Development of a fault tolerant CAN bus interface based on the Raspberry Pi single board computer

---

Bachelor-Thesis

by

Timothy Dominik Widmer

at

Physik-Institut

Faculty of Science

University of Zurich

Supervisors

Prof. Dr. Ulrich Straumann
Dr. Arno Gadola
Dr. Achim Vollhardt

December 2014

Abstract

The Cherenkov Telescope Array (CTA) is a project run by a multinational research collaboration which plans to construct an array of ground based imaging atmospheric Cherenkov telescopes (IACT). The array will consist of multiple telescopes of three different sizes for a specific energy window each. The telescopes consist of a light reflecting parabolic mirror and a detection camera which lies in the mirror's focal plane. CTA is currently in the prototype stage.
The presented work concentrates on solving one of these problems. The CTA telescope camera for detection of the secondary Cherenkov radiation consists of many detection sensors, which need to be controlled and monitored by a slow control system. Using the flexible and reliable Controller Area Network (CAN) field bus system is a comprehensible approach. It offers minimal wiring costs and provides a wide range of hardware resources due to its widespread use in automobile industry.
This thesis describes the realization of a fault tolerant CAN bus interface using the combination of the multifunctional single board computer Raspberry Pi with commercial CAN device chips. In contrast to other commercially available CAN interfaces, this implementation is fully transparent and modifiable by the user. The different characteristics of the CAN protocol are presented and how arbitration and prioritization are realized. It is described further how to set up a CAN device using Linux software and how it is integrated into the Linux network stack. Different operation modes and their limitations are discussed at the end.

Contents

Chapter 1

Introduction

This chapter gives a short introduction in the very high energy gamma ray research and introduces the Cherenkov Telescope Array (CTA) project. The concept of FlashCam is described and why a field bus system was chosen to implement the slow control for a sub part of the FlashCam. At the end of the chapter there is an outline to give the reader an idea of the topics that are covered in this work.

1.1 Introducing VHE gamma ray telescopes

To obtain a better understanding of our universe, how it evolves and why things work the way they do, people are observing and mapping the sky since the earliest days of mankind. But visible light is by far not the only interesting part of the electromagnetic (EM) spectrum which is worth to be researched. The EM spectrum ranges over more than 35 orders of magnitude, from 1 feV (10^{-15} eV) up to almost 100 EeV (10^{20} eV). In particle physics, energy is measured in units of electron Volt (eV), where 1eV equals 1.6x10^{-19} Joule. The term 'gamma ray' is used for photons with energies starting from about 100 keV and the term 'very high energy gamma ray' (VHE) for the energy window from about 100 GeV up to 100 TeV.
Known sources of VHE gamma rays are pulsars, pulsar wind nebulae [2], supernova remnants [3] and active galactic nuclei [4]. As shown in fig. 1.1, there are still many unidentified gamma ray sources. Possible candidates are isolated massive stars or white dwarfs and accreting neutron stars [2]. This sources could be identified by the next generation of imaging atmospheric Cherenkov telescopes (IACT's), which are described below.
Of special interest for gamma ray detection is the atmospheric energy window starting from about 10 GeV up to 100 TeV. In this range an electro-

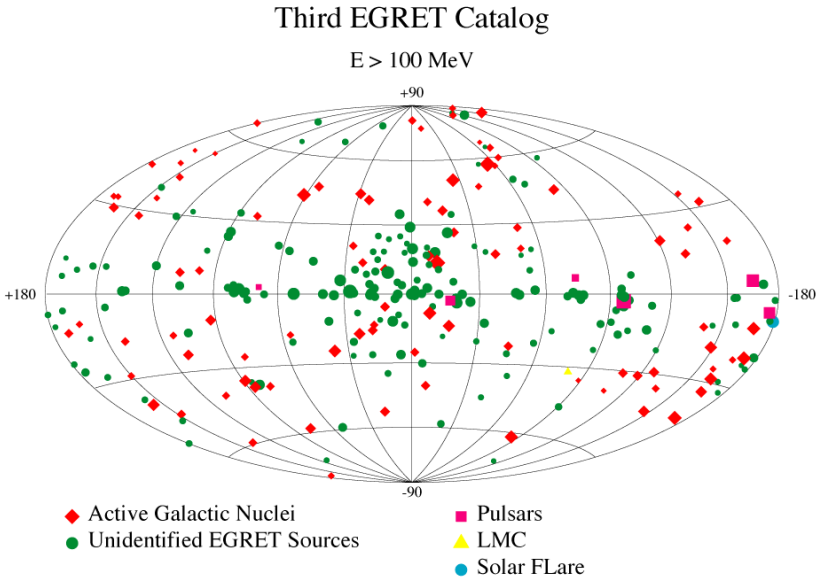


Figure 1.1: Shows an all-sky map of detected gamma ray sources by the EGRET telescope. LMC is the Large Magellanic Cloud. There are still many unidentified gamma ray sources. [1]

magnetic cascade is observable, shown in fig. 1.2, caused by an incoming gamma ray undergoing pair production, which by Bremsstrahlung and further pair production produces an electromagnetic shower [5]. If the produced charged particles travel faster than light does in the atmosphere, they will cause the atmosphere to radiate (see 'Cherenkov radiation' in [5]). This radiation, partly in the visible spectrum, ranges from ~300 nm (4.14 eV) to ~600 nm (2.07 eV). The radiated photons can then be detected by ground based telescopes. It is possible to reconstruct the original track of the incoming gamma ray to determine the direction the photon entered the atmosphere (fig. 1.2) and thus, where in the universe its origin is located. The CTA project uses the above described method. By using an array of telescopes, the imaging technique can be significantly improved. It is planned to build arrays at two sites containing telescopes of three different sizes, for different energy ranges. The sites will be located on the southern and northern hemisphere to cover the full observable night sky. CTA will help to identify the already mentioned unknown gamma ray sources and detect new ones. It is expected that CTA will answer the question on the production mechanism of relativistic particles and enlarge the knowledge about the extragalactic background light (EBL) [6] and gamma ray bursts [7].

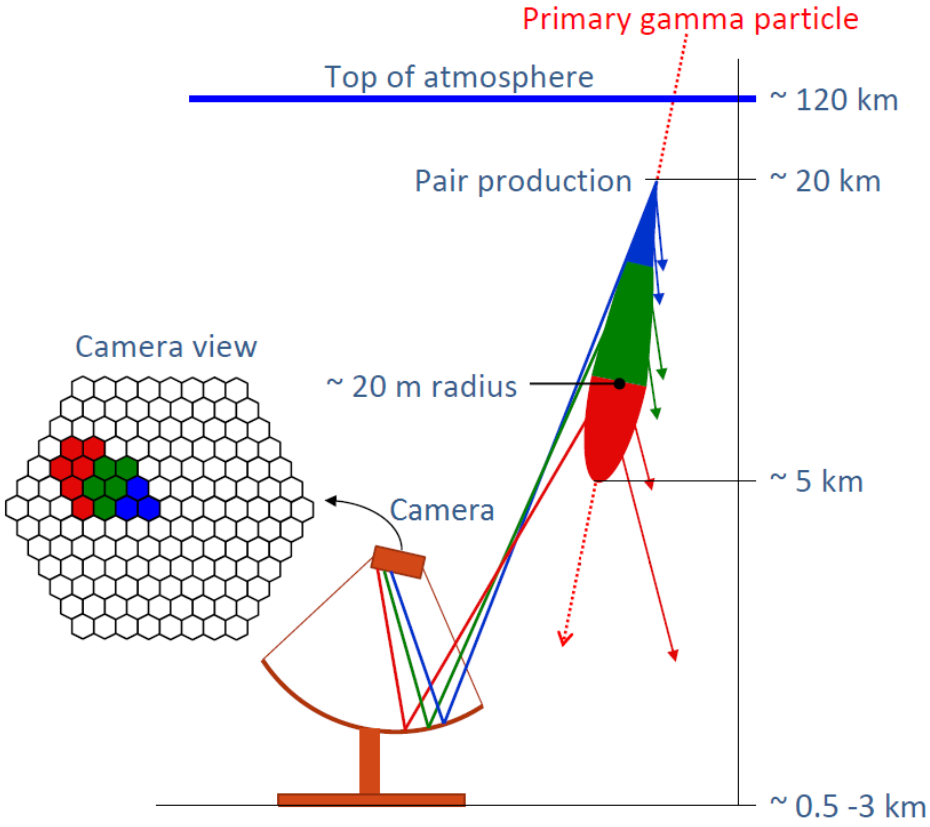


Figure 1.2: Cherenkov light detected by the camera of a telescope. The start of the EM cascade and the particle shower are shown in the figure. Every pixel shown in the camera view image is representing one photomultiplier. Distances are in kilometers above sea level. [8], modified.

.

1.2 CTA FlashCam

Figure 1.2 shows the camera on top of the telescope, which detects the Cherenkov light. This is done by the electronic system FlashCam, containing photomultiplier tubes (devices for photon detection) and readout electronics (fig. 1.3). The basic concept of the FlashCam is the continuous digitization and the modular design of the photon-detection electronics [8]. The detection with photomultiplier tubes and amplification of the signal is realized in the photon-detector boards (PDP) which are spatially separated from the digitization and triggering electronics part. We will focus on the PDP boards, because they are the important part for this work. To control

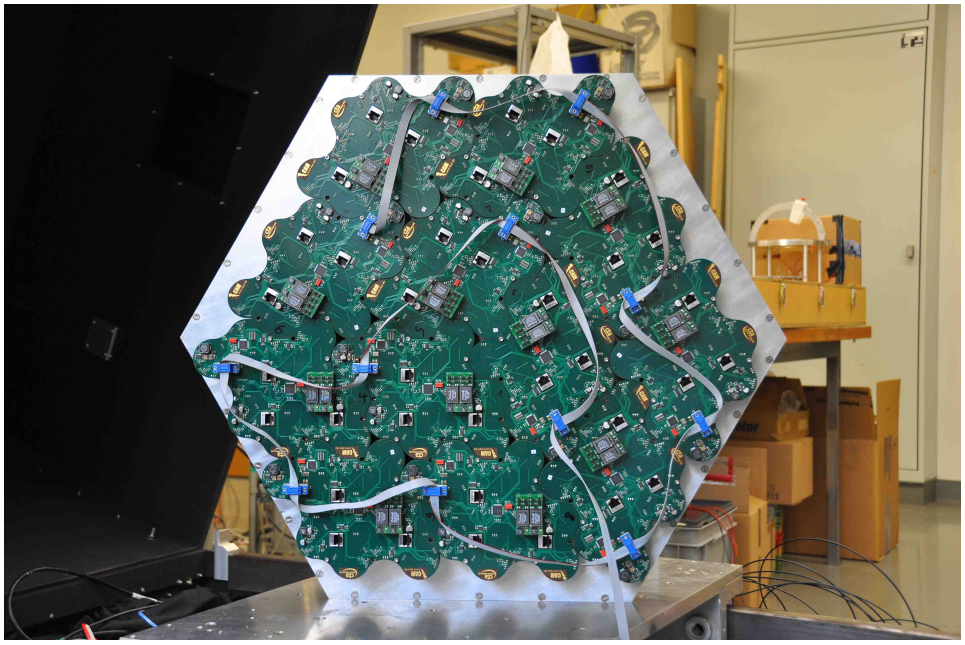


Figure 1.3: The back side of a scaled down prototype camera showing twelve PDP boards arranged in hexagonal shape. Due to the modular concept, expanding the arrangement to enlarge the number of pixel is not complex. The PDP boards are connected over one common ribbon cable (grey) to each other and to a control unit (not in the picture), which enables slow control and power supply. When signal transmission is realized in such a way, it is called a (field) bus system. Picture by A. Gadola.

a system like FlashCam, where wiring costs have to be reduced and new boards should be added in an easy way, a field bus system is a good choice.

1.2.1 PDP modules

Each single PDP board as shown in fig. 1.4 is connected to the common slow control network. Over this line the various parameters used to control the board, such as high voltage (HV) level or amplifier gain, can be requested and set. The board is able to send back the requested parameter values and report errors. The PDP boards implement a temperature sensor which can be read out by the described request method. The parameters are described in detail in [8] and [9]. Of special interest for this work are the HVbits, which enable high voltage supply for the single photomultiplier. This HVbits will later be set and read out.

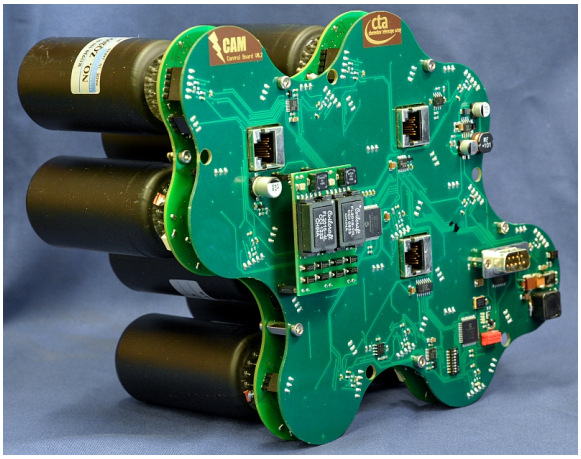


Figure 1.4: One PDP board containing twelve photomultiplier, amplifiers, high-voltage generation and sockets. The geometry is given due to the modular structure of the camera. Picture by A. Gadola.

1.3 Goal and Outline

The goal of this bachelor thesis was to implement an interface to a fault tolerant data transmission system using the single board computer Raspberry Pi. The data transmission system is realized by a field bus. This interface can be used to control the PDP boards of the CTA FlashCam. But the interface can be used as well in any other system, where the specific field bus protocol is used, that is described in the next chapter. The hardware was implemented by designing a circuit board which handles the physical tasks of data transmission as well as error handling and the prioritization of messages. A C library was developed, which is based on Linux network routines, for providing user friendly functions for transmission and reception of messages. Alarm indication of serious faults occurring in the system is implemented as well. A major part is handling and understanding the linux software drivers and how they interact with each other. Different modes of operation were tested on performance and stability. Remaining problems and possible solutions are discussed at the end and some conclusions are given.

Chapter 2

The field bus system and CAN

In a bus system, data transmission between several devices (called nodes) is realized over one common communication path. In a field bus, there are several peripheral units (sensor, actuators or even micro-processors) connected to each other. These devices can be spread over a wide area or 'field'. A field bus reduces wiring cost but increases complexity. Because only one node can use the common communication path at once, the communication has to be organized. This is called arbitration. There is a wide range of protocols which are defining the arbitration. In a master-slave based bus system, one master node controls the communication between himself and one or several slave nodes. The master gives the permission for communication. In a multi-master system, all nodes have equal rights and the arbitration and prioritization information is part of the transmitted messages (message based arbitration). The most common realization is the serial bus (see figure 2.1), but there are many others. For example the star topology, tree topology or ring topology (but all of them require one common communication path).
Many protocols have error detection routines, like the cyclic redundancy check (CRC), which is based on polynomial division and adds check sum bits to a message. That allows the receiving node to detect errors.

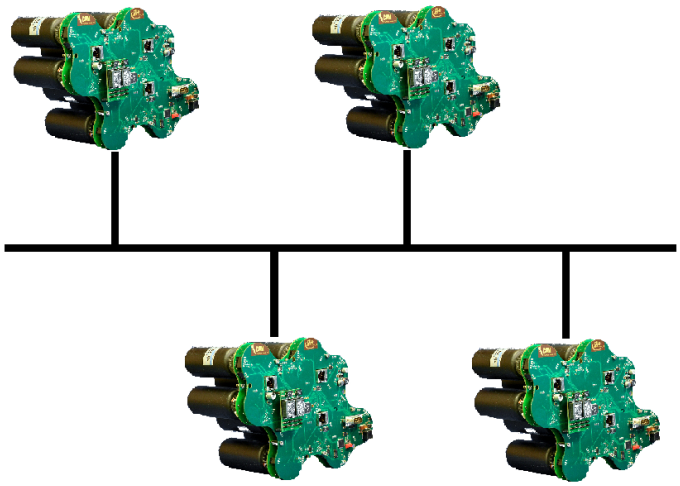


Figure 2.1: Serial bus topology: Multiple devices on one common communication path in series. The PDP modules described in section 1.2.1 were taken as example nodes.

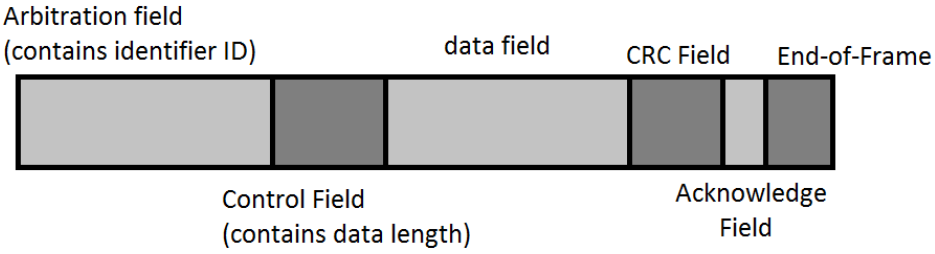


Figure 2.2: General CAN frame format. The message is divided into different parts. Most important is the arbitration field, containing the identifier, and the data field. The ID can represent one specific device of a node with several devices or it can represent the message's content.

2.1 Controller Area Network protocol

The Controller Area Network (CAN) is a serial field bus system with a message based protocol. The protocol is multi-master based. CAN was developed by the Robert Bosch GmbH and released in 1986. Developed to reduce wiring cost in automobile industry, today CAN is used also in elevator systems, shipbuilding, biomedical engineering and aerospace engineering. CAN protocol versions are revised and implemented by the CAN

in Automation (CiA) group.
Figure 2.2 shows the general structure of a CAN message, called frame. Each field is explained in detail in section 2.1.3.

2.1.1 System structure

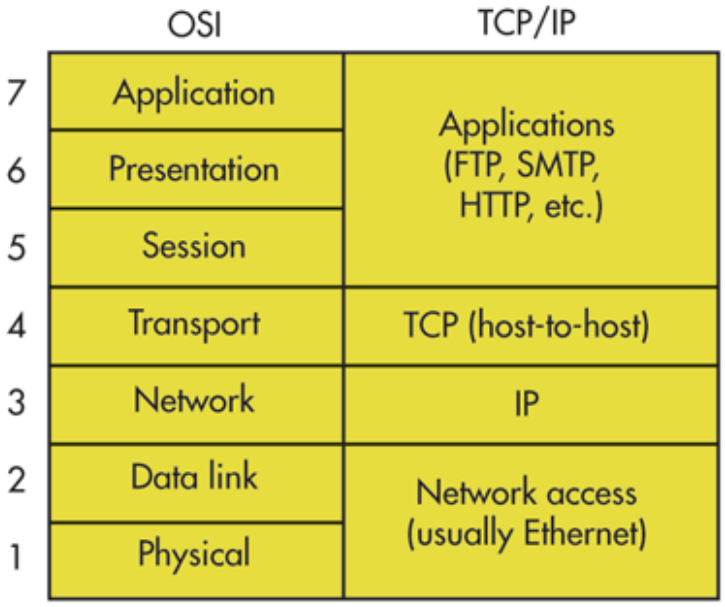


Figure 2.3: The seven OSI layers as a guideline for the structure of communication systems. The TCP/IP protocol is shown as an example. [10]

The Open Systems Interconnection model (OSI) partitions communication systems in different abstraction layers (fig. 2.3), for a better characterization and standardization. The CAN protocol can then be decomposed into three OSI Layers (fig. 2.4). Each layer consisting hardware and software parts with specified tasks.

Physical layer
This layer implements all the physical properties. The specifications, like the bit representation and the electrical aspects (currents, voltage levels) are defined in standards (see below). Wires and connectors have to be specified yet, but there are some de facto standards. One of these is the use and pinning of a 9-pin D-sub type male connector as implemented in the PDP boards (fig. 1.3). Except for wires, connectors and termination resistors, the main aspects of the physical layer are realized by a CAN transceiver.

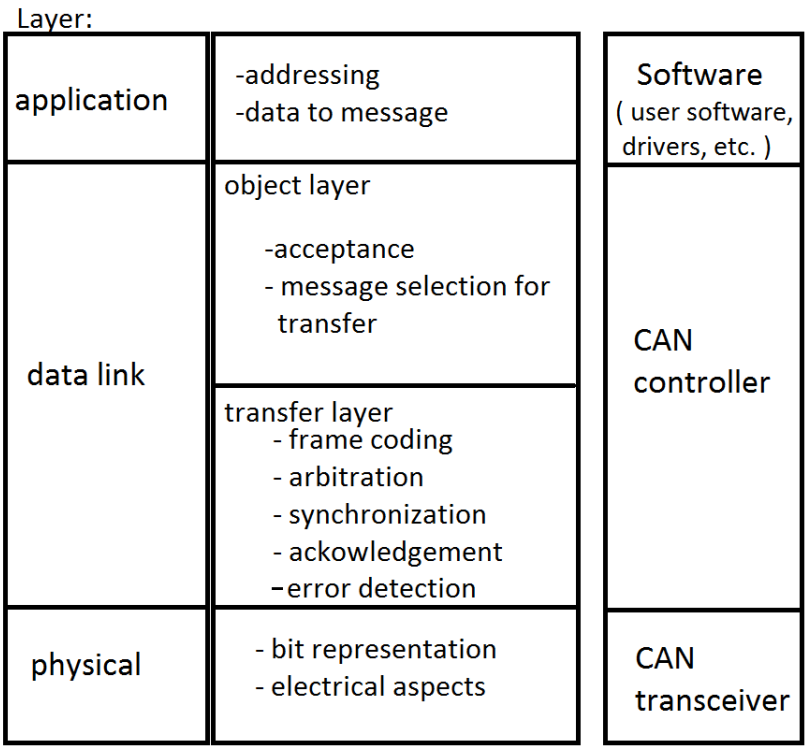


Figure 2.4: CAN implements three of the seven abstraction layers of the OSI. The data link layer is divided into the object and transfer layer, both implemented by the CAN controller. The physical layer tasks are mainly realized by the CAN transceiver, the data link layer by the CAN controller.

Data link layer
The data link layer consists of two sub layers:

- **Transfer layer**
  The transfer layer implements the bus protocol. It generates the message frame, does the arbitration/prioritization processes (see below) and monitors the bus state (idle/busy). The layer performs bit timing and synchronization (fig. 2.6), message validation, acknowledgment and error detection.

- **Object layer**
  This layer implements the message and status handling. It decides which message should be transferred. When receiving a message, this layer decides on the basis of the identification field (ID) to either reject or keep the message.

The data link layer is completely realized by a CAN controller.

CAN application layer (CAL)
In this layer, data, destination[1] and instructions from a user space program are merged to one message. Destination and instructions are represented a by a unique identifier. The ID represents a certain priority as well, the lower the ID, the higher the priority.

There are several specifications of CAN by Bosch and the International Organization for Standardization.
These **standards** are:

- CAN 2.0A, which describes the standard 11-bit identifier (ID) CAN format [11].
- CAN 2.0B, which describes the extended 29-bit ID CAN format [12].
- ISO 11898-1, which covers the CAN data link layer.
- ISO 11898-2, which covers the CAN physical layer for high speed CAN (10 kbit up to 1 Mbit).
- ISO 11898-3, which covers the CAN physical layer for low speed, fault tolerant CAN (10 kbit up to 125 kbit).

High speed CAN and low speed CAN differ not only in data rate, but also in voltage level, currents and bus fault performance and are therefore not compatible.

2.1.2 Data transmission, topology and arbitration

CAN is a real time prioritized protocol, it resolves collision by **bit arbitration** (fig. 2.5). Every node can start transmitting a frame, beginning with transmitting its ID. The message transmitting node reads simultaneously the bits back which are on the bus. When several nodes are writing bits on the bus, the dominant bit 0 always wins (a logical AND), the recessive bit 1 loses. When the node transmits a 1 (recessive bit), but receives a 0 (dominant bit), it realizes another node is transmitting a message with higher priority and stops transmission (called losing arbitration). After the higher prioritized message has been transferred, the node tries to submit its message again. So the arbitration is done during the transmission of

[1] This can be one or several nodes

the messages ID (identification). This means, each node must transmit a unique, distinguishable ID. The lower the ID, the higher the priority of the message, due to the fact that 0 is the dominant bit. The ID can be used to identify the device or the contend of the message (or both).
For example, assume we want to address the temperature sensor of one of the in section 1.2.1 described PDP modules. We reserve the first three bits for the devices of the PDP module and give the temperature sensor the device-sub-ID 011. The next three bits are reserved to address the PDP modules. Say the module-sub-ID is 100. Then the combined CAN frame ID is 00000 | 100 | 011 (for a normal 11-bit identifier). The first 5 bits are not used in this example. But we could also send a request to all the boards to transmit their data using just one frame with an ID, which addresses all the boards simultaneously. This makes CAN a very flexible protocol. In the

	Start Bit	ID 10	ID 9	ID 8	ID 7	ID 6	ID 5	ID 4	ID 3	ID 2	ID 1	ID 0
Node 15	0	0	0	0	0	0	0	0	1	1	1	1
Node 16	0	0	0	0	0	0	0	1	Stopped Transmitting			
CAN Data	0	0	0	0	0	0	0	0	1	1	1	1

Figure 2.5: Message based arbitration. 'CAN Data' is the actual bit on the bus, every node is reading this information. When a transmitting node detects disagreement, it stops transmission and tries again after reception of the actual message. Figure taken from Wikipedia.

CAN protocol the data is NRZ (non-return-to-zero) coded, which means two equal logical states are not separated by a delimiter. This is in conflict with the CAN's synchronization method. Due to the lack of a common clock signal, the CAN bus uses **bit timing** (fig. 2.6) for synchronization. Every node has an intrinsic bit time, which may differ from the actual time one bit stays on the bus, due to phase shifts between the oscillators or delays in the circuit. Every time the node detects a recessive-to-dominant (logical 1 to logical 0) edge, it resynchronizes to the actual bit time.
This is the reason **bit stuffing** is needed. After a maximum of 5 transmitted bits with equal polarity, one bit with inverse polarity is transmitted, which forces a recessive-to-dominant edge and allows the node to resynchronize (fig. 2.6). To reduce disturbances, one bit is transferred on two separate lines, CAN-(H)igh and CAN-(L)ow, the second line containing the inverted signal of the first. The transferred logical signal is the difference of the two lines, called a differential signal. Disturbances affect both lines mostly and

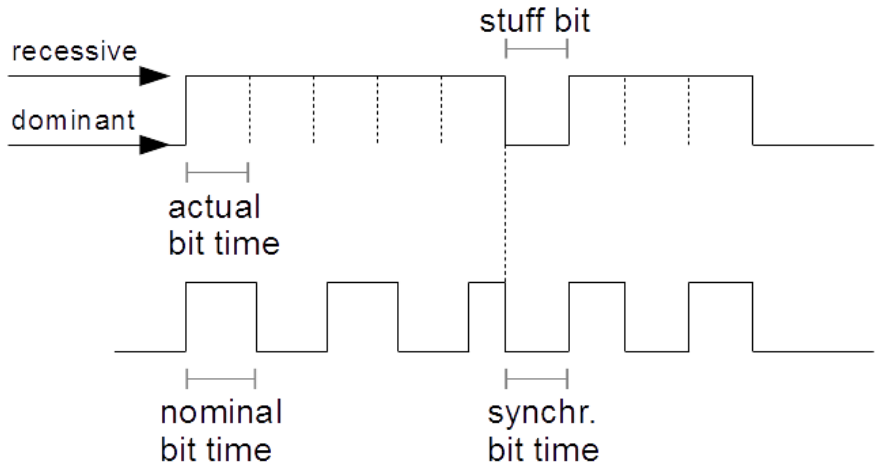


Figure 2.6: Bit timing and bit stuffing. Eight recessive bits should be transmitted. Every time a recessive-to-dominant edge is detected, the bit time is resynchronized. To 'force' such edges, bit stuffing is used.

keep the differential signal intact. For fault tolerant (low speed) CAN, the dominant state (logical 0) corresponds to a differential signal of 2.2 V and the recessive state (logical 1) corresponds to -5 V. For high speed CAN, the dominant state (logical 0) corresponds to a differential signal of 2 V and the recessive state (logical 1) corresponds to 0 V (fig. 2.7). The cyclic redundancy check (CRC) method is used for error detection.

To build a CAN network, at least two nodes are required. Each node containing a processor (CPU, microprocessor). In this work we used the Raspberry Pi's ARM processor, a CAN controller and a CAN transceiver. The controller stores the received bits serially the transmitter submits to it until the controller has received an entire message, which is then submitted to the processor (mostly interrupt triggered). On the other hand, if the controller gets a message from the processor, it submits the bits serially to the transceiver. The transceiver converts the bits received from the controller in voltage levels and vice versa. High speed CAN must have 120 Ω terminating resistors. Each low speed node has their own terminating resistors. Its values depend on the number of nodes (fig. 2.8), varying from 500 Ω to 16 kΩ. If an error occurs on the wire, the low speed CAN transceiver may switch into single wire mode (voltage level taken from CAN-L or CAN-H to ground) and transmits an error signal directly to the processor. This

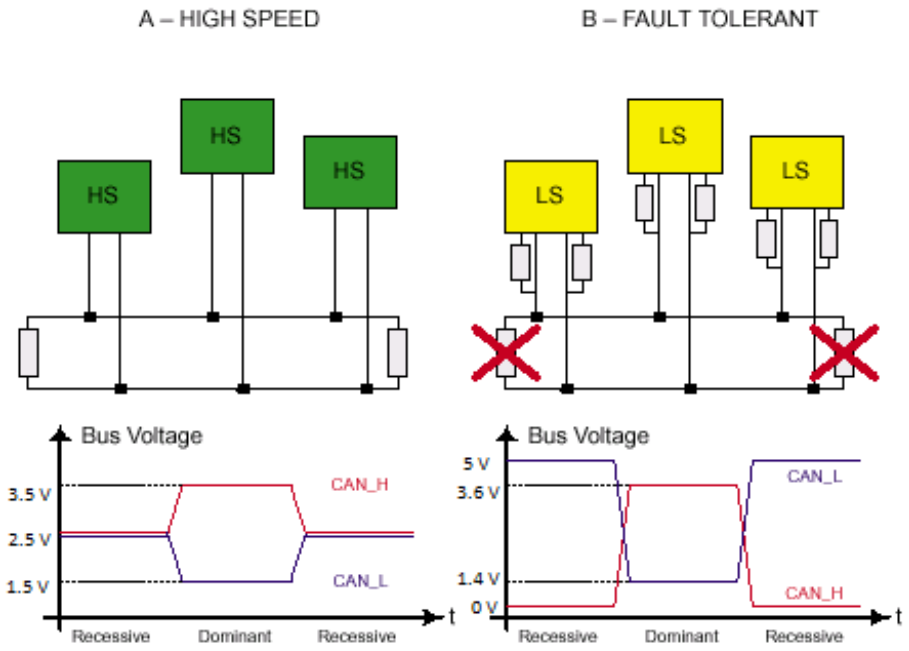


Figure 2.7: The voltage levels and resistor wiring for high speed and low speed CAN. Showed are the voltage levels for CAN-High (red) and CAN-Low (blue) and the voltage representation for recessive (logical 1) and dominant (logical 0) states. [13]

mode is called 'limping home mode'. When the corruption is removed, the transceiver resumes to differential signals and stops error signalization.

2.1.3 CAN frames

The messages are structured in so called frames. There are four different kind of frames:

- Data frames: Transports up to 8 byte of data (fig. 2.2, fig. 2.10 and fig. 2.9).
- Remote frames: Are used to request a data frame from another node (or a specific device of a node). This frame has an ID but no data section. Remote frames are not used in this work.

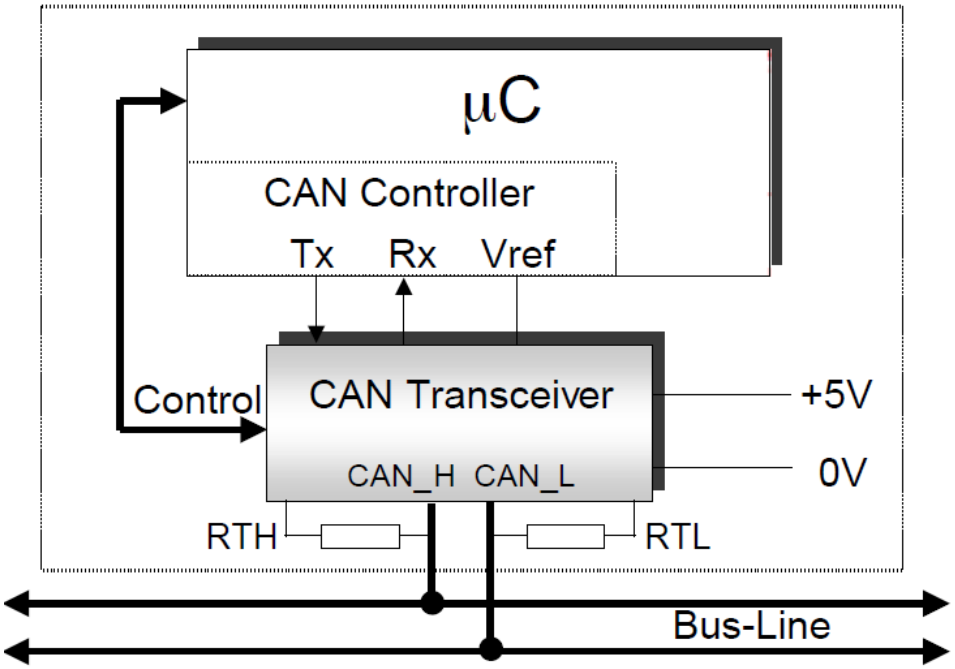


Figure 2.8: A node in a fault tolerant CAN bus system with terminating resistors and power supply. Connections to the controller for receiving (RX) and transmitting (TX) messages (from controllers view). The error signal is directly connected to the μC. [14]

- Error frame: Signalizes all nodes that an error was detected in the transmission. There are two kinds of error frames, active and passive ones.

- Overload frame: Forces a pause between remote and data frame. The CAN controller used in this work is fast enough, it will not produce overload frames.

If the remote transmission request (RTR) bit (fig 2.10) is set, the frame is a remote frame. No data will be transmitted in a remote frame. The remote frame can be used for fast readout of a device's data. Looking at our example in section 2.1.2, this means we can request the data of a PDP's temperature sensor, without sending unnecessary data bytes.

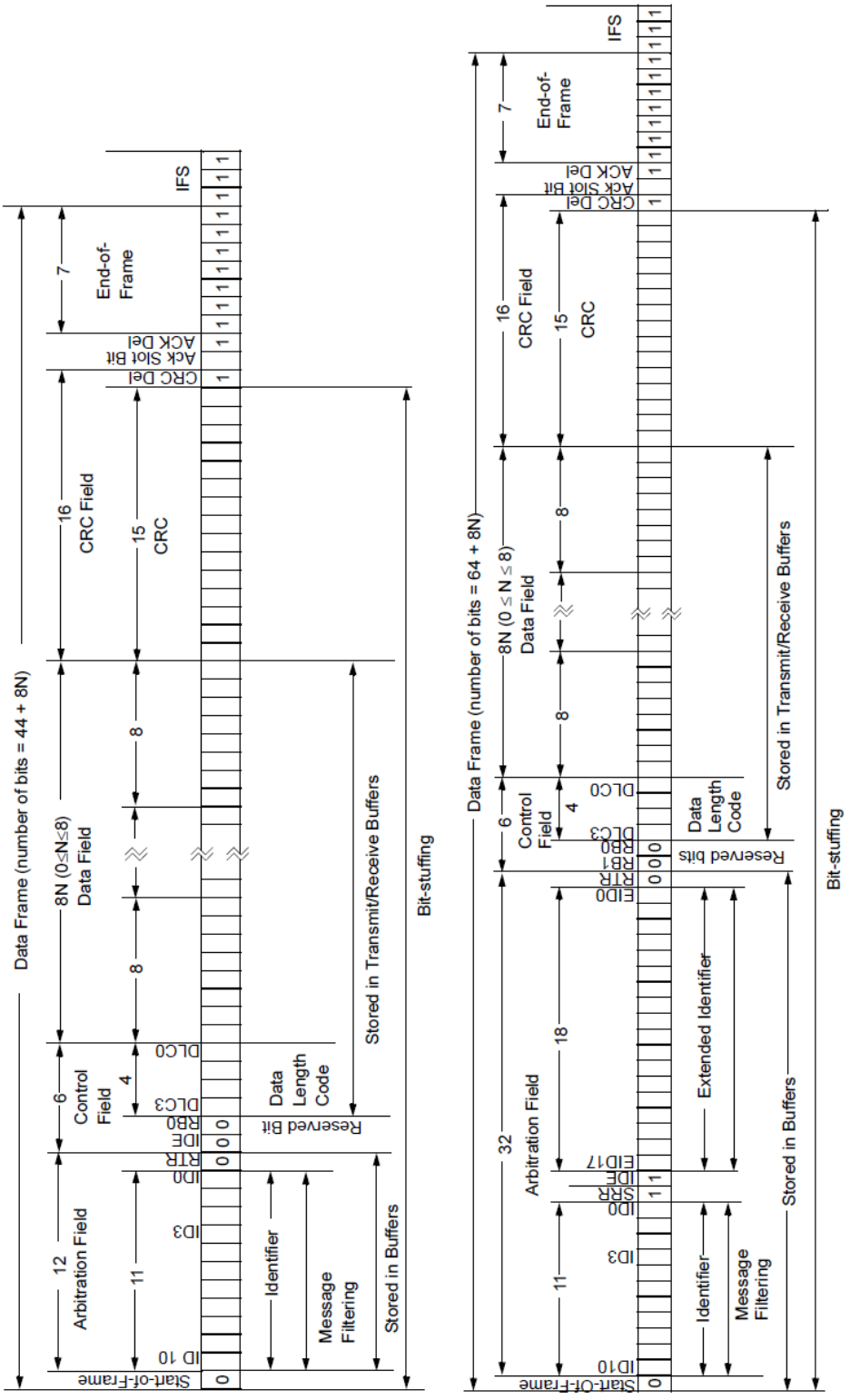


Figure 2.9: A normal CAN frame and a extended CAN frame. A normal frame has an 11-bit identifier, an extended frame a 29-bit identifier. [15]

normal frame			extended frame	
logical state	Bits	bit type	Bits	logical state
	44 +8N		64 +8N	
0	1	Start of Frame	1	0
0/1	11	Identifier	11	0/1
0	1	RTR (for normal)	-	-
-	-	SRR	1	1
0	1	IDE	1	1
0	1	RB0 (normal)	-	-
-	-	Extended Identifier	18	0/1
-	-	RTR (for extended)	1	0
-	-	RB1	1	0
-	-	RB0 (extended)	1	0
0/1	4	Data length code	4	0/1
0/1	8N	Data fied	8N	0/1
0/1	15	CRC field	15	0/1
1	1	CRC delimiter	1	1
0/1	1	Acknowledge Slot Bit	1	0/1
1	1	Acknowledge delimiter	1	1
1	7	End of Frame	7	1
1	3	Inter Frame Space (not part of frame)	3	1

RTR = Remote Transmission Request
SRR = Substitute Remote Request
IDE = Identifier Extension
CRC = Cyclic Redundancy Check
RB = Reserved Bit
N = Number of Bytes in Data Frame
0/1 = Both logical states possible

both frames
only normal
only extended
different values

Figure 2.10: The bits and structure of a normal data CAN frame compared to a extended data CAN frame.

When an error is detected in the frame, nodes are able to transmit **error** frames. Depending in which state they are, active error state or passive error state (fig. 2.11), they send 6 dominant bits (active state), which interrupts the message immediately, or 6 recessive (passive state) bits, which will not be noticed by the other nodes. In error active mode, this will violate the bit-stuffing rule (fig. 2.6) and provoke the other nodes to send an error frame as well. At the end, all nodes are sending 8 recessive error delimiter bits (fig. 2.12). Each controller has two error counter implemented, one for reception message errors and one for transmission message errors. The controller is initially in error active mode. If one of the two counters reaches the value of 128, the controller switches to error passive mode. If one of the counters reaches 255 it switches to "bus off" mode. The node takes no longer part in the transmission. If the bus is idle for 128 x 11 bit times it automatically switches back to error active mode.

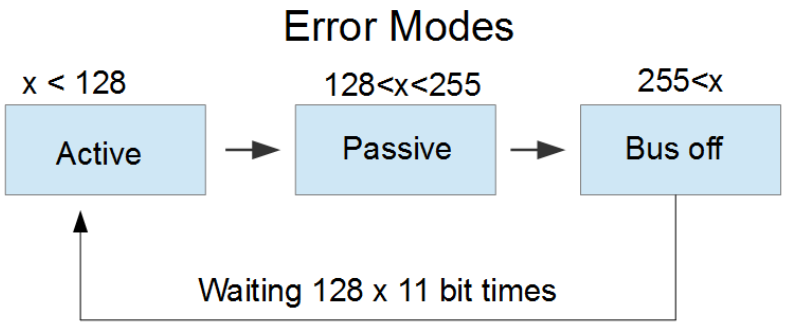


Figure 2.11: The CAN controller has three error modes. By default it is in the error active mode. If one of the internal error counters (represented by x in the figure) reaches a certain threshold, the controller switches to the next error state. After being switched off, the controller waits for 128 x 11 bit times before it restarts.

When a message successfully is transmitted, every node is setting the ACK bit on a dominant level, overwriting the recessive level of the sending node. That is the end of a transmission, after three recessive inter space bits, the next message can be transmitted.

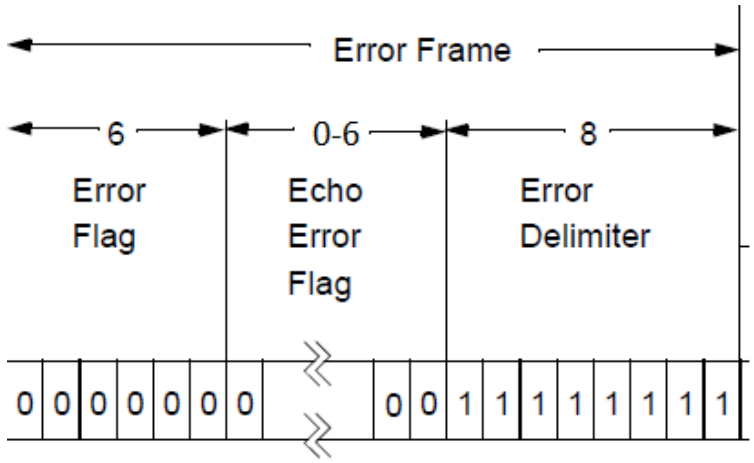


Figure 2.12: An (active) CAN error frames contains no data or ID. It holds between 6 and 12 dominant bits, depending on the time the other nodes realize there is an error frame on the bus. [15]

Chapter 3

Hardware

This chapter describes all the hardware components which are used to realize a CAN Interface. Via the CAN Interface all parameters of the photon detector boards are transmitted. One can send messages to request the status information of the PDP boards and receive these information. The chapter starts with the commercial available Raspberry Pi computer and describes some of its properties. The CAN protocol is implemented using the CAN controller MCP2515 and the fault tolerant CAN transceiver TJA1055 (fig. 3.1). All these parts are used in chapter 5 to build the CANPi Interface board.

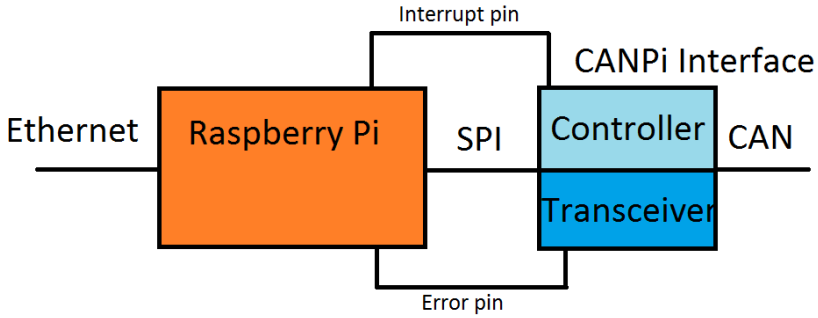


Figure 3.1: Structure of the CAN interface using the Raspberry Pi single board computer, the CAN controller MCP2515 and the CAN transceiver TJA1055. Communication between RasPi and controller uses SPI. The direct communication path via interrupt pin (controller) and error pin (transceiver) is showed as well.

3.1 Raspberry Pi single board computer

The Raspberry Pi (RasPi) is a single board mini computer developed in the UK by the Raspberry Pi foundation. It is based on the Broadcom BCM2835[1] system-on-chip which includes an ARM processor. For this work important is the Serial Peripheral Interface (SPI) [16] and the General Input Output Pins (GPIO) of the BCM2835. SD cards are used for booting an operating system and store memory. The RasPi needs a power supply of 5 V and at least 600 mA which is connected via micro USB. If some peripheral devices are connected, as in our case, a power supply of 1200 mA is recommended. The RasPi contains 26 low-level peripheral pins, called GPIO Interface, which are connected directly to the BCM2835 chip's GPIO pins (fig. 3.2). With the installed secure shell (SSH) client, it is possible to access the RasPi over Ethernet. The RasPi uses Linux-kernel based operating systems. For this work, the manufacturer's recommended Linux distribution Raspbian (based on Debian) is used.

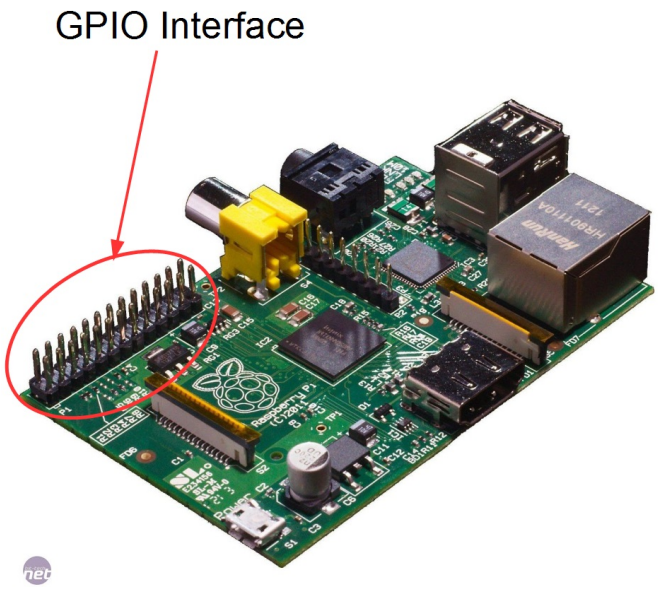


Figure 3.2: Raspberry Pi model B with 26 GPIO pins. This pins are used to communicate with the controller using a serial peripheral interface (SPI) bus. [17]

[1] To avoid confusion, BCM2708 is the device familiy name and BCM2835 the only implementation for Linux. That's the reason for the even more confusing fact, that there are drivers called bcm2835 as well as bcm2708.

3.2 CAN controller MCP2515

The MCP2515 controller chip implements the in section 2.1 described CAN protocol up to 1 MBit/s transmission speed. It handles the transmission of normal (11-bit ID) and extended (29-bit ID) CAN frames and works both with high speed (1 Mbit/s) and low speed (125 kbit/s) CAN transceivers. This CAN controller includes two reception buffer and three transmission buffer. The controller supports a loop-back mode. In this mode, the CAN frames are not transmitted over the CAN bus, but handled as been received from the bus and then transmitted over the SPI back to the RasPi (ideal for testing).
The device consists of three blocks shown in figure 3.3.

The **CAN module** handles the transmission of messages. It first loads the message from the RasPi, which was transmitted over SPI, into a buffer and waits for the command by the RasPi to transmit the message over the CAN bus (fig. 3.4). This 'start of transmission' command can be transmitted over the SPI interface or over the transmit enable pins. Any messages received over the CAN bus are checked for errors with the CRC, see chapter 2.

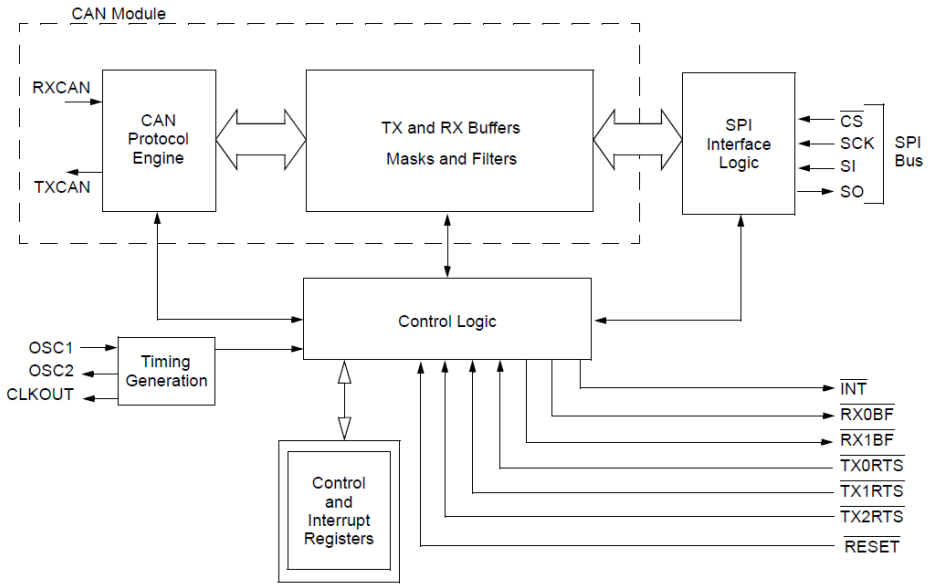


Figure 3.3: The MCP2515 CAN controller consists of three modules and 18 pins. [15]

The **control logic** block controls the setup and operation of the other blocks. It handles the various GPIO pins as well. There is one GPIO interrupt pin ($\overline{\text{INT}}$)[2] which is set low when a valid message has been transmitted over the CAN bus. The controller expects an acknowledgment by the RasPi to set the $\overline{\text{INT}}$ high again. If the appropriate register bit is not cleared by the RasPi (due to some interrupt handling problem) the controller is unusable and must be restarted. This will be a problem in multi bus operation mode (see sec. 6.1.2 and sec. 6.4). There are three special pins to initiate immediate transmission (TXnRTS), but their usage is optional.

The **SPI protocol block** is the interface to the Raspberry Pi. It uses usual SPI read and write commands to access the controllers registers as well as special commands like reset or request-to-send. There are four pins reserved for SPI (see fig. 3.5), the clock pin (clk), the master-input-slave-output (MISO) pin, the master-output-slave-input (MOSI) pin and the chip select (CS) pin, sometimes slave select (SS) called, to switch between multiple controllers on the same SPI bus.
CAN does not have a common clock wire, every controller needs a separate clock to perform bit timing (see 2.1.2). The MCP2515 controller has two pins to be connected to a oscillator. In this work a 16 MHz crystal is used.

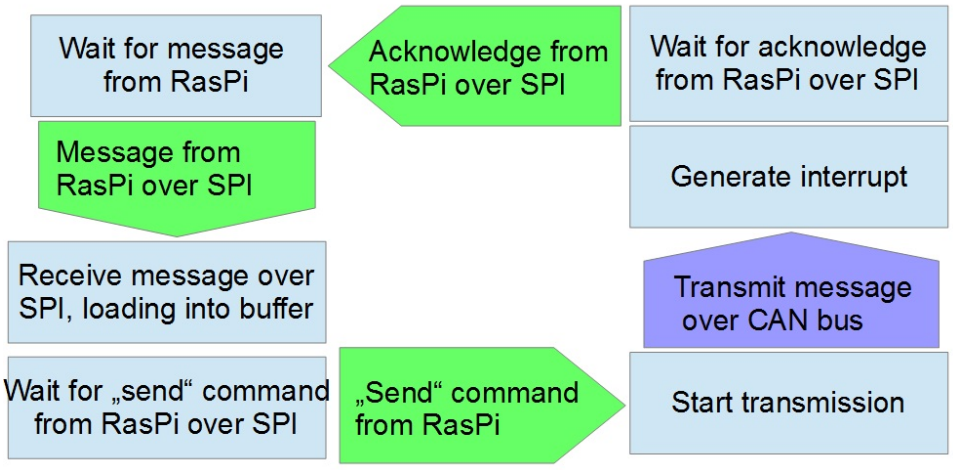


Figure 3.4: Transmission of a CAN frame from the controller's perspective. The controller waits for initiation of message transfer by the RasPi. After successful transmission, the controller generates an interrupt, which has to be acknowledged by the RasPi.

[2] notice that the signal is inverted

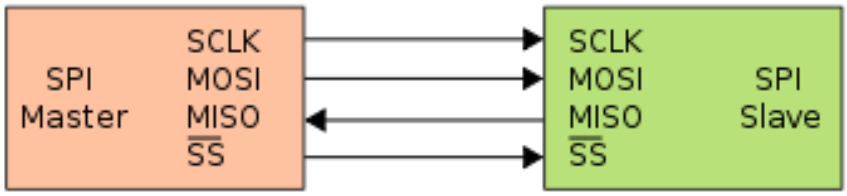


Figure 3.5: A master and a slave in a SPI bus. With a GPIO interrupt pin, the slave contacts the master (not shown in the figure). Figure from Wikipedia.

3.3 Fault tolerant CAN transceiver TJA1055

The fault tolerant (low speed) CAN transceiver TJA1055 implements the in Section 2.1 mentioned physical layer. It supports a bus speed up to 125 kB/s and up to 32 nodes connected to the bus. In case of a bus failure it may switch to single wire mode (but must not, fig. 3.6) , and still works. In this case, the transceiver uses an error ($\overline{ERR}$) pin to inform the RaspPi about the corruption. In figure 3.7 all possible bus failures, which can be dealt with, are listed.

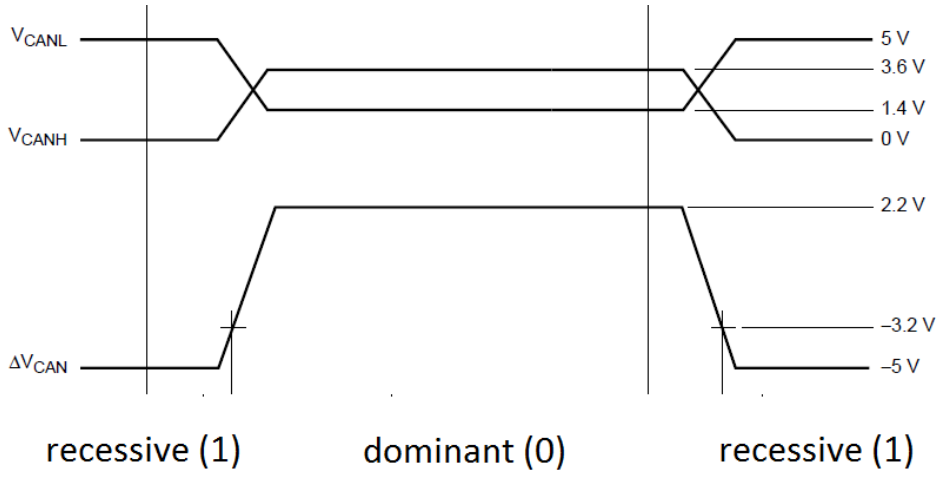


Figure 3.6: CAN-L and CAN-H voltage levels for low speed CAN. A recessive state is a logical 1, a dominant state a logical 0. [18]

Failure	Description	Termination CANH (RTH)	Termination CANL (RTL)	Receiver mode
1	CANH wire interrupted	on	on	differential
2	CANL wire interrupted	on	on	differential
3	CANH short-circuited to battery	weak[1]	on	CANL
3a	CANH short-circuited to V_{CC}	weak[1]	on	CANL
4	CANL short-circuited to ground	on	weak[2]	CANH
5	CANH short-circuited to ground	on	on	differential
6	CANL short-circuited to battery	on	weak[2]	CANH
6a	CANL short-circuited to V_{CC}	on	on	differential
7	CANL and CANH mutually short-circuited	on	weak[2]	CANH

[1] A weak termination implies a pull-down current source behavior of 75 μA typical.

[2] A weak termination implies a pull-up current source behavior of 75 μA typical.

Figure 3.7: Bus failures, which the transceiver TJA1055 is able to detect. Depending on the failure, the transceiver changes the termination resistor CAN-H or CAN-L and may switch to single wire mode. [18]

Chapter 4

Software

This chapter describes the software needed to set up SPI communication with a CAN controller on the Raspberry Pi. SPI is used for transmitting CAN frames from user space applications to the CAN controller. The development of the software library flashCAN is described in section 5.3.

4.1 Used kernel modules

The main modules[1] (fig. 4.1) required are already integrated in the latest linux kernel. They are:

- The CAN controller driver mcp251x.ko.
- The SPI master driver spi-bcm2708.ko for the SPI provided by the chip BCM2835 on the RasPi.
- The SPI driver spidev.ko, which allows user space access to the SPI devices.
- The SPI configuration tool spi-config.ko.
- The modules can.ko and can-raw.ko, which are described in section 4.3

The only module not yet integrated in the kernel is the spi-config.ko module, which has a pull request on Github and will probably be integrated in future versions. This tool is a program for registration of an SPI device to the SPI master driver and set the device's parameter. Another way is to add

[1] A software module is a computer program, which can be loaded into a running kernel to extend its functionality, for example hardware driver are often modules. In linux, kernal modules have a *.ko* ending

the code for a new device directly in the source code of the spi-bcm2708.ko master driver and recompile the driver together with the kernel.
The software library SocketCAN , which is a Low Level CAN Framework (LLCF) by Volkswagen Research [19], is already integrated in the Raspian kernel and allows transmission of CAN frames from user space via sockets to the CAN controller. The controller is then handled as a network device.

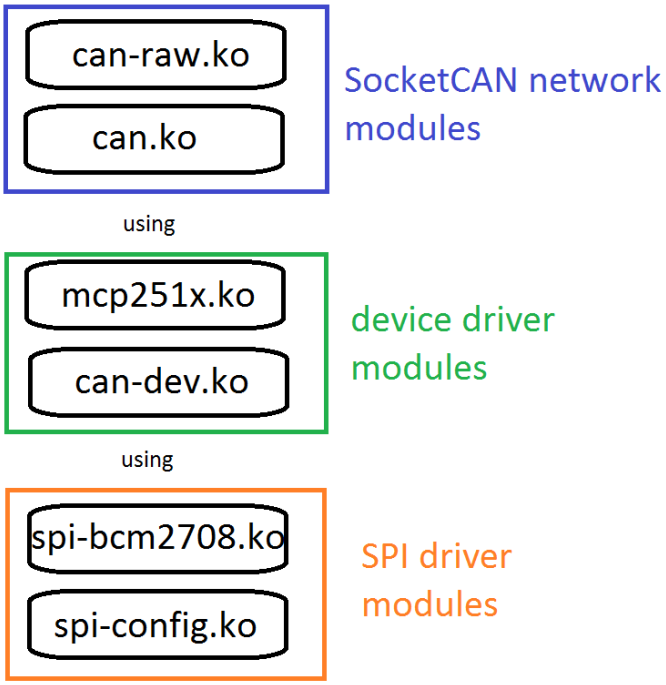


Figure 4.1: Kernel modules required for the CAN interface and their dependencies.

4.2 The rpi-can software package

This software package contains the original and additional drivers and the configuration tool spi-config, as well as the SocketCAN programs. As already mentioned, spi-config makes it much easier to install SPI devices than to recompile the kernel. The package can be downloaded at [20]. There are pre-compiled versions for Raspian 3.12.20+, 3.12.22+, 3.12.26+ and 3.12.28+. Alternatively the driver code and makefiles can be downloaded using the linux program Git and compiled with the actual kernel version. In this way modification of the drivers is realized. Instructions of

how to build the modules, meaning adding them to the kernel and compiling them together with the kernel, can be found here [21] and here [22].

4.2.1 SPI master driver spi-bcm2708

The SPI master driver spi-bcm2708 runs the SPI interface of the BCM2835 chip on the Raspberry Pi. It has to be loaded into the kernel at run time, using the linux command:

```
modprobe spi-bcm2708
```

The module enables SPI access. With the program spi-config new SPI devices are added. The spi-bcm2708 module is on a list in the file:

```
/etc/modprobe.d/raspi-blacklist.conf
```

which prevents loading modules at boot time. If the module should be loaded at boot time, it has to be removed from the list and the command to load it added to the file:

```
/etc/modules
```

4.2.2 Software tool spi-config

The tool spi-config allows to register a connected SPI device, for example as a "can" device, and initialize it at run time with:

```
modprobe spi-config devices=\
bus='':cd='':modalias='':speed='':gpioirq='':mode='':pd=''\
:pdsX-y='':pduX-y='':force_release
```

When more than one device should be added, they must be divided by a comma. The following parameters have to be defined when loading the module:

- bus: Specifies the bus, only bus number 0 can be used on the RasPi.
- cs : Selects the chip select pin. Two GPIO pins are reserved for chip select.
- modalias: The alias of the SPI device, mcp2515 in our case.
- speed: Sets the bit rate of the SPI bus.

- gpioirq: Selects the GPIO pin for interrupts.
- mode: The SPI mode, mcp2515 supports 0 and 3 [16].
- pd: Defines how much memory for the 'platform data' (initialization data) has to be allocated.
- pdsX-y or pduX-y: Defines the value of the platform data at a certain offset, see further explanations below.

The driver mcp251x for example has one variable in its platform data (pd) with a size of 4 Byte, so we have to allocate 4 Byte of memory. To address those variables we use $pdsX-y$ and $pduX-y$. Here pds is a signed variable, pdu a unsigned. X stands for the size of the variable in bit (8,16,32 or 64) and y for the offset in byte. See section 5.4 for an application example.

4.2.3 CAN controller driver mcp251x and can-dev

The kernel module can-dev.ko is a general driver for all CAN devices. It provides functions which all CAN controllers have in common, like bit timing and bit stuffing (section 2.1.2). The driver mcp251x is a specific driver for the CAN controller MCP2515 and its older version MCP2510. It knows the address of the internal registers of the MCP2515 chip and is able to *read* and *write* those registers using the SPI. The driver activates the device by initializing the registers and handles the reception (RX) and transmission (TX) of messages over the SPI. The mcp251x only supports the interrupt method to check if a message was transferred or received. Polling is not implemented.

4.3 CAN protocol family SocketCAN

SocketCAN is a Controller Area Network protocol family for Linux using the Berkeley socket application programming interface (API). That means SocketCAN uses the Linux network stack to implement CAN devices as network interfaces. This allows programmers to use Linux system calls, like *bind()*, *read()* and *write()* to address the CAN device. In figure 4.2 the linux network stack is shown (compare OSI model 2.3). In the protocol agnostic interface the socket API is defined. SocketCAN implements a new protocol family in the network protocol layer to address CAN device. A new CAN controller device driver has to register itself as a network device.

In this way CAN messages can be passed from the hardware device to the network layer. The socket principle allows to handle messages from different sources (for example SPI, Ethernet or CAN) with the same user space routines (fig. 4.3).
SocketCAN provides a wide range of programs for monitoring CAN traffic and generation of CAN frames for test purpose.

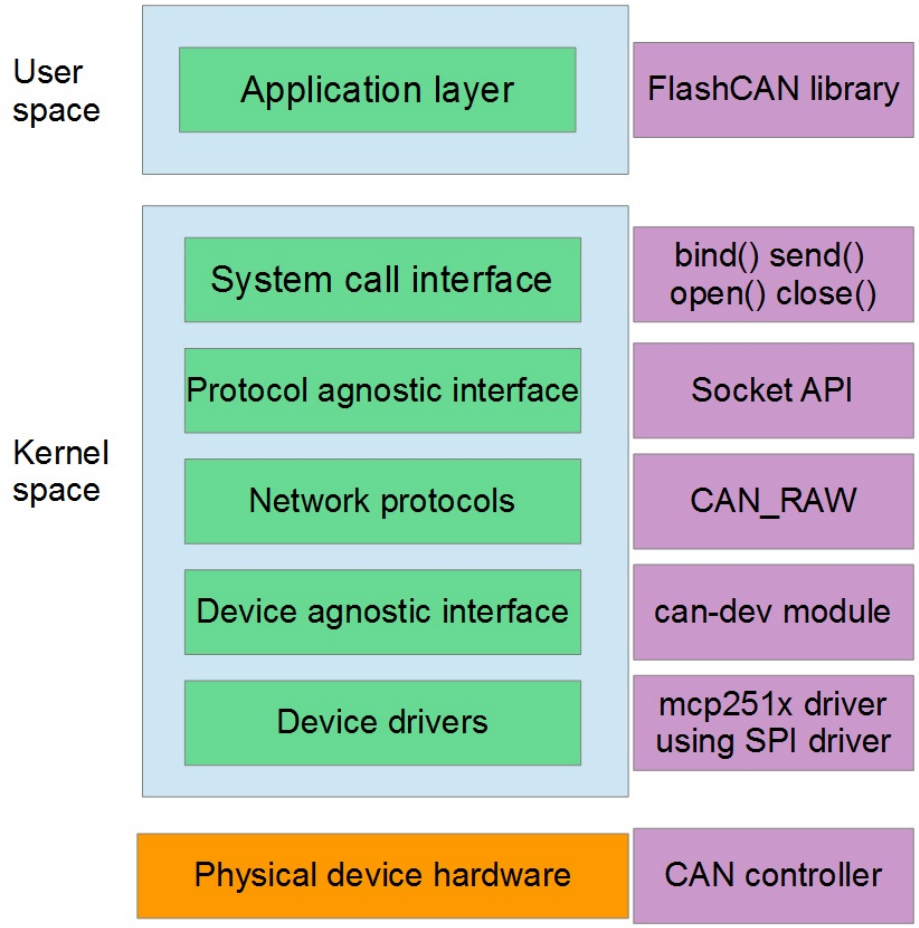


Figure 4.2: The different layers of the linux network stack and how they are implemented in this work. The advantage of this implementation is the easy exchange of hardware device with the cost of complexity. The flashCAN library will be described in the next chapter.

Other CAN implementations for Linux come as device drivers for some specific CAN hardware and provide a character device interface for transmission of CAN frames. When exchanging the device, most user space pro-

grams won't work with the new device driver. Whereas using SocketCAN, only the driver has to be changed (fig. 4.2), but all user space applications still can be used.

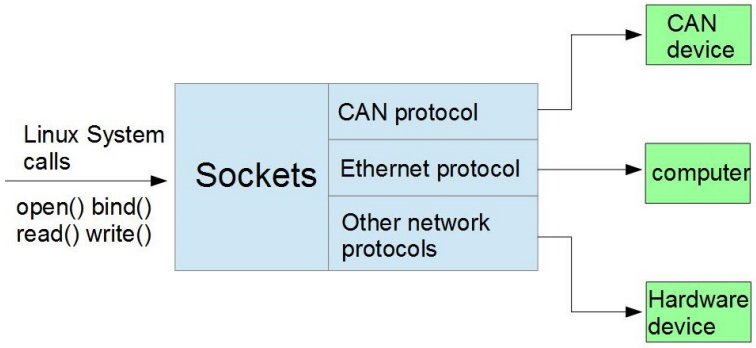


Figure 4.3: Using sockets as defined in the protocol agnostic interface. Different kind of network protocols can be used by the same system calls.

4.3.1 Kernel modules needed for SocketCAN

The module can.ko is the core module of SocketCAN and implements the CAN protocol family, but does not come with any specific protocol. For raw sockets [2], the module can-raw.ko has to be loaded as well. Both modules have to be loaded into the kernel at run time.

4.3.2 Using SocketCAN

For using SocketCAN for example in a C program, the header files can.h, raw.h and socket.h have to be included. Then a new socket can be opened and the CAN protocol family plus a specific CAN protocol can be chosen. In this work only the CAN protocol CAN_RAW is used. In the C struct *can_frame* all information of a valid CAN frame is stored. Using the system call *send()* and selecting the opened socket, a frame can be passed to the software socket. The frame will then be transferred over SPI to the controller. An example C file which demonstrates how to use SocketCAN is found in appendix A.1.1
Very useful built-in programs are *cansend* and *candump*. With *candump*, one

[2]The protocol RAW_SOCKET enables only basic message transmission, there are more powerful broadcast manager CAN protocols for sockets, which are not used in this work.

can observe the traffic on one or several connected can buses. The command *candump* only displays a message when the transmission was confirmed by the CAN controller over the interrupt pin. The command *cansend* transfers CAN frames to the controller:

```
//shows traffic on bus 0
candump can0
//shows traffic on all buses
candump any
//sends a CAN frame with 11 bit ID and 2 Bytes data in hex form
cansend 013#34AB
//sends a CAN frame with 29 bit ID and 3 Bytes data in hex form
cansend 00123456#FFAB12
```

4.4 General Purpose I/O (GPIO)

For using CAN on the Raspberry Pi, user space access to the GPIO pins is not required. But for using the LED's on the CANPi Interface (section 5.2) and the CAN transceivers error detection ability it is. Figure 4.4 shows which pins are available for GPIO usage. Following the Linux principle 'everything is a file', the GPIO pins are enabled by writing the pin number (xx) to the file:

```
/sys/class/gpio/export
```

Then one has to specify the direction by writing *in* resp. *out* to the file:

```
/sys/class/gpio/gpioxx/direction
```

where xx is the pin number. Setting the pin to logical 0 or logical 1 is done by writing the number to:

```
/sys/class/gpio/gpioxx/value
```

4.4.1 Software library wiringPi

A way to use the GPIO pins in a C program is the library wiringPi. Installation by using the program Git :

```
git clone git://git.drogon.net/wiringPi
cd WiringPi
./make
```

For usage in a C program:

```
#include <wiringPi.h>

if (wiringPiSetup() == -1)
  return 1;

pinMode(0, OUTPUT);

digitalWrite(0, 1);
```

This program sets a digital 1 on pin 11 of the GPIO Interface (see fig. 4.4 for the pin numbering). To compile the program the software library wiringPi has to be included when compiling:

```
gcc -o testprogram testprogram.c -lwiringPi
```

wiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	wiringPi Pin
–	–	3.3v	1 | 2	5v	–	–
8	2	SDA	3 | 4	5v	–	–
9	3	SCL	5 | 6	0v	–	–
7	4	GPIO7	7 | 8	TxD	14	15
–	–	0v	9 | 10	RxD	15	16
0	17	GPIO0	11 | 12	GPIO1	18	1
2	27	GPIO2	13 | 14	0v	–	–
3	22	GPIO3	15 | 16	GPIO4	23	4
–	–	3.3v	17 | 18	GPIO5	24	5
12	10	MOSI	19 | 20	0v	–	–
13	9	MISO	21 | 22	GPIO6	25	6
14	11	SCLK	23 | 24	CE0	8	10
–	–	0v	25 | 26	CE1	7	11
wiringPi Pin	BCM GPIO	Name	Header	Name	BCM GPIO	wiringPi Pin

Figure 4.4: RasPi's GPIO interface and different numberings of the pins. BCM refers to the internal numbering to the BCM2835 chip. The reserved pins for power supply and SPI connection are showed as well. [23]

Chapter 5

Realization of a CAN Interface

In this chapter the process of designing, constructing and initializing of the CAN Interface is described. Figure 5.1 shows a complete communication system of a computer operating PDP boards over a CAN bus. The Raspberry Pi is used together with a new designed circuit board as interface to the CAN bus. The chapter begins with the description of a small test board. This board was designed for testing the CAN controller. Then the design of the CANPi Interface is described. This board implements a fault tolerant CAN transceiver, a CAN controller, connection to CAN bus wires and an interface to the Raspberry Pi.

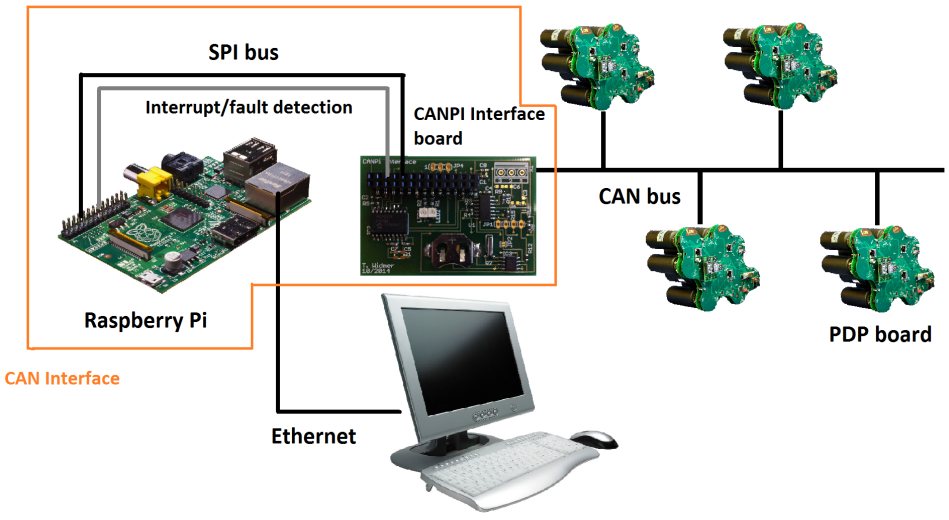


Figure 5.1: Several PDP boards connected over a CAN bus to a computer. The interface to the CAN bus is realized by the Raspberry Pi and the new designed CANPi Interface board. The computer is connected to the RasPi via Ethernet. The RasPi to the CANPi board via SPI.

The CAN application layer is realized in the software library flashCAN. This library uses the SocketCAN protocol, but implements GPIO access via wiringPi (section 4.4) as well. A description of its functions can be found in the appendix A.1.2.
At the end of the chapter a detailed step-by-step instruction is guiding through a complete installation of all required software packages and kernel modules and shows how to start the CAN Interface.

5.1 CANPi test-board

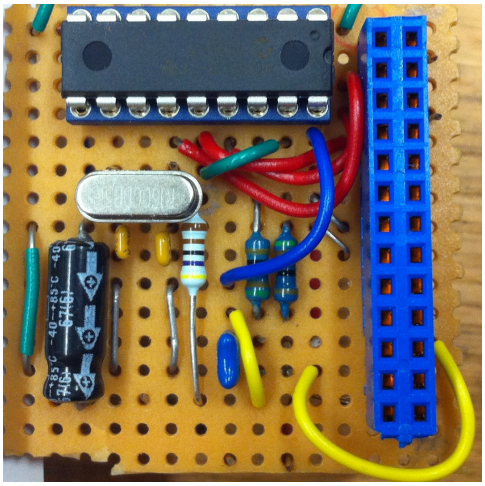


Figure 5.2: The CANPi test board with CAN controller, oscillator crystal and GPIO interface connector.

For testing the functions of the SocketCAN library and the CAN controller, a small test-board was designed (fig. 5.2). This mini board includes the interface to the Raspberry Pi as well as a circuit to operate the CAN controller MCP2515 and two control diodes for testing the RaspPi's GPIO pins. Because a CAN transceiver is missing in this circuit, the CAN input pin (RXCAN) of the controller has to be set on a logical 1 state, according to the CAN protocol this means the bus is idle. The CAN driver and the controller support a loop-back mode (see section 3.2). In this mode, no physical CAN bus is needed for testing the controller.

5.2 CANPi Interface

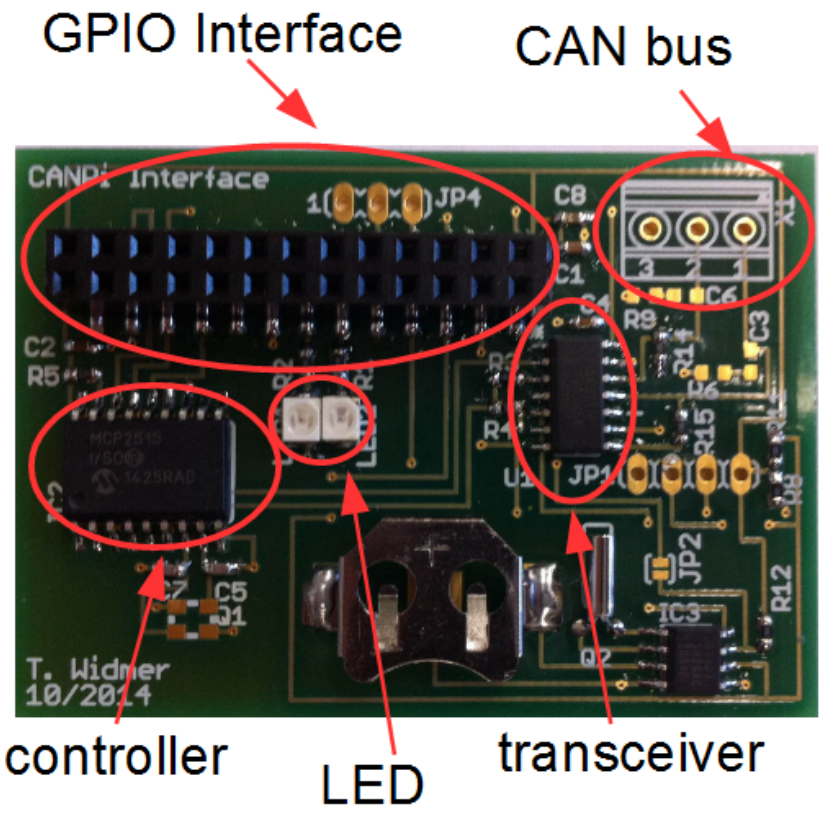


Figure 5.3: The CANPi Interface with MCP2515 controller and TJA1055 transceiver. The board is designed to be put direct onto the Raspberry Pi.

Based on the circuit diagram of a MCP2515 CAN controller and a high speed CAN transceiver by industrialberry [24], a new circuit diagram was designed using the layout editor EAGLE (fig. 5.4). The CANPi Interface board implements the low speed, fault tolerant CAN transceiver TJA1055 (fig. 5.3). The board contains a 16 MHz crystal to clock the controller and a screw terminal for the three CAN wires CAN-L, CAN-H and ground. The TXnRTS pins, which allow immediate transmission, and RXnBF pins, which are special interrupt pins for the reception buffers are not connected. For a future version, one could connect them to the RasPi's GPIO pins to force immediate transmission. The transceiver's error ($\overline{\text{ERR}}$) pin is connected to a GPIO pin of the RasPi. With this pin a bus fault can be reported to the RasPi, this is implemented in the flashCAN software as well. The two status LEDs are controlled over the GPIO pins too.

Power supply is realized by reserved pins of the GPIO Interface (section 4.4.1). The controller has a power supply of 3.3 V and the transceiver one of 5 V. The SPI related pins of the controller are connected to the reserved pins of the GPIO interface. To keep up functionality in fault case (limping home mode), the transceiver's pins CANH and CANL are connected over a 500 Ω resistance to the termination resistor connection (section 3.3).
On the CANPi Interface board, a Real Time Clock (RTC) is integrated as well due to the Raspberry Pi's lack of such a device. A DS1307 RTC chip with a connected 32.768 kHz crystal was used. Connected over reserved I/O pins and using the software driver rtc-ds1307 the RasPi keeps time when switched off. This function however is not necessary for the CAN interface to work.

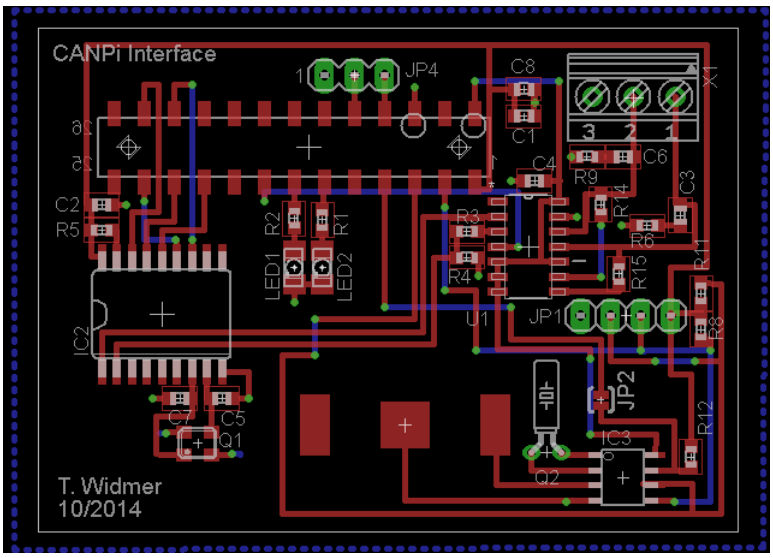


Figure 5.4: EAGLE board layout of the CANPi Interface. The jumpers JP2 and JP4 are only for testing.

5.2.1 Testing CANPi Interface

The CANPi Interface board can be put directly onto the Raspberry Pi (fig. 5.5). For testing the CANPi Interface, the MCP2515 was first switched into loop-back mode (section 3.2) to test the functionality of the controller and the communication over SPI with the RasPi. After switching the loop-back mode off, real CAN frames can be transmitted over the CAN bus to the PDP test board. The communication with a test PDP board worked. It was

possible to transmit a data frame to request temperature and humidity. The PDP board transmitted back frames which contained valid data of temperature and humidity, read out from the board's sensors. The PDP board's HVbits could be successfully set and read back (see section 1.2.1).

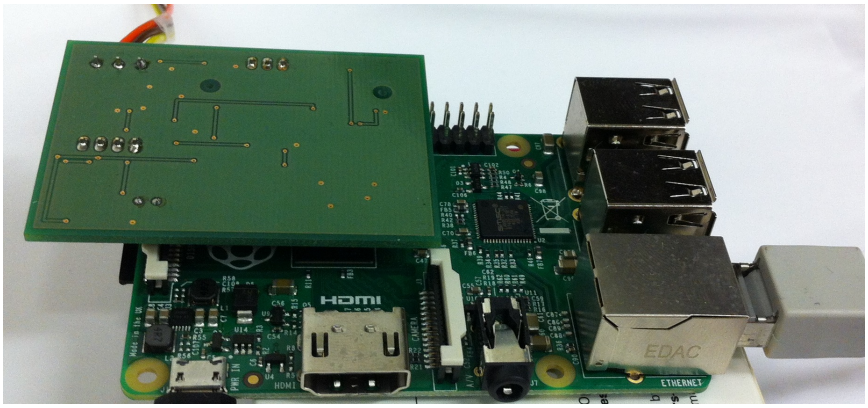


Figure 5.5: The CANPi Interface board put on the Raspberry Pi. The backside of the CANPi Interface is showed in the figure.

5.3 Development of the Software Library flashCAN

The software flashCAN provides user space functions for transmission of messages to one or several buses without troubling the user with the handling of sockets. One can transmit and receive normal and extended CAN frames and has access to the GPIO pins for fault detection and controlling the status LED's.

5.3.1 Structure of flashCAN

The header file flashCANlib.h is the core of the library as shown in figure 5.6. It declares all containing functions. In the C file transfer_frames.c all functions for transmission and reception of CAN frames are implemented. Sockets can be opened and matched to a chosen CAN bus and closed after successful transmission. The file canfunc.c provides some useful support functions and the file can_GPIO.c contains functions to access the RasPi's GPIO pins using the software library wiringPi (section 4.4.1). In the header file flashCAN.h the basic structure of a CAN frame is defined.
A description of all functions can be found in the flashCANlib.h header file in appendix A.1.2 To compile a C function using the flashCAN library:

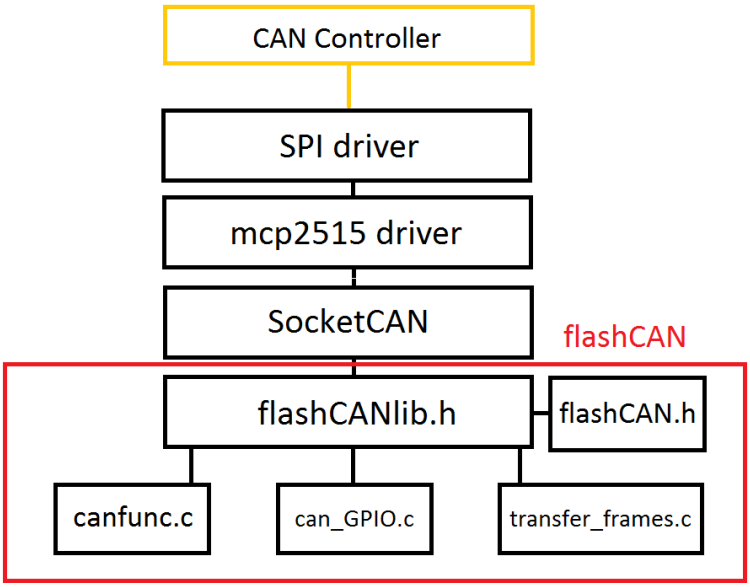


Figure 5.6: Structure of the flashCAN library and the underlying software levels.

```
gcc -o testprogram testprogram.c -lflashCANlib
```

5.4 Activating a CAN bus interface on Raspberry Pi

To activate the CAN controller and SPI connection to this controller from a Raspberry Pi, all required modules have to be loaded into the kernel at run time. If they are not integrated by default, they have to be compiled together with the kernel (see section 4.2) or precompiled modules [1] have to be extracted to the RasPi's SD card. After extraction, the modules are registered with:

```
sudo depmod -a
```

Except of the spi-config.ko module, all modules should already be integrated in new kernel versions.
To load the modules into the kernel, the commands *insmod* or *modprobe* are used. The command *modprobe* takes into account module dependencies, whereas *insmod* only loads the specified module. One has to be root to use those commands, or type *sudo* first:

[1] They must have been compiled together with the same kernel version which runs on the RasPi!

```
modprobe spi-bcm2708
modprobe spi-config devices=bus=0:cs=0:modalias=mcp2515:\
speed=1000000:gpioirq=25:mode=0:pd=4:pds32-0=16000000:\
force_release
modprobe mcp251x
```

or using insmod and the complete directory path (for kernel version 3.12.28+):

```
 cd /lib/modules/3.12.28+
 insmod /kernel/drivers/spi/spi-bcm2708.ko
 insmod /kernel/net/can/can.ko
 insmod /kernel/drivers/net/can/can-dev.ko
 insmod /kernel/net/can/can-raw.ko
 insmod /kernel/net/can/can-bcm.ko
 insmod /extra/spi-config devices=bus=0:\
cs=0:modalias=mcp2515:speed=1000000:gpioirq=\
25:mode=0:pd=4:pds32-0=16000000:force_release
insmod /kernel/drivers/net/can/mcp251x.ko
```

The parameters of *spi-config* are explained in section 4.2.2. In the above example, an SPI bus with number 0 with chip select pin number 0 was initialized. SPI speed was set to 1 MHz. The device module alias is mcp2515 and the interrupt pin number 25 was selected (fig. 4.4 for GPIO pin numbering). SPI mode is 0 [16]. The latest mcp251x driver only has one platform data variable which is an integer and defines the MCP2515's oscillator frequency. Therefore 4 Byte of memory are allocated and a 32 bit signed variable with 0 Byte offset is defined as 16 MHz. The *force_release* command releases all other SPI devices which are registered.
Now the CAN controller is registered as a network device and a new socket family for CAN devices is available. We can use standard network system commands to start the controller:

```
ip link set can0 type can restart-ms 10
ip link set can0 up type can bitrate 125000
ifconfig can0 txqueuelen 0
candump can0
```

The *restart-ms* value defines the time the controller waits to restart after it went into bus off state (fig. 2.12). In this example the bit rate was set to 125 kbit, the maximum value for a fault tolerant CAN bus. The command *up* activates the bus. With *candump can0* one monitors the bus traffic. To change the SPI parameters, the *spi-config* module has to be off-loaded by:

```
modprobe -r spi-config
```

and reloaded with new parameters.

Chapter 6

CAN interface in operation

In this chapter are the operation modes of the CANPi Interface described. To test the performance of the interface, a single bus system (fig. 5.1 and 6.1) was connected to the Raspberry Pi and messages transmitted to the bus. Then a double bus system (fig. 6.2) was tested of performance as well. The performance of the arrangement was measured by testing broadcast speed and stability. At the end of the chapter, known errors and possible solutions are discussed. To understand why the errors occur, message transmission is explained in detail. Then follows a conclusion of how usable the combination of Raspberry Pi and controller MCP2515 is.

6.1 Modes of operation

There are two modes of operation which were tested. Single bus mode with one CANPi Interface and double bus mode with two CANPi Interfaces in parallel. SPI speed is 80x faster then fault tolerant CAN, which means during CAN frame transmission the SPI is idle most of the time. In this time the RasPi could, in theory, serve multiple controllers without impact on the individual CAN bus frame rate over SPI.

6.1.1 Single bus operation mode

In single bus mode, one CANPi Interface connected to the RasPi over SPI is communicating with two PDP test boards via one CAN bus. It is possible to send specific commands to the boards (appendix A.3). For example setting the HV bits and reading them back and request temperature and humidity data and read them back (section 5.2.1).

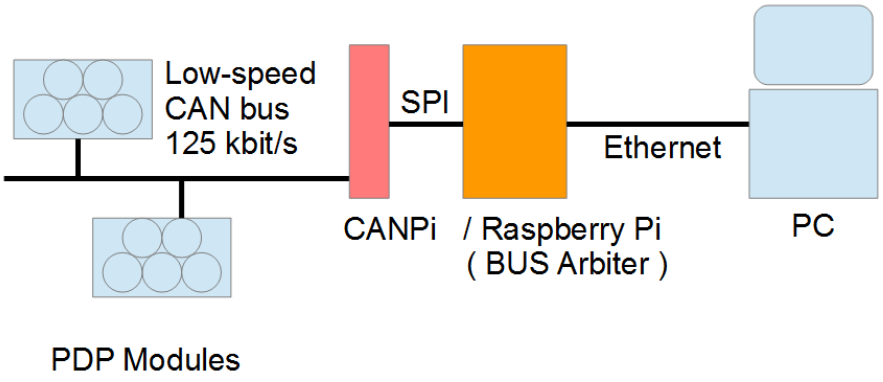


Figure 6.1: Single bus system. Two PDP boards are connected over one CAN bus to the CANPi Interface. In this mode no errors are detected and a bus speed of 770 frames/s was reached.

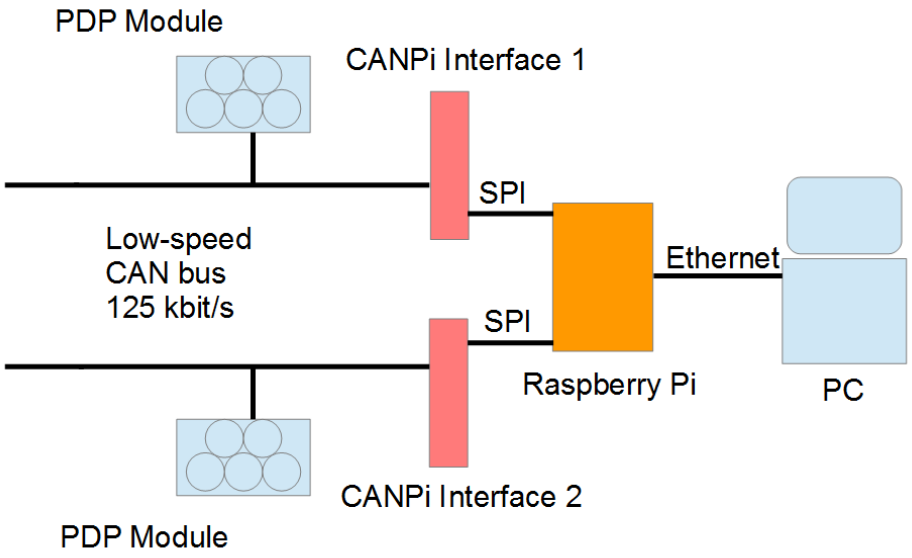


Figure 6.2: Double bus system. One PDP board is connected to one CAN bus each. Two CANPi Interfaces are connected to one Raspberry Pi. Switching between the buses results in an interrupt error after some time.

6.1.2 Multi bus operation mode

There are only two chip-select pins on the RaspPi, which limits the number of SPI devices connected to one RasPi. But other GPIO pins could be used for chip selection as well. In that case, the driver has to be modified. But each controller needs a separate interrupt pin as well. And with the new CAN transceiver TJA1055T there is an additional need for a GPIO pin for fault detection. So there is a limitation of the number of CANPi Interface boards one RaspPi can handle. Not using any GPIO pins for a Real Time Clock, any status LED's or fault detection, there are 14 GPIO pins left. So at most seven CANPi Interface boards could be connected to one RasPi.
To test a two bus system a second CANPi Interface parallel with the first one is connected to the RasPi. One PDP test board is connected to each CAN bus. The RasPi switches communication over SPI between the CANPi Interface boards by selecting the corresponding chip select pin. The CANPi boards contact the RasPi via their interrupt pins.

6.2 Performance

To determine the speed of data transmission in single bus mode, some test programs were written. When the CAN frames were sent too frequently by the system call *send()*, some frames got lost, because the controller's transmit buffers were full. This happens because the SPI bus transmits bits 80x faster than the CAN bus. First a delay was implemented to find the maximal frame rate, where no frames get lost. In addition a method was implemented, by which the function tries to send a message until it is accepted by the socket. This method, called polling, works well but needs unnecessary CPU speed. Using the system calls *select()* or *poll()*, which wait for an interrupt, would be a better solution, but is not implemented yet.

6.2.1 Broadcast speed

Low speed CAN bit rate has a maximum of 125 kbit/s. Assuming a CAN frame having an average size of 100 bit (extended frame with 4 Byte of data and 4 bit by bit stuffing) we get a transmission time of 0.8 μs. This gives a theoretical value of 1250 frames/s. Of course this value is never reached in reality. For every sent frame, the controller communicates five times with the RasPi over SPI (fig. 6.5). Every time the chip select pin is set on a logical high state, there is SPI communication. This raises the transmission time to about 1.3 μs and leads to a frame rate of 770 frames/s. This value was

confirmed by sending 1 Million frames and taking the time in single bus mode.
In double bus mode, speed measurements were limited due to the system's instability.

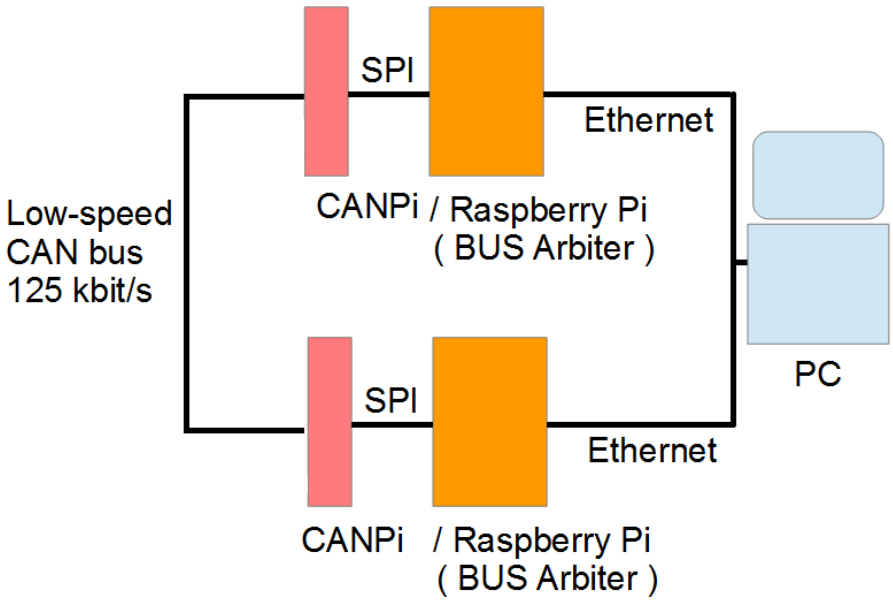


Figure 6.3: Two Raspberry Pi single board computer connected via CAN bus. Via Ethernet, connection from one computer to both RasPis is possible.

6.2.2 Stability

In single bus mode no interrupt conflict was observed when just transmitting frames. The system was stable in all made observations but long term stability has not been tested yet. When receiving frames with a fast rate[1] (~940 frames/s) there is a SPI conflict, occurring when the interrupt handler driver of the BCM2835 chip takes too much time. This was tested by connecting two Raspberry Pi computers via CAN bus (fig. 6.3).

When applied two controllers to the RasPi (double bus mode), the system crashes very fast. With a oscilloscope it is possible to examine the interrupt pins of the two MCP2515 controllers. The controllers interrupt pin ($\overline{\text{INT}}$) is inverted, meaning the logical state being 1 normally, and 0 when

[1] Transmitted by a Raspberry Pi B+ with CANPi Interface board.

an interrupt occurs. The RasPi triggers on the negative edge of the interrupt signal. It is observed, that when the interrupts are close (<10μs), one interrupt pin stays low. The problem seems to be the RasPi and its SPI interrupt handling. The SPI driver only detects one interrupt, the other gets lost. The $\overline{\text{INT}}$ pin is raised by the controller only after getting the command from the RasPi to do so over the SPI. If it doesn't receive such a command, the pin stays low. The controller is waiting for acknowledge by the processor (which never comes), not allowing to receive another message from the processor. Figure 6.4 shows this interrupt conflict. Reducing SPI speed to 250 kHz keeps the system stable over some time, several thousand transmitted frames, but results then in a system crash as well. Possible reasons for this behavior are discussed below.

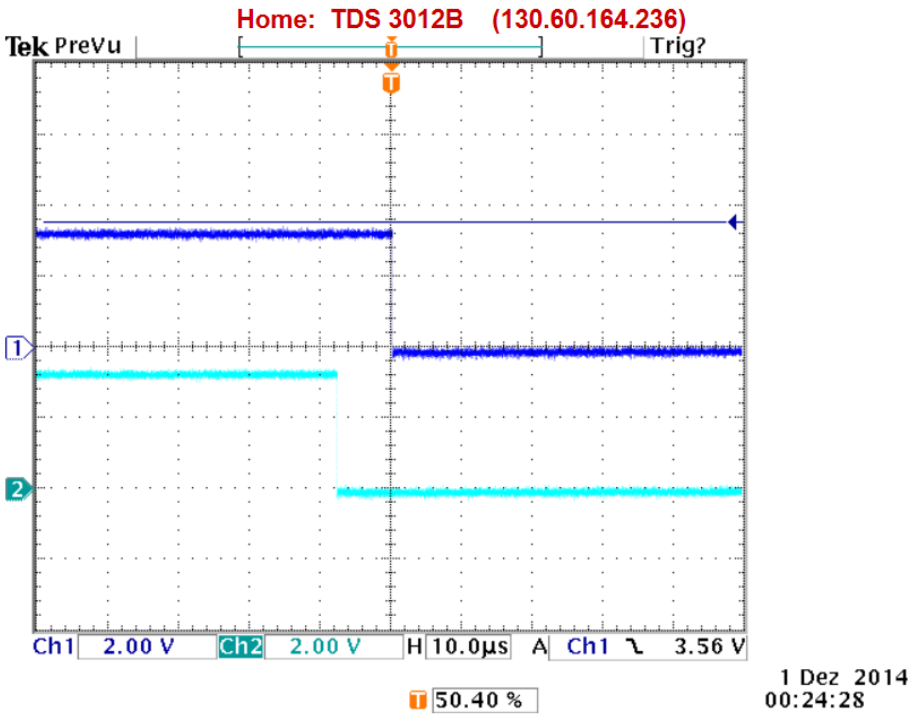


Figure 6.4: Interrupt pin signals taken with a oscilloscope of two CAN controller when a system crash occurs. When in less than 10μs two interrupts occur, the second interrupt gets lost.

6.3 Message transmission in detail

To understand the interrupt conflict, the message transmission shown in figure 6.5 must be explained (see figure 3.4 as well). The driver reads via SPI the controller's register and checks, if the buffer is empty. Then the message is transmitted to the controller via SPI. The controller transmits the message via transceiver over the CAN bus. After successful transmission, the controller generates an interrupt. The driver checks the register via SPI, to identify what triggered the interrupt. Another SPI command is sent to clear the interrupt register, whereupon the controller stops generating the interrupt signal. The last SPI command is used to read the status register again. The whole procedure takes ~1.2 ms, plus 1 ms computation time between the messages gives ~1.3 ms transmission time.

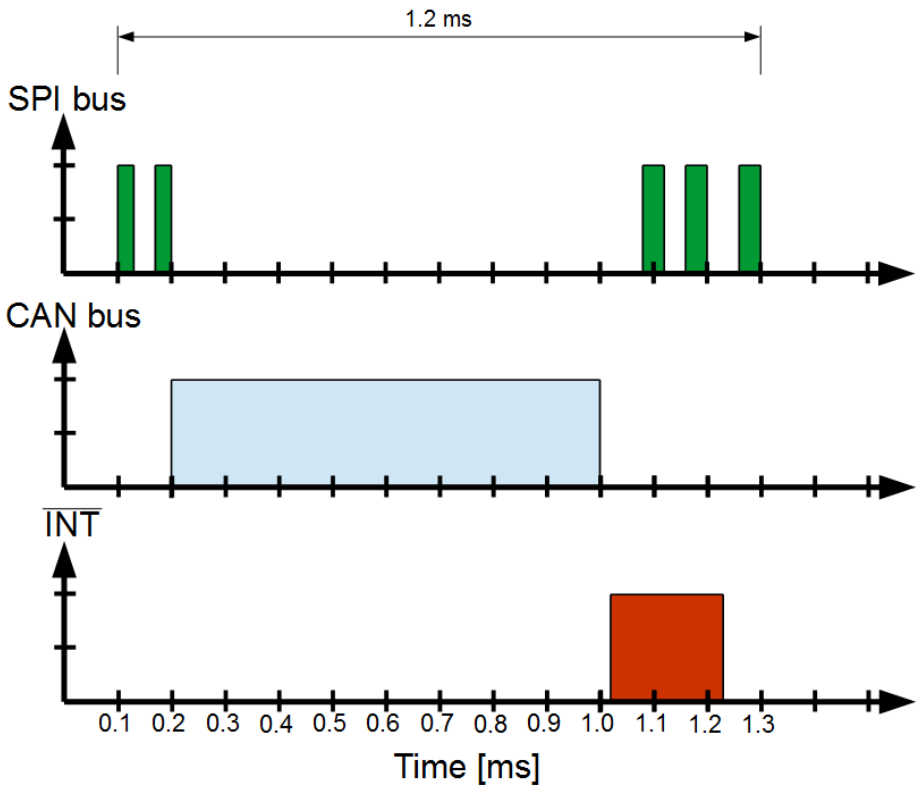


Figure 6.5: Message transmission. SPI transmission, CAN frame transmission and interrupt triggering are shown.

6.4 Known errors and solutions

During the development of the CAN Interface some time consuming errors turned up, concerning hardware and software. Those errors and possible solutions are listed in this section.

Pin conflict: The two CANPi Interface boards are both identical, therefore both chip select ($\overline{CS}$) pins share one common wire when connected via ribbon cable. The same holds for the two interrupt ($\overline{INT}$) pins and the fault detection pins. For the SPI Interface driver to select one specific device, it needs to select the device via a defined pin.
Solution: The Raspberry's CE1 pin,which corresponds to the second chip select, has to be wired to the $\overline{CS}$ pin of the second CAN controller via a separate wire. The interrupt pin can be chosen as one of the GPIO pins of the RasPi, this is done when initializing the SPI device with *spi-config*, see section 5.4. To distinguish between the two interrupts, a separate wire is needed for the second CANPi Interface board as well.

Bus disappears: Before kernel version 3.12.28+, sometimes the CAN bus device disappeared from the network list and is not accessible any more. This is caused by a network daemon called *ifplugd*, which activates and deactivates networks (ethernet, wireless LAN but CAN as well) when connectors are plugged in or out. For some unknown reason this daemon deactivated CAN, not allowing to restart it by *ifconfig can0 up*.
Solution: The only method to bring the CAN device back was to *kill* the *ifplugd* process.

Interrupt conflict: The most serious and not yet solved problem is the interrupt conflict in multi bus mode as described in section 6.2.2. Reducing SPI speed and pause between transmission increases stability, but does not solve the problem. Updating raspian to version 3.12.28+ and using the improved kernel modules (see section 4.2) *mcp2515a.ko* (instead mcp251x.ko) and *spi-bcm2835dma.ko* (instead *spi-bcm2708.ko*) didn't improve the system. These new kernel modules were written because people faced some latency problems when using high speed CAN together with the Raspberry Pi. The new modules do not only not improve the system, CAN frames were corrupted using them. Data and identifier are not transferred correctly anymore.
Possible solutions: As the problem seems to be the RasPi's BCM2835 chip

driver and its GPIO interrupt handling routine, modifying this driver may help. Improving the *spi-bcm2708.ko* and *mcp251x.ko* driver code for faster SPI communication could help as well.
An other solution is writing an own CAN character driver for the controller MCP2515, using a software library like wiringPi or bcm2835 [2], instead of using the Linux network stack and SocketCAN.
The CAN controller has an intern register which enables interrupts. One could use SPI commands from user space to set this register to logical 0 and disable interrupts. Polling the registers to detect message transmission must be implemented in that case.

SPI conflict: When receiving frames with maximal available speed, that is 930 frames/s, it happens that the BCM2835 interrupt driver still processes the interrupt routine [3] and overlaps with the next interrupt from the controller. This results in the same error as in the interrupt conflict. The reason for this behavior, and if it is related to the interrupt error, is not yet understood.

6.5 Conclusions

Using Raspberry Pi to implement a fault tolerant CAN bus system works. Fault tolerant CAN is not fast enough to cause any latency problems with the Raspberry Pi when just transmitting frames. Receiving frames with a rate of 770 frames/s causes no failures. Increasing the speed to over 900 frames/s causes in rare cases a system crash. Connecting two or even more CAN controllers to one Raspberry Pi causes some serious interrupt problems and is not recommended at the actual state.

6.6 Outlook

Using the new Raspberry Pi model B+ seems to increase broadcast speed significantly. Writing an own CAN controller character device driver would simplify debugging and increase system knowledge. The Raspberry Pi foundation launched the new Raspberry Pi compute module 6.6. It has the same performance as the original Raspberry Pi, but only a size of 67.6 x 30 mm. This could reduce the size of the CAN interface and save space.

[2] Not to mistaken with the spi-bcm2835 driver or BCM2835 chip

[3] Reading the controllers register to determine what kind of interrupt happened. Resetting the registers and reading the message out of the RX buffer. This takes 3 SPI commands

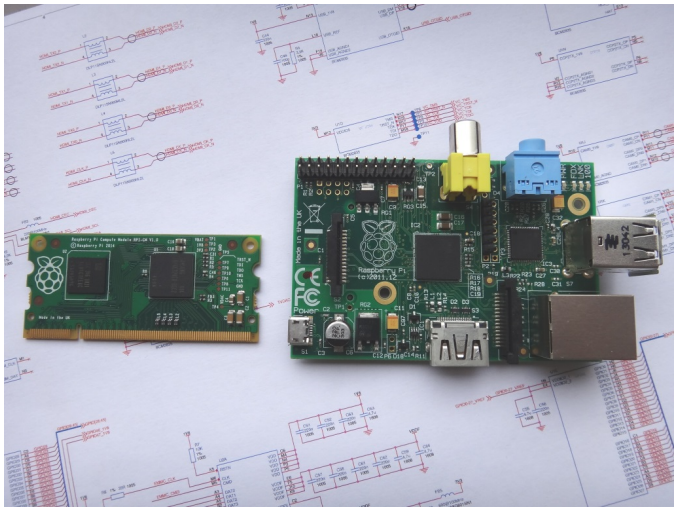


Figure 6.6: The compute module compared to the original Raspberry Pi. The module sizes only 67.6x30 mm and implements 48 GPIO pins. [17].

Appendices

A.1 Code

A.1.1 Example for using SocketCAN

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <net/if.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <linux/can.h>
#include <linux/can/raw.h>

int main(void)
{
int s;
int nbytes;
struct sockaddr_can addr;
struct can_frame frame;
struct ifreq ifr;

char *ifname = "can1";

if((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
perror("Error while opening socket");
return -1;
}

strcpy(ifr.ifr_name, ifname);
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family  = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

if(bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
perror("Error in socket bind");
```

```
return -2;
}

frame.can_id  = 0x2D + CAN_EFF_FLAG;
frame.can_dlc = 2;
frame.data[0] = 0xFF;
frame.data[1] = 0x00;

nbytes = write(s, &frame, sizeof(struct can_frame));

return 0;
}
```

A.1.2 flashCANlib

The header file flashCANlib.h which defines all functions in the flashCAN software library:

```
/* C library for a CAN-Interface */
/*defines the basic structure for the use in a flashCam System*/
#include "flashCAN.h"

/*socket functions */

//opens a socket for a given bus
int opensocket(int bus_id);
// close socket s
void closesocket(int s);

/* functions for normal CAN-Frames*/

// sends a normal CAN-Frame, opens a socket,
//sends the frame and closes the socket
int send_norm(struct flash_can_frame flashframe);
// only sending, needs to know on which socket
int sendonly_norm(int pdp_id, int device_id,
int datalen, int *data, int sock);


/* functions for extended CAN-Frames */
```

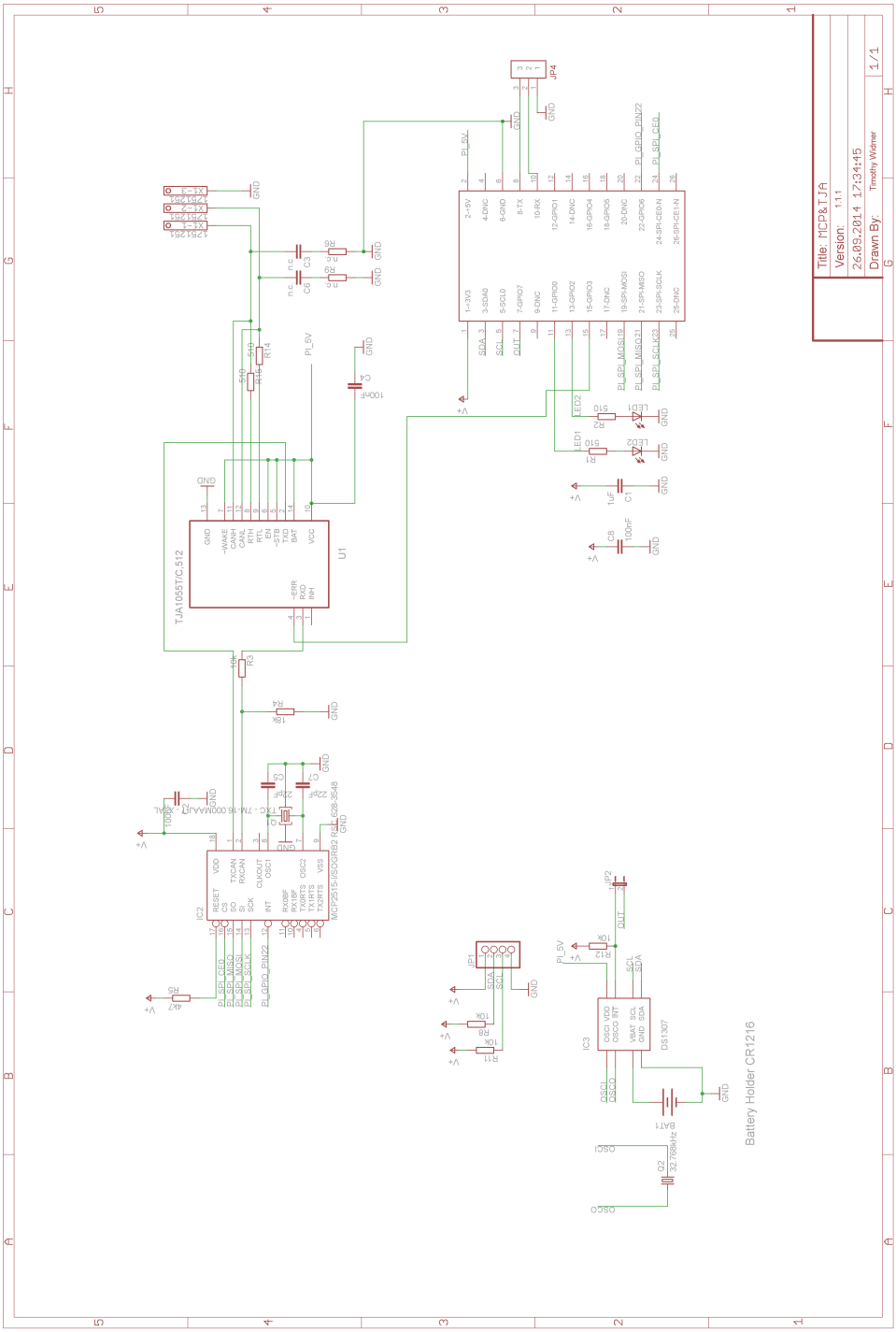
```
// sends a extended CAN frame, opens a socket,
//sends the frame and closes the socket
int send_ext(struct flash_can_frame flashframe);
//only sending, needs to know on which socket
//and all frame informations
int sendonly_ext(int pdp_id, int device_id,
int datalen, int *data, int sock);

/* receive functions for both normal and extended frame*/

// only receiving, needs the can bus number
struct flash_can_frame receiveonly_frame(int cannr);
// opens a socket, receives frame, closes the socket
struct flash_can_frame receive_frame(int cannr);

/* supporting functions */

//transform a number in a bus id. 4 -> can4
char* make_can_char(int cannr);
// not implemented
int giveframesize(void);

/* functions for the GPIO's*/

// reads from a GPIO pin
int readpin(int pin);
// writes to a GPIO pin
void writepin(int pin,int bit);
// initialzes a GPIO pin
int initGPIO(int pin,char *inout);
// maps the wiringpi pins to actual GPIO numbers
int mappin(int pin);
```

The header file flashCAN.h:

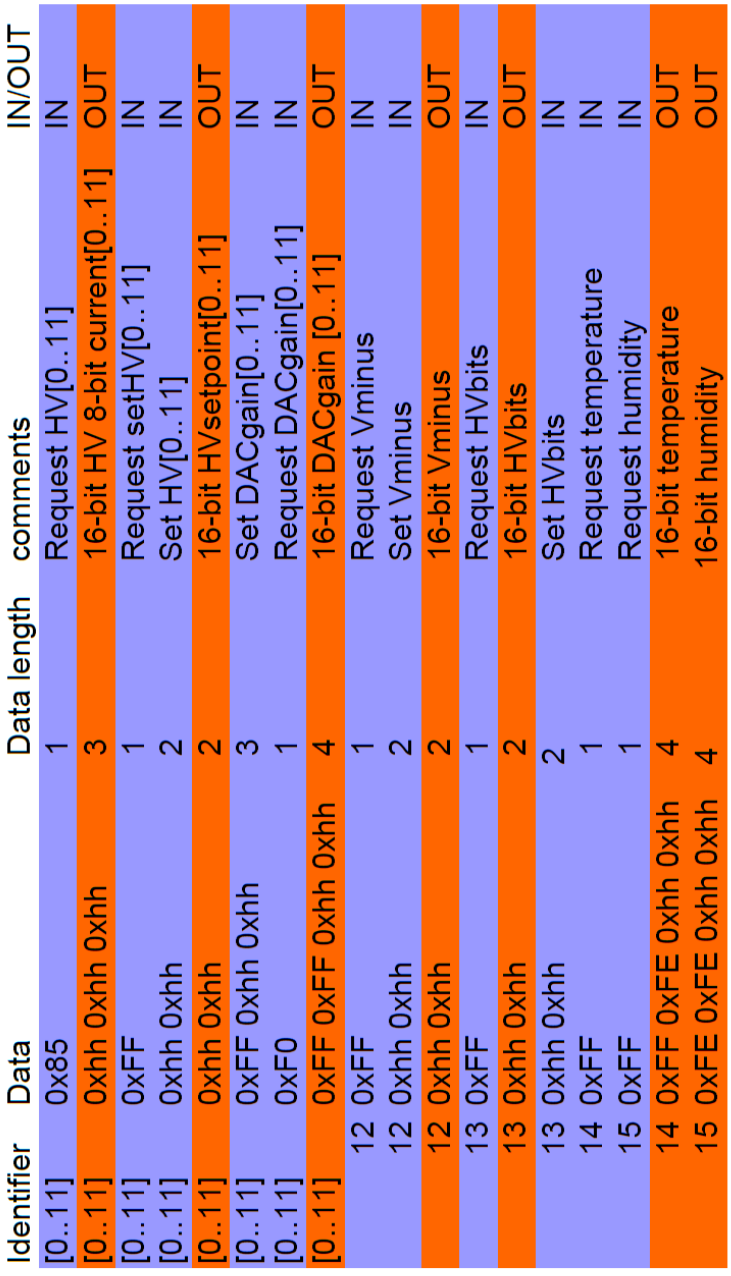
```
/* defines structs and variables */

/*defines the basic structure for the use in a flashCam System*/
struct flash_can_frame {
int pdp_id;
int device_id;
```

```
int data[8];
int datalen;
int bus_id;
//int s;
};
```

A.2 CANPi Interface wiring diagram



A.3 PDP board CAN commands



Identifier	Data	Data length	comments	IN/OUT
[0..11]	0x85	1	Request HV[0..11]	IN
[0..11]	0xhh 0xhh 0xhh	3	16-bit HV 8-bit current[0..11]	OUT
[0..11]	0xFF	1	Request setHV[0..11]	IN
[0..11]	0xhh 0xhh	2	Set HV[0..11]	IN
[0..11]	0xhh 0xhh	2	16-bit HVsetpoint[0..11]	OUT
[0..11]	0xFF 0xhh 0xhh	3	Set DACgain[0..11]	IN
[0..11]	0xF0	1	Request DACgain[0..11]	IN
[0..11]	0xFF 0xFF 0xhh 0xhh	4	16-bit DACgain [0..11]	OUT
12	0xFF	1	Request Vminus	IN
12	0xhh 0xhh	2	Set Vminus	IN
12	0xhh 0xhh	2	16-bit Vminus	OUT
13	0xFF	1	Request HVbits	IN
13	0xhh 0xhh	2	16-bit HVbits	OUT
13	0xhh 0xhh	2	Set HVbits	IN
14	0xFF	1	Request temperature	IN
15	0xFF	1	Request humidity	IN
14	0xFF 0xFE 0xhh 0xhh	4	16-bit temperature	OUT
15	0xFE 0xFE 0xhh 0xhh	4	16-bit humidity	OUT

Figure A.1: PDP board CAN commands.

References

[1] Surveying the universe with egret 1991-1996. http://heasarc.gsfc.nasa.gov/docs/cgro/egret/3rd_EGRET_Cat.html.

[2] W. Bednarek. High energy gamma-ray emission from compact galactic sources in the context of observations with the next generation cherenkov telescope arrays. *Astroparticle Physics*, 43(0):81 – 102, 2013.

[3] Felix A. Aharonian. Gamma rays from supernova remnants. *Astroparticle Physics*, 43(0):71 – 80, 2013.

[4] H. Sol et al. Active galactic nuclei under the scrutiny of {CTA}. *Astroparticle Physics*, 43(0):215 – 240, 2013.

[5] T.C. Weekes. *Very High Energy Gamma-Ray Astronomy*. Institute of Physics Pub, 2003.

[6] Daniel Mazin et al. Potential of {EBL} and cosmology studies with the cherenkov telescope array. *Astroparticle Physics*, 43(0):241 – 251, 2013.

[7] Peter Meszaros. Gamma ray bursts. *Astroparticle Physics*, 43(0):134 – 141, 2013.

[8] Arno Gadola. *Towards the first imaging atmospheric Cherenkov telescope camera with continuous signal digitization*. PhD thesis, Univ. Zuerich, 2013.

[9] Gerd Puehlhofer + FlashCam team. Flashcam section for preliminary technical design report. Jul. 2013.

[10] Lou Frenzel. Whats the difference between the osi seven-layer network model and tcp/ip. http://electronicdesign.com/what-s-difference-between/what-s-difference-between-osi-seven-layer-network-model-and-tcpip.

[11] BOSCH. Can specification, Nov. 1991. http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf.

[12] BOSCH. Can with flexible data-rate, Dec. 2012. http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd_spec.pdf.

[13] carbussystems. Can. http://www.carbussystems.com/CAN.html.

[14] CiA. Can physical layer. http://www.datasheetarchive.com/ISO11992-datasheet.htmlf.

[15] Microchip Technology Inc. *MCP2515 Data Sheet*, 2007. http://ww1.microchip.com/downloads/en/DeviceDoc/21801e.pdf.

[16] mikrocontroller.net. Serial peripheral interface. http://www.mikrocontroller.net/articles/Serial_Peripheral_Interface.

[17] Raspberry pi compute module. http://www.raspberrypi.org/raspberry-pi-compute-module-new-product/.

[18] NXP Semiconductors. *TJA1055 Data Sheet*, 2013. http://www.nxp.com/documents/data_sheet/TJA1055.pdf.

[19] O. Hartkopp et al. can.txt socketcan documentation. https://www.kernel.org/doc/Documentation/networking/can.txt.

[20] G. Bertelsmann. Can + raspberry pi. http://lnxpps.de/rpie.

[21] Raspi-anleitung. http://lnxpps.de/rpie/raspi-anleitung.txt.

[22] Rpi canbus. http://elinux.org/RPiCANBus.

[23] Wiringpi documentation. https://projects.drogon.net/raspberry-pi/wiringpi/pins/.

[24] Industrialberry open electronics project. http://www.industrialberry.com/canberry-v-1-1/.

Acknowledgments

I would like to thank my two excellent supervisors, Dr. Achim Vollhardt and Dr. Arno Gadola. They always had the time and patience to explain even the most basic principles to me. I think I've never learned that many new things in such a short time (not even in Prof. Straumann's lectures). The two of them motivated me every day with their open minded nature to keep on working on my thesis. Then, I want to thank Prof. Straumann for giving me the opportunity to write my thesis in his group. Having a CTA meeting together with him and the rest of the group was a very interesting experience. Many thanks to Roman, who stayed with me very late trying to find bugs in the system. I did profit a lot of his knowledge as much as I did from the others in the office. Nicola, Christian, Marco and Dario were always eager to help when there was a problem. Thank you all!