

Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering

Automotive control unit with CAN and FlexRay

Jiří Záhora

2018

Supervisor: Ing. Martin Vajnar

Acknowledgement / Declaration

I would like to thank the supervisor Martin Vajnar for his straightforward leading of my work. Next, Michal Sojka for his help in my general problems during the work and Jiří Kadlec for his fast and exhaustive introduction to the FPGA problematic. Last but not least, the hackerspace MacGyver for providing a workshop and tools for making and assembling my device.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. 5. 2018

.....

Abstrakt / Abstract

Tato práce se zabývá návrhem automobilové řídicí jednotky s rozhraními CAN, FlexRay a Ethernet, jejíž hlavní účel je poskytnutí komunikačního rozhraní mezi automobilem a externí platformou pro vývoj algoritmů autonomního řízení.

První část této práce poskytuje přehled zmíněných komunikačních standardů a popis použitých vývojových nástrojů. Ty jsou především vývojový kit Microzed, tvořící jádro celého zařízení, Vivado design suite, pro práci s programovatelnými logickými poli kitu, a operační systémy používané v embedded systémech. V následující části jsou diskutovány všechny kroky hardwarového návrhu a výběr vhodných komponent. Zde je největší prostor věnován zajištění obsluhy FlexRay sběrnice. Dále je stručně okomentován postup portování driveru FlexRay sběrnice na systém Linux. Závěr práce je věnován testování vytvořeného zařízení a zhodnocení jeho funkčnosti.

This thesis deals with the proposal of car control unit with a CAN, FlexRay and Ethernet interfaces, whose main purpose is to provide a communication interface between a car and an external control unit for the development of self-driving algorithms.

The first part of this thesis provides an overview of aforementioned communication standards and description of development tools. These are mainly the Microzed development kit, the core of the whole device, the Vivado design suite for work with a programmable logic array of the kit, and operating systems used in embedded systems. In following part, all steps of hardware design and choosing of suitable components are discussed. Here, the main focus belongs to putting the FlexRay bus into operation. Afterward, a porting of FlexRay driver implementation to the Linux system is briefly commented. End of this thesis is devoted to testing the device and evaluation of its function.

Contents /

1 Introduction	1	4.1 Power supply	21
1.1 Goal of this thesis	1	4.2 FlexRay controller connection .	22
1.1.1 Meaning of the term		4.3 Bus transceivers connection ...	23
“real-time”	2	4.4 PCB design	23
2 Theoretical background	3	5 FPGA design	25
2.1 CAN bus overview	3	5.1 SJA1000 IP Core	26
2.1.1 SocketCAN Linux driver ..	4	5.2 Finite State Machine	
2.2 FlexRay bus overview	4	for communication with	
2.3 AXI4 bus overview	5	MB88121	27
2.3.1 AXI-Lite signal de-		5.3 AXI interconnect IP core	29
scription	6	5.4 System reset IP core	29
2.4 Schedulers in operating sys-		5.5 Clock generating IP core	29
tems	7	5.6 GPIO and interrupt routing ...	30
2.4.1 FIFO	8	6 Hardware design summary	31
2.4.2 Round-robin	8	7 FlexRay driver porting	32
2.4.3 Deadline schedulers	8	8 Experiments and testing	33
3 Components and development		8.1 Basic CAN interfaces test	33
tools	9	8.2 Basic FlexRay interfaces test ..	33
3.1 MicroZed development board ...	9	8.3 Round-trip time measure-	
3.2 FlexRay controller MB88121 ..	10	ment	34
3.3 Vivado design suite	12	8.3.1 Measurement results	34
3.3.1 Creating a custom IP		8.4 Precise measurement of a	
in IP packager	13	delay on the FlexRay gateway .	36
3.3.2 Note on handling tris-		8.5 Power consumption	38
tate buffers	13	9 Future work	39
3.3.3 Writing the IO con-		10 Conclusion	40
straints file	14	References	41
3.3.4 Debugging of FPGA		A Shortcuts	43
design	14	B Schematics	44
3.3.5 Versioning of Vivado		C PL block diagram	52
project	15	D Obsah příloženého CD	53
3.3.6 Running Vivado on a		E Assignment of this thesis	54
remote server	15		
3.4 Embedded Linux	15		
3.4.1 Device Tree	16		
3.4.2 Access to hardware			
from userspace	16		
3.4.3 Real-time Linux	16		
3.4.4 Useful libraries for par-			
allel applications	17		
3.5 FreeRTOS	17		
3.6 Rapid Prototyping Platform			
(RPP)	18		
3.7 Bus traffic analyzers	18		
3.8 Microzed APO education kit ..	20		
4 Electronic design	21		

Chapter 1

Introduction

The modern-day automotive industry relies more and more on a X-by-wire control. Classical mechanical solutions are replaced by electronics circuits. An electronic control of the main engine is nothing new. Also, driving assistants such as power steering or safety systems such as ABS are standards. An automatic gearbox, an adaptive cruise control or a parking assistant is becoming common in modern cars. In the end, we can say that absolutely everything is driven by electronic signals in nowadays cars.

With the development of artificial intelligence and increasing computational performance of embedded systems, a challenging task of developing self-driving cars arises. Sure, there are already several successful self-driving car models in the real operation. But the know-how is often proprietary and still, a lot of problems need to be solved. Problems range from a fast and precise car localization over an optimal trajectory planning to recognizing car surroundings.

At the Department, many algorithms for solving these problems are in a development. We already have a 1/10 model car for testing the self-driving algorithms. My colleagues are quite successful in this area, what shows their recent victory in the international F1/10 Autonomous Racing Competition. Thus, we would like to move our algorithms to the next level and test them on a real car.

To be able to communicate with the real car, we need to somehow send control commands inside the car system. It's not so easy at all. The car system consists of several tens of control units. These units are interconnected by several different bus architectures. Also, some units can supervise other units, to check if they operate correctly, by inspecting communication on the various buses. If we want to test the algorithms for autonomous driving, that will run on some control unit, with the real car, we need to be able to communicate with the other control units present in the car.

1.1 Goal of this thesis

In this thesis, I will develop a new control unit. Its main purpose is to provide a gateway function between several different bus interfaces in the car. These are mainly the CAN bus and the FlexRay bus. Via this gateway, we want to integrate our computational control unit into the car system. Because this device is intended for use on the real car, it should work reliably and in real-time.

This goal consists of several tasks. First, we need to choose or create a suitable hardware. Then port or implement all needed drivers and functionality. In the end, the basic performance tests have to be done.

We will work mainly with the CAN and the FlexRay buses since they are most commonly used for communication between ECUs. The CAN bus is well known and time-proven. The FlexRay is relatively new protocol. FlexRay bus is not so commonly used as the CAN, so the availability of open source drivers and controllers is somewhat limited, also possibilities for obtaining FlexRay controller chips are scarce. Therefore, the main focus of this thesis will be in a handling of the FlexRay.

It is out of the scope of this thesis to have a self-driving car. The goal is just to prepare a suitable instrument for future work on this problem.

■ 1.1.1 Meaning of the term “real-time”

The term “real-time” is often used in this text. The meaning of this term may vary in different applications and needs to be specified. It means that on every one event in the system, the response comes, in the worst case, within some specified time interval. An example of the event and the response in our gateway system is receiving a message from the bus and forwarding it.

I derived this worst case delay from the speed of a human response. This is usually taken as 0.1 s. Most of the events, which we want to manage, serve for some control. The control reference is set by a human or by some system which replaces a human action. A control speed has to be much faster. We have to take into account a possibility of summing delays on several system segments. Thus, I conservatively want to have the maximum 5 ms worst case delay on my gateway for the most critical events.

Chapter 2

Theoretical background

In this chapter we first introduce the CAN and FlexRay protocols. Then we look at AXI4 bus with the main focus on the AXI4-Lite version of this protocol. At the end of this chapter, the main principles of scheduling in operating systems are discussed.

2.1 CAN bus overview

Controller Area Network (CAN) is a standard widely used in automotive since the nineties. The main purpose of this norm is to provide a simple and reliable interface for communication between multiple MCU's (typically dozens) inside vehicles. Basically, CAN is a multi-master serial bus allowing message priority. According to ISO OSI model, CAN defines the physical and the data link layer.

On a physical layer CAN consists of one twisted pair of cables with the typical impedance of $120\ \Omega$. Each bus also has to be terminated by a resistor of this resistance. Bits are transferred by a differential signal. When logical 0 is transferred, also called a dominant bit, the voltage on the wires is driven to 0 and 5 V. When logical 1, called a recessive bit, is transmitted, the bus is not driven and there is zero voltage between the wires caused by termination resistor. All nodes should be synchronized. Because there is not shared clock signal between nodes resynchronization occurs at each recessive to dominant transition. Each message is transmitted in a frame. To determine which node should transmit at a time all nodes receive what they transmit. At the beginning of the frame, all nodes start to transmit simultaneously. In a case of transmitting the recessive bit and receiving the dominant bit, the node should stop the transmission and start again in a next frame.

On a data link layer, we can show how the frames should look like. The basic format of the frame is the following. First dominant bit demarks start-of-frame followed by identifier (ID) bits. After there is control sequence declaring a count of data bytes following data. At the end of the message comes CRC code and acknowledgment. Value of ID part can determine the priority of the message. ID with an earlier recessive bit has lower priority. After transmission of ID, just one node should keep transmitting. The standard length of ID is 11 bits but also some variants allow 29 bits length. Maximum of data per one transfer is 8 bytes. This was a brief description of the most commonly used data frame. Next types are the remote frame for request of data, error and overload frame. The last two types serve for reporting of unexpected states on the bus.

There are several variants of CAN definitions varying mainly in maximum allowed bit rate. In our project we use high speed CAN with the maximum speed of 1 Mbit/s on a several meters long bus. Next variant is, for example, CAN FD (flexible data rate). It allows switching to up to eight times faster transfer during the data part of the frame.

2.1.1 SocketCAN Linux driver

An open source implementation of socketCAN is part of the Linux kernel. It provides for a programmer interface similar to UDP and TCP/IP networking based on Berkeley sockets. This driver allows multiple applications to access one CAN device or one application to handle multiple CAN devices. For setting up the communication first we have to set up CAN interface and create a socket. Afterwards, we need to bind the socket to a CAN interface. Then we can simply use `read()` and `write()`.

2.2 FlexRay bus overview

In a modern day, automotive industry demands on safety, reliability and data rates rapidly increase. In order to fulfill these needs a new standard was developed. The FlexRay bus is a protocol which should replace the CAN bus in the most critical automotive applications such as steer-by-wire, brake-by-wire, etc. Like a CAN, the FlexRay also defines how the communication should look like on the physical and the data link layer.

Generally, the physical layer of the FlexRay is similar to the CAN. It consists of one or optionally two channels. Each channel is a twisted pair of wires with the impedance between 80 – 100 Ω with a differential signal. The optional second channel can be used for duplicated communication increasing reliability by data redundancy or for doubling data rate. The maximum allowed data rate is 10 Mbit/s on one channel. Because the FlexRay bus is time triggered, all nodes in a bus have to have a clock drift less than 0.15 % from the reference clock.

There are main differences between CAN and FlexRay in how the communication is organized and how the single one bit is transferred. A single message cycle consists of a static and dynamic segment, symbol window and idle time. For the static segment, there is a fixed schedule. Each node has its own time slot where the other nodes cannot transmit. This segment is used for the transfer of the most safety-critical data, which we typically need to send periodically. In the dynamic segment, the nodes can process arbitrary communication with the assigned priority. Here it is the same like on the CAN bus. The symbol window is used for network maintenance. A short portion of the network idle time (NIT), where no data is transferred, is used for a global time synchronization. In the simplest FlexRay variant, only the static segment and NIT are mandatory.

The one message cycle usually takes about 1 ms and is driven by macroticks (1 μ s typical). One macrotick consists of several microticks, the smallest time unit of the local clock. The number of microticks per macrotick can vary on different nodes. We need to synchronize the start time of messages at each node. For this purpose, at least 2 nodes in a FlexRay cluster has to transmit a synchronization message in their static segment. From the time of receiving synchronization messages, a time shift is computed. Then the desired amount of microticks is added or removed in the NIT. For measuring of global time, macroticks and 6-bit cycle counter are used. The zero time is the start of a frame with cycle counter value equals to zero.

The data are transferred in frames. The standard format of the frame is shown in the figure 2.1. As in a CAN frame, we can see the frame ID in the header, followed by data and CRC bits. In addition to CAN, the cycle counter bits are contained.

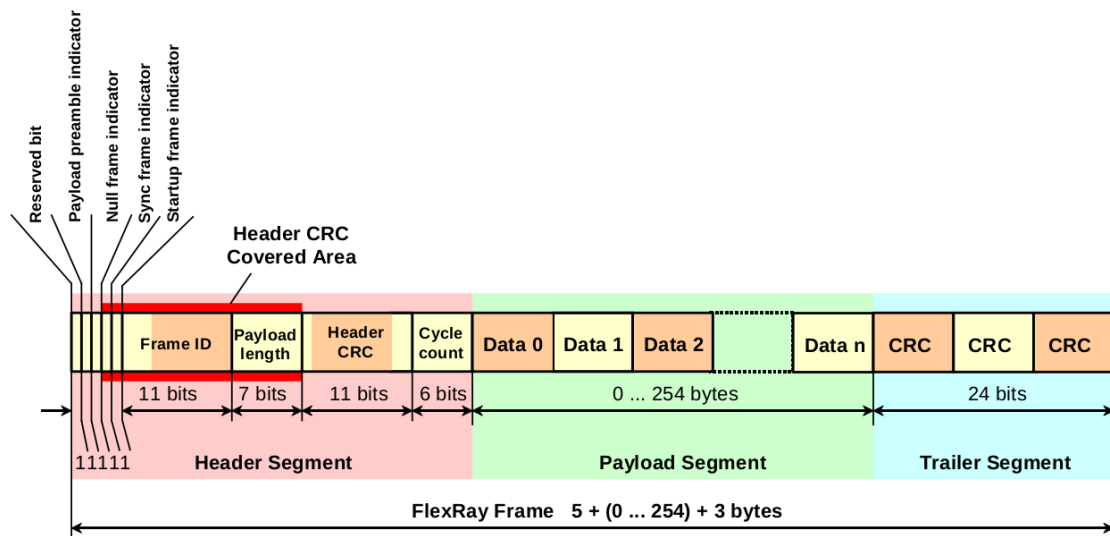


Figure 2.1. FlexRay frame format, taken from [1].

At the level of a single bit transfer, the FlexRay provides a fault tolerant mechanism for receiving bits. Each bit is held for several clock cycles and its value is determined as a majority of received bytes. This approach can filter a single bits faults caused by incorrect synchronization or a clock drift.

More information about FlexRay specification can be found in [1].

2.3 AXI4 bus overview

Advanced extensible interface (AXI) is a parallel bus designed for interconnection of a processor with its peripherals. In this project, we will use it for an integration of the CAN and the FlexRay bus controllers into the system. The bus architecture is a master-slave with independent parallel channels for a data and address reading and writing and one channel for a write response. This interface is suitable for direct mapping of peripherals to the processor memory. The full AXI specification provides a variable data and address buses width. We will use a less complex version AXI-Lite for a basic connection of peripheral with fixed size registers. There also exists the AXI-Stream version which is designed for streaming of a large amount of data from or to the peripheral.

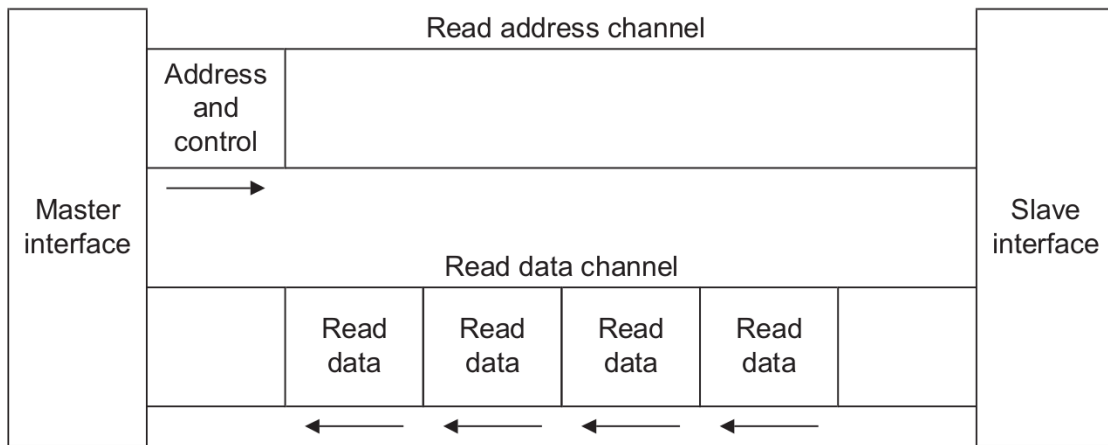


Figure 2.2. AXI bus channel architecture of reads, taken from [2].

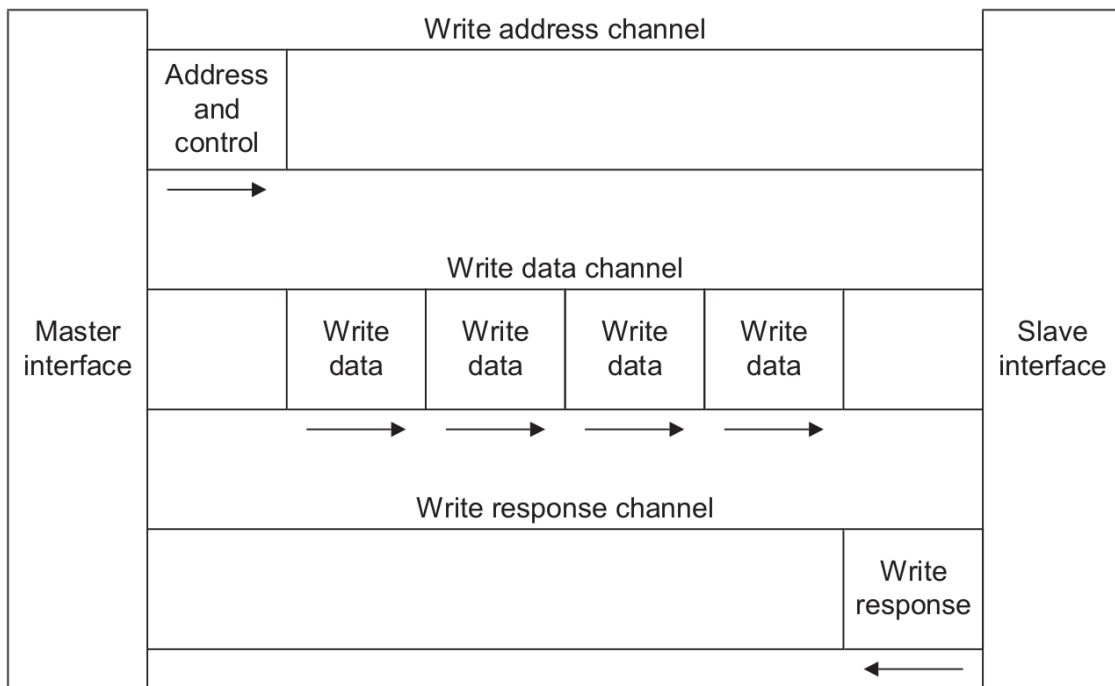


Figure 2.3. AXI bus channel architecture of writes, taken from [2].

2.3.1 AXI-Lite signal description

Because we will need to implement our own AXI-Lite peripheral, in this section, I discuss all needed signals in detail. All signal names and design flow are taken from [3].

Read Address Channel

S_AXIARADDR

Address bus from master to slave peripheral.

S_AXIARVALID

Valid signal, asserting that the S_AXIARADDR can be sampled by the slave peripheral.

S_AXIARREADY

Ready signal, indicating that the slave is ready to accept the value on S_AXIARADDR.

Read Data Channel

S_AXI_RDATA	Data bus from the slave peripheral to the master.
S_AXI_RVALID	Valid signal, asserting that the S_AXI_RDATA can be sampled by the master.
S_AXI_RREADY	Ready signal, indicating that the master is ready to accept the value of the other signals.
S_AXI_RRESP	A response status signal showing whether the transaction completed successfully or whether there was an error.

Write Address Channel

S_AXI_AWADDR	Address bus from master to slave peripheral.
S_AXI_AWVALID	Valid signal, asserting that the S_AXI_AWADDR can be sampled by the slave peripheral.
S_AXI_AWREADY	Ready signal, indicating that the slave is ready to accept the value on S_AXI_AWADDR.

Write Data Channel

S_AXI_WDATA	Data bus from the master to the slave peripheral.
S_AXI_WVALID	Valid signal, asserting that the S_AXI_WDATA can be sampled by the master.
S_AXI_WREADY	Ready signal, indicating that the master is ready to accept the value of the other signals.
S_AXI_WSTRB	A strobe status signal showing which bytes of the data bus is valid and should be read by the slave.

Write Response Channel

S_AXI_BREADY	Ready signal, indicating that the master is ready to accept the BRESP response signal from the slave.
S_AXI_BRESP	A response status signal showing whether the transaction completed successfully or whether there was an error.
S_AXI_BVALID	Valid signal, asserting that the S_AXI_BRESP can be sampled by the Master.

All channels control the transactions by handshaking using READY and VALID signals. When these signals are asserted the transaction on a channel is processed in one clock cycle. During an implementation, it is important to take care of correct handling of these signals. For example, waiting for the READY signal before asserting the VALID signal can lead to a deadlock.

Through the response channel, we will send just an acknowledgment that the data writing was successfully done. Full AXI implementation defines a several error states but we won't need them in our project.

2.4 Schedulers in operating systems

Our final application should work in the soft real-time. This means that every task finishes its execution within some defined time interval. Only in some special and rare

situations is allowed to overrun the deadline. This interval can vary between tasks according to their period and priority. Usually, there are many tasks ready to be executed at a single point in time. A task execution is scheduled by the operating system. Several scheduling policies are commonly used for this purpose. Generally, we divide scheduler algorithms into groups based on their priority handling (static or dynamic) and preemptive policy (preemptive or non-preemptive). The static priority means, that task's priority is the same all the time. Whereas, in the dynamic priority case it can change in time. The preemptive scheduler means, that a running task is stopped when a new task with a higher priority becomes ready for execution.

Differences are in schedulers used in the Linux kernel and real-time operating systems. A good comparison of real-time scheduling in the Linux and in the RTOS can be found in the article [21]. Briefly, the Linux kernel, extended to the real-time version, offers a more options for a scheduler configuration than a typical RTOS.

Here are introduced main policies used by schedulers.

■ 2.4.1 FIFO

This is the simplest scheduling policy. Each new task is added to the end of a task queue before all other tasks with lower priority. The running task runs until it is blocked or preempted by a new task with a greater priority. The disadvantage of this algorithm is that the processed task can run for a long time and another task has to wait.

■ 2.4.2 Round-robin

An improvement of the previous method is the round-robin scheduling. Here a time quantum is set and each task is executed for the maximum of this quantum, then it is sent to the end of FIFO and next task is executed. The problem with the FIFO and static priority task are that low priority tasks can wait for a long time until all other higher priority tasks are finished. This can be solved by the dynamic priority scheduler, where the priority of tasks increases with time.

■ 2.4.3 Deadline schedulers

For the most critical tasks, the dynamic priority can be set according to a task deadline. We have to determine the deadline for each arriving task. Then the “earliest deadline first” scheduling policy is used.

Chapter 3

Components and development tools

This chapter provides a description of basic parts used for building the gateway. These are the MicroZed board and FlexRay bus controller Cypress MB88121. Next, we will discuss how to work with Xilinx Vivado design suite and how to implement a custom peripheral inside the programmable logic array using this toolkit. The end of this chapter briefly shows how to configure a Linux distribution to run on the platform with custom peripherals.

3.1 MicroZed development board

The core of the gateway is MicroZed board with Xilinx Zynq-7000 system on a chip. The Zynq chip contains dual-core ARM Cortex-A9 processing system (PS) and FPGA programmable logic (PL). There are also several peripherals such as USB, Ethernet, two CANs, GPIO and other common interfaces directly connected to the PS. The main advantage of this chip is the already mentioned PL which allows us to implement all the missing hardware we need for the gateway. A detailed description can be found in the Zynq-7000 technical reference manual [4].

On the MicroZed board are assembled all components needed for running the Zynq processor, without any carrier board. The core is a power supply cascade with voltage level supervisors. The board can be supplied through the micro USB from a PC or through the barrel jack connector, which optionally can be assembled on the board, or through the supply pin from the carrier board. The power supply over the USB port is suitable just for basic testing of standalone MicroZed.

The processor contains DDR memory controller so 1 GB DDR3 RAM is also populated on the board. The maximum frequency of DDR is 1066 MHz. Another memory present on the board is 128 MB flash on a QSPI bus. The flash memory can be used for a system deployment. The last memory we can use is a microSD. The maximum supported size is 32 GB. The SD card will store a Linux kernel, a root filesystem and a program performing the gateway function.

For communicating with the user a USB port is present on the MicroZed board. The basic use is like a serial line terminal to allow user to monitor and interact with the running system. The next interface we can directly use is Gigabyte Ethernet port. It significantly reduces the time needed for the development of software because it is possible to set the system to boot over an Ethernet network. From a practical point of view, it is much faster than to download a new program to the SD card after each little change. The last interface ready for use on the standalone MicroZed is JTAG connector. The JTAG is used for a bare metal programming of the processor or for deploying the PL design. The important ability of the JTAG is monitoring signals inside the PL on the running system. But our design during this project was not so complicated so I didn't need to use this feature for a debugging. For interacting with the MicroZed board over the JTAG interface a special JTAG/USB converter is needed. This converter is not supplied with the MicroZed board and has to be ordered separately.

Other peripherals have to be routed out from the MicroZed through two 100 pins microheaders. On microheaders, we can access PL ports and several GPIO's. Also, the JTAG, the reset pin and power supply pins for MicroZed are here. It is possible to connect all PS peripherals through the PL ports. The PCB with all expansions is called the carrier board. How the design should be done is described in the document [5]. The design of the carrier board is one of the main quests of this project. Details can found in chapter 4.

3.2 FlexRay controller MB88121

Next core component of the gateway is the FlexRay controller. There are a lot of processors with integrated FlexRay peripheral available but all of them usually provide just one dual channel interface. There is not freely available and fully working HDL implementation of IP (intellectual property) core which we could easily port to the Zynq PL. There is one project at the faculty at the Department of Measurement where they developed their own FlexRay IP core running on the programmable logic. More information related to this project can be found for example in [15] But this core was developed on a different platform and for a performance testing purpose. Porting it to the Zynq platform would be quite time-consuming. So we decided to use an external bus controller and integrate it into the system. There are no many standalone FlexRay controllers available on a market. Some producers, which offers these components, are NXP Semiconductors, Cypress Semiconductor or Infineon Technologies. The NXP MPC5777M MCU provides what we need, but its FlexRay core is based on the different implementation, for which we would need to write a new driver. From all alternatives, the best one is the standalone FlexRay bus controller MB88121 from Cypress because it contains the Bosch E-Ray FlexRay core for which our department already has a driver code from some older projects. The next important reason is that this chip was the only one available on the stock of electronics components suppliers.

As already mentioned, this chip provides FlexRay bus controller. Also, it contains 8 KB message RAM for input and output buffers and it provides filtering of received messages. The maximum allowed speed on the bus is 10 Mb/s which is the maximum speed defined by FlexRay specification. For connecting to the bus we need to add a physical layer transceivers to the chip. I choose the TJA1082, the same as was already used in the projects with FlexRay at the department.

The MB88121 provides several possibilities how we can connect it to the system. These are the 16-bit parallel multiplexed and non-multiplexed bus and SPI bus. The SPI bus is the most simple one but it is serial and it can run at the maximum frequency of 8 MHz. The maximum possible data rate on the FlexRay bus is greater then we can transfer via SPI bus so this interface is not suitable for our purpose.

We will use the parallel non-multiplexed interface. It provides the fastest access to the chip registers. The bus consists of 16 data wires (D15 to D0), 11 address (A10 to A0) wires, bus clock (BCLK) and bus control signals. These are active low chip select, read, write and active high ready (RDY) signal. Address wires always have the input direction to the chip. A direction of data pins depends whether read or write signal is asserted. If none of them is asserted, data pins are in the high impedance state. The RDY signal is output from the chip and it signalizes if data on the bus are valid or already processed. The datasheet [6] doesn't specify the maximum frequency of the

bus clock. In examples¹, they use 32 MHz BCLK so I set this frequency as the allowed maximum.

Figures 3.1 and 3.2 show the timing diagrams of the reading and the writing operations. Internal registers of the MB88121 chip have 32-bit width. For this reason, we need to process two writes or two reads operation with the 16-bit data bus in one access.

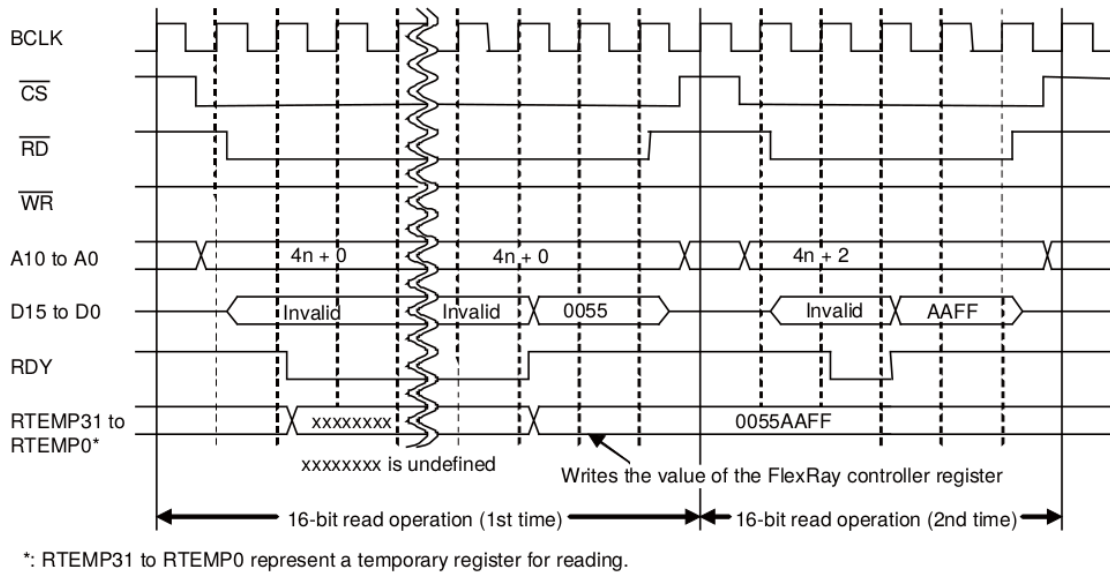


Figure 3.1. MB88121 read register timing diagram in 16-bit non-multiplexed mode, taken from [6].

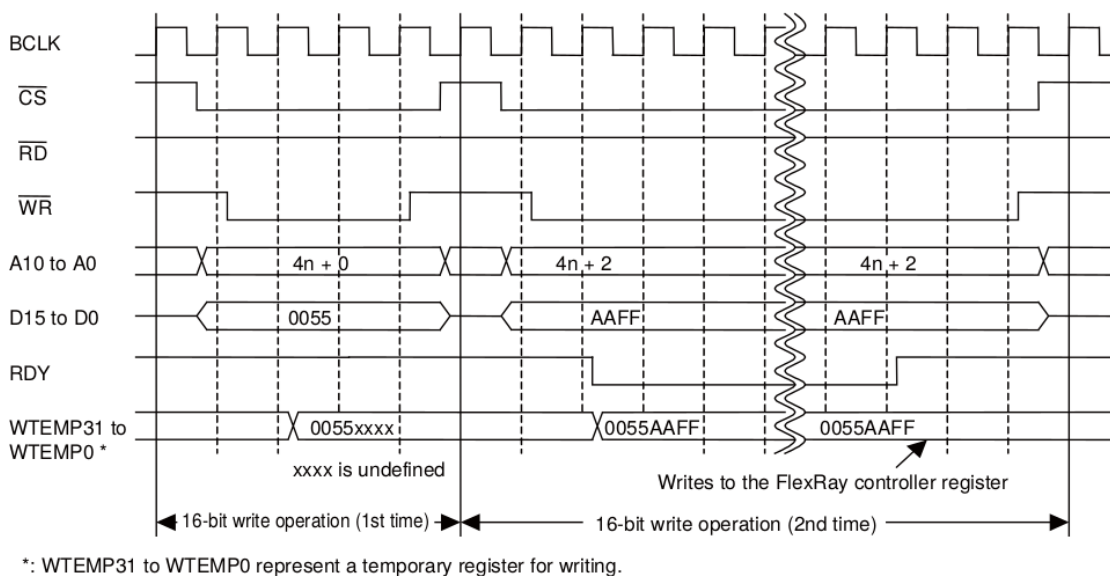


Figure 3.2. MB88121 write register timing diagram in 16-bit non-multiplexed mode, taken from [6].

¹ Section “Operation in 16-bit non-multiplexed parallel bus mode” of the datasheet [6]

Based on these timing diagrams we can implement finite state machine (FSM) in the Zynq PL for the bus control. This FSM should be connected and mapped through the AXI bus to the PS to provide access to the chip.

Next, three interrupt signals are available in the parallel non-multiplexed mode.

The last important notice is about handling the RDY signal during the write operation. From the timing diagram 3.2 one can understand that RDY pin is de-asserted after each full write operation. But testing the chip showed us that the RDY signal falls down only when writing to some registers, not all of them. So the FSM cannot expect the RDY low every time but has to handle it when this situation occurs.

3.3 Vivado design suite

For working with Xilinx FPGA we will use Vivado IDE. It provides intellectual property (IP)-based design. Each IP can be independently instantiated and configured and interconnected with the system via the AXI4 interface. On each design level, Vivado allows us to process logic simulation, IO and clock planning, power analysis, a definition of constraints and a lot of other features.

On the lowest level, a behavior of IP's is defined with the use of VHDL or Verilog language. I choose the VHDL for my work. Vivado also provides an IP core generation from C or C++ code. This process is called high-level synthesis (HLS) and it is a way how a program running on the PS can be parallelized by moving it into hardware peripheral. The HLS is useful for example for DSP functions which can be quite hard and time-consuming to describe in HDL. Our project doesn't require computationally demanding functions so I didn't study the HLS in more detail.

There are hundreds of Vivado manuals on the Xilinx websites. The most common approach how to work with this IDE is via its GUI. But all control is based on Tool control command (Tcl) language which is widely used standard. We can see all processed Tcl commands in the Tcl command line in the GUI window.

With default workflow, also called project mode, the Vivado automatically generates a project directory tree. All project files are automatically managed and tracked by Vivado. A second possible approach is a non-project mode. Here we have to create our custom Tcl scripts for managing design sources and process all necessary steps. We use the project mode workflow. Although a project directory tree seems to be unnecessarily extensive, my experiments with non-project workflow lead to problems with synchronization and propagation of changes in source files into the whole design. The list of Tcl commands implemented in Vivado is in the manual [7].

Design steps are showed on a diagram in a figure 3.3. First, we have to create custom IP blocks or use predefined IP's from Vivado IP catalog. Next step is to configure all IP's and interconnect them in a block diagram. Here Vivado offers a lot of automation. Both in automatic configuration of default blocks properties and an automatic connection of corresponding signals. Afterwards, we need to define IO ports and associate them with physical pins of Zynq's PL. This can be made in IO planning editor in Vivado GUI. This generates a constraints file with a port to package pin assignment and some other properties definitions. Next steps are a synthesis, which converts the design into interconnection of basic FPGA components and implementation, which computes how the design is routed in a real FPGA. The last step is generating a bitstream file for PL configuration.

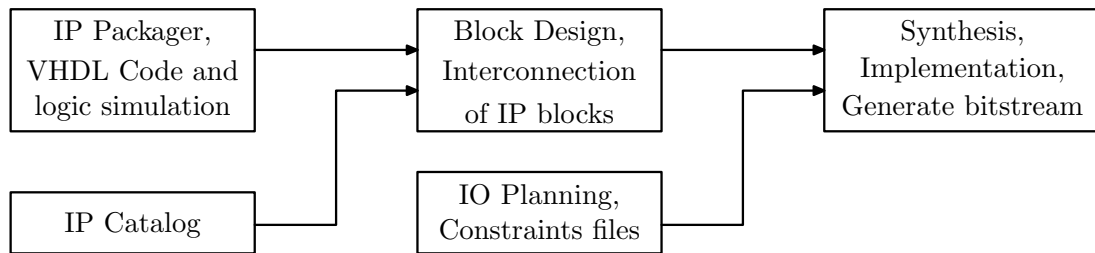


Figure 3.3. Vivado IDE design flow.

■ 3.3.1 Creating a custom IP in IP packager

An IP packager provides a way how to create a new IP core block for use in the Vivado IDE. It is possible to package HDL sources or the whole project. When creating a new IP block, the Vivado offers a predefined AXI4 interface with memory-mapped registers coded in VHDL language. These default files are a good starting point when implementing a custom AXI slave IP core.

Next ability of the IP packager is a definition of IP blocks ports and parameters. To each port is possible to assign a special signal type such as a clock or reset signals. These properties are used for optimizations during synthesis and implementation. Several ports associated with one interface should be grouped together to provide more organized block diagrams. IP packager allows us to define such interfaces. All the new IP blocks and interface definitions have to be sourced in the main Vivado project manually. Then, the Vivado manages all changes in IP sources automatically.

■ 3.3.2 Note on handling tristate buffers

Sometimes it is necessary to use bidirectional signals. In this project, it is the case of the data signal from and to the MB88121 FlexRay controller. The standard way how to code input-output buffer in VHDL is described in [8]. The following schema is recommended.

```

entity three_st is
    port(T : in std_logic;
         I : in std_logic;
         O : out std_logic);
end three_st;

architecture archi of three_st is
begin
    O <= I when (T='0') else 'Z';
end archi;
  
```

This code tells that port O is output or in the high impedance state depending on the tristate control signal T. The problem arises when we use this approach inside the Zynq FPGA. Internally there exists just logic zero or logic one signal values and nothing like the high impedance state. So although this structure is allowed by VHDL language, the logic simulation doesn't show any mistakes and the synthesis and implementation don't throw any errors or warnings, the resulting design works unpredictably wrong. In the worst case, it can lead to shunt between a connected peripheral, driving signal to some logic level and wrongly set Zynq pin, driving to the opposite logic level.

For correct handling of tristate signals, we need to work with a triple of input, output and tristate control signals in the whole design. Each tristate signal triple should be grouped into a tristate interface predefined in Vivado. After, correct port settings are

done automatically by Vivado. Going through generated source files showed us that tristate control is implemented similarly to the previous listing in an HDL wrapper file generated from the block design. The high impedance state is allowed only on the outermost level when working with physical PL ports.

3.3.3 Writing the IO constraints file

There are several parameters which need to be specified during the I/O planning phase. These are stored in already mentioned constraints file (.xdc - Xilinx design constraints). We can edit them with IO planner editor or by hand in a .xdc file with the use of Tcl commands `set_property` and `get_ports`. The main parameters are listed in the following table 3.1. The full Tcl command in the constraint file can look as follows.

```
set_property PACKAGE_PIN P16 [get_ports mb_clk];
```

Tcl Property	Values	Description
PACKAGE_PIN	W14, R18, ...	Number of package pin/pad.
IOSTANDARD	LVC MOS33, LVC MOS18, LVTTTL, ...	I/O standard defines voltage levels, port impedance, etc.
PULLTYPE	PULLUP, PULLDOWN, NONE	Setting the internal resistor pull type.

Table 3.1. Parameters of I/O ports

Ports of the PL are organized in banks. The Microzed platform contains two banks available for PL interfacing. Some parameters, like an I/O standard derived from bank voltage supply, we want to set for all ports in the bank. This can be simply done by a single Tcl command.

```
# Set the bank voltage for IO Bank 34 to 3.3V by default.
set_property IOSTANDARD LVC MOS33 [get_ports -of_objects [get_iobanks 34]]
```

I didn't find a manual describing all possible parameters of Tcl commands for creating I/O constraints. During their discovery, I used example constraints files or output of Tcl command line in the Vivado GUI or intuition.

3.3.4 Debugging of FPGA design

The Vivado offers several tools for testing and debugging the design in different development phases. First is the behavioral simulation. We use this for a validating of our HDL codes. The behavioral simulation is suitable for short pieces of code describing basic components. In the simulator, we create a test-bench for the tested component. In the test-bench, inputs signals are defined and applied to the component. Then, outputs are simulated and the user has to determine if they are correct or not. Writing of the test-bench needs almost the same effort as an implementing of components. For more complex components, it can be hard to determine and test all possible input combinations.

Second is a timing simulation. It is done on a synthesized FPGA design. Here, we can check a signal delay and a jitter on physical logic blocks, as it would be wired in the chip. Because our application is relatively slow in comparison to capabilities of the used FPGA, I didn't use this type of simulation. I expected that timing requirements are easily met in every configuration.

In simulations, we can hardly cover all possible states of the whole design. It is good to see what happen inside the FPGA during a runtime. An in-circuit logic analyzer can

be composed to the design. An arbitrary signal can be selected to be monitored. In the block diagram, we can use the Integrated Logic Analyzer IP core. This core provides an interface via the JTAG connector. The JTAG is automatically recognized by the Vivado and an analyzer GUI is a default part of the IDE.

■ 3.3.5 Versioning of Vivado project

The project directory contains a lot of automatically generated content, compilation products and a lot of files that are not in a human-readable format. It is not a good idea to track changes by git or other versioning systems. Since the Vivado projects are internally managed using Tcl commands, we can export a Tcl script for recreating the project in any development phase. The script is created by the following command.

```
vivado -mode batch -nolog -nojournal -source recreate.tcl
```

This script together with HDL design sources should be versioned in git. But there is one annoying issue. If we make some changes in a project recreated by Tcl script, these changes are not propagated into all automatically generated files. For example, if we add some physical ports in the top level block diagram, we have to manually change an HDL wrapper used for synthesis. This should be normally done by Vivado. I didn't discover if this behavior is a feature or a bug. To save an effort with this and similar issues, I did just backups, not versioning during the project development.

■ 3.3.6 Running Vivado on a remote server

The synthesis and the implementation are quite time-consuming and computationally demanding process. On a laptop, with i5 CPU and 8 GB RAM, it takes between 5 to 10 minutes to generate a new bitstream file ready for deploying to the PL after every little change in the Vivado project. This really slows down the project development. But it is possible to run bitstream generation remotely on some server.

If we have a good network connection, the simplest option is to mount our directory with installed Vivado and the project directory to the server machine. For this purpose we use the program `sshfs`, which needs to be installed on the server. Afterwards, we can run a custom Tcl script for building the project by the following command.

```
vivado -mode batch -nolog -nojournal -source build_remote.tcl
```

Vivado in the batch mode processes a given Tcl script without running GUI. All generated files are then stored on a local computer in the mounted project directory.

By this workaround, we can move computations to the remote machine. Not all steps of synthesis and implementation can be parallelized on multiple processors so time savings are not so significant as I hoped. I expect a greater advantage of this approach when computing an HLS which is much more computationally demanding. But I didn't test it because I didn't need to use the HLS in this project.

■ 3.4 Embedded Linux

The basic applications running on the Zynq can be written as a standalone C program. The Zynq processor is relatively powerful so we need some operating system for running advanced applications using different resources and requiring some level of multitasking. There are two options. One of them is to use FreeRTOS real-time operating system, second is to use the Linux operating system. At the department, there is already a

configured and tested Linux kernel image for the Microzed platform. I decided to use this Linux image and adjusted it for the gateway.

Usually, we don't need a fully featured Linux platform in the embedded applications. But it is necessary to have a more user control over hardware peripherals than we usually need when working with a Linux distribution running on a personal computer. For making a custom distribution containing only necessary kernel parts and a basic root file system is suitable to use the Buildroot tool¹ and Xilinx Linux kernel². Both of them is freely available on the Github.

In the kernel configuration, we specify which packages and peripheral drivers are needed for our platform. A user-friendly GUI for configuration pops up after calling `make menuconfig` in the kernel build directory. In a similar way also the Buildroot can be configured.

■ 3.4.1 Device Tree

It is a data structure describing hardware components so that the operating system can use them. Each new peripheral created in the PL, which should be controlled by the kernel, needs to be added to the device tree. The Vivado offers a tool for generating the device tree for custom PL peripherals. Manual how to generate the device tree is on the webpage [17]. The generated folder contains a file `pl.dtsi` which can be included in the main device tree file. After this step, the kernel can work with all IPs in the PL.

■ 3.4.2 Access to hardware from userspace

The main purpose of the Linux operating system is to “separate a user and the hardware”. In embedded applications, we want to work with a hardware. For this purpose, we need to use a kernel drivers providing some interface for working with peripherals. A lot of them can be controlled by an `ioctl` command³.

In the case of memory mapped peripheral, there is no needed for Linux kernel drivers or device tree definitions. All we need to do is to tell the Linux where the peripheral is mapped. The routine for this purpose can be made based on the `mmap` command. This routine is also already available and tested in our department. When the peripheral is mapped into the userspace, it is possible to implement device driver outside the kernel space.

■ 3.4.3 Real-time Linux

A standard way of using the Linux on a personal computer doesn't require real-time behavior. But for embedded applications, the Linux offers several ways how it can be configured to provide responses fast enough for real-time systems.

First, we don't use the whole Linux kernel but just the most basic parts. Also, we don't need to focus on a power consumption optimality, thus all system suspending functions are disabled. It reduces overheads significantly.

Next, we can manage how the Linux schedule the execution of processes. Functions for work with the scheduler are available in the library `sched.h`⁴. By default, the Linux process is scheduled with the `SCHED_OTHER` attribute. This means that it is a non-real-time process with a static priority 0. The running thread is chosen from the static priority 0 list, based on a dynamic priority determined inside the list.

¹ <https://github.com/buildroot/buildroot>

² <https://github.com/Xilinx/linux-xlnx>

³ `man ioctl`

⁴ <http://man7.org/linux/man-pages/man7/sched.7.html>

For real-time processes, a priority greater than 0 is set and different policy is selected. The `SCHED_FIFO` attribute for simple priority queue execution can be used. Or `SCHED_RR` for the round-robin policy. Hard-real-time processes can set the `SCHED_DEADLINE` attribute. Here also the runtime, the deadline and the period of the execution have to be set in addition to the priority.

The last way how to make the Linux real-time is to use `realtime-preempt` patch set. Now, these patches are not part of the mainline kernel, so they must be installed additionally. These patches provide for example a better handling of interrupts. This means that only necessary workaround is done inside an interrupt handler before it returns to the normal execution.

■ 3.4.4 Useful libraries for parallel applications

In addition to standard Linux libraries, we may need to use some other libraries, which make our work easier. For multiple threads and shared resources management serves the `pthread.h` library. From this library, we will use an interface for work with semaphores and mutexes. This mechanism allows us to protect critical sections in the code against a access from multiple threads at one time.

Our device should work as a gateway in the final application. This means that it should handle several I/O sources at one time. Messages on different I/O's can come with different time intervals and it can be non-deterministic. We have to process each message as soon as possible after it comes. The way how to achieve this is to poll all sources of messages. Some of I/O's can also generate interrupts. We can write such an application with the use of Pthreads (POSIX Threads) library and create several threads for several I/O sources. It is easy to make a mistake in a multiple threads management which can easily lead to a deadlock. The `libev`¹ library will save our effort. It is an event loop for handling different types of events. The one type is timeout event, the second is I/O event, which we use. The I/O event is a new input on the I/O source. When the event occurs, an assigned callback function for processing a new message is executed. We don't need to do any other work than to write callbacks with the use of this library.

■ 3.5 FreeRTOS

Some hardware, which we will use during a gateway testing, uses the FreeRTOS². This is small and simple real-time operating system kernel implementation written in C. It is deterministic and fast. The FreeRTOS application is organized in independent tasks. Each task has assigned its priority. Here higher number means higher priority. Tasks are scheduled by the round-robin algorithm. For tasks and resources management, the FreeRTOS offers mutexes, semaphores, timers and other mechanisms. The usage and all features are described in the document [9]. There are no advanced features, such as device drivers, memory management, etc. Some libraries are written to provide more functionalities, such as lwIP, a lightweight open source TCP/IP stack designed for embedded systems, and others.

All settings of FreeRTOS are done in one file `FreeRTOSConfig.h`. For example, we can configure the scheduler to be preemptive or we can set the system tick interval. When working with timers, a special care has to be taken to not work with the timer which drives system ticks. Otherwise, the application can behave wrongly.

¹ <http://software.schmorp.de/pkg/libev.html>

² <https://www.freertos.org/>

3.6 Rapid Prototyping Platform (RPP)

For testing the gateway, a working FlexRay bus with some traffic is needed. We have a testbed consisting of several RPP boards at our department (figure 3.4). The RPP board was also developed at the department. Its description can be found in [19]. It is a board based on the TMS570 safety MCU with FlexRay peripheral, several power outputs and others peripherals. Its purpose is a development of power control applications, with a possibility of code generation from Simulink.

On the testbed, a steer-by-wire demo application is implemented. Two RPP boards with DC motors with encoders are connected through the FlexRay bus. One DC motor simulates a steering wheel, second an actuator. A position of a shaft is measured, then sent via the FlexRay bus to the second board which controls the motor to received position.

The testbed will be used for experiments. It is possible to switch steer-by-wire demo to use a CAN bus too, so the whole gateway can be tested here.

For a work with testbed and RPP boards, we need a Code Composer Studio IDE from Texas Instruments. Here, we can write and compile the code and also debug the application via JTAG interface. All source codes and support packages for RPP board were developed with the version 5 of Code Composer Studio. Using the newer version can be problematic.

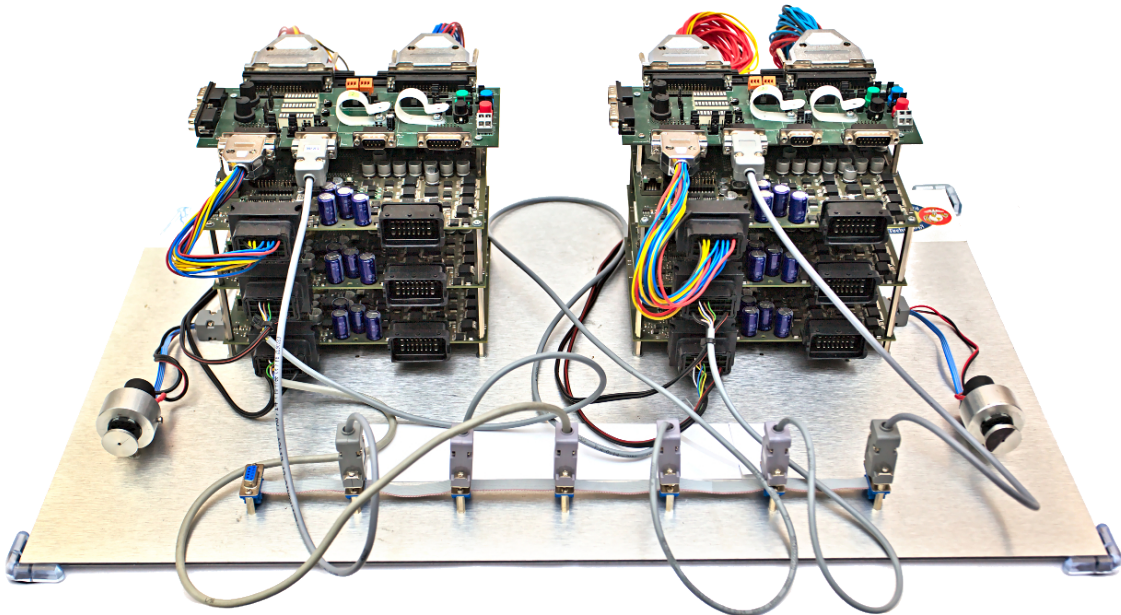


Figure 3.4. Testbed with RPP boards

3.7 Bus traffic analyzers

During an application development, it is good to see what happens on the bus. For the CAN bus, we have the USB2CAN device. This is a converter from CAN to USB, which can be connected to the PC and serves as a regular CAN node. The connection to the CAN bus is done by the standard DB9 connector with pinout showed in the following table.

Pin	Signal	Description
1	-	No connection
2	CANL	CAN low bus line
3	CAN GND	CAN Ground
4	-	No connection
5	CAN_SHLD	Connected to CAN GND via $100 \Omega/0.1 \mu\text{F}$
6	CAN GND	CAN Ground
7	CANH	CAN high bus line
8	-	No connection
9	-	No connection

Table 3.2. Pin assignment of USB2CAN device

Next, we have the VN3600 FlexRay analyzer from the Vector company. It works only on a Windows machine with the specialized commercial software CANoe. The CANoe offers a lot of features from a bus monitoring to a simulation of a whole CAN or FlexRay cluster with a graphical interface. A data format in the CANoe can be defined with use of FIBEX (Fieldbus Exchange Format). Arriving messages are handled as events. The event-based language CAPL is used for writing applications in the CANoe. A documentation of the CAPL language can be found in [16]. For the FIBEX format, I didn't find any comprehensive documentation. In this project, we use the CANoe tool just for simple monitoring of traffic on the FlexRay bus. I had need to study the CANoe deeply.

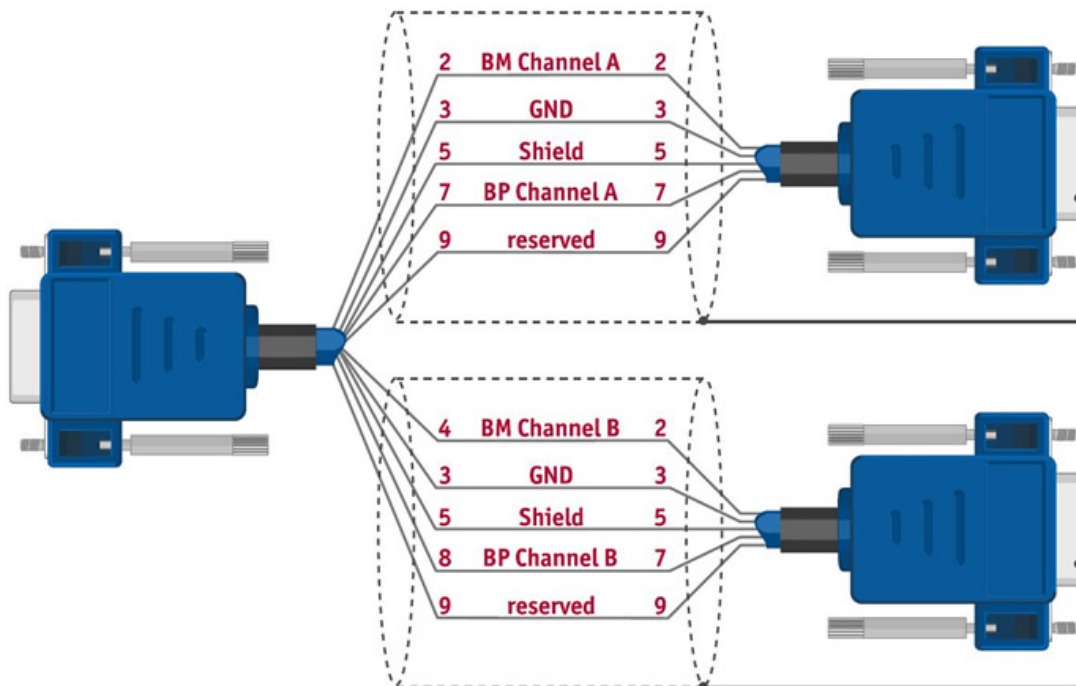


Figure 3.5. Pinout of FlexRay AB to A and B channels cable [20].

The VN3600 has one dual channel FlexRay interface. It serves for passive monitoring of the traffic without any influence on the FlexRay cluster, or it can be used as regular FlexRay node for interconnecting the real cluster and simulation in CANoe. Both FlexRay channels are routed through a single DB9 connector. A cable wiring for routing the signal to two distinct channels is shown in the figure 3.5.

3.8 Microzed APO education kit

A great resource for inspiration and starting point of my design is education kit MZ APO. It is a device based on the Microzed board, developed by Pavel Píša. We have several pieces of this kit at the department for the teaching of the Computer Architectures course.

The device consists of the Microzed and carrier board. The carrier board contains an LCD display, LED diodes, which can be used for visualization of a 32-bit word, three rotational (IRC) inputs connected by the SPI, and several other peripherals for demonstration basic principles of a computer system. The whole description of the kit can be found at the Computer Architectures course webpage [22]. The project is also available at the git repository¹.

On this kit, I learn how to work with the Microzed board. Next, it was an inspiration how to properly create a power supply for the Microzed and how to create an FPGA design.

¹ http://rtime.felk.cvut.cz/gitweb/fpga/zynq/canbench-sw.git/shortlog/refs/heads/microzed_apo

Chapter 4

Electronic design

Following chapter focus on concrete steps and decisions made during the development of hardware part of the gateway. Namely design of the carrier board for the Microzed. We will discuss main components used, schematics and some rules of placing components and routing tracks on a PCB. All schematics referred by this chapter are in appendix B.

4.1 Power supply

The core of each electronic device is a power supply. Our gateway is powered by standard car-battery. This means that the input voltage is normally about 12 V, but the gateway has to stay operational with a much lower input voltage. Such a situation can occur during a car starting when the battery voltage rapidly steps down. For this reason, the input voltage is filtered by a large battery of capacitors. The purpose of capacitors is to slow down input voltage changes, so the main regulator can handle them.

Between the input jack and capacitors is a diode. Its purpose is a protection against polarity switch. Also, it protects capacitors against a discharging back to the input during a voltage decrease.

I choose switching regulators for the Microzed power supply. From all technologies, switching regulators have the greatest efficiency, up to 90 %, so they produce less heat. Negative of switching regulators is noise caused by switching. For making the noise as low as possible we need to take care of placing external components around regulator circuit. Mainly, the output capacitor, inductance, and diode which are connected as shown in figure 4.1. On a PCB, These components should be placed close to the regulator circuit so that area inside current loops I_1 and I_2 is small. The inductor has to be shielded and feedback track routed out of these current loops.

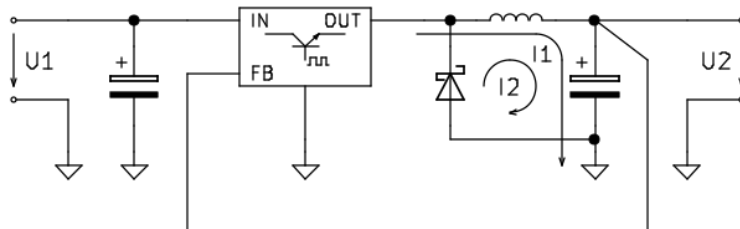


Figure 4.1. Switching regulator simplified schematic.

Inside Microzed board, there is a power supply cascade for creating all needed voltage levels for running Zynq processor and included peripherals, except the supply for banks with I/O pins from PL. The cascade is driven by series of power good (PG) signals as shown in figure 4.2. The last PG signal is used as a power-on-reset (POR) signal. The carrier board is designed to provide 5 V input voltage for main supply of Microzed. On

the output of 5 V regulator is voltage supervisor circuit. Its output is used as power enable signal (PWR_Enable in the figure). Next regulator is for creating 3.3 V VCCIO voltage for PL banks. This regulator is driven by VCCIO_EN signal from the Microzed. An output of 3.3 V supervisor is connected to the PG_Module. All PG signals are open-drain, the output from the supervisor circuit is also open-drain so we can safely use it as next POR.

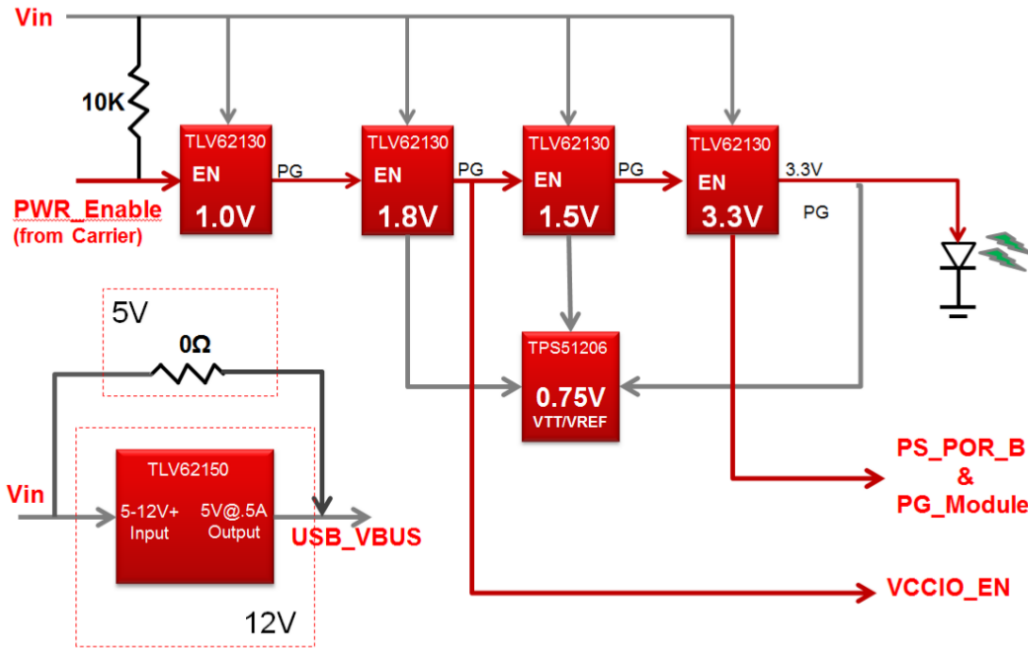


Figure 4.2. Power supply cascade on the Microzed board, taken from [10].

4.2 FlexRay controller connection

Function and interface of the controller MB88121 were already discussed in a section 3.2. We need to connect two controllers. Data, address and bus clock wires are common for both of them. Bus control signals have to be routed separately to each controller. When we are routing high-frequency signals, we should have in mind some rules of design. Because the speed of signal propagation is not infinite, the first step is to determine a length of wires where problems can occur. When the wavelength of a signal is comparable or less than a length of tracks, signal values on the opposite ends of the wire can be different. Length and impedance of all tracks of the bus have to be the same, to provide the same delay on each track. In some applications, a signal propagation delay has to be estimated and used when driving a bus. Differential signals are commonly used in high-speed applications.

From the datasheet [6], the maximum frequency of the bus is 32 MHz. With a signal speed approximated by the speed of light, we get a signal wavelength of about 10 m. Length of wires on the PCB is several centimeters long. That's much less than the wavelength so I don't expect problems caused by slow signal propagation. But still, it is a good practice to route tracks with the similar lengths and with constant distances between tracks. The most noise producing signal is the bus clock. For reducing this noise, a ground plate is split around the whole length of this track.

Other pins of controllers are connected according to the datasheet. Close to each power supply pin is a capacitor filtration. 8 MHz crystal oscillator is used. Unused

input pins are pulled down by a resistor, output pins are left unconnected. Mode selection pins are hardwired to select 16-bit non-multiplexed bus mode. Next time I would route mode selection pins to pin headers because I needed to change their configuration during the debugging process to a simple SPI mode when I checked if the controller is operational.

4.3 Bus transceivers connection

Each FlexRay controller needs two FlexRay transceivers. The TJA1082 chips are populated. The transceiver needs two voltage supplies. One for a logic level, in our case it is 3.3 V. Second for a FlexRay bus driving, it is 5 V. Both voltage levels are already present in the carrier board power supply circuit. A bus plus (BP) and bus minus (BM) signals are routed out from the board via a standard DB9 connector. The FlexRay bus can optionally be terminated by 100 Ω resistor connected to pin header. For a monitoring of bus and transceiver status, the chip contains two status bytes which can be read via SPI bus. For a simple error detection, just one wire can be used. More details about TJA1082 chip are in its datasheet [11].

Both, SPI line and simple error signal are connected to the PL. But only one way of error and status detection can be used at a time. To activate the simple error detection mode, logic zero has to be connected to the SPI chip select signal and logic one to the SPI clock.

A connection of CAN bus transceivers is similar. Again we need two voltage levels, 3.3 V for the logic and 5 V for the bus. Bus signals are also routed via the DB9 connector and can be terminated by 120 Ω resistor with use of a jumper. Two CAN controllers are present in the Zynq's PS, third is implemented in the PL.

We don't use CAN standby mode thus a logic zero is connected to the standby activation signal. This is done in the PL so we can start using standby mode in the future.

4.4 PCB design

The main idea for the design of the carrier board PCB was to place Microzed footprint into the middle of the PCB such that USB and ethernet connectors are at the edge of the board. Then put all peripheral components around the Microzed and power supply components as far as possible from the processor and all signal lines. Double sided PCB was enough for routing all wires. The board was made by the Pragoboard company by the POOL service method. We set the design rule limits in a PCB editor according to the specification provided by the manufacturer. Next time I would use higher tolerance and larger vias than those presented by the supplier. During assembling, I found several nonconductive vias which I had to repair.

Purpose of one already finished board was testing and debugging the gateway design. Thus it contains several indicating LED diodes, several push buttons for manually driving some signals which purpose was not clear from the Microzed documentation and test points for easier signal monitoring. Several design bugs were made in the first board, such as mirrored footprints for DB9 connectors, wrongly placed labels in silkscreen layer and others details. This thesis contains already corrected schematics and PCB layouts. All user buttons were removed from the carrier board because they are unnecessary. A reset button is already on the Microzed board. Only one user-LED

connected to the PL for its basic testing and power on LED are still presented in the new carrier board design.

On the testing board, all components were soldered by hand and some of them were several times reassembled. The mode selection pins on one MB88121 chip were reconnected during the debugging process so an ugly hack had to be made on the board. After finishing the debugging process, the board is fully working but not suitable for reliable use in the real car. A next board should be created from the provided production materials.

The testing board is depicted in figure 4.3. Battery of capacitors were not assembled because we connected the board only to a laboratory power supply, not to the real car battery. A layout of all connectors and other parts, important for user, is highlighted. In the new design, layout of these interfaces is almost the same.

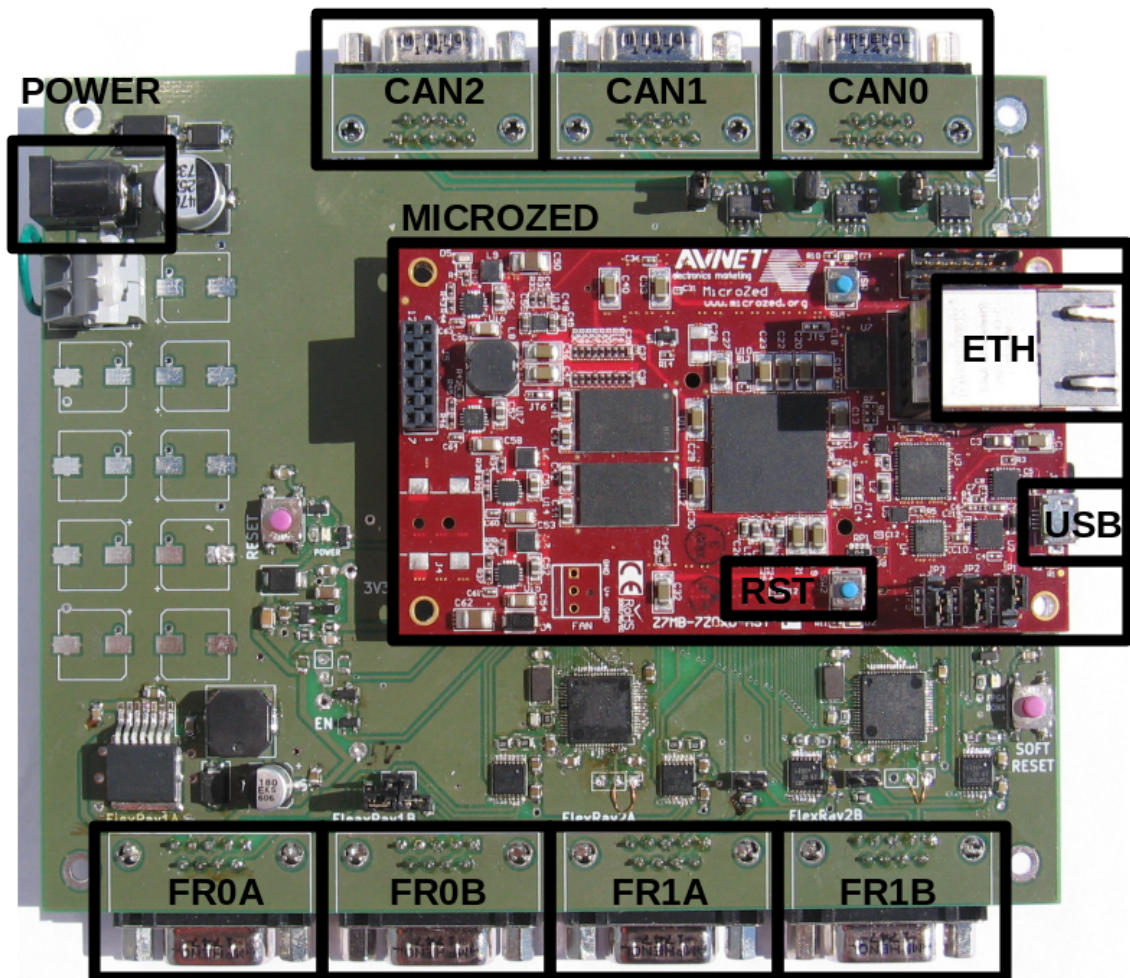


Figure 4.3. Carrier board.

Chapter 5

FPGA design

This chapter continues in the description of hardware design. Here, we comment the work which was done with the programmable logic arrays on the Zynq processor. All used IP cores are briefly introduced, and one self-made IP core is described in a more detail.

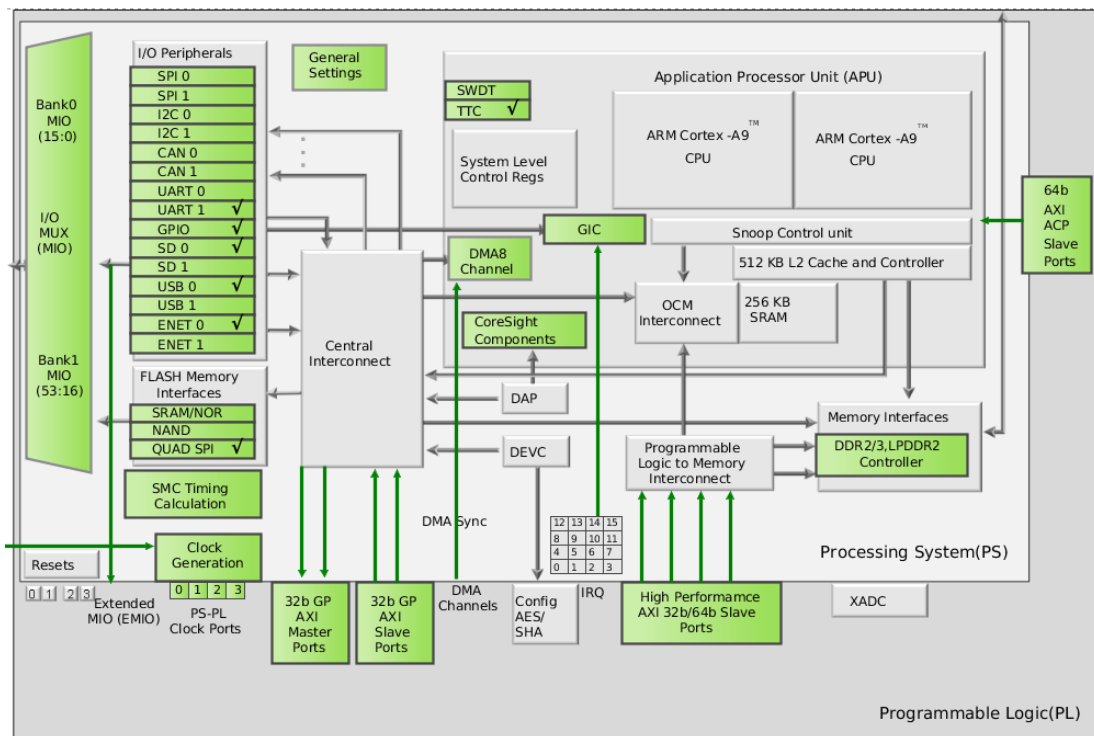


Figure 5.1. Zynq configurable block diagram. Taken from the Vivado suite.

A design of PL peripherals was made in the Vivado IDE, which was introduced in the section 3.3. The first component in a block diagram is a Zynq7 Processing System. This represents all peripherals and CPU's in the PS. In a configuration menu, we see a block diagram of the Zynq (figure 5.1). Here we choose which peripherals are needed for a project. By default, DDR interface, USB and timer are set. These default configurations are for peripherals contained on the Microzed board. Next, the CAN0 and CAN1 is selected. It is possible to route each peripheral from PS to some concrete GPIO pins or via arbitrary PL pins. I used PL pins because it provides more flexibility in designing the PCB. For connection of all custom peripherals, the AXI master interface is enabled. Some of the peripherals generate interrupt signals. The Zynq allows up to 16 interrupt lines from PL to PS shared by both cores. In addition, both CPU cores can be interrupted separately by a fast private interrupt line and interrupt generated by PS peripherals can be routed from PS to the PL. In this project, only standard shared interrupts from PL to PS are enabled. Up to 64 GPIO pins are available for a user in

the PL, we need just several of them. The main driving of PL is done by a clock and reset signal. A clock frequency can range from 0.1 to 250 MHz and several independent clock sources can be used. For our gateway, two clocks are needed. A fast clock for all PL logic such as an AXI bus management and slow clock for driving the parallel bus for FlexRay controllers. Settings of clock sources from PS is done during a booting process by the first stage bootloader (FSBL). The Vivado automatically generates a `ps7_init.c` and `ps7_init.h` for the FSBL. However I couldn't discover how to include these files in the bootloader, called u-boot, so I use just one 100 MHz clock source, which is running by default. The slower clock for FlexRay controllers bus are synthesized in the PL and its frequency is set to 31.25 MHz, which is a value closest to the maximum frequency allowed for the MB88121 chip, easily synthesizable in the PL.

Other components of the block design are:

- Third CAN controller, an SJA1000 IP core
- Finite state machine for control of the parallel bus to MB88121 controllers
- AXI interconnect IP
- Processor system reset IP, driving the reset signal of all peripherals
- Clocking wizard for synthesizing a slower clock
- IPs for concatenating GPIO's and interrupts

The whole block design of the Zynq's PL is attached in the appendix C

5.1 SJA1000 IP Core

We use this open-source Verilog implementation of CAN controller. It is available from the site [18]. The implementation is fully compatible with its hardware counterpart. The SJA1000 core was already tested at the department by Martin Jeřábek in his thesis [12].

Only one difference between the hardware part and soft IP core is its interfacing to the processor. The hardware part uses an 8-bit parallel bus, as we can see in the documentation [13]. The IP core uses memory mapped registers through the AXI bus. The core contains one interrupt signal which needs to be handled by the PS.

Support for SJA1000 chip is in the Linux mainline kernel. It provides several ways of integrating controllers into the system. The direct memory mapping is suitable for our purpose.

The Vivado doesn't generate correct device tree when this IP core is contained in the PL design. The reason may be some missing Vivado specific definitions and settings in the core. A manually written instance in the device tree looks as follows. An example is taken from [12].

```

sja1000_0: sja1000@43c10000 {
    compatible = "nxp,sja1000"; /* driver from the kernel */
    reg = <0x43c10000 0x10000>; /* registers offset and size */
    nxp,external-clock-frequency = <100000000>;
    interrupt-parent = <&intc>;
    interrupts = <0 29 1>;      /* number of interrupt line */
    reg-io-width = <4>;
};

```

We had to set correct drivers for CAN controller and interrupt handling. Next the correct address in the memory, clock frequency and correct interrupt line. I didn't discover how interrupts are numbered. The correct number doesn't match to any pieces

of information I could read from Vivado's and Zynq's documentation. I just guessed the correct number from device trees generated for default components contained in the Vivado, connected to the same interrupt in the block design.

Functions provided by this IP core and its Linux driver are sufficient for our project so I didn't go deeply into implementations and reading datasheets.

5.2 Finite State Machine for communication with MB88121

The main part of work with FPGA was done on an implementation of IP core interconnecting an AXI bus from the PS and a parallel bus from the MB88121 chip. The core consists of two main parts. First, the AXI bus interface providing a bus control described in the section 2.3, the second part is an FSM (Finite State Machine) for read and write operations from the chip. The FSM implements timing diagrams on figures 3.1 and 3.2 described in the section 3.2. The main problems to solve are the correct handling of AXI bus control signals and different clock domain synchronization. The AXI bus is driven by a clock with frequency 100 MHz whereas the chip's bus works at the 31.25 MHz.

The figure 5.2 shows the main signal flows in the IP core. During the write transaction, data and address, where the data should be written, is sent by the PS, read from the AXI write data and address channels and latched in internal registers. Then a write enable signal `mb_wren`, triggering the write transfer to the chip, is generated and sent to the FSM. Latched data (`WDATA`) and address (`WADDR`) are written to the parallel bus. Now the whole IP is waiting until the FSM processes the write transaction. During this time the AXI bus is blocked and waiting for the write response signal. When the writing to the chip is finished, a `write_fin` signal is generated and the write response is sent through the write response channel to the PS.

A read transaction is implemented in a similar way. A read address (`RADDR`) is sent by the PS through the AXI read address channel and latched in the IP core. Then a read-enable (`mb_rden`) signal is generated and read transaction from the MB88121 chip is started. The AXI bus has to wait until data (`RDATA`) are read and a `read_fin` signal is generated. Finally, `RDATA` is sent to the PS through the AXI read data channel.

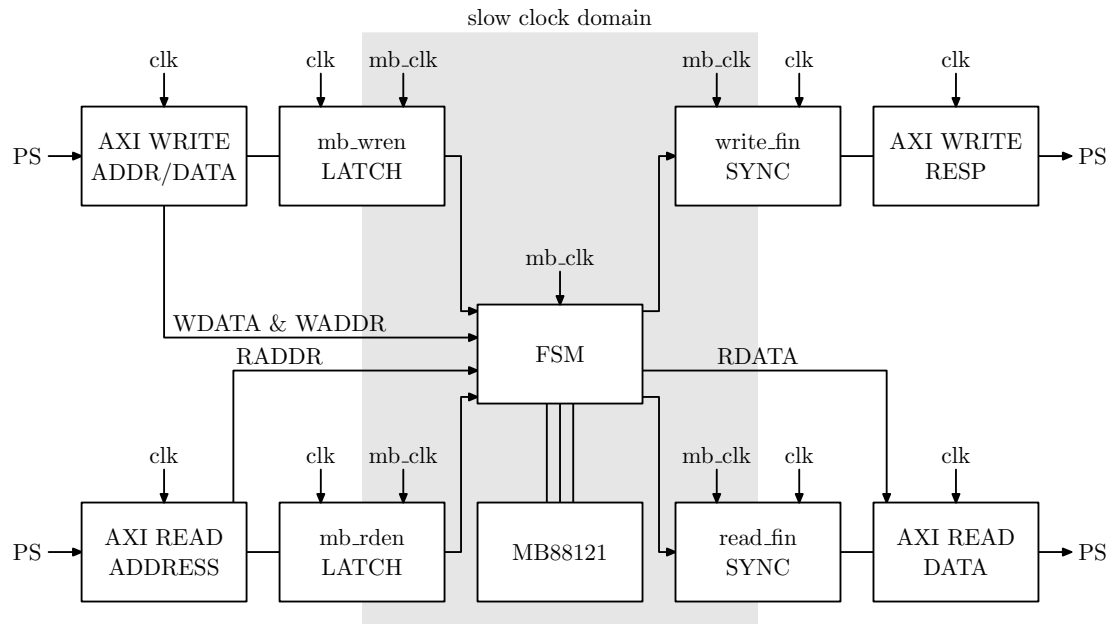


Figure 5.2. Block diagram of the MB88121 communication controller.

The LATCH and SYNC block provides adjusting control signals timing between different clock domains. In our case, this task is relatively simple because of a big difference in clock speeds. The faster clock is used as the main driving signal. When routing a signal from fast clock domain to the slower one, we need to latch a signal value long enough to be sampled by the slower clock.

In the opposite way, when routing a signal from the slower clock domain, a correct sampling by the fast clock has to be done. The clock frequencies are not divisible thus we have to handle this problem as an asynchronous signal synchronization. Generally, the signal change occurrence is important. The easiest way how to detect signal change is by using the `xor` gate and two last sampled values. Then we can catch a rising or falling edge of a slower signal by adding an `and` gate. A resulting implementation in the VHDL can look as follows.

```
process (clk) is
begin
    if rising_edge(clk) then
        delayed_signal <= signal;
    end if;
end process;
sig_rising_edge <= (signal xor delayed_signal) and (signal and '1');
```

A next thing which has to be solved is handling a chip select signal to determine which chip is accessed. I decided to add chip select bit as a next bit on the AXI address bus. We use 12-bit address bus for one chip although only 11 bits are needed for accessing all registers on the MB88121 chip. The reason for this extension is that Linux on the ARM architecture uses 4K large pages, so we align the address range to this size. It should be better for managing the device by the kernel. By the chip select bit, we extended address bus to 13-bit width. A correct mapping to the Linux user-space leads to no additional procedures needed for the chip select control in the PS. The chip select is driven only by selecting to which mapped address space we write to or read from. In the IP core, an 11-bit address and chip select signals are separated from the

AXI read and write address channels. Now we use just two FlexRay controllers. Using this approach, we can easily extend the IP core for driving more slaves.

This IP core doesn't require any settings. Nor special driver nor device tree instance is needed. Now we can directly access registers of the FlexRay controller on addresses defined in the MB88121 datasheet [6].

The last thing still has to be implemented before this gateway can be used in a real application. When an unexpected situation occurs, for example, a broken wire on PCB or badly soldered pin, the FSM can be stuck in some state. Thus the AXI bus keeps blocked and an application will wait forever for finishing the transaction. Such a behavior is not acceptable in safety-critical applications. This situation should be recognized by adding a counter for checking a timeout for the transaction. Then set a bus response signal (S_AXI_BRESP) to indicate that some fault has occurred.

5.3 AXI interconnect IP core

This IP provides an automatically generated interface for connecting more AXI slaves to one master in the PS. All slaves can vary in their parameters. Some components from which the AXI interconnect IP is composed are:

- AXI crossbar - for connecting similar slaves
- AXI Data Width Converter - connect one slave to one master with a different data width
- AXI Clock Converter - connect one slave to one master with different clock frequencies

The Vivado processes all settings of this IP without any user effort and each change in the design is propagated into the IP core instance.

5.4 System reset IP core

This IP generates reset signals. We use only external reset input from the PS but auxiliary reset signal can be used too. All input signals can be configured as active low or active high. A default configuration is an active low. A POR signal is generated by the IP when power-on reset conditions occur. Output reset signals are sequenced such that AXI interconnect IP is reset first, peripherals are reset after 16 clock cycles. With this IP, no additional procedures are needed for correct resetting of PL peripherals. The MB88121 chips resets are also driven by this IP core.

5.5 Clock generating IP core

The slower clock for the control of MB88121 bus is generated in the clocking wizard IP core. Outputs from the IP are a clock signal and a locked signal which indicates that the clock is stable and can be used for peripheral clocking. The clock signal in the MB88121 communication controller is enabled by an `and` gate on the line:

```
clk <= clk_in and locked;
```

5.6 GPIO and interrupt routing

Each GPIO is a tristate buffer. Input-output-tristate triples are routed out from the PS. We use only input direction for connecting bus error detection signals from FlexRay transceivers. So output and the tristate control signal can be left unhandled.

Constant signals are added to the design. One for disabling a standby mode of CAN transceivers. Next two to activate simple error detection mode for FlexRay transceivers and disable SPI status reading mode.

From each FlexRay controller, three interrupt signals are connected to the PS. Although currently, a FlexRay driver implementation doesn't use interrupt routines, it is prepared for a possible future extension.

Chapter 6

Hardware design summary

Previous two chapters described all steps of the hardware design. Before we move towards to software and performance testing, I will sum up what we already have.

We designed a carrier board for the Microzed with all physical interfaces. The gateway is built on the Microzed board. Some of the interface controllers are directly on the Microzed board, some of them are synthesized in the PL or assembled as an external chip on the carrier board. The layout of interface controllers is shown in the diagram 6.1. Namely, the USB, Ethernet and two CAN controllers are parts of the Zynq processing system. One CAN controller is realized as an IP core in the Zynq's programmable logic and two FlexRay controllers are connected as standalone chips.

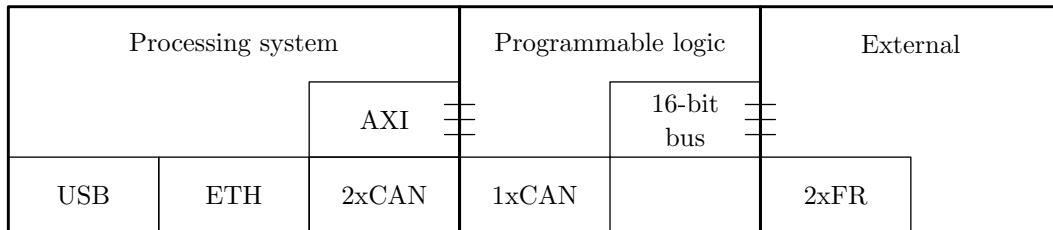


Figure 6.1. Gateway interfaces diagram.

Chapter 7

FlexRay driver porting

There is already available FlexRay driver implementation. I took it from the master thesis of Michal Horn [14]. The driver is written in C language for the FreeRTOS real-time operating system running on the TMS570 processor with integrated FlexRay controller. The FlexRay controller's functions and registers are the same in the TMS570 like in the MB88121 chip.

Structure of the driver has several layers. First, definitions of basic structures and register offsets, then low-level procedures for accessing and configuring peripheral registers. Above the main low-level functions, interface functions which check return values and provide thread safety, are implemented. Last, user-friendly functions are implemented in the main applications, varying in their purposes.

Thread safety is assured by the use of a mutex. An interface for working with mutexes in FreeRTOS is similar like that provided by the `pthread.h` library available in Linux.

Next thing, which I needed to add to the code, was mapping the peripheral's physical address to the virtual one, inside the user-space process. I took the function `map_phys_address` implemented by Pavel Píša for work with the MZ_APO education kit. The function first opens the `/dev/mem` file, which provides access to the physical memory. Then the memory is mapped by the `mmap` function. A pointer to the memory is returned. All memory blocks are aligned to the page size used by the operating system.

After these changes, I was able to successfully run one FlexRay node on the gateway. For controlling both FlexRay interfaces by the driver, I had to remove all global variable definitions. To do that, I rewrote some configuration structure definitions. Now they store also the base address of peripheral and other things which were hard-coded in the previous implementation. All functions were rewritten to use a pointer to node-specific configuration structure instead of global variables. Now the driver is more portable but it leads to little less readable code because of more function parameters.

The driver provides a development error detection mode. A lot of checking functions and driver status monitoring is performed to make the development easier, but it slows down the application. We can set this mode on by the definition `#define DET_ACTIVATED` in a drivers header file.

Several things are still missing in the driver implementation after my changes. First, an important issue is the lack of dynamic segment control. The second thing is a missing transceiver's status monitoring.

Chapter 8

Experiments and testing

After assembling the hardware, configuring the system and adjusting the driver, we have to test if everything works correctly. First, we deal with basic CAN and FlexRay testing, then we move on to FlexRay performance tests.

8.1 Basic CAN interfaces test

To prove that all CAN interfaces on the gateway are working, first, we check that all CAN interfaces are recognized by the system. After executing the command `ip a` in a command line, the following lines should be listed.

```
# ip a
...
2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
   link/can
3: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
   link/can
4: can2: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
   link/can
...
```

This indicates that both internal CAN controllers and one in the PL can be accessed as network interfaces.

To check whether the interface is really working, we have to configure it and set it up. It is done by the commands:

```
# link set can0 type can bitrate 1000000
# ip link set can0 up
```

The `set can0` parameter determines selected interface, and `bitrate 1000000` sets the communication speed. Then we can send some message manually by the command `cansend` or read messages by the command `candump`.

An external USB to CAN interface, connected to the PC, was used for checking the communication.

8.2 Basic FlexRay interfaces test

For FlexRay testing, RPP boards and the steer-by-wire demo is used. Same FlexRay nodes, as are on the sensor and actuator RPP board, are implemented also on the gateway. Then the bus between RPP boards is split into two separate FlexRay buses, interconnected by the gateway. A schema of this experiment is in the figure 8.1. Starting of communication and receiving and sending of FlexRay frames can be tested on this arrangement. The simplest test contains just receiving a shaft position, from one RPP board, then send it to the second board. If the steer-by-wire demo works with a bus cut by the gateway, we can say that FlexRay controllers and drivers are working.

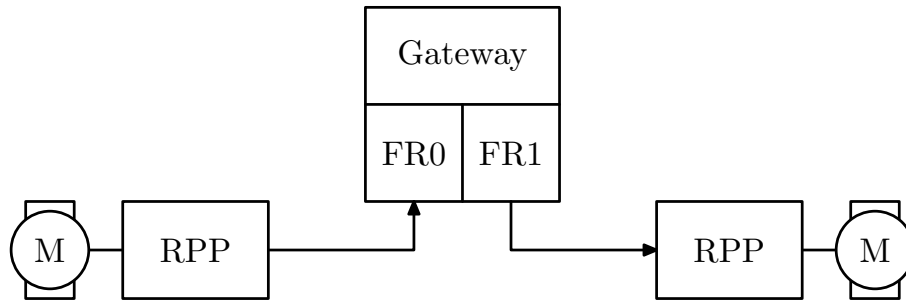


Figure 8.1. Arrangement for basic FlexRay testing.

8.3 Round-trip time measurement

A next thing, we should discover, is how the gateway affects a data transmission latency. To do that, I measured the round-trip time in different configurations. The measurement is programmed on the one RPP board. Two tasks with high priority do this job. In the first task, a timer is started and a unique message is sent to the FlexRay cluster. The second task waits until the same message is received from the cluster, then stops the timer and compute delay between sending and receiving the message. Afterward, the delay is stored in a histogram and a semaphore is released to start a new measurement. A route, which the message go through before it is sent back, vary between experiment configurations. All configurations are variants of the arrangement from the figure 8.1.

In all experiments, the FlexRay bus is configured to work with 1 ms long cycles. The timer used for measurement is the main FreeRTOS timer configured also to period 1 ms. Thus, the time measurement is not so accurate, mainly when we measure short times. But in real-time applications, we need to know the maximum delay. If we round it up, nothing happens. An information about the shortest delay is not so interesting for us.

8.3.1 Measurement results

First, as a reference measurement, a round-trip from one RPP board (here RPP1) to the second (RPP2) and back is measured. Because RPP boards run just several simple FreeRTOS tasks, I expect the shortest possible round-trip time. It turned out to be the truth. From the table 8.1, we see that all messages returned at latest at 3 ms. During all measurements in this section, 100000 messages were sent.

Delay [ms]	1	2	3	4	5
Count [%]	0	88.5	11.5	0	0
Delay [ms]	6	10	14	18	> 18
Count [%]	0	0	0	0	0

Table 8.1. Round-trip time histogram: RPP1 → RPP2 → RPP1.

The second measurement, in the table 8.2, shows a round-trip from the RPP board to the gateway and back. Here the maximum delay of the returned message is 4 ms. Although, the Zynq processor on the gateway is much faster than that on the RPP board. The Linux on the gateway has much more overheads during the user application execution than the FreeRTOS on the RPP board.

Delay [ms]	1	2	3	4	5
Count [%]	0	85.9	13.1	1.0	0
Delay [ms]	6	10	14	18	> 18
Count [%]	0	0	0	0	0

Table 8.2. Round-trip time histogram: RPP1 → GW → RPP1.

Next experiment shows a possible problem, which can occur on the gateway. The experiment configuration is the same as during the previous measurement. Messages are sent only between the RPP board and the gateway. The only difference is that the second FlexRay interface on the gateway cannot properly initialize a communication. The bus cable is unplugged. From the table 8.3 we see that a real-time behavior is broken. Here maximum measured delays are hundreds of milliseconds. The reason is obvious. Both FlexRay interfaces are independent but they have common device driver. While the driver is trying to initialize one interface, it blocks the communication with the second interface. The initialization process is relatively time-consuming, especially when the interface cannot be initialized due to some faults. But this case can be handled in the application by the monitoring of FlexRay status.

Delay [ms]	1	2	3	4	5
Count [%]	0	75.8	22.5	0.3	0
Delay [ms]	6	10	14	18	> 18
Count [%]	0	0	0	0	1.4

Table 8.3. Round-trip time histogram: RPP1 → GW → RPP1, when the problem occurs on the second FlexRay interface on the gateway.

In previous experiments, we tested and compared a response of the FlexRay interface on the RPP board and on the gateway. Now we will test an effect of the gateway when connecting two distinct FlexRay buses. The experiment arrangement is exactly the same as in the figure 8.1. The message is sent from the RPP1 to the RPP2 through the gateway, then returned the same way back to the RPP1. The table 8.4 shows us that now the maximum delay is 6 ms. That's twice longer than sending messages only between two RPP boards. But still, it can be seen as a good result.

Delay [ms]	1	2	3	4	5
Count [%]	0	0	8.6	48.2	40.2
Delay [ms]	6	10	14	18	> 18
Count [%]	3.0	0	0	0	0

Table 8.4. Round-trip time histogram: RPP1 → GW → RPP2 → GW → RPP1.

But when there is a big load on the gateway, the response is much worse. We put a more traffic on FlexRay buses and a load is generated on the gateways processor. I run the command

```
while true; do echo; done
```

during the measurement. Now the maximum delay is increased to the value 18 ms, as shown in the table 8.5. This is not a good behavior.

Delay [ms]	1	2	3	4	5
Count [%]	0	0	7.8	49.2	39.0
Delay [ms]	6	10	14	18	> 18
Count [%]	3.0	0.4	0.4	0.2	0

Table 8.5. Round-trip time histogram: RPP1 → GW → RPP2 → GW → RPP1, when gateway is busy.

We can fix this problem by changing the default configuration of the Linux scheduler. A library `sched.h` provides the function `sched_setscheduler()`. We can select several real-time scheduler policies. In this experiment, the FIFO policy is selected and a priority of message forwarding process is set to the highest possible. Then the round-trip time measurement, during a high load on the gateway, is repeated. From the table 8.6, we see that now the delay is acceptable.

Delay [ms]	1	2	3	4	5
Count [%]	0	0	2.6	62.3	31.5
Delay [ms]	6	10	14	18	> 18
Count [%]	3.5	0.1	0	0	0

Table 8.6. Round-trip time histogram: RPP1 → GW → RPP2 → GW → RPP1, when gateway is busy, with increased priority of FlexRay management process.

The last experiment measures the influence of the development mode of the FlexRay driver. Everything is the same like in the previous experiment, only the definition `#define DET_ACTIVATED` is removed from the driver header file. From the table 8.7, we see, that changes are not so significant.

Delay [ms]	1	2	3	4	5
Count [%]	0	0	2.5	62.1	31.9
Delay [ms]	6	10	14	18	> 18
Count [%]	3.5	0	0	0	0

Table 8.7. Round-trip time histogram: RPP1 → GW → RPP2 → GW → RPP1, when gateway is busy, with increased priority of FlexRay management process and deactivated the development mode.

8.4 Precise measurement of a delay on the FlexRay gateway

Previous measurements give us a general idea about what to expect when using the FlexRay part of the gateway. A more accurate measurement of the delay of FlexRay messages going through the gateway can be measured by the Vector VN3600 device and the CANoe software.

The measurement setup is shown in the figure 8.2. A general setup is the same as in previous experiments. We have two separate FlexRay clusters connected by the gateway. In addition, the VN3600 device is connected to each cluster. Because it has only one dual channel FlexRay interface, we use only one channel for monitoring of each cluster. A traffic is the same on both channels, so we don't lose any information.

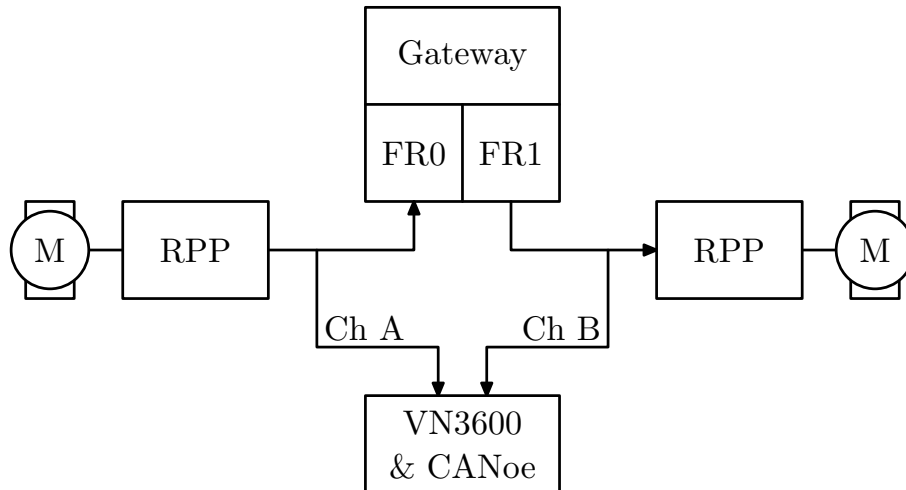


Figure 8.2. Arrangement for measurement a delay on the gateway.

In the CANoe software, we can detect each message in both clusters and measure the time of reception. The accuracy of measurements is about several microseconds. One message being forwarded by the gateway is monitored. In the CANoe, the message is determined by its slot ID in the static segment. Then the time of reception in both clusters is logged. Afterward, logged messages are paired according to their unique content and the delay is computed. The measurement was run on the gateway with high load and with scheduler set to real-time function. We use the same settings like in the last experiment in the previous section 8.3

A histogram of delays of about 100000 messages is in the figure 8.3. We can see that the maximum delay, caused by one passage through the gateway is less than 2.2 ms.

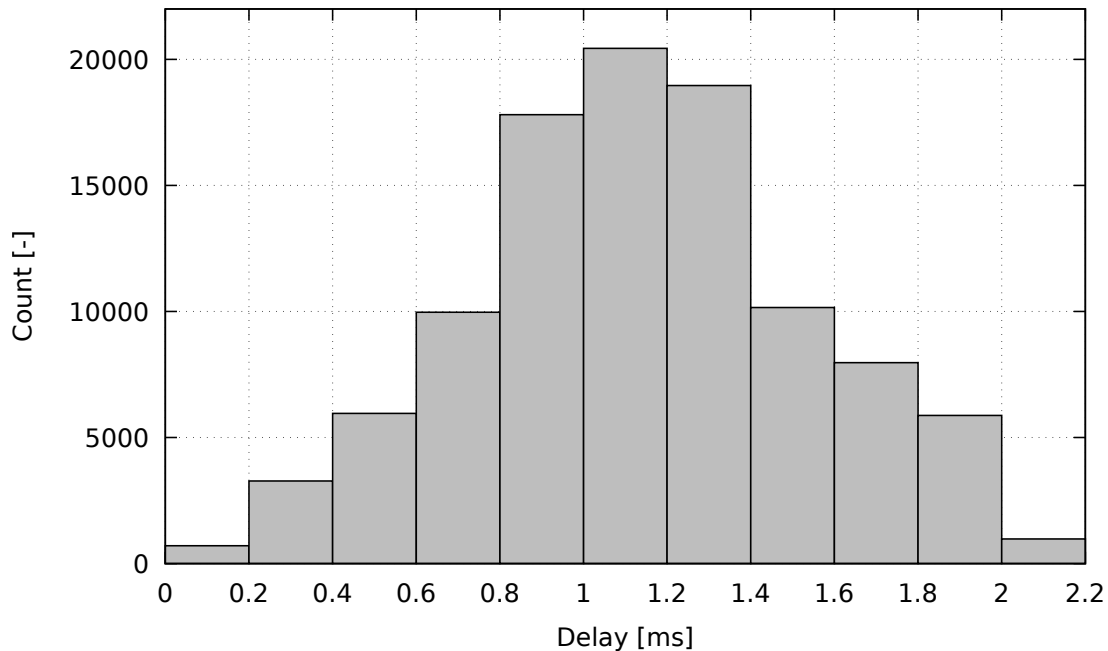


Figure 8.3. Histogram of message delays on the gateway.

If we think a little about the histogram, we can expect, that delays should be multiples of the communication cycle length. It is 1 ms in this case. But both clusters are

independent and not synchronized together. A little difference between communication cycle lengths may appear. Thus, during a long time of measurement, delays are linearly changing. This issue is shown in the figure 8.4. Here, several thousands of measured delays are plotted in the graph during a measurement time.

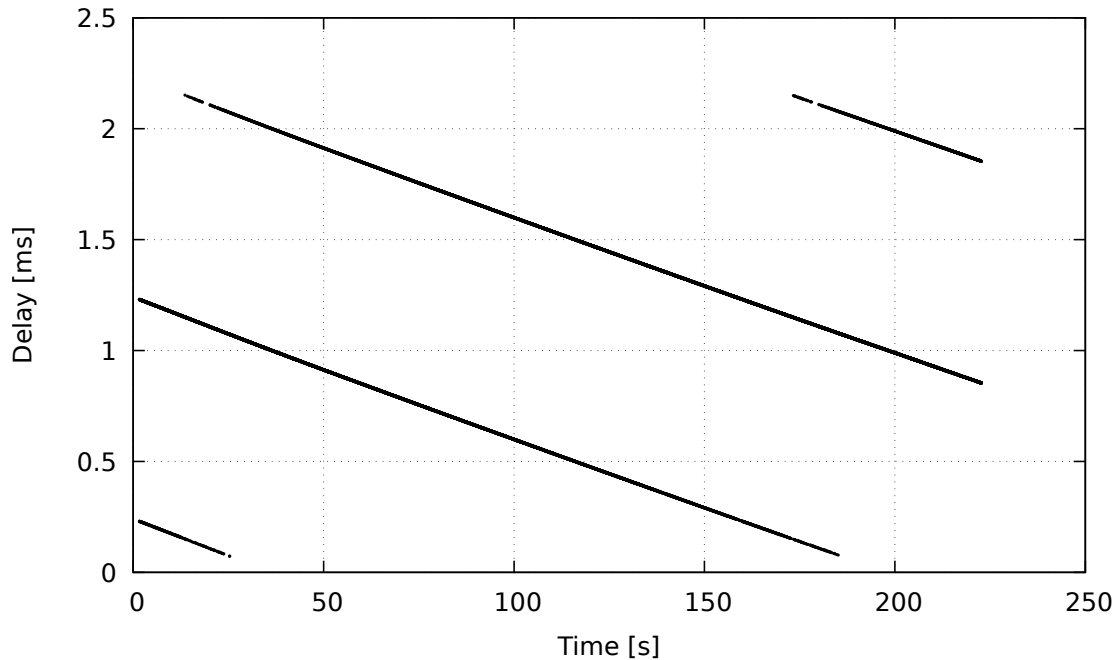


Figure 8.4. Time development of delays on the gateway.

Based on the graph 8.4, we can say that delays would be multiples of cycle period only with synchronized clusters. Next, we see that the message is sent at the latest in the third cycle after it is received. If clusters would be synchronized, in the second cycle. In the ideal case, all messages would be sent in the first following cycle after receiving. If we use a message cycle longer than 2.2 ms, this ideal case is already achieved.

8.5 Power consumption

Although this gateway is not expecting to be a big power load in the car system, to finish the measurement and testing, the power consumption is measured. The table 8.8 shows the measurement results. In all cases, the gateway is supplied by 12 V power supply.

State	Current [mA]	Power [W]
Startup	260	3.12
Standby	210	2.52
Running	220	2.64

Table 8.8. Gateway power consumption

The current was measured during several possible working states. As can be expected, the maximal power consumption is during the booting and startup process. A difference between standby mode and during processing of some high load application is not so significant. This is caused by disabling of all power saving functions of the Linux kernel, which could affect the real-time behavior.

Chapter 9

Future work

At this point, we have a working gateway. In the near future, we would like to connect this gateway to the real car. Before an experimenting with self-driving algorithms can start, we have to create a new bug-free carrier board based on materials in appendixes of this thesis. Next, the communication interface between FlexRay controller and AXI bus needs to be extended with bus response signals. These are missing in the current implementation, that can lead to unsafe behavior of the device. This issue was already discussed in the section 5.2.

From the point of view of CAN interfaces, everything should work just now. But in the FlexRay driver implementation, a control of communication over the dynamic segment is still missing. Although it is not a mandatory part of the FlexRay communication, we expect, that this segment will be present in the car. Our FlexRay driver offers much more opportunities for the future work. The goal could be to provide the Berkeley socket interface for standardized networking. Also, the full driver implementation could be provided for other users as a part of the Linux.

Chapter 10

Conclusion

In this thesis, the control unit for use as the gateway between automotive buses was designed, assembled and tested. Namely, the unit contains three CAN interfaces, two dual-channel FlexRays and an ethernet and a USB in addition. Next, I ported the already available FlexRay driver implementation to the Linux machine.

With the end of the work, the gateway is ready to start experimenting with the real car. Although, some work needs to be done before it could be safely used for self-driving algorithm testing.

The main advantage of the developed device is the number of available network interfaces. This should fulfill our needs with a reserve. In the case that we would need to use more interfaces, the power of the programmable logic allows us to simply implement new ones by the copy-paste method. Only the carrier board would have to be redesigned.

On this device, no closed source or commercial software is running. We have a source code available for all parts of the gateway, that means an easier debugging of future applications. This is the main advantage over other alternative devices, available on the market.



References

- [1] *FlexRay Communications System Protocol Specification Version 3.0.1*. FlexRay Consortium, 2010.
- [2] *AMBA AXI Protocol Specification* ARM, 2004.
- [3] Rich Griffin. *Designing a Custom AXI-lite Slave Peripheral* SILICA, 2014.
- [4] *Zynq-7000 All Programmable SoC Technical Reference Manual* Xilinx, 2017.
- [5] *MicroZed Carrier Design Guide* Avnet, 2017.
- [6] *FlexRay ASSP MB88121 Data Sheet* Cypress, 2013.
- [7] *Vivado Design Suite Tcl Command Reference Guide* Xilinx, 2013.
- [8] *XST User Guide* Xilinx, 2008.
- [9] Barry R. *Mastering the FreeRTOS Real Time Kernel, A Hands-On Tutorial Guide* Real Time Engineers Ltd., 2016.
- [10] *MicroZed Hardware User Guide* Avnet, 2014.
- [11] *TJA1082 Data Sheet* NXP, 2012.
- [12] Jeřábek M. *FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation*, Bachelor's Thesis CTU, 2016.
- [13] *SJA1000 Stand-alone CAN controller, Application note* Philips Semiconductors, 1997.
- [14] Horn M. *Software obsluhující periferie a FlexRay na automobilové řídicí jednotce*, diplomová práce CTU, 2013.
- [15] Blecha J. *Programovatelná testovací platforma, diplomová práce* CTU, 2014.
- [16] *Programming With CAPL* Vector, 2004.
- [17] *Build Device Tree Blob* <http://www.wiki.xilinx.com/Build+Device+Tree+Blob>, 2018.
- [18] Mohor, I. *SJA1000-compatible CAN Protocol Controller IP Core* <http://opencores.org/project,can,overview>, 2018.
- [19] *Rapid Prototyping platform* <https://rttime.felk.cvut.cz/rpp-tms570/>, 2018.
- [20] *Vector cable connections* https://vector.com/vi_cable_en.html, 2018.
- [21] *Comparing real-time scheduling on the Linux kernel and an RTOS* <https://www.embedded.com/design/operating-systems/4371651/Comparing-the-real-time-scheduling-policies-of-the-Linux-kernel-and-an-RTOS->, 2018.
- [22] *CourseWare Wiki, B35APO* https://cw.fel.cvut.cz/wiki/courses/b35apo/documentation/mz_apo/start, 2018.



Appendix A

Shortcuts

AXI	Advanced extensible interface
CAN	Controller area network
ECU	Electronic control unit
FPGA	Field-programmable gate array
FSBL	First stage bootloader
FSM	Finite state machine
HDL	Hardware description language
HLS	High level synthesis
IP	Intellectual property
QSPI	Quad serial peripheral interface
PCB	Plane connections board
PL	Programmable logic
POR	Power-on reset
PS	Processing system
SPI	Serial peripheral interface
TCL	Tool command language



Appendix B
Schematics

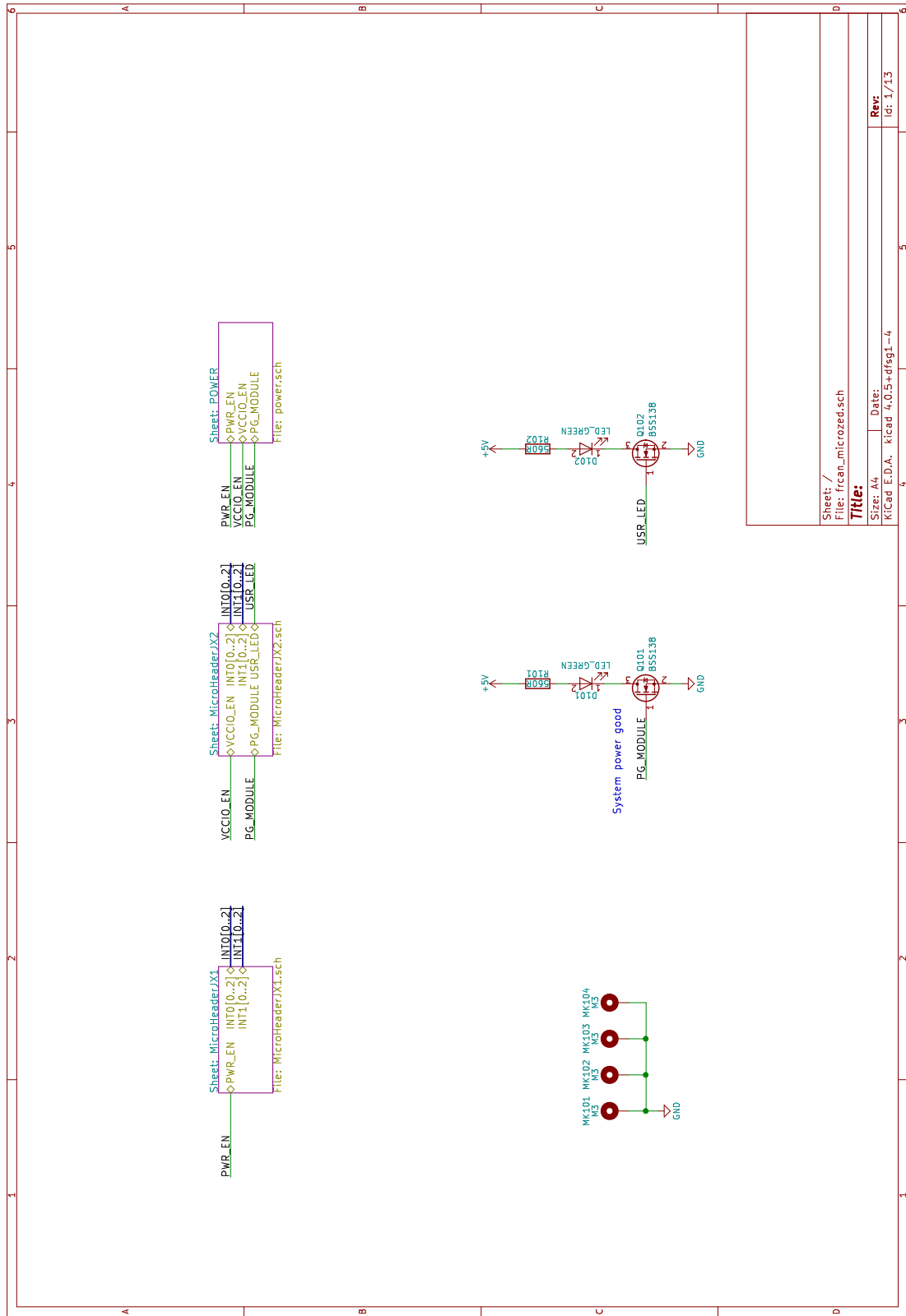
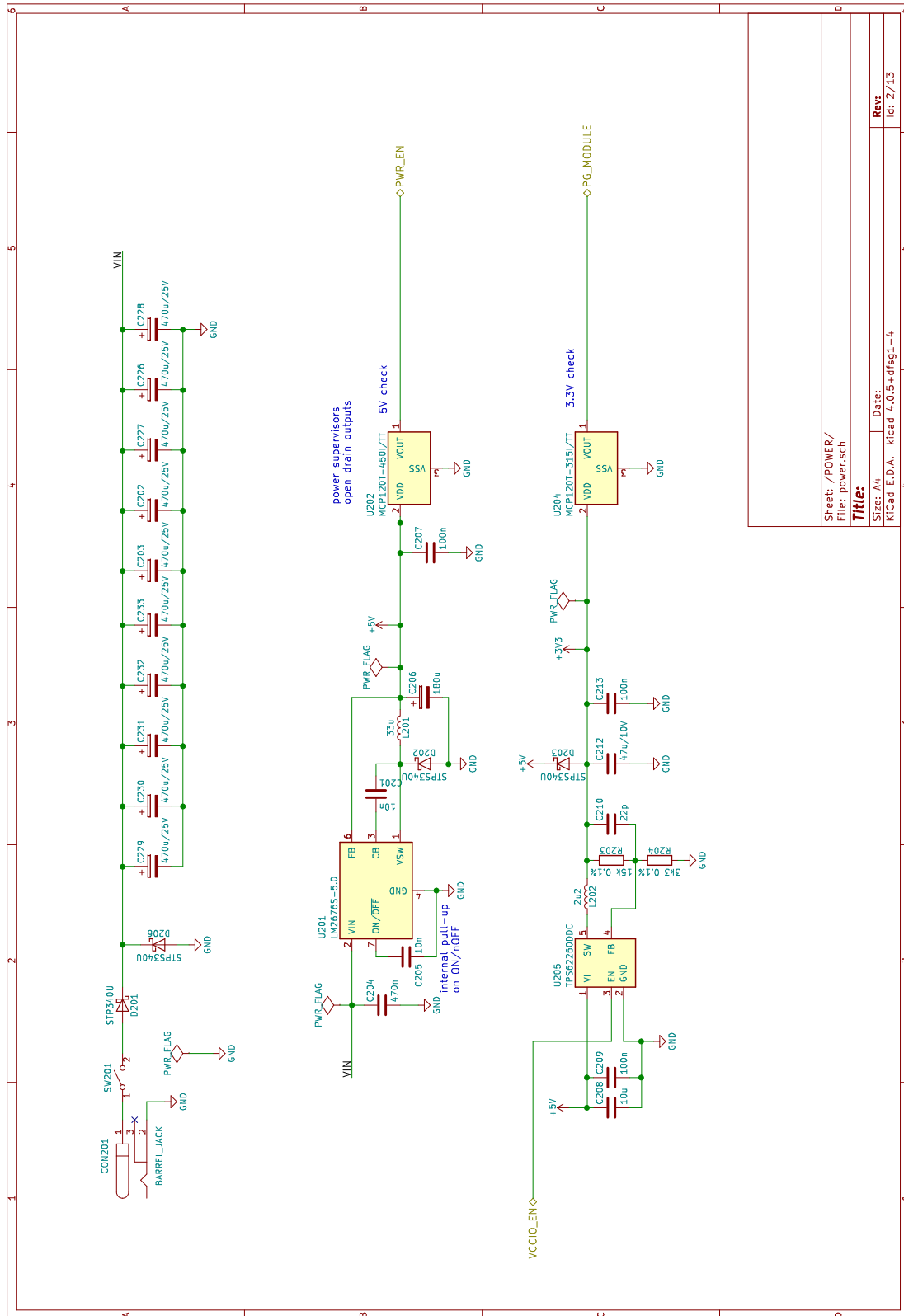
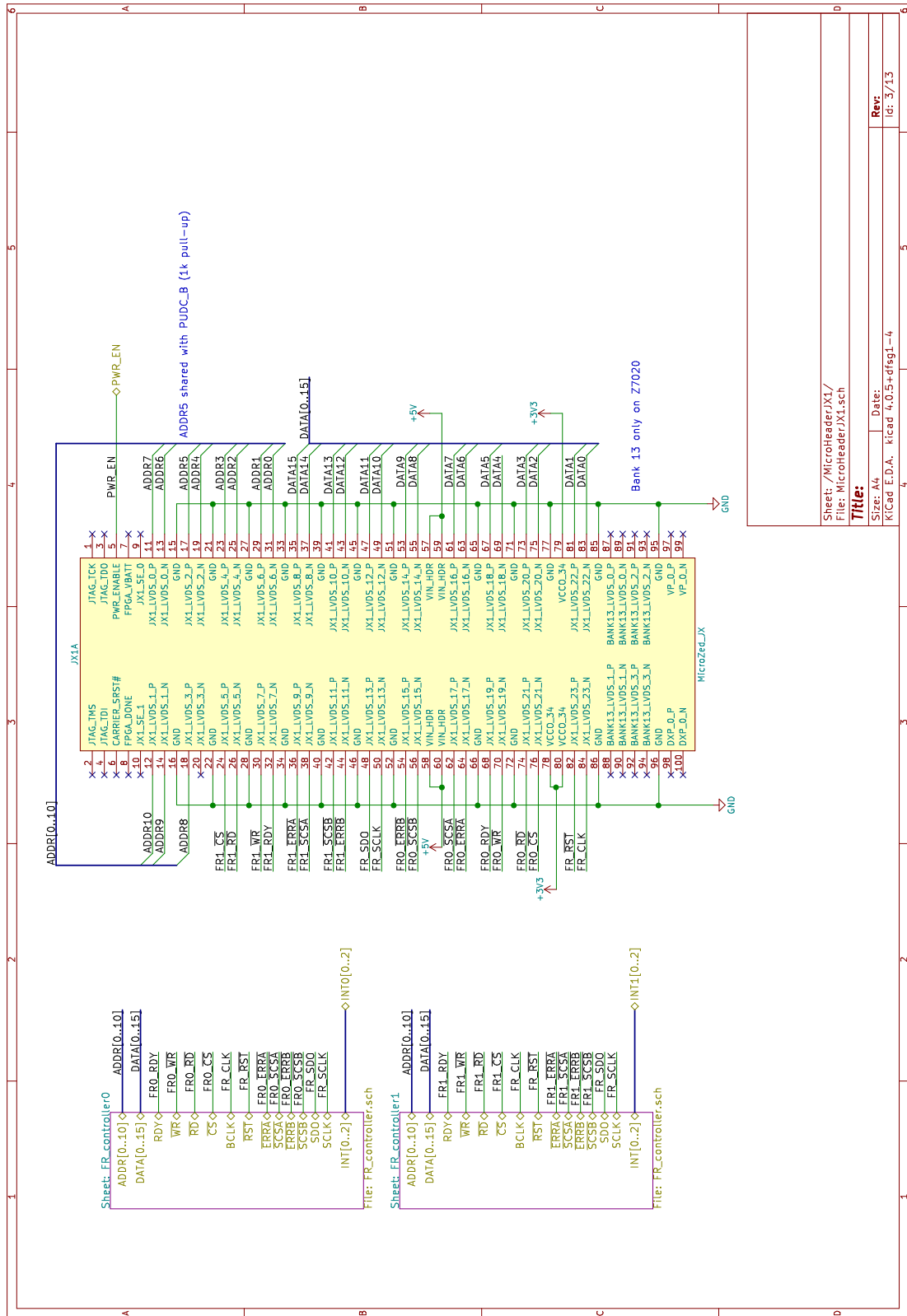


Figure B.1. Root sheet



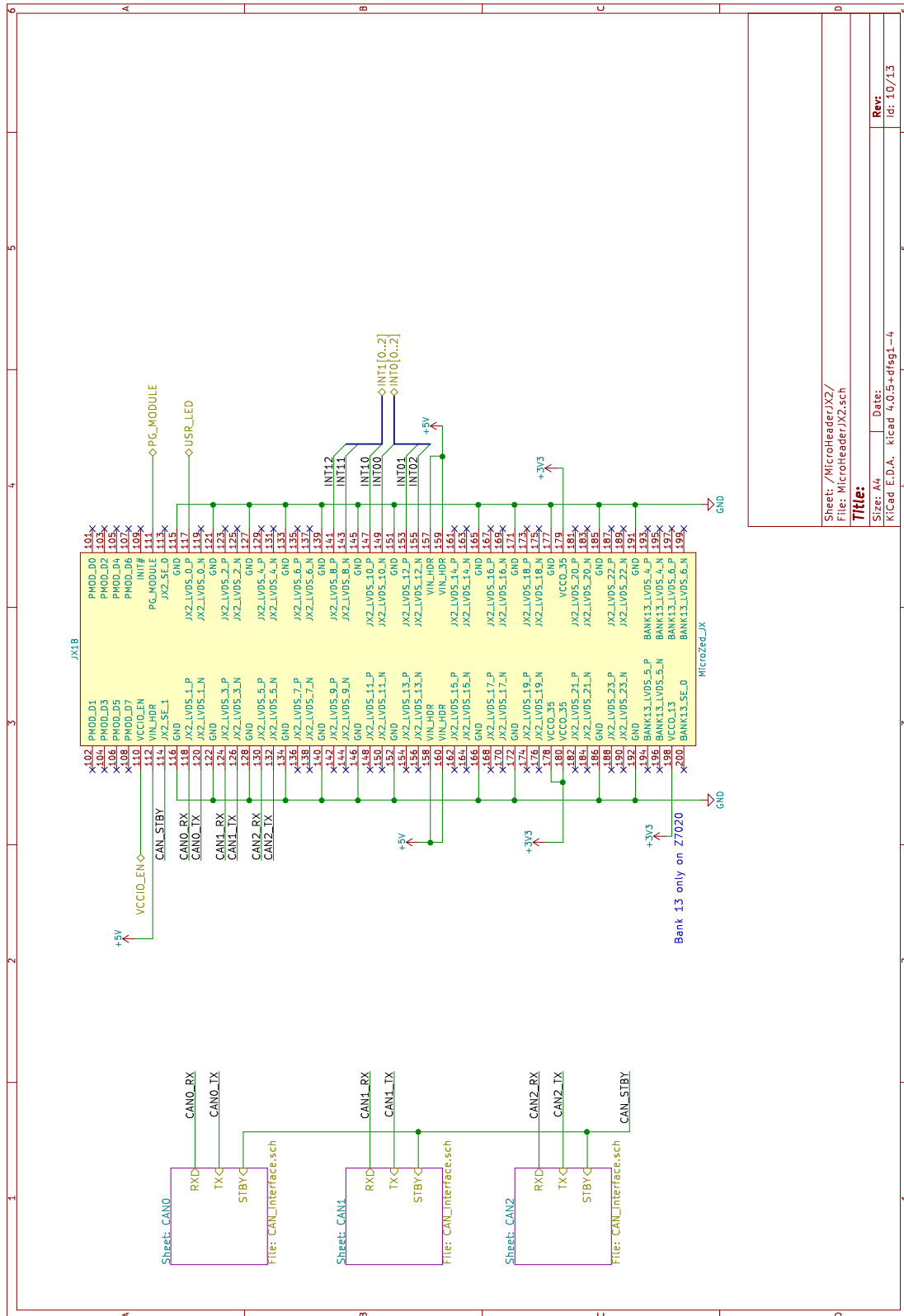
Sheet: /POWER/
File: power.sch
Title:
Size: A4
Date:
KiCad E.D.A. kicad_4.0.5+dfsg1-4
Rev:
Id: 2/13

Figure B.2. Power supply



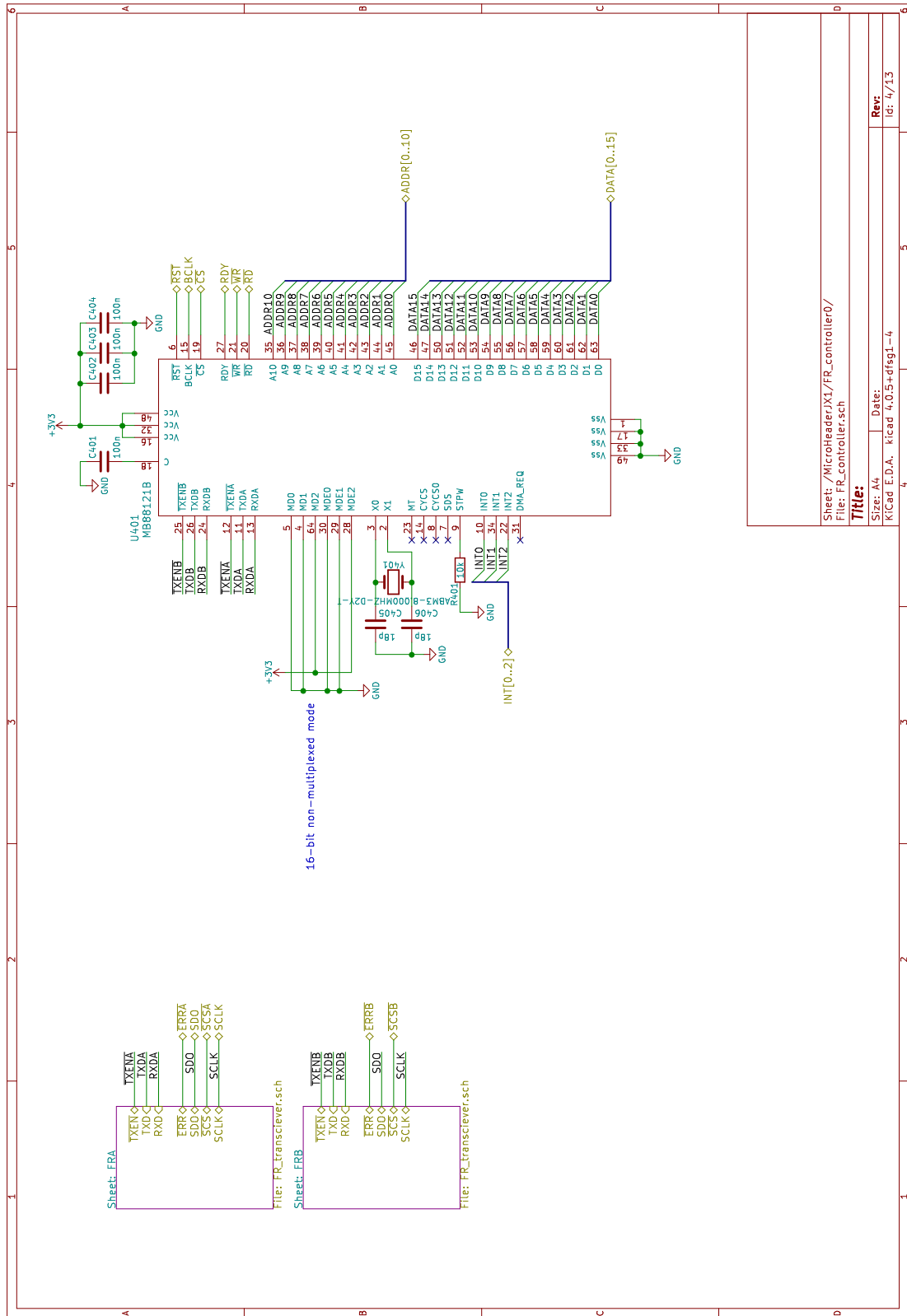
Sheet: /MicroHeader/JX1/
 File: MicroHeader/JX1.sch
Title:
 Size: A4 Date:
 Kicad E.D.A. Kicad 4.0.5+dfsg1-4
 Rev: Id: 3/13

Figure B.3. Connector JX1



Sheet: /MicroHeader/JX2/
 File: MicroHeader/JX2.sch
Title:
 Size: A4 Date:
 Kicad E.D.A. Kicad 4.0.5+dfsg1-4
 Rev: 10/13

Figure B.4. Connector JX2



Sheet: /MicroHeader/1/FR_controller/ File: FR_controller.sch
Title:
 Size: A4 Date:
 Kicad E.D.A. Kicad 4.0.5+dfsg1-4
 Rev: Id: 4/13

Figure B.5. FlexRay controller

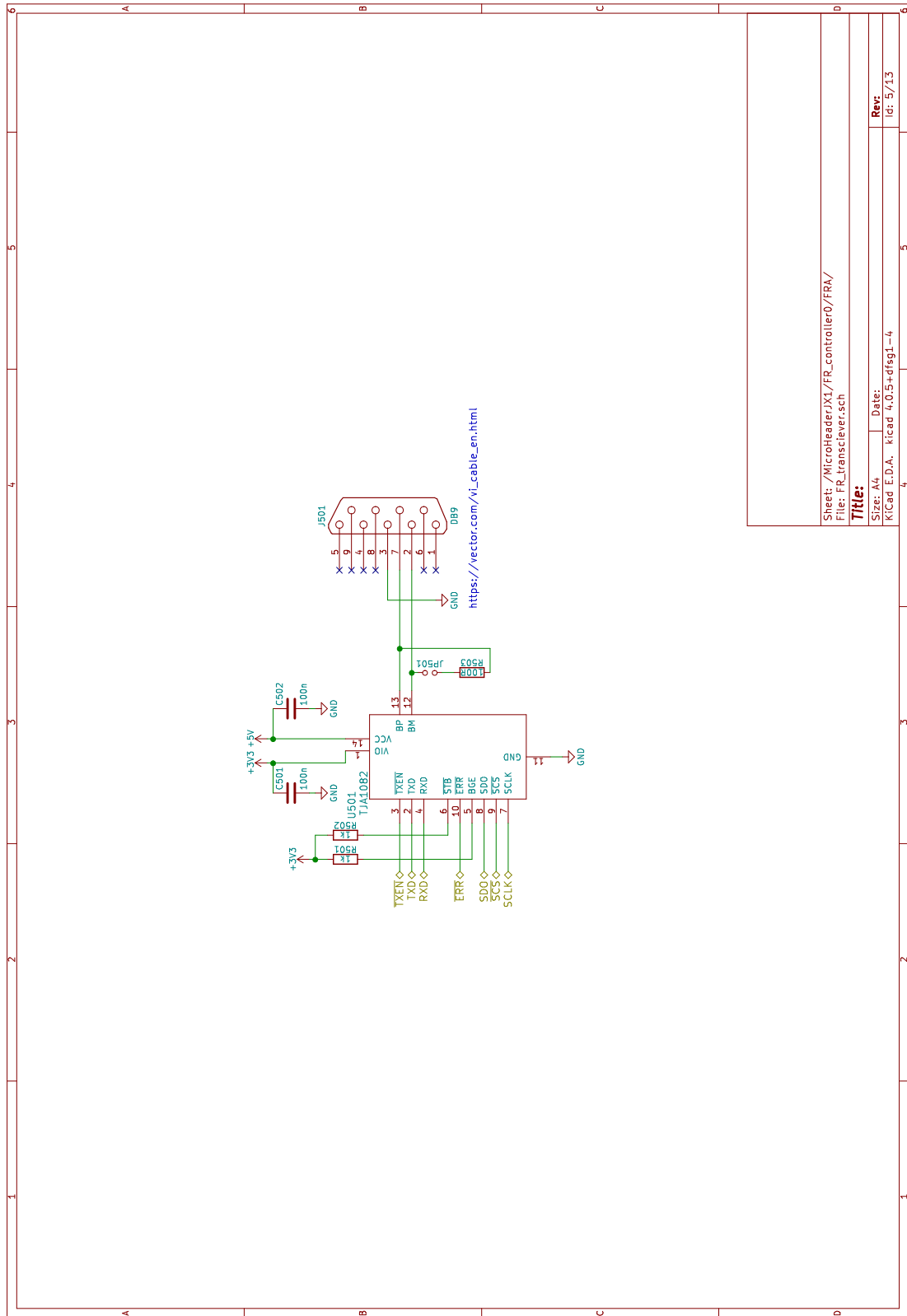


Figure B.6. FlexRay transceiver

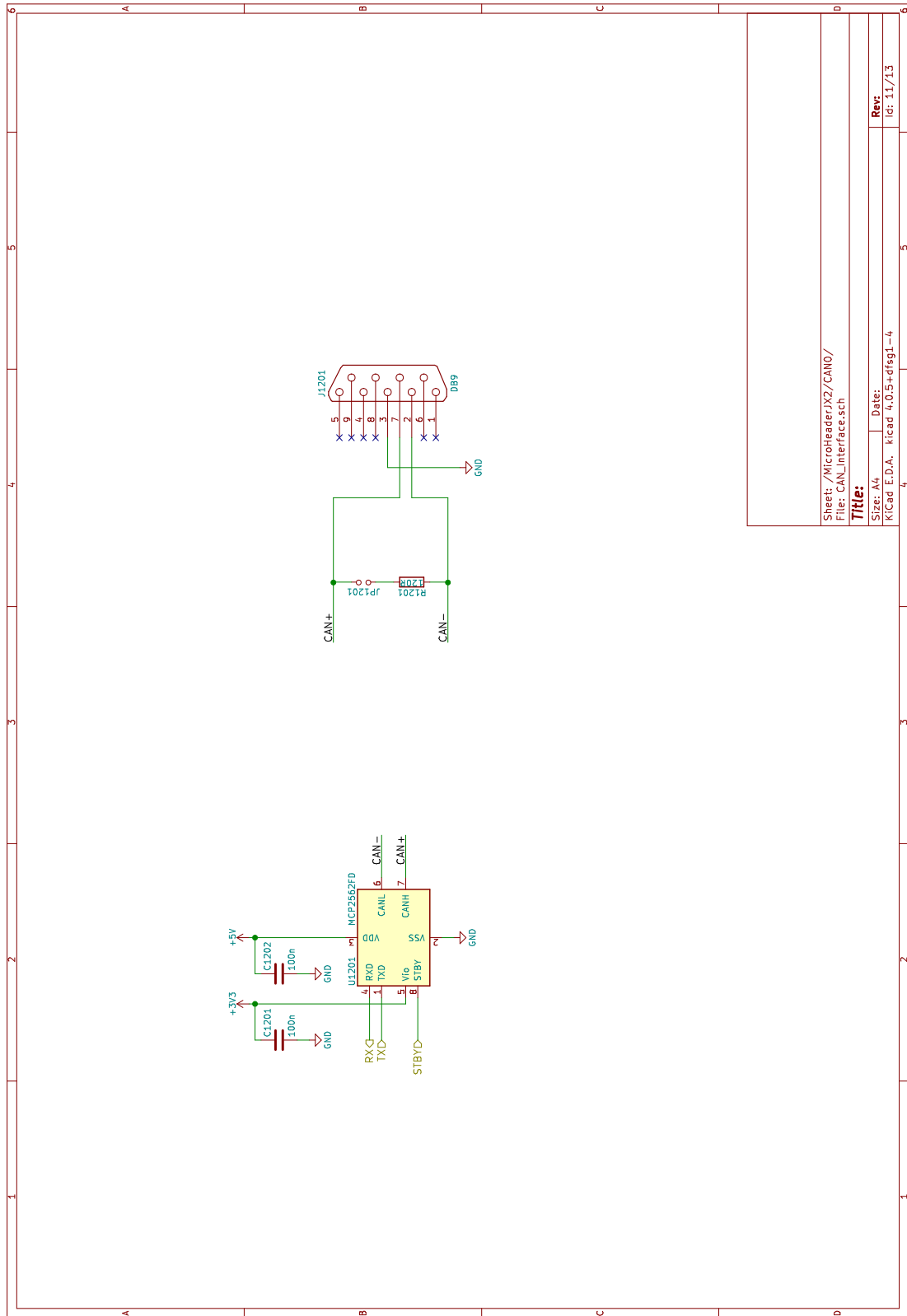
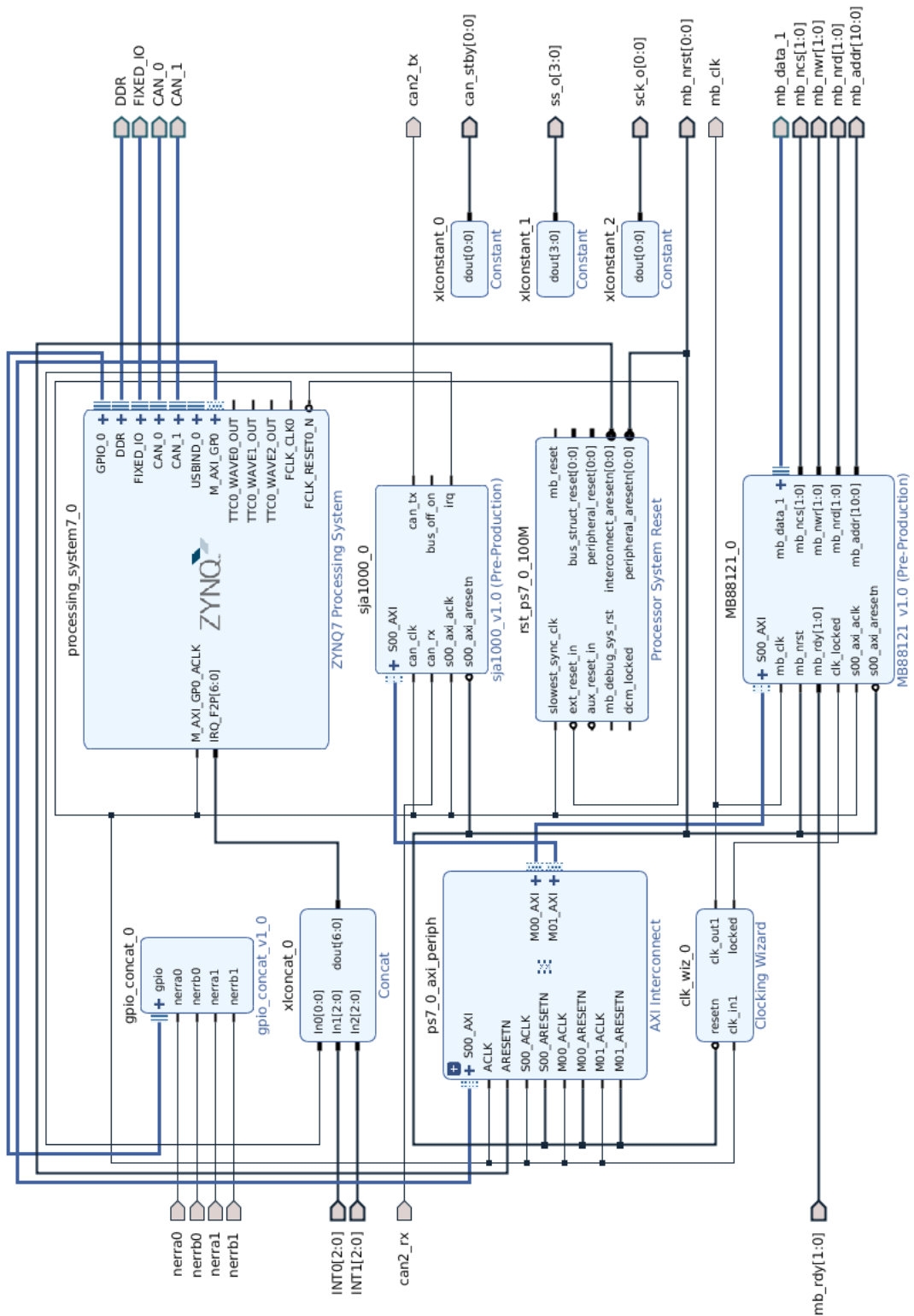


Figure B.7. CAN transceiver

Appendix C

PL block diagram





Appendix D

Obsah přiloženého CD

text/	This thesis in the pdf format.
pcb/	Kicad project with schematics and PCB design of the device.
datasheets/	Documentation of main used electronic components.
vivado/	Vivado project and IP repository.
linux.zynq.zip	Linux image for the device.

Appendix E

Assignment of this thesis



MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	Záhora Jiří	Personal ID number:	409015
Faculty / Institute:	Faculty of Electrical Engineering		
Department / Institute:	Department of Control Engineering		
Study program:	Cybernetics and Robotics		
Branch of study:	Cybernetics and Robotics		

II. Master's thesis details

Master's thesis title in English:
Automotive Control Unit with CAN, FlexRay and Ethernet Interfaces

Master's thesis title in Czech:
Automobilová řídicí jednotka s rozhraními CAN, FlexRay a Ethernet

Guidelines:

1. Familiarize yourself with the MicroZed Zynq development kit and development environment Xilinx Vivado. See also the MZ_APO project, developed at the department, which uses a programmable logic array on Zynq.
2. Select suitable components and make a PCB for a gateway containing these interfaces: 3x CAN, 2x dual channel FlexRay and one Ethernet. Gateway should be based on the MicroZed kit. Use the Cypress MB88121 FlexRay controllers.
3. In a programmable logic array implement an interface for connecting the external FlexRay controller to the Zynq chip.
4. Port an existing driver for the FlexRay controller based on the Bosch E-Ray core to the Zynq platform and the MB88121 chip. Test the result.
5. Document the results and justify decisions taken during the development.


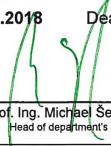
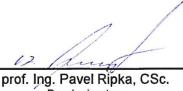
Bibliography / sources:

- [1] Zynq-7000 All Programmable SoC Technical Reference Manual (https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf)
- [2] "Software obsluhující periferie a FlexRay na automobilové řídicí jednotce", Michal Horn, Diplomová práce, FEL ČVUT, 2013
- [3] http://rtime.felk.cvut.cz/gitweb/fpga/zynq/canbench-sw.git/shortlog/refs/heads/microzed_apo

Name and workplace of master's thesis supervisor:
Ing. Martin Vajnar, CIIRC, ČVUT v Praze

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **16.01.2018** Deadline for master's thesis submission: **25.05.2018**
Assignment valid until: **30.09.2019**

 Ing. Martin Vajnar Supervisor's signature	 prof. Ing. Michael Šebek, DrSc. Head of department's signature	 prof. Ing. Pavel Řípka, CSc. Dean's signature
---	--	---

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

_____	_____
Date of assignment receipt	Student's signature