent file system. The complete root file system is unpacked and started by the second stage bootloader. This allows for a verified and safe boot environment, but also ensures a sane file system upon every reboot. However, changes made to the running system are therefore lost upon each reboot. The whole root file system is created anew from the image and only files residing on the SD card (mounted under */mnt/*) are persistent. Changing the system boot image is described in subsection 5.3.3. More information about the boot process of the Zynq can be found in [9].

For this project, a custom Linux version called "Lianx" was created. Basing on the standard Linux distribution from MathWorks it offers some additional tools to ease the system engineer's work and flexibility. It includes the following tools:

- **nano** is a simple bash editor, which especially to beginners is more intuitive to use than the standard editor *vi*.
- **htop** is a process viewer, which offers more functionality than the normal *top*.
- **mc** (called Midnight Commander) is a powerful bash file system browser.
- **Startup and Shudown Scripts** are executed automatically upon start and stop. The scripts themselves reside on the SD card and therefore can easily be changed persistently. As an example, this can be used for starting and stopping some additional services.
- **Programming of the FPGA at startup** is done by the startup script if a FPGA bitstream with the filename *system.bit* is present in the *Model* folder of the SD card.
- **Execution of User Program at startup** is possible if the executable is present as *system.elf* in the *Model* folder of the SD card.

The standard login is *root* with the password being *root* also. The network interface is set to the IP address 192.168.2.101 per default. If another IP address or hostname is desired, this can be changed in the files *interfaces* and *hostname* respectively.

Be aware that the default user on the target system has superuser rights. Also, no measures for hardening the system against security risks like setting up a firewall or even a halfway decent password are taken. Be sure to understand the implied risks and security flaws. The provided system may never be used in a risky or commercial aspect unless appropriate modifications are undertaken.

5.3.3 Customizing the Linux System Image

Since the MathWorks Linux uses a packed system image to boot, changes to the environment are lost upon reboot. To be able to add custom packages, scripts and other changes, these must be packed into the image as well. This is done by generating a new, custom image with a framework called buildroot.

Prerequisites for Building a Custom Linux Distribution

MathWorks provides the necessary scripts to build its tailored Linux distribution. The sources can be found in the publicly accessible GitHub repository at [32]. Note that the buildroot framework is intended to be used on a Linux system. At the time of writing, Ubuntu 14.04 is recommended. Newer versions can suffer from a conflict arising in different Python generations, which will the framework cause to fail. Also note, that a native installation of Linux on a machine with enough memory, processing power and a decent SSD is highly recommended [22]. Please note that this section presumes a basic knowledge of working on a Linux system like installing packages, using the bash and file handling/editing. As a hint, to successfully use the buildroot framework on a fresh Ubuntu 14.04 system, also install *gcc*, *g++*, *make* and *libssl-dev*. On a x86-64 system, add the repositories for i386 software and install *libc6:i386* as well.

Since the system to be generated will run on the Zynq's ARM achitecture processor, the Linux packages need to be cross-compiled for said architecture. The buildroot framework uses the Xilinx Software Development Kit (SKD) in version 2015.2 for this process. It can be downloaded and installed for free with the WebPACK license at [55].

Basic Usage and Process of the buildroot Framework

To use the framework, the GitHub repository is cloned to a local folder. The main script to look for is called *./build.py*. Calling it with

```
~> ./build.py -p zynq -b zed
```

causes the build to be customized for the Zynq device family (-p flag) and the ZedBoard (-b flag) specifically. By using subsequent scripts, the executables and file system are now generated. The necessary source files are downloaded automatically and are stored for future build processes.

Passing the additional -u flag, the buildroot scripts try to update the changed parts of the system only, which can speed up the process significantly.

The generated file system and images can be found in *./output/zed_linux_xilinx/images/*. The subfolder *sdcard* contains the file and folders structure which can be copied to the root folder of a SD card directly. The file *zed_sdcard_zynq7000ec_<date>.zip* contains

the same within a zip folder. With a supplied Matlab or Shell script, a new system image can be uploaded to a running target directly (see Appendix C).

Note that the versions of packages used by buildroot are subject to change. At the time of writing, the Linux kernel was used in Version 4.0.0, other than the standard MathWorks Linux which used a kernel of the 3.x generation. By using buildroot and a customized package list, a system running on the ZedBoard can be kept up to date and security patches can be integrated.

Adaption and Modification

The system which is configured by buildroot can be changed in several ways. All changes can be gathered in a subfolder *./sandbox* and integrated in the complex buildroot folder structure with a simple copy script.

- **Software Packages**
  Additional software can be included by expanding the list of make options. Custom options are collected in a separate file and included in *./board/mathworks/-common/scripts/br_config.py* in the *gen_target* function. (Hint: Fitting options are found by executing *make menuconfig* and save the changes to a new file.)
- **Custom Startup and Shutdown Scripts**
  Scripts which shall be executed upon startup or shutdown can be placed in the base folder of the SD card for simpler editing. Copy those to *./board/mathworks/zynq/sdcard/*. For them to be called, a script has to be set up in */etc/init.d* on the target system, which is done by placing it in *./board/mathworks/zynq/fs-overlay/etc/init.d/*.
- **System Calls**
  For a command to register as a system call, its script or a calling function can be copied to *./board/mathworks/zynq/fs-overlay/usr/bin/* .
- **Network Settings**
  The settings used for the network connection are defined in *./board/mathworks/common/fs-overlay/etc/network/* .

5.3.4 Linux Device Tree and Drivers

This section covers a basic introduction to how Linux can be configured for external peripheral hardware.

Device Tree

Peripheral hardware connected to the Zynq must be known to the Linux kernel to be usable. Linux uses memory mapped I/O, which means that access to external hardware is done by using a virtual memory address. This address must be reserved and marked so that the correct location can be found by the kernel. Linux uses a Device Tree to achieve this. The Device Tree files used in Lianx are already suitable for the hardware of the ZedBoard, so for using the XADC, PMODs or similar interfaces no modification is necessary.

The different versions for the usage of axilite and axistream can be found in the root folder of the SD card and are automatically loaded upon boot. Device Trees usable for the kernel are compiled and can be decoded into a readable format with the Linux tool *tdtdump*. As an example, the Device Tree entry for the XADC looks as follows:

```
xadc@f8007100 {
        compatible = "xlnx,zynq-xadc-1.00.a", "xlnx,ps7-xadc-1.00.a";
        reg = <0xf8007100 0x000000a1>;
        interrupts = <0x00000000 0x00000004 0x00000008>;
        interrupt-parent = <0x00000001>;
        clocks = <0x00000003 0x7073372d>;
};
```

If a system engineer wants to expand or change the hardware connected to the Zynq, the Device Tree must be adapted. For more information about memory mapped I/O, see [46]. The Device Tree subsystem is described well in [12].

Device Drivers

When a peripheral device is known to the Linux kernel, a driver is necessary to talk correctly to it. More complex device drivers are best implemented as Linux Kernel Modules. A good introduction into Kernel Modules and an example can be found in [10]. For Simulink device drivers see [8].

For simpler functions, a plain bash script can serve the same purpose as well. The script can be integrated into the root file system for it to serve as a system call as well. An example to drive LD9 on the ZedBoard, which is an LED connected to the PS directly, is integrated in Lianx. It is located in */usr/bin* and named *psled*. It controls the LED by writing the corresponding value into a file dedicated to control the appripriate GPIO. The initialisation is done upon boot when the init-function is called by the custom startup script (see subsection 5.3.3).

For more on Shell scripting, see [48].

Another method for exchanging data with sensors, ADCs and others is the Linux Industrial I/O (IIO) subsystem. This is used by the Avnet FOC Demo Model for example [7]. A good introduction to the IIO subsystem is given in [4]. Since this method is rather specific for sensor data and there was no specific peripheral to be set up, this method was not investigated any further.

5.3.5 Performance Tweaking with Linux

As stated in subsection 5.2.2, there are a lot of ways to enhance the realtime capabilities of Linux. Most of them call for the use of special frameworks, tools or additions to the system itself. Such efforts are only called for when the additional performance is really needed by a specific project. However, there are some ways to increase or at least monitor performance on a normal multi core Linux system with system tools. Be aware that usually the implemented multitasking schemes of Linux do a good job already and manipulating the default behaviour of both processes and interrupts can do more harm than good.

Task Niceness and Priority

As on most multitasking systems, every task has a priority assigned. The priority determines which processes will be granted system resources if they get scarce. Priority values range from 0 (highest) to 139 (lowest), where the range from 0 to 99 is for realtime system processes only and user priorities range from 100 to 139. According to a tasks priority, the resource assignment to it are handled. Of course this only works if priorities differ relatively from each other. When no other tasks can give up some ressources, rising a priority will have no effect at all. Critical system tasks must of course be granted a higher priority than minor functions or the system becomes compromised. Additionally, Linux uses a niceness value to determine the resource hogging behaviour of a user space process. The niceness is used to calculate the priority accordingly. Values range from 20 (very nice) to -19 (not nice at all). A higher niceness and therefore priority marks a tasks capabilities and readiness to give up resources for others sake.

$$\text{Priority} = \text{Niceness} + 120$$

This results in user process priorities from 100 (niceness -19) to 139 (niceness 20) [42]. See *man nice* and [20] for more information. In Appendix B some matching tools and commands are listed as well.

Task Affinity

On systems with more than one processor core available, tasks can be distributed to share the total system load. Normally, tasks are dispatched according to the current system load and the respective task priority. But this allocation, which is called affinity, can be changed manually. This enables a manual division of resources between critical and unimportant tasks even in an SMP system. When allocating all tasks to one core and only the critical one to the other, it is less likely that the vital functionality of a system is interrupted.

See *man taskset* for more information and Appendix B for appropriate tools.

Interrupt Priority and Affinity

Just as with processes, interrupts have a prioritising and affinity scheme. The interrupt priorities are mostly handled by the operating system in combination with the underlying hardware. Therefore, altering those requires profound changes to the system. But instead of changing the interrupt system itself, distributing its load evenly or specifically amongst processor cores can also help to achieve better performance. To get an overview over the load interrupts generate on a system, look into the */proc/interrupts* file. The affinity of individual interrupts can be changed by modifying the file */proc/irq/<irq-number>/smp_affinity*.

The following output shows the executed interrupts for the ethernet connection. The first 395 interrupts are automatically dealt by CPU0, while the next 14 are dealt by CPU1 after manually setting the affinity.

```
root@zynq-lianx:~> cat /proc/interrupts | grep eth0
33:         238          0       GIC  54  eth0
root@zynq-lianx:~> cat /proc/irq/33/smp_affinity
3
root@zynq-lianx:~> echo 2 > /proc/irq/33/smp_affinity
root@zynq-lianx:~> cat /proc/interrupts | grep eth0
33:         395         14       GIC  54  eth0
```

A comprehensive introduction into the Linux interrupt system and interrupt affinity can be found in [16].

Chapter 6

Demonstration and Tutorials

This chapter contains step-by-step procedures to set up the ZedBoard, a Demo Model and a custom Reference Design. Please ensure that the prerequisites listed in section 4.1 are met, otherwise unexpected problems can arise while following the tutorials.

6.1 Setup of the ZedBoard and connecting with a Remote Console

This section should allow any technically skilled person to set up the ZedBoard without any knowledge about the involved hardware and/or tools. The successful execution of all steps are vital for any work on the ZedBoard outlined in this Thesis.

To be able to complete this procedure, the following tools and parts are needed:

- Hardware:
  - Digilent ZedBoard with power supply
  - SD card with at least 64 MB capacity
  - Computer with SD card reader
  - Two micro USB cables and LAN cable
- Software:
  - Lianx SD card image (see Appendix C)
  - Windows only: PuTTY (can be downloaded from `http://www.chiark.greenend.org.uk/~sgtatham/putty/`)

6.1.1 Setup SD Card

First, an SD card must be set up as a boot source for the ZedBoard. Insert the card into the computer and format the SD card with a FAT or FAT32 file system. Now copy all contents of the Lianx SD Card Image zip-file onto the freshly formatted SD card. Safely eject the SD card from the card reader after copying has completed.

6.1.2 Hardware Setup

To ensure a correct boot procedure, some jumpers on the ZedBoard must be set correctly. JP7 to JP11 must be set to SD card as the boot source. JP7, JP8 and JP11

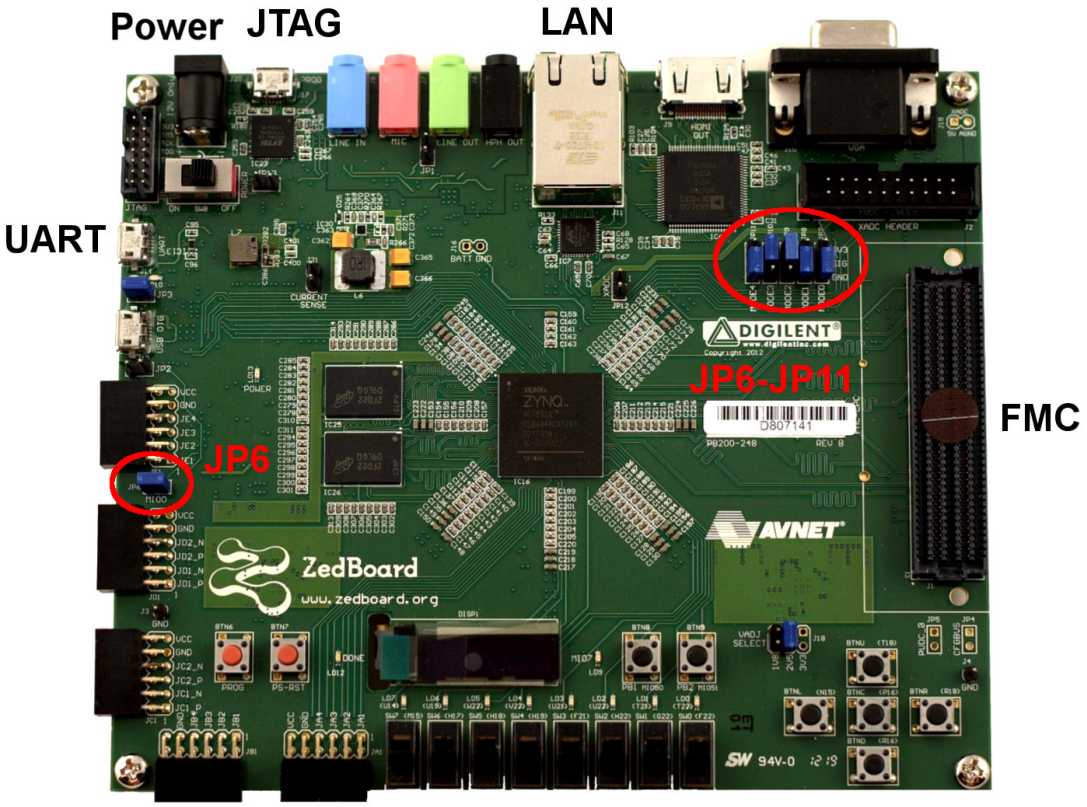

FIGURE 6.1: ZedBoard with Connectors and Jumpers [6, modified]

must be set to GND, while JP9 and JP10 must be set to 3V3. If you use a board in revision C, also make sure to set JP6. Figure 6.1 shows the location of the jumpers and their correct configuration.

Insert the SD card into the slot on the backside of the ZedBoard. Connect the USB cables for the UART and JTAG interfaces and LAN cable, both to the computer and the ZedBoard. Also connect the power supply and switch on the ZedBoard. See Figure 6.1 for further reference of the connections. The ZedBoard will now boot up and start the embedded Linux.

6.1.3 Setup the Connection

The connection method to the target depends on the desired channel. If you wish to connect directly with an USB-cable, you can use the serial console. The connection via Ethernet and SSH is much more versatile and offers remote connection even over LAN or the internet. However, the setup of such a connection has more pitfalls. When using a non-isolated LAN, appropriate security measurements like the usage of a VPN connection must be taken, which aren't discussed in this document.

Connection via a Serial Console

Find out which COM-port the USB-to-UART adapter on the ZedBoard was assigned. Pressing [Windows] + [Pause], select the *Device Manager* in the list left and expand the list of Serial and Parallel Connections (COM and LPT) as depicted in Figure 6.2.

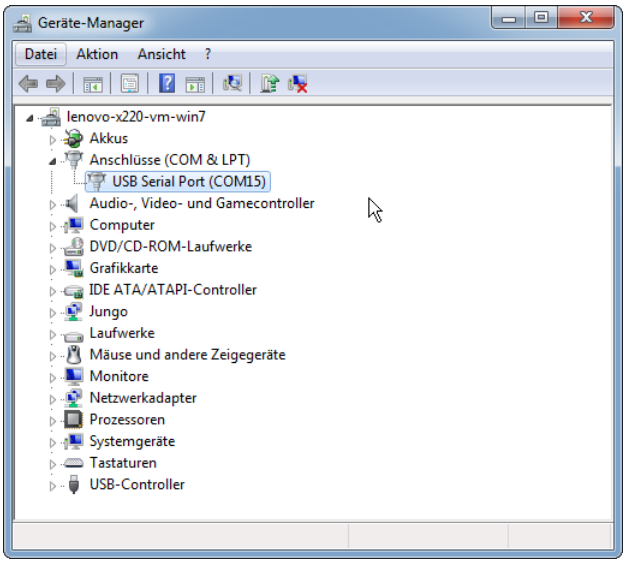

FIGURE 6.2: Windows Device Manager with the COM-port of the ZedBoard USB-to-UART Adapter

Start up PuTTY, choose a serial connection and enter both the correct COM-port number as well as the speed of 115 200 baud as shown in Figure 6.3. Connect to the target and a remote console on the ZedBoard will open. Press enter and the bash prompt will show as in Figure 6.4.

Users of Linux can use the *screen* tool to connect via a serial connection.

```
~> ls /dev/ | grep ttyACM
ttyACM0
~> screen /dev/ttyACM0 115200
```

Users of OSX can also use the *screen* tool to connect via a serial connection.

```
~> ls /dev/ | grep 'tty\.'
tty.usbmodem1451
~> screen /dev/tty.usbmodem1451 115200
```

Kill the connection with [Ctrl] + [A], [K].

Connection via SSH

The default IP adress of the ZedBoard is 192.168.2.101. It can be changed by editing the file 'interfaces' on the Lianx SD card. It is assumed that the user is familiar with setting up the necessary network connection on his computer in an existing LAN or a direct connection using static IPv4 adresses in the same subnet as the ZedBoard resides in.

Start up PuTTY, enter the host name 'root@192.168.2.101' and connect as shown in Figure 6.5. You will be asked for the root password which is *root* and after hitting enter, you will be welcomed with the message in Figure 6.6 on the ZedBoard.

To disconnect, simply close the PuTTY window.

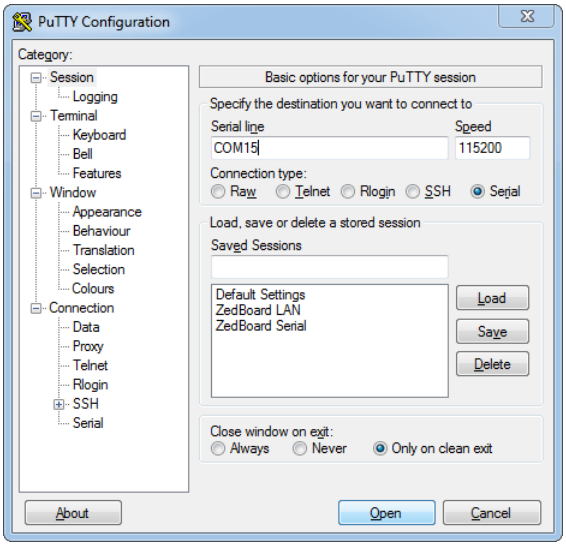

FIGURE 6.3: PuTTY Settings for Serial Connection

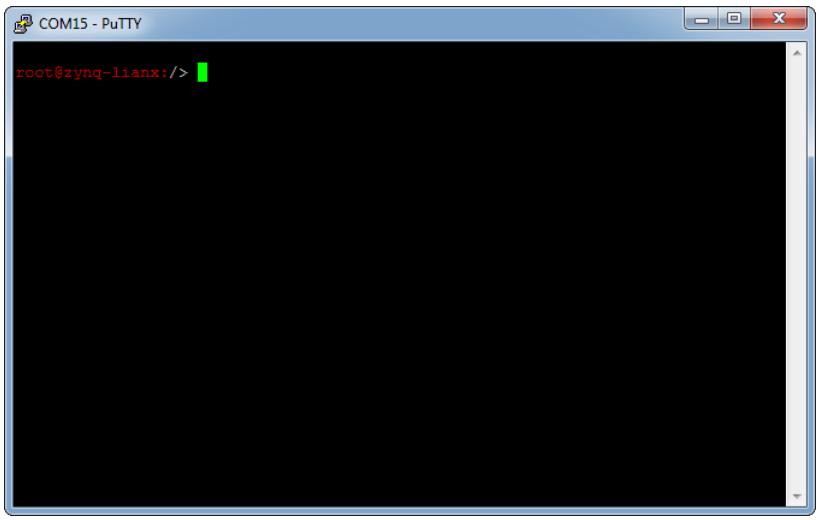

FIGURE 6.4: Remote Bash on the ZedBoard via a Serial Connection

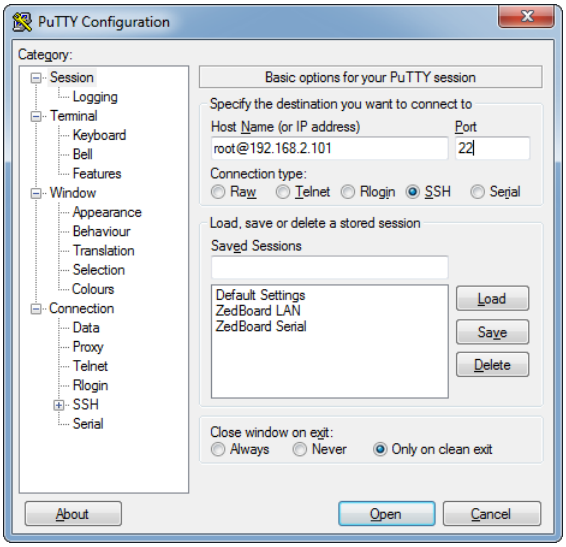

FIGURE 6.5: PuTTY Settings for SSH Connection

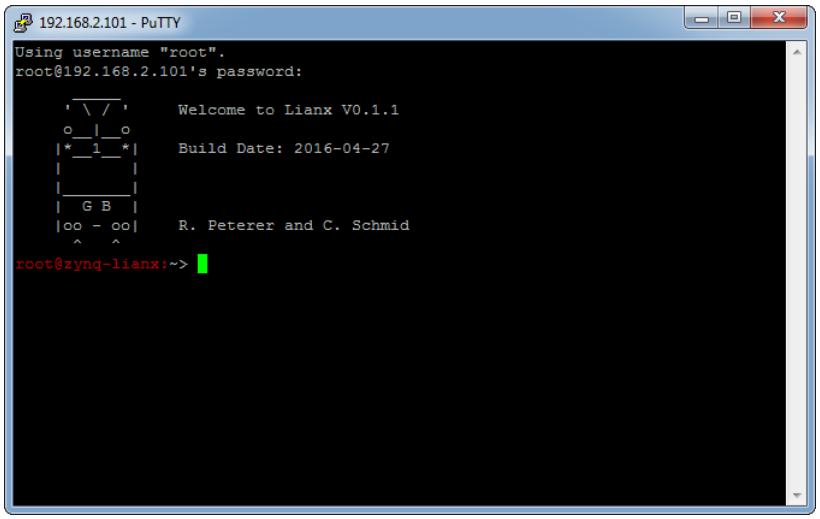

FIGURE 6.6: Remote Bash on the ZedBoard via a Ethernet Connection using SSH

6.1.4 Usage of the Remote Console

When connected to the ZedBoard with a remote console, you can interact with the running Linux directly. Several tools are available, such as the text editor *nano*, the process and system load monitor *top* and many more. The installation of programs like on normal Linux environments is not possible, but can be done by creating a new system image using the buildroot tool (for details see subsection 5.3.3). The contents of the root file system are created anew upon each reboot from the packed ramdisk images. The contents of the SD card however remain persistent and can be found in the */mnt* path.

To show some functionality of the connection, a simple driver for controlling the PS LED (LD9) can be used.

```
root@zynq-lianx:/> psled on
root@zynq-lianx:/> psled off
```

Be aware that you are granted root access to the running Linux. Be sure to understand the implied responsibility and always be cautious when working directly on the target.

Before turning off the ZedBoard, always make sure to shutdown the running operating system by entering:

```
root@zynq-lianx:/> halt
```

6.2 Setting up the Demo Project

In this tutorial, a demonstration project is introduced. First, the general setup is explained, and second a step-by-step guide helps to get it up and running.

6.2.1 Overview over the Demo Project

To show the capabilities of the framework and how to work with it, the Demo Project implements some simple tasks:

- **Simple LED Counter**
  A basic LED counter uses the eight LEDs on the ZedBoard. The speed and direction of the counter can be influenced from the PS.
- **PWM-driven Capacitor**
  A PWM signal on PMODA JA2 charges and discharges a capacitor and the resulting voltage is read back with the internal XADC. The PWM can be enabled, the base frequency set and the duty cycle can be used for step response analysis (FiFo buffer).
- **Monitoring**
  The monitoring compares the maximal capacitor voltage and sets the system into the Failsafe state if it passes a configurable value. Also, SW0 triggers the monitor if switched on.

A file contained in the Demo Project folder in Appendix C called *signal_list.txt* lists the signals used in this project. Figure 6.7 shows the used external hardware connected to the ZedBoard.

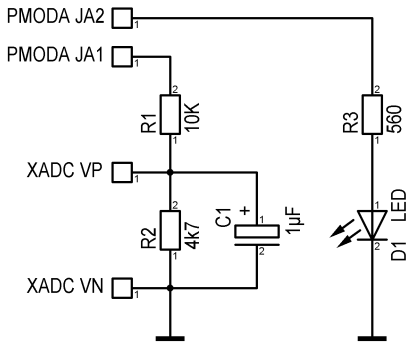

FIGURE 6.7: Schematic of the Demo Project Hardware

6.2.2 Setting up the Development Framework

To set up the Demo Project, make a copy of the supplied folder structure to work in. The source files are provided in Appendix C. Connect the ZedBoard, running the Lianx System Software (to set this up see section 6.1). Now open the Simulink project in the Demo Project folder, which initialises the Matlab environment and the connection to the ZedBoard. Open the User Canvases with the first Simulink Project shortcut which can be found in the upper left corner of the Projects Shortcuts tab in Matlab. In the opened tabs, the implemented functions can be seen. Explore the model to get an idea how and where what can be done. Also, the structures of the framework can be viewed

by navigating through the hierarchy. When done, the model should be closed without saving any changes.

The next step is to generate the PL bit-file. Use the second Simulink Project shortcut to open the HDL Workflow Advisor. Follow the advisor steps until the end. Explore the settings, which all should already be set correctly. The interface model generated in step 4.2 must be saved in the work folder under the proposed name. Also note that before executing the last step 4.4, wait for the external console to indicate the end of the bitfile generation from step 4.3.

By using the third project shortcut, the generated interface model can be opened and the system tested in the External Mode. When the constants providing signals to the PL block are changed, they will be sent to the target and thus can be used to interact with the model running on the Zynq. The systems reaction can be watched by adding scopes to interesting signals. As described in subsection 4.2.9, the Watchdog must be turned off when using the External Mode.

The bitfile is programmed only until the next reboot. To reprogram it, use the Simulink Project shortcut for the temporary deployment of the bitfile (as well as the interface model software).

6.2.3 Implementing the Remote System Interaction

To use the system from remote without the need of a Matlab license, the interface model needs to be adapted. There are five blocks to be added and some parameters to be set for this demo project. The user needs to open the Simulink model by double clicking on the model *gm_framework_main_interface.slx*. The blocks can be taken from the provided library *lianx_lib_ps* which can be found in the Simulink library browser.

To make the Demo Model work, it is important to connect the following blocks correctly. Figure 6.8 illustrates how to connect the block *Lianx Control Target State Control*. The default parameter values set in the block mask can be kept.

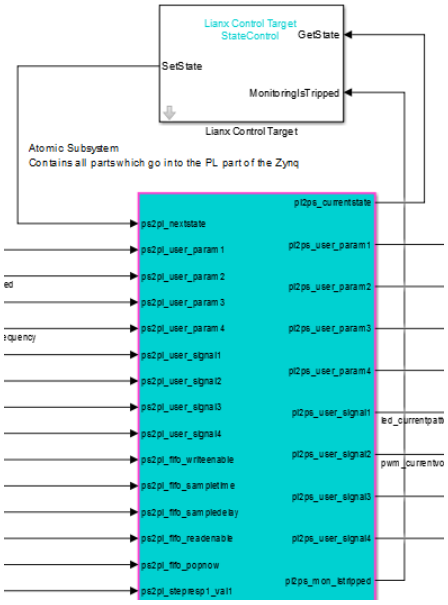

FIGURE 6.8: Connecting the Lianx Control Target State Control Block

The block *Lianx UDP Receive SetParam* is next. The necessary connections are depicted in Figure 6.9. It is important to set a signal name to have it available afterwards in the GUI. To do so, double click on the signal line and enter the corresponding name.

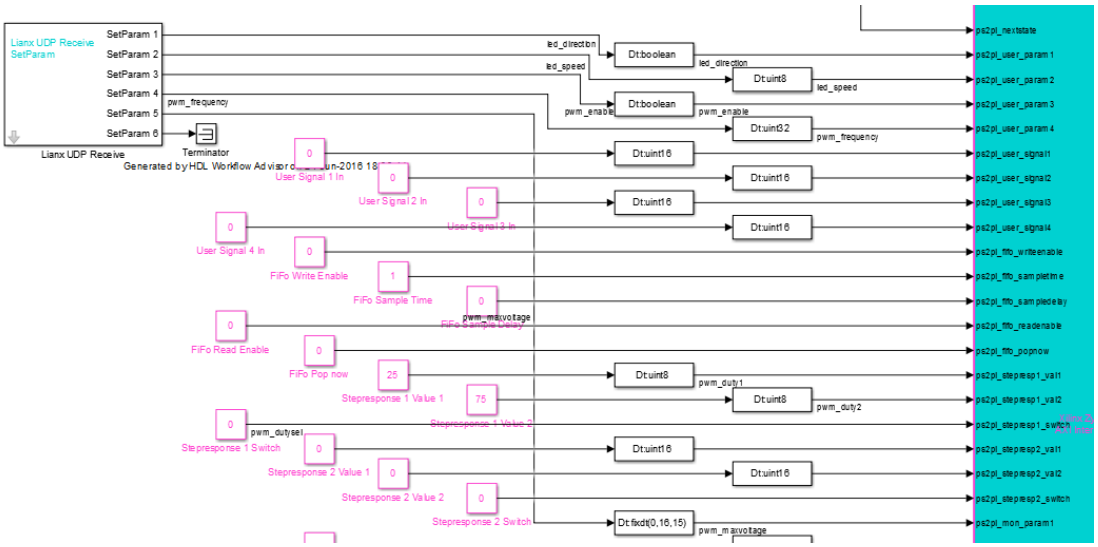

FIGURE 6.9: Connecting the Lianx UDP Receive SetParam Block

There are also some mask parameters to be set. Double click on the block and the mask window will open. In Figure 6.10 all parameters to set for the demo model are shown. All used output ports need to have a min and max value. For the IP port configuration tab the default values can be used.

The blocks *Lianx Continuous Signal* and *Lianx UDP Send GetParam* are depicted in Figure 6.11. Also here it is important to name the signal. The mask parameters can be left with default values.

The last block to connect is the *Lianx PL FIFO Buffer PS Control* block. All connections can be seen in Figure 6.12 and Figure 6.13. To have the signal names available afterwards in the GUI, they have to be set at the signal line to the block itself. Finally the values in the block mask need to be set correct. All values which have to be set are depicted in Figure 6.14. The other parameter can be left with its default values.

When all blocks are connected and parameters are set as demonstrated, the model has to be saved and the user can go ahead and generate code of the model. This can be done by clicking on the shortcut *04 - Generate PS Software and GUI API*. This will generate an executable for the target and an API list with all necessary information for the GUI. These two files are stored in the work folder. The executable has the same name as the model itself but without file extension. The created API list is called *API_Data_gm_framework_main_interface.mat*.

To test the just generated code, the executable and its corresponding bit-file can be deployed temporarily by clicking on the shortcut *05 - Deploy PL and PS code temporarily*. Alternatively the deployment can be made permanently by using the shortcut *06 - Deploy PL and PS code persistently* this ensures the reprogramming of the PL and starting of the PS software upon reboot. To revoke the permanent deployment one can click on the shortcut *07 - Remove persistently deployed PL and PS code*. Since the model was adapted for remote control, the target can now be controlled with the GUI.

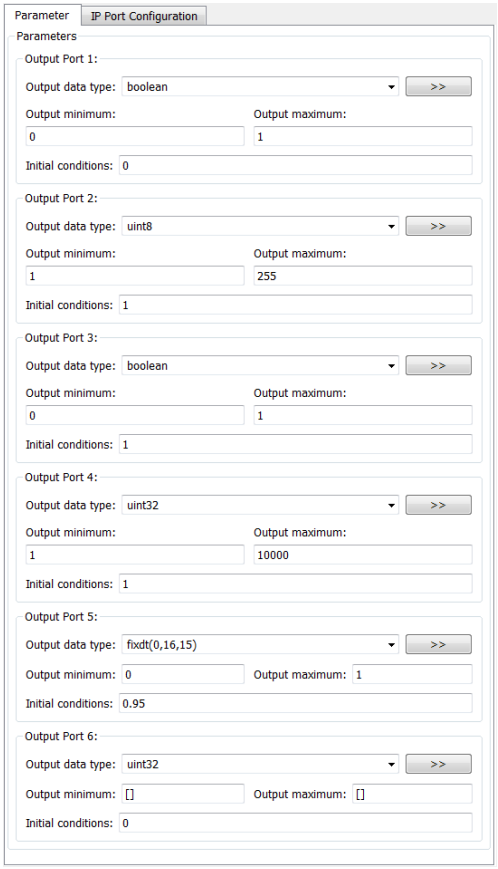

FIGURE 6.10: Mask Parameters of the Lianx UDP Receive SetParam Block

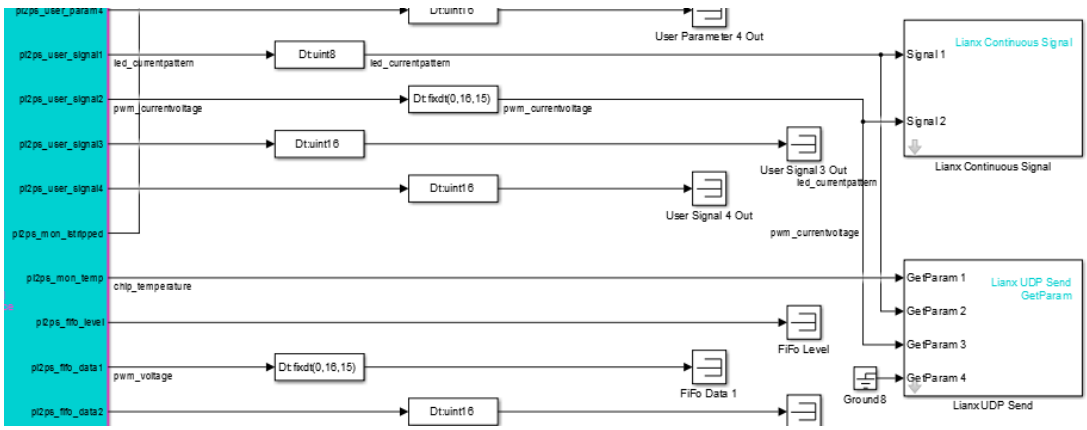

FIGURE 6.11: Connecting the Lianx Continuous Signal and Lianx UDP Send GetParam Block

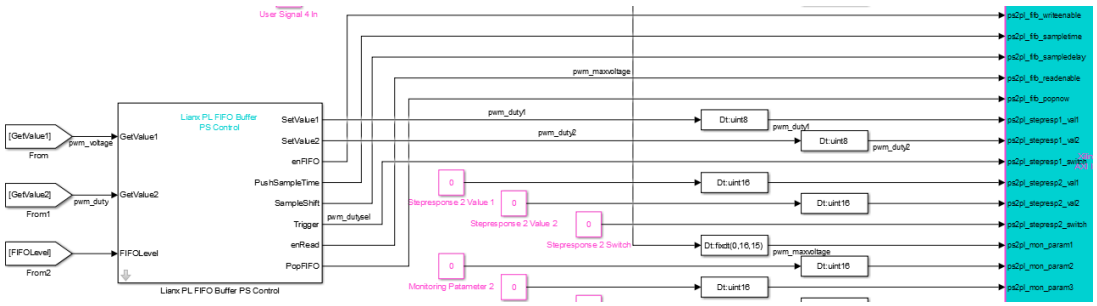

FIGURE 6.12: Connecting the Lianx PL FIFO Buffer PS Control Block Outputs

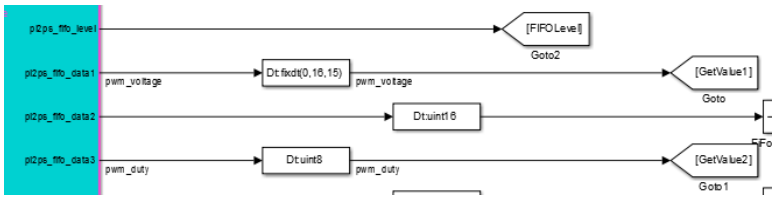

FIGURE 6.13: Connecting the Lianx PL FIFO Buffer PS Control Block Inputs

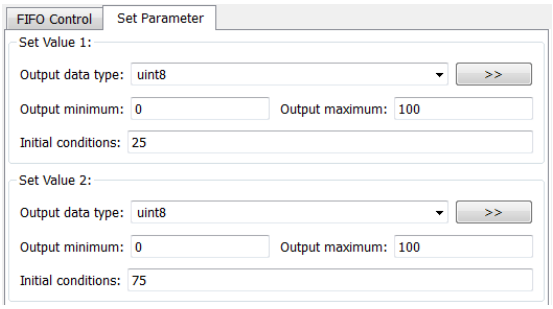

FIGURE 6.14: Mask Parameters of the Lianx PL FIFO Buffer PS Control

6.2.4 Running the Demo Model

When the software is deployed, the GUI can be started by clicking on the shortcut *08 - Start GUI*. Make sure that the following ports are open and not blocked by a firewall, otherwise the corresponding module will not work:

- 25000, 25001, 25002, 25003: for the set / get parameter module
- 26000, 26001: for the control target module
- 27000, 27001: for the continuous signal module
- 28000, 28001, 28002, 28003: for the trigger module

After the GUI has started up the before created API Data has to be loaded by clicking on *Read API Data*. The GUI afterwards will load all necessary modules and look as in Figure 6.15. One module after the other can now be enabled and tried out. Additional information how to use the GUI can be found in subsection 4.3.5.

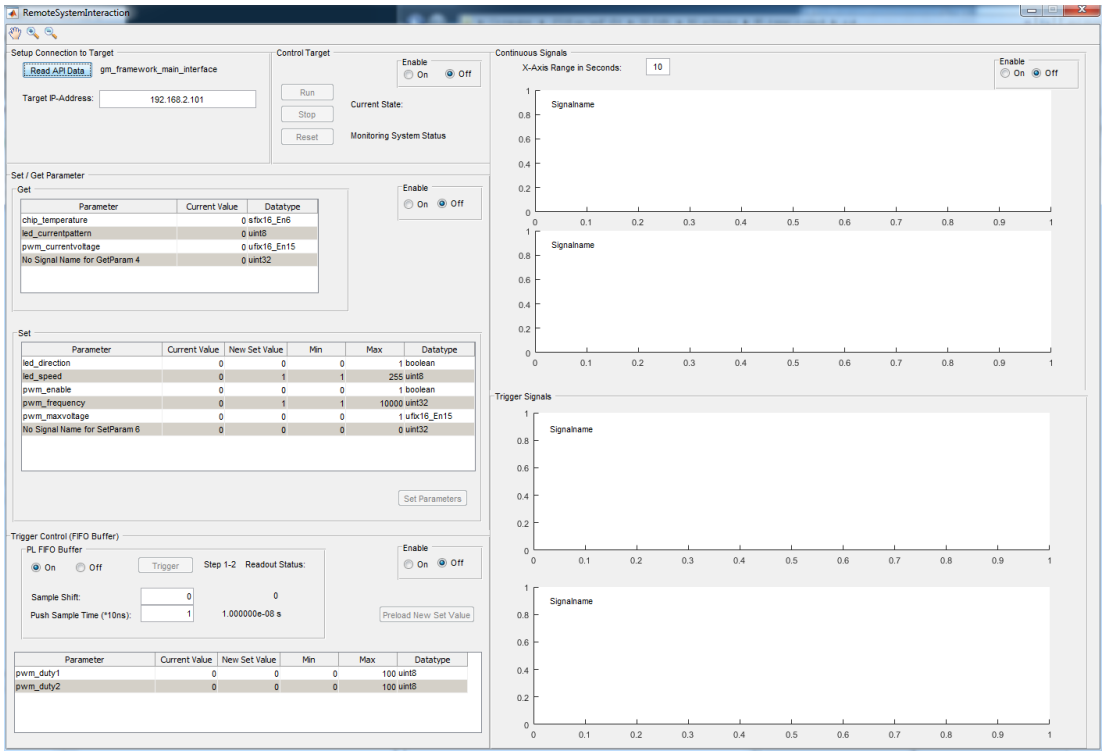

FIGURE 6.15: GUI Start Screen

Here are two features of the Demo to try out:

LED Counter

Enable at least the Control Target, Set / Get Parameter and Continuous Signal module. To start the LED counter, assure that the monitor does not trip. Therefore set the *New Set Values* as follows:

led_direction	0
led_speed	10
pwm_enable	0

To set the new values push *Set Parameters*. Afterwards the *Run* button in the control target panel can be used to start the LED counter. The continuous signal plot should now show the current led pattern in the upper plot area. If the system monitor was tripped before it might be necessary to push the *Reset* button. The y-axis can be scaled by right clicking into the plot area and choosing a new range.

Step response

If the external hardware which was mentioned in subsection 6.2.1 is available, it is possible to record a step response. To do so, some values have to be changed as follows:

pwm_enable	1
pwm_frequency	1000
pwm_maxvoltage	0.95

The values of the LED parameters do not matter, because the LED counter can run parallel. The trigger module has to be enabled and the following parameter can be set within this module:

Sample Shift	250 samples
Push Sample Time (*10 ns)	100 (1×10^{-6} s)
pwm_duty1	0 %
pwm_duty2	75 %

After the values are changed, push the *Preload New Set Value* button. This prepares a trigger event with a sample rate of 1 MHz and switch the parameter after 250 samples. As the *Trigger* button is now pushed, the view depicted in Figure 6.16 will appear. Be aware that the read out process of the FiFo buffer will take at least 10 seconds in this configuration. The user is encouraged to play around using other parameters within this Demo Project.

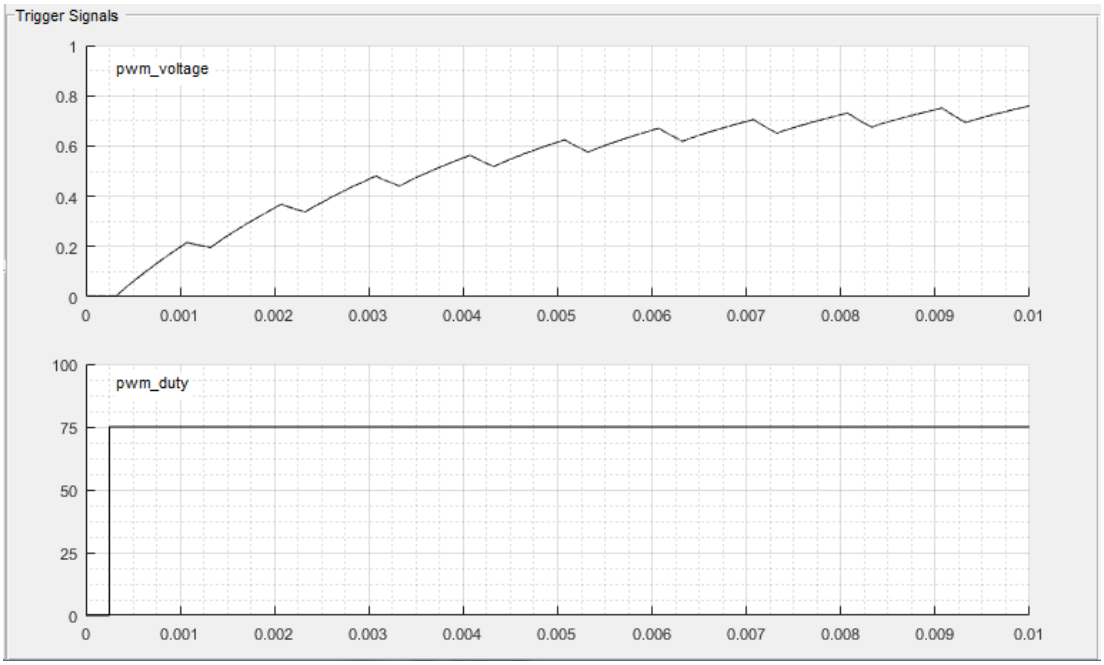

FIGURE 6.16: Step Response of Trigger Shot

6.3 Setup Zynq XADC in Custom Reference Design using Matlab/Simulink

The following tutorial shows how to use the integrated XADC of the Zynq-7000 SoC and demonstrates a basic implementation method of it. It further shows how to create and register a custom Reference Design in a step by step guide. Useful additional references for further development are also given. The given example targets the use of three differential analog inputs which are available on the XADC header of the ZedBoard. The pin out for it can be found in [6, p. 25].

The folder *SetupXADC* in the digital appendix contains all supporting files and scripts.

6.3.1 Create Vivado Reference Project

Step 1: The first step is to create a Reference Design which can be expanded with an XADC IP block. To do so, the first section *1: create Vivado Project* in the script *utility_SetupXADC.m* can be run. Make sure that you change the current Matlab folder to the file/script location. The first section of the provided script creates a Vivado project. It is based on the Reference Design provided by the MathWorks hardware support package *Xilinx Zynq-7000*. After the project is created, it will open and focus the user to the block design diagram. The just created basic reference design is based on a clock frequency of 50 MHz. To change the clock frequency, one can double click on the clock wizard *clk_wiz_0* and change the output clock frequency to 100 MHz under the output clocks tab. The hardware block for the XADC is implemented using the XADC Wizard, provided by Xilinx. To add this block, right click in the block design Diagram and click *Add IP ....* A search window will appear, fill in *xadc wizard* and press enter. The IP block *xadc_wiz_0* is added. There are many different ways to parametrise this IP block, for further information see [56]. Figure 6.17 to Figure 6.20 show the set values for this example. The Dynamic Reconfiguration Port (DRP) communication interface is used to interact with the XADC from the Simulink PL Canvas.

The just parametrised IP core does need some hardware connection. The block port *xadc_wiz_0/dclk_in* is connected to the block port *clk_wiz_0/clk_out1*. The three differential channels (vn/vp, vaux0n/vaux0p and vaux8n/vaux8p) have to be external. To do so, expand the channel buses, select all ports of the channels and press [Ctrl] + [T]. Figure 6.21 illustrates the just augmented basic reference design.

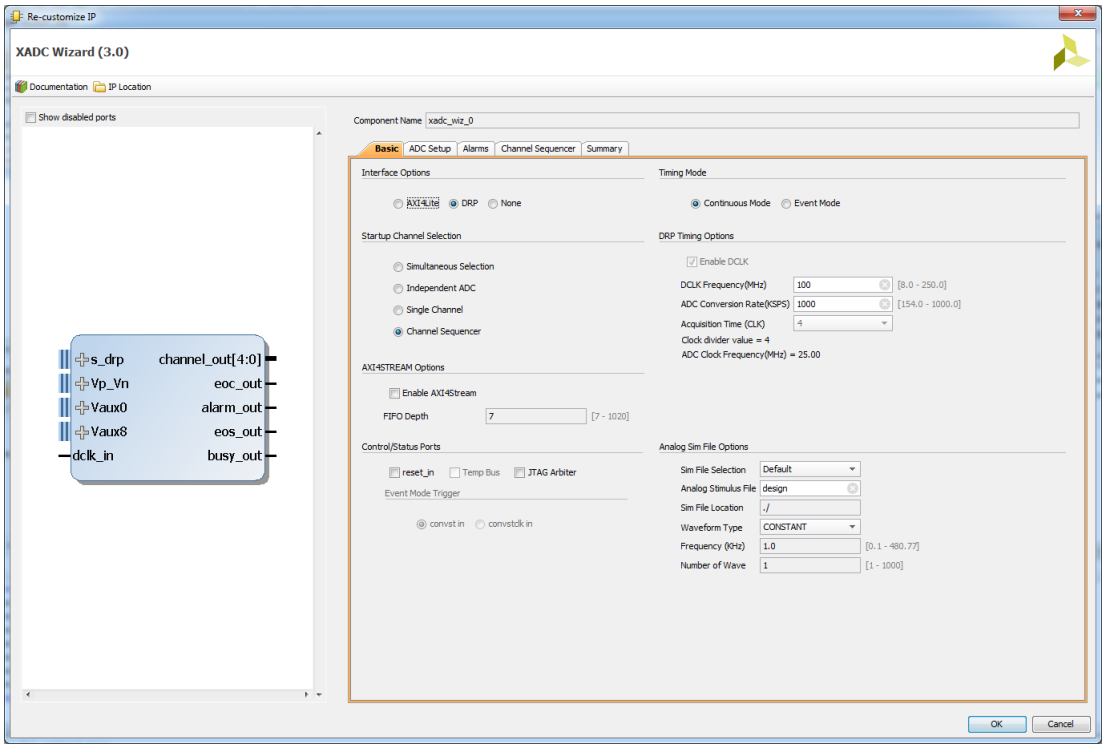

FIGURE 6.17: XADC Wizard Tab settings: Basic

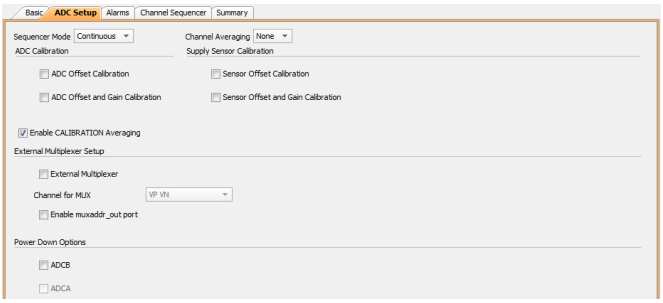

FIGURE 6.18: XADC Wizard Tab settings: ADC Setup

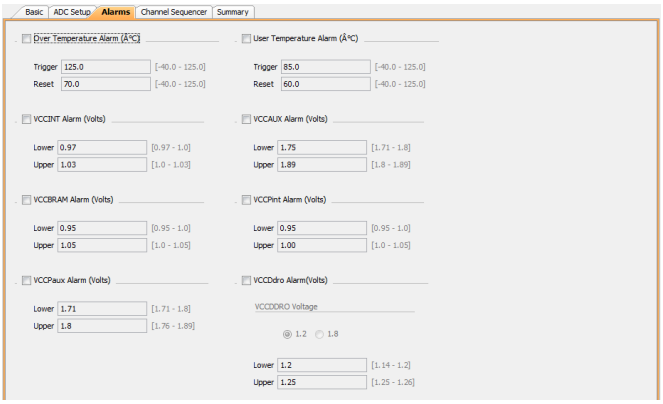

FIGURE 6.19: XADC Wizard Tab settings: Alarms

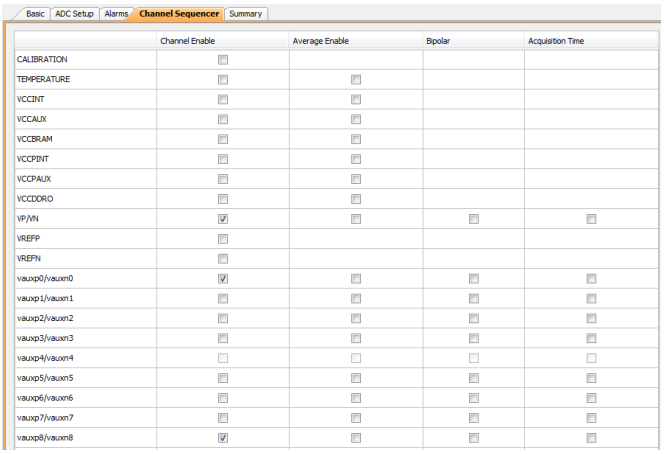

FIGURE 6.20: XADC Wizard Tab settings: Channel Sequencer

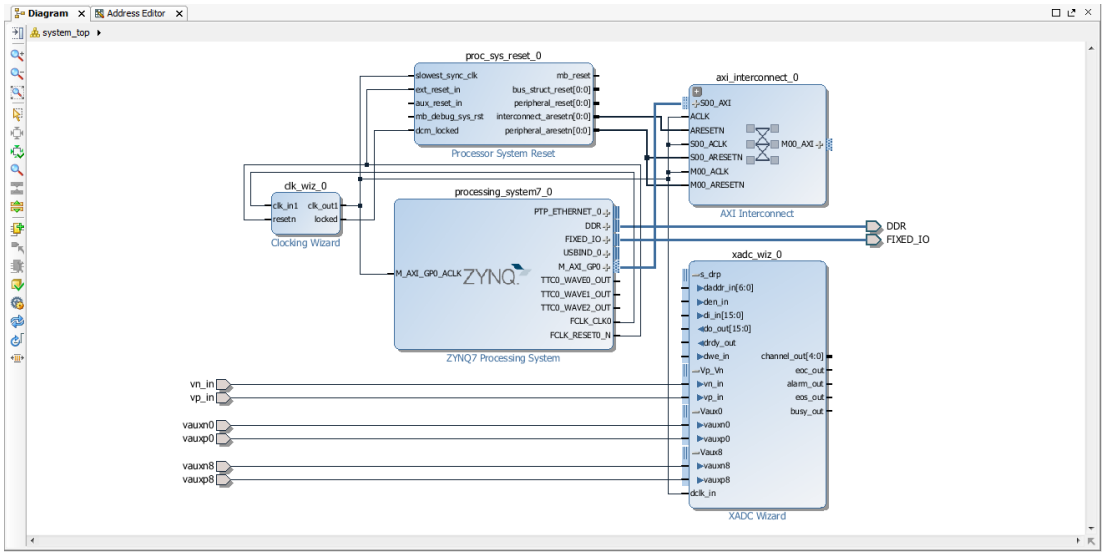

FIGURE 6.21: Reference Design for the XADC Tutorial

Step 2: The next step is to create a tcl-script for this reference design. This can be achieved by writing the following code into the Tcl Console of Vivado which can be found in the lower part of the window.

```
set path [pwd]
write_bd_tcl $path/system_top.tcl
```

This commands will create the file *system_ top.tcl* in the current folder. The just created tcl-script of the reference system will be used in step 5 to execute the registration of the custom Reference Design.

6.3.2 Interface Definition

Step 3: It is worth to write an interface list for the registration of the Reference Design to have a better overview. This interface list should include the *Internal Interface* and *External Port Interface* as mentioned in section 5.1.4. A template for such a list is available in Appendix C as *Interface.xlsx*. A script could be written to automate the registration using this interface list, if a lot of reference designs need to be registered. Also a constraint file needs to be written for the external ports of the Reference Design. These ports need to be assigned to a specific pin on the Zynq SoC.

In addition to the basic XADC functionality we also include two GPIO pins which reside on the XADC header. For a proper external IO interface registration, the following arguments need to be defined:

- InterfaceID e.g. 'LEDs General Purpose'
  This name will appear in the HDL Workflow Advisor in the Target Platform Interface dropdown list.
- InterfaceType e.g. 'OUT'
  Interface direction, if OUT it can only be assigned to a Simulink Outport, if IN it can only be assigned to a Simulink Inport, if INOUT can be assigned to both.
- PortName e.g. 'GPLEDs'
  This is the Board top-level name.
- PortWidth e.g. 8
  This defines the port bit width.
- FPGAPin e.g. {'T22','T21','U22','U21','V22','W22','U19','U14'}
  This defines the pin name of the defined interface. Information about the FPGA pins can be found in Appendix C under *01_ documentation/xc7z020clg484_ pin.ods*.
- IOPadConstraint e.g. 'IOSTANDARD = LVCMOS33'
  This defines the applying IO standard.

In the base reference design the following external interfaces are already available:

- 8 LEDs
- 8 DIP Switches
- 5 Push Buttons
- 4 Pmod Connector with 8 ports each

It is possible to utilise these external ports later in the Simulink PL canvas for direct interaction with the mentioned periphery.

The internal interface between the reference block design and the Simulink PL canvas need to be defined as well. In our configuration the XADC uses a DRP communication interface whose connection need to be implemented. Therefore the following two input and two output ports for the DRP are needed:

- Output data bus
  This ports delivers the measured signals and acts therefore as an input for the Simulink PL canvas
- Data ready signal
  This port sends a signal when new data is ready and acts therefore as an input for the Simulink PL canvas
- Address bus
  This ports sends the address of the desired channel to read out and acts therefore as an output for the Simulink PL canvas
- Enable signal
  This ports sends a signal when the Simulink PL canvas wants to read new data and acts therefore as an output for the Simulink PL canvas

Further information about this DRP communication interface can be found in [49, 56]. For a proper registration the following arguments are necessary for each internal port:

- InterfaceID e.g. 'Output data bus (XADC DRP)'
- InterfaceType e.g. 'IN'
- PortName e.g. 'XADC_DO_OUT'
- PortWidth e.g. 16
- InterfaceConnection e.g. 'xadc_wiz_0/do_out'
- IsRequired e.g. 'false'
  If there is a connection necessary to run the reference design, then this argument should be set to true.

To finish this step, a constraint file for the external ports of the reference design is needed. There is a template available to create a constraint file for the external ports of the reference design *XADC_vpvn_vaux0_vaux8.xdc*. For the analog inputs there would be no need to assign an IOSTANDARD but since there is an issue with this, a workaround is needed [50]. Hence it has to be set to the IOSTANDARD that is compatible with the digital I/Os in the bank. Therefore in our design it is set to the IOSTANDARD LVCMOS18 because the additional GPIO pins are set to this IOSTANDARD.

6.3.3 Register Reference Design

Step 4: The next step is to create the necessary folder structure for the registration of the custom Reference Design. This folder structure is explained in section 5.1.4. As a start point it is good practise to copy the provided base Reference Design from MathWorks. To do so, one can run section 4 in the script *utility_SetupXADC.m*. This

will generate the whole folder structure and also copy the necessary files. These files have to be adapted in the following steps.

Step 5: The before created files (*system_top.tcl* and *XADC_vpvn_vaux0_vaux8.xdc*) can be copied to the right place into the folder structure. They both need to be placed in the package folder named *+vivado_base_2014_4*. If we would have relied on custom IP cores in the Reference Design, they would also have to be copied to the right place and packed as a zip-container. This step is also explained in the script.

Step 6: The board plugin registration has to be adapted. Running section 6 of the provided script will open the right file to adapt. All existing board plugins have to be deleted except ZedBoard.plugin. This plugin has to be adapted to the created folder name. The board folder is a package folder therefore the '+' character does not have to be added. Save the changes and close the file.

Step 7: The board definition has to be adapted. Running section 7 of the provided script will open the right file to adapt. Follow the instruction in the script.

Step 8: The reference design plugin registration has to be adapted. Running section 8 of the provided script will open the right file to adapt. Follow the instruction in the script.

Step 9: The reference design definition has to be adapted. Running section 9 of the provided script will open the right file to adapt. Follow the instruction in the script.

Step 10: Finally the just created board support package has to be added to the Matlab search path to make the board available.

For more information about the registration of a board and/or a reference design the following commands will open the corresponding help files of MathWorks.

```
doc hdlcoder.Board
doc hdlcoder.ReferenceDesign
```

6.3.4 Implement a Simulink Example Model

This section describes how to implement a Simulink example model, based on the before created and registered custom reference design. To communicate with the XADC on the Zynq, the DRP communication interface has first to be programmed. Therefore a state machine block is very handy to fulfil this task. There are three differential channel to read out and four states are needed for each read out. These states are as follows:

- IDLE: In the idle state the address is incremented.
- ADDR: In the address state the address is written to daddr_o and the data enable port (den_o) is set to false.
- EN: In the enable state the data enable port (den_o) is set to true.
- WAIT: In the wait state the data enable port (den_o) is set to false and it waits for the data ready signal (drdy_i) to read data from the XADC and outputs on the output port (val_o).

The address specification of the used channels can be found in [49].

The Simulink model *XADC_Demo.slx* shows an example implementation uf such a system. To generate the bit-file for this model the HDL Workflow Advisor can be run

on the atomic subsystem *XADC_HP_HDL*. In task 1.1 the before generated Reference Design can be chosen. In task 1.2 the interface to the registered iIP cores, external ports and to the PS (AXI4-Lite) can be configured.

Hint: There is also a bit-file available for those who want to try out the model directly. The bit-file is named *XADC_Demo.bit*. To program it onto the ZedBoard via Ethernet, the following code can be run in Matlab:

```
z = zynq;
z.checkConnection
z.programFPGA('XADC_Demo.bit');
```

To try out the model, start the interface model *gm_XADC_Demo_interface.slx* in External Mode. Remember to set the simulation time to inf. There are multiple ways to test the XADC. One way could be to use a signal generator. Another way could be a small voltage divider circuit, where the output signal should be max 1 V. There is a pulsed output on port PMOD JA1 which outputs a 25 kHz rectangular signal with 55 % duty cycle and 3.3 V. To demonstrate the speed capability of the XADC, three FiFo buffers are attached, to store the data from the three XADC channels. This FiFo buffer can be read out from the interface model. The model is parametrised to store 10'000 data points. The sample rate of the processing system is parametrised to 1 kHz. Hence it takes 10 s to read out the FiFo buffer. The 10'000 samples which are acquired with the FiFo buffer correspond to a time of 100 µs.

Chapter 7

Conclusion and Expandability

In this chapter, a short review over the Thesis should sum up the work done. Noticed shortcomings are discussed and some options for improvements and further work are listed.

7.1 Summary

The overall goal of this thesis was to gather information about the ZedBoard and provide tools to make the first steps of a system engineer using this embedded system platform easier. This was done by developing the Development Framework and listing a lot of background information. Also, to provide an interaction method for the system's operation after the development phase, the Remote System Interaction methods were designed.

7.1.1 The Development Framework

The main tool for easing the implementation of an embedded system on the Zynq is the Development Framework. The achieved solution offers a simple way for the implementation uf a given system into a provided environment. This provides a given set of possibilities and options which can be used in a straightforward way. Techniques for testing an achieved system were also outlined and enable a comprehensive way of evolving a solution from the theoretical design to its implementation and test in a virtual environment.

The same goes for the implementation on the target platform. The code generation, deployment and test of a system onto the ZedBoard is outlined and assistance is provided for a more straightforward development process.

7.1.2 The Remote System Interaction

For the interaction with an implemented system, Simulink Library Blocks and a GUI were developed and implemented. Their use is described at large and the provided functionality offers a good flexibility and performance.

With the provided solution it is possible to control the system state, change and read parameters, log signals over time and to analyse step responses in detail. Also, the acquired data is stored for later use. This enables a flexible way to work with an implemented system from afar.

7.2 Shortcomings, Improvements and Possible Extensions

Of course the capabilities to achieve a perfect and thoroughly field-tested solution are not on hand when writing a thesis in such an enormously complex field. Therefore, an overview over noticed shortcomings and possible extensions should be given in this section.

7.2.1 The Development Framework

The Development Framework provides a basic functionality and especially a given set of available signals. For most systems, the given Framework should suffice. However, some additions and further automation or scripts could ease the use of it, for example:

- For now, all projects basing on the Development Framework are named identically. This could lead to confusions when working on several projects in parallel. Therefore it would be desirable to add the possibility to rename a project and propagate this name throughout all stages of the development process, e.g. the generation of the executable and bit-file with the corresponding project name.
- Since the achieved solutions tries to be as generic as possible, the signal names follow a fixed scheme. It would be possible to use a script to rename the used signals to more intuitive terms, which would be propagated throughout the whole framework for the engineer's convenience.
- The addition of more signals or the tailoring of the FiFo stage could also be done with scripts. This could analyse the requirements of a system in a defining list, check the available resources and by copying and adapting the given structures expand the capabilities on demand.

Performance Enhancement

For a better performance on a given implementation (which itself should be optimised of course), the Embedded Code Generation can be biased and optimised. The same goes for the Embedded HDL Coder. Currently the whole process of generating the necessary hardware description and bit-file for the PL is not analysed and influenced. However this is absolutely necessary when developing a system which should satisfy a real-world application.

Security Concerns

In using Matlab/Simulink for the development process, a proven and well-maintained tool is utilised. The same goes for the Embedded System Software which bases on Linux. However, no security aspects were regarded. A system even for serious beta-testing should be hardened against potential threats. This includes the usage of passwords, communication encryption, separation and/or tunnelling from other systems and networks and so on. Also, the maintenance of such a complex system must be done with care, especially by observing and resolving potential security flaws. It has to be repeated that the given tools and systems aren't fit for serious usage and deployment without modifications.

7.2.2 The Remote System Interaction

The Remote System Interaction as provided offers a basic toolkit to work with a remote system. Most common applications are covered and even some more advanced operations like the analysis of a step response with fast signal capture and readout is available. Of course a lot of enhancements and improvements conceivable. Some examples are listed below.

Connection Check

Since the communication between the target and the GUI uses UDP, the correct transmission of data is not inherently ensured. Therefore, a check mechanism for correct communication would be a good addition to the current GUI solution. Also a periodic check if the target is still running and a corresponding error message in the GUI would add a certain security of operation.

Lianx Library Blocks

The provided library blocks could be scripted for variant signal and parameter sizes such that only signals actually needed are available. This would expand their usability and effectiveness because only necessary data is transmitted. Also the support of floating point data could be added.

Another nice feature would be to add support for the standardised VISA Virtual Instrument Software Architecture. This would even allow the interaction with industry standard systems and software.

GUI

So far, the development of the GUI was based on Matlab and the there available tools. While doing so offered a steep learning curve, it also had some impacts on the performance of the final solution. For example, the development took part on a full hd screen and no regard was given to the resolution for other screens. There would be a nicer way to have a resizeable GUI which keeps the font size correct and adds scrollbars for a smaller screen resolution to keep the GUI nice and clean. A progress bar could visualise how long recording and then transmitting captured data still takes. Also, updating some of the GUI contents (especially the tables) takes a long time and thus limits the responsiveness. Other improvements could include a dynamic creation of the GUI which loads the necessary modules only, which fits the size of the GUI modules on demand, enables rearranging of modules or their display in separate windows. Another nice feature would be the support for several models which could be selected from the GUI and started on the target at will.

While some of these points could be optimised with Matlab, of course the GUI could be created anew with another program language which has its own advantages and disadvantages.

7.3 Verdict

The Development Framework eases the design and implementation of a system on the Zynq by leading the system engineer through the necessary steps and by takeing over some tasks and complexity of the system. All basic requirements are fulfilled and a working solution is delivered. For the analysis and enhancement of performance on the target, a lot of information was gathered and documented. However the elaboration of a generic solution to this problem is not feasible since the performance highly depends on the desired functionality.

The Remote System Interaction provides the demanded functionality. It enables the interaction with the remote target system with all basic features in a simple interface. The development focus was set rather on showing a running solution for each function than creating a fully custom and efficient software. The provided interaction method can be used as a good base for further work.

The documentation focuses mainly on the application of the elaborated solution. While most methods to satisfy a certain demand are outlined, also a lot of information can be found when looking inside the provided files and scripts. Even tough the application of the elaborated solution could not be demonstrated on the intended hardware, a sample implementation of a simple functionality demonstrates all features in a comprehensive way. The tutorials and given background information contribute to the overall substance as well.

To sum up, it can be stated that the elaborated solutions satisfy the posed demands in most cases. A broad knowledge about embedded systems, Matlab/Simulink and managing the design and implementation of complex systems was gathered and will provide a sound base for further work.

Appendix A

Additional Reading

The Zynq Book [9]

The Zynq Book is a comprehensive description of the Xilinx Zynq-7000 SoC. Besides the Zynq-specific parts, it also includes a lot of information about embedded systems in general. Additionally, there is a collection of tutorials available. All in all, the Zynq Book is a great way to get a good insight into the Zynq platform.

Managing Model-Based Design [1]

Roger Aarenstrup's book gives a good introduction to model-based design approaches. Traditional and modern methodologies are compared and the general benefit of model-based design are outlined. Practical hints how these tools can be introduced and made use of in real-life situations round up this well-structured and comprehensible book.

Simulink User's Guide [35]

The Simulink User's Guide is a comprehensive reference for this complex and powerful software. Divided in sections covering basics and more advanced topics, it provides useful information to new and experienced users alike.

Matlab Fixed Point Designer [26]

The Getting Started Guide to the Matlab Fixed Point Designer explains the concept of fixed point data types and how they are used in Matlab/Simulink. Further information is contained in User's Guide [28] and Reference Manual [27].

Simulink Embedded Coder Getting Started Guide [23]

The Simulink Embedded Coder Getting Started guide introduces the reader into the basics of generating executable code from Simulink models, specifically targeted for Embedded Systems. More information can also be found in the Embedded Coder User's Guide [25] and Reference Manual [24].

Simulink HDL Coder Getting Started Guide [29]

As for the Embedded Coder, the same documentation also exists for the Simulink HDL Coder which allows the generation of HDL code for hardware descriptions from models. Similarly the Reference Manual [30] and the User's Guide [31] provide more detailed information.

Writing a Simulink Device Driver Block [8]

This step-by-step guide helps with the development of custom Simulink blocks, in this case device drivers. While the examples target the Arduino device family, the given information is valid for any other platform as well.

The Mathworks Course on Programming Xilinx Zynq with Matlab and Simulink [33]

MathWorks offers a course which introduces the partipicant to the use of Matlab/Simulink to program the Xilinx Zynq. While all basic steps are outlined, it also covers several methods of testing the system in the development phase or more comprehensive tasks such as adapting a custom Reference Design or writing and handling IP cores. The course documentation contains all main information, but is not publicly available.

Introduction to Linux - A Hands on Guide [20]

While there are a lot of Linux introductions, Machtelt Garrels et Al. managed to condense a lot of information into a comprehensive guide. Both new users and experts can find hints and background information to make their life easier. Also, besides the Hands on guide, there are lots of other interesting resources available at [19].

Appendix B

Cheatsheets

The idea of this appendix is to give some hints on using Linux or Matlab more efficiently. This list of commands is not complete at all and only contains a fragment of all useful information. However, going through it sometimes can give the reader a new idea or hint on how something could be done.

A good source for cheatsheets is `http://overapi.com/`, which provides helpful command and tool lists for all kinds of software.

B.1 Linux Cheatsheet

Command	Explanation
» alias name='command'	create a command alias (for persistence: add alias commands to /.bashrc)
» apropos 'command'	search manpage database for command
» cat /proc/cpuinfo	pipe information about CPU to stdout
» cat /proc/meminfo	show memory related infotmation
» cat /proc/interrupts	list interrupts and the core they are running on (in /proc/irq those affinities can be changed)
» cd /path/to/dir	change directory
» dd if=/dev/source | pbzip2 -9f > ./target.img.bz2	make a compressed image of a device, for example an SD card
» bunzip2 -dc ./quelle.img.bz2 | dd of=/dev/ziel	unpack compressed image
» dd if=/path/to/source of=/path/to/target	redirect bitstream from source to target
» dhclient eth0	Invoke dhcp client on eth0 (use 'ifconfig eth0 ipadress' to set IP adress manualy)
» dmesg	show hardware-related log messages
» tail -f /var/log/messages	show tail of system messages
» du	show disk usage (also use 'du -sch /path/to/folder *' for more details about path)
» htop	more sophisticated top with options to show CPU affinity etc.
» ifconfig	show information about LAN connections
» iwconfig	show information about WLAN connections
» less /path/to/file	displays text files in a comfortable pager
» ls /dev/pts/	shows all logged in users
» lsof	lists open files, folders, unix sockets, IP sockets and pipes
» lsof -X -r 2 /dev/device	monitor device access in a 2 seconds interval
» man 'command'	show the manpage (users manual) of the given command
» mkdir /path/to/new/dir	make a new directory
» nano 'filename'	open a file in the nano text editor or create it
» nice 'pid'	set relative priortity (niceness) of a process
» ps aux	show running processes
» reset	reset shell output, eg. if mangled bitwise data messed up the shell
» rm -rf /path/to/delete	delete files or folders (recursively, forced)
» rsync	rsync is a very flexible file sync tool
» scp /path/to/source /path/to/target	copy file(s) via ssh (use user@ip:/path/to/file for remote side)
» screen 'command'	start program detachable
» screen /dev/ttyACM0 115200	open serial console on ttyACM0
» split -b 1024m /path/to/source /path/to/target	split source file into chunks of size b (which will be named target.#)
» cat target.split.* > source	combine file chunks target.split.# back together to source file
» ss -s	list currently established, closed, orphaned and waiting TCP sockets
» ssh user@remoteip	start ssh connection
» tar -I pbzip2 -cvf foo.tar.bz2 ./some/files	create compressed tarball from files
» pbzip2 -dc filename.tar.bz2 | tar -xf -	decompress
» taskset	sets affinity of a task to a cpu
» top	show task manager
» tty	find out in which serial console you are
» wget adress /path/to/target	download file via wget

B.2 Matlab Cheatsheet

General Commands	Explanation
» system('dir')	executes commands in OS terminal
» !command	executes terminal commands from matlab console
» !vivado &	opens vivado in own window
» clipboard('copy',which('zynq'))	get path of zynq to clipboard
» cs = getConfigSet('model_name','config_set_1_name')	get config file of Simulink model
» f = fullfile('myfolder', 'mysubfolder', 'myfile.m')	build full filename from parts (OS invariant)
» feature('DefaultCharacterSet')	show current character set
» feature('DefaultCharacterSet','UTF8')	change character set (startup.m)
» filesep	fileseperator OS specific (/ vs. \)
» license('inuse')	list all currently used licenses
» matlabshared.supportpkg.getInstalled	to check which hw-support pkg are installed
» methods('object')	list all methods of an object
» save('configfile.mat','cs');	
» ver	lists all installed toolboxes
» visdiff('configfile1.mat','configfile2.mat');	compare two files or folders
» winopen(pwd)	opens files on windows OS
Simulink Specific Commands	**Explanation**
» open_system('system')	opens Simulink model
» get_param('system/block', 'Param')	get block parameter
» set_param('system/block', 'Param', 'Value')	set block parameter
» set_param('lianx_lib_ps', 'Lock', 'off')	unlock ps library
» set_param('lianx_lib_ps', 'Lock', 'on')	lock ps library
Zynq Specific Commands	**Explanation**
» z = zynq('linux','IPAddress')	create zynq object (Linux Service)
» z.putFile('sourcefile','destination')	copy file from Matlab to target
» z.getFile('sourceFile','destination')	copy file from target to matlab
» z.programFPGA('system_top_wrapper.bit')	copy bit file on target and program FPGA
» z.execute('command',true)	executes a command on the target and feeds back its output

Appendix C

Digital Appendix

This is a short description of the Digital Appendix. It lists the folders which accompany this document and should help to understand the contents and structure.

- **01_documentation**
  This folder contains the main document of this Thesis. On the CD-version of the Digital Appendix, also a lot of referenced documents can be found. These are missing in the uploaded version due to restrictions in the total file size.
- **02_custom_ref_design**
  The custom Reference Design created for the Development Framework and the Demo Project can be found in this folder. The creation and structure of a Reference Design is explained in section 6.3.
- **03_development_framework**
  This folder contains the Simulink Project with all scripts and models for the development framework. Also, some parts for using the Remote System Interaction are already integrated here.
- **04_remote_system_interaction**
  The scrips and functions to generate the different parts of the Remote System Interaction are provided in here.
- **05_sim_library**
  The here provided Simulink Library contains all blocks which were developed during this Thesis.
- **06_demo_project**
  This folder contains the Simulink Project for the Demo Project with all appertaining files. Note that this is very similar to the Development Framework, but contains the implemented functions and structures for the demonstration.
- **07_system_software**
  Here the necessary files and scripts to customise the system software with the MathWorks buildroot toolchain can be found. Also, the image of the latest release of the Lianx System Software can be found here, together with some remarks and scripts for uploading it to the target.
- **08_tutorials**
  In this folder, all files necessary to work through the tutorials in chapter 6 are provided.

Working with the Framework or Demo Project

While the usage of the Development Framework is described in section 4.2, the setup of the Demo Project is explained in section 6.2. To work with those, at least a copy of the relevant folder as well as the folders *02_custom_ref_design* and *05_sim_library* are needed.

Bibliography

[1] Roger Aarenstrup. *Managing Model-Based Design*. 1st ed. CreateSpace Independent Publishing Platform, 2015. ISBN: 978-1512036138. URL: http://www.amazon.co.uk/Managing-Model-Based-Design-Roger-Aarenstrup/dp/1512036137%3FSubscriptionId%3DAKIAJEOIHAJER6RL7KQQ%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D1512036137.

[2] *About FreeRTOS*. URL: http://www.freertos.org/RTOS.html.

[3] Yair M. Altman. *Accelerating MATLAB performance: 1001 tips to speed up MATLAB programs*. Chapman & Hall Book. Boca Raton, Florida: CRC Press, 2015. ISBN: 9781482211306. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10991614.

[4] Analog Devices, ed. *Linux Industrial I/O Subsystem*. URL: https://wiki.analog.com/software/linux/docs/iio/iio.

[5] Avnet. *ZedBoard: Avnet Product Brief*. Ed. by Avnet. URL: http://zedboard.org/sites/default/files/product_briefs/PB-AES-Z7EV-7Z020_G-v12.pdf.

[6] Avnet. *ZedBoard (Zynq Evaluation and Development): Hardware User's Guide*. Ed. by Avnet.

[7] Avnet. *Zynq Intelligent Drives Kit 2: High Performance SoC Motor Control*. Ed. by Avnet. URL: http://zedboard.org/sites/default/files/product_briefs/PB-AES-ZIDK2-ADI-G-v7.pdf.

[8] Campa Giampiero. *Writing a Simulink Device Driver block: a step by step guide*. Ed. by Inc The MathWorks. 2015.

[9] Louise H. Crockett et al. *The Zynq Book: Embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC / Louise H. Crockett ; Ross A. Elliot ; Martin A. Enderwitz ; Robert W. Stewart ; Department of Electronic and Electrical Engineering, Univ. of Strathclyde*. 1. ed. Glasgow: Strathclyde Academic Media, 2014. ISBN: 9780992978709.

[10] Derek Molloy. *Writing a Linux Kernel Module: Part1: Introduction*. URL: http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction.

[11] Dr. Jeremy H. Brown and Brad Martin. *How fast is fast enough? Choosing between Xenomai and Linux for real-time applications*. Ed. by Inc. Rep Invariant Systems. URL: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf.

[12] elinux.org. *Device Tree*. URL: http://elinux.org/Device_Tree.

[13] Embedded Centric. *Embedded Operating Systems*. URL: https://embeddedcentric.com/embedded-operating-systems/.

[14] FreeRTOS. *Certification*. URL: http://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS-Safety-Critical-Certification.shtml.

[15] Girish Rao Bulusu. "Asymmetric Multiprocessing Real Time Operating System on Multicore Platforms". Masterthesis. Arizona: Arizona State University, 2014.

[16] *Introduction to Linux Interrupts and CPU SMP Affinity*. URL: http://www.thegeekstuff.com/2014/01/linux-interrupts/.

[17] Beck Kent. *Make It Work Make It Right Make It Fast*. URL: http://c2.com/cgi/wiki?MakeItWorkMakeItRightMakeItFast.

[18] kernelnewbies.org. *Where Do I Begin*. URL: http://kernelnewbies.org/FAQ/WhereDoIBegin.
[19] *Linux Documentation Project Guides*. URL: http://www.tldp.org/guides.html.
[20] Machtelt Garrels. *Introduction to Linux: A Hands on Guide*. URL: http://www.tldp.org/guides.html.
[21] MathWorks. *Bug Report 1293244*. Ed. by MathWorks. 2016. URL: http://www.mathworks.com/support/bugreports/1293244.
[22] MathWorks, ed. *Buildroot: faq-troubleshooting*. URL: https://github.com/mathworks/buildroot/blob/master/docs/manual/faq-troubleshooting.txt.
[23] MathWorks. *Embedded Coder: Getting Started Guide*. Ed. by MathWorks.
[24] MathWorks. *Embedded Coder: Reference*. Ed. by MathWorks.
[25] MathWorks. *Embedded Coder: User's Guide*. Ed. by MathWorks.
[26] MathWorks. *Fixed-Point Designer: Getting Started Guide*. Ed. by MathWorks.
[27] MathWorks. *Fixed-Point Designer: Reference*. Ed. by MathWorks.
[28] MathWorks. *Fixed-Point Designer: User's Guide*. Ed. by MathWorks.
[29] MathWorks. *HDL Coder: Getting Started Guide*. Ed. by MathWorks.
[30] MathWorks. *HDL Coder: Reference*. Ed. by MathWorks.
[31] MathWorks. *HDL Coder: User's Guide*.
[32] MathWorks, ed. *MathWorks Buildroot*. URL: https://github.com/mathworks/buildroot.
[33] MathWorks. *Programming Xilinx Zynq SoCs with MATLAB and Simulink*. Ed. by MathWorks | Training Services. 2015.
[34] MathWorks. *Simulink Projects: Simulink in Teams*. Ed. by MathWorks. URL: http://www.mathworks.com/discovery/simulink-projects.html.
[35] MathWorks. *Simulink: User's Guide*. Ed. by MathWorks.
[36] Michael Katz. *Speed Up Your GUIs With Profiling*. Ed. by MathWorks. URL: http://blogs.mathworks.com/community/2011/06/28/speed-up-your-guis-with-profiling/.
[37] Microsoft, ed. *Windows Embedded: Automotive 7*. URL: http://download.microsoft.com/download/0/A/1/0A1E07D6-7562-4566-AACF-E04DF4FF8879/Windows%20Embedded%20Automotive%207%20Datasheet.pdf.
[38] *Multi-core API debuts*. URL: http://archive.linuxgizmos.com/multi-core-api-debuts/.
[39] *Multitasking - Essential to Any RTOS*. URL: http://electronicdesign.com/embedded/multitasking-essential-any-rtos.
[40] *open-amp: Open Asymetric Multi Processing framework project*. URL: https://github.com/OpenAMP/open-amp.
[41] *OpenAMP heterogeneous multicore standard targets Linux*. URL: http://hackerboards.com/openamp-heterogeneous-multicore-standard-targets-linux/.
[42] *Process niceness vs. priority*. URL: http://askubuntu.com/questions/656771/process-niceness-vs-priority.
[43] Ray Walker. *Examining Load Average: Understanding work-load averages as opposed to CPU usage*. Ed. by Linux Journal. URL: http://www.linuxjournal.com/article/9001.
[44] RehiveTech, ed. *RSoC Framework: framework utilized in asymmetric multiprocessing application*. URL: http://rsoc-framework.com/rsoc-framework-utilized-amp-application/.
[45] Wikipedia. *KISS principle*. URL: https://en.wikipedia.org/wiki/KISS_principle.
[46] Wikipedia. *Memory-mapped I/O*. URL: https://en.wikipedia.org/wiki/Memory-mapped_I/O.

[47] Wikipedia. *Windows Embedded Industry*. URL: https://en.wikipedia.org/wiki/Windows_Embedded_Industry.

[48] William E. Shotts. *Writing Shell Scripts*. URL: http://linuxcommand.org/lc3_writing_shell_scripts.php.

[49] Xilinx. *7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter: User Guide UG480*. Ed. by Xilinx.

[50] Xilinx. *IO Standard for Vp_0/Vn_0*. Ed. by Xilinx. URL: https://forums.xilinx.com/t5/7-Series-FPGAs/IOSTANDARD-for-VP-0-VN-0/td-p/402733.

[51] Xilinx. *Multi-OS Support: (AMP & Hypervisor)*. Ed. by Xilinx. URL: http://www.wiki.xilinx.com/Multi-OS+Support+%28AMP+%26+Hypervisor%29.

[52] Xilinx. *OpenAMP*. Ed. by Xilinx. URL: http://www.wiki.xilinx.com/OpenAMP?responseToken=97db321952bf041c97c2ef10ec6b6860.

[53] Xilinx. *Real-Time Linux*. Ed. by Xilinx. URL: http://www.wiki.xilinx.com/Real-Time+Linux.

[54] Xilinx. *Vivado Design Suite Evaluation and WebPACK*. Ed. by Xilinx. URL: http://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html.

[55] Xilinx. *Vivado Web Install Client*. URL: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2015-2.html.

[56] Xilinx. *XADC Wizard v3.0: LogiCORE IP Product Guide: PG091*. Ed. by Xilinx. URL: http://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf.

[57] Xilinx. *Xilinx Leads Multicore Association OpenAMP Group to Accelerate Time to Market of Heterogeneous System Development*. Ed. by Xilinx. 26. Januar 2016. URL: http://press.xilinx.com/2016-01-26-Xilinx-Leads-Multicore-Association-OpenAMP-Group-to-Accelerate-Time-to-Market-of-Heterogeneous-System-Development.

[58] Xilinx. *Zynq-7000 All Programmable SoC: Technical Reference Manual*. Ed. by Xilinx.

[59] Xilinx. *Zynq-7000 All Programmable SoCs: Product Tables and Product Selection Guide*. Ed. by Xilinx. URL: http://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf.