ROBOTICS RESEARCH LAB
DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF KAISERSLAUTERN

Bachelor Thesis

Implementation of a FlexRay Communication Interface for Linux

Michael Arndt

April 13, 2009

Bachelor Thesis

Implementation of a FlexRay Communication Interface for Linux

Robotics Research Lab
Department of Computer Sciences
UNIVERSITY OF KAISERSLAUTERN

Michael Arndt

Hereby I declare that I have self-dependently composed the Bachelor Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

April 13, 2009 – Kaiserslautern

(Michael Arndt)

Preface

Thanks go to the staff of the Robotic Research Lab at the University of Kaiserslautern that helped me solve tricky problems during the work on this thesis.

Hereby I would also like to apologize to all of my friends and family that suffered from my moods and constant pressure during these ten weeks, which are – in my opinion – much too less time to create anything that holds water.

The image on the cover shows the *FlexRay for Linux* logo, designed by Joachim Folz and kindly released under the Creative Commons BY-NC-SA 3.0 Germany license[1]. It depicts the ray[2] from the official FlexRay logo (see Figure 1) morphed with Linux' mascot, the penguin *Tux*.

Figure 1: The official FlexRay logo

Contents

1. Introduction

1.1 Motivation

Today the need for deterministic communication protocols for security critical applications constantly increases. These applications can escpecially be found in the automotive domain. As concepts like x-by-wire with $x \in \{\text{drive}, \text{steer}, \text{brake}, \ldots\}$, where steering arms and brake pipes are replaced by electric wires and integrated circuits, become more and more popular, robust and fail-safe communication systems are needed to realize such functionality. Failures of these critical systems will almost immediately lead to severe accidents and personal injury.

Of course, the use of communications protocols such as FlexRay is not limited to automotive applications. Especially in robotics, similar problems exist, consider a heavy autonomous offroad robot, like RAVON[1]. A robot of this size can easily harm people if the safety mechanisms fail.

But not only safety concerns are of interest in robotics. There are often situations where a lot of sensor values need to be queried and actuator values have to be set. Often, also exceptional events that occur sporadically, such as user interaction or debug messages that need to be transmitted. For such scenarios FlexRay is well-suited, because it provides guaranteed, periodic *static* slots and optional *dynamic* slots that transmit data every once in a while. At the Robotic Research Lab[2] (RRLab) at the University of Kaiserslautern, currently Controller Area Network (CAN) is used to transmit sensor and actuator values. Future applications with many sensors and actuators will probably exceed the capabilities of CAN and need a communication systems with more capacity, such as FlexRay.

To effectively debug or to interact with embedded systems that employ FlexRay, it is neccessary to somehow connect a personal computer to the system. Because Linux is the default operating system at the RRLab, a framework for Linux is needed to work with it. As FlexRay is a quite young technology that has not yet been intensively

---

[1] http://agrosy.cs.uni-kl.de/en/robots/ravon/technical-data-of-ravon/
[2] http://rrlab.cs.uni-kl.de/

used in the industry, there exist few attempts to bring this technology to Linux. Solutions by Vector Informatik GmbH[3] only include Microsoft Windows drivers at the time of writing. TZ Mikroelektronik[4] offers products with support for a variety of operating systems, including Linux, but does not seem to make these drivers and frameworks publicly or even freely available. Also their code is probably limited to their own hardware products only.

1.2   Goals

The goals of this Bachelor thesis are the following:

- Provide an introduction to and a basic understanding of the FlexRay protocol

- Development of a cheap and universal interface to connect a FlexRay cluster to virtually any computer running Linux

- Establishment of an infrastructure in the Linux kernel to communicate over FlexRay, similar to the SocketCAN architecture [LLCF 06]

- Make this infrastructure freely available

- Describe a specific hardware setup of a FlexRay cluster, leaving the reader with enough knowledge to implement own FlexRay systems

1.3   Overview

The thesis is divided into several chapters. chapter 1 contains the introduction you are currently reading, chapter 2 gives a general, controller independent overview of FlexRay. The next chapter, chapter 3, introduces the hardware components that have been used during the research as well as the low level FlexRay driver by Freescale. chapter 4 describes the implementation of FlexRay for Linux, followed by chapter 5 which contains conclusions and possible future work, based on this thesis.

1.4   Notations

For the sake of readability, the following notations will be used troughout the thesis:

- Hexadecimal numbers will be prefixed by 0x, such as 0x1337

- Binary numbers (bitvectors) will be suffixed with b, such as 1001100110111b

- Function names, datatypes, variables and constants will be printed in monospace font, like `ioctl()`, `struct sockaddr_fr`, `Fr_buffer_info_type`, `FLEXRAY_INIT`

- File-system paths and systems commands will be printed in monospace font, `/dev/ttyUSB0`, `lsmod`

---

[3]http://www.vector.com/
[4]http://www.tzm.de/

- Network device names and Kernel module names will be printed in monospace font, e.g `flexray0`, `flexray_serial.ko`

- Web-links are colored blue and should be "clickable" in the online version of the document, example: `http://www.uni-kl.de/`

2. FlexRay

This chapter will first give a short overview of FlexRay and its history. Then, the communication cycle and the format of FlexRay frames is explained in detail. Most of the information in this chapter can be found well-explained in [Rausch 07], but the FlexRay protocol specification [PS 05] is also a very valuable source when studying FlexRay.

2.1 Overview

FlexRay has been designed by the FlexRay consortium as a *flexible* (hence part of the name) and fast communication system that can satisfy many of the needs of automotive applications.

The FlexRay consortium has existed since 2000, its core members are BMW, Volkswagen, Daimler AG, General Motors, Robert Bosch GmbH, NXP semiconductors and freescale semiconductors. Additionally to the core members, there are over 100 other members that develop FlexRay hard- and software. FlexRay has always been developed to suit the needs of automotive applications, such as emerging trends like steer- or break-by-wire.

The need for deterministic communication systems comes clear while looking at the Controller Area Network , which is in use for automotive applications for some time now. CAN uses priority-based bus arbitration that allows the user to transmit important messages in favor of less important ones. Unfortunately, a lot of highly important messages can surpress all other messages quite easily. It is quite difficult to transmit two data streams with equal priorities over CAN.

FlexRay overcomes this problem by dividing the medium into static and dynamic time slots. During the static segment, a deterministic access schema is used to transmit periodic messages while the dynamic segment allows priority-based transmission of sporadic messages.

Another great feature of FlexRay is the capability to use two physical transmission channels, named channel A and B. Each channel can transmit up to 10MBit/s on the physical layer, which is ten times the factor of the maximum rate that is possible

with a CAN channel. The two channels can be used to double the overall datarate, provide redundancy or even both, by transmitting only important messages on both channels (see subsection 2.3.1 for details).

To connect to a FlexRay cluster, a *FlexRay Communication Controller (CC)* is required. It handles the protocol according to the specifications and connects to some host, e.g. a microprocessor. There exist mainly two types of FlexRay CCs, standalone integrated circuits and controllers integrated into Microcontrollers (μCs) or Digital Signal Processors (DSPs).

2.2 FlexRay Network Topologies

FlexRay is not a classical bus system with a single broadcast transmission medium to which all nodes are connected. This is the reason why the reader will rarely find the word "bus" in this document. FlexRay networks are not limited to simple busses. It is also possible to use active star couplers, similar to Ethernet topologies with hubs. Even the possibility to use different network topologies on the two channels A and B exists. When speaking of a FlexRay communication system, one usually calls it a FlexRay *cluster*.

A simple topology where each channel is connected to a bus can be found in Figure 2.1. The same logical topology with a different physical realization is the star topology in Figure 2.2. Finally, a mixed topology consisting of a bus and a star coupler is illustrated in Figure 2.3. Please note that FlexRay networks are not limited to the few examples here, there are some more possible topologies that will not be further discussed.
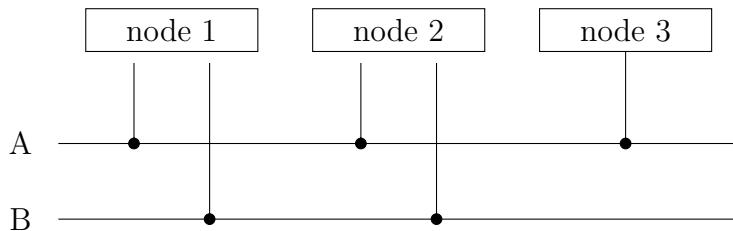
Figure 2.1: Two channel bus topology

2.3 The Communication Cycle

A FlexRay cluster transfers data in cycles. The frequency of the cycles is defined by the cycle time of the cluster, which can be configured at configuration time. Cycles are numbered from 0 to 63, numeration starts again at 0 after 64 cycles. Figure Figure 2.4 shows the visualization of the cycle time at a real FlexRay node that has been captured using an oscilloscope.

One whole communication cycle consists of four parts, a static segment, a dynamic segment, the Symbol Window and the Network Idle Time (NIT), see Figure 2.5. The static segment and the Network Idle Time are mandatory for every FlexRay cluster while dynamic segment and Symbol Window are optional.
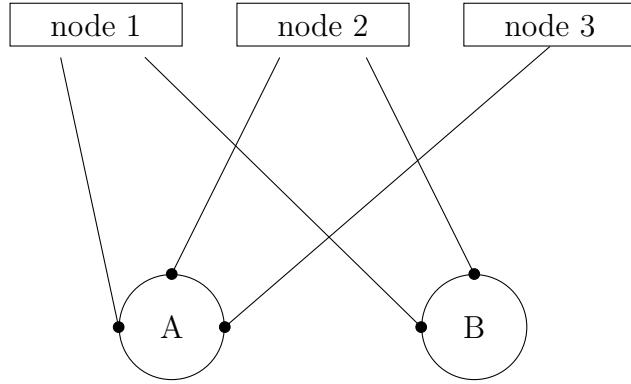
Figure 2.2: Star topology with two star couplers
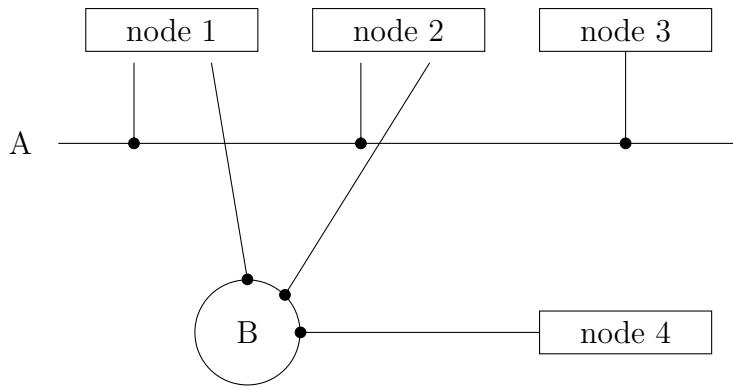
Figure 2.3: Mixed topology, channel A on bus, channel B on star coupler

The Symbol Window is used to transmit symbols like the Media Test Symbol (MTS), which can be used to validate that the all nodes are correctly operating, but their scope is beyond this document. Interested readers may refer to the FlexRay Protocol Specification [PS 05]. The Network Idle Time is a phase in the communication cycle where no communication takes place. The time can be used by the FlexRay Communication Controller to perform calculations that cannot be done during normal communication, such as running algorithms to synchronize the time on all nodes.

Of greater interest for the user of the protocol are the concepts of the static and the dynamic segment. Both of them will now be explained in detail.

2.3.1 Static Segment

The static segment uses Time Division Multiple Access (TDMA) to reserve bandwidth for nodes. TDMA allows multiple users to access a shared medium by dividing the time into several, usually same-sized, slots. Each node is assigned one or more of these slots. Only the "owner" of a slot may send in it, see Figure 2.6. One important prerequisite for TDMA to work is a common understanding of time. The clocks of all participants must be synchronized, so that each one can send in the correct slot and at the correct slot-boundaries.
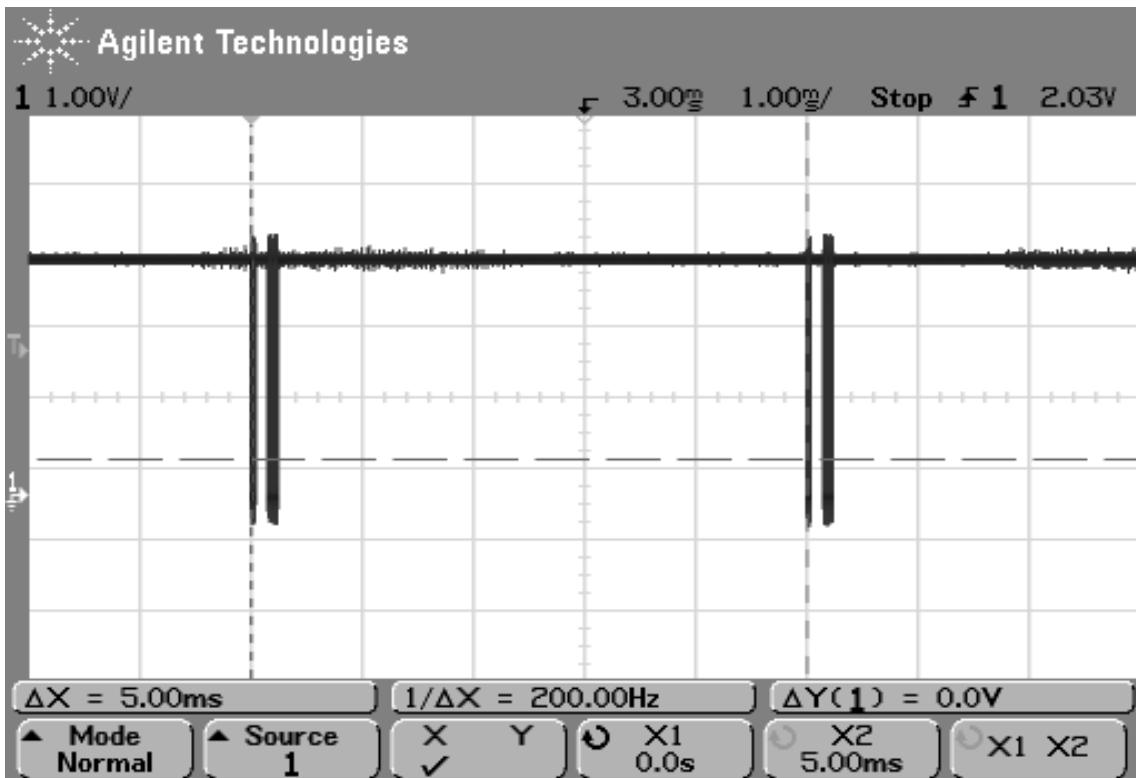
Figure 2.4: Visualization of the cycle time, which is $5ms$ (or 5000 macro ticks) for this example

TDMA is exactly the procedure that is used in the static segment of FlexRay to allow guaranteed, contention-free[1] communication. It is important to note that the static segment is usually configured before deploying a FlexRay system. As soon as the communication is running, the configuration of the static segment cannot be changed anymore[2]. Essential for the organization of the static segment is the division of the segment into slots. These slots are all of the same length. The number of static slots can be configured for a FlexRay cluster using the variable `gNumberOfStaticSlots`. Each slot carries exactly one FlexRay frame which usually contains payload. The

---

[1]Contention-free means that there is no competition among senders, slots are exclusively reserved.

[2]Except when the communication is stopped, then reconfiguration is possible.

static segment	dynamic segment	symbol window	NIT

$\longleftarrow$ cycle time $\longrightarrow$
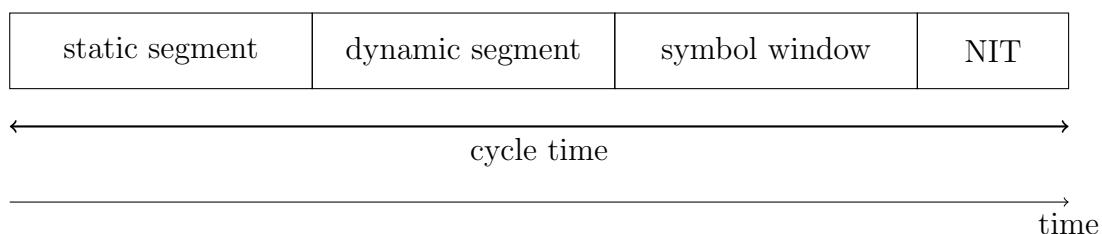
time

Figure 2.5: Contents of one FlexRay communication cycle
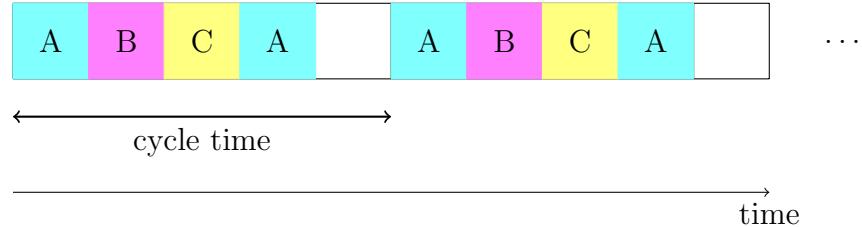
Figure 2.6: The principle of Time Division Multiple Access (TDMA), there are three participants, A, B and C. Note that A has two time slots per cycle reserved and that the fifth slot is unused.

frame-ID of this frame is always equal to the id of the slot it is sent in. In slot 1 only frames with an ID of 1 will be sent, in slot 2 all frames have the ID 2 and so on.

The length of each slot is given by the configuration variable `gdStaticSlot` and needs not only be long enough to carry a whole FlexRay frame but also an idle delimiter as well as some safety margin for the cluster to operate properly. See Figure 2.7 for a visualization of the static segment within one communication cycle. The first slot in the static segment is always numbered one, not zero.
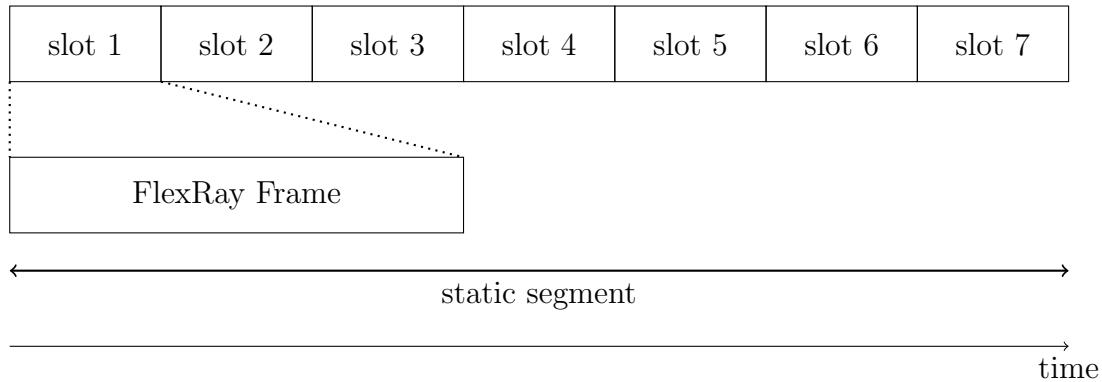
Figure 2.7: Contents of the static segment of an example communication cycle

When designing a FlexRay system, especially the static segment must be tailored well for the intented usage-scenario. This process includes assigning static slots to nodes. Keep in mind that FlexRay provides two physical channels. During each slot, channel A and channel B can be used to transmit data. For improved redundancy a node may use both channels for the same data, but may also transmit different data on each channel. A slot can even be used by two nodes, one channel for each node.

Not all static slots need to be used, of course. There may also be empty slots where no frames will be sent in. Figure 2.8 shows how such a typical assignment of slots might look like.

2.3.2 Dynamic Segment

The dynamic segment is more complex as the static segment as it does not use a simple access schema like TDMA, but a schema that implements contention among

	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7
A	node 1		node 2	node 4	node 5	node 1	
B	node 1		node 3			node 1	node 6

static segment
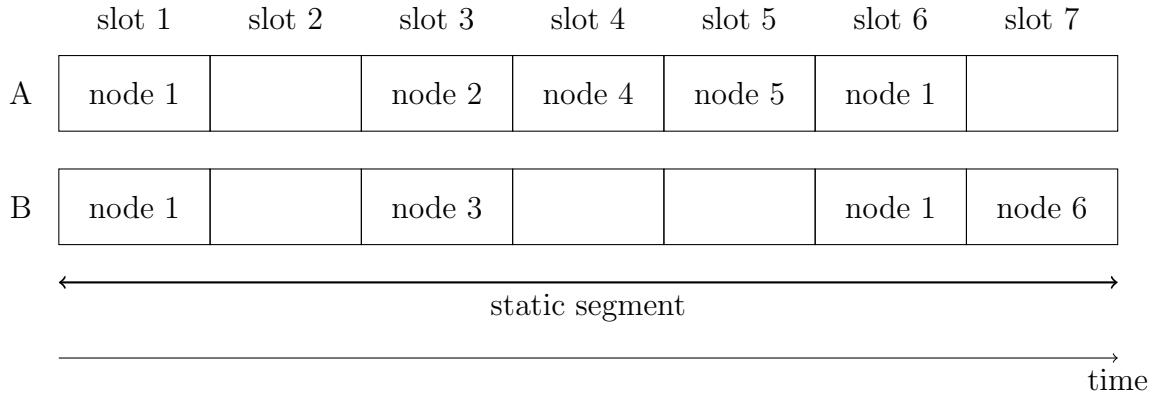
time

Figure 2.8: Assignment of slots (on two channels) to nodes of the cluster

nodes. This allows a priority-based access similar to that of Controller Area Network
(CAN). The method used for FlexRay is also called the Minislot Procedure, as the
time of the dynamic segment is also slotted into a number of Minislots (given by the
configuration variable `gNumberOfMinislots`).

During the dynamic segment, a priority based access schema is used. Numbering of
the dynamic slots continues the numbering of the static slots seamlessly, so if the last
static slot was numbered 60, the first dynamic slot will be numbered 61. A node may
send in the dynamic segment, when its slot counter matches the ID of the frame to
be sent. It is important to notice, that each frame sent in the dynamic segment may
occupy *several* Minislots (in contrast to the static segment, where exactly one slot
must be used). The slot counter is increased as soon as a frame has been completely
sent, so there is no direct mapping of minislot numbers to slot numbers. If a slot is
not used, it will only "waste" one (quite small) minislot before the next slot begins.
It can easily be seen, that higher slots IDs have lower priority. If there is much to
send in lower slots, the higher slots might just "drop out" of the dynamic segment,
which means the slot counter will never reach a slot ID of e.g. 70 and so a node
using slot 70 may not send in that particular cycle. See Figure 2.9 for a visualization
of the communication in the dynamic segment.

Minislot numbers

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Slot 61	Slot 62		Slot 63		Slot 64	Slot 65	Slot 66	Slot 67		Slot 68	Slot 69	Slot 70	Slot 71

dynamic segment

time
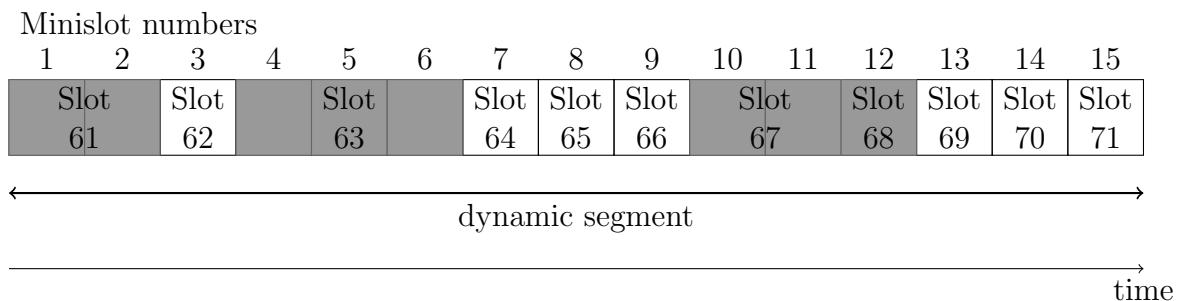
Figure 2.9: Example communication in the dynamic segment. Only one channel is
depicted.

2.4 Frame Format

This sections describes the layout of FlexRay frames as they are transmitted in either static or dynamic slots. The description of the frame-format can be found in the FlexRay Communications System Protocol Specification [PS 05], each frame consists of a header, the payload and a trailer containing a checksum, see Figure 2.10.
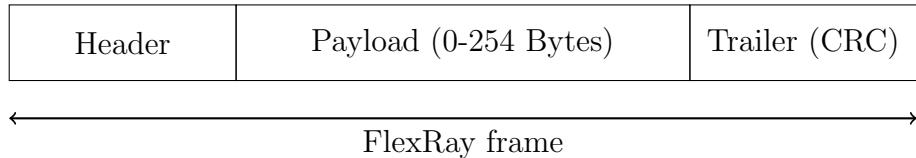
Figure 2.10: Layout of one FlexRay frame

2.4.1 Frame Header

The header has an overall size of 5 Bytes. Figure 2.11 shows the format of the Frame Header. The fields in it will now be explained in detail.
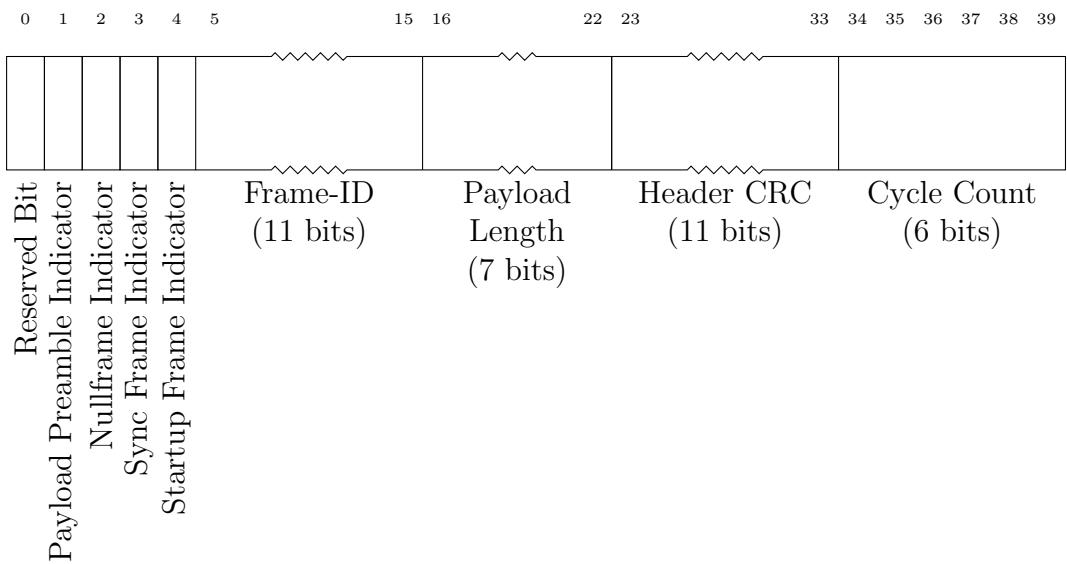
Figure 2.11: Format of the frame header

2.4.1.1 Reserved Bit (1 bit)

The first bit in the frame is a bit that has been reserved for further expansions of the protocol. For protocol version 2.1, a sending node always sends a zero but a receiving node must accept both zero and one in this field.

2.4.1.2 Payload Preamble Indicator (PPI) (1 bit)

The Payload Preamble Indicator indicates the presence of special control information in the frame. For the static segment, this bit indicates whether there is a Network

Management Vector contained in the payload. Network Management Vectors can be used for powersaving purposes but are optional to implement according to the FlexRay protocol specification. In the dynamic segment the bit is set to tell the receiver that a message identifier is present in the payload, the purpose of this message identifier is not discussed here.

2.4.1.3 Null Frame Indicator (1 bit)

If the Null Frame Indicator bit is set to *zero*[3], the frame contains no usable payload. The payload bits are all set to zero in this case. There are several situations in which null frames may be sent, e.g. during startup or when no new data is ready to be transmitted in the static segment.

2.4.1.4 Sync Frame Indicator (1 bit)

A set Sync Frame Indicator bit indicates that this frame is a sync frame which is used to synchronize the whole FlexRay cluster. Note that if a FlexRay node sends sync frames (i.e. it is a sync node), it does this in one slot, the so-called Keyslot.

2.4.1.5 Startup Frame Indicator (1 bit)

The Startup Frame Indicator is set to one for startup frames. Startup frames are needed during startup of the cluster and are only sent by coldstart nodes.

2.4.1.6 Frame ID (11 bits)

The Frame ID reflects the slot number in which the frame is transmitted. The ID has a length of 11 bits, thus theoretically 2048 different frame IDs are possible, but ID 0 is invalid.

2.4.1.7 Payload Length (7 bits)

The Payload Length field is seven bits long and describes the number of 16 bit words in the payload. This value must be multiplied by two to get the size of the payload in bytes. Values from 0 to $2^7 - 1 = 127$ are possible, thus the maximum payload per frame is 254 bytes.

2.4.1.8 Header-CRC (11 bits)

The Header-CRC is caculated over the Sync Frame Indicator, Startup Frame Indicator, Frame-ID as well as the Payload Length. The CRC polynomial is defined as

$$x^{11} + x^9 + x^8 + x^7 + x^2 + 1 \tag{2.1}$$

which will be represented as 101110000101b = 0xB85. Header-CRC generation assumes a fixed initialization vector of 0x1A. A reference implementation of the checksum algorithm can be found in Listing A.1 on page 53.

2.4.1.9 Cycle Count (6 bits)

The Cycle Count field contains the value of the cycle counter at the transmitting node, it ranges from 0 to 63.

---

[3]Note the inverted semantics here!

2.4.2 Frame Trailer

After the payload, the Frame Trailer follows. It contains another CRC-sum, the Frame CRC. This CRC is calculated over the whole frame. The polynomial is defined as

$$
\begin{aligned}
x^{24} + x^{22} + x^{20} + x^{19} + x^{18} + x^{16} + x^{14} + \\
x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^6 + x^3 + x + 1
\end{aligned}
\tag{2.2}
$$

which will be represented as 101011101011011011001011b = 0x15D6DCB. The trailer CRC uses different initialization vectors[4] for the two different channels A and B. Channel A uses 0xFEDCBA while channel B uses 0xABCDEF. This measure makes the system robust against accidently interchanging the two channels.

2.4.3 Frame Encoding

Before a frame is physically transmitted on the medium, a Transmission Start Sequence (TSS) is sent out. The TSS consists of a configurable time of low level on the medium. It is used to activate bus drivers and active star couplers on the cluster. After the TSS follows the so-called Frame Start Sequence (FSS), a single high-bit. Then the real frame data including header, payload and trailer follows byte per byte. Each byte is prepended by a Byte Start Sequence (BSS), consisting of a high bit directly followed by a low bit. The purpose of the BSS is to avoid long sequences of high or low bits that make synchronization of the receiver difficult. At the end of the data, another sequence, called the Frame End Sequence (FES) is sent out. This sequence is formed by a low bit followed by a high bit. If the frame is sent in the static segment the transmission is over now, see Figure 2.12.

For a frame in the dynamic segment, a little more has to be done. After the FES, a Dynamic Trailing Sequence (DTS) has to be sent to extend the duration of the transmission till the next minislot begins, see Figure 2.13.
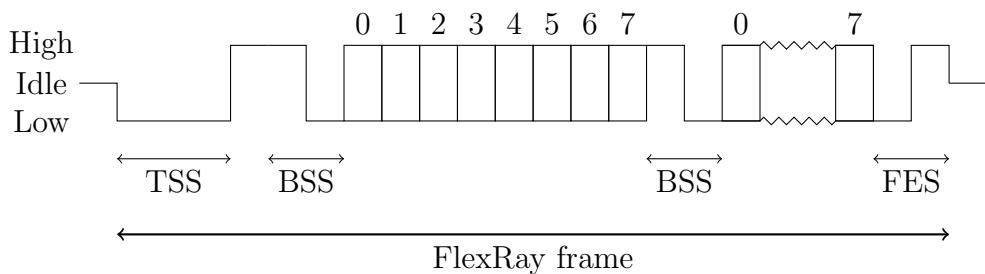

Figure 2.12: Encoding of a frame sent in the static segment

2.4.4 Example Frame

A typical Startup Frame, captured on the physical layer, can be seen in Figure 2.14. The Null Frame Indicator is zero, so there will be only zeroes as data. Both Sync

---

[4]The shift-register used to compute the CRC sum is initialized with this value before shifting any data in.
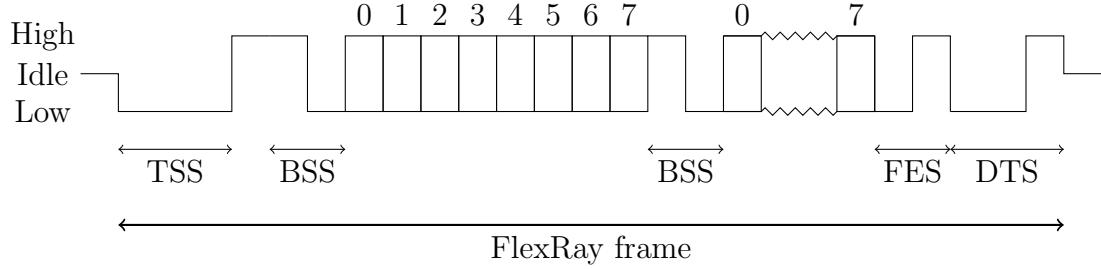
Figure 2.13: Encoding of a frame sent in the dynamic segment

Frame Indicator and Startup Frame Indicator are set so this frame will be used to synchronize clocks as well as for synchronization during the startup phase. The Frame-ID is set to 4 for this frame because the frame is sent in slot 4 (this has been configured a priori). The Payload Length indicates $16 \cdot 2 = 32$ bytes of payload. The following 11 bits form the CRC-sum which equals 0x6D3. The Cycle count is zero which means that this is the first Startup Frame sent by this node. The 32 data bytes announced in the header can be counted in the upper part of the figure, it is followed by a trailer containing a CRC-sum for the frame. Figure 2.15 shows the end of the Startup Frame with the trailer containing the 24 bit long CRC-sum which is 0x955F4E in this example (this data has been collected on channel A, so the sum is correct).
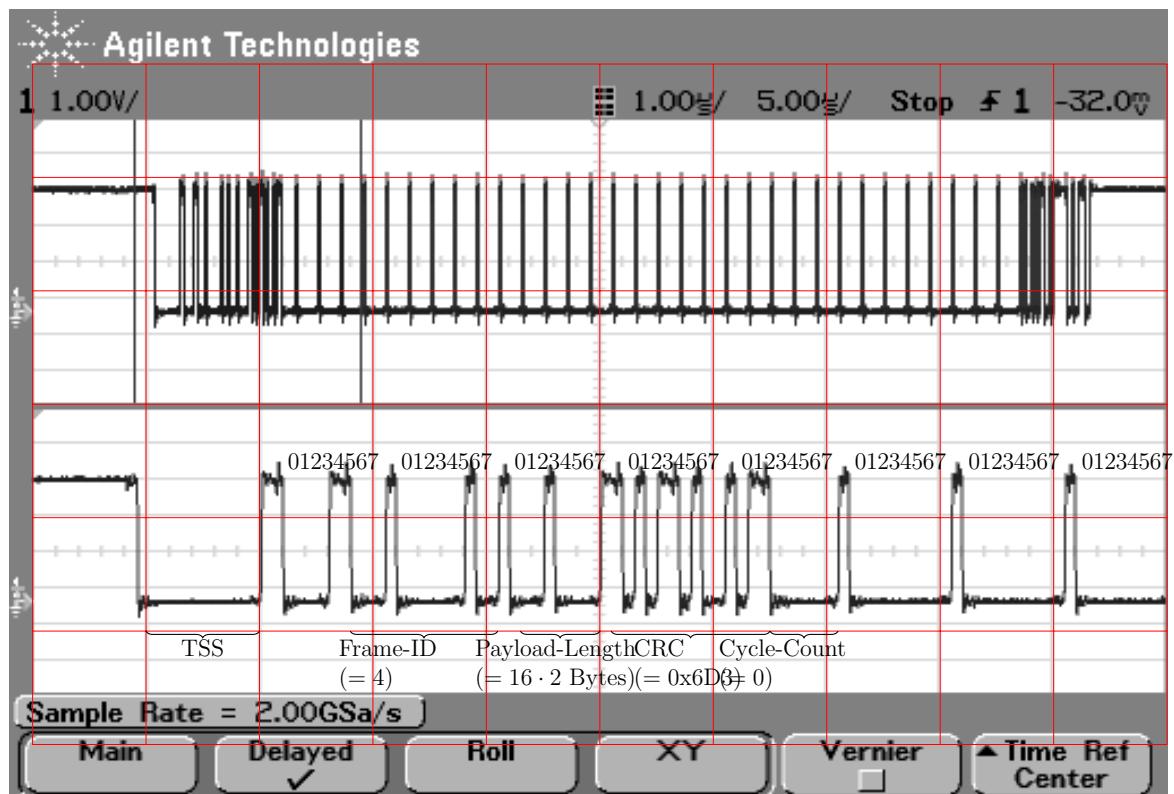
**Figure 2.14:** Startup Frame with magnified frame header in the lower part of the image
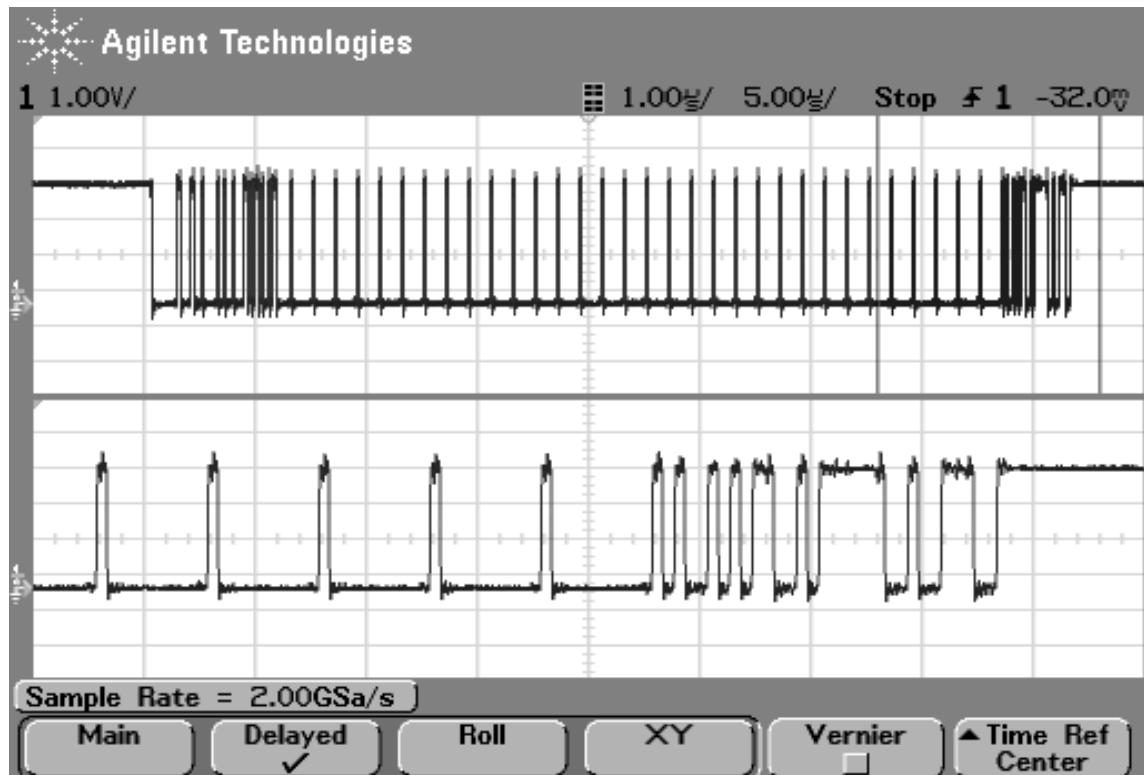
Figure 2.15: Startup Frame with magnified frame trailer in the lower part of the image

3. Hardware Components

After the general description of the principle of operation of FlexRay, the specific implementation that has been used during the research for this thesis will now be explained. The experimental boards that carry the FlexRay Communication Controllers and that have been used throughout the work will also be presented. Note that these board have been designed before this thesis and are not part of the work.

3.1 Experimental Board

For the research, two identical experimental boards served as the hardware to tinker with. Each of the board hosts a MC56F8357 digital signal processor, a MFR4310 FlexRay Communication Controller, an Altera MAX II CPLD, 512 KiB SRAM, as well as some other components. All important components such as DSP and FlexRay CC will introduced in the following sections.

The logical layout of the board can be seen in Figure 3.1, see Figure 3.2 for an image of the real board.
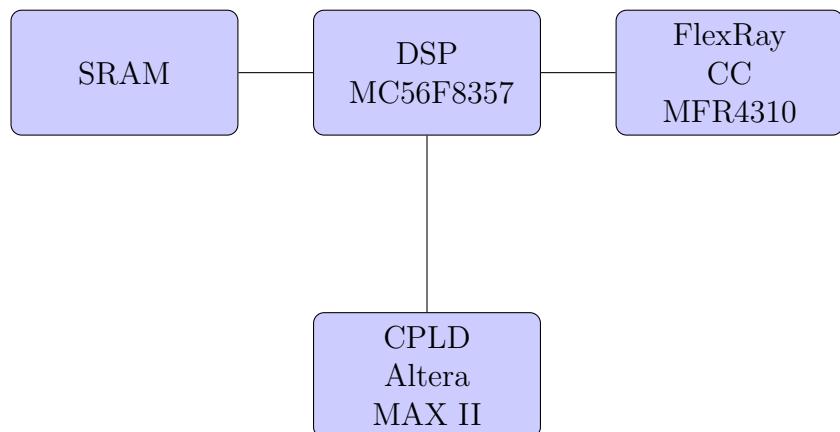
**Figure 3.1:** Logical layout of the experimental FlexRay board
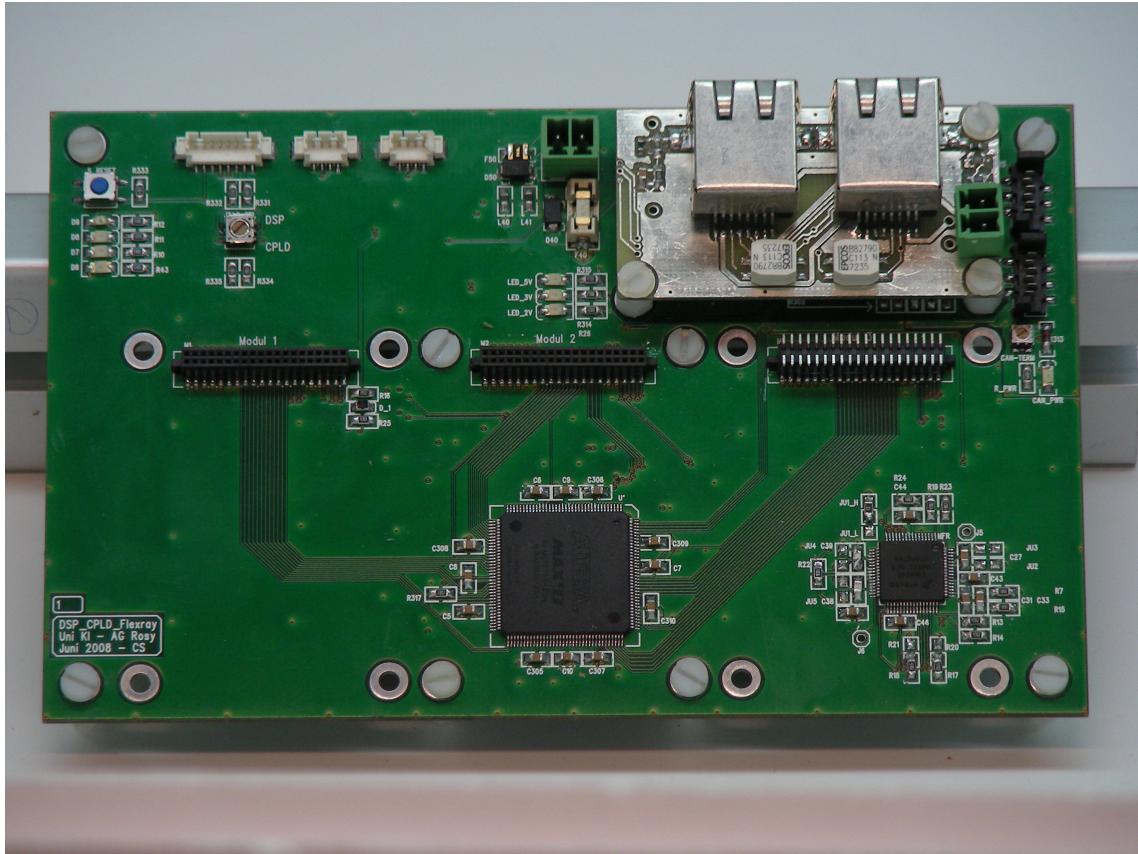
Figure 3.2: Top of the experimental FlexRay board

3.2 Freescale MC56F8357 DSP

The MC56F8357 is a 16bit digital signal processor from freescale semiconductor. It is capable of executing up to 60 MIPS[1] at 60MHz core frequency. Internal peripherals include 16kB data RAM, 256kB program flash, 2x6 PWM channels, 4 timers and 1 CAN (Controller Area Network) controller [56F8357 07].

3.2.1 Memory Layout

The DSP makes use of some external devices in memory mapped mode, such as the external RAM which is connected to the chip-select pin $\overline{\text{CS0}}$. The external RAM starts at address 0x40000 and and ends at 0x7FFFFF, 512 KiB later. Unfortunately the RAM on the board or the connection seems to be broken as it is not working reliably, thus it remains unused.

The FlexRay Communication Controller is connected to chip-select $\overline{\text{CS4}}$ and mapped to the addresses 0x3000-0x4FFE.

TODO: insert image of memory map

---

[1]Million Instructions Per Second

3.3 Freescale MFR4310 FlexRay Communication Controller

Freescale's MRF4310 is a FlexRay Communication Controller. It has support for variable bit rates such as 2.5, 5, 8 or 10 MBit/s. Three selectable host interfaces provide connectivity to microcontrollers: the HCS12 interface, the Asynchronous Memory Interface (AMI) as well as the MPC Interface [MFR4310 08].

The controller provides no physical layer interface so an external physical layer interface has to be used, see the next section about the physical layer.

3.4 Physical Layer Interface

The FlexRay Physical Layer is specified in [EPL 06], it specifies what cables should be used, how bus drivers should be designed and much more. Signals are always transmitted differentially over two wires to be more robust against external disturbances.

TODO: what transceiver exactly is used, cite datasheet, draw image illustrating the transceiver.

3.5 FlexRay UNIFIED Driver

Fortunately, Freescale, the manufacturer of the MFR4310 FlexRay Communication Controller has released a driver for a variety of their FlexRay controllers. This driver is called the *FlexRay UNIFIED Driver* [Freescale 06]. The driver code runs well on the MC56F8357 DSP.

The goal of this driver is to provide a hardware independent API to various Freescale FlexRay Communication Controllers including standalone controllers as the MFR4310 employed here as well as microcontrollers with integrated FlexRay modules.

To use the driver, the Communication Controller should first be hardware reset. This is not part of the driver but can easily be done manually by triggering the reset pin of the controller. The first action by the driver is to use `Fr_init` to put the module into the configuration state *POC:config*. In this state, `Fr_set_configuration` sets the configuration parameters. The function `Fr_buffers_init` configures the necessary message buffers, according to the configuration given.

For each message buffer, a callback function may be registered using `Fr_set_MB_callback`. This function will then be called when data has been received for the buffer or new data can be sent (in case of a transmission buffer).

`Fr_leave_configuration_mode` has to be used to exit the *POC:config* and enter the *POC:ready* state. In this state, the controller will wait for a successful startup of the cluster. Manually stopping the FlexRay communication can be achieved by using `Fr_stop_communication`.

3.5.1   Message Buffers

Reception and transmission of FlexRay frames is realized using message buffers. These message buffers are a little non-intuitive thus they will be discussed in detail.

First, there exist four types of buffers that can be configured using their corresponding configuration-datatypes:

- Transmit buffers

- Receive buffers

- FIFO receive buffers

- Receive shadow buffers

Some configuration options are present in all types of message buffers[2]. Each message buffers must have a slot ID (or even ranges of IDs in the case of FIFO message buffers) as well as a channel-type (A, B or AB) assigned to define when and on which channel reception of transmission takes place. Receive and transmit buffers also provide a filter configuration that is matched against the current cycle.

Transmit message buffers are used to transmit data to the FlexRay cluster, they are configured using the struct `Fr_transmit_buffer_config_type`. Configuration of a transmit message buffer includes payload length, buffer type (single of double buffered), transmission mode (i.e. state or event driven transmission) and some more. If double buffering is used, the buffer will occupy *two* message buffers. Receive buffers can be used for receiving data from the FlexRay cluster. They are configured with `Fr_receive_buffer_config_type`.

Transmission buffers configured for state driven transmission are automatically transmitted on every cycle, regardless of whether the buffer contents have been updated or not. On the other hand, event driven buffers will only be transmitted when the buffer contents have been updated. If no new data is available a Nullframe is sent, in case the buffer is configured for the static segment or nothing is sent for the dynamic segment.

Finally, receive shadow buffers must be configured. They play a special role as they are not user-accessible like the three buffers introduced before. These buffers are used internally for the reception of frames from the two channels. As soon as a frame has been received correctly it is transferred from the receive shadow buffer to the individual message buffer that has been assigned to that frame. Configuration is done using the struct `Fr_receive_shadow_buffers_config_type`. It is important to notice, that receive shadow buffers must be configured to ensure proper operation of the node.

As soon as a configuration for each message buffer is ready in memory an array of `Fr_buffer_info_type` structs must be created that contains the buffer-type, a pointer to the configuration-struct as well as the index of the message buffer that should be used as the buffer. Here again, it is important to note, that double

---

[2]Excluding shadow buffers which serve a special purpose

transmit buffers need two buffer places, thus the message buffer index for the next buffer must be increased by two instead of one.

Finally – to make things even more complicated – another array of types `Fr_index_selector_type` needs to be created that contains only integers that represent indexes to the `Fr_buffer_info_type` array. Each buffer that is selected using the selector will be used later. In the case of the FlexRay 4 Linux integration only buffers that are really needed are added to the `Fr_buffer_info_type` array, so that the `Fr_index_selector_type` array basically contains indexes for all buffers.

3.5.2 Cycle Filter Configuration

The MFR4310 provides a cycle filter that can be used for transmit as well as for receive buffers. The configuration is done using three parameters. The first is just a boolean that switches the filter on or off, the other two are called *filter value* and *filter mask*. Filter value and filter mask are each 6 bit wide (as the cycle counter). If the filter is enabled the following condition must hold for the filter to match:[3]

$$\text{cycle} \wedge \text{mask} = \text{value} \wedge \text{mask} \tag{3.1}$$

Consider a mask of 0x1 = 000001b and a value of 0x0 = 000000b, then the left side of Equation 3.1 will be 000001b iff the cycle is odd. The right side is always 000000b as $0x0 \wedge 0x1 = 0x0$. As a result this filter will only match for even cycles. When a value of 0x1 = 000001b is chosen, the filter matches only the odd cycles. Some more examples of using the cycle filter can be found in Table 3.1.

3.5.3 Transmitting and Receiving

Transmitting and receiving is done using the previously configured message buffers. Unfortunately there seems to be no way to transmit/receive by slot ID in the driver. Transmitting and receiving can only be done by specifying the correct buffer index that has been assigned before. To transmit or receive by slot ID nonetheless, a mapping between slot ID and message buffer is needed. Fortunately the `Fr_buffer_info_type` array contains all information needed to map the values.

---

[3]Note that the meaning of the logical-and "∧" has been extended to bitvectors instead of single bits for the equation.

Cycle	M/V 0x1 / 0x0	M/V 0x1 / 0x1	M/V 0x2 / 0x0	M/V 0x2 / 0x2	M/V 0x3 / 0x0	M/V 0x3 / 0x1	M/V 0x3 / 0x2	M/V 0x3 / 0x3
0	✓		✓		✓			
1		✓	✓			✓		
2	✓			✓			✓	
3		✓		✓				✓
4	✓		✓		✓			
5		✓	✓			✓		
6	✓			✓			✓	
7		✓		✓				✓
8	✓		✓		✓			
9		✓	✓			✓		
10	✓			✓			✓	
11		✓		✓				✓
12	✓		✓		✓			
13		✓	✓			✓		
14	✓			✓			✓	
15		✓		✓				✓

Table 3.1: Filter matching examples, *M/V* denotes *mask* and *value* pairs

4. FlexRay 4 Linux

To integrate FlexRay communication into personal computers it must somehow be connected to it. There exist commercial solutions to interface FlexRay systems using USB or PCI[1]. These solutions are very powerful but also quite expensive and they do only provide closed source drivers for Microsoft Windows. Features of these solutions include the ability to monitor even the startup phase were no synchronization has been established yet [Vector 08].

4.1 FlexRay over Serial Links

To encapsulate the FlexRay protocol over serial links such as USB or serial ports, many different operations must be put into a serializable format. This is realized using a simple packet-based protocol that will now be explained.

4.1.1 Packet Format

The protocol uses a specific packet for each operation. These packets can then be transmitted byte per byte over a serial link. Each packet carries a header that contains a one-byte field that is used to encode the type of operation as well as optional arguments, see Figure 4.1. The packets do not carry a length field, because the length can be calculated using the operation header and the length-field of the FlexRay frame in case of a data packet.

There exists the following types of packets:

- Packets without additional arguments (init, start communication, stop communication, wakeup, enter config, leave config), Figure 4.2

- Set low level configuration, Figure 4.3

- POC state, Figure 4.4

- Add RX/TX buffer, Figure 4.5

- Frame data, Figure 4.6

---

[1]E.g. Vector VN3600 or Vector VN3300

Operation (8 bit)	Arguments (optional, size variable)

Figure 4.1: Format of a generic FlexRay over Serial packet

Operation (8 bit)

Figure 4.2: Format of the FlexRay over Serial packet without arguments, it just contains the operation header-field.

4.1.2 FlexRay over UART

If the UART is chosen as the low level communication medium, data units are typically chunks of 8 bits (1 byte). The FlexRay over Serial packets just discussed consist of up to several hundred bytes, but now have to be transmitted byte-per-byte. Because a simple two-wire UART cannot signal packet starts using dedicated wires, in-band signaling is needed to properly separate packets from each other. This is done by encasulating the FlexRay over Serial packets into FlexRay over UART frames. Those frames contain a 16 bit length field, the payload (which is always a FlexRay over Serial packet) and a checksum to validate the payload, see Figure 4.7. The start of a new frame is signalled using a reserved START-byte. Whenever a START-byte is encountered in the data-stream, this START-byte needs to be escaped by an escape byte ESC followed by the byte ESC_START. Whenever a ESC-byte in found in the data-stream, it will be replaced by ESC, followed by ESC_ESC. At the receiver-side, the START-byte can then be used to detect the beginning of a frame. Before passing the data to the upper layers, the receiver has to recover the escaped bytes. To sum up:

- START marks the beginning of a frame

- START in the data is escaped as ESC, ESC_START

- ESC in the data is escaped as ESC, ESC_ESC

4.2 An Introduction to the Linux Kernel

To goal of this section is *not* to give a an explanation about how operating systems work. It shows merely how some typical functionality is implemented in the Linux Kernel.

Operation (8 bit)	Low-Level Conf. (72 bit)

Figure 4.3: Format of the FlexRay over Serial Low-Level configuration packet, operation is always set to FR_SERIAL_PACKET_OPERATION_SET_LOW_LEVEL_CONFIG for these packets.

Operation (8 bit)	POC-State (8 bit)

Figure 4.4: Format of the FlexRay over Serial POC-state packet, operation is always set to `FR_SERIAL_PACKET_OPERATION_POC_STATE` for these packets.

Operation (8 bit)	TX/RX Buffer Conf.. (13/7 bit)

Figure 4.5: Format of the FlexRay over Serial Low-Level configuration packet, operation is always set to `FR_SERIAL_PACKET_OPERATION_ADD_TX_BUFFER` or `FR_SERIAL_PACKET_OPERATION_ADD_RX_BUFFER` for these packets.

4.2.1 Loadable Modules

If configured and compiled with the option `CONFIG_MODULES`, the Linux Kernel can dynamically link additional code during runtime. This code is available in the form of Kernel modules. Modules can provide almost any functionality, they can for example add new network protocols or devices drivers for new network interfaces to the Kernel. Modules can be loaded using the commands `insmod` or `modprobe`, the first one needs a path to the module while the latter searches in well-known paths for the module and can also automatically load dependencies. The command `rmmod` is used to unload modules, as long as they are not currently in use.

When modules are loaded, a function declared with `module_init()` is executed where e.g. data-structures can be initialised. When the module is unloaded, the function passed to `module_exit()` is executed, here cleanup operations, such as freeing allocated memory should be performed.

4.2.2 Linux Network Interfaces and Sockets

Linux supports a lot of different network interfaces, including typical every-day hardware such as ethernet adapters, wireless LAN cards, but also virtual interfaces like the TUN/TAP devices that are e.g. used in virtual private network solutions. Every network device can be identified by the user by its name, for ethernet this is `eth0` for the first ethernet device, `eth1` for the second, . . . , for wireless lan usually `wlanX` and for tun-devices `tunX` (with $X \in \mathbb{N}_0$). Every network interface in the system internally gets a number (called the interface index – `ifindex`) that can be used within the kernel (or even in userspace as can be seen later) to uniquely identify it.

Operation (8 bit)	FlexRay Frame (size variable)

Figure 4.6: Format of the FlexRay over Serial data packet, data packets are used to encapsulate FlexRay frames. Operation is always set to `FR_SERIAL_PACKET_OPERATION_DATA` for these packets. FlexRay Frame is of type `flexray_frame`.

| Length (16 bit) | Payload (0...x Bytes) | CRC (16 bit) |

Figure 4.7: Format of the FlexRay over UART frames

Network interfaces can be configured using the tool `ifconfig` for devices that can handle the internet protocol it can e.g. be used to set the IP-address. Common for all devices is the ability to bring a network interface up or down, using the command `ifconfig devX up` respectively `ifconfig devX down`.

Userspace applications cannot use the network interfaces directly, there is no way to "open" e.g. `eth0`. Instead applications must create sockets using the `socket(domain, type, protocol)` call. The arguments specify for what type of protocol the socket is created. By using `socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)`, a typical TCP/IP socket is allocated while `socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)` creates a UDP/IP socket. The protocol-family is the same in both cases, `PF_INET` for IPv4 traffic. By replacing the protocol-family with `PF_INET6` IPv6 connections can be established.

There are several operations that can be done with sockets. Two of them are very important, `send()` and `recv()`. They are used for sending or receiving data to/from the socket. By using `send()`, data is passed over to kernel space and by using `recv()` data from kernel is passed into userspace. Another method to interact with sockets or the network devices that are "behind" the sockets is by using specific `ioctl()` commands, see the next section.

4.2.3 The `ioctl()` Syscall

The `ioctl()` call can be used to do dozens of operations with devices or also sockets. Almost anything that cannot be done semantically correct with `recv()` or `send()` calls can probably be done with `ioctl()`s. Consider for example setting the baudrate for a serial device. Theoretically this could somehow be encoded into a `write()` call, but that would probably easily cause trouble and especially confusion, because setting the baudrate has nothing to do with "writing" to a serial port. Similar problems arise for the FlexRay network device. Many tasks have to be accomplished that are neighter sending nor receiving, such as configuring the Communication Controller, reading POC-states or stopping the communication.

The `ioctl()` requests are fairly easy to understand, each request consist of a request-id to distinguish between different request-types and a `void *`[2] pointer to some memory, usually an application-specific struct. Because a pointer (or call-by-reference) is used, `ioctl()` is basically bidirectional, there exists request that only transfer data from user to kernel space, there are some that transfer data from kernel to user space or there are even some that do both at a time.

---

[2]Actually, it is not a `void` pointer, which has traditional reasons, but one could imagine it to be one.

4.2.4 The `select()` Syscall

When it comes to non-blocking input/output operations the `select()` call comes handy. It allows to wait for input, the ability to write or for exceptions on several file-descriptors (which include sockets) at the same time. Optionally a timeout for the wait-operation can be specified.

4.3 FlexRay 4 Linux

This section can only give a overview of the situation for – at the time of writing – current Linux versions (e.g. 2.6.28). Many parts of the Kernel API change from time to time so that this information here might easily be outdated. A great overview about Linux Kernel programming in general gives [Corbet 05][3], for Linux networking internals, see [Benvenuti 05].

4.3.1 The `PF_FLEXRAY` Protocol Family

FlexRay communication for Linux is implemented as a new network protocol. A network protocol matches the nature of a communication system such as FlexRay better than e.g. a character driver.The advantages of doing so a numerous – straight-forwarded calls such as `socket()`, `recv()` or `send()` can be used to establish communication and the whole interface can easily be used by several users or processes.

Linux provides several protocol families that lets the user create specific sockets to communicate using a large variety of protocols. Well known protocols include `PF_INET`, `PF_INET6` which provide sockets for the Internet Protocol (IP) version 4 respecively 6. There are also less known protocol families, e.g. `PF_BLUETOOTH` which implements important protocols for the Linux Bluetooth stack *BlueZ* or the quite interesting protocol family for the controller area network (CAN) named `PF_CAN`. All supported protocol families are defined in `linux/socket.h`.

Defining the protocol in `linux/socket.h` is of course not enough, the protocol family must also be implemented, e.g. in form of a Kernel module. When such a Kernel module is loaded it must register the protocol family it implements at the Linux networking core. This is done using the API function `sock_register` (see `net/socket.c`). This function takes a pointer to a `struct net_proto_family` which contains the number of the protocol family and a pointer to a function to create protocol specific sockets.

A typical protocol family registration could look like in Listing 4.1.

Listing 4.1: Registering a protocol family

```
/* the description of the protocol */
static struct net_proto_family fr_family_ops = {
        /* the protocol family, see linux/socket.h */
        .family = PF_FLEXRAY,
        /* a pointer to a function to create sockets */
        .create = fr_create,
        /* the "owner" of this protocol */
```

---

[3]This book is also available online under Creative Commons License.

```
           .owner = THIS_MODULE,
  };

10
  /* register it at the socket subsystem */
  sock_register(&fr_family_ops);
```

This is how the output looks when the protocol is registered and unregistered:

```
# insmod flexray.ko
flexray: Initializing FlexRay4Linux protocol stack
NET: Registered protocol family 36
# rmmod flexray
NET: Unregistered protocol family 36
flexray: Successfully unloaded.
```

A "Ethernet" protocol ID must be registered in `linux/if_ether.h`. This is named `ETH_P_FLEXRAY`. This is needed to register handlers for the new protocol family. `af_flexray` registers this handler to receive the FlexRay frames from the network driver.

4.3.2   Raw `FR_RAW` Sockets

Once the protocol family has been registered at the socket subsytem, single protocols for this protocol family can register themselves in turn at the protocol family.

Particularly useful protocols are *RAW* protocols that can be used for debugging because they usually pass more data to the user than neccessary during normal operation. The FlexRay subsystem also provides a raw protocol named `FR_RAW` that can be used to receive and send on all registered FlexRay slots using specially crafted structs.

The FlexRay core exports the symbol `fr_proto_register` which must be used to register FlexRay protocols, see Listing 4.2 for an example on how to register the raw protocol.

Listing 4.2: Registering a FlexRay protocol

```
  /**
   * description of the FR_RAW protocol
   */
  static struct fr_proto raw_fr_proto = {
5         .type     = SOCK_RAW,
          .protocol = FR_RAW,
          .ops      = &raw_ops,
          .prot     = &raw_proto,
  };
```

4.3.3   FlexRay Network-Devices

The identifier `ARPHRD_FLEXRAY` must be defined in `linux/if_arp.h`.

TODO: some more words, how does it work

4.3.4 Serial Line Discipline

When the FlexRay controller is connected to a serial port (a so-called "tty") of the Linux system it is usually not available for use within a Kernel driver but only for userspace using the corresponding device-file, e.g. `/dev/ttyS0` for the first serial port. Note that this does not only apply for the classic RS232-style serial ports still found on older hardware but also for other serial devices such as USB-Serial adapters (`/dev/ttyUSBx`), or even cellphones (`/dev/ttyACMx`). Luckily, a mechanism called *Serial Line Discipline* is available in the Linux Kernel that allows to virtually "connect" these usually only userspace accessible devices to kernel drivers. A prominent example for a line discipline is the *Serial Line Internet Protocol* (SLIP) [Romkey 88] discipline (`N_SLIP`) which is used to transfer IP[4] data over serial links[5]. The SLIP-implementation takes serial data from the tty, extracts the IP packets and output them on a network interface (e.g. `slip0`). The other direction works similarly – the modules takes incoming IP packets from the network device, transforms them into a SLIP compatible data stream and outputs this stream on the tty.

Note the similarity of the problem of the SLIP implementation and the `flexray-serial` module. Both have to transfer data between a tty and a network device. The `slip` module for IP packets and the `flexray-serial` module for FlexRay frames.

So obviously, a line discipline is needed to connect FlexRay controllers over serial ports/ttys. The line discipline "Serial FlexRay" has been defined as `N_SLFLEXRAY` in `linux/tty.h`. The module `flexray-serial` registers the line discipline at initialization time, see Listing 4.3. The `tty_ldisc_ops` struct is used to set functions that are called by the kernel in case of operations on the discipline such as opening or closing the it or when data is available on the tty.

Listing 4.3: Registering the FlexRay line discipline

```
/**
 * serial line discipline options
 */
struct tty_ldisc_ops fr_tty_ldisc = {
        .owner        = THIS_MODULE,
        .magic        = TTY_LDISC_MAGIC,
        .name         = "flexray over serial",
        .open         = fr_tty_open,
        .close        = fr_tty_close,
        .ioctl        = fr_tty_ioctl,
        .receive_buf  = fr_tty_receive_buf,
        .write_wakeup = fr_tty_write_wakeup,
};


/* register the serial line discipline */
tty_register_ldisc(N_SLFLEXRAY, &fr_tty_ldisc);
```

---

[4]IP as in Internet Protocol

[5]Note that today, SLIP has been widely replaced by the *Point-to-Point Protocol* (PPP) [Simpson 94], e.g. for analog modem or DSL connections

So far, only the kernel side of serial line discpiplines has been discussed. However, line disciplines always also needs the userspace to be involved. As stated before, by default ttys are available to the userspace only, so the userspace has to explicitly pass the tty to kernel control by setting the appropriate line discipline for it. This can be done using a special `ioctl` request called `TIOCSETD`. This process is usually called "attaching" the tty to some kernel part, hence the userspace programs that are designed for this task are usually suffixed `attach`, e.g. for the already mentioned SLIP the program is called `slattach` and for FlexRay it will be called `frattach`. How such an program would use the `ioctl` can be seen in Listing 4.4.

Listing 4.4: Opening a line discipline from userspace (attaching)

```
int i = N_SLFLEXRAY;
if(ioctl(fd, TIOCSETD, &i) == -1) {
        perror("ioctl (set line discipline)");
        exit(1);
}
```

4.3.5 Important Data Types

4.3.5.1 struct flexray_frame

The `flexray_frame` struct is used to describe FlexRay frames that have been received from the FlexRay cluster or that are to be sent out to the cluster. This struct is used at two places: at the userspace/Kernel interface when sending/receiving frames as well as in the FlexRay over Serial data packet, when encapsulating FlexRay frames over serial links. See Listing 4.5 for the definition of the struct.

Listing 4.5: The `flexray_frame` struct

```
/**
 * This type describes a FlexRay frame.
 */
struct flexray_frame {

        /** ID of the slot the FlexRay frame has been sent in
          * or should be sent in */
        u16 slot_id;

        /** number of the cycle the FlexRay frame has been sent in */
        u8 cycle;

        /** length of the data in 16bit words (not bytes!) */
        u8 length;

        /** the status of the slot the frame was been received in */
        u16 slot_status;

        /** the payload */
        u16 data[127]; /* this must be the last field in this struct */
```

```
} __attribute__((__packed__));
```

4.3.6 Configuring the FlexRay Device

The FlexRay devices can be configured using the already introduced `ioctl()`s. The `ioctl()` is issued to an open `PF_FLEXRAY` socket. Usually `ioctl`s on sockets act just on the socket itself, but special requests have been allocated to control the underlying network device instead of the socket itself. These so-called "device private" requests are numbered from `SIOCDEVPRIVATE` to `SIOCDEVPRIVATE+15`.

The FlexRay for Linux framework only uses one `ioctl`-request which is called `SIOCDEVFLEXRAY`. This constant is defined as the first possible device private request. To distinguish between several requests to the FlexRay device, a special data structure named `flexray_conf_ioctl` is passed to the driver. This data structure contains a command-field that is used to encode different operations, e.g. initialization of the controller or entering the configuration mode. For more complicated requests that need to carry additional data, like configuration-descriptions, a pointer to more data is also included, see Listing 4.6.

Listing 4.6: FlexRay network device `ioctls`

```
#define SIOCDEVFLEXRAY SIOCDEVPRIVATE

struct flexray_conf_ioctl
{
        /** see below */
        u8 cmd;

        /** pointer to additional data */
        void *data;
};
#define FLEXRAY_INIT  0x1
#define FLEXRAY_SET_LOW_LEVEL_CONFIG 0x2
[...]
#define FLEXRAY_GET_POC_STATE 0xA
```

4.3.7 Sending and Receiving Frames

TODO: insert simple send(), recv() example here

4.3.8 Userspace Tools

During the implementation of the FlexRay functionality in Kernel space, some useful userspace tools emerged that can help debug a FlexRay cluster.

4.3.8.1 Userspace Library

To simplify the use of FlexRay under Linux, a simple userspace library has been created that can be used to hide the syscalls behind a C-interface.

TODO: write a lot more about the library here

4.3.8.2 FlexRay Sniffer: `flexdump`

The program `flexdump` is a simple sniffer, similar to the universal network sniffer `tcpdump`[6]. `flexdump` opens a raw FlexRay socket at startup and then just listens for changes in POC-state as well as for FlexRay frames. It then decodes received frames and outputs useful information such as:

- Size of the payload

- Slot-Number

- Cycle-Counter

- Flags (Valid, Nullframe, Startup, ...)

- Payload

Please note, that `tcpdump` itself can also be used to sniff on the FlexRay interface, e.g. by invoking it with `tcpdump -n -i flexray0`. This works, because `tcpdump` uses the `PF_PACKET` protocol family which registers ethernet protocol id `ETH_P_ALL` to receive *all* possible network traffic on the machine. Because `tcpdump` does not know about the FlexRay (at least at the time of writing) it can of course not decode the frames like `flexdump` does.

4.4 Demo Application

4.4.1 Overview

The demo applications consists of two major parts, a FlexRay controller connected to a Linux computer (node 1) and a another FlexRay controller in an embedded system (node 2). The embedded system has some components attached to it, a motor driver with a DC motor, a LED, a display and a switch. The embedded system uses the same experimental board as the FlexRay adapter for the computer, that has been introduced in chapter 3.

Both nodes are connected using a standard shielded twisted pair cable of category 5e and approximately 1 meter length. Both FlexRay channels (A and B) are available through the cable with the possibilty to manually interrupt one of the channels to demonstrate the failure safety of FlexRay.

4.4.2 Embedded System

The embedded system uses Pulse Width Modulation (PWM) to control the speed of the motor, the direction can also be changed. To achieve this, a 16 bit word is used. The highest bit specifies the direction of the motor and the lower 15 bits carry the PWM value between 0 and 32767. Pin `PWMA0` serves as the PWM source, `ISB0` and `ISB1` are used to set the direction of the motor.

The LED is connected to pin `ISA0` and the switch to `ISA1` of the DSP. The switch is pulled-low by a 22kΩ resistor, pressing the switch causes the pin to be held on high level.

---

[6]http://www.tcpdump.org/

A 2×20 character ASCII display with a HD44780-compatible controller is attached to the DSP using the pins `TD1`, `TD2`, `PhaseA0`, `PhaseB0`, `Home0` and `Index0`. It is interfaced in 4-bit mode and therefore it only needs 4 data lines.

TODO: move all the pin stuff in the appendix .. it's not that interesting

4.4.3   Linux Computer

The Linux computer runs Linux 2.6.28.7 with all the neccessary FlexRay patches applied. The serial FlexRay adapter is connected to it using a USB-Serial adapter of type Prolific PL2303.

A small graphical application, programmed in C using the GTK++-Toolkit, serves as an interface to the embedded system. The GUI shows the current POC-state and the value of the switch. A slider can be used to control the motor speed and direction, a button is used to switch the LED on or off. Text that should appear on the display of the embedded system can be entered into a text field. See Figure 4.8 for a screenshot. TODO: update screenshot
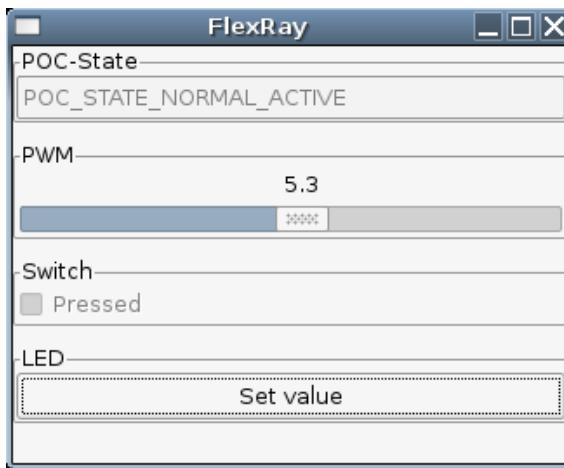
Figure 4.8: Screenshot of the GUI running on the Linux system

4.4.4   FlexRay Configuration

There are really a lot of parameters that configure a FlexRay cluster, compared to classic bus systems. These options make out a lot of the flexibility of FlexRay but are also quite confusing and hard to understand. This section cannot provide complete instructions on how to configure a FlexRay system, but it tries to give an introduction. For a deeper understanding, please refer to [Rausch 07], especially chapter 6.

4.4.4.1   Protocol Parameters

Here is the (non-complete) list of protocol parameter that are used on the demo cluster:

`gColdStartAttempts = 10` The nodes will try up to 10 attempts to cold-start the cluster

`gdActionPointOffset = 3` Send-start for static frames is 3 macroticks after beginning of slot

`gdCASRxLowMax = 83` Maximum length of a Collision Avoidance Symbol (CAS) is 83 bit times.

`gdDynamicSlotIdlePhase = 0` Length of the idle phase during a dynamic slot is 0 minislots.

`gdMinislot = 40` The duration of a minislot is 40 macroticks

`gdMiniSlotActionPointOffset = 3` Send-start for dynamic frames in 3 macroticks after beginning of slot

`gdStaticSlot = 50` The duration of a static slot is 50 macroticks

`gdSymbolWindow = 13` The duration of the symbol window is 13 macroticks

`gdTSSTransmitter = 11` The duration of the Transmission Start Sequence is 11 bit times

`gdWakeupSymbolRXIdle = 59` The minimal duration of the idle phase during a wakeup symbol at the receiver is 59 bit times

`gdWakeupSymbolRXLow = 50` The minmal duration of the low phase during a wakeup symbol at the receiver is 50 bit times

`gdWakeupSymbolRXWindow = 301` The width of the window to receive two wakeup symbols is 301 bit times

`gdWakeupSymbolTXIdle = 180` The duration of the idle phase of a sent wakeup symbol is 180 bit times

`gdWakeupSymbolTXLow = 180` The duration of the low phase of a sent wakeup symbol is 180 bit times

`gListenNoise = 2` The upper limit for the startup/wakeup listen timeout is 2

`gMacroPerCycle = 5000` A FlexRay cycle is 5000 macroticks long

`gMaxWithoutClockCorrectionPassive = 10` Node will go into passive state after $10 \cdot 2 = 20$ cycles without clock synchronization

`gMaxWithoutClockCorrectionFatal = 14` Node will halt after $14 \cdot 2 = 28$ cycles without clock synchronization

`gNumberOfMinislots = 22` There are 22 minislots in the dynamic segment

`gNumberOfStaticSlots = 60` There are 60 slots in the static segment

`gOffsetCorrectionStart = 4920` Start of offset correction in Network Idle Time is 4920 macroticks away from start of cycle

`gPayloadLengthStatic = 16` The payload length in the static segment is $16 \cdot 2 = 32$ bytes

`gSyncNodeMax = 5` The maximum number of synchronization nodes (i.e. nodes that send Sync Frames) is 5

`gNetworkManagementVectorLength = 2` The Network Management Vector is 2 bytes long

`pAllowHaltDueToClock = false` No direct transitions to *POC:halt* due to synchronization problems

`pAllowPassiveToActive = 20` The number of valid cycles to translate from *POC:normal passive* to *POC:normal active* is $20 \cdot 2 = 40$

`pChannels = CHANNEL_AB` The node is connected to both channels

`pdListenTimeout = 401202` The time to wait before initiating wakeup/startup is 401202 microticks

`pdMaxDrift = 601` The maximum clock drift between two unsynchronized nodes over one communication cycle is 601 microticks

`pExternOffsetCorrection = 0` The external offset correction is not used

`pExternRateCorrection = 0` The external rate correction is not used

`pKeySlotId = 1` The keyslot of this node (the Linux side in this example) is slot 1

`pKeySlotUsedForStartup = true` The keyslot is used for the startup of the cluster

`pKeySlotUsedForSync = true` The keyslot is used for clock synchronization

`pKeySlotHeaderCRC = 0xf2` The header checksum of the keyslot is 0x2f, this must be calculated using `pKeySlotId`, `gPayloadLengthStatic`, `pKeySlotUsedForStartup` and `pKeySlotUsedForSync`

`pLatestTx = 21` The number of the last minislot where a new transmission may begin in the dynamic segment is 21

`pMicroPerCycle = 200000` There are 200000 microticks in one cycle

`pOffsetCorrectionOut = 1201` The maximum allowed offset correction is 1201 microticks

`pRateCorrectionOut = 600` The maximum allowed rate correction is 600 microticks

`pSingleSlotEnabled = false` The Single Slot Mode, where only one slot is used, is disabled

`pWakeupChannel = CHAN_A` This node sends wakeup patterns on channel A

`pWakeupPattern = 16` The wakeup symbol is repeated 16 times

`pMicroPerMacroNom = 40` The number of microticks per macrotick that all implementations must support is 40

`pPayloadLengthDynMax = 64` The maximum payload length for dynamic slots is $64 \cdot 2 = 128$ bytes

4.4.4.2 Startup Configuration

Both nodes are configured as coldstarters, because at least two nodes must be configured as coldstarters to successfully start a FlexRay cluster. Node 1 has its keyslot set to 1 while node 2 uses slot 2 as keyslot. Both nodes attempt up to 20 coldstarts (see protocol parameter `gColdStartAttempts = 20`).

4.4.4.3 Static Segment Configuration

There are 60 static slots available in each communication cycle, see protocol parameter `gNumberOfStaticSlots = 60`. Node 1 sends in slot 1 and 3 of the static segment. Slot 1 is used as keyslot and to transmit the motor speed, the speed gets transmitted periodically in every cycle (state driven transmission) and for higher reliability on both channels. Slot 3 carries the value of the LED, the transmission is also state driven but only channel A is used.

Node 2 only sends in slot 2, which is also its keyslot. This slot carries the value of the switch, that is transmitted state driven and redundantly on both channels. See Figure 4.9 for an overview of the static segment.
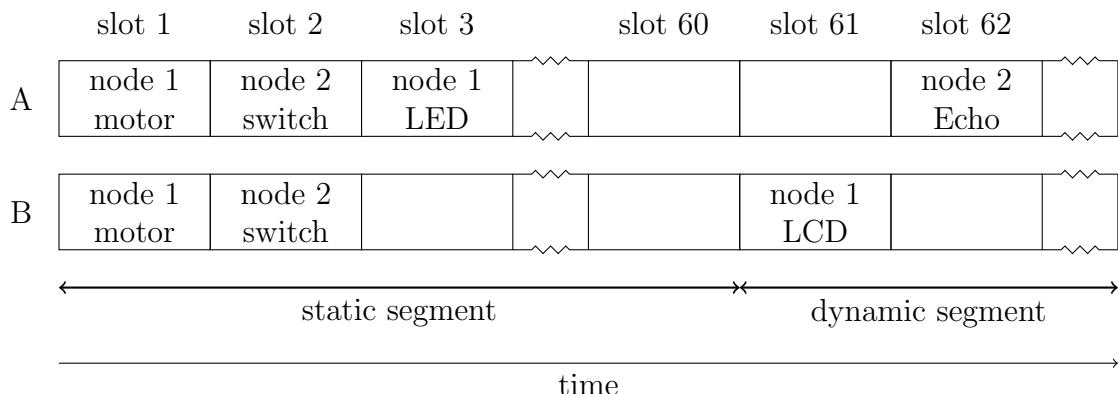
**Figure 4.9:** Assignment of static and dynamic slots for the demo cluster

4.4.4.4 Dynamic Segment Configuration

The number of slots in the dynamic segment cannot be calculated as easy as for the static segment, as dynamic slots may have varying sizes. The first dynamic slot is numbered 61, because the last static slot has the id 60 (`gNumberOfStaticSlots = 60`).

In the dynamic slot 61, node 1 sends up to 128 bytes of data on channel B. The data represents text that should be written to the display. Node 2 uses slot 62 on channel A to echo the data that have been received in slot 61 back to node 1. The transmission buffers for slot 61 and slot 62 are both set to event transmission, so that the slots are only occupied when data is available.

4.4.5 Functional Description

When the POC-state changes, the GUI gets updated to reflect the changes. The user can choose speed and direction of the motor and the status of the LED. Speed and direction are sent periodically in slot 1 by the FlexRay controller on the Linux side, the LED value in slot 3. As soon as a frame containing the values is properly received, they are propagated to the PWM subsystem or the LED respectively.

When the time has come for the embedded system to send in slot 2, it queries the pin status of the switch and puts the value into the payload. As soon as Linux system receives the switch data it updates the GUI to visualize the state of the switch.

If the user enters some text and presses enter, the text gets sent in the dynamic segment to the embedded system, which in turn puts the text on the display.

4.4.6 Experimental Results

Now, one question should come into mind – what happens when one FlexRay channel fails, let it be channel A or channel B? Looking at Figure 4.9, the theoretical results can be deduced:

- Startup and continued synchronization of the cluster will still be possible, because the startup and sync frames are transmitted redundantly on both channels

- The motor-values are transmitted on both channels so it can still be controlled

- The switch-value is also transmitted on channel A and B so the switch could serve as an emergency shutdown

- The LED can no longer be controlled when channel A is interrupted, it should not be a safety-critical device

- The display can no longer display new messages when channel A is interrupted

All these theoretical results have been validated on the real demonstration system.

5. Conclusions and Future Work

5.1   FlexRay over CAN

The first attempts to interface the FlexRay controller via CAN were quite promising, except of some shortcomings in the CAN implementation that has been used on the DSP at the RRLab for a while. Configuring the FlexRay parameters and starting the communication worked pretty well, receiving FlexRay frames on Linux still.

Unfortunately, as soon as data had to be sent in both directions, a big disadvantage of CAN came into effect: because it is a shared medium with "unfair"[1] arbitration, one direction (DSP → Linux) was preferred over the other direction, leading to lost CAN frames and thus to not-received FlexRay frames. One possibility to work around this problem had been to detect and retransmit lost CAN-frames, but unfortunately the existing CAN drivers for Linux did not provide easy means to detect frame losses and even retransmitting would not have given any guarantees for successful transmission.

Because of the very limited physical bitrate of only 1 MBit/s and the severe arbitration problems even at quite low loads, the usage CAN has been dropped very soon. To overcome the problems of accessing the FlexRay controller through a shared medium, a solution capable of full-duplex[2] transmissions had to be found.

5.2   FlexRay over UART

One of the first and most simple things that come into mind when thinking about serial communication with full-duplex support is the universal asynchronous receiver and transmitter (UART). Almost all microcontrollers feature an integrated UART compatible serial interface. Also interaction with PC hardware is easy as most hardware still has serial transceivers (in the form of RS232) and almost every other

---

[1]Unfair meaning the message with highest priority will always win.

[2]A full-duplex system can simultaneosly send and receive on a logical communication channel, this can be achieved e.g. by using two physical wires, one for transmission and one for reception or by the means of multiplexing methods such as Time-Division or Frequency-Division Multiplexing (TDM/FDM).

hardware can be equipped with USB-Serial converters that provide virtual serial ports of the Universal Serial Bus.

The FlexRay over UART concept has proven to be a reliable solution for encapsulating and transmitting FlexRay frames. It does not suffer the severe arbitration issues encountered with CAN and is also a much more flexible solution as there are many more ways to connect serial devices to a PC than when using CAN.

5.3   FlexRay over USB

Talking about USB and USB-Serial converters one may also consider using USB directly without additionally pusing all data trough a UART. The best solution would be to use a DSP with built-in USB controller as from the MFC525x-family. Those processors offer a USB high-speed controller that should exceed the datarates of FlexRay generously.

Another solution that does not involve switching processors is to connect a dedicated USB controller to the existing DSP. Thanks to the modular design of the DSP experiment board such a controller can be easily plugged onto it. During the research for this Bachelor Thesis an ad-on board has been designed with a Maxim MAX3420E USB controller. This controller connects to the DSP via the Serial Peripheral Interface Bus (SPI) at up to 26MHz bus-clock frequency. On the other side it connects to the Universal Serial Bus as a full-speed device (12 MBit/s), see [MAX3420E 07]. The PCB layout can be found in Figure 5.1 and the schematic of the circuit in Figure 5.2.
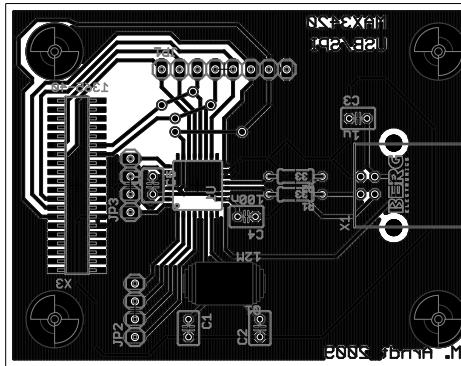

Figure 5.1: The PCB layout of the MAX3420E adapter board

This circuit will hopefully allow the connection to the Universal Serial Bus without using a USB-Serial converter, soon.

5.4   FlexRay over PCI

The RRLab has already considered designing an expansion card for the PCI Local Bus. Thanks to the now existing framework the development of a driver for the card will be much easier than before.
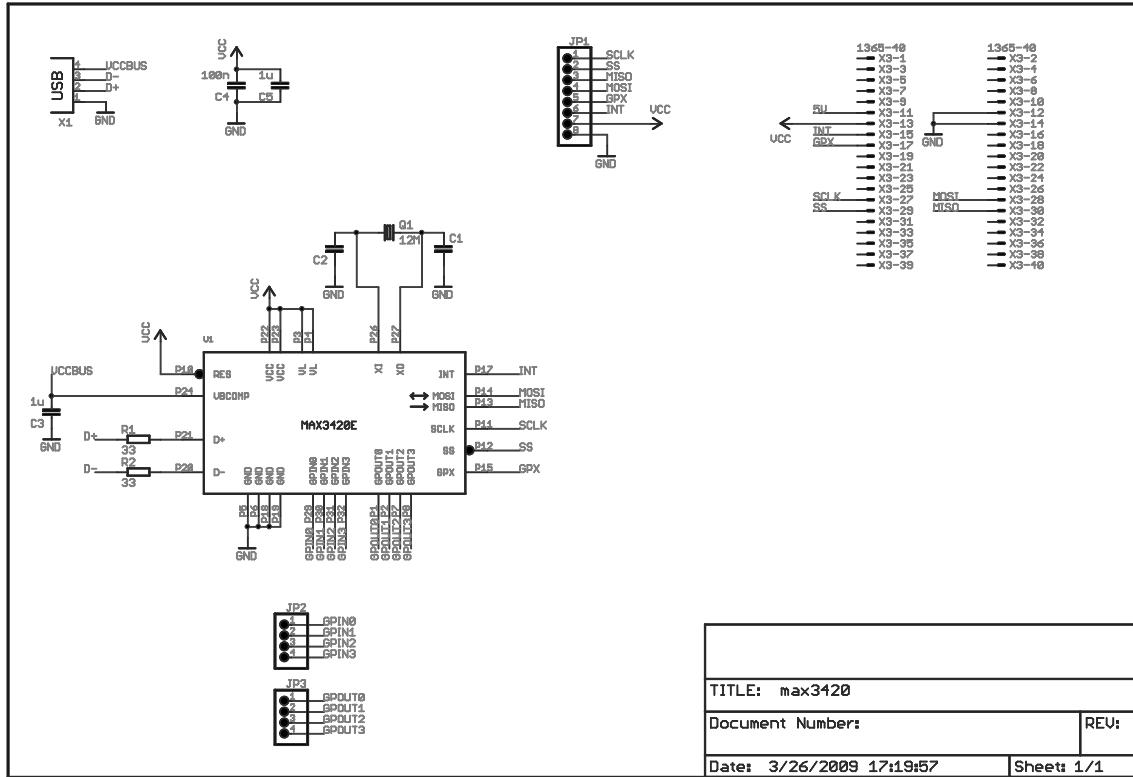
Figure 5.2: Schematic for the MAX3420E adapter board

5.5 Modular Controller Architecture (MCA)

At the Robotics Research Lab at the University of Kaiserslautern the Modular Controller Architecture (MCA) framework is widely used for a variety of different robots. This framework already is integration for the controller area network and if FlexRay is to be used at some robotic applications, also FlexRay would need to be integrated into MCA. Thanks to the now-existing Linux support, the first step for this integration has just been taken.

5.6 Inclusion in the Linux Kernel

Care has be taken to obey the Linux Kernel Coding Style [Torvalds 07] for all Kernel code. Additionally, as the FlexRay code is not very invasive (just a few constants need to be added to external modules) a inclusion in the official Linux source tree should not be too hard to achieve. Though no further actions have been taken to do this, yet.

Bibliography

[56F8357 07] freescale semiconductors. *56F8357/56F8157 Data Sheet*. 2007.

[Benvenuti 05] C. Benvenuti, *Understanding Linux Network Internals*, O'Reilly, 2005.

[Corbet 05] J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers*, O'Reilly, 2005.

[EPL 06] FlexRay Consortium. *FlexRay Communications System Protocol Electrical Physical Layer Specification Version 2.1 Rev B*. 2006.

[Freescale 06] freescale semiconductors. *FlexRay UNIFIED Driver User Guide*. 2006.

[LLCF 06] Volkswagen Group Electronic Research. *Low Level CAN Framework – Application Programmers Interface*. 2006.

[MAX3420E 07] Maxim Integrated Products. *MAX3420E USB Peripheral Controller with SPI Interface*. 2007.

[MFR4310 08] freescale semiconductors. *MFR4310 Reference Manual*. 2008.

[PS 05] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 2.1 Rev A*. 2005.

[Rausch 07] M. Rausch, *FlexRay: Grundlagen, Funktionsweise, Anwendung*, Hanser Verlag, 2007.

[Romkey 88] J. L. Romkey, *RFC 1055: Nonstandard for transmission of IP datagrams over serial lines: SLIP*, june 1988.

[Simpson 94] W. Simpson, *RFC 1661: The Point-to-Point Protocol (PPP)*, july 1994.

[Torvalds 07] L. Torvalds, *Linux Kernel Coding Style*, 2007. Can be found in the Linux source tree at `Documentation/CodingStyle`.

[Vector 08] Vector Informatik GmbH. *Hardware Interfaces for FlexRay and CAN*. 2008.

Index

A. Selected Source Files

Listing A.1: CRC Reference Implementation in C

```
#define CRC_HEADER_POLY 0xb85
#define CRC_HEADER_IV 0x1a
#define CRC_HEADER_LENGTH 11
#define CRC_HEADER_DATA_LENGTH 20


/**
 * Header-CRC reference implementation in C
 */
unsigned int crc_header_ref(unsigned int data) {
        unsigned int shiftreg = CRC_HEADER_IV;
        int i;
        int bit;

        for(i = CRC_HEADER_DATA_LENGTH-1; i >= 0; i--) {

                bit = ((shiftreg >> (CRC_HEADER_LENGTH-1)) & 0x1) ^
                  ((data >> i) & 0x1);

                shiftreg <<= 1;

                if(bit)
                        shiftreg ^= CRC_HEADER_POLY;

                shiftreg &= (1 << CRC_HEADER_LENGTH)-1;
        }

        return shiftreg;
}
```

```
30


   unsigned int create_header(unsigned int frame_id,
           unsigned char payload_length,
           unsigned char sync_bit,
35         unsigned char startup_bit) {

           /*
            * SYNC sync frame indicator
            * START startup frame indicator
40          *
            * X_10 frame-id (key slot = X)
            * X_9
            *
            * X_8
45          * X_7
            * X_6
            * X_5
            *
            * X_4
50          * X_3
            * X_2
            * X_1
            *
            * X_0
55          *
            * Y_6 Payload length (Y*2 bytes)
            * Y_5
            * Y_4
            *
60          * Y_3
            * Y_2
            * Y_1
            * Y_0
            *
65          * "Value" is: (SYNC << 19) | (START << 18) | (X << 7) | Y;
            *
            */

           return
70              (sync_bit << 19) |
                (startup_bit << 18) |
                payload_length |
                (frame_id << 7);


75 }
```

```
unsigned int crc_header(unsigned int frame_id,
        unsigned char payload_length,
        unsigned char sync_bit,
        unsigned char startup_bit) {

        return crc_header_ref(create_header(frame_id, payload_length,
                    sync_bit, startup_bit));

}
```