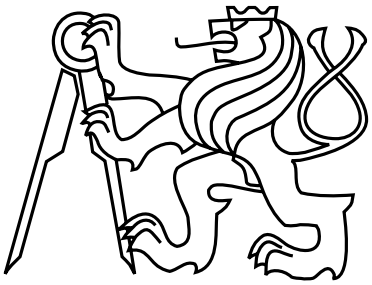
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering



Master's Thesis

Open-source and Open-hardware CAN FD Protocol Support

Bc. Martin Jeřábek

Supervisor: Ing. Jiří Novák, Ph.D.

Study Programme: Open Informatics, Master

Field of Study: Computer Engineering

January 8, 2019

Aknowledgements

I would like to thank my supervisor, Ing. Jiří Novák, Ph.D., for supervising my thesis, reviews, and lending me the USB-CAN adapter; Ing. Pavel Píša, Ph.D., for supervising the earlier parts of the project. Together with Ing. Ondrej Ille, they both have provided many valuable inputs to the projct and without their advice, many of the problems would remain unsolved. Lastly, I must not forget to thank my family and friends for their support during writing this thesis.

Declaration

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

Prague, on January 8, 2019 .....

Abstract

This thesis describes three subprojects, each dealing with support of the new CAN FD standard in open-source CAN controllers. It begins with extending the free SJA1000-compatible controller from OpenCores to tolerate FD frames on the bus and in detail describes all the problems that have surfaced during the implementation, together with their solutions. The second subproject deals with implementing Linux SocketCAN driver for a new open-source CAN FD soft core – CTU CAN FD; and finally, the design of an automated testing and verification framework, with complex result reporting and line coverage.

Keywords: CAN bus, CAN FD, Linux, SocketCAN, Xilinx Zynq, MicroZed, CTU CAN FD, SJA1000, CAN IP Soft Core, Gitlab, CI, verification, zlogan

Abstrakt

Tato práce popisuje tři subprojekty, z nichž každý se zabývá podporou nového standardu CAN FD v open-source CAN kontrolérech. Zpočátku pojednává o rozšiřování volně dostupného kontroléru z OpenCores, kompatibilního s SJA1000, o tolerování FD rámců na sběrnici a detailně popisuje všechny problémy, které vyvstaly během implementace, společně s jejich řešeními. Druhý subprojekt se zabývá implementací ovladače pro linuxový SocketCAN pro nový open-source CAN FD IP soft core – CTU CAN FD; a konečně pojednává o návrhu automatizovaného testovacího a verifikačního frameworku s komplexním hlášením výsledků a řádkovým pokrytím (line coverage).

Klíčová slova: CAN bus, CAN FD, Linux, SocketCAN, Xilinx Zynq, MicroZed, CTU CAN FD, SJA1000, CAN IP Soft Core, Gitlab, CI, verifikace, zlogan

Contents

List of Figures

List of Listings

Nomenclature

ACK Acknowledge

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

API Application Programming Interface

AXI Advanced eXtensible Interface

BRS Bit Rate Switch [23]

CAN Controller Area Network

CI Continuous Integration

CRC Cyclic Redundancy Check

CTU Czech Technical University

DCE Department of Control Engineering

DMA Direct Memory Access

DOI Data Overrun Interrupt [9]

DOR Data Overrun Flag [9]

DT Device Tree

DTB Device Tree Blob

DTO Device Tree Overlay

EDL Extended Data Length [23]

EOF End of Frame [23]

EPI Error Passive Interrupt [9]

EWLI Error Warning Limit Interrupt [9]

FD Flexible Data Rate

FEE Faculty of Electrical Engineering

FPGA Field Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

IP Intellectual Property

IRQ Interrupt Request

ISR Interrupt Service Routine

NAPI New API [15]

PCI Peripheral Component Interconnect

PSL Property Specification Language

RTL Register Transfer Logic

RXNE RX FIFO Noe Empty [9]

SKB Socket Buffer (Linux)

SOF Start of Frame [23]

SSH Secure Shell

TCL Tool Command Language

TXB TX Buffer

UIO Userspace I/O [13]

USB Universal Serial Bus

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

XML eXtensible Markup Language

XSLT eXtensible Stylesheet Language Transformations

Chapter 1

Introduction

CAN FD (Flexible Datarate), described in ISO 11898-1:2015, is an extension to the earlier CAN 2.0 specification (ISO 11898-1:2003). It introduces two new main enhancements:

- Increase of maximum payload length from 8 bytes to 64 bytes.
- Transfer the data on another, higher, bitrate (nominal bitrate is used in arbitration phase, different one may be used in data phase).

There is further information available about CAN by CiA group, both general CANbus description [3, 5] and overview of the FD extension [4, 17].

1.1 Availability of CAN FD Cores

There are many commercial CAN and even CAN FD IP cores, but not so many free ones, let alone open source. One notable example of an open-source CAN 2.0 core is a core with SJA1000-like interface [16]. No open implementation of the CAN FD protocol was found. That is why somewhere around 2015, CTU CAN FD core originated its life as Ondrej Ille's Master's Thesis.

While it is nice to have an IP which "seems to work on the table", for serious use it must be optimized, verified and documented; that demands a lot of time and resources. Fortunately, the project spiked an interest in an unnamed automotive company, and a project was born to do just that: optimize and verify the core and create Linux driver for it.

It is worth noting that costs in the IP segment are generally very high, as in addition to a team of skilled engineers, it must pay for all the commercial tools. And usually, the core has fees per instance, in addition to a fixed cost. A full-featured open source IP has the potential to reduce the costs, but of course requires more time to mature and a lot of cooperation with its potential users. Using open source tools also significantly reduces the costs for tooling.

1.2 FD-tolerant controller

Taking a fresh design and bringing it into production state takes a long time, but the industrial partner needed a passive FD-compatible solution in the early phase of the project. A controller which can be present in CAN FD networks but does not necessarily have to understand the FD traffic. In other words, an FD-tolerant controller was needed.

For this job, it was decided to use and modify the SJA1000 soft core from OpenCores [16]. There are several reasons for this choice:

- The core is free (both in price and source code).
- The core works and the author claims that it has been tested against Bosch's reference model[1].
- SocketCAN driver is available in mainline Linux kernel.
- At DCE, we already had some experience with the core, as I have been adapting it for use on Xilinx Zynq for my Canbench Bachelor's Thesis [11].

1.3 Goals of this project

This thesis describes the multiple stages of the project. The results of extending the SJA1000-compatible controller to be FD-tolerant, as well as all the encountered pitfalls, are in detail described in chapter 2 on page 5.

Chapter 3 on page 17 begins with an overview of the CTU CAN FD core, followed by an introduction to Linux SocketCAN and device driver structure. The major part of the chapter is then dedicated to the implementation and design choices for the CTU CAN FD SocketCAN driver in particular.

As stated at the very beginning of this chapter, the CTU CAN FD core needed to be well verified and the process needed to be automated and fully transparent to the developers. The steps of implementing the automated simulation, synthesis, and FPGA tests are in detail discussed in chapter 4 on page 29.

During all the development and testing, a number of tools had to be written or extended. The most notable ones are described in chapter 5 on page 41.

Finally, an assessment of the project with our current achievements, together with further goals, is present in the conclusion in chapter 6 on page 47.

1.4 Project repositories

All the source code is available on CTU FEE's GitLab server. There is the repository of CTU CAN FD Core [26], containing the RTL sources of the core itself,

[1] However, the results are not opensourced due to licensing issues. This is not sufficient for massive commercial usage but provides a good starting point.

Vivado component file for IP encapsulation, full simulation testbench, documentation, register map, SocketCAN driver and a simple userspace testing application.

The FD-tolerant SJA1000 Core repository [27] contains the RTL sources, Vivado component file and a small number of verification testbenches. And of course, the top-level project for synthesis on MicroZed board [29], which includes the abovementioned IP cores as git submodules.

Zlogan, the in-chip logical analyzer, is developed in the Github repository of its initial author Marek Peca [30]. Finally, sources of this thesis, as well as all the figures and data logs, are also available in a public git repository [28].

1.5 Used hardware and software

The FPGA project is synthesized for Xilinx Zynq on MicroZed board. As the daughter board with CAN transceivers, the canbench [11] and mzapo [19, 20] boards were used. As a reference CAN FD controller, Kvaser USB-CAN adapter has been used.

Xilinx Vivado is used for synthesis. GHDL and VUnit, together with custom extending framework, are used for simulation; gtkwave then for viewing the waveforms. Gitlab with all its infrastructure is used for the project's management and versioning, and many Docker images are used for the various Continuous Integration jobs.

Chapter 2

Making OpenCores SJA1000 FD-Tolerant

In section 1.2 on page 2, it was stated that an FD-tolerant controller was needed until the CTU CAN FD controller was ready and that SJA1000 from OpenCores was selected as a basis for this purpose. The following chapter describes the modifications done to the core in order to make it FD-tolerant, together with some interesting problems encountered on the way.

Solving the problems brought me further understanding of the CAN standard and made me understand the reasoning behind some particulars of the protocol. Parts of the standard looking as some random details turned out to be a vital part of the specification in many corner cases (which may still happen rather frequently).

The changes to the core may be summed up roughly to these points. The following sections will discuss the details of them.

- Detection of the borders of an FD frame (especially of its end) and properly ignoring it.
- Circumventing the FSM during FD ignore period (adding a condition to all transitions).
- Preventing TX during FD ignore period.
- Fix waiting for bus idle after bus-off to enable going out of reset.

2.1 Basic idea

The idea of making the controller FD-tolerant is fairly simple: If the EDL bit, indicating an FD frame, is set, ignore the rest of the frame and resume normal bus participation only after the FD frame is finished.

We thus have to detect two things:

- Beginning of the ignore period

- End of the ignore period

Finding the beginning is trivial – the received EDL bit is recessive, indicating an FD frame. The end condition is a bit more complicated. To know the length of the frame, it would be necessary to understand the CAN FD protocol and to know the data bitrate – but that way we would end up implementing the FD support, which is not trivial at all.

Fortunately, the CAN standard is designed in a way that it *is* possible to detect when the bus is idle:

- Every frame must be followed by Interframe space – at least 3 recessive bits
- Every data frame ends with an EOF field consisting of 7 recessive bits (+ 8th for ACK delimiter)
- Every error (or overload) frame ends with Error delimiter field consisting of 8 recessive bits.

Remote frames do not have to be considered since remote frames do not exist in FD format. They are, however, structurally identical to data frames. A curious reader may wonder why the amount of bits in EOF and Error Delimiter differs – surely it would seem more sensible to have them the same length, so that the minimum "space" between frames is the same, regardless of the frame type. However, upon further inspection of the data frame, one may note that there is 1 extra recessive bit before the EOF field – ACK Delimiter. This small detail remained unrecognized at first and had led to several difficulties during the testing, which were resolved after repeated examination of the CAN protocol specification.

So for the end condition, it is enough to wait for 8 consecutive *recessive* bits and then go to INTERMISSION. This condition may not occur in the middle of frame because of bit stuffing. The alternative of waiting for 11 consecutive *recessive* bits instead and then go to IDLE is not acceptable – there would arise obscure problems with Early TX, resynchronization and suspend, which are already being handled correctly in the INTERMISSION field. See section 2.4 on page 8 for details.

This again looks simple enough, but in practice, several problems arise.

2.2 The bitrate shift

In FD frames, the data part may be transmitted in a different (higher) bitrate, but the SJA1000 controller does not process this part of the frame nor does it detect the bitrate shift. If we sample the bus only in sample point of the nominal bitrate, we miss some bits. That might deceive the detection of the FD ignore period.

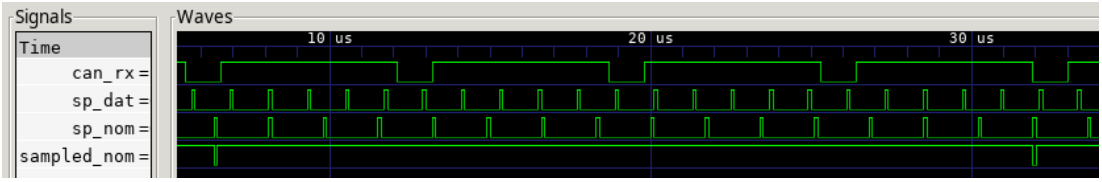


Figure 2.1: Example of failing Bus Idle detection with sampling only in nominal bitrate. `sp_nom` is high in nominal bitrate sample point, `sp_dat` in data bitrate sample point. `sampled_nom` is low when in nominal sample point and low (*dominant*) is sampled. On `can_rx` are *recessive* data with correct stuffing in data bitrate. Resolution to 100 ns time quanta, nominal bit time 16 tq (sp in $13/16 \approx 81.25\%$), data bit time 11 tq (sp in $9/11 \approx 81.2\%$).

Suppose the FD controller is transmitting a 64B frame with data payload made of 0xFF bytes. The bus is then in *recessive* state most of the time with only *dominant* stuff bits after every 5 *recessive* bits. It is not hard to imagine a scenario where the *dominant* stuff bit (in data bitrate) falls out of the sample time in nominal data bitrate for at least 11 nominal bit times (see Figure 2.1).

That way, the end of the FD frame is detected prematurely, and the rest of the frame is treated as a new frame, which will most probably be malformed and trigger an error condition. That in turn causes Error Frame to be sent on the bus and the FD frame not to be received by any node, leading to retransmission and perpetual errors (until the not-so-FD-tolerant node goes *error passive* or *bus_off*).

The solution is again simple. We do not know the data bitrate, but we do have the undivided peripheral clock available. Instead of sampling the RX signal only in sample time, it can be sampled at the undivided clock, checking for a *recessive-to-dominant* edge between this sample point and the preceding one. This way the signal levels transitions on data bitrate are captured, and the abovementioned problem is solved.

One potential drawback of this solution is making the controller more sensitive to short-term glitches during the FD ignore period, but CAN FD controllers default to single sampling during the bitrate-shifted data phase anyway. Moreover, a *dominant* glitch is much less likely than a *recessive* glitch (which is of no significance in this situation), as the *recessive* level is weaker. Better solutions require knowing (or guessing) the actual (or maximal) data bitrate.

This situation may at first glance seem identical to that of a glitch to be interpreted as SOF due to hard synchronization. However, while it is true that the hard synchronization is performed on the *recessive-to-dominant* glitch, the SOF is sampled only in sample point – and the bus is correctly detected as Idle if the *dominant* state is only a short glitch.

2.3 Handling errors

In the spirit of "perform only minimal, surgical-like changes to the core so that it works", the initial design did not stop the main CAN FSM, and only masked TX for the duration of the FD ignore period. This had some unpleasant implications:

- The core likely goes into error state during an FD frame.
- Error counters are increased.
- When the FD ignore period ends, the core may still be "transmitting" error frame.

The solution was simple – the whole FSM must be blocked until the end of FD ignore period. Due to the core design, this required changing every transition condition.

2.4 Clock skew

Another non-obvious problem occurs if the FD-tolerant receiver's and the FD frame transmitter's clock are not synchronized. This is normally countered by employing hard synchronization at SOF or resynchronization at every *recessive-to-dominant* edge. The problem is that during the data phase, the bitrate may be shifted and is not in sync with the nominal bitrate. Synchronization of the FD-tolerant receiver may theoretically be broken so that it is shifted by half the nominal bitrate in the worst case. That poses a problem if another frame immediately follows the FD frame. In an earlier version, the end condition for the FD ignore period was the detection of 11 consecutive *recessive* bits (in nominal bitrate), and only in the next bit would the controller rejoin participation on the bus. If the drift is too large, the SOF of the next frame might be received in place of the 11th bit of the quiescent state, resulting in the FD ignore period to be extended over the duration of the next frame.

This problem is not unique to this situation but might also happen in normal CAN communication. The CAN standard, however, employs 2 counter-measures, which completely prevent this problem:

- Hard synchronization on SOF
- Transmitter must send SOF only after the 3 bits of INTERMISSION, but the receiver must interpret the SOF even in the last bit of INTERMISSION.

The second step is already handled by not waiting for 11 *recessive* bits, but only for 8 and going to INTERMISSION. However, to further minimize the possible drift (and, admittedly, for historical reasons), hard synchronization is performed on every *recessive-to-dominant* edge during the FD ignore period.

2.5 Non-issues

The Suspend Transmission field of Interframe Space is not an issue, as it is relevant only if the node is currently a *Transmitter*, but the node is capable of only *receiving* FD frames[1].

Arbitration loss cannot occur when transmitting a frame simultaneously with another node sending an FD frame. Even if the transmitted frame IDs were identical for both nodes, the EDL bit indicating an FD frame is *recessive*, so the FD-tolerant (but not FD-capable) node would "win" with its non-FD frame and cause Bit-Error for the FD-capable node, as the EDL bit is already in Control Field, and not Arbitration Field.

2.6 Stress-test: degenerate case

Imagine the following stress-test:

- Only one FD-capable device and one FD-tolerant SJA1000 are on the bus.
- The FD-capable device is sending a stream of FD frames with high CAN IDs (low priority), as tightly packed as possible.
- The FD-tolerant controller is sending a stream of CAN 2.0 frames with low CAN IDs (high priority). The frames should be packed tightly, but allow for the occasional transmission of the low-priority FD frame.

The FD-capable node is spamming FD frames, which should be ignored in the FD-tolerant SJA1000. SJA1000 should also correctly step into the traffic and send its own high-priority frames.

The waveforms in figures are captured by zlogan (see section 5.1 on page 41) on FPGA board. The CAN bus is routed internally, and all nodes are on the same clocks. Nominal bitrate is 500K, data bitrate 5M. FD CAN ID 0x0FF, non-FD CAN ID 0x0F0. Details may be seen in thesis source in `figures/sja_crosstx`.

2.6.1 Variant A: FD frames are acknowledged

There is another FD-capable node on the bus, which ACKs the FD frames. Thus no errors should occur anywhere on the bus, and all the frames from both transmitters should be delivered.

2.6.2 Variant B: FD frames are not acknowledged

There are only the transmitting FD-capable node and the FD-tolerant SJA1000 on the bus. Nobody ACKs the FD frames, which results in an error frame from the transmitter. The non-FD frames should be delivered successfully.

[1] Neither will the node *transmit* an Error Frame in this situation.

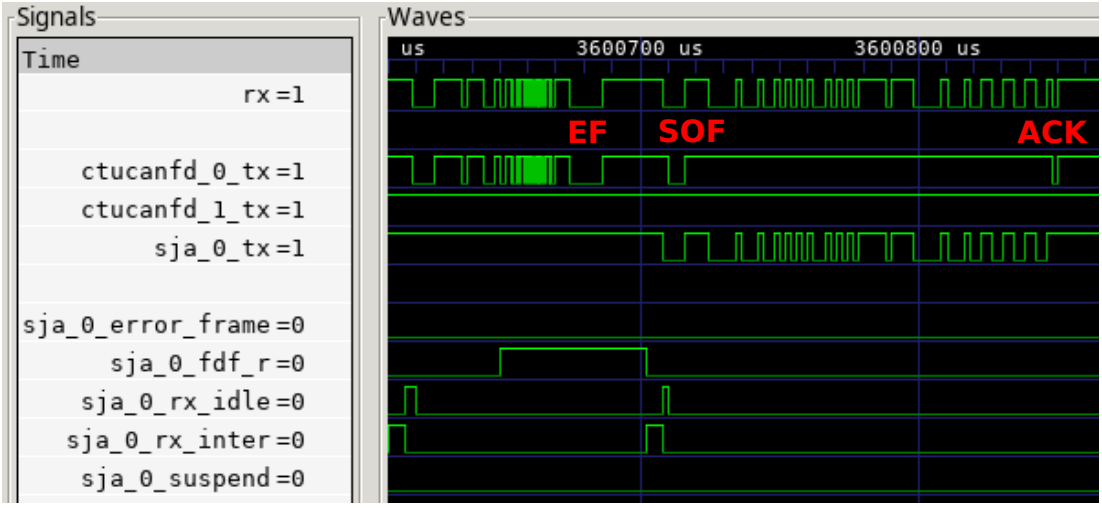


Figure 2.2: SJA cross TX test, Phase 1. Both the SJA and FD-capable (in this case CTU CAN FD) nodes are *error active*.

As the FD-capable transmitter fails to receive ACK for its frame, it sends an error frame and increments its TX Error Counter by 8 ([23], Fault Confinement rule 3). The TX Error Counter has no means to naturally decrease, and once it crosses 127, the node becomes *error passive*. From that moment, the Acknowledgment-Error does not cause the increase of TX Error Counter (exception 2 from rule 3), but the Bit Error when the SJA starts to transmit its own frame in the middle of Error Delimiter does. One of the nodes thus reaches *bus_off* state, while the other stays *error passive* and continues to retransmit its frame forever.

The test has 3 phases:

1. Both nodes are *error active*.
2. The FD-capable node becomes *error passive*. SJA1000 remains *error active*.
3. SJA1000 goes *error passive*. Both nodes are *error passive*.

2.6.2.1 Phase 1

The FD-capable controller is transmitting a frame and does not receive Ack. After the Ack Slot, it sends an Active Error Flag, followed by 8 *recessive* bits of Error Delimiter and 3 *recessive* bits of Intermission.

The FD-tolerant SJA1000 is in FD ignore mode and remains there for the duration of the whole Error Frame. After the 8 *recessive* bits of Error Delimiter, the FD ignore period ends, and the core goes to Intermission. Both nodes remain in sync.

The non-FD frames from SJA1000 are all delivered successfully.

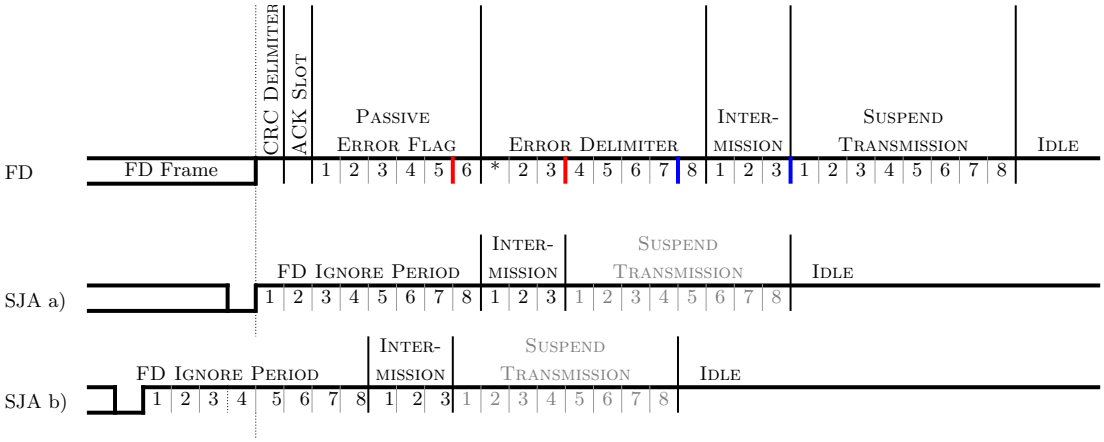


Figure 2.3: SJA cross TX test, model for non-acknowledged FD frame. The FD-capable node is *error passive*. The SJA node may be either *error active* (Phase 2), in which case it does not transmit the Suspend Transmission field and may transmit SOF immediately after its own Intermission (blue lines), or *error passive* (Phase 3; red lines). In variant (a), the last bit of CRC Sequence is dominant[a]. Variant (b) shows the other extreme when the last 4 bits are recessive[b].

[a] Or more precisely, there has been a *recessive-to-dominant* edge in the nominal bit time preceding CRC Delimiter.

[b] FD frames use fixed stuffing. The exact reasoning why this is the maximum is given in section 2.6.3 on page 14

2.6.2.2 Phase 2

When the FD controller goes *error passive*, the situation changes. There is no more the synchronizing Active Error Flag, and the controllers go off sync. As can be seen in 2.3, the FD ignore period ends in the middle of the FD-capable controller's Error Delimiter, and so does all of SJA's Intermission.

SJA then starts its own transmission, and since the other controller is not ready for it, it will not be acknowledged – the FD-capable node instead starts to transmit a Passive Error Frame, which will not be detected by the SJA. That causes the SJA to send an Active Error Frame, which again synchronizes the two nodes.

The retransmitted frame from SJA should now be delivered successfully. After an FD frame, the situation repeats, and one non-FD frame is sent non-acknowledged, leading to Error Frame, increase of TX Error Counter, and then successful retransmission.

The situation changes again when the SJA becomes *error passive*.

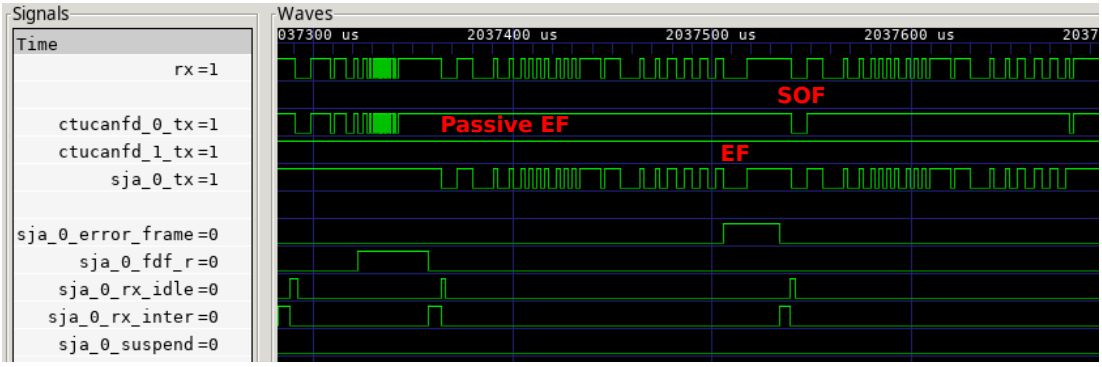


Figure 2.4: SJA cross TX test, Phase 2. The CTU CAN FD node is *error passive*, while the SJA remains *error active*.

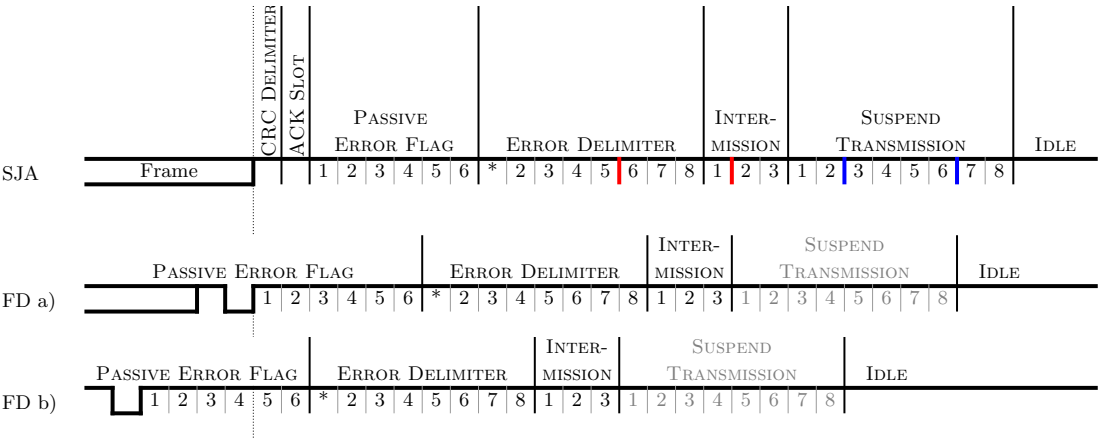


Figure 2.5: SJA cross TX test, model for non-acknowledged SJA frame. Both nodes are *error passive* (Phase 3). In variant (a), the last bit of CRC Sequence is dominant. Variant (b) shows the other extreme when the last 4 bits are recessive. Blue lines indicate the start of retransmission in conformant case, red lines represent the case when the FD node does not send Suspend Transmission.

2.6.2.3 Phase 3

Now both nodes are *error passive*, and no form of forced synchronization may occur, except for the bus being IDLE long enough for both nodes to become IDLE. However, as this is a stress test, there is no such luxury.

When an FD frame is being transmitted, the situation starts as in the previous phase. SJA starts sending its own frame in the middle of the other's node PASSIVE ERROR FRAME, resulting in SJA ACKNOWLEDGMENT-ERROR when no acknowledgment is received for the frame.

However, now the SJA sends a PASSIVE ERROR FLAG. The FD-capable core has been transmitting its own PASSIVE ERROR FLAG at the time[2]. It depends on the frame transmitted by SJA when the PASSIVE ERROR FLAG of the FD-capable node will end. Anything between two border cases may occur – either the last bit of CRC SEQUENCE field is *dominant*, or the 4 last bits of it are *recessive*[3].

In all cases, the *Transmitter* SJA node will reach IDLE in the FD-capable node's SUSPEND TRANSMISSION, as indicated by the blue lines in fig. 2.5 on the facing page. The FD-capable node now attempts to retransmit its FD frame. As the SJA is still in SUSPEND TRANSMISSION, it does not participate in arbitration. The FD frame is not acknowledged, the FD-capable node sends PASSIVE ERROR FRAME, into which the SJA starts to send its own frame and the whole cycle repeats. From this point onwards, no frame will be delivered[4].

Figure 2.6 on the next page shows a different outcome. At the time of writing, CTU CAN FD does not transmit SUSPEND TRANSMISSION after the PASSIVE ERROR FRAME, and starts the retransmission sooner[5]. As is evident from the red lines in fig. 2.5 on the facing page, two distinct situations may now occur:

- The FD-capable node starts the retransmission in the first two bits of SJA's INTERMISSION, which is interpreted as an OVERLOAD FRAME. That causes the nodes to synchronize again, and they will both participate in the next arbitration.
- The retransmission starts in SJA's ERROR DELIMITER, causing it to start a new PASSIVE ERROR FRAME. This is exactly what happened before to the FD-capable controller – their roles are now swapped. The SJA does transmit SUSPEND TRANSMISSION, and the FD node will start its next retransmission in this period. The SJA detects the FD frame, and the cycle starts anew.

[2]The error passive station waits for six consecutive bits of equal polarity, beginning at the start of the PASSIVE ERROR FLAG. The PASSIVE ERROR FLAG is complete when these 6 equal bits have been detected. ([23], 3.2.3)

[3]If 5 last bits were *recessive*, a *dominant* stuff bit would follow.

[4]As all the frames are retransmissions, there is no option from SW to delay sending the frames.

[5]Submitted as issue #225.

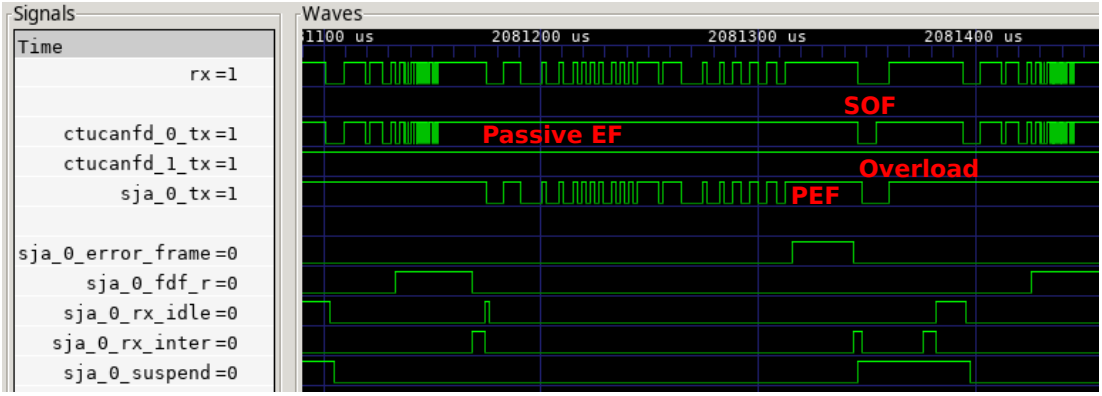


Figure 2.6: SJA cross TX test, Phase 3. Both nodes are now *error passive*. The captured waveform differs from the theoretical prediction because CTU CAN FD fails to transmit Suspend Transmission after the Passive Error Frame.

2.6.3 Possible mitigations

First of all, let it be noted that this whole situation happens only under very specific conditions – there is no node to acknowledge the FD frames, and there is no other node (except the FD-capable one) to acknowledge the non-FD frames. That is not how real-world buses are likely to look. A slightly more likely case is that all other nodes go *bus_off*. But in that case, the bus is in a bad condition anyway, so this does not add too much of a risk. Moreover, as soon as another node reintegrates, it will ACK one of the frames and the endless cycle will break.

In any case, one way to mitigate this issue is to add some wait states after ignoring an FD frame, as the initial cause of the problem is the SJA starting to transmit in the middle of Passive Error Frame. The simplest way is for the node also to transmit Suspend Transmission field of Interframe Space if the node was *Receiver* and did just ignore an FD frame. However, this would introduce a new set of problems. Calling the new field After-FD Suspend Transmission, here are the requirements for it (and its differences from Suspend Transmission at the same time):

- The time to wait has to be 8 + *however many end bits of the FD frame CRC may be recessive*. In FD frames, CRC Sequence is stuffed using fixed bit stuffing – a stuff bit is inserted at the beginning and after every 4 bits. Let us suppose that the last fixed stuff bit is *recessive*, and the remaining bits of CRC Sequence are also *recessive*. For CRC_15, CRC_17, and CRC_21, the number of these bits is 3, 1, 1, respectively. The After-FD Suspend Transmission field thus needs to be 12 bits long.
- It should not unduly delay higher-priority frames. Thus the node transmitting After-FD Suspend Transmission should participate in arbitration if another node starts transmitting during the period. It will not, however,

initiate with SOF on its own until it finishes After-FD Suspend Transmission.

This mitigation would at worst cause 12bit delay of a frame directly following an FD frame, in a situation that no other node wants to transmit at the time. At best, it can prevent a "lockup" in one engineered pathologic situation.

This mitigation is *not* implemented in the extended FD-tolerant SJA1000 controller.

2.6.4 Other findings

This contrived test helped discover a bug in the original SJA1000 core (see next section), a bug in the CTU CAN FD controller (see Figure 2.6), and an issue with the SJA's FD tolerance implementation itself[6]. At the time of writing, the CTU CAN FD issue has an open gitlab issue, and the other bugs are fixed.

2.7 Perpetual reset bug

A rather serious bug was discovered in the SJA1000 core. When SJA1000 goes to *bus_off*, it switches to Reset mode [18]. When the driver attempts to reset the controller and re-enable it by requesting to leave the Reset mode, the controller first waits for 128 occurrences of 11 consecutive *recessive* bits, as per specification [22]. However, due to internal signal duplication and unmatched conditions, the core remained forever in Reset mode. This was fixed in commit 02c7660645fb and further improved in 8e13b92c361d.

[6] As described earlier, the problem was with detecting the end of the FD ignore period for data frames, where the CRC Delimiter field was ignored. As the result, the core resumed bus participation one bit time late.

Chapter 3

SocketCAN Driver for CTU CAN FD

3.1 About CTU CAN FD

CTU CAN FD is an open source soft core written in VHDL. It originated in 2015 as a Master's project by Ondrej Ille at the Department of Measurement of FEE at CTU. After a few years, a company became interested in the core and expressed the desire to be able to synthesize several such cores in FPGA and access them from Linux via SocketCAN.

This lead to new interest in the project with much higher demand for test coverage and design reliability. It was necessary to optimize the core and redesign the register map for easy and safe use from a potentially multi-processor environment. That was kept on the core's author, with me and my supervisor as consultants.

It was also necessary to create the SocketCAN driver and integrate the core to Xilinx Zynq SoC in our MicroZed board for testing. As the core was being significantly rewritten, there arose the need to have some form of automated tests to detect errors soon. These tasks were assigned to me, and they are described in this and the following chapters.

To sum up, the following tasks were necessary to perform on the core (the italicized ones were performed by someone else and are out of the scope of this thesis):

- *Optimize the core (resource usage, maximum operating frequency)*
- Integrate the core to MicroZed testing platform
- Create SocketCAN driver
- Create an automated verification framework and environment to automatically run the tests

3.2 About SocketCAN

SocketCAN is a standard common interface for CAN devices in the Linux kernel. As the name suggests, the bus is accessed via sockets, similarly to common network devices. The reasoning behind this is in depth described in [14]. In short, it offers a natural way to implement and work with higher layer protocols over CAN, in the same way as, e.g., UDP/IP over Ethernet.

3.2.1 Device probe

Before going into detail about the structure of a CAN bus device driver, let's reiterate how the kernel gets to know about the device at all. Some buses, like PCI or PCIe, support device enumeration. That is, when the system boots, it discovers all the devices on the bus and reads their configuration. The kernel identifies the device via its vendor ID and device ID, and if there is a driver registered for this identifier combination, its probe method is invoked to populate the driver's instance for the given hardware. A similar situation goes with USB, only it allows for device hot-plug.

The situation is different for peripherals which are directly embedded in the SoC and connected to an internal system bus (AXI, APB, Avalon, and others). These buses do not support enumeration, and thus the kernel has to learn about the devices from elsewhere. This is exactly what the Device Tree was made for.

3.2.2 Device tree

An entry in device tree states that a device exists in the system, how it is reachable (on which bus it resides) and its configuration – registers address, interrupts and so on. An example of such a device tree is given in listing 1 on the next page.

3.2.3 Driver structure

The driver can be divided into two parts – platform-dependent device discovery and set up, and platform-independent CAN network device implementation.

3.2.3.1 Platform device driver

In the case of Zynq, the core is connected via the AXI system bus, which does not have enumeration support, and the device must be specified in Device Tree. This kind of devices is called *platform device* in the kernel and is handled by a *platform device driver*[1].

A platform device driver provides the following things:

- A *probe* function

[1] Other buses have their own specific driver interface to set up the device.

```
/ {
    /* ... */
    amba: amba {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";

        CTU_CAN_FD_0: CTU_CAN_FD@43c30000 {
            compatible = "ctu,ctucanfd";
            interrupt-parent = <&intc>;
            interrupts = <0 30 4>;
            clocks = <&clkc 15>;
            reg = <0x43c30000 0x10000>;
        };
    };
};
```

Listing 1: An excerpt from device tree declaring an AMBA bus, to which one CTU CAN FD core is attached.

- A *remove* function
- A table of *compatible* devices that the driver can handle

The *probe* function is called exactly once when the device appears (or the driver is loaded, whichever happens later). If there are more devices handled by the same driver, the *probe* function is called for each one of them. Its role is to allocate and initialize resources required for handling the device, as well as set up low-level functions for the platform-independent layer, e.g., *read_reg* and *write_reg*. After that, the driver registers the device to a higher layer, in our case as a *network device*.

The *remove* function is called when the device disappears, or the driver is about to be unloaded. It serves to free the resources allocated in *probe* and to unregister the device from higher layers.

Finally, the table of *compatible* devices states which devices the driver can handle. The Device Tree entry `compatible` is matched against the tables of all *platform drivers*.

3.2.3.2 Network device driver

Each network device must support at least these operations:

- Bring the device up: `ndo_open`
- Bring the device down: `ndo_close`

```
/* Match table for OF platform binding */
static const struct of_device_id ctucan_of_match[] = {
    { .compatible = "ctu,canfd-2", },
    { .compatible = "ctu,ctucanfd", },
    { /* end of list */ },
};
MODULE_DEVICE_TABLE(of, ctucan_of_match);

static int ctucan_probe(struct platform_device *pdev);
static int ctucan_remove(struct platform_device *pdev);

static struct platform_driver ctucanfd_driver = {
    .probe  = ctucan_probe,
    .remove = ctucan_remove,
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = ctucan_of_match,
    },
};
module_platform_driver(ctucanfd_driver);
```

Listing 2: Platform device driver declaration. Only prototypes of the *probe* and *remove* functions are included.

- Submit TX frames to the device: `ndo_start_xmit`
- Signal TX completion and errors to the network subsystem: ISR
- Submit RX frames to the network subsystem: ISR and NAPI

There are two possible event sources: the device and the network subsystem. Device events are usually signaled via an interrupt, handled in an Interrupt Service Routine (ISR). Handlers for the events originating in the network subsystem are then specified in `struct net_device_ops`.

When the device is brought up, e.g., by calling `ip link set can0 up`, the driver's function `ndo_open` is called. It should validate the interface configuration and configure and enable the device. The analogous opposite is `ndo_close`, called when the device is being brought down, be it explicitly or implicitly.

When the system should transmit a frame, it does so by calling `ndo_start_xmit`, which enqueues the frame into the device. If the device HW queue (FIFO, mailboxes or whatever the implementation is) becomes full, the `ndo_start_xmit` implementation informs the network subsystem that it should stop the TX queue (via `netif_stop_queue`). It is then re-enabled later in ISR when the device has some space available again and is able to enqueue another frame.

All the device events are handled in ISR, namely:

1. **TX completion**. When the device successfully finishes transmitting a frame, the frame is echoed locally. On error, an informative error frame[2] is sent to the network subsystem instead. In both cases, the software TX queue is resumed so that more frames may be sent.
2. **Error condition**. If something goes wrong (e.g., the device goes bus-off or RX overrun happens), error counters are updated, and informative error frames are enqueued to SW RX queue.
3. **RX buffer not empty**. In this case, read the RX frames and enqueue them to SW RX queue. Usually NAPI is used as a middle layer (see section 3.2.4).

3.2.4 NAPI

The frequency of incoming frames can be high and the overhead to invoke the interrupt service routine for each frame can cause significant system load. There are multiple mechanisms in the Linux kernel to deal with this situation. They evolved over the years of Linux kernel development and enhancements. For network devices, the current standard is NAPI – *the New API*. It is similar to classical top-half/bottom-half interrupt handling in that it only acknowledges the interrupt in the ISR and signals that the rest of the processing should be done in softirq context. On top of that, it offers the possibility to *poll* for new frames for a while. This has a potential to avoid the costly round of enabling interrupts, handling an incoming IRQ in ISR, re-enabling the softirq and switching context back to softirq.

More detailed documentation of NAPI may be found on the pages of Linux Foundation [15].

3.3 Integrating the core to Xilinx Zynq

The core interfaces a simple subset of the Avalon [10] bus as it was originally used on Alterra FPGA chips, yet Xilinx natively interfaces with AXI [2]. The most obvious solution would be to use an Avalon/AXI bridge or implement some simple conversion entity. However, the core's interface is half-duplex with no handshake signaling, whereas AXI is full duplex with two-way signaling. Moreover, even AXI-Lite slave interface is quite resource-intensive, and the flexibility and speed of AXI are not required for a CAN core.

Thus a much simpler bus was chosen – APB (Advanced Peripheral Bus) [1]. APB-AXI bridge is directly available in Xilinx Vivado, and the interface adaptor entity is just a few simple combinatorial assignments.

[2]Not to be mistaken with CAN Error Frame. This is a `can_frame` with `CAN_ERR_FLAG` set and some error info in its `data` field.

Finally, to be able to include the core in a block diagram as a custom IP, the core, together with the APB interface, has been packaged as a Vivado component.

3.4 CTU CAN FD Driver design

The general structure of a CAN device driver has already been examined in section 3.2.3 on page 18. The next paragraphs provide a more detailed description of the CTU CAN FD core driver in particular.

3.4.1 Low-level driver

The core is not intended to be used solely with SocketCAN, and thus it is desirable to have an OS-independent low-level driver. This low-level driver can then be used in implementations of OS driver or directly either on bare metal or in a user-space application. Another advantage is that if the hardware slightly changes, only the low-level driver needs to be modified.

The code[3] is in part automatically generated and in part written manually by the core author, with contributions of the thesis' author. The low-level driver supports operations such as: set bit timing, set controller mode, enable/disable, read RX frame, write TX frame, and so on.

3.4.2 Configuring bit timing

On CAN, each bit is divided into four segments: SYNC, PROP, PHASE1, and PHASE2. Their duration is expressed in multiples of a Time Quantum (details in [22], chapter 8). When configuring bitrate, the durations of all the segments (and time quantum) must be computed from the bitrate and Sample Point. This is performed independently for both the Nominal bitrate and Data bitrate for CAN FD.

SocketCAN is fairly flexible and offers either highly customized configuration by setting all the segment durations manually, or a convenient configuration by setting just the bitrate and sample point (and even that is chosen automatically per Bosch recommendation if not specified). However, each CAN controller may have different base clock frequency and different width of segment duration registers. The algorithm thus needs the minimum and maximum values for the durations (and clock prescaler) and tries to optimize the numbers to fit both the constraints and the requested parameters.

A curious reader will notice that the durations of the segments PROP_SEG and PHASE_SEG1 are not determined separately but rather combined and then, by default, the resulting TSEG1 is evenly divided between PROP_SEG and PHASE_SEG1. In practice, this has virtually no consequences as the

[3] Available in `/driver` in CTU CAN FD repository [26]

```
struct can_bittiming_const {
    char name[16];      /* Name of the CAN controller hardware */
    __u32 tseg1_min;    /* Time segement 1 = prop_seg + phase_seg1 */
    __u32 tseg1_max;
    __u32 tseg2_min;    /* Time segement 2 = phase_seg2 */
    __u32 tseg2_max;
    __u32 sjw_max;      /* Synchronisation jump width */
    __u32 brp_min;      /* Bit-rate prescaler */
    __u32 brp_max;
    __u32 brp_inc;
};
```

Listing 3: Fields of `can_bittiming_const`

sample point is between PHASE_SEG1 and PHASE_SEG2. In CTU CAN FD, however, the duration registers `PROP` and `PH1` have different widths (6 and 7 bits, respectively), so the auto-computed values might overflow the shorter register and must thus be redistributed among the two[4].

3.4.3 Handling RX

Frame reception is handled in NAPI queue, which is enabled from ISR when the RXNE (RX FIFO Not Empty) bit is set. Frames are read one by one until either no frame is left in the RX FIFO or the maximum work quota has been reached for the NAPI poll run (see section 3.2.4 on page 21). Each frame is then passed to the network interface RX queue.

An incoming frame may be either a CAN 2.0 frame or a CAN FD frame. The way to distinguish between these two in the kernel is to allocate either `struct can_frame` or `struct canfd_frame`, the two having different sizes. In the controller, the information about the frame type is stored in the first word of RX FIFO.

This brings us a chicken-egg problem: we want to allocate the `skb` for the frame, and only if it succeeds, fetch the frame from FIFO; otherwise keep it there for later. But to be able to allocate the correct `skb`, we have to fetch the first work of FIFO. There are several possible solutions:

1. Read the word, then allocate. If it fails, discard the rest of the frame. When the system is low on memory, the situation is bad anyway.
2. Always allocate `skb` big enough for an FD frame beforehand. Then tweak the `skb` internals to look like it has been allocated for the smaller CAN 2.0 frame.

[4] As is done in the low-level driver functions `ctu_can_fd_set_nom_bittiming` and `ctu_can_fd_set_data_bittiming`.

3. Add option to peek into the FIFO instead of consuming the word.
4. If the allocation fails, store the read word into driver's data. On the next try, use the stored word instead of reading it again.

Option 1 is simple enough, but not very satisfying if we could do better. Option 2 is not acceptable, as it would require modifying the private state of an integral kernel structure. The slightly higher memory consumption is just a virtual cherry on top of the "cake". Option 3 requires non-trivial HW changes and is not ideal from the HW point of view.

Option 4 seems like a good compromise, with its disadvantage being that a partial frame may stay in the FIFO for a prolonged time. Nonetheless, there may be just one owner of the RX FIFO, and thus no one else should see the partial frame (disregarding some exotic debugging scenarios). Basides, the driver resets the core on its initialization, so the partial frame cannot be "adopted" either. In the end, option 4 was selected[5].

3.4.3.1 Timestamping RX frames

The CTU CAN FD core reports the exact timestamp when the frame has been received. The timestamp is by default captured at the sample point of the last bit of EOF but is configurable to be captured at the SOF bit. The timestamp source is external to the core and may be up to 64 bits wide. At the time of writing, passing the timestamp from kernel to userspace is not yet implemented, but is planned in the future.

3.4.4 Handling TX

The CTU CAN FD core has 4 independent TX buffers, each with its own state and priority. When the core wants to transmit, a TX buffer in Ready state with the highest priority is selected.

The priorities are 3bit numbers in register TX_PRIORITY (nibble-aligned). This should be flexible enough for most use cases. SocketCAN, however, supports only one FIFO queue for outgoing frames[6]. The buffer priorities may be used to simulate the FIFO behavior by assigning each buffer a distinct priority and *rotating* the priorities after a frame transmission is completed.

In addition to priority rotation, the SW must maintain head and tail pointers into the FIFO formed by the TX buffers to be able to determine which buffer should be used for next frame (`txb_head`) and which should be the first completed one (`txb_tail`). The actual buffer indices are (obviously) modulo 4 (number of TX buffers), but the pointers must be at least one bit wider to be able to

[5] At the time of writing this thesis, option 1 is still being used and the modification is queued in gitlab issue #222

[6] Strictly speaking, multiple CAN TX queues are supported since v4.19 [21] but no mainline driver is using them yet.

TXB#	0	1	2	3
Seq	A	B	C	
Prio	7	6	5	4
		T		H

(a) 3 frames are queued.

TXB#	0	1	2	3
Seq		B	C	
Prio	4	7	6	5
		T		H

(b) Frame A was successfully sent and the priorities were rotated.

TXB#	0	1	2	3	0’
Seq	E	B	C	D	
Prio	4	7	6	5	
		T			H

(c) 2 new frames (D, E) were enqueued. Notice that the priorities did not have to be adjusted. `txb_head` now contains the value 5 which indicates TXB#0, but allows us to detect that all buffers are full.

Figure 3.1: TXB priority rotation example. Empty Seq means the buffer is empty. Higher priority number means higher priority. H and T mark `txb_head` and `txb_tail`, respectively [9].

distinguish between FIFO full and FIFO empty – in this situation, $txb\_head \equiv txb\_tail \pmod 4$. An example of how the FIFO is maintained, together with priority rotation, is depicted in fig. 3.1.

3.4.4.1 Timestamping TX frames

When submitting a frame to a TX buffer, one may specify the timestamp at which the frame should be transmitted. The frame transmission may start later, but not sooner. Note that the timestamp does not participate in buffer prioritization – that is decided solely by the mechanism described above.

Support for time-based packet transmission was recently merged to Linux v4.19 [6], but it remains yet to be researched whether this functionality will be practical for CAN.

Also similarly to retrieving the timestamp of RX frames, the core supports retrieving the timestamp of TX frames – that is the time when the frame was successfully delivered. The particulars are very similar to timestamping RX frames and are described in section 3.4.3.1 on the facing page.

3.4.5 Handling RX buffer overrun

When a received frame does no more fit into the hardware RX FIFO in its entirety, RX FIFO overrun flag (STATUS[DOR]) is set and Data Overrun Interrupt (DOI) is triggered. When servicing the interrupt, care must be taken first to clear the DOR flag (via COMMAND[CDO]) and after that clear the DOI interrupt flag. Otherwise, the interrupt would be immediately[7] rearmed.

[7] Or rather in the next clock cycle

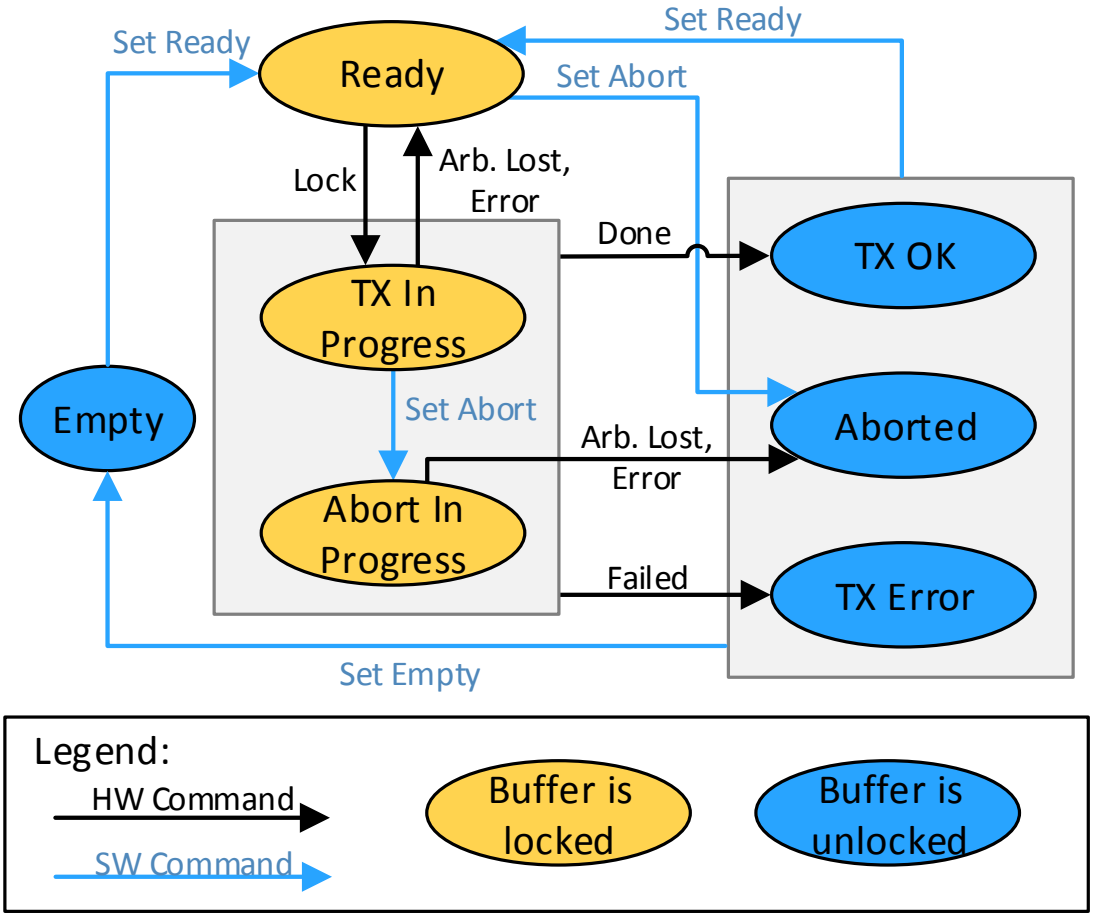


Figure 3.2: TX Buffer states with possible transitions [9].

Note: During development, it was discussed whether the internal HW pipelining cannot disrupt this clear sequence and whether an additional dummy cycle is necessary between clearing the flag and the interrupt. On the Avalon interface, it indeed proved to be the case, but APB being safe because it uses 2-cycle transactions. Essentially, the DOR flag would be cleared, but DOI register's Preset input would still be high the cycle when the DOI clear request would also be applied (by setting the register's Reset input high). As Set had higher priority than Reset, the DOI flag would not be reset. This has been already fixed by swapping the Set/Reset priority (see issue #187).

3.4.6 Reporting Error Passive and Bus Off conditions

It may be desirable to report when the node reaches *Error Passive*, *Error Warning*, and *Bus Off* conditions. The driver is notified about error state change by an interrupt (EPI, EWLI), and then proceeds to determine the core's error state by reading its error counters.

There is, however, a slight race condition here – there is a delay between the time when the state transition occurs (and the interrupt is triggered) and when the error counters are read. When EPI is received, the node may be either *Error Passive* or *Bus Off*. If the node goes *Bus Off*, it obviously remains in the state until it is reset. Otherwise, the node is *or was Error Passive*. However, it may happen that the read state is *Error Warning* or even *Error Active*. It may be unclear whether and what exactly to report in that case, but I personally entertain the idea that the past error condition should still be reported. Similarly, when EWLI is received but the state is later detected to be *Error Passive*, *Error Passive* should be reported.

Chapter 4

Testing

4.1 Simulation framework

The core already included a set of simulation testbenches together with a set of simple TCL scripts for Modelsim to run them. Together with saved waveform layouts, this was perfect for debugging or semi-automated testing, but as more and more changes were performed in the core, it became apparent that a fully automated testing workflow would benefit us immensely.

Although Modelsim's debugging capabilities are indisputable superior, it was decided to use an open-source simulator – GHDL [7], for the automated testing. The reasons behind this decision include:

- GHDL's simulation speed is better than that of free Modelsim edition
- Possible license issues with Modelsim usage
- Support for line coverage
- Support for functional coverage via PSL

Since the core repository is hosted on the university Gitlab [26], it is only natural to leverage Gitlab's Continuous Integration.

4.1.1 CTU CAN FD Testcases overview

The core includes an extensive set of verification tests. The tests may be divided into several groups, based mainly on the test interface:

- Unit tests
- Feature tests
- Sanity test
- Reference test

The unit tests generally verify the individual small entities, whereas the feature tests check the behavior of the core as a whole in various scenarios. The sanity test instantiates multiple instances of the core, connected in a specified topology with simulated transceiver and bus delay, and has them communicate with each other. The reference test replays the bus traffic logged from a reference CAN FD controller and checks that the core receives all the frames correctly. The tests are in detail described in the core documentation [9] in chapter 5.

4.1.2 Installing GHDL

GHDL compiles the simulation sources to a machine executable binary. The code generation may be performed by 3 different backends: GCC, LLVM or mcode. While the gcc-flavoured ghdl is the most complicated to build, it is the only one that is able to provide a basic code coverage functionality. Although the code coverage works almost "by accident", for basic line coverage, this is fairly accurate. While it is undoubtedly true that professional IPs get tested while observing expression coverage, toggle coverage and many more, the free tools in the area of digital design do not possess these advanced capabilities.

Gcc-flavoured GHDL has requirements on the version of gcc which is building it and on the version used to compile the testbenches. Moreover, if it is desired to use the code coverage feature, the version of the GCOV library used during compilation and when running the tests must be the same. The manual process of setting up all the dependencies is rather complicated and fragile. Fortunately, this complexity and inter-dependency are eliminated by containerizing it all in Docker.

Docker is a tool for lightweight "virtualization", implemented via operating system containers. Most instruction files to build a typical docker image share common patterns: based on a specific distribution image (e.g., debian stable), install some packages, build an application, copy over some user-supplied files and package it all in a ready-to-use image. This image is then later used as a base to run a *container* executing the desired command.

Moreover, GHDL has set up their own CI/CD and the most recent docker images, with prebuilt ghdl, are published on Docker Hub. Using Docker is thus very convenient in this instance. Docker images are also the most common way to use custom build environment in Gitlab CI.

4.1.3 Extending the simulation framework

In addition to the testbenches and simulator, it is necessary to:

- build the sources
- run the testbenches, with configurable parameters
- collect test results and show a summary

This is a non-trivial amount of work to do; however, the requirements are not very special. A suitable simulation framework was searched for to use as a basis, and eventually it was decided to use VUnit.

4.1.3.1 VUnit

VUnit is a complex HDL simulation framework. It is written in Python, with the HDL support libraries written in VHDL. It also supports a lot of simulators, including GHDL and Modelsim. So as a bonus, it is still possible to open the simulation in Modelsim and use the prepared weveform layouts to diagnose problems quickly, with modelsim-specific code kept to the minimum.

It implements both the running part (compile, run, collect results) and the VHDL part (error reporting, logging library, setting up test timeout, and many more).

4.1.3.2 The framework

The test framework is located in the CTU CAN FD repository in `/test/testfw`, with its entry point `/test/run.py`. It is a command line Python application, whose primary purpose is to run the desired tests. All the compiling and running functionality comes directly from VUnit. What the framework adds is mainly unified configuration and the ability to set up waveforms in GUI mode automatically.

The framework supports several types of tests, grouped by common interface: unit tests, feature tests, sanity test, and reference test.

The configuration is hierarchical, with values from the default section being inherited by the particular tests if not overridden. That enables both fine-grained configurability and concise test definition if no special treatment is needed.

As there is a lot of unit tests and feature tests, it is quite possible that when adding a new one, the implementor forgets to add it to the configuration. Fortunately, the framework detects such unconfigured tests and prints a clearly visible warning.

Most of the configuration options are passed to the VHDL test case as generics of the top-level entity. GHDL does not support passing composite types (arrays, records, vectors) in this way, some values have to be serialized on the Python side and again deserialized on VHDL side.

More information about the particular tests may be found in CTU CAN FD documentation [9].

4.1.3.3 Running tests

Running the tests requires Python 3 and dependencies specified in `/test/testfw/requirements.txt`, as well as at least one HDL simulator. All simulators sup-

```
unit:
    default:
        log_level: warning
        iterations: 50
        timeout: 100 ms
        error_tolerance: 0
        # randomize: false
        # seed: 0 # use to reconstruct results from randomized runs
    tests:
        bit_stuffing:
            iterations: 10
            wave: unit/Bit_Stuffing/bsdt_unit.tcl
        apb:
            iterations: 1
        bus_sync:
            wave: unit/Bus_Sampling/bsnc_unit.tcl
feature:
    default: {...}
    tests:
        abort_transmittion:
        arbitration:
        fault_confinement:
```

Listing 4: Example of test suite configuration

ported by VUnit are supported, but it has been only tested (and optimized for) Modelsim and GHDL.

In case no simulator is set up on a computer, one may use the prepared Docker image with all the dependencies and GHDL. A convenience script to run the tests in the docker image is included as `/run-docker-test` in the CTU CAN FD repository.

4.1.3.4 Automatic waveform layout in GUI mode

When debugging with a test case, one generally wants to observe a given set of internal signals and preserve this view between runs. Each test case has its own specific set of useful signals, and this layout may be saved in a file and attached to the test case in configuration, as may be seen in listing 4.

At the time of writing, the layout files are only supported when simulating in Modelsim, because they originated as Modelsim-specific TCL files. There is, however, a development version[1], which may either specify native `gtkwave` layout

[1] That might be found in branch `gtkw-gui` in the CTU CAN FD repository

```
# Print help for the test subcommand
$ ./run.py test --help

# Run all tests from tests_fast.yml
$ ./run.py test tests_fast.yml

# Print help for VUnit; you may specify VUnit options after --
$ ./run.py test tests_fast.yml -- --help

# List all tests
$ ./run.py test tests_fast.yml -- --list

# Read configuration from tests_fast.yml, but run just one test
$ ./run.py test tests_fast.yml lib.tb_feature.retr_limit

# Run all feature tests (configured in tests_fast.yml)
$ ./run.py test tests_fast.yml 'lib.tb_feature.*'

# Force using a specific simluator. Read more in VUnit documentation.
$ VUNIT_SIMULATOR=modelsim ./run.py test tests_fast.yml

# Run a test in GUI mode
$ ./run.py test tests_fast.yml lib.tb_feature.retr_limit -- -g
```

Listing 5: Examples of running simulation tests

files, or attempt to create them from the Modelsim TCL file. To be reasonably usable, it requires patching VUnit and still has some minor problems. Therefore, it has not been merged yet.

4.1.3.5 Test results

The test results are printed to the terminal, with a summary at the end, but also gathered in a JUnit-compatible XML file, which may then be rendered in an HTML browser using a shipped XSLT stylesheet.

4.1.3.6 Line coverage

As mentioned above in section 4.1.2 on page 30, GHDL with GCC backend supports generating line and function[2] coverage using GCOV. This is achieved by specifying GHDL flags `-fprofile-arcs -ftest-coverage` during the analyze

[2]Not to be confused with functional coverage, which is also supported, but via different means.

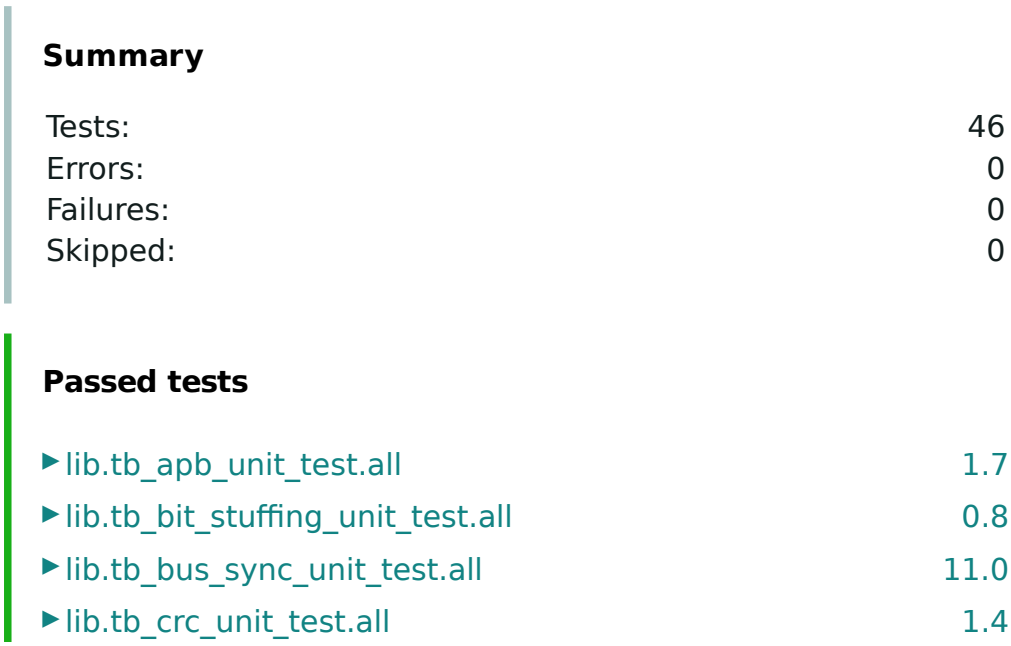


Figure 4.1: Example test results from the testing framework. After clicking on a test case name, its output is shown.

```
$ lcov --capture --directory build --output-file code_coverage.info
$ genhtml code_coverage.info --output-directory code_html
```

Listing 6: Generating HTML report from test code coverage

phase and `-Wl,-lgcov -Wl,--coverage -Wl,-no-pie` during the elaboration phase.

During the compilation, one part of profile data is generated by the compiler (the `.gcno` files) and when the tests are run, the second part of the profile data is generated in the form of `.gcda` files. All the profile data are then processed by commands `lcov` and `genhtml`, as seen in listing 6.

The necessary compile options for GHDL are enabled in the testing framework in `/test/testfw/test_common.py` and the generating of HTML coverage report is handled in the supplied Makefile[3]. A link to the report for the most recent core version is available on the project's gitlab page [26].

4.2 Automated builds

For automated testing on actual hardware, it is desirable to be able to build the FPGA bitstream automatically from the freshest sources. In addition, as the

[3] `/test/Makefile`

synthesis takes a rather long time (around 20 minutes), it would be wasteful to rebuild the image after every change. Before delving into the particulars of the system setup, you might want to revise the project structure and repositories in 1.4. The following requirements were gathered:

- Before the build, submodules should be automatically updated to the newest version.
- The build should run only if the sources changed (including submodules).
- The build should run only once a day, ideally at night when the build server is free.

This is a tricky combination for Gitlab's CI, as it may trigger the pipeline on push (changes) or periodically (at night), but not on their combination. Another difficulty comes with updating the submodules – but that actually presents a nice solution.

Every night:

- The master branch is merged into an autobuild branch.
- Submodules are updated in the autobuild branch.
- The autobuild branch is pushed to the repository.

Finally, on a push to any autobuild branch, the build job (and subsequent tests) is triggered. The build may also be triggered manually by creating a pipeline for the autobuild branch[4]. For the future, there are planned multiple autobuild branches, each for different stability phase of the submodules (i.e., stable, bleeding edge, etc.). The detailed CI configuration may be found in the top-level repository in `/.gitlab-ci.yml`.

4.2.1 Pushing to repository from CI job

Unfortunately, gitlab does not yet provide read-write deploy tokens for CI jobs, and thus an alternative approach must be used. A dedicated SSH key pair is generated for the CI. The public key is registered into Gitlab (under some user) and the private key is made available to the job runner.

The private key may be passed to the runner via Gitlab CI secret variables, but since we have our own runner anyway, it is possible to keep the private key only on the build server, in a docker volume, and specify that the volume should be mounted in the project's runner.

[4]Or the master branch, in which case the autoupdates and pushes are performed.

```
[[runners]]
  ...
  executor = "docker"
  [runners.docker]
    volumes = ["/cache", "/opt/xilinx:/opt/xilinx:ro",
               "depkey_cantop:/depkey:ro"]
```

Listing 7: Specifying a volume in `config.toml` for gitlab-runner

4.2.2 Making Vivado available in the build image

A nice thing about docker images is that they are self-contained and may run everywhere. It is thus convenient to upload the built image to Docker Hub and let whichever runner download it. That is not possible with the synthesis tools, as they are not free and must not be distributed.

One solution is to create a *private* image on Docker Hub or use our own Docker Registry[5]. Unfortunately, the synthesis tools also take a lot of disk space.

Taking inspiration from the solution with deploy keys (see the previous section), the Vivado synthesis tools are made available to the runner as a bind mount. That way, we are limited on our one runner, but we do not have to be concerned about gigabytes of licensed software traveling around the network. The configuration may be seen in listing 7.

4.3 Automated FPGA tests

While most hardware problems should be caught in simulation, it is still desirable to perform at least some simple tests on real hardware, for several reasons:

- There might be problems introduced by synthesis.
- Simulation is slow and can only perform that many tests.
- It is an opportunity also to test the driver.
- It is an opportunity also to test interoperability with different controllers.

After every automated build (described in section 4.2 on page 34):

1. The FPGA bitstream, CTU CAN FD driver, and the whole test suite is uploaded to the MicroZed testing board.
2. The new bitstream is loaded into the FPGA (see section 4.3.1 on the facing page).
3. The new kernel driver is loaded.

[5] Self-hosted backend of Docker Hub

4. The test suite is run (see section 4.3.2 on the next page).
5. The results are collected and downloaded to the job runner.

4.3.1 Updating FPGA bitstream

The FPGA bitstream may be updated in 3 distinct ways:

1. via U-Boot on boot (requires restart)
2. via `/dev/xdevcfg` (requires Xilinx-flavored kernel)
3. via kernel FPGA Manager interface

For the purposes of automated testing on HW, variant 3 was chosen for this project as it is the most elegant one. The FPGA Manager subsystem finally got into the mainline kernel in v4.4 with significant upgrades later on. Unlike the previous vendor-specific implementations, the FPGA Manager approach addresses the fact that when the bitstream is updated, the hardware *changes*. Old devices disappear, and new ones appear. That means the old devices must be deinitialized before the update and new devices must be probed afterward. Even if the uploaded bitstream is identical to its predecessor, the hardware is reset, and the drivers needs to reinitialize it.

The question is how to tell the kernel *which* devices will appear or disappear. Traditionally, there is one monolithic Device Tree Blob (DTB) which is loaded at boot time, and it contains definitions of both Hard Core peripherals and Soft Core peripherals implemented in FPGA. Alternatively, the Device Tree may be divided into a base image and one or more Device Tree Overlays. This elegantly solves the problem, as the FPGA image is paired with a DTO, describing the peripherals implemented by the bitstream. When a bitstream is loaded, the DTO is applied afterward and the kernel probes the newly appeared device. Similarly, when the bitstream is about to be removed (or replaced), the DTO is unloaded and the disappearing devices are correctly deinitialized (together with all bookkeeping data structures in the kernel).

4.3.1.1 Using FPGA Manager from userspace

The FPGA Manager interface is directly available only from the kernel itself. For some reason, the mainline kernel does not provide any method to access this interface from user space. Fortunately, there exists an external module `dtbocfg`[6]. It creates a directory hierarchy under ConfigFS, which may be accessed using standard shell commands.

Full explanation and description of how to pack the bitstream, configure the kernel, set up the device tree, and finally apply the overlay are given in [24, 12]. A very brief summary is also given in listings 8, 9, and 10.

[6] Available at https://github.com/ikwzm/dtbocfg

```
# Wrap the BIT bitstream as BIN
# system.bif contains just "all: {system.bit}"
bootgen -image system.bif -w -process_bitstream bin

# Copy the bitstream into /lib/firmware
cp system.bit.bin /lib/firmware/system.bit.bin

# Compile the base Device Tree
dtc -O dtb -b 0 -@ -o base.dtb base.dts

# Compile the Device Tree Overlay
dtc -O dtb -b 0 -@ -o overlay.dtbo overlay.dts
```

Listing 8: Preparing the FPGA bitstream for loading via FPGA Manager interface and compiling the Device Trees.

4.3.2 The test suite

The tests are written in Python using the Pytest framework and pycan for CAN bus communication. The sources are located in the top-level repository in `/ci/cantest`. This is an ongoing effort as more tests may always be added. The progress may be observed in the top-level project's issue #1.

Tests:

- Check that all the required CAN interfaces are present (sanity test).
- Run CTU CAN FD's *regtest* tool, which checks the IP integration (word, halfword, and byte reads and writes work correctly, identification register contains expected value).
- Series of communication tests, with one transmitter and one or more receivers.

Unstable tests in development:

- RX FIFO Overrun test.
- Series of communication tests with multiple transmitters.

For the communication tests, both CTU CAN FD and SJA1000 FD-tolerant controllers are tested. The tests have variants based on which of the controllers is the transmitter (CTU CAN FD or SJA1000) and on the communication mode (only CAN 2.0 frames, ISO FD frames, non-ISO FD frames).

To reliably test the handling of RX FIFO Overrun, the particular test requires a special debug `ioctl` in the device driver to disable or re-enable handling of RX frames in the driver.

```
# Load the module
modprobe dtbocfg

# Create new overlay under ConfigFS. The name is not important.
cd /sys/kernel/config/device-tree/overlays
mkdir fpga-overlay
cd fpga-overlay

# Load the DTBO file contents
cat overlay.dtbo >dtbo

# Activate the DT overlay
echo 1 >status
```

Listing 9: Loading the FPGA bitstream via FPGA Manager interface.

For the communication tests with multiple transmitters, care must be taken when randomly generating the CAN IDs. If multiple nodes try to transmit a frame with identical CAN ID (but different payload) at the same time, the mismatch in Data Phase will be interpreted as a Bit-Error instead of Arbitration-Loss. This fact was overlooked at first, and the core was thought to contain a bug – which is not the case in this instance; the behavior is consistent with the CAN specification.

During the tests, the kernel message log (`dmesg`) is monitored, and if a message with severity Warning or above appears during a test, the test fails. Also, the network interface error counters are checked and must be zero; otherwise the test fails.

```
/dts-v1/;
/plugin/;
/ {
    fragment@0 {
        target-path = "/fpga-full";

        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;

            firmware-name = "system.bit.bin";
        };
    };

    fragment@1 {
        target-path = "/amba";
        __overlay__ {
            /* new devices */
        };
    };
};
```

Listing 10: Example of device tree overlay source.

Chapter 5

Extra work

5.1 Extending zlogan

Zlogan[1] is an in-chip logical analyzer originally developed by Marek Peca. It captures its input signals and streams all changes via AXI-Stream to a FIFO, from where it is read by DMA to an application and saved into a file.

The analyzer has been an essential component during testing both the modified SJA1000 and CTU CAN FD in hardware.

5.1.1 Modifications

The base operation remains unchanged, but the following changes had to be made:

- Encapsulate zlogan as Vivado IP.
- Create register map and APB interface.
- Fix the IP to be able to handle a (theoretically) arbitrary number of input signals.
- Extend the streaming application to
  - Allow for reading more data than the DMA engine's maximum transfer size.
  - Allow to abort the stream while saving already transferred data.
  - Detect and report errors.
  - Use a portable way to allocate buffer for DMA (via `udmabuf` driver).

The updated version may be found in the original github repository in branch `zlogan-component`.

At the time of writing, there are several (unstable) changes in a development branch, including:

[1] Available at https://github.com/eltvor/zlogan

- Separating parts of the IP into different entities
- Creating unit tests for the core entities
- Correctly resetting registers on zlogan enable and flushing all buffers at disable.

5.1.2 Usage

The *zlogan* IP requires additional components (on Zynq):

- AXI DMA
- AXI4-Stream Data FIFO
- AXI-APB Bridge

An example top-level design (except the actual analyzer inputs) is depicted in fig. 5.1 on the facing page. At the moment, the whole design (from `la_inp` to AXIS FIFO) is synchronous to one clock. If sampling on higher frequency is desired, the AXIS FIFO may be made asynchronous, and the whole core may operate on a higher frequency than PS's AXI.

On the SW side, at the moment everything runs in userspace. In the `rxla` program, modify the config file `hw_config.h` and set zlogan registers address, AXIS DMA registers address, and width of the DMA transfer length register[2]. Now it is ready to be compiled. In addition, the *udmabuf* module has to be compiled and loaded, to provide a memory region safe for DMA transfers to userspace.

The `rxla` application transfers data from the analyzer and saves them into a file, until either the given maximum number of bytes is transferred or the program is interrupted (via Ctrl+C).

5.2 Rewrite of CAN crossbar IP

During the testing, it was necessary to test the various CAN controllers connected to either an external CAN bus or more simply to an in-chip internal bus. To avoid having to rebuild the whole FPGA image for every possible configuration, the `can_crossbar` IP exists.

The basic idea is that there are N in-chip *controllers* and M external *buses*. Each of these may be connected to exactly one of L *lines*, which facilitate the N:M mapping of controllers to buses. Multiple buses or controllers may be attached to each line, the driving signals merged by logical AND.

[2]It is recommended to set it as high as possible/reasonable in the DMA IP configuration. While the stream is fetched using multiple DMA transfers if needed, there might be some glitches on transfer boundaries.

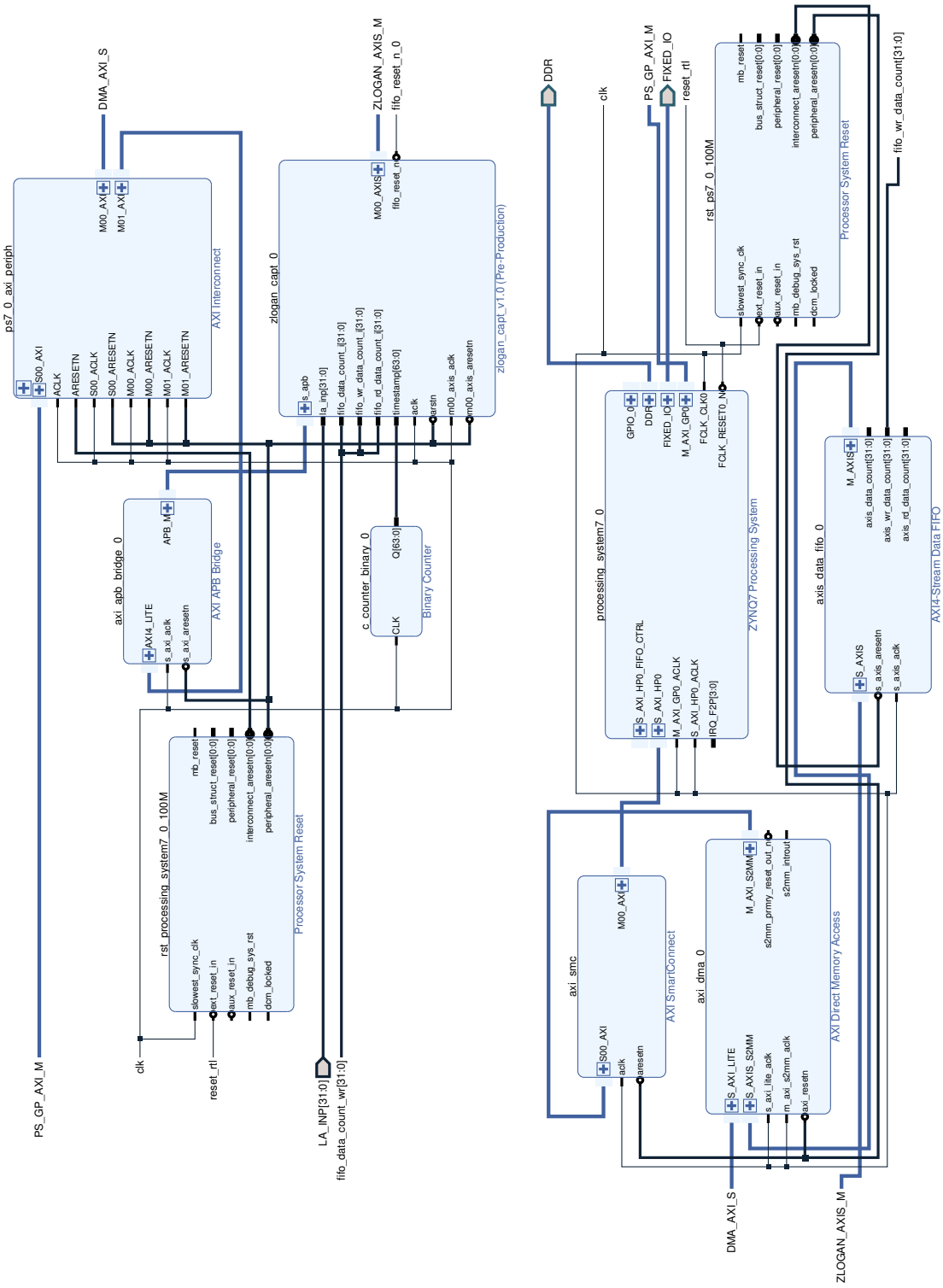
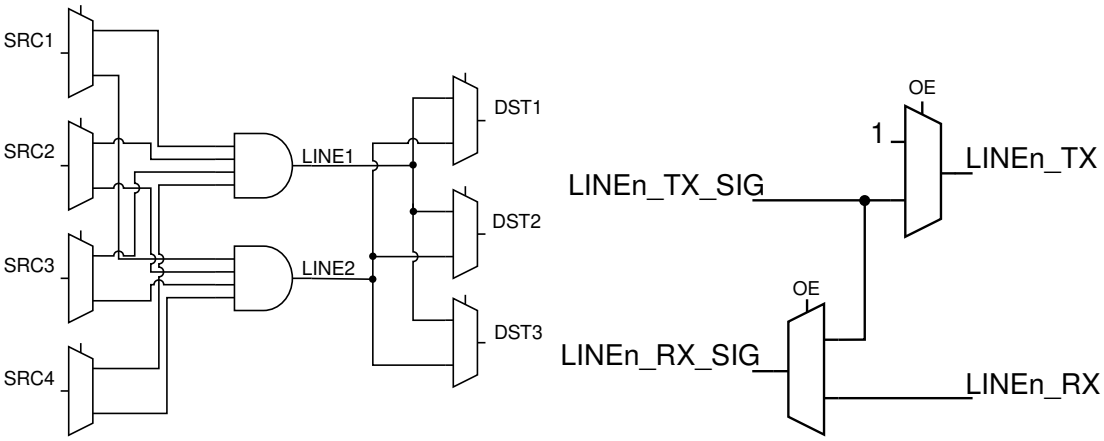


Figure 5.1: Example of top-level design with zlogan in Vivado



(a) One half of *can_crossbar*, instantiated with 4 *sources*, 3 *sinks* and 2 *lines*

(b) Output Enable function

Figure 5.2: Structure of *can_crossbar*

While the `can_crossbar` IP was already available from my Bachelor's thesis [11], it was not tested and even I, the author of the IP, doubted about its functionality (and didn't like its naive structure). Since it is a very simple IP, it was decided to rewrite it, this time properly and with testbenches[3].

The structure is simple. For each line, merge all unmasked sources as its input, and connect each sink to its selected line. RX and TX directions are handled independently, so there are independent RX and TX channels of a line. The masking is done by demultiplexers with non-selected outputs driven to logical 1 (*recessive* state). An example of this structure may be seen in fig. 5.2a.

Lastly, each *line* may either be connected to external buses (which are assigned to the line) or looped into itself (line's TX channel is connected to its RX channel), in which case the TX channel of the assigned external buses is driven high.

The new IP interfaces the system via APB instead of the much more heavyweight AXI-Lite. The number of *controllers*, *buses*, and *lines* is configurable for the underlying component, but it is fixed for the whole IP as to simplify the register map design.

5.2.1 Driver

There is no Linux driver for the IP. Its register map may be accessed either directly via `/dev/kmem` (and command-line utility `devmem`) or via Userspace I/O driver (UIO). The UIO driver takes information about the register map from device tree and offers applications to `mmap(2)` the registers into the application's address space. [13, 8]

[3]The old IP may be found in the toplevel design repository at `/system/ip/can_crossbar_1.0`, the new one at `/system/ip/can_crossbar_2.0`

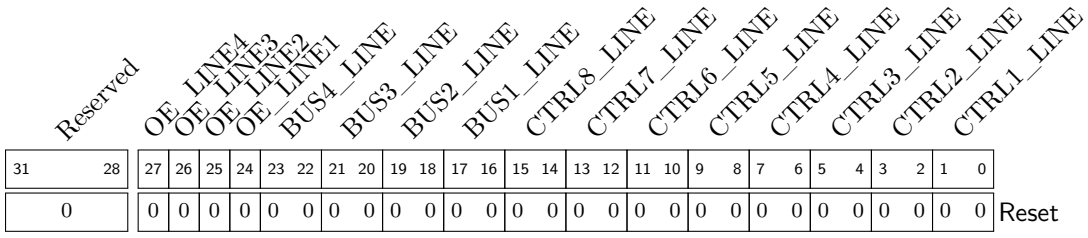
5.2.2 Register Overview

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

5.2.2.1 CAN Configuration Register

Address offset: 0x000
Reset value: 0x00000000

Register 5.1: CCR (0x000)



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

BUSn_LINE Specifies which line is the external bus connected to:
00: The external bus is connected to LINE1
01: The external bus is connected to LINE2
10: The external bus is connected to LINE3
11: The external bus is connected to LINE4

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

Chapter 6

Conclusion

The goals of this project were to extend the OpenCores SJA1000 Core to be FD-tolerant, implement a SocketCAN driver for CTU CAN FD, implement automated testing and verification for CTU CAN FD, and to document the whole solution.

The FD-tolerant SJA1000 core was extensively tested in various worst-case scenarios – 0xFF frames with BRS bit set, or stress test with only one FD-capable node. The various shortcomings encountered during implementation were fixed, or mitigations have been proposed (2.6.3). The core behavior corresponds with the constructed theoretical model. Unfortunately, the industrial partner did not provide us with additional feedback.

The SocketCAN driver for CTU CAN FD has been implemented and is available in the CTU CAN FD GitLab repository [26]. It has been tested in the FPGA tests, which are discussed in section 4.3 on page 36, and also manually with other CAN controllers, such as the in-chip FD-tolerant SJA1000 or external Kvaser USB CAN adapter.

At the time of writing, the core contains a few known corner-case bugs, which are mapped out in the project's issue tracker, together with additional planned features. It is our intention with the core's author, Ondrej Ille, to further work on the core and eventually pass the certification according to ISO 16845-1 and bring the core to production quality. Despite the core's immaturity, some additional parties have already shown interest in the CTU CAN FD core [25]. The driver is also planned to be extended to support retrieval of HW timestamp of received and transmitted frames, or support placement of the device on different buses, such as PCI. After some adjustments and cleanups, the driver will be submitted for inclusion to mainline Linux kernel.

The automated testing, both in the form of design simulation and behavioral tests on real hardware, have been implemented and are in detail described in chapter 4 on page 29. The high-level overview of the framework is described in this thesis and will be used in the project documentation. The commands usage is available via the tool's help command, and the developer documentation is

included in the sources in the form of comments.

All of the relevant parts are open-source, available in git repositories mentioned in the introduction. While not yet being ready to be used in critical systems, both the cores are more than usable for hobbyists or non-safety-critical systems. The progress made in the previous year is significant, and there are high hopes of the development continuing.

Bibliography

[1] ARM. *AMBA APB Protocol Specification v2.0*. https://developer.arm.com/docs/ihi0024/latest/amba-apb-protocol-specification-v20 [Online; accessed 2019-01-06].

[2] ARM. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite*. https://developer.arm.com/docs/ihi0022/latest/amba-axi-and-ace-protocol-specification-axi3-axi4-axi5-ace-and-ace5 [Online; accessed 2019-01-06].

[3] CAN in Automation. *CAN data link layers*. http://www.can-cia.org/can-knowledge/can/can-data-link-layers/ [Online; accessed 2019-01-06].

[4] CAN in Automation. *CAN FD – The basic idea*. https://www.can-cia.org/can-knowledge/can/can-fd/ [Online; accessed 2019-01-05].

[5] CAN in Automation. *CAN physical layer*. http://www.can-cia.org/can-knowledge/can/systemdesign-can-physicallayer/ [Online; accessed 2019-01-06].

[6] J. Corbet. *Time-based packet transmission*. LWN, Mar 2018. https://lwn.net/Articles/748879/ [Online; accessed 2019-01-06].

[7] T. Gingold. *GHDL: VHDL 20018/93/98 simulator*. https://github.com/ghdl/ghdl [Online; accessed 2019-01-06].

[8] J. Gray. *How to Design and Access a Memory-Mapped Device in Programmable Logic from Linaro Ubuntu Linux on Xilinx Zynq on the ZedBoard, Without Writing a Device Driver — Part Two*, May 2013. http://fpga.org/2013/05/28/how-to-design-and-access-a-memory-mapped-device-part-two/ [Online; accessed 2019-01-06].

[9] O. Ille and M. Jeřábek. *CTU CAN FD IP Core Datasheet*. http://illeondr.pages.fel.cvut.cz/CAN_FD_IP_Core/Progdokum.pdf [Online; accessed 2019-01-06].

[10] Intel Corporation. *Avalon® Interface Specifications*. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf [Online; accessed 2019-01-06].

[11] M. Jeřábek. FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation. Bachelor's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2016. https://rtime.felk.cvut.cz/can/F3-BP-2016-Jerabek-Martin-Jerabek-thesis-2016.pdf [Online; accessed 2019-01-06].

[12] I. Kawazome. *Device Tree Blob Overlay Configuration File System*. https://github.com/ikwzm/dtbocfg [Online; accessed 2019-01-06].

[13] H.-J. Koch. *The Userspace I/O HOWTO*, Dec 2006. https://www.kernel.org/doc/html/v4.18/driver-api/uio-howto.html [Online; accessed 2019-01-06].

[14] Linux Foundation. *Linux SocketCAN*. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/can.txt [Online; accessed 2019-01-06].

[15] Linux Foundation. *NAPI*. https://wiki.linuxfoundation.org/networking/napi [Online; accessed 2018-12-10].

[16] I. Mohor. *SJA1000-compatible CAN Protocol Controller IP Core*. Opencores.org. https://opencores.org/projects/can [Online; accessed 2019-01-06].

[17] A. Mutter. *CAN FD and the CRC issue*. CAN in Automation Newsletter, Mar 2015. https://can-newsletter.org/uploads/media/raw/604de101b0ecaed387518831d32b044e.pdf [Online; accessed 2019-01-05].

[18] Philips Semiconductors. *SJA1000 Stand-alone CAN controller Data Sheet*, Jan. 2000. https://www.nxp.com/documents/data_sheet/SJA1000.pdf [Online; accessed 2019-01-06].

[19] P. Píša. *MicroZed APO Board*. https://cw.fel.cvut.cz/b172/courses/b35apo/documentation/mz_apo/start (in Czech) [Online; accessed 2019-01-07].

[20] P. Píša and P. Porazil. *MicroZed APO Board Schematics and PCB Layout.* http://cmp.felk.cvut.cz/~pisa/apo/mz_apo/ [Online; accessed 2019-01-07].

[21] Robert Bosch GmbH. *can: enable multi-queue for SocketCAN devices.* https://lore.kernel.org/patchwork/patch/913526/ [Online; accessed 2018-12-10].

[22] Robert Bosch GmbH. *CAN Specification, Version 2.0.* http://esd.cs.ucr.edu/webres/can20.pdf [Online; accessed 2018-12-10].

[23] Robert Bosch GmbH. *CAN with Flexible Data-Rate Specification, Version 1.0*, Apr 2012. https://can-newsletter.org/uploads/media/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf [Online; accessed 2019-01-06].

[24] Xilinx Inc. *Solution Zynq PL Programming With FPGA Manager.* https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841645/Solution+Zynq+PL+Programming+With+FPGA+Manager [Online; accessed 2019-01-06].

[25] H. Zeltwanger. *CAN FD core as an open source project.* CAN in Automation Newsletter. https://can-newsletter.org/uploads/media/raw/58bba6274be7d31a50a69cf92211b0f5.pdf [Online; accessed 2019-01-05].

[26] Repository for CTU CAN FD core. https://gitlab.fel.cvut.cz/illeondr/CAN_FD_IP_Core.

[27] Repository for FD-tolerant SJA1000 core. https://gitlab.fel.cvut.cz/canbus/zynq/sja1000-fdtol.

[28] Repository for this thesis. https://gitlab.fel.cvut.cz/jerabma7/canfd-thesis.

[29] Repository with toplevel design for Zynq, with CTU CAN FD and FD-tolerant SJA1000 cores. https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top.

[30] Repository for zlogan core, the in-chip logical analyzer. https://github.com/eltvor/zlogan.

Appendix A

Contents of attached CD

```
/
+-- canfd-thesis
    +-- DtsLexer
    +-- figures
    +-- utils
        +-- find-abbreviations
        +-- mkdtb
        \-- package-build-for-extmodules
    +-- contents.tex
    +-- Jerabek-thesis-2019-canfd.tex
    +-- Makefile
    +-- README.md
    +-- reference.bib
    +-- 10-intro.tex
    +-- 20-sja1000.tex
    +-- 30-ctucanfd-driver.tex
    +-- 40-testing.tex
    +-- 50-zlogan.tex
    +-- 60-conclusion.tex
    \-- 97-appendices.tex
\-- zynq-can-sja1000-top
    +-- ci
        +-- cantest
        +-- fetch_ctucanfd_driver.py
        +-- requirements.txt
        +-- test_can.yml
        \-- upload_and_run.sh
    +-- modules
        +-- CTU_CAN_FD
            +-- doc
            +-- driver
            +-- scripts
```

```
        +-- spec
        +-- src
        +-- synthesis
        +-- test
            +-- feature
            +-- lib
            +-- others
            +-- reference
            +-- sanity
            +-- testfw
            +-- unit
            +-- Makefile
            +-- run.py
            +-- tests_debug.yml
            +-- tests_fast.yml
            +-- tests_nightly.yml
            +-- tests_reference.yml
            \-- xunit.xsl
        +-- tools
        +-- LICENSE
        +-- README.md
        +-- run-docker-mkdoc
        \-- run-docker-test
    +-- device-tree-xlnx
    +-- sja1000
        +-- drivers
        +-- hdl
        +-- testbench
        +-- tests
        \-- component.xml
    +-- udmabuf
    \-- zlogan
        +-- hw
            \-- zlogan_capt
                +-- hdl
                \-- component.xml
        +-- sw
            +-- rxla
            +-- zlo.h
            \-- zlo2vcd.c
        +-- udmabuf
        +-- LICENSE
        \-- README.md
+-- scripts
+-- system
    +-- ip
```

```
            +-- can_crossbar_1.0
            +-- can_merge
            +-- CTU_CAN_FD_1.0 -> ../../modules/CTU_CAN_FD/src
            +-- sja1000_1.0 -> ../../modules/sja1000
            \-- zlogan_capt_1.0 -> ../../modules/zlogan/hw/zlogan_capt
        +-- script
            +-- build.tcl
            +-- dist.tcl
            +-- gendevtree.tcl
            +-- mkfsbl.tcl
            +-- recreate.tcl
            \-- top.tcl
        +-- src
            +-- constrs
            \-- hdl
                \-- top.vhd
        \-- system.bif
    +-- AUTOBUILDS.md
    +-- Makefile
    \-- README.txt

383 directories, 909 files
```

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jeřábek Martin** Personal ID number: **420017**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Open Informatics**

Branch of study: **Computer Engineering**

II. Master's thesis details

Master's thesis title in English:

Open-source and Open-hardware CAN FD Protocol Support

Master's thesis title in Czech:

Open-source a Open-hardware podpora pro CAN FD

Guidelines:

Design and implement an open source/hardware support for CAN FD in Linux:
a) Modify SJA1000 soft IP core from OpenCores to tolerate CAN FD traffic on the bus.
b) Implement and test Linux SocketCAN driver for CTU CAN FD soft IP core controller.
c) Implement and document automated testing via continuous integration for CTU CAN FD.
d) Document all components.

Bibliography / sources:

Etschberger, K.: Controller Area Network, IXXAT Press 2001, ISBN 3-00-007376-0
ISO 11898-1:2015, Road vehicles -- Controller area network (CAN) -- Part 1: Data link layer and physical signalling
ISO 16845-1:2016, Road vehicles -- Controller area network (CAN) Conformance Test Plan -- Part 1: Data link layer and physical signalling
Linux Kernel and Driver Development Training, https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Name and workplace of master's thesis supervisor:

doc. Ing. Jiří Novák, Ph.D., K 13138 - katedra měření

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

Date of master's thesis assignment: **27.09.2018** Deadline for master's thesis submission: **08.01.2019**

Assignment valid until: **19.02.2020**

doc. Ing. Jiří Novák, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

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

. | Date of assignment receipt

Student's signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC