Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation

Martin Jeřábek

Supervisor: Ing. Pavel Píša, Ph.D.

Study Programme: Open Informatics, Bachelor

Field of Study: Computer and Information Science

May 27, 2016

Aknowledgements

I would like to thank my supervisor, Ing. Pavel Píša, Ph.D., for providing invaluable consultations and offering detailed analyses of many peculiar problems; the author of the original testing system, Ing. Michal Sojka, Ph.D., for introduction to the whole system, providing support, and his valuable advice. I would also like to thank Ing. Petr Porazil, who helped me with assembling the boards. Lastly, I must not forget to thank my family for their support throughout my whole study.

Declaration

I declare that the presented work was developed independently and I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, on May 27, 2016 ....

Abstract

This thesis describes design and development of software and hardware system for precise measuring of processing latency of various CAN bus gateway implementations, with sub-microsecond precision. The intended application is evaluation of the software gateway implemented in the Linux kernel. This work extends the previous software-only PC-based tester. Xilinx Zynq SoC with integrated FPGA is used as the new development platform, which brings a significant increase in measurement precision. Latency measurements obtained with this system are compared to those from the previous software-based solution. Experimental results show 60 % improvement in measurement stability. The developed system is deployed in as a complete setup continuously monitoring latencies of the Linux kernel. The results, as well as all the source code, hardware schematics, PCB layouts and other materials, are freely available.

Keywords: CAN bus, Linux, RTEMS, Xilinx Zynq, MicroZed, latency, gateway, SJA1000, CAN IP Soft Core, UIO

Abstrakt

Tato práce popisuje podobu a vývoj softwarového a hardwarového systému pro přesné měření latence různých implementací brány (gateway) pro sběrnici CAN, a to s rozlišením na mikrosekundy. Účel projektu je především testování softwarové brány implementované v Linuxu. Tato práce rozšiřuje předchozí čistě softwarový tester. Jako vývojová platforma je použit Xilinx Zynq SoC s integrovaným FPGA, což přináší značné zvýšení přesnosti měření. Latence získané tímto systémem jsou pak porovnány s výsledky z předchozího čistě softwarového řešení. Experimentální výsledky ukazují zvýšení přesnosti měření o 60 %. Vyvinutý systém je začleněn do testovací konfigurace pro souvislé monitorování latencí Linuxového jádra. Výsledky, spolu se všemi zdrojovými kódy, hardwarovými schematy a výkresy desky plošných spojů, jsou volně dostupné.

Klíčová slova: CAN bus, Linux, RTEMS, Xilinx Zynq, MicroZed, latence, brána, SJA1000, CAN IP Soft Core, UIO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Controller Area Network (CAN) is still by far the most widespread networking standard used in the automotive industry today, even in the most recent vehicle designs. Although there are more modern solutions available on the market (such as FlexRay or various industrial Ethernet standards), CAN represents a reliable, cheap, proven and well-known network. Thanks to its non-destructive and strictly deterministic medium arbitration, CAN also exhibits very predictable behavior, making it ideally suited for real-time distributed systems. Because of these indisputable qualities, it is unlikely that the CAN is going to be phased out in foreseeable future (cited from [12]).

Linux is often used in embedded devices since it offers more flexibility and supports a huge number of user-space applications, thus speeding up the development process. While not a real-time operating system, many applications require decent performance in terms of average and especially worst-case latencies. In distributed systems interconnected with multiple CAN buses, such as in cars, the latency of processing CAN frames is important. The Linux kernel is, however, a complex project and as it is not primarily targeted at real-time systems, the latency profile changes significantly with each released version. The only practical way to determine its latency profile is by benchmarking.

Despite this, there exists a patch set called RT Linux, which enables to compile the kernel as fully preemptive, lowering the worst case latencies and making the Linux kernel more suitable for (soft) real-time systems.

As the development of Linux goes rapidly forward, there is a need to continually test each new version.

Benchmarking also has the potential to discover bugs or regressions introduced into the kernel, as has already happened with the previous software-only system [12].

This work aims at extending the previously used testbed for CAN bus latency measurements. Software source codes of the extended system, as well as all hardware-related materials (such as schematics and PCB layout) are publicly available in GIT[1] repositories [23, 22].

[1] `https://git-scm.com/`

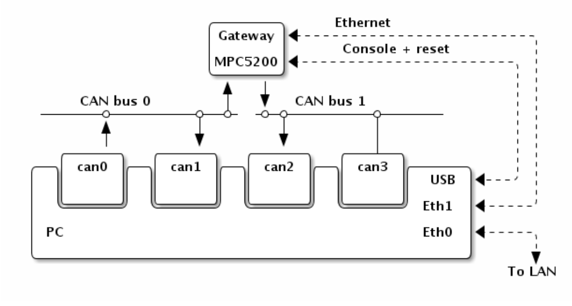


Figure 1.1: Original testbed configuration [12]

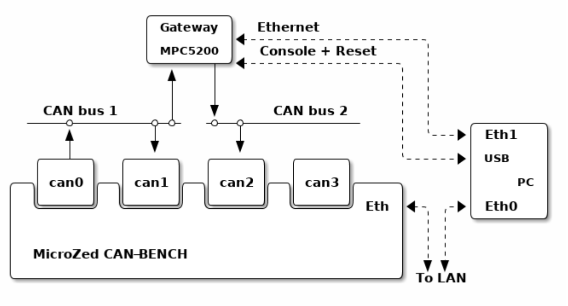


Figure 1.2: Extended testbed configuration

1.1 Testbed setup

The original testbed consists of the gateway under test and a standard PC. For the extended version, the testing instead takes place in a dedicated MicroZed board (based on Zynq SoC) with specially designed carrier card and the PC serves merely to coordinate the testing process. Diagrams of both testbeds are depicted in figures 1.1 and 1.2.

The testing platform – in this case the MicroZed board – needs the total of 3 CAN bus interfaces: one for transmission and two for reception. The additional reception interface is connected to the same bus as the transmission one so that the exact time when the frame actually appears on the bus is captured. This could be instead implemented with a CAN bus controller capable of reporting exact timestamp for transmitted frames or a timestamping controller capable of self-reception.

While the need for 3 interfaces could be eliminated in the hardware-assisted system, in the previous software-only system this was a necessity, as the timestamping was performed in software interrupt handler. The transmitted frame would be enqueued into TX FIFO of the CAN controller and its transmission could be delayed by waiting for preceding transmissions to be finished or even by retransmissions in case of bus error or arbitration loss.

The CAN-to-CAN gateway, whose latency is being measured, is an embedded board based on MPC5200B (PowerPC) microcontroller running at 400 MHz [12].

The benchmarking application running on MicroZed dedicated hardware transmits CAN frames via `can0`. The frame is then received by the gateway and at the same time on `can1`, which is connected to the same bus as `can0`. After the gateway processes and forwards the frame, it is received on `can2`. The difference of timestamps of the two frames received on `can1` and `can2` then gives the total latency, which besides the measured gateway processing latency also includes the frame transmission time.

Bitrate of all buses is configured to 1 Mbps.

1.2 Project goals

The main goal of this project is to increase the precision of measuring timestamps of all sent and received frames, moving the timestamping to hardware and eliminating additional latencies introduced by the host system.

Chapter 2

Theoretical background

2.1 CAN bus overview

CAN is a multi-master serial bus with high reliability and an arbitration mechanism allowing for minimal latency when transmitting a high-priority message [6]. The first CAN specification originated in 1986, later in 1993 the updated specification, CAN 2.0, was standardized as ISO 11898.

An extensive description of all CAN sub-layers is given in [6, 5, 15], and the CAN 2.0 specification is freely available on Robert Bosch GmbH website [3]. Overview of the physical layer is also available in [4].

CAN bus consists of several layers as shown in Figure 2.1. The link layer assumes the bus may be in two states – recessive or dominant. If two nodes are transmitting recessive and dominant bits at the same time, the bus will be in dominant state. This is similar to wired-AND. Zero bits are dominant, ones are recessive. If a node transmitting a recessive bit detects a dominant bit on the bus, it loses arbitration and the node transmitting the higher-priority message will take over the transmission.

CAN specifies 4 frame types: data frames, remote frames, error frames and overflow frames. A data frame consists of Start of Frame bit (SOF), header, data, footer, 7 End of Frame bits (EOF) and 3-bit intermission field (IMF). The header contains most importantly a 11bit or 29bit long frame identifier, CAN ID. This serves as unique message type identifier and at the same time determines the frame priority. Frames with lower CAN IDs have higher priority. A CAN frame may contain 0–8 data bytes. The footer contains a 15bit CRC field. All one-bit errors are guaranteed to be detected, most two-bit errors will also be detected [6]. After the CRC follows ACK bit. The transmitting node sends this bit as recessive, and all receiving nodes must acknowledge correct reception of the frame by sending a dominant bit.

CAN bus uses non-return-to-zero (NRZ) signal encoding, and no clock signal is part of the physical interface. Therefore, it must be ensured that the maximum permissible interval between two edges is not exceeded. This is achieved by a process called *bit stuffing.* After 5 consecutive bits of the same value, one additional bit of the opposite polarity must be inserted. The receiver must then undo this process. This is important for synchronization purposes.

One bit time consists of several *time quanta* and is separated into 4 segments – Sync,

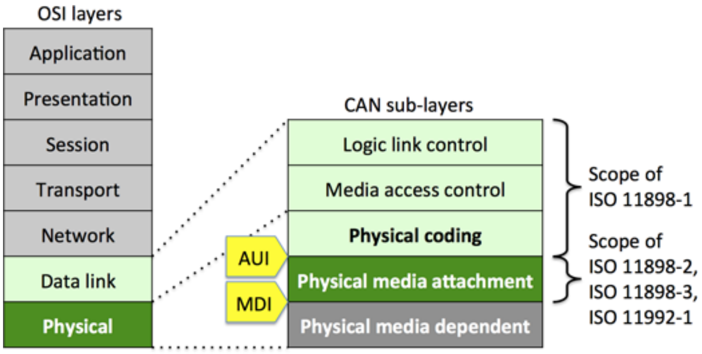


Figure 2.1: CAN sub-layers. The AUI (Attachment Unit Interface)is the interface between CAN controller and CAN transceiver; the MDI(Medium-Dependent Interface) is the interface to physical bus-lines [6].

Propagation, Phase 1, Phase 2. Their durations are configuration-dependent. These are important for node synchronization. More information is available in [6]. For the arbitration process to function correctly, the signal must be able to propagate from the transmitting node to every other node and back in less that one bit time so that a node transmitting a recessive bit can detect if the bus is in dominant state and interrupt its transmission. This effectively limits the length of the bus.

A new standard, CAN FD 1.0, was approved in 2015. It offers higher throughput by adding second (higher) bitrate for data and increasing maximal data length to 64 bytes.

2.1.1 Implementation in Operating Systems

2.1.1.1 Linux: SocketCAN

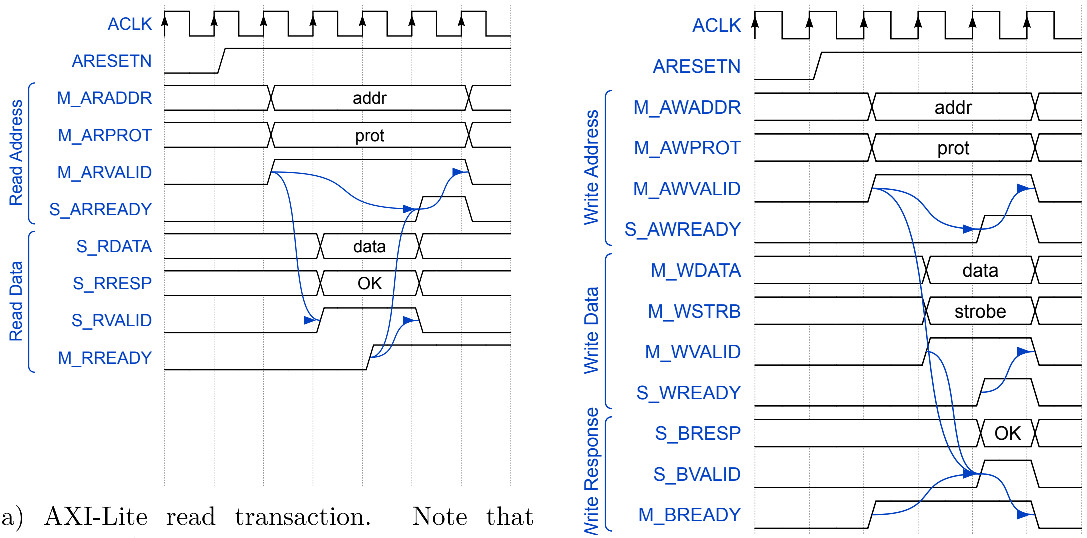
Linux CAN subsystem is built upon the networking subsystem. The basic idea is that similarly as in networking, higher-level protocols may be implemented upon the CAN link layer. Moreover, the advantage of this approach is that many applications may use the CAN interface simultaneously. This could be achieved by implementing the CAN controller drivers as character devices, but a significant portion of the networking subsystem would have to be duplicated

The networking subsystem is not, however, optimized for small messages and has rather large overhead. The performance in conjunction with CAN is thus substantially sub-optimal. There exist an alternative CAN subsystem, called LinCAN[1], which design its own generic API, avoiding the bottleneck. Further information and motivation behind SocketCAN may be found in [7].

2.2 Xilinx Zynq SoC

Xilinx Zynq is an integrated SoC including two ARM Cortex-A9 CPU cores, an FPGA and lots of integrated peripheral controllers, most importantly two CAN bus controllers, UART, and Ethernet.

[1] http://ortcan.sourceforge.net/lincan/



(a) AXI-Lite read transaction. Note that if M_RREADY was asserted together with M_ARVALID, the slave would not have to wait for it and the transaction would take 2 cycles less.

(b) AXI-Lite write transaction. Master may set address and data independently of each other, however the AXI-Lite slave waits until both channels have a valid value.

Figure 2.2: Example of AXI-Lite transactions. All signals beginning with M_ are driven by master, these beginning with S_ are driven by slave.

The SoC consists of Processing System (PS) and Programmable Logic (PL). All the embedded peripherals belong to PS. Signals to peripherals may be connected via MIO pins (Multiplexed IO) or through EMIO interface (Extended MIO) which goes through PL.

2.3 AXI4

AXI is an on-chip bus with master-slave architecture, used to connect CPU and peripherals. It is part of AMBA specification and three subtypes exist: AXI, AXI-Lite, AXI-Stream. All subtypes are fully duplex, allowing parallel read and write access. The peripherals on AXI bus are memory-mapped. AXI-Lite, which was used to connect peripherals in this project, is the least complex and serves to access fixed-size peripheral registers. (Full) AXI then maps a memory region and adds support for transaction reordering, cache control and supports burst transactions. AXI-Stream is used for streaming large amounts of data from or to a peripheral.

The full specification may be downloaded from ARM website [1] after registration and accepting the license agreement. An example of single read and write AXI-Lite transactions is depicted in figure 2.2.

2.4 Message timestamps in Linux networking subsystem

There exist several standard methods of acquiring incoming packet timestamp from userspace. All of them use the so-called control messages (retrievable via `recvmsg(2)`) to pass the timestamp to user space. Different methods are enabled with so-called socket options (SO).

The following list is taken directly from [8], where detailed information is available.

- `SO_TIMESTAMP`
  Generates a timestamp for each incoming packet in (not necessarily monotonic) system time. Reports the timestamp via recvmsg() in a control message as struct timeval (usec resolution).
- `SO_TIMESTAMPNS`
  Same timestamping mechanism as `SO_TIMESTAMP`, but reports the timestamp as struct timespec (nsec resolution).
- `IP_MULTICAST_LOOP + SO_TIMESTAMP[NS]`
  Only for multicast:approximate transmit timestamp obtained by reading the looped packet receive timestamp.
- `SO_TIMESTAMPING`
  Generates timestamps on reception, transmission or both. Supports multiple timestamp sources, including hardware. Supports generating timestamps for stream sockets.

In this project, hardware timestamping using `SO_TIMESTAMPING` is implemented.

Chapter 3

MicroZed CAN-BENCH System Description

Hardware of the MicroZed CAN-BENCH measuring system consists of a MicroZed board featuring Zynq SoC with embedded FPGA and a custom-designed CAN-BENCH carrier card, which has been developed in the scope of this project. The device is running PetaLinux, an embedded Linux distribution from Xilinx. The kernel version used is 4.0.0-xlnx.

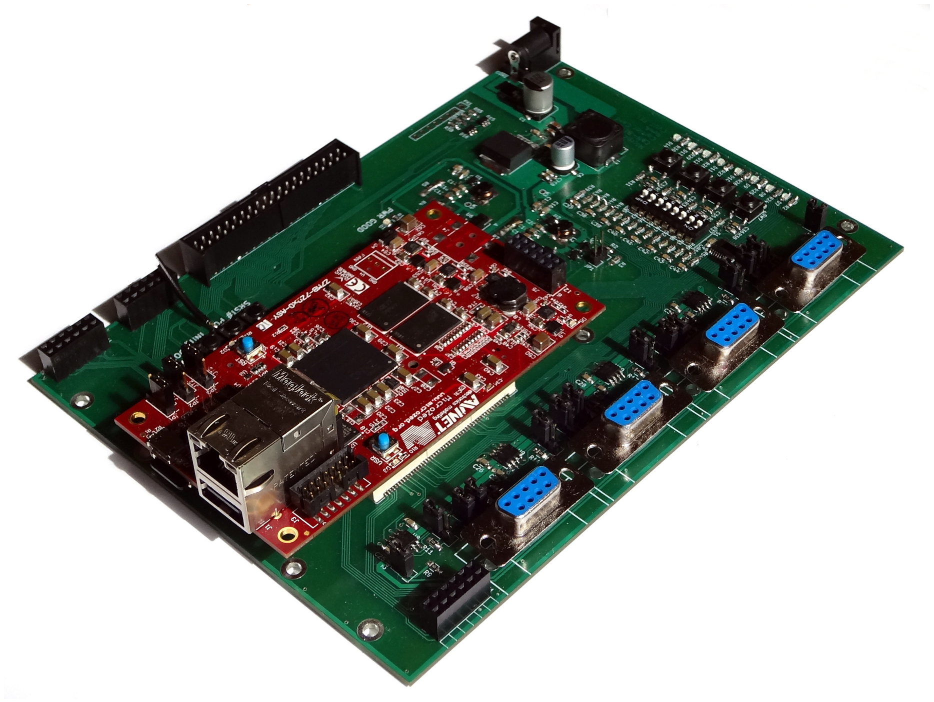


Figure 3.1: CAN-BENCH Carrier Card

3.1 CAN-BENCH Carrier Card

Features:

- 4x CAN bus interface: high-speed CAN Transceiver (CAN FD ready), standard D-SUB connector, optional on-board bus termination, each optionally connectable to on-board common bus
- on-board common CAN bus with optional double-sided termination
- 8x General purpose DIP switch
- 4x General purpose push button
- 8x General purpose LED (red)
- 8V–40V Power supply via 5.5/2.5mm barrel jack connector
- 1x Raspberry Pi Expansion Header connector
- 2x PMOD connector (PL side)
- 1x PS PMOD connector (same as on MicroZed)
- Power Good indicator LED (green)
- FPGA DONE indicator LED (blue)
- Reset buttons

3.1.1 Power Supply

Power supply for the CAN-BENCH Carrier Card is connected via 5.5/2.5mm barrel jack connector. The input voltage in the range from 8 V to 40 V is converted to 5 V by LM2676-5.0 buck DC-DC regulator. The board has no overvoltage protection; however, a series diode is present to prevent damage if power supply with reversed polarization is used.

The 5 V supply is routed via board-to-board connectors to MicroZed, where it serves as the main power supply. The USB UART on MicroZed is separated by a diode, so the USB UART may be used simultaneously with the carrier card power supply connected.

Voltages for both PL I/O Banks are also generated. Proper power sequencing is maintained, as required in [20].

3.1.2 CAN bus bridging

The particular CAN interfaces may be bridged together either at hardware side or in software. While the former way requires manual jumper setting and is mainly useful for hardware testing or hard setting, the latter may be used from applications for various reasons and is actually used by `latester` for broadcasting a frame for synchronizing hardware times between CAN interfaces.

3.1.3 Reset Buttons

The board design includes three distinct reset buttons, each having a slightly different effect. The INIT# button was adopted from AvNet I/O Carrier Card, although its function is unclear, as the Zynq TRM explicitly states that it should not be externally held low [17].

Button / Signal name	**Function**
SRST#	Soft-reset. Resets only the Processing Systems,
POR	Power-on-Reset. Resets the whole device.
INIT#	Delay the initialization of PL.

Table 3.1: CAN-BENCH Carrier Card Reset Buttons

3.2 Peripherals

3.2.1 Embedded Xilinx CAN Controllers

The Zynq SoC contains two independent embedded CAN controllers, with RX and TX FIFO and support for hardware timestamping of RX frames [16]. The timestamp is captured from a free-running 16bit counter register which is incremented once per every peripheral clock cycle. The clock is set to the frequency of 20 MHz in this project, so the overflow period is 3.768ms.

Despite the official documentation stating that the timestamp is sampled at last EOF bit [16], this was experimentally found to be untrue – the timestamp is sampled at the beginning of CAN frame.

That is because minima of delays between sampling the timestamp and receiving an interrupt were found to differ significantly for frames of unequal lengths. This difference very precisely corresponds to the time required to transmit the additional data bits, including bit stuffing.

The possibility that the timestamps are instead sampled at the end of the previous frame was refuted by the following experiment:

1. `can0` and `can1` are bridged together in PL.
2. Timestamps of RX packets on `can0` are recorded.
3. A program sends repeatedly bursts of two frames on `can1` with a short delay between each pair. The delay should be longer than the frames transmission time. For the frames in the burst to be sent tightly following each other it is necessary that `can1` be associated to Xilinx CAN.
4. Differences between receive timestamps are printed out. The values should form a regular pattern. The difference should be higher for the first frame in a burst than for the second one. If timestamps of the preceding frame were erroneously captured, the pattern would be reversed (i.e. shifted by one). This behavior was not observed.

RX and TX signals of the controller are routed to PL via EMIO interface. Here they are connected through CAN Crossbar to physical CAN interfaces.

3.2.1.1 Linux driver

Driver for Xilinx CAN Controller is present in mainline Linux kernel, although an extended version is available in the Xilinx tree. Neither, however, implements retrieving the RX frame timestamp provided by hardware.

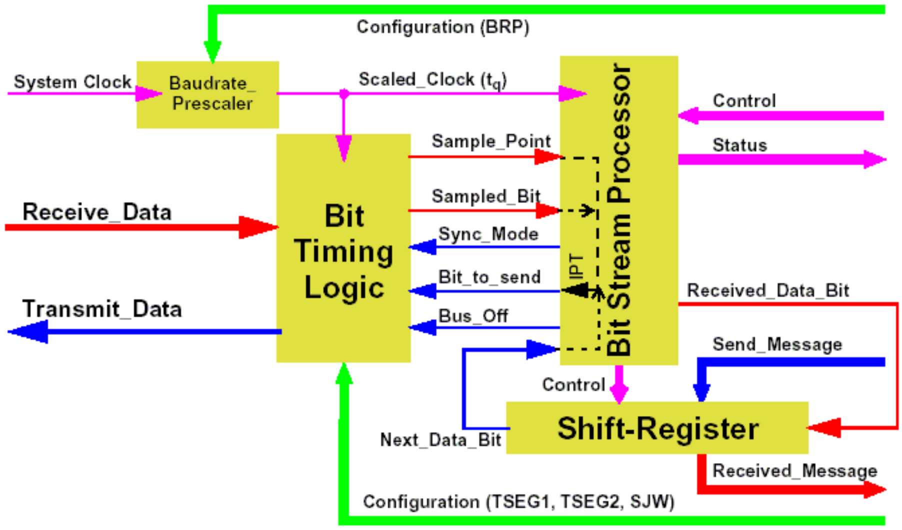


Figure 3.2: SJA1000 IP Soft Core Block Diagram [9]

This functionality had to be added, with the mainline driver version as the base. For general mechanism of handling message timestamps in Linux networking subsystem, refer to section 2.4. The patch for the `xilinx_can` driver is then described in section 4.4.

3.2.2 SJA1000 IP Soft Core

The soft core CAN controller used in this project is based on SJA1000 implementation in Verilog, available at Opencores.org [9]. The core is fully compatible with its hardware counterpart, including support for both compatibility and extended mode (PeliCAN).

Currently, the SJA1000 core is used only for transmission.

3.2.2.1 Register overview

The register layout and function is the same as for the original hardware SJA1000 part, with the exception that the 8bit registers are mapped into memory as 32bit registers. Future versions of the IP core may extend the functionality and use reserved space in some registers or add new registers into the mapping (see 6.1).

The registers may be accessed by 32bit, 16bit or 8bit access, however only the least significant bits are used by the peripheral. When read, the higher bits are zero and any value written into them is ignored.

The complete register description, as well as general structure of the SJA1000 chip, is not included here and may be found in [10].

3.2.2.2 Interrupts

Each SJA1000 IP core uses one Shared Peripheral Interrupt (SPI) line to signal all interrupt types. The interrupt source is then determined by reading the Interrupt Register (IR).

The interrupt is level-triggered active-high and is connected to SPI #61 and #62 for `sja1000_0` and `sja1000_1`, respectively. The interrupt is active if any bit in IR is set. After the Interrupt Register is read by the CPU all bits are reset, except for the receive interrupt bit, which is left intact[10].

3.2.2.3 Linux driver

Linux includes support for SJA1000 chips in mainline, supporting multiple ways of connecting the chip to the system. Some of the connection options are PCI, ISA or direct memory mapping. The `sja1000_platform` driver expects that the device registers are directly mapped into memory space, which is the case when using the soft core implementation. The device parameters for the driver are described in device tree. In this project, the configuration looks as follows:

```
sja1000_0: sja1000@43c00000 {
    compatible = "nxp,sja1000";
    reg = <0x43c00000 0x1000>;
    nxp,external-clock-frequency = <100000000>;
    interrupt-parent = <&intc>;
    interrupts = <0 29 4>;
    reg-io-width = <4>;
};
```

This means that an SJA1000 device is present in the system and

- has registers mapped in memory in address range `0x43c00000`–`0x43c00FFF`
- is clocked by 100MHz clock
- has active-high level-triggered interrupt #61 (the interrupt numbers are biased by −32 for some reason)
- registers are 32bits wide

The configuration is partly automatically generated[1] and partly written manually[2]. Further information on SJA1000 binding may be found in [11].

3.2.3 CAN Crossbar Soft Core

As mentioned in 3.1.2, the device supports configurable bridging of CAN interfaces. This is implemented by the `can_crossbar` IP core in PL. Additionally, CAN_STBY output pin is driven by this peripheral.

3.2.3.1 Functional description

The `can_crossbar` soft core peripheral enables to arbitrarily interconnect physical CAN bus interfaces (denoted `ifcN`) with embedded CAN controllers (denoted `canN`) in an M:N

[1] <canbench-sw>/petalinux/subsystems/linux/configs/device-tree/pl.dtsi

[2] <canbench-sw>/petalinux/subsystems/linux/configs/device-tree/system-top.dts

mapping. This is achieved by mapping both physical interfaces and controllers to virtual interconnect buses (denoted `lineN`).

Alternatively, the physical interfaces may be disconnected and the respective TX and RX lines in one bus connected together in PL.

All inputs to each bus line (`canN_TX`, `ifcN_RX`) are merged together by an AND gate. If any one of the inputs is in the state of logical zero, then, in terms of the CAN bus Specification, the whole bus is in the dominant state.

3.2.3.2 Register Overview

All registers are 32bits wide and should only be accessed by 32bit words.

CAN Configuration Register

Address offset: 0x000
Reset value: 0x000fe4e4

Register 3.1: CCR (0x000)

Reserved	STBY	OE_LINE4	OE_LINE3	OE_LINE2	OE_LINE1	CAN4_LINE	CAN3_LINE	CAN2_LINE	CAN1_LINE	IFC4_LINE	IFC3_LINE	IFC2_LINE	IFC1_LINE	
31 21	20	19	18	17	16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
0	0	1	1	1	1	1 1	1 0	0 1	0 0	1 1	1 0	0 1	0 0	Reset

STBY CAN Transceivers standby mode. If set to 1, all CAN Transceivers are put into standby mode and are not operational.

OE_LINEn LINEn Output Enable. If set to 1, the line is connected to its assigned physical interface. If set to 0, the line TX signal is connected to its RX signal and the physical interface is disconnected. This effectively connects all CAN controllers attached to the line together.

CANn_LINE Specifies which line is the physical interface connected to:
00: The physical interface is connected to LINE1
01: The physical interface is connected to LINE2
10: The physical interface is connected to LINE3
11: The physical interface is connected to LINE4

IFCn_LINE Specifies which line is the CAN controller connected to:
00: The CAN Controller is connected to LINE1
01: The CAN Controller is connected to LINE2
10: The CAN Controller is connected to LINE3
11: The CAN Controller is connected to LINE4

3.2.3.3 Linux driver

As the core has very limited functionality and simple interface, a full-featured driver is unnecessary. Instead Userspace I/O (UIO) driver is used. The UIO driver basically offers user-space applications to map a region of physical address range, belonging to a given peripheral, to the application's virtual memory. The driver is informed of which memory regions belong to a device from device tree. The application then calls `mmap(2)` on the driver device file descriptor [14]. Peripheral registers may then be accessed by the application directly. Interrupt forwarding is also supported by the driver, however, is not used for this peripheral.

3.3 Booting process

The boot process is configured in a way to allow most flexibility. Linux kernel image, device tree blob, and FPGA configuration bitstream are loaded from network via TFTP. The root filesystem is then accessed remotely via NFS. Only the bootloader (generated FSBL and U-Boot) is stored on the SD card together with configuration.

The boot mode jumpers JP3–JP1 on MicroZed should be configured to SD card boot mode, as shown in Figure 3.3. The SD card should be formatted with FAT32 filesystem and contain two files:

- `BOOT.BIN` – the bootloader image
- `uEnv.txt` – U-Boot configuration

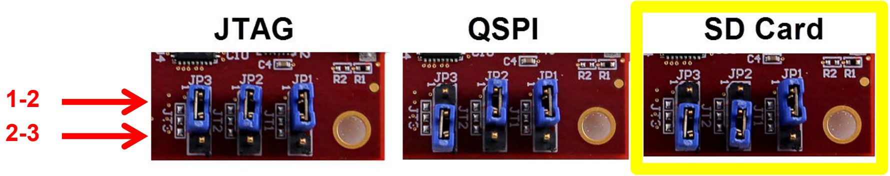


Figure 3.3: MicroZed Boot Mode Jumper Settings [2]

In the configuration file are stored the device IP address, TFTP server IP address, and a path to a bootscript file. The addresses may also be chosen to be configured via DHCP.

When the device is powered on, the bootloader reads `uEnv.txt` from the SD card and optionally retrieves the device and TFTP server IP addresses via DHCP. Then a bootscript is downloaded and executed. This offers maximal flexibility, as the bootscript may contain any sequence of U-Boot commands.

The default bootscript included in `canbench-sw` repository then proceeds to download a packed image via TFTP, which contains Linux kernel, flattened device tree blob and compressed FPGA bitstream. The bitstream is loaded into the FPGA chip via U-Boot built-in command. Finally, the Linux kernel is executed, with root filesystem device set to an NFS server specified in the bootscript.

The FPGA configuration may also be updated later from the running system, as described later in 4.7.4.

Interface	Driver	Support for HW timestamping
can1	xilinx_can	Yes
can2	xilinx_can	Yes
can3	sja1000_platform	No
can4	sja1000_platform	No

Table 3.2: CAN bus network interfaces in Linux

3.4 Software

Standard Linux command-line tools are installed, together with Midnight Commander, Dropbear (an SSH server), and canutils. An overview of CAN interfaces is listed in Table 3.2.

3.4.1 latester: the benchmarking application

`latester` is the application responsible for sending and receiving CAN frames and generating files with results. The basic concept of the testing method was given in section 1.1.

The frames may be generated one at a time or with a given fixed period. A unique identifier is stored in the first 2 bytes of each frame and is later used to match the received frame with the sent one.

3.5 Building the system

The exact steps for building the whole system are described in `<canbench-sw>/README.txt` in detail. All the steps (except user configuration and setting up external services) are fully automated. Here is a brief summary:

1. **Build Hardware Description File** (system.hdf)
   - Recreate the Vivado project
   - Build system.hdf
2. **Configure a TFTP server**
3. **Configure an NFS server**
   The server must support NFSv2, otherwise the MicroZed board will not boot, and no sensible error message will be printed.
4. Modify the module IP, server IPs and paths in u-boot environment and bootscript
5. Modify the NFS server IP and path in bootscript (petalinux/bootscript.txt)
6. Configure PetaLinux and applications
7. Build PetaLinux and applications
8. Copy kernel, FPGA bitstream, device tree and compiled bootscript into /tftpboot
9. Copy boot files to SD card

Chapter 4

Development and Implementation Choices

4.1 Hardware

The whole design was made with simplicity and future reusability in mind. The parts were selected to be available from Farnell.com and TME.eu, with regards to their qualities and price. KiCad[1] was used for designing the schematics and PCB layout.

Board schematics and PCB layout are enclosed in Appendix B and Appendix C and may be found in publicly available repository [22].

Board revision A contains some minor errors, which will be fixed in the next revision and made available in the repository. The only serious error is swapping of power supply pins of all voltage supervisor circuits. This may be fixed by connecting the parts to the correct inputs by wires.

4.1.1 Form of hardware solution

There was a fundamental choice to be decided during the project analysis phase, namely whether a full-featured carrier card would have to be designed, a commercially available carrier card used together with a simple CAN I/O board or if the need for a carrier card could be eliminated altogether.

The two embedded Xilinx CAN Controllers may have their I/O signals routed to PS MIO pins which are accessible via J5 PMOD connector on MicroZed. However, for the purpose of this project at least one additional CAN bus controller is required and must be implemented in FPGA. Unfortunately, MIO signals cannot be routed to Programmable Logic [19], and thus a need for a carrier card arises.

The second alternative was to purchase MicroZed I/O Carrier Card from AvNet and design only a small board with CAN transceivers and connectors and a PMOD connector. It soon became apparent that the dimensions of the CAN expansion board would be comparable to the size of the carrier card itself. Moreover, disregarding the development, the price of a custom full-fledged carrier card would be far lower. Considering this all and the

[1] `http://kicad-pcb.org/`

fact that the board might be reused in the future as a multi-purpose evaluation board, the variant of designing a custom carrier card was finally chosen.

4.1.2 CAN bus Transceiver

There were several criteria for CAN bus transceiver selection:

- Availability
- Separate V_{IO} supply pin to interface directly to 3.3 V logic
- High maximal frequency (for future testing of CAN FD)
- Simple interface

The following chips were considered:

- MCP2562FD
- TJA1057GTJ
- TJA1041AT

MCP2562FD finally showed as the best candidate by all criteria and was selected for the design.

4.1.2.1 CAN bus termination and testing

A 120 Ω terminator may optionally be connected to each bus by closing the associated jumper. Additionally, each CAN bus interface may be connected to a common internal bus via two jumpers (for CAN-LO and CAN-HI). The common bus may be optionally terminated at either or both sides. This allows for easy testing of hardware by looping a frame from one interface to another.

4.1.3 I/O protection and isolation

Galvanic isolation on CAN bus interface was considered, but for simplicity and cost reasons was deemed unnecessary and finally omitted from the design.

PMOD connectors and Raspberry Pi Expansion Header connector are not ESD protected. This would mean additional cost and would add extra capacitive load to the pins, lowering the maximal transmission rate.

The pins leading to a screw terminal block are, however, protected by a serial ESD-protection circuit and a serial 100 Ω resistor.

4.1.4 Power Supply

The device was required to be able to operate from 12–24 V power supply; MicroZed, however, requires a 5 V supply. An extra step-down DC-DC regulator thus had to be incorporated into the design. The MicroZed itself may draw up to 1.2 A at 5 V [20]. To be able to provide sufficient power at the same time acceptable efficiency, the LM2676-5.0 fixed-voltage Buck regulator with the maximal output current of 3 A was finally chosen as the best candidate.

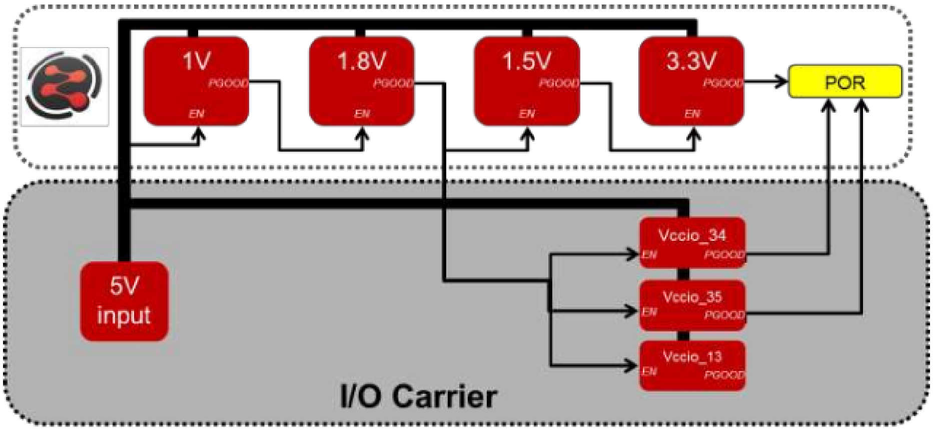


Figure 4.1: Power Architecture and Sequencing Diagram [20]

Each of the two PL I/O Banks in Zynq 7010 has separate power supply – VCCIO_34 and VCCIO_35 – which may be 1.8 V, 2.5 V or 3.3 V. On CAN-BENCH board, both voltages are fixed to 3.3 V but may be reconfigured to another voltage by resoldering the reference voltage divider resistors if the need should arise.

Although only one regulator would be sufficient for both banks, for greater flexibility and better robustness an independent regulator circuit is used for each bank. TPS62260DDC adjustable voltage synchronous buck regulator with the maximal output current of 1 A has been selected.

4.1.4.1 Power Supply Sequencing

Proper sequencing must be followed on power-up as well as on power-down, as specified in MicroZed Carrier Card Design Guide [20].

An open-collector voltage supervisor (U4) guards the output of the main 5 V regulator and is connected to the PWR_EN signal, which enables the regulators following in the power-up sequence.

The VCCIO voltage regulators are enabled by VCCIO_EN signal tied to JX2 pin 10, which is the open-collector PGOOD output of 1.8 V regulator on MicroZed, pulled up to 1.5 V by 1K/4K99 voltage divider. The Enable input of VCCIO regulators must thus be compatible with these levels. PGOOD outputs of voltage supervisors for both VCCIO voltages (U6, U7) are tied to PG_MODULE signal, as on Figure 4.1.

There is one additional voltage supervisor U5, which together with protection diodes D4, D5 serves to maintain proper power-down sequencing, i.e. that the VCCIO voltages are shut down before the main power supply. When the main power goes down, the VCCIO_EN signal is pulled low as soon as possible. The protection diodes ensure that the VCCIO voltages are always lower than the voltage of the 5 V supply network. This could happen on shutdown when excessive charge is stored in capacitors connected to VCCIO network.

4.1.5 General Purpose User I/O

Some additional I/O connectors and peripherals have been included in the design to make the board usable even for future applications and experiments.

All the buttons and switches have transient spike suppression and are pulled up by 3.3 kΩ resistors to VCCIO_35, which is 3.3 V, and are thus active in logical 0. The inputs are connected to JX2 header with additional series resistance to prevent damage in case of misconfigured pins.

The LEDs are connected through 74HCT245 octal 3-state bus transceiver, which serves here simply as a buffer. Its power supply is connected to 5 V and its TTL-compatible inputs are routed to 3.3 V CMOS FPGA outputs. All LEDs are active in the input state of logical one.

4.2 Software

The software development was divided into following tasks:

- Preparation of necessary IP Cores
- Configuration of the Processing System
- Modifying device drivers and device tree
- Configuring and building bootloaders and Linux kernel
- Modifying and building applications

4.2.1 Tools Used for Development

Xilinx Vivado Design Suite has been used for FPGA development and PS configuration. This software is available for download after registration on Xilinx website. The free WebPACK license should be sufficient for purposes of this project. The SDK must also be installed by explicitly selecting it in the installer.

Linux distribution PetaLinux from Xilinx is used. It offers user-friendly configuration and building of all necessary components (FSBL, U-Boot, Linux kernel, rootfs). In addition, it provides good interoperability with outputs of Vivado. This software is available for download after registration on Xilinx website.

4.2.2 Configuring the Processing System in Vivado

This section brings brief description how the Vivado project for CAN-BENCH system was created. Most of the information could be gathered elsewhere as these steps do not differ significantly from creating a generic project targeted at the MicroZed board. A great source of knowledge has been [13] and numerous Xilinx user guides.

Note that it is not necessary to repeat these steps to build the software components for the CAN-BENCH measuring system. Automatic scripts, which are part of the `canbench-sw` repository, take full care of this task.

1. Download MicroZed board definition files from MicroZed website[2] and extract them into Vivado installation directory. Download the PS Preset TCL file as well and save it for later use.
2. Open Vivado and create new RTL project. Add constraints file `microzed_CAN-CC_RevA.xdc` which may be found in the `canbench-sw` repository [23]. This file creates named I/O ports assigned to Zynq I/O pins, compatible with the CAN-BENCH Carrier Card. In the *Select Default Part* dialog, select MicroZed 7010 Board.
3. Create a Block Design and add ZYNQ7 Processing System IP. Now source the PS Preset file downloaded in step 1 by executing `source MicroZed_PS_properties_v03.tcl` in the TCL console. This configures the PS IP Core according to the hardware choices made on the MicroZed board.
4. In Customize Block dialog, enable CAN0 and CAN1 peripherals, assigning their I/O ports to EMIO. Then in the PS-PL Configuration tab enable the option AXI Non-secure Enablement → GP Master AXI Interface → M AXI GP0 Interface. This will make a master AXI interface available in the block design and allows to connect custom peripherals to the AXI bus.
5. After closing the Customize Block dialog, automatic connection of signals is offered by Vivado. Full automation works well.
6. Add the desired IP cores and connect them or let Vivado connect them automatically.
7. Right-click on the top-level block design file in Sources and select Create HDL Wrapper and "Let Vivado manage and auto-update". Now the synthesis, implementation and bitstream generation may be run.
8. The resulting bitstream file, as well as the hardware definition file, is created in Implementation Run directory, local to project root. From there it may be copied manually or by exporting it via File → Export → Export Hardware → Include Bitstream.

The Vivado project files are impractical to be directly stored in version control systems. There exist two officially suggested ways to solve this problem [21]:

1. Abandon the project workflow and use the so-called Non-project workflow, i.e. write custom TCL build scripts and manage every aspect of the project manually.
2. Export the project to a TCL script which is able to recreate the project from scratch. There is rarely a need to change the script and project-mode offers automatic management of sources, included IPs and compilation runs. The build script may thus be very simple.

While the first option offers most flexibility, at the time of project beginnings I had no previous experience with the Vivado Design Suite and chose the simpler second option, which for purposes of this project is fully sufficient.

The generated script was then manually edited and simple build script was created. Both are part of the project build process and may be found in directory `/system/scripts` in the `canbench-sw` repository.

[2] `http://zedboard.org/support/documentation/1519`

The CAN-BENCH Makefile copies the resulting hardware definition file and bitstream file to /system/system.bit and /system/system.hdf, respectively.

Learn the script commands and language syntax is greatly simplified by the fact that commands for all operations made in Vivado GUI are printed to TCL Console.

4.3 Creating PetaLinux Build

PetaLinux SDK must be installed. This is only available for GNU/Linux operating system. The steps below assume that both Vivado SDK tools and PetaLinux SDK tools have been added to execution path. This may be done by sourcing the appropriate settings.sh files in their installation directories.

The PetaLinux project was created as usual, and then system configuration was loaded.

```
$ petalinux-create -t project -n canbench --template zynq
$ mv canbench petalinux
$ cd petalinux
$ petalinux-config --get-hw-description ../system --oldconfig
```

All necessary components – U-Boot, Linux kernel, and basic user-space applications – are part of the PetaLinux SDK. To support a fully customized network boot, it was, however, necessary to hook into the build process encapsulated by the petalinux-build command. Many times this presented quite a challenge, as the system is very tightly integrated. Nonetheless, after the U-Boot auto-configuration was disabled in configuration and the relevant parts copied from the respective Makefile, the build process could be altered, and U-Boot configuration is now being patched to include the necessary features. The code may be found in /petalinux/Makefile in canbench-sw repository.

4.4 Extending the Xilinx CAN Linux driver

As has been already stated in 3.2.1.1, the xilinx_can driver both in mainline and Xilinx tree lacks support for retrieving hardware timestamps from the peripheral. This support is added by a custom patch which might in the future be merged into the mainline kernel.

The embedded Xilinx CAN Controller passes 16bit frame timestamps in the two least significant bytes of the DLC field of an incoming frame. Detailed information on the timestamp resolution and sampling have already been provided in 3.2.1.

After the hardware timestamp is retrieved, it is converted to a 64bit timestamp with nanosecond resolution. Overflows of the 16bit timestamp must be handled properly.

The algorithm goes as follows:

```
ktime_t get_frame_timestamp(u16 frame_cantime, ktime_t frame_ktime) {
  if first frame
    ref_ktime = frame_ktime
    ref_cantime = frame_cantime
    exact_frame_ktime = frame_ktime;
  else
    frame_cantime_full = ktime_to_cantime(frame_ktime) - ref_cantime
    replace 16 least significant bits of frame_cantime_full by frame_cantime
    frame_cantime_full += ref_cantime
    exact_frame_ktime = cantime_to_ktime(frame_cantime_full)
  return exact_frame_ktime
}
```

This algorithm has proven to be superior to an earlier version, as that had suffered slight problems at hardware counter overflows.

The `frame_ktime` is retrieved by software in device interrupt handler and thus represents a time point **after** the actual frame timestamp. This delay (latency) is not constant, and it must be noted that the references – `ref_ktime` and `ref_cantime` – do not represent the same timepoint either. Forgetting this may introduce anomalies into calculations, as was the case with the previous algorithm.

This algorithm will, however, experience problems if the hardware clock and ktime start to diverge. If the drift exceeds half the hardware counter overflow period, some timestamps may then be off by one overflow period. While this is unlikely when ktime is provided directly by a hardware timer, there may exist scenarios where this poses a real danger, such as when the time flow speed gets altered by NTP to compensate for hardware clock frequency inaccuracies. In that case, the reference times must be periodically updated or adjusted.

Dynamic adjustment of references is, however, unsuitable for this project, as the time drift between interfaces must remain constant during a test. For this reason, as the CAN-BENCH board is intended to run non-stop, the driver is reloaded before each measurement to eliminate any potential inaccuracies.

4.5 Adapting the SJA1000 IP Core

The original SJA1000 IP core provides either Wishbone[3] interface or the 8051 interface. For interfacing with the Zynq SoC, AXI wrapper needed to be implemented.

As the AXI bus supports full duplex communication, the IP core was modified to provide read and write register access simultaneously, and the CAN clock is set to the same source as the AXI clock so that clock domain crossing is avoided. This is the simplest way of porting the core, as this allows to use auto-generated AXI peripheral template from Vivado. Also, this solution offers more performance compared to arbitrating between read and write accesses and synchronizing over clock domains. It also leaves less room for errors.

4.6 Extending latester

The benchmarking application itself is a slightly extended version of the original application called `latester`, developed at Department of Control Engineering, FEE CTU.

`latester` is used with only minor modifications:

- Added support for retrieving hardware timestamps of RX frames
- Determining time offset between the two hardware time counters

[3] `http://cdn.opencores.org/downloads/wbspec_b4.pdf`

4.7 Debugging

4.7.1 Testing the Embedded Xilinx CAN Controllers

The two embedded CAN controllers were enabled, and their I/O pins were configured to EMIO, i.e. programmable logic, where the two CAN buses may be bridged together.

If the controllers are not properly connected, the associated network interfaces will fail to start as the controller fails to enter NORMAL mode on chip startup, which is detected by the driver. This is because that "After the CEN bit is set to 1 the CAN controller waits for a sequence of 11 recessive bits before exiting configuration mode." [TRM 18.3.2].

The controllers were then tested by looping a frame from one interface to another. The SJA1000 controllers were then tested the same way.

```
# canconfig can0 bitrate 1000000
# canconfig can1 bitrate 1000000
# canconfig can0 start
# canconfig can1 start
# candump can0 &
# cansend can1 0x55 0x88
```

4.7.2 Testing the SJA1000 IP Core

Only minimal modifications have been made to the SJA1000 Core. Furthermore, the AXI slave interface was automatically generated by Vivado. Both parts were assumed to be functioning correctly, and thus only their interaction had to be tested. This proved to be feasible directly in hardware, without simulations. As expected, no significant problems were encountered.

4.7.3 Debugging the xilinx_can driver timestamping patch

For debugging the timestamp calculation code in `xilinx_can` driver, the RX and TX signals of all CAN bus controllers – two embedded Xilinx CAN controllers and two soft core SJA1000 controllers – were tied together in PL fabric. This ensured that all the controllers would receive a frame in exactly the same moment.

When `latester` was run in this configuration, using the Xilinx CAN controllers for reception, the difference in reception timestamps on the two interfaces should be exactly zero. To be more precise, the difference always has a constant offset, because the software timestamps used in calculations are not synchronized between the devices. This synchronization is done in `latester` as described in Section 3.4.1.

As the algorithm did not work the first time, debugging information had to be retrieved from the driver. To prevent undesired delays caused by printing to kernel log, the relevant values were embedded directly into the received frame, overwriting the data, and then parsed and displayed by `latester`.

4.7.4 Deployment of binary images

As with every development, there was a large amount of modify–compile–test cycles. Since the programs had to be run on external hardware, it was essential that the deployment procedure be as much automated as possible.

The MicroZed CAN-BENCH board boots via TFTP from network and then mounts its root filesystem via NFS. Both TFTP and NFS servers were configured to be on the development PC, and the board could be connected to via SSH. Deploying a modified binary was thus as simple as copying it to the NFS root on the local machine.

The FPGA bitstream may also be updated without restarting the system. The generated bitstream file must first be converted into a different format, and is then simply written to `/dev/xdevcfg`, which will cause the FPGA reconfiguration. This conversion is performed automatically by the build scripts and is described in [18]. The driver is not included in mainline Linux kernel and the Xilinx version must be used.

Chapter 5

Running the Benchmarks

The configuration of the testbed used to run all the tests has already been described in section 1.1. In this chapter, the framework for automatic continuous testing (and even backtesting) will be described. Evaluated test cases, the results, and their comparison with the measurements obtained by the original software-only system will then be presented.

5.1 Continuous Testing System at DCE Servers

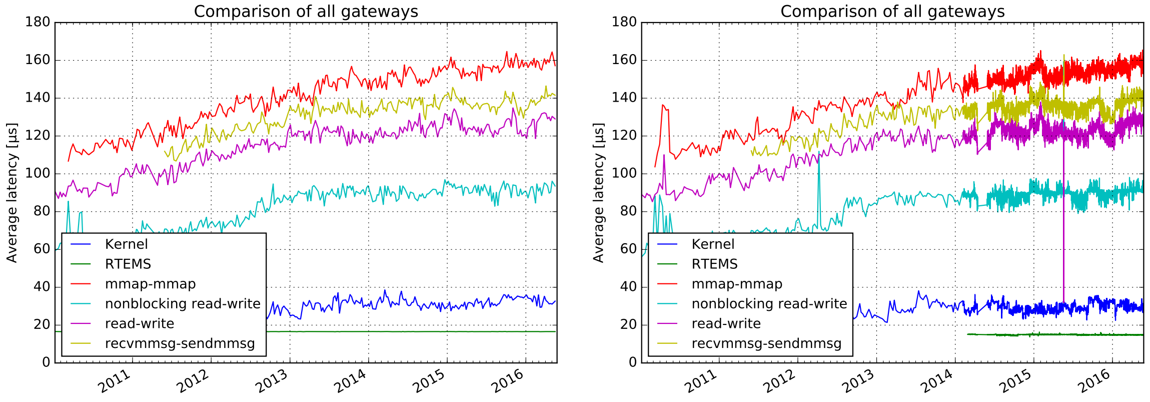
The continuous testing system resides in its own repository, together with `latester`, and is included in the `canbench-sw` repository as a submodule.

The tests are run periodically every 8 hours on a DCE server from a Cron job. The testing job consists of the following steps:

1. Testing is initiated by Cron on DCE compile server.
2. New Linux kernel version is fetched from its main repository and compiled for the testing platform (PowerPC).
3. The compiled kernel is uploaded to the PC connected to the PowerPC evaluation board.
4. The target board is reset, boots the desired image, and starts the selected gateway.
5. `latester` is run on MicroZed CAN-BENCH board. The relevant measurement results are printed to standard output, which is captured to a logfile.
6. Steps 4–5 are repeated for each tested gateway type.
7. The log file is parsed, and graphs including the new values are generated and published to the website[1].

At the time of writing this thesis, only the results of the original system are available on the website. However, the testing platform will be migrated in the foreseeable future and the website updated accordingly.

[1] https://rtime.felk.cvut.cz/can/perf/



(a) Measurements from extended system

(b) Measurements from original system

Figure 5.1: Comparison of different gateways latencies in history

5.2 Results

For re-evaluating the tests with the extended system, all tagged Linux versions (releases and release candidates) from version 2.6.33-rc1 to 4.6 were tested. The PowerPC testing board cannot boot with older kernels. Also note that the kernel CAN gateway is available since version 3.2-rc1 and some of the user space gateways also require a higher kernel version.

The comparison of latencies of all tested gateways may be observed in 5.1 for both the original and extended testing system. Figure 5.2 then shows latencies of each gateway type in a separate graph.

The RTEMS gateway always runs the same RTEMS version and serves as a reference and is very deterministic, as can be seen from results of the hardware-assisted system. Several “steps” are noticeable in the plotted results of the software-based system. These probably represent the host computer upgrades, as the testing environment had then changed. Spikes are also clearly visible and the overall spread of measured latencies is rather big, compared to the new results.

We may assume that the measurements of the extended system are exact. This should be true by design and have been proven empirically by testing with CAN interfaces bridged together in PL, as described in 4.7.3. Under this assumption, we may compare the two datasets and determine the difference in means, ranges[2], and standard deviations. For the software-only system, only the samples after the last “step” were considered, with the spikes filtered out. The averages of both data sets differ by approximately 1.5 μs and the range and standard deviation are both reduced by approximately 60 %.

A similar test was performed with the Linux kernel gateway: one kernel version was repeatedly tested with both the original and extended systems and the same analysis was performed. The results were similar, only the spreads were naturally bigger due to Linux being less deterministic.

[2] `https://en.wikipedia.org/wiki/Range_(statistics)`

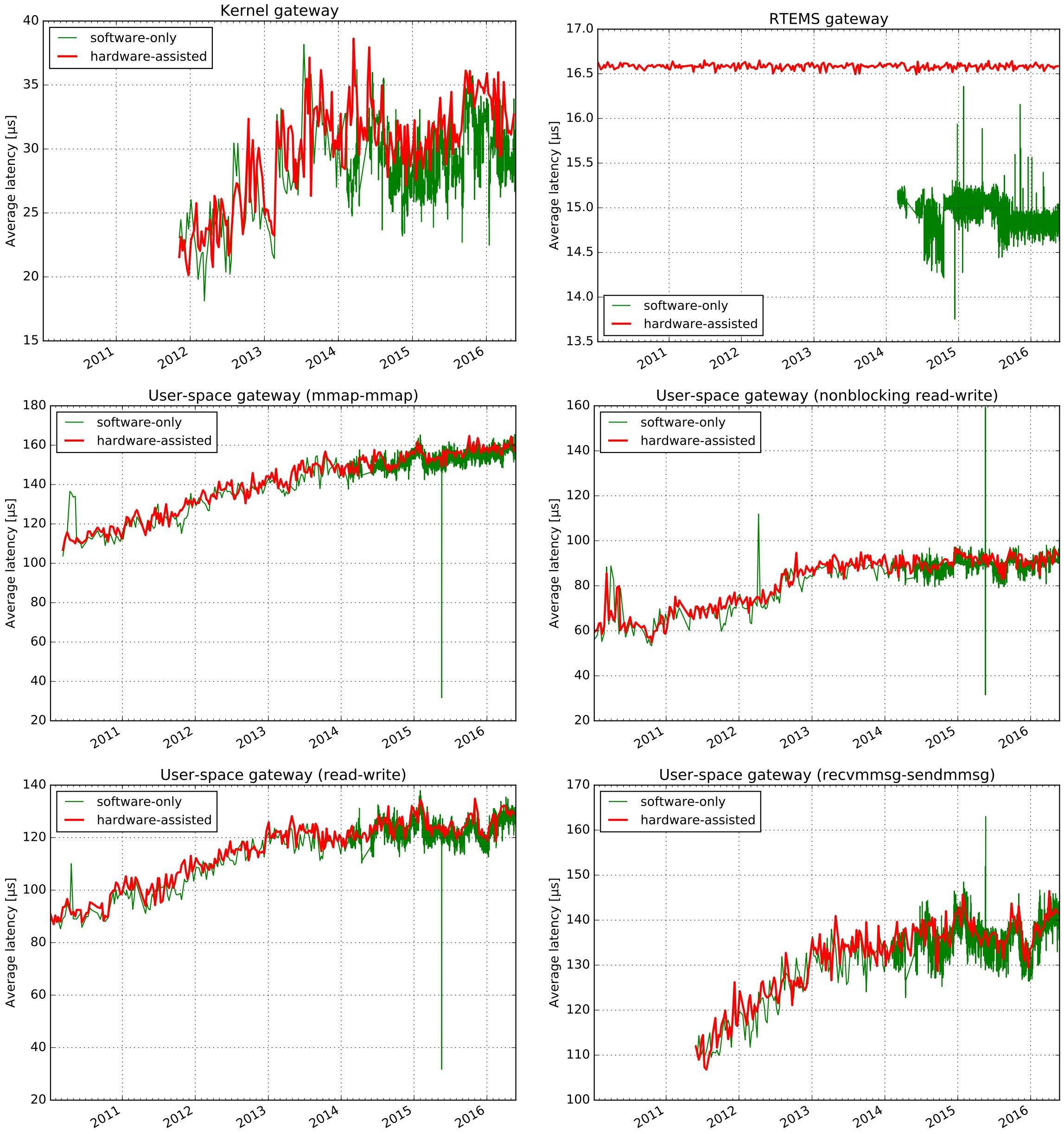


Figure 5.2: Comparison of individual gateway latencies between the original and extended system

Chapter 6

Conclusion

The goal of this project was to increase the precision of measuring processing latency of software CAN-to-CAN gateways. This was successfully achieved. The new solution has been integrated into the system for continuous testing and will be migrated to in close future.

6.1 Future improvements

There is always room for improvements. Some the more prominent ones are briefly described in this section.

Adding TX FIFO to SJA1000 IP

As SJA1000 has only one TX buffer, it would be benefitial to amend this by implementing a TX FIFO similar to the RX FIFO already present in the extended mode (PeliCAN). This would, in its simplest form, require no modifications to the Linux driver as the interface would remain (mostly) unchanged.

The frame to be transmitted is written into TX registers, then the Transmission Request (TR) bit is set in the Command register. The software has to wait until the Transmit Buffer Status (TBS) bit in the Status register is set to indicate that the buffer is available. With standard SJA1000 this happens after the transmission is complete. The extended version would simply move the window to next free slot in the TX FIFO, set the TBS bit and issue an interrupt immediately (if the queue is not full).

The meaning of transmission abort request would have to be slightly redefined to either abort the whole queue or somehow shift the frames in the TX FIFO to allow to inject a more urgent frame to the front.

Precise Frame Transmission Timing

To allow precise frame timing with resolution to CAN bittime clock, the SJA1000 Soft Core could be extended by adding a new field to TX frame registers, representing the number of ticks since the last frame end to wait before transmitting current frame. This obviously assumes the TX FIFO is implemented as the software is not able to enqueue next frame so quickly and is the reason why this feature would be useful in the first place.

Extending the Test Cases

It would be beneficial to integrate collecting best-case and worst-case latencies to the continuous testing and graphs. Currently only average latencies are collected. With the increased accuracy, it would be possible to extend the system to continuously measure performance of gateways based on real-time operating systems, for instance RTEMS, already used for tests in signle version.

Also detailed test with cummulative latency histograms as outputs, as performed before with the software-only systems, could be re-evaluated with increased precision. The testing had been done for various combinations of kernel version, bus saturation, ethernet traffic, and CPU load. The results, which may in the future be updated, are publicly available at WWW[1].

[1] http://rtime.felk.cvut.cz/can/benchmark/3.0/, http://rtime.felk.cvut.cz/can/benchmark/1/

Bibliography

[1] ARM. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022d` [Online; accessed 2016-05-27].

[2] AvNet. *MicroZed Zynq™ Evaluation and Development and System on Module Hardware User Guide, Figure 8*, 2015. `http://microzed.org/sites/default/files/documentations/MicroZed_HW_UG_v1_6.pdf` [Online; accessed 2016-05-27].

[3] CAN specification, Version 2.0. `http://www.kvaser.com/software/7330130980914/V1/can2spec.pdf` [Online; accessed 2016-05-27].

[4] CAN bus Physical Layer. `https://support.dce.felk.cvut.cz/pub/hanzalek/_private/ref/canphy.pdf` [Online; accessed 2016-05-27].

[5] CAN data link layers. `http://www.can-cia.org/can-knowledge/can/can-data-link-layers/` [Online; accessed 2016-05-27].

[6] CAN physical layer. `http://www.can-cia.org/can-knowledge/can/systemdesign-can-physicallayer/` [Online; accessed 2016-05-27].

[7] Linux SocketCAN. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/can.txt` [Online; accessed 2016-05-27].

[8] Message timestamping in Linux networking subsystem. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/timestamping.txt` [Online; accessed 2016-05-27].

[9] Mohor, I. SJA1000-compatible CAN Protocol Controller IP Core. `http://opencores.org/project,can,overview` [Online; accessed 2015-11-04].

[10] Philips Semiconductors. *SJA1000 Stand-alone CAN controller Data Sheet*, Jan. 2000. `https://www.nxp.com/documents/data_sheet/SJA1000.pdf` [Online; accessed 2016-05-27].

[11] Devicetree memory mapped SJA1000 CAN controller bindings. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/devicetree/bindings/net/can/sja1000.txt` [Online; accessed 2016-05-27].

[12] M. Sojka, P. Píša, O. Špinka, O. Hartkopp, and Z. Hanzálek. Timing Analysis of a Linux-Based CAN-to-CAN Gateway. In *Thirteenth Real-Time Linux Workshop*, pages 165–172, Schramberg, 2011. Open Source Automation Development Lab eG.

[13] Taylor, A. Adam Taylor's MicroZed Chronicles. http://adiuvoengineering.com/?page_id=285 [Online; accessed 2016-05-27].

[14] How to Design and Access a Memory-Mapped Device in Programmable Logic from Linaro Ubuntu Linux on Xilinx Zynq on the ZedBoard, Without Writing a Device Driver — Part Two, May 2013. http://fpga.org/2013/05/28/how-to-design-and-access-a-memory-mapped-device-part-two/ [Online; accessed 2016-05-27].

[15] Wikipedia. Can bus — wikipedia, the free encyclopedia, 2016. https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=719551093 [Online; accessed 2016-05-27].

[16] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual*, v1.10 edition, Feb. 2015. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf [Online; accessed 2016-05-27].

[17] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual, Table 6-24*, v1.10 edition, Feb. 2015. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[18] Zynq-7000 Example Design - Program the PL using the Linux driver for DEVCFG. http://www.xilinx.com/support/answers/46913.html [Online; accessed 2016-05-27].

[19] ZedBoard forum thread on impossibility of routing PS PMOD signals to PL. http://zedboard.org/content/pmod-0.

[20] MicroZed™ Carrier Design Guide, 2014. http://zedboard.org/sites/default/files/documentations/MicroZed_Carrier_Design_Guide_rev_1_5.pdf [Online; accessed 2016-05-27].

[21] Using Vivado Design Suite with Version Control Systems. http://www.xilinx.com/support/documentation/application_notes/xapp1165.pdf [Online; accessed 2016-05-27].

[22] CAN Benchmark hardware repository. http://rtime.felk.cvut.cz/gitweb/fpga/zynq/canbench-hw.git.

[23] CAN Benchmark software repository. http://rtime.felk.cvut.cz/gitweb/fpga/zynq/canbench-sw.git.

Nomenclature

AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
CAN	Controller Area Network
DCE	Department of Control Engineering
EMIO	Extended Multiplexed I/O
FIFO	First In Last Out
FPGA	Field Programmable Gate Array
FSBL	First Stage Bootloader
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input/Output
IDE	Integrated Development Environment
IP	Intellectual Property
IR	Interrupt Register
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MIO	Multiplexed I/O
NFS	Network File System
NTP	Network Time Protocol
PCB	Printed Circuit Board
PL	Programmable Logic
PS	Processing System

RTEMS	Real-Time Executive for Multiprocessor Systems
RX	Reception
SDK	Software Development Kit
SoC	System on Chip
SPI	Shared Peripheral Interrupt
SSH	Secure SHell
TCL	Tool Command Language
TFTP	Trivial File Transfer Protocol
TX	Transmission
UIO	Userspace I/O
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

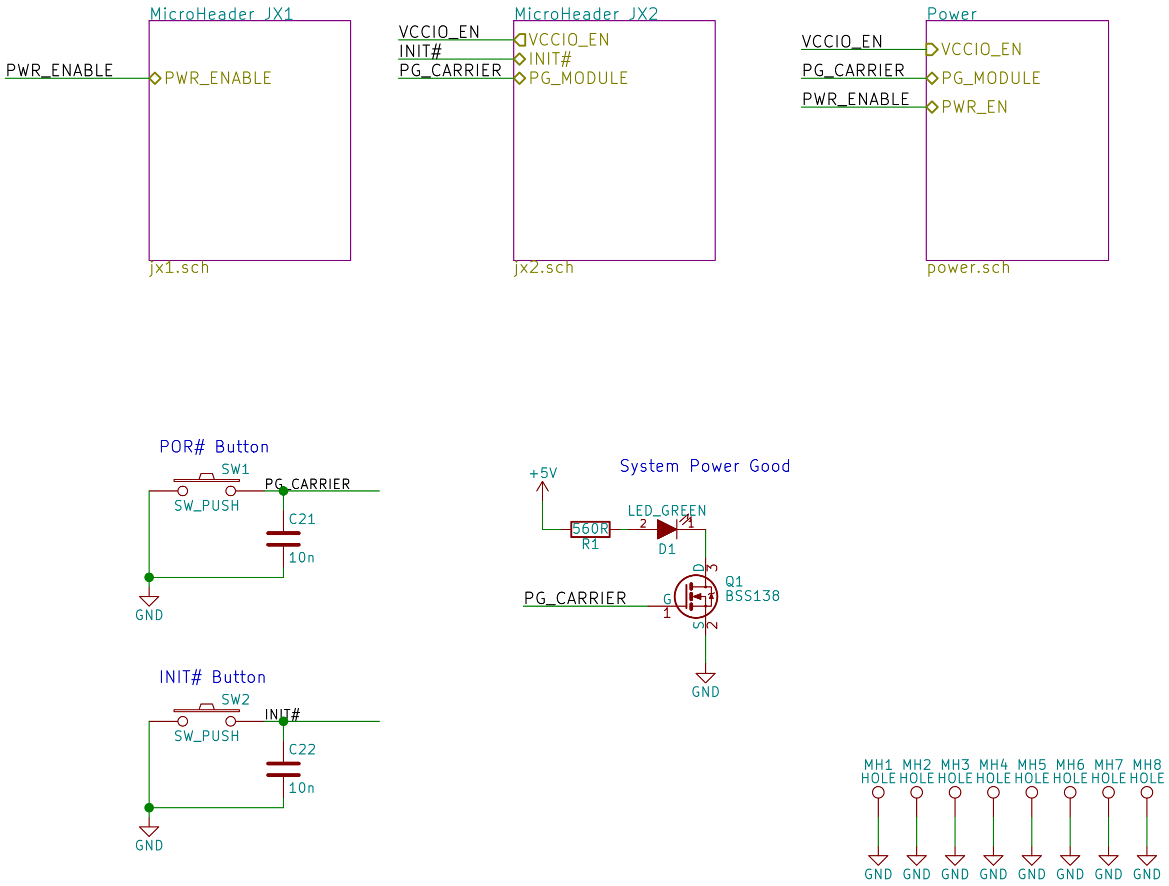
Appendix A

Contents of attached CD

```
/
|-- canbench-hw                     Hardware-related sources
|   |-- canbench-BOM.ods            Bill of Material
|   |-- canbench-hw.kicad_pcb       PCB Layout
|   |-- canbench-hw.pro             KiCad project
|   |-- canbench-hw.sch             Top-level schematics file
|   |-- lib                         KiCad part/footprint libraries
|   |   +-- [...]
|   |-- README.txt
|   |-- [...]
|   +-- xsl
|       +-- bom2groupedCsv.xsl      Template for generating BOM
|-- canbench-sw                     CAN-BENCH source codes
|   |-- can-benchmark               Submodule of testing system with latester
|   |   |-- continuous              Scripts for continuous testing
|   |   |-- latester                latester sources
|   |   +-- [...]
|   |-- petalinux                   PetaLinux SDK project directory
|   |   |-- bootscript.its          U-Boot configuration for packaging bootscript
|   |   |-- bootscript.txt          Default bootscript
|   |   |-- components
|   |   |   |-- apps                Makefiles for applications to be built
|   |   |   |   |-- bc              GNU bc
|   |   |   |   |-- canhwtstamp     Tool for dumping CAN frames with HW timestamps
|   |   |   |   |-- canutils        canutils precompiled for Zynq
|   |   |   |   |-- latester        Build files for latester
|   |   |   |   +-- mc              Midnight Commander
|   |   |   |-- generic
|   |   |   |   +-- scripts         Init and utility scripts
|   |   |   |-- libs                Makefile for libraries to be built
|   |   |   |   +-- talloc
|   |   |   +-- modules
|   |   |       +-- xilinx_can      Patched xilinx_can Linux kernel module
|   |   |-- Makefile
|   |   |-- subsystems              Subsystems configuration
|   |   |-- uboot-extra-env.h       Extra environment to build into U-Boot
|   |   |-- uboot-image.its       U-Boot configuration for packaging boot image
|   |   |-- uEnv.txt              Default runtime U-Boot environment
|   |   +-- [...]
|   |-- README.txt
|   +-- system                    Sources for Vivado project
|       |-- ip                    IP Cores
|       |   |-- canbench_cc_gpio
|       |   |-- can_crossbar_1.0
|       |   |-- can_merge
|       |   +-- sja1000_1.0
|       |-- script                Vivado TCL scripts to recreate and build project
|       |   |-- build.tcl
|       |   |-- dist.tcl
|       |   +-- recreate.tcl
|       |-- src
|       |   |-- constrs
|       |   |   +-- microzed_CAN-CC_RevA.xdc
|       |   +-- top
|       |       +-- top.bd        Top-level block design
|       +-- system.bif            Configuration for packaging FPGA bitstream
|   Jerabek-thesis-2016.pdf       Text of this thesis
+-- results                       Raw measured data and processing scripts
    |-- analyze.py                Script to analyze RTEMS data and print summary
    |-- highstock.js
    |-- index.html                HTML page with graphs
    |-- kernel-gateway.json
    |-- orig                      Data from the original system
    |-- plot.py                   Script to produce graphs from the data
    |-- rtems-gateway.json
    |-- user-space-gateway-mmap-mmap.json
    |-- user-space-gateway-non-blocking-read-write.json
    |-- user-space-gateway-read-write.json
    +-- user-space-gateway-recvmmsg-sendmmsg.json
```

Appendix B

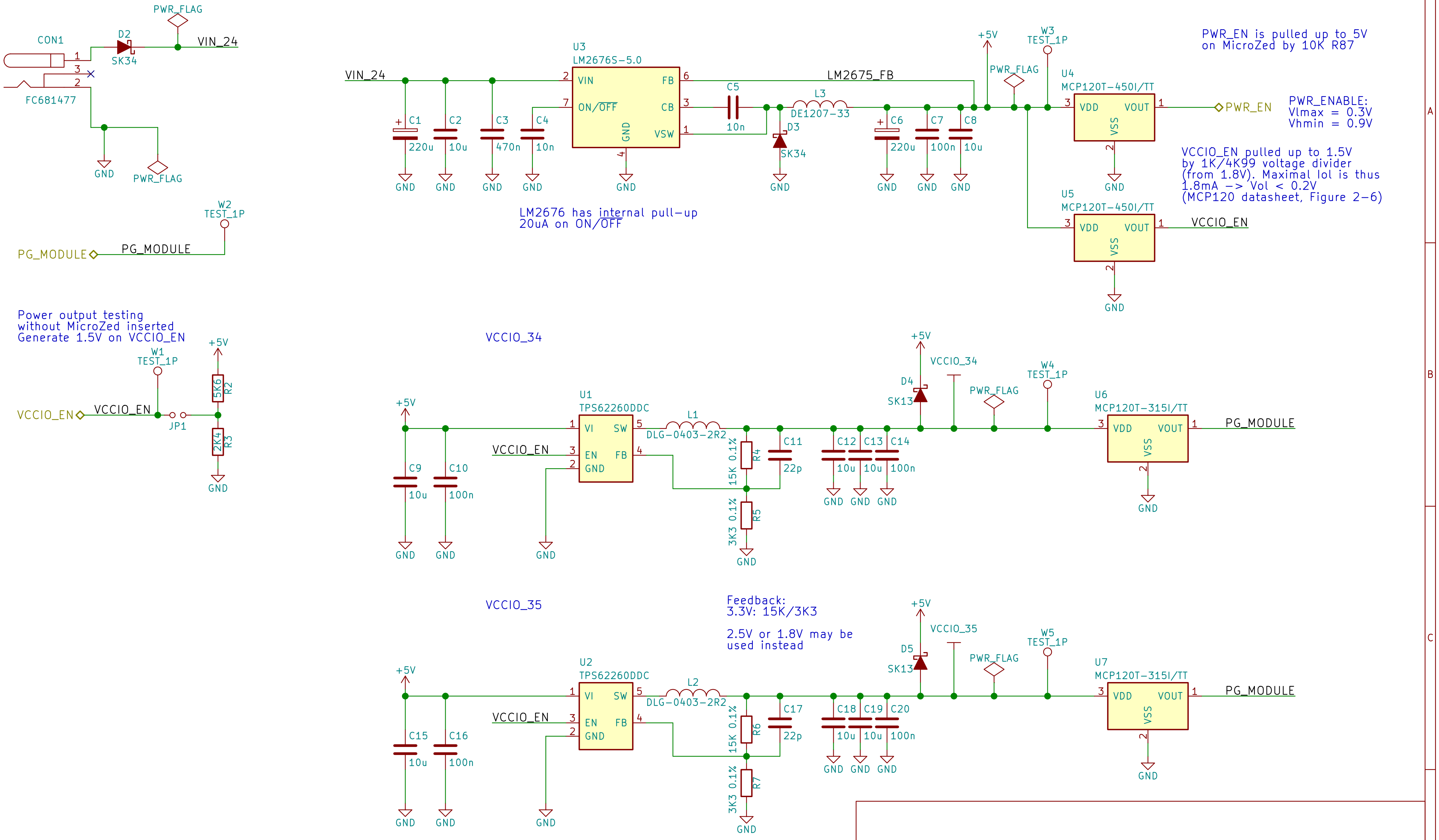
CAH-BENCH Schematics



Sheet: /
File: canbench-hw.sch

Title: CANbus MicroZed CarrierCard

Size: A4 Date: 25.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable	Id: 1/10



PWR_EN is pulled up to 5V
on MicroZed by 10K R87

PWR_ENABLE:
Vlmax = 0.3V
Vhmin = 0.9V

VCCIO_EN pulled up to 1.5V
by 1K/4K99 voltage divider
(from 1.8V). Maximal Iol is thus
1.8mA -> Vol < 0.2V
(MCP120 datasheet, Figure 2-6)

LM2676 has internal pull-up
20uA on ON/OFF

Power output testing
without MicroZed inserted
Generate 1.5V on VCCIO_EN

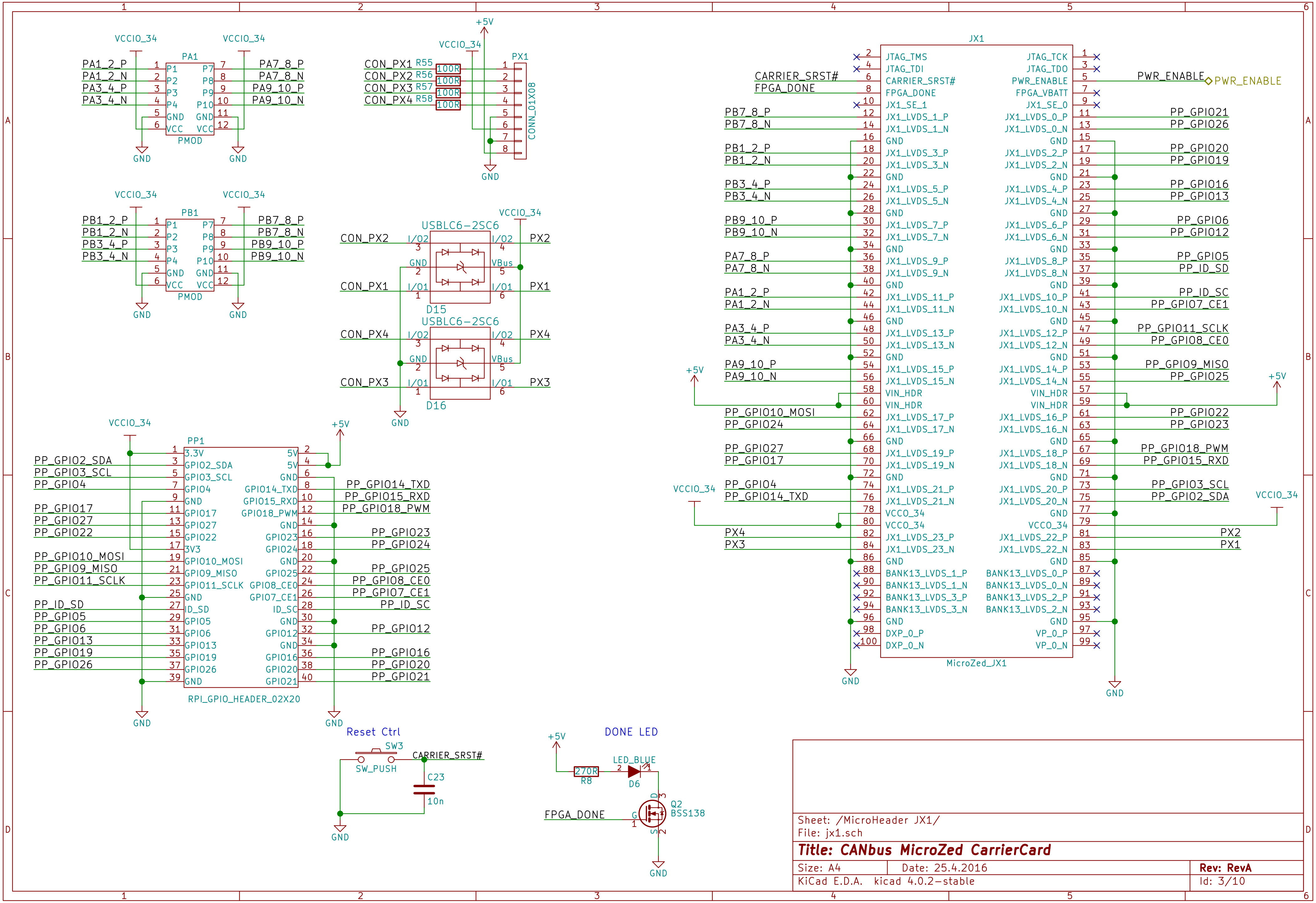
Feedback:
3.3V: 15K/3K3

2.5V or 1.8V may be
used instead

Sheet: /Power/
File: power.sch

Title: CANbus MicroZed CarrierCard

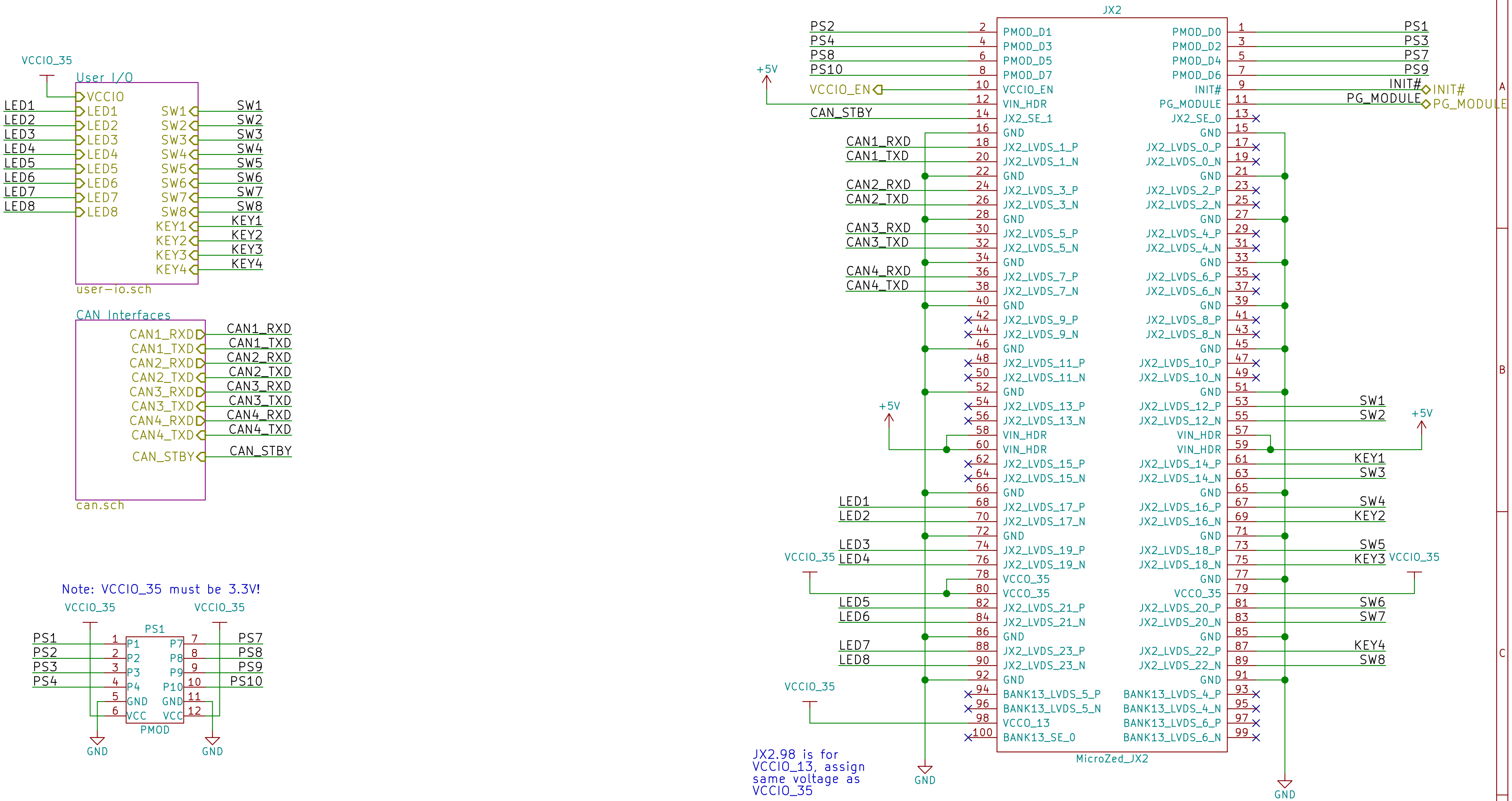
Size: A4 Date: 25.4.2016 **Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable Id: 2/10



Sheet: /MicroHeader JX1/
File: jx1.sch

Title: CANbus MicroZed CarrierCard

Size: A4 Date: 25.4.2016 **Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable Id: 3/10



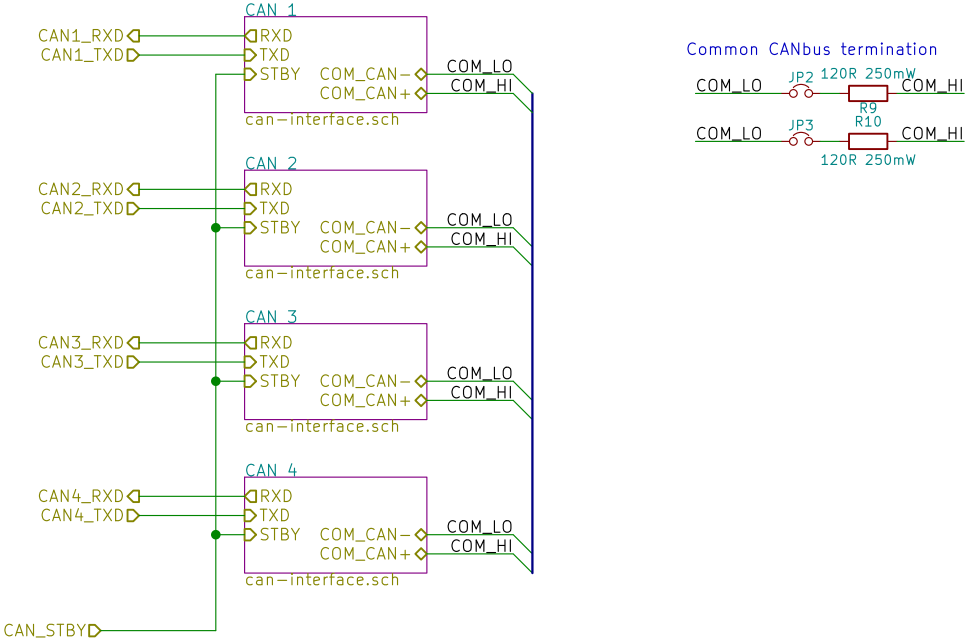
Note: VCCIO_35 must be 3.3V!

JX2.98 is for VCCIO_13, assign same voltage as VCCIO_35

Sheet: /MicroHeader JX2/
File: jx2.sch

Title: CANbus MicroZed CarrierCard

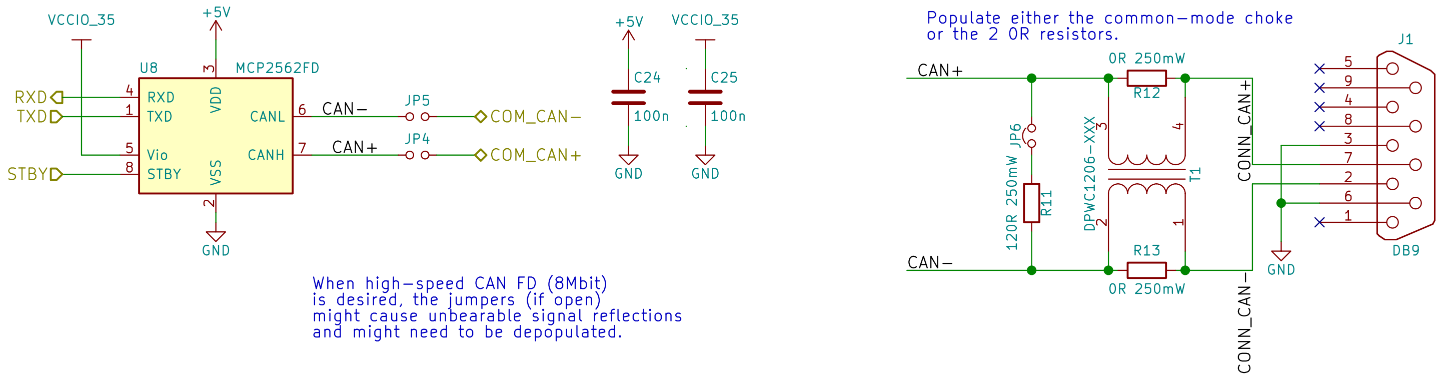
Size: A4	Date: 25.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable		Id: 4/10



Sheet: /MicroHeader JX2/CAN Interfaces/
File: can.sch

Title: CANbus MicroZed CarrierCard

Size: A4 Date: 25.4.2016 **Rev: RevA**
KiCad E.D.A. kicad 4.0.2−stable Id: 5/10



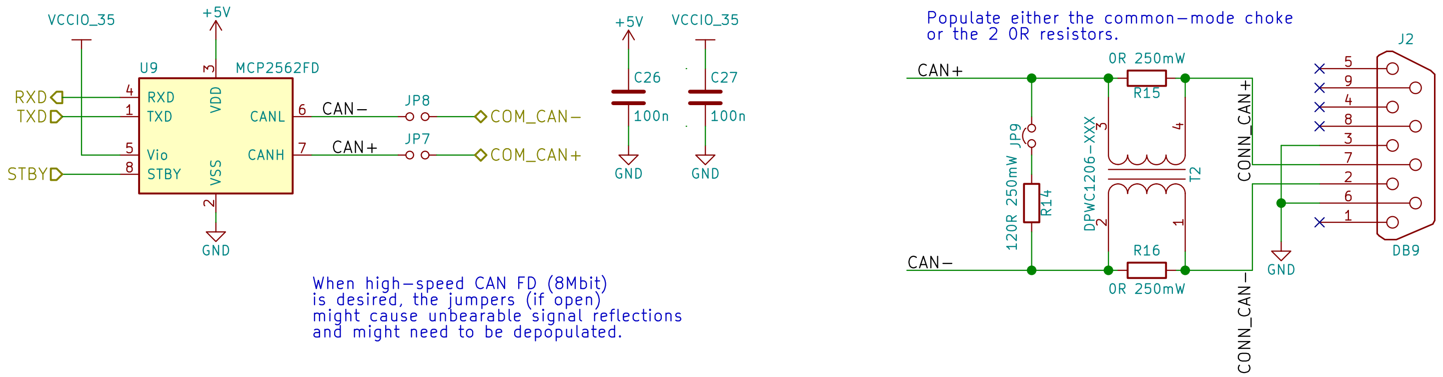
Populate either the common−mode choke or the 2 0R resistors.

When high−speed CAN FD (8Mbit) is desired, the jumpers (if open) might cause unbearable signal reflections and might need to be depopulated.

Sheet: /MicroHeader JX2/CAN Interfaces/CAN 1/
File: can−interface.sch

Title: CANbus MicroZed CarrierCard

Size: A4	Date: 25.4.2016	Rev: RevA
KiCad E.D.A. kicad 4.0.2−stable		Id: 6/10



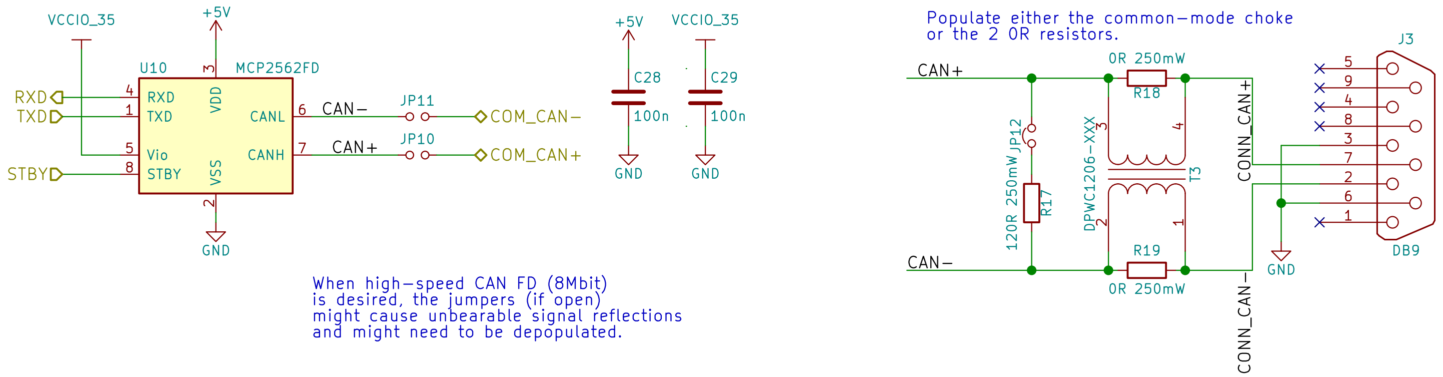
Populate either the common−mode choke
or the 2 0R resistors.

When high−speed CAN FD (8Mbit)
is desired, the jumpers (if open)
might cause unbearable signal reflections
and might need to be depopulated.

Sheet: /MicroHeader JX2/CAN Interfaces/CAN 2/
File: can−interface.sch

Title: CANbus MicroZed CarrierCard

Size: A4	Date: 25.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2−stable		Id: 7/10



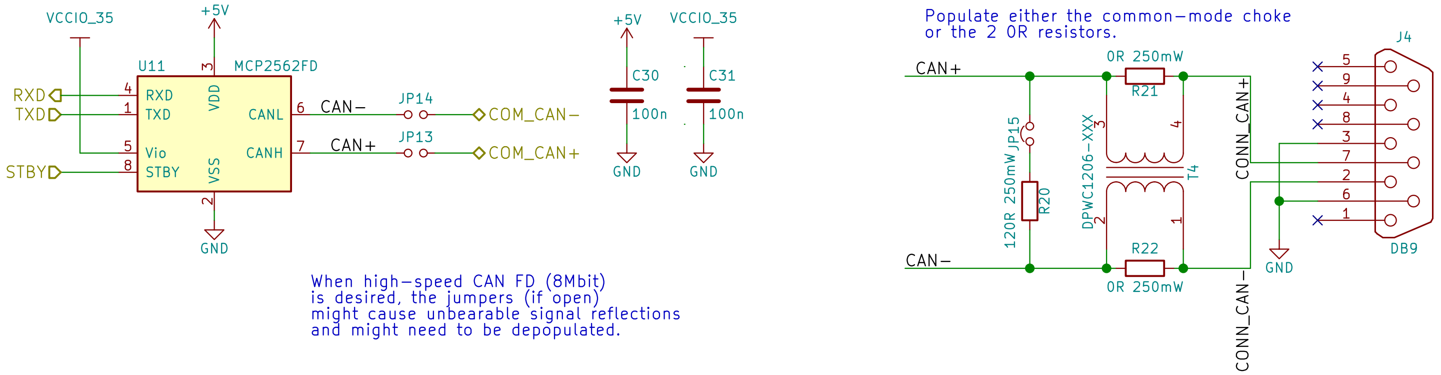
Populate either the common−mode choke or the 2 0R resistors.

When high−speed CAN FD (8Mbit) is desired, the jumpers (if open) might cause unbearable signal reflections and might need to be depopulated.

Sheet: /MicroHeader JX2/CAN Interfaces/CAN 3/
File: can−interface.sch

Title: CANbus MicroZed CarrierCard

Size: A4	Date: 25.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2−stable		Id: 8/10



Populate either the common−mode choke
or the 2 0R resistors.

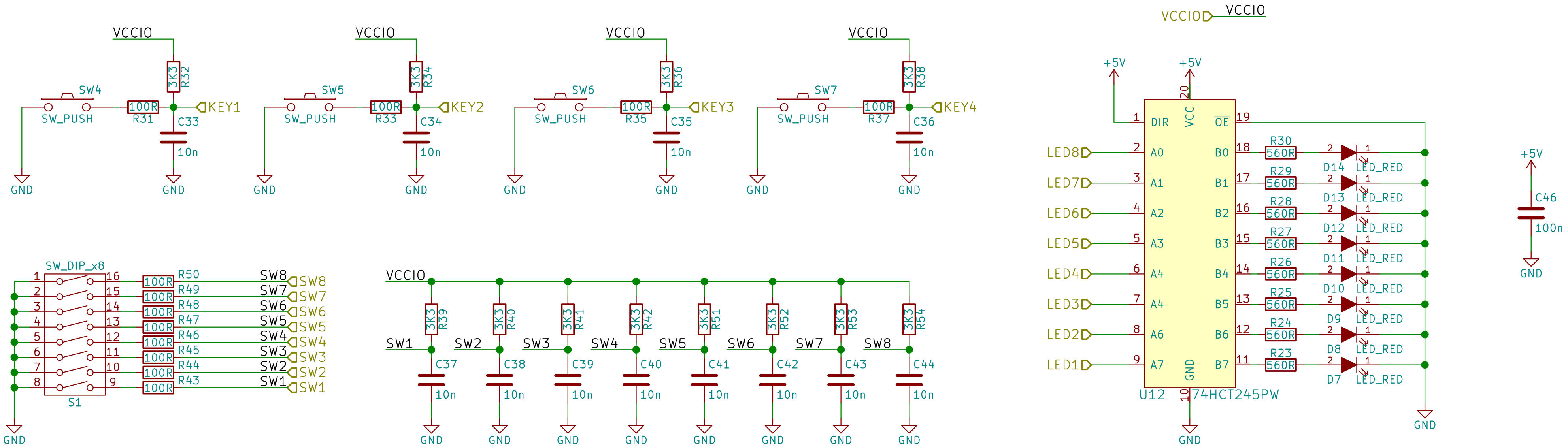
When high−speed CAN FD (8Mbit)
is desired, the jumpers (if open)
might cause unbearable signal reflections
and might need to be depopulated.

Sheet: /MicroHeader JX2/CAN Interfaces/CAN 4/
File: can−interface.sch

Title: CANbus MicroZed CarrierCard

Size: A4	Date: 25.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2−stable		Id: 9/10

User LEDs


VCCIO
SW4 SW_PUSH R31 100R R32 3K3 C33 10n KEY1 GND
SW5 SW_PUSH R33 100R R34 3K3 C34 10n KEY2 GND
SW6 SW_PUSH R35 100R R36 3K3 C35 10n KEY3 GND
SW7 SW_PUSH R37 100R R38 3K3 C36 10n KEY4 GND
VCCIO VCCIO
+5V +5V
U12 74HCT245PW
DIR VCC OE A0 A1 A2 A3 A4 A4 A6 A7 B0 B1 B2 B3 B4 B5 B6 B7 GND
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
LED8 LED7 LED6 LED5 LED4 LED3 LED2 LED1
R30 R29 R28 R27 R26 R25 R24 R23 560R
D14 D13 D12 D11 D10 D9 D8 D7 LED_RED
+5V C46 100n GND
SW_DIP_x8 S1
R50 R49 R48 R47 R46 R45 R44 R43 100R
SW8 SW7 SW6 SW5 SW4 SW3 SW2 SW1
R39 R40 R41 R42 R51 R52 R53 R54 3K3
C37 C38 C39 C40 C41 C42 C43 C44 10n
GND

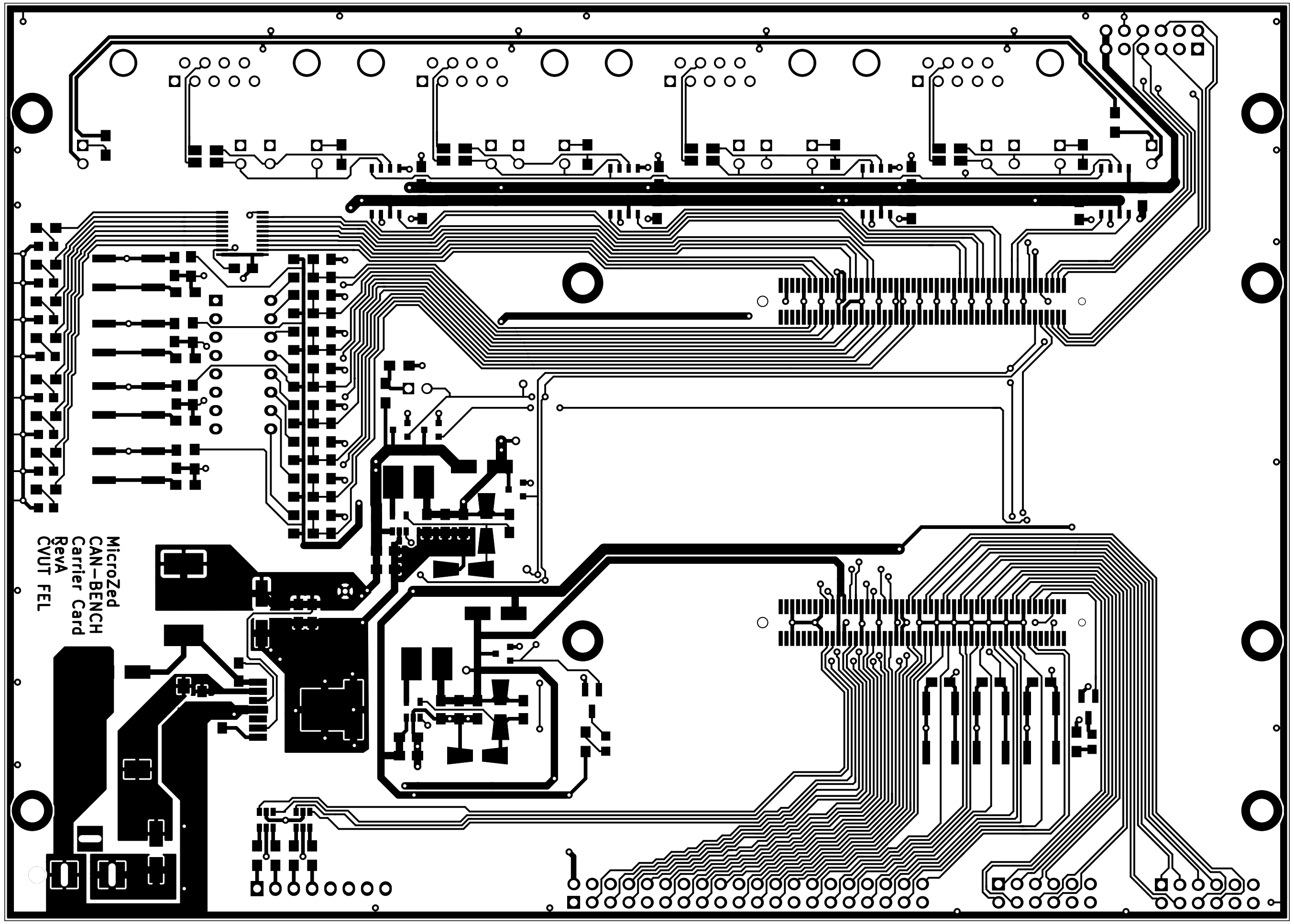
Sheet: /MicroHeader JX2/User I/O/
File: user-io.sch

Title: CANbus MicroZed CarrierCard

Size: A4	Date: 25.4.2016	Rev: RevA
KiCad E.D.A. kicad 4.0.2-stable		Id: 10/10

Appendix C

CAH-BENCH PCB Layout



FEE CTU

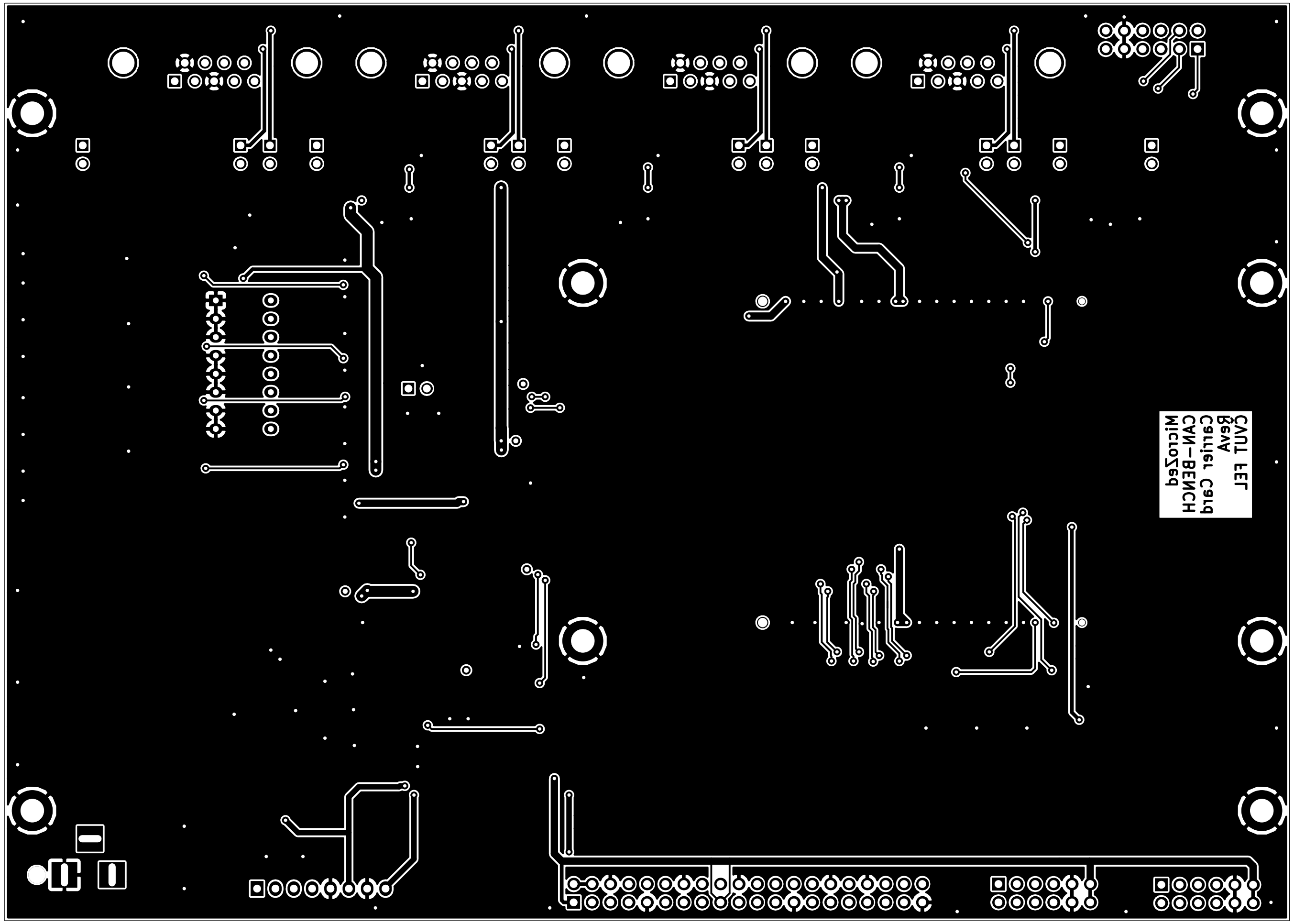
Sheet:

File: canbench-hw.kicad_pcb

Title: MicroZed CAN Carrier Board

Size: A4   Date: 20.4.2016   **Rev: RevA**

KiCad E.D.A. kicad 4.0.2-stable   Id: 1/1

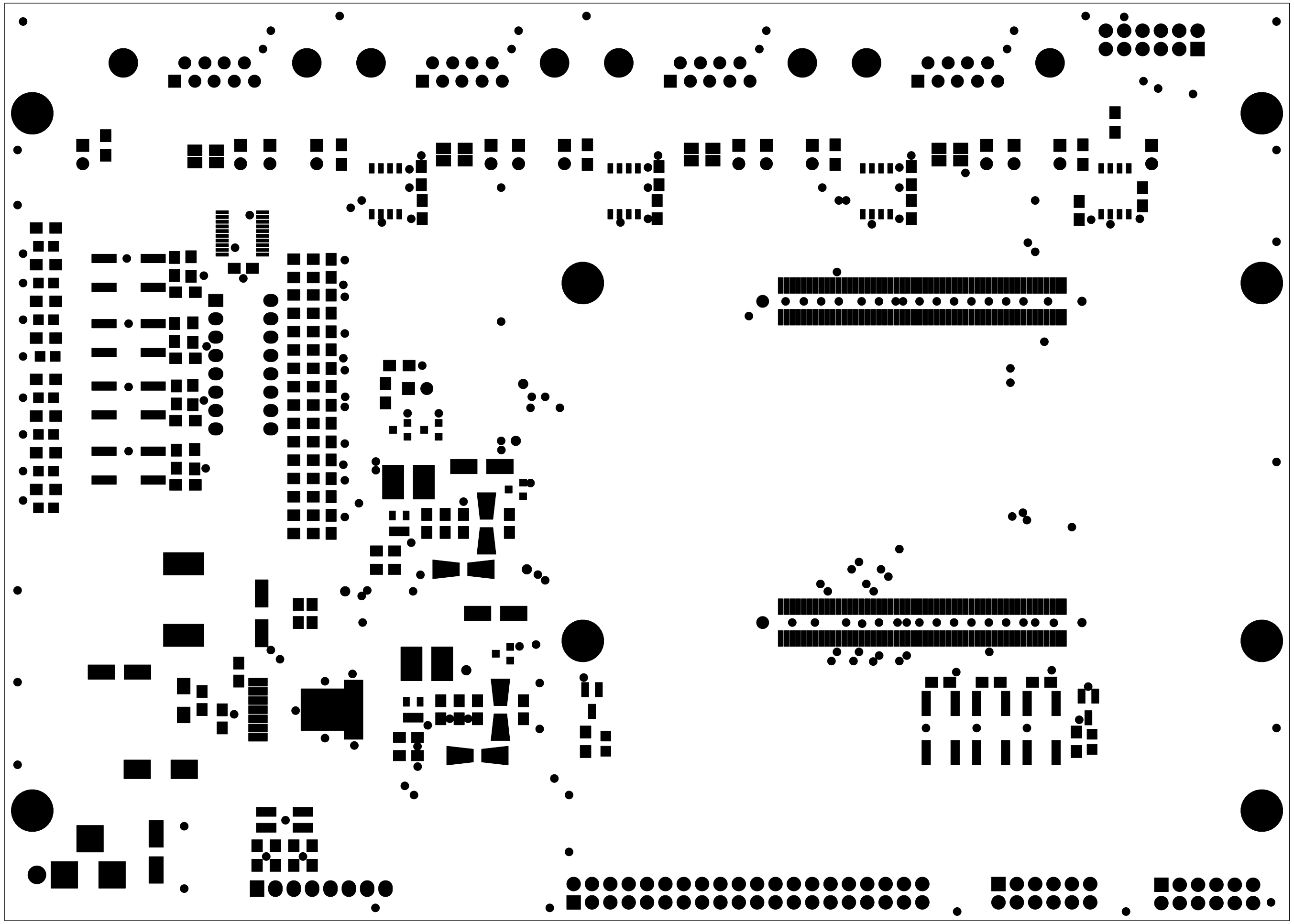


FEE CTU

Sheet:
File: canbench-hw.kicad_pcb

Title: MicroZed CAN Carrier Board

Size: A4 Date: 20.4.2016 **Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable Id: 1/1

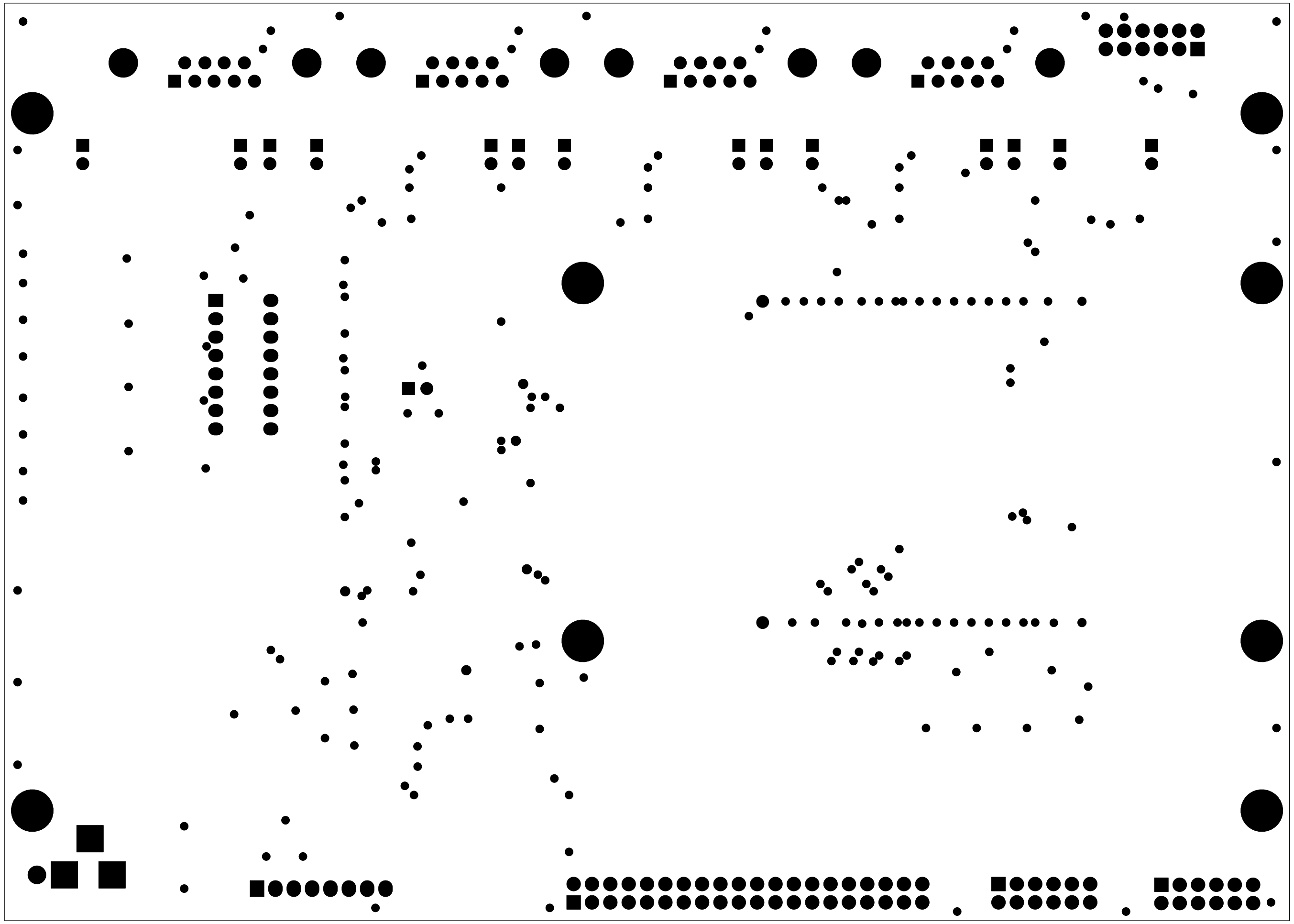


FEE CTU

Sheet:
File: canbench-hw.kicad_pcb

Title: MicroZed CAN Carrier Board

Size: A4 Date: 20.4.2016 **Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable Id: 1/1



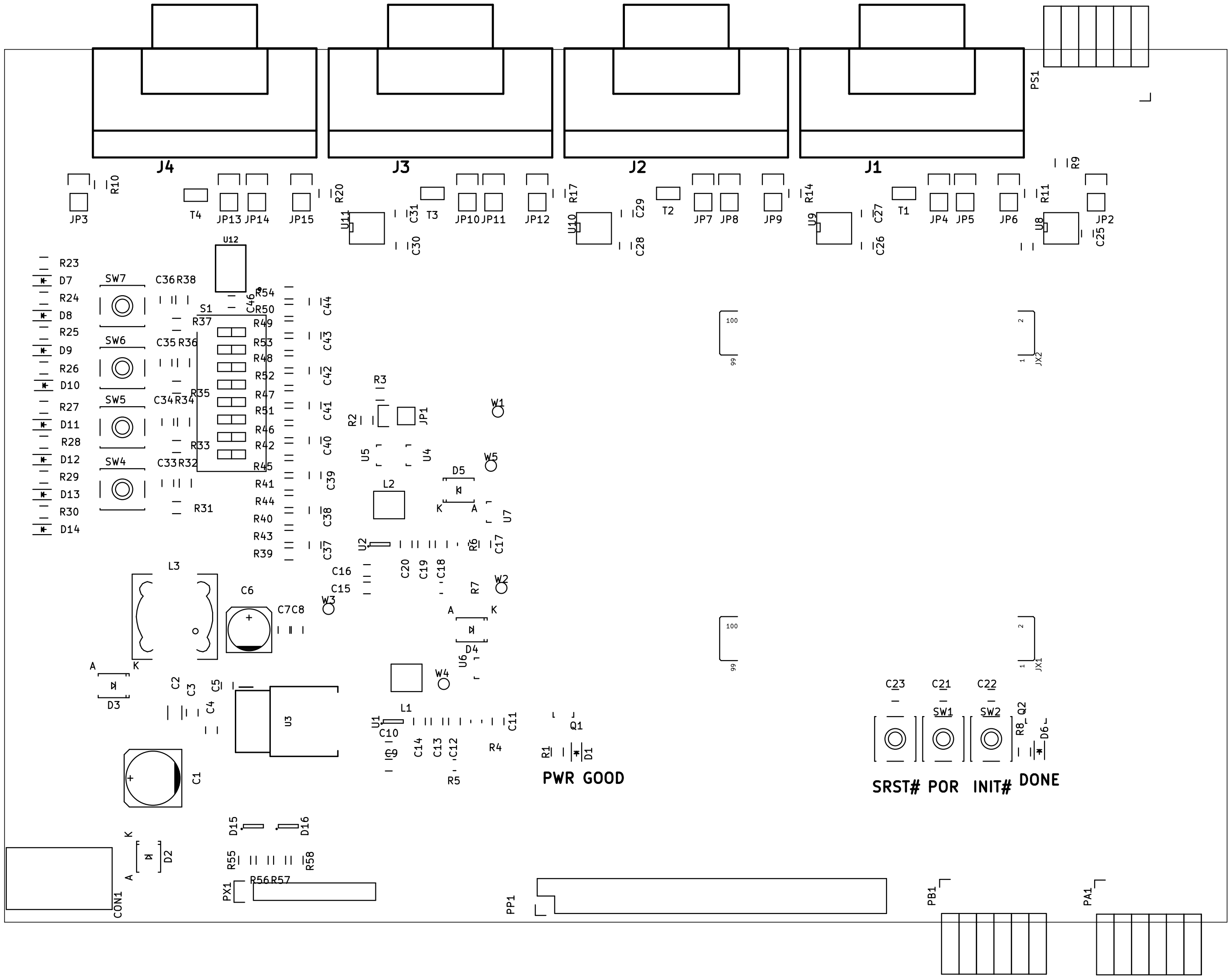
FEE CTU

Sheet:

File: canbench–hw.kicad_pcb

Title: MicroZed CAN Carrier Board

Size: A4	Date: 20.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2–stable		Id: 1/1



FEE CTU

Sheet:

File: canbench-hw.kicad_pcb

Title: MicroZed CAN Carrier Board

Size: A4	Date: 20.4.2016	**Rev: RevA**
KiCad E.D.A. kicad 4.0.2-stable		Id: 1/1

Appendix D

CAN-BENCH Bill of Material

Reference	**CNT**	**Value**	**Description**	**Footprint**
C11 C17	2	22p		C_0805
C1	1	220u	220u 35V	c_elec_8x10
C4 C5 C21 C22 C23 C33 C34 C35 C36 C37 C38 C39 C40 C41 C42 C43 C44	17	10n		C_0805
C2	1	10u	10uF 35V	C_1206
C3	1	470n	30V	C_0805
C6	1	220u	16V	c_elec_6.3x7.7
C7 C10 C14 C16 C20 C24 C25 C26 C27 C28 C29 C30 C31 C46	14	100n	6V	C_0805
C8 C9 C12 C13 C15 C18 C19	7	10u	6V	C_0805
CON1	1	2.5mm DC Barrel Jack Socket, FC681477		
D15 D16	2	USBLC6-2SC6		SOT-23-6
D1	1	LED_GREEN		LED_0805
D2 D3	2	SK34SMA-DIO	30V, 3A	DO-214AB
D4 D5	2	TME:SK13-DIO	10V, 1A	SMB
D6	1	LED_BLUE		LED_0805
D7 D8 D9 D10 D11 D12 D13 D14	8	LED_RED		LED_0805

Continued on next page

Table D.1 – continued from previous page

Reference	CNT	Value	Description	Footprint
J1 J2 J3 J4	4	D-SUB9 Female		
JP1 JP2 JP3 JP4 JP5 JP6 JP7 JP8 JP9 JP10 JP11 JP12 JP13 JP14 JP15	15	0.1" 1x2 Pin Header		
JX1, JX2	1	FCI BERGSTAK 100p 0.8mm Pitch Plug		61083_10x
L1 L2	2	DLG-0403-2R2	2.2u, >1.2A	Choke_SMD_0403
L3	1	DE1207-33	33uH, 3A	Choke_SMD_12x12mm
PA1 PB1 PS1	3	PMOD	0.1" 2x6 Socket Header	
PP1	1		0.1" 2x20 Pin Header	
PX1	1	CONN_01X08	8x screw terminal block 2.54mm	
Q1 Q2	2	BSS138		SOT-23
R12 R13 R15 R16 R18 R19 R21 R22	8	0R 250mW	Populate instead of T1 T2 T3 T4	R_0805
R1 R23 R24 R25 R26 R27 R28 R29 R30	9	560R		R_0805
R2	1	5K6		R_0805
R32 R34 R36 R38 R39 R40 R41 R42 R51 R52 R53 R54	12	3K3		R_0805
R3	1	2K4		R_0805
R31 R33 R35 R37 R43 R44 R45 R46 R47 R48 R49 R50 R55 R56 R57 R58	16	100R		R_0805
R4 R6	2	15K 0.1%		R_Uni_0805_1206
R5 R7	2	3K3 0.1%		R_Uni_0805_1206
R8	1	270R		R_0805
R9 R10 R11 R14 R17 R20	6	120R	0,25W	R_0805
S1	1	SW_DIP_x8		SW_DIP_x8_Slide

Continued on next page

Table D.1 – continued from previous page

Reference	**CNT**	**Value**	**Description**	**Footprint**
SW1 SW2 SW3 SW4 SW5 SW6 SW7	7	SW_PUSH		SW_SPST_EVQP0
T1 T2 T3 T4	4	DPWC1206-XXX	or DPWC0805-XXX or DNP	Choke_Dual_1206
U12	1	74HCT245PW		TSSOP20
U1 U2	2	TPS62260DDCTG4		SOT-23-5
U3	1	LM2676S-5.0		TO-263-7-TEXAS
U4 U5	2	MCP120T-450I/TT		SOT-23
U6 U7	2	MCP120T-315I/TT		SOT-23
U8 U9 U10 U11	4	MCP2562FD-E/SN		SOIC-8-N
W1 W2 W3 W4 W5	5	TEST_1P		
	15	JUMPER		

Appendix E

Official Assignment

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Martin Jeřábek

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation

Guidelines:

The goal of this thesis is to replace system for measurement of throughput and latencies of CAN devices connected to multiple CAN buses. An original software only system has been developed on request of Volkswagen Research to test Linux CAN gateway implementation and drivers.
Microsecond time resolution is required for new system.

1. Select appropriate FPGA platform for three-channel CAN bus tester.
2. Design CAN transceivers board which connects to FPGA SBC and provides at least two channels for messages time-stamping and at least one separate for messages traffic generation.
3. Adapt and implement FPGA logical design and Linux kernel drivers for designed solution.
4. Integrate solution into continuous integration tester running on server of Department of Control Engineering.

Bibliography/Sources:

[1] Sojka, M. - Píša, P. - Hanzálek, Z.: Performance evaluation of Linux CAN-related system calls. In Proceedings of the 10th IEEE International Workshop on Factory Communication Systems. Piscataway: IEEE, 2014, art. no. 6837608, p. 1-8. ISBN 978-1-4799-3235-1.

[2] Sojka, M. - Píša, P. - Špinka, O. - Hartkopp, O. - Hanzálek, Z.: Timing Analysis of a Linux-Based CAN-to-CAN Gateway. In Thirteenth Real-Time Linux Workshop. Schramberg: Open Source Automation Development Lab eG, 2011, p. 165-172. ISBN 978-3-00-036193-7.

[3] MicroZed Zynq™ Evaluation and Development and System on Module Hardware User Guide, Version 1.6, Avnet Inc./Zedboard.org, January 2015, available online: http://zedboard.org/product/microzed6787

Bachelor Project Supervisor: Ing. Pavel Píša, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 4, 2016