CAN Flexible Data-rate IP core

Datasheet

Ille Ondrej
Czech technical university in Prague
Faculty of Electrical Engineering
Department of measurement

September 2, 2016

Contents

1. Introduction

Testing of automotive buses is a challenging task these days. Due to this reason device described in [5] is being developed. Since CAN 2.0 specification was recently extended with Flexible Data-rate, CAN bus now offers more bandwidth than ever before. To integrate this functionality into the device in [5] Flexible Data Rate functionality needs to be covered. The predecessor of this component is CAN IP function developed in [6]. Due to an absence of proper RTL tests in [6] new implementation is considered (instead of extending the previous one).

1.1 Functionality requirements

To perform tests on CAN bus, several functions of the CAN controller are required. These requirements can be implemented in hardware (controller itself) or software (firmware or driver). A software implementation is usually more complicated than hardware one (due to strict timing requirements). Due to this reason all test requirements are implemented in hardware as part of CAN FD IP core in VHDL language. These requirements are:

1. Implement CAN FD protocol according to [1]. Additionally implement ISO FD protocol. ISO or Non-ISO protocol can be switched by a configuration bit in a memory map of IP Core.
2. Provide "Logical Link Control" extension of MAC layer. It enables to send and receive frames by manipulation with buffers (instead of direct manipulation with CAN state machine). This approach simplifies firmware implementation.
3. Be able to capture external time stamp when received frame is stored in a buffer, as well as start transmitting a frame when specific value of external time stamp is reached.
4. Record various events on the bus with the time stamp.
5. After a frame is inserted with specified transmission time, provide information when it was truly sent.
6. Be able, to trigger event recording (point 4) by a specific event on the bus.
7. Be able to manipulate Fault confinement state and counters (see [1] for details).

The IP Core is supposed to be synthesized in ALTERA Cyclone IV FPGA. Several parts of the design are implemented in a way that Synthesis tools are able automatically to infer native hardware blocks (e.g. RAM blocks). However, special signal attributes are not used (like memory type). This approach makes the implementation resource independent. It is possible to use the IP core also in ASIC but it requires modifications which are common for FPGA to ASIC transition.

1.2 Commercial research

A short commercial research was made before implementation to avoid overcharging of implementation. Following products were found:

- CAN FD IP Core by CAST inc., an unknown prize for RTL source code. Apparently, academic institutes are not even worthy to answer an email for CAST. It can be only speculated how much does the IP Core costs. This core (with custom wrapper) is also offered by other companies, e.g Xilinx.
- IFI CAN FD IP Core Ingenieurbüro Für Ic-Technologie, prize: 12375 euros, encrypted netlist only

Both products are too expensive (for the purpose of [5]) and don't fulfill requirements 4-7 in 1.1. On the other hand, both IP cores are very well verified and ready for ASIC manufacture or SoC integration. Since the main application in [5] is FPGA based, this advantage is less important. Due to stated reasons own implementation is selected.

1.3 Development tools

To develop this IP Core following tools are used:

- ModelSim ALTERA Edition 6.5b for VHDL implementation and RTL test-benches.
- Quartus II 9.1sp Web Edition for synthesis (target device from ALTERA EP4CE55F23C8N) and Timing analysis (Time Quest timing analyzer).
- Code Composer Studio v 5.0 for accessing IP function as a periphery of Texas Instruments MCU (via EMIF interface) from C code.
- HALCoGen v04.04 for processor configuration (refer to [5] where whole test system is described).
- CANoe program (with CAN/USB converter) for post-synthesis verification by communication with reference controller.
- LyX v.2.1.2 to write documentation.
- Microsoft Vision 2016 to create block diagrams.

2. System architecture

IP Core implementation consists of several modules. Each module has a unique function. The number of dependencies between modules is minimal, thus keeping modularity design rule. The whole system is implemented as synchronous design with asynchronous reset (assumed to be connected to an input pin of FPGA or ASIC). CAN FD IP core is memory mapped periphery. In order to be compatible with FlexRay IP core implemented in [18], Avalon bus interface is used. This interface is described in [17].

Whole IP core is implemented with VHDL 2008 version of the language (it also compiles with 2002 version). Following VHDL packages from IEEE library are used by the IP core (synthesizable part):

- std_logic_1164
- numeric_std
- std_logic_unsigned

Additional package **CANconstants** is implemented. This package contains type and constants definitions used in the IP Core implementation.

Table 1 lists the ports of the IP core. Table 2 lists generic configuration parameters of the IP Core which can be modified before synthesis.

Name	Direction	Width	Type	Description
clk_sys	in	1	Clock	System clock. IP core has only this clock domain
res_n	in	1	Reset	Asynchronous reset, active low
data_in	in	32	Avalon bus	Input data of the memory bus
data_out	out	32	Avalon bus	Output data of the memory bus
address	in	24	Avalon bus	Address of the memory bus
scs	in	1	Avalon bus	Chip select signal , active high
srd	in	1	Avalon bus	Read, active high during memory bus read
swr	in	1	Avalon bus	Write, active high during memory bus write
int	out	1	Interrupt	Interrupt output from IP Core
CAN_tx	out	1	CAN bus	TX signal to the CAN bus
CAN_rx	in	1	CAN bus	RX signal from the CAN bus
time_quanta_clk	out	1	Clock	Output clock with period of time quanta
timestamp	in	64	Time	Timestamp for frame transmition

Table 1: IP Core ports

Name	Type	Default	Description
use_logger	boolean	true	Event logger is/is not synthesized. Refer to 2.5.10 Event logger (optional)
rx_buffer_size	natural	128	Size of receive buffer in 32 bit words. Refer to 2.5.5 Receive buffer (RX)
useFDSize	boolean	true	TXT buffer size: 8 bytes (Normal)/64 bytes(FD). Refer to 2.5.6 Transmit buffer - Time (TXT)
use_sync	boolean	true	Use/Don't use synchronisation chain for received data. Can be set to false when synthesis tools automatically insert the synchronisation chain. Otherwise set to true.
ID	natural	1	ID of the controller. Unique identifier in case of more instances of the IP core. Address(19:16) to which the IP core is mapped.
logger_size	natural	8	Size of the event logger memory in number of events. Refer to 2.5.10 Event logger (optional)

Table 2: IP Core generic settings

2.1 Addressing

As mentioned earlier CAN FD IP core is accessed over Avalon bus (described in [17]). Avalon bus is SoC parallel bus by Altera corporation. It consists of separate write/read data lines. Read and write cycles are distinguished via dedicated signals. Meaning of address signal (to access CAN FD IP Core) is explained in Table 3. Note that generic parameter "ID" is part of the address! It is possible to configure IP Core address before synthesis! Avalon address bits 19:16 must be matching the ID value during the access, otherwise core is not accessed. With this architecture, it is possible to create up to 16 instances of the IP Core on single Avalon bus. Address bits 23:20 must have constant value 0x3. This requirement is given by the system implemented in [5], since the core was implemented to fit into this system. With small modifications of Memory registers module (refer to) it is possible to remove this requirement.

Address 23:20	Address 19:16	Address 15:0
Constant value 0x3	ID value	Address offset of accessed register

Table 3: IP Core addressing

2.2 Clock domains, external signals synchronization and reset

The whole IP Core is synchronous to one clock signal, **clk_sys**. Every other time period is derived from clk_sys (Time quantum, Bit time...). Every register has **asynchronous reset**, res_n, which is active low, by default. The design is intended to be latch-free.

Input signals of Avalon bus interface and time stamp value are expected to be synchronous to clk_sys and no clock synchronization is implemented on these signals. CAN_RX signal is synchronized by simple synchronization chain with two flip-flops. This synchronization chain is optional, but it is recommended to use it, unless synthesis tools automatically insert synchronization chain! Insertion of this synchronization chain is set by **use_sync** generic.

2.3 Block diagram

The block diagram is shown in Figure 1. Every block is separate VHDL entity and is implemented in a standalone file. CAN controller core consists of several sub-blocks.

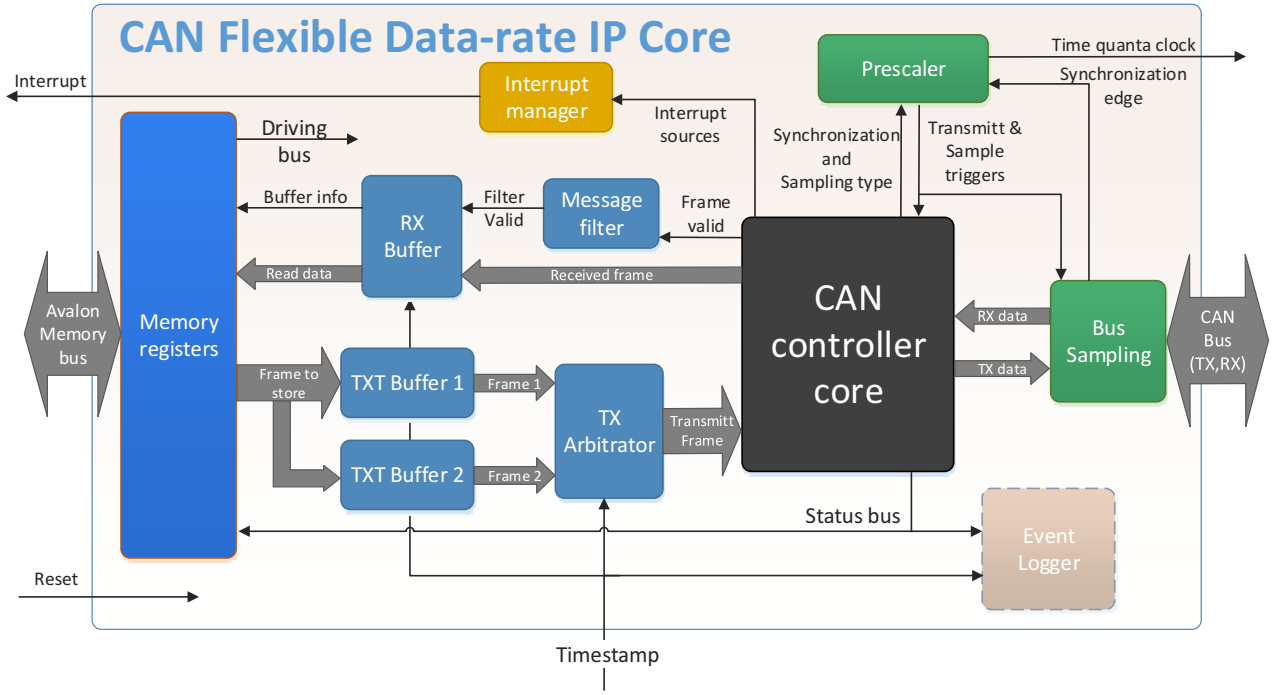

Figure 1: IP function block diagram

2.4 Signals architecture

To simplify VHDL implementation, two parallel "buses" are used: **Driving bus** and **Status bus**. These buses have a form of std_logic vector. Each bus is driven from one entity and connected to other entities.

2.4.1 Driving bus

The Driving bus is used to control functionality of every module from user registers. It is driven in **Memory registers** (). Every parameter of the IP Core which can be configured by the user is part of the Driving bus. An exact definition of Driving bus separates memory registers from other blocks. Thus it is possible to change the registers structure without modification of internal signaling. List of signals in Driving bus is in Appendix A. Driving bus has the following format:

```
signal drv_bus: std_logic_vector(1023 downto 0)
```

2.4.2 Status bus

Status bus is used to provide information about state and state registers of CAN Controller Core. It's driven in CAN Controller Core. Status bus is used by Memory registers and Event logger modules. List of signals in Status bus is in Appendix B. Status bus has the following format:

```
signal stat_bus: std_logic_vector(511 downto 0)
```

2.5 Components description

2.5.1 CAN Core

File: core_top.vhd

CAN Core covers the functionality of serial data transmission according to CAN FD standard. Storing frame to be transmitted, storing received frame, transmission, reception, arbitration, bit stuffing, bit destuffing, CRC calculation, error handling and fault confinement are implemented in this module. Block diagram (just most important signals are displayed) is in Figure 2. Furthermore, valid CRC selection, transmit trigger and receive trigger multiplexing, status bus assignment and bus traffic measurement are implemented in this module.

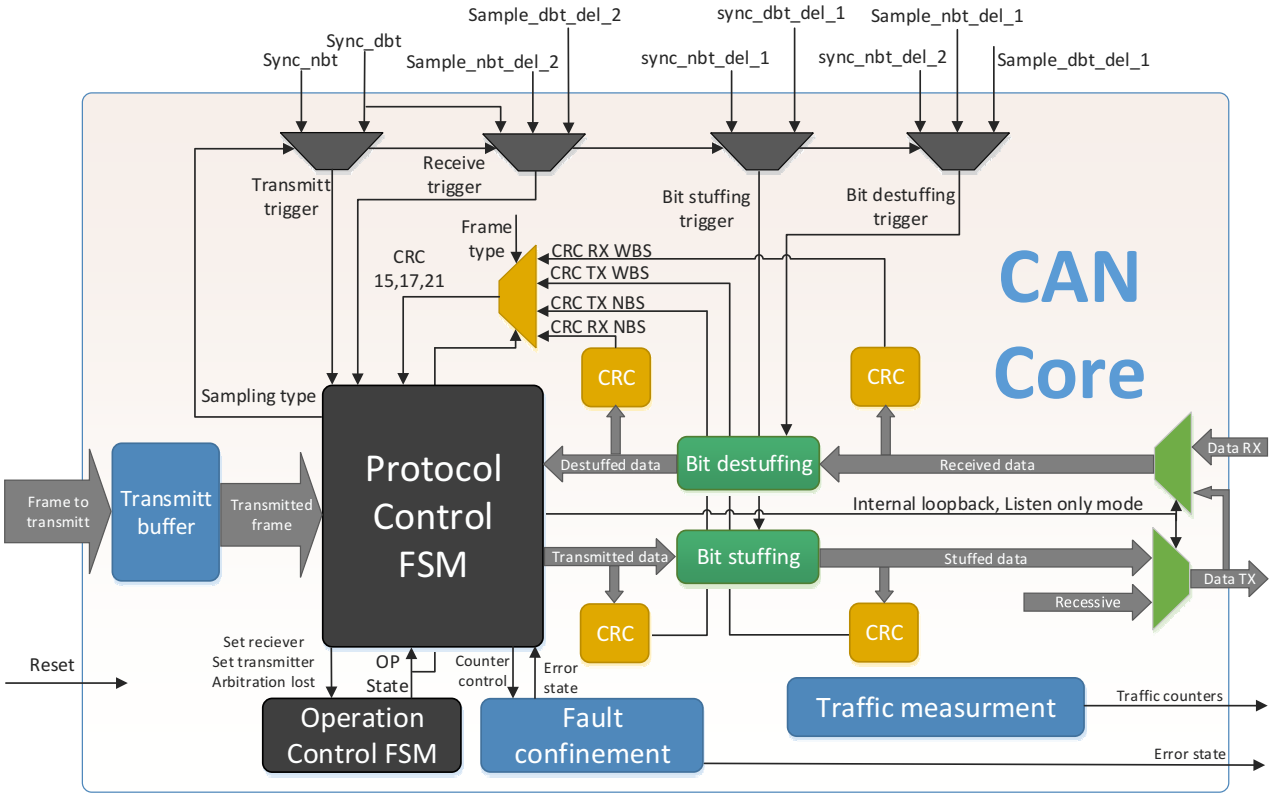

Figure 2: CAN Core block diagram

2.5.1.1 Protocol control

Source file	protocolContorol.vhd
Instanced in	core_top.vhd
Entity name	protocolControl

Description

Protocol control is the main state machine handling CAN FD protocol. Protocol control processes received data with the "rec_trig" signal which is signal two clock cycles delayed from sample signal. Data are transmitted with "tran_trig" signal (in synchronization segment of bit time). Circuit operation starts from off state. Reception of a frame is started when the "hard_sync_valid" input is in logic 1 or dominant is sampled during bus idle. Transmission starts when intermission lasted for at least 3 bit times and data are available for transmission. When frame is available and hard_sync_valid is in logic 1 then transmission also starts. In this case, the arbitration mechanism is applied. Several of Protocol control states have a sub-state machine to simplify the implementation within a field. Protocol control state transition diagram is displayed in Figure 3. Please note that only the main state transitions are displayed!

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
tran_data	512	in	std_logic_vector	Data to transmit
tran_ident	29	in	std_logic_vector	Identifier to transmitt
tran_dlc	4	in	std_logic_vector	DLC code to transmitt
tran_is_rtr	1	in	std_logic	RTR flag to transmitt
tran_ident_type	1	in	std_logic	Identifier type to transmitt (0-BASE,1-EXTEN.)
tran_frame_type	1	in	std_logic	Frame format to transmitt (0-NORMAL,1-FD)
tran_brs	1	in	std_logic	Bit rate shift bit to transmitt
frame_store	1	out	std_logic	Store the frame to Transcieve buffer
tran_frame_valid_in	1	in	std_logic	Frame on input is valid
tran_data_ack	1	out	std_logic	Acknowledge for TXT Buffers, frame is stored
rec_data	512	out	std_logic_vector	Recieved data
rec_ident	29	out	std_logic_vector	Recieved identifier
rec_dlc	4	out	std_logic_vector	Recieved DLC code
rec_is_rtr	1	out	std_logic	Recieved RTR flag
rec_ident_type	1	out	std_logic	Recieved identifier type (0-BASE,1-EXTEN.)
rec_frame_type	1	out	std_logic	Recieved frame format (0-NORMAL,1-FD)
rec_brs	1	out	std_logic	Recieved Bit rate shift bit
rec_crc	21	out	std_logic_vector	Recieved CRC sequence
rec_esi	1	out	std_logic	Recieved ESI bit
OP_state	-	in	oper_mode_type	Operational state (integrating, Transm., Rec.)
arbitration_lost	1	out	std_logic	Arbitration lost (recess. sent, domin. sampled)
is_idle	1	out	std_logic	Bus is idle
set_transciever	1	out	std_logic	Force OP_State to be transciever
set_reciever	1	out	std_logic	Force OP_State to be reciever

Ports (continued)				
Name	Width	Direction	Type	Description
alc	5	out	std_logic_vector	Arbitration lost capture register
error_state	-	in	error_state_type	error_state (active,passive,bus off)
form_Error	1		std_logic	Form error occured
CRC_Error	1		std_logic	CRC error occured
ack_Error	1	out	std_logic	Acknowledge error occured
unknown_state_Error	1	out	std_logic	Controller is in unknown state
bit_stuff_Error_valid	1	in	std_logic	Bit or Stuff error was detected
inc_one	1	out	std_logic	Increase error counter by one
inc_eight	1	out	std_logic	Increase error counter by eight
dec_one	1	out	std_logic	Decrease error counter by one
tran_valid	1	out	std_logic	Frame transmittion finished sucesfully
rec_valid	1	out	std_logic	Frame reception finished sucesfully
ack_recieved_out	1	out	std_logic	Acknowledge was recieved
br_shifted	1	out	std_logic	Bit rate is shifted
tran_trig	1	in	std_logic	Transmitt trigger
rec_trig	1	in	std_logic	Recieve trigger
data_tx	1	out	std_logic	TX Data
stuff_enable	1	out	std_logic	Bit stuffing is enabled
fixed_stuff	1	out	std_logic	Fixed bit stuffing method should be applied
stuff_length	3	out	std_logic_vector	Length of bit stuffing rule
data_rx	1	in	std_logic	RX Data
destuff_enable	1	out	std_logic	Bit de-stuffing is enabled
stuff_error_enable	1	out	std_logic	Stuff error detection is enabled
fixed_destuff	1	out	std_logic	Fixed bit de-stuffing method should be applied
destuff_length	3	out	std_logic_vector	Length of bit de-stuffing rule
dst_ctr	-	in	natural 0 to 7	Stuff or destuff counter modulo 8
crc_enable	1	out	std_logic	CRC calculation is enabled
crc15	15	in	std_logic_vector	CRC 15 result
crc17	17	in	std_logic_vector	CRC 17 result
crc21	21	in	std_logic_vector	CRC 21 result
sync_control	2	out	std_logic_vector	Synchronization type (no, hard, resync)
sp_control	2	out	std_logic_vector	Sampling type (Nominal, Data, Secondary)
ssp_reset	1	out	std_logic	Restart Secondary sampling shift register
trv_delay_calib	1	out	std_logic	Start Transciever delay measurment
bit_err_enable	1	out	std_logic	Bit error detection is enabled
hard_sync_edge	1	in	std_logic	Valid synchronization edge occured
int_loop_back_ena	1	out	std_logic	Internal loopback mode is turned on
PC_State_out	-	out	protocol_type	Protocol state type

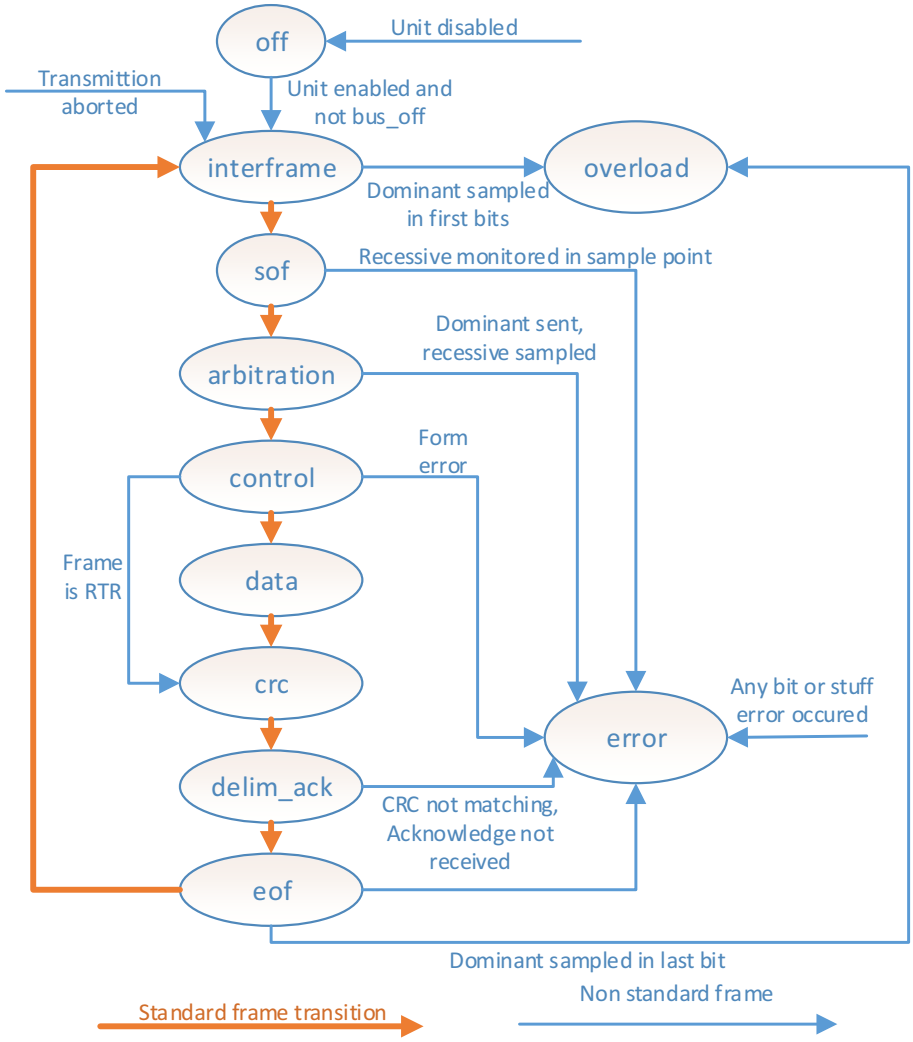

Figure 3: Protocol control state machine

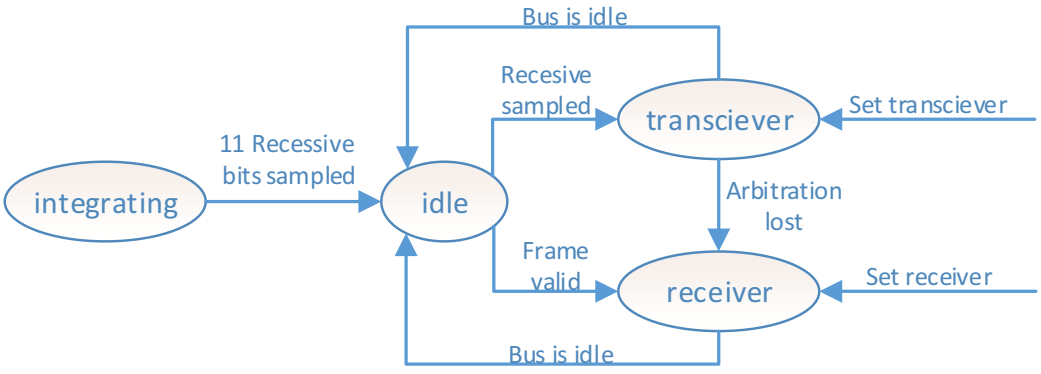

Figure 4: Operation control

2.5.1.2 Operation control

Source file	operationContorol.vhd
Instanced in	core_top.vhd
Entity name	operationControl

Description

Operation control is state machine handling "Operation mode" as defined in [1]. It means unit integration after start, transmission mode (transmitter), reception mode (receiver) and bus idle. Other modes defined in [1] (bus monitoring mode, self -test mode) are implemented via dedicated signals of Driving bus (drv_bus_mon_ena, drv_self_test_ena signals). State machine is described in Figure 4.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
arbitration_lost	1	in	std_logic	Arbitration was lost, signal from Protocol Control
PC_State	-	in	protocol_type	Protocol Control state
tran_valid_in	1	in	std_logic	Frame is valid and should be transmitted
set_transciever	1	in	std_logic	Force transciever state
set_reciever	1	in	std_logic	Force receiver state
is_idle	1	in	std_logic	Unit is Idle according to [1]
tran_trig	1	in	std_logic	Transcieve trigger
rec_trig	1	in	std_logic	Recieve trigger
data_rx	1	in	std_logic	Recieve data
OP_State	-	out	oper_mode_type	Operational State

2.5.1.3 Fault confinement

Source file	faultConf.vhd
Instanced in	core_top.vhd
Entity name	faultConf

Description

Fault confinement module implements Fault confinement state (Error active, Error passive or Bus off), error counters, bit error and stuff error detection. Error counters for Fault confinement (rx_counter, tx_counter) are incremented via inc_one, inc_eight, dec_one signals from Protocol control, therefore all exceptions in [1] (Fault confinement chapter) are managed by Protocol control. These error counters are implemented to be read/write (via Driving bus), so that Fault confinement state can be manipulated by user. This provides an extended testing functionality of CAN controller. Furthermore, two additional counters (err_counter_norm, err_counter_fd), are implemented to distinguish between errors which appeared in Nominal bit rate and Data bit rate.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
stuff_Error	1	in	std_logic	Stuff error ocurred and is validated

Ports (continued)				
Name	Width	Direction	Type	Description
error_valid	1	out	std_logic	Error appeared and is validated
error_passive_changed	1	out	std_logic	Error state change occured
error_warning_limit	1	out	std_logic	Error warning limit was reached
OP_State	-	in	oper_mode_type	Operational state
data_rx	1	in	std_logic	Recieved data
data_tx	1	in	std_logic	Transmitted data
rec_trig	1	in	std_logic	Recieve trigger
tran_trig_1	1	in	std_logic	Transcieve trigger delayed by one clk_sys
PC_State	-	in	protocol_type	Protocol control state
sp_control	2	in	std_logic_vector	Sample type
form_Error	1	in	std_logic	Form error ocurred
CRC_Error	1	in	std_logic	CRC error occured
ack_Error	1	in	std_logic	Acknowledge error occured
unknown_state_Error	1	in	std_logic	PC state is in unknown state
bit_stuff_Error_valid	1	out	std_logic	Bit or stuff error occured and is validated
bit_Error_out	1	out	std_logic	Bit error occured and is validated
inc_one	1	in	std_logic	Increase error counter by one
inc_eight	1	in	std_logic	Increase error counter by eight
dec_one	1	in	std_logic	Decrease error counter by one
enable	1	in	std_logic	Enable Fault confinement module
bit_Error_sec_sam	1	in	std_logic	Bit error from secondary sampling
tx_counter_out	9	out	std_logic_vector	Transmitt error counter
rx_counter_out	9	out	std_logic_vector	Receive error counter
err_counter_norm_out	9	out	std_logic_vector	Error counter in Nominal sampling
err_counter_fd_out	9	out	std_logic_vector	Error counter in Data sampling
error_state_out	-	out	error_state_type	Fault confinement state

2.5.1.4 Bit stuffing

Source file	bitStuffing_v2.vhd
Instanced in	core_top.vhd
Entity name	bitStuffing_v2

Description

Bit stuffing module implements the functionality of bit stuffing into a serial data stream. The number of equal consecutive bits is variable (stuff_length input) as well as fixed bit stuffing method for CRC field of CAN FD frames (fixed_stuff input). Since CAN FD Protocol requires fixed stuff bit to be inserted in the first bit of CRC field, Bit stuffing circuit automatically stuffs first bit after a change on fixed_stuff input (0->1). Circuit processes input data with triggering signal (trig_spl_1) one clock cycle delayed from transmit trigger (tran_trig_1 input). Stuff bit insertion is signaled to Protocol Control to stop the data transmission via data_halt signal. Circuit also provides the number of stuffed bits modulo 8 to implement ISO FD functionality.

Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
data_in	1	in	std_logic	Serial data input stream
trig_spl_1	1	in	std_logic	Trigger signal to sample the data_in
stuff_Error	1	out	std_logic	Stuff error output
data_out	1	out	std_logic	Serial data input (destuffed data)
destuffed	1	out	std_logic	Actual data_out is not valid, bit was destuffed
enable	1	in	std_logic	Circuit enable
stuff_Error_enable	1	in	std_logic	Allow detection of stuff error
fixed_stuff	1	in	std_logic	Use fixed bit stuffing method of CRC CAN FD
length	3	in	std_logic_vector	Length of bit stuffing rule
dst_ctr	-	out	natural range 0 to 7	Destuffed bits counter modulo 8

2.5.1.5 Bit destuffing

Source file	bitDestuffing.vhd
Instanced in	core_top.vhd
Entity name	bitDestuffing

Description

Bit de-stuffing module implements the functionality of Bit de-stuffing from serial data stream. The number of equal consecutive bits is variable ("stuff_length" input) as well as fixed bit stuffing method for CRC field of CAN FD frames (fixed_stuff input). Since CAN FD Protocol requires fixed stuff bit to be discarded in the first bit of CRC field, Bit destuffing circuit automatically discards first bit after the change on "fixed_stuff" input (0->1). Circuit processes input data with triggering signal (trig_spl_1) one clock cycle delayed from sample trigger (trig_spl_1 input). Stuff bit discarded is signaled to Protocol Control to stop the data transmission via "destuffed" signal. Circuit also provides the number of destuffed bits modulo 8 to to Implement ISO FD functionality. Stuff error is detected when N-th bit is not inverse preceding bit where N is length of bit stuffing rule. When circuit is disabled it propagates data from input to output without bit de-stuffing.

Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
data_in	1	in	std_logic	Serial data input stream
trig_spl_1	1	in	std_logic	Trigger signal to sample the data_in
stuff_Error	1	out	std_logic	Stuff error output
data_out	1	out	std_logic	Serial data input (destuffed data)
destuffed	1	out	std_logic	Actual data_out is not valid, bit was destuffed
enable	1	in	std_logic	Circuit enable
stuff_Error_enable	1	in	std_logic	Allow detection of stuff error
fixed_stuff	1	in	std_logic	Use fixed bit stuffing method of CRC CAN FD
length	3	in	std_logic_vector	Length of bit stuffing rule
dst_ctr	-	out	natural range 0 to 7	Destuffed bits counter modulo 8

2.5.1.6 CRC

Source file	tranBuffer.vhd
Instanced in	core_top.vhd
Entity name	tranBuffer

Description

CRC module implements cyclic redundancy check calculation according to [1]. Circuit operation is started with rising edge on enable input (the circuit is still synchronous to clk_sys. Rising edge on "enable" input means detection of 0 to 1 transition). Input data are processed with trig signal. After finishing the calculation, CRC value remains valid until next rising edge on enable input. The result value is propagated to outputs ("crc15","crc17","crc21" signals) of the circuit. All three CRC values are calculated at the same time. CRC calculation is implemented via shift register in order to avoid long combination paths if only combinational implementation would be used. Since ISO FD and Non-ISO are different in the highest bit of shift register initialization, circuit is configurable via Driving bus.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
data_in	1	in	std_logic	Serial data input
trig	1	in	std_logic	Trigger to process the data input
enable	1	in	std_logic	Circuit operation is enabled
crc15	15	out	std_logic_vector	CRC 15 result
crc17	17	out	std_logic_vector	CRC 17 result
crc21	21	out	std_logic_vector	CRC 21 result

2.5.1.7 Transceive buffer

Source file	CRC.vhd
Instanced in	core_top.vhd
Entity name	CRC

Description

Auxiliary component for storing the frame to be transmitted. Circuit stores input frame when logic 1 is detected on "frame_store" input.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
tran_data_in	512	in	std_logic_vector	Data to be stored
tran_ident_in	29	in	std_logic_vector	Identifier to be stored
tran_dlc_in	4	in	std_logic_vector	Data length code to be stored
tran_is_rtr_in	1	in	std_logic	RTR frame type to be stored
tran_ident_type_in	1	in	std_logic	Identifier type to be stored
tran_frame_type_in	1	in	std_logic	Frame format to be stored
tran_brs_in	1	in	std_logic	Bit rate shift bit to be stored

Ports (continued)				
frame_store	1	in	std_logic	Store the frame on input
tran_data	512	out	std_logic_vector	Data for Protocol Control
tran_ident	29	out	std_logic_vector	Identifier for Protocol Control
tran_dlc	4	out	std_logic_vector	Data length code for Protocol Control
tran_is_rtr	1	out	std_logic	RTR frame type for Protocol Control
tran_ident_type	1	out	std_logic	Identifier type for Protocol Control
tran_frame_type	1	out	std_logic	Frame format for Protocol Control
tran_brs	1	out	std_logic	Bit rate shift bit for Protocol Control

2.5.2 CAN bus sampling

Source file	busSync.vhd
Instanced in	CAN_top_level.vhd
Entity name	busSync

Description

This circuit samples the CAN bus. Optional (recommended) synchronization chain with two flip-flops is used to avoid metastability since CAN_RX signal is input from CAN physical layer transceiver (asynchronous signal from outside of FPGA/ASIC). The bus is sampled with sample signal from Prescaler (sample_nbt, sample_dbt). Type of sampling is set by "sp_control" input. "Sync_edge" output signals that valid (recessive to dominant) edge has appeared on CAN_RX input. Furthermore, transceiver delay measurement and secondary sampling point generation are implemented here via two shift registers and counter. Transceiver delay measurement is started by "trv_delay" calib signal. One of the shift registers is dedicated to storing transmitted data in order to compare it with delayed received data (bit error detection for the transmitter in Data bit time).The second one stores sample points generated by Prescaler thus providing secondary sampling point. Additionaly this circuit implements tripple sampling by 3 registers long shift register.A majority of 3 decoder is used with tripple sampling. Tripple sampling is optional and can be configured via Driving bus.

Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
CAN_tx	1	out	std_logic	CAN bus TX line
CAN_rx	1	in	std_logic	CAN bus RX line
sample_nbt	1	in	std_logic	Sample signal (Nominal) from prescaler
sample_dbt	1	in	std_logic	Sample signal (Data) from prescaler
data_tx	1	in	std_logic	Transmitted data from CAN Core
data_rx	1	out	std_logic	Received data to CAN Core
sp_control	2	in	std_logic_vector	Sample control (Nominal,Data,Secondary)
ssp_reset	1	in	std_logic	Clear the shit register for secondary sampling
trv_delay_calib	1	in	std_logic	Start transciever delay measurment
bit_err_enable	1	in	std_logic	Bit error detection is enabled
sample_sec_out	1	out	std_logic	Secondary sampling point
sample_sec_del_1_out	1	out	std_logic	Secondary sampling point del. 1 clock cycle

Ports (continued)				
Name	Width	Direction	Type	Description
sample_sec_del_2_out	1	out	std_logic	Secondary sampling point del. 2 clock cycles
trv_delay_out	16	out	std_logic_vector	Measured value of transciever delay
bit_Error	1	out	std_logic	Bit error is detected

2.5.3 Prescaler

Source file	prescaler_v3.vhd
Instanced in	CAN_top_level.vhd
Entity name	prescaler_v3

Description

Prescaler implements the functionality of bus timing. It counts Time quanta and Bit time clock. It contains state machine for phase of bit time (Synchronisation, Propagation, Phase 1, Phase 2, Reset). It generates synchronization signals (and appropriate delayed signals) and sample signals (and appropriate delayed signals). Synchronization signals (sync_nbt, sync_dbt) are used to transmit data (SYNC part of bit). Sample signals (sample_nbt, sample_dbt) are used to sample the bus (between PH1 and PH2). Furthermore it covers the functionality of Hard synchronization and Resynchronization (sync_control signal). Based on "sample_control" signal Nominal bit time or Data bit time is used.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
clk_tq_nbt	1	out	std_logic	Time quantum clock output (Nominal,unused)
clk_tq_dbt	1	out	std_logic	Time quantum clock output (Data,unused)
sample_nbt	1	out	std_logic	Sample signal (Nominal)
sample_dbt	1	out	std_logic	Sample signal (Data)
sample_nbt_del_1	1	out	std_logic	Sample signal (Nominal) del.1 clock cycle
sample_dbt_del_1	1	out	std_logic	Sample signal (Data) del.1 clock cycle
sample_nbt_del_2	1	out	std_logic	Sample signal (Nominal) del.2 clock cycles
sample_nbt_del_2	1	out	std_logic	Sample signal (Data) del.2 clock cycles
sync_nbt	1	out	std_logic	Synchronization signal (Nominal)
sync_dbt	1	out	std_logic	Synchronization signal (Data)
sync_nbt_del_1	1	out	std_logic	Synchronization signal (Nominal) del.1 clock
sync_dbt_del_1	1	out	std_logic	Synchronization signal (Data) del.1 clock
bt_FSM_out	-	out	bit_time_type	Bit time state output (sync,ph1,ph2,prop,...)
hard_sync_edge_valid	1	out	std_logic	Hard synchronization happened
sp_control	2	in	std_logic_vector	Sample control (Nominal,Data,Secondary)
sync_control	2	in	std_logic_vector	Synchronization type (No sync, hard, resync)

2.5.4 Message filtering

Source file	messageFilter.vhd
Instanced in	CAN_top_level.vhd
Entity name	messageFilter

Description

Message filter validates received frame on the output of CAN Core. If frame identifier matches one of three (A,B,C) mask filters or one range filter and has a valid type and format (NORMAL-FD, BASE-EXTENDED), then "out_ident_valid" is set with one clock cycle delay. Filter logic is combinational but one flip-flop is inserted on the output (out_ident_valid) to avoid long combinational paths. The output is valid if identifier matches at least one filter. The filter is configured via driving bus from user registers. Note that for range comparison identifier is first converted to its decimal value!

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
rec_ident_type_in	29	in	std_logic_vector	Recieved identifier (EXTENSION&BASE)
ident_type	1	in	std_logic	Identifier type (0-BASE,1-EXTENDED)
frame_type	1	in	std_logic	Frame type (0-Normal,1-FD)
rec_ident_valid	1	in	std_logic	Recieved identifier is valid
out_ident_valid	1	out	std_logic	Identifier is matching the filters

2.5.5 Receive buffer (RX)

Source file	rxBuffer.vhd
Instanced in	CAN_top_level.vhd
Entity name	rxBuffer

Description

Receive buffer implements FIFO memory for storing received CAN frames. Basic unit of buffer is 32 bit wide word. Size of the buffer is configurable (generic buff_size) before synthesis. However because of addressing logic, only powers of 2 can be used as buffer size! One frame in received buffer contains from 5 to 20 words. Therefore if size of less than 32 words is used, long CAN FD frames won't be stored and data overrun will appear even if buffer is empty. Data overrun occurs always when there is less free memory in the buffer as size of recieved frame. Recieved frame memory layout is the described in Section 3.16 - RX_DATA

When "rec_message_valid" input signal is in logic 1 first word is stored. In following up to 19 clock cycles remaining words are stored. This requires the received data to be valid for at least 20 clock cycles (register in CAN Core). Since frame is validated at the end of EOF field, until received data are erased by the next frame, bus is in the intermission field. Having minimum 7 clock cycles per nominal bit time this gives minimum 21 clock cycles (3 bit times is minimal length of intermission field) when received data, frame type, frame format, identifier are stable.

Description (continued)				
Always one word (given by "read_pointer" signal) is on the output of circuit to be read. Read_pointer is incremented by one via falling edge on "drv_read_start" signal (part of driving bus). Memory registers block drives this signal to automatically increment the "read_pointer" by one when data from RX buffer are read. Receive buffer is implemented as array of std_logic_vector (see signal "memory" in source code). Memory is not initialized so that synthesis tools can infer True dual port RAM memory. Additional "memory_valid" vector is used to provide erase-on-reset behaviour from users perspective.				
Ports				
iName	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
rec_ident_in	29		std_logic_vector	Recieved identifier
rec_data_in	512		std_logic_vector	Recieved data
rec_dlc_in	4		std_logic_vector	Recieved data length code
rec_ident_type_in	1		std_logic	Recieved identifier type
rec_frame_type_in	1		std_logic	Recieved frame type
rec_is_rtr	1		std_logic	Recieved frame is RTR frame
rec_brs	1		std_logic	Recieved BRS bit
rec_message_valid	1		std_logic	Recieved frame is valid
rx_buf_size	8		std_logic_vector	RX Buffer size
rx_full	1		std_logic	RX Buffer is full
rx_empty	1		std_logic	RX Buffer is empty
rx_message_count	8		std_logic_vector	Number of frames stored in RX Buffer
rx_mem_free	8		std_logic_vector	Number of free words in RX Buffer
rx_read_pointer_pos	8		std_logic_vector	RX Buffer memory read pointer position
rx_write_pointer_pos	8		std_logic_vector	RX Buffer memory write pointer position
rx_message_disc	1		std_logic	Frame was discarded and not stored
rx_data_overrun	1		std_logic	Data overrun flag on RX buffer
timestamp	64		std_logic_vector	Timestamp
rx_read_buff	32		std_logic_vector	Acual read row of data from RX Buffer

2.5.6 Transmit buffer - Time (TXT)

Source file	txtBuffer.vhd
Instanced in	CAN_top_level.vhd, two instances with ID 1 and 2
Entity name	txtBuffer
Description	
Transmit buffer is a memory which contains one CAN frame to be transmitted. It is accessed via committing content of TX_DATA_1 to TX_DATA_20 registers into either TXT buffer 1 or TXT buffer 2 (two instances). If the buffer is full and data are committed new data are not stored in the buffer. To avoid this situation buffer status can be read. In order to save LUTs on FPGA, it is possible to synthesize buffer which only supports frame length up to 8 bytes (via generic useFDsize=false).	

Description (continued)				
TX_DATA_7 to TX_DATA_20 registers have no meaning then. When this "reduced" buffer is synthesized it is still possible to insert frame dlc with length up to 64 bytes. However data 9 to 64 will be lost and only zeroes will loaded to CAN Core and then transmitted. Transmit buffer implementation allows synthesis of the buffer only to LUTs. It is intended for future to infer SRAM memory (FPGA resource).				
Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
tran_data_in	640	in	std_logic_vector	Data (from registers) to be stored into the buffer
txt_empty	1	out	std_logic	Buffer is empty
txt_disc	1	out	std_logic	Frame was discarded due to store into full buffer.
txt_buffer_out	640	out	std_logic_vector	Output of the buffer
txt_data_ack	1	in	std_logic	Buffer can be emptied, data are loaded to CAN Core

2.5.7 TX arbitrator

Source file	txArbitrator.vhd
Instanced in	CAN_top_level.vhd
Entity name	txarbitrator

Description				
TX Arbitrator circuit covers the functionality of frame selection between TXT buffer 1 and TXT buffer 2. Additionally it implements the functionality of propagating a frame to CAN Core at a specific time. If external time stamp value is lower than time stamp specified in a frame, then a frame is propagated to CAN Core. If timestamps in both TXT buffers are lower than external time stamp then the one with the lower time stamp is propagated to CAN Core for transmission. When both timestamps are the same and lower than external time stamp, frame with lower identifier (ID_BASE&ID_EXT) is propagated to output. If both timestamps are equal and both identifiers are equal then frame from Buffer 1 is propagated. Additionally, circuit can be configured to forbid propagation from each of the TXT Buffers.				
Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
txt1_buffer_in	640	in	std_logic_vector	Input Frame from Buffer 1
txt1_buffer_empty	1	in	std_logic	Buffer 1 is empty
txt1_buffer_ack	1	out	std_logic	Acknowledge to Buffer 1
txt2_buffer_in	640	in	std_logic_vector	Input Frame from Buffer 2
txt2_buffer_empty	1	in	std_logic	Buffer 2 is empty
txt2_buffer_ack	1	out	std_logic	Acknowledge to Buffer 2
tran_data_out	512	out	std_logic_vector	Data to be transmitted
tran_ident_out	29	out	std_logic_vector	Identifier to be transmitted

Ports (continued)				
Name	Width	Direction	Type	Description
tran_dlc_out	4	out	std_logic_vector	Data length code to be transmitted
tran_is_rtr	1	out	std_logic	RTR frame type to be transmitted
tran_ident_type_out	1	out	std_logic	Identifier type to be transmitted
tran_frame_type_out	1	out	std_logic	Frame format to be transmitted
tran_brs_out	1	out	std_logic	Bit rate shift bit to be transmitted
tran_frame_valid_out	1	out	std_logic	Output frame is valid
tran_data_ack	1	in	std_logic	Frame is loaded in the CAN Core
timestamp	64	in	std_logic_vector	Timestamp

2.5.8 Interrupt manager

Source file	intManager.vhd
Instanced in	CAN_top_level.vhd
Entity name	intManager

Description

Interrupt manager provides interrupt via "int" output of the CAN_top_level entity. Interrupt sources are configurable from driving registers via Driving bus and every interrupt is marked into interrupt vector (register INT in registers). The interrupt output is active "int_length" clock cycles. If another interrupt source is activated when an interrupt is active, no further interrupts will be produced on output, but interrupt source will be accumulated into interrupt vector. Thus one active period (int_length) of "int_out" signal can be a superposition of up to 11 interrupt sources going active! It is recommended that driver will always read interrupt vector after an interrupt is detected, to determine the events that caused the interrupt! Interrupt manager can be configured to forbid or allow various interrupt sources.

Ports				
Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
error_valid	1	in	std_logic	Interrupt source - Error occured
error_passive_changed	1	in	std_logic	Interrupt source - Fault confinement change
error_warning_limit	1	in	std_logic	Interrupt source - Error warn. limit reached
arbitration_lost	1	in	std_logic	Interrupt source - Node lost arbitration
wake_up_valid	1	in	std_logic	Interrupt source - Wake up signal (unused)
tx_finished	1	in	std_logic	Interrupt source - Frame transmittion finish
br_shifted	1	in	std_logic	Interrupt source - Bit rate is shifted
rx_message_disc	1	in	std_logic	Interrupt source - RX frame discarded
rec_message_valid	1	in	std_logic	Interrupt source - Frame reception finish
logger_finished	1	in	std_logic	Interrupt source - Logging has finished
int_out	1	out	std_logic	Interrupt output
int_vector	11	out	std_logic_vector	Interrupt vector

2.5.9 Memory registers

Source file	registers.vhd
Instanced in	CAN_top_level.vhd
Entity name	registers

Description

Memory registers provide the interface between Avalon compatible 32-bit bus and the Driving bus used to control the CAN FD IP function. Driving bus assignment is implemented in this module. Register structure is in detail described in Chapter 3.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	out	std_logic_vector	Driving bus from memory registers
res_out	1	out	std_logic	Async reset combined with memory write reset
data_in	32	in	std_logic_vector	Avalon data input
data_out	32	out	std_logic_vector	Avalon data output
address	24	in	std_logic_vector	Avalon address
scs	1	in	std_logic	Avalon chip select
swr	1	in	std_logic	Avalon serial write
srd	1	in	std_logic	Avalon serial read
stat_bus	512	in	std_logic_vector	Status bus from CAN Core
rx_read_buff	32	in	std_logic_vector	Read buffer output word
rx_buf_size	8	in	std_logic_vector	Recieve buffer size
rx_full	1	in	std_logic	Recieve buffer is full
rx_empty	1	in	std_logic	Recieve buffer is empty
rx_message_count	8	in	std_logic_vector	Number of frames in Recieve buffer
rx_mem_free	8	in	std_logic_vector	Number of free 32 bit words in Recieve buffer
rx_read_pointer_pos	8	in	std_logic_vector	Read pointer position in Recieve buffer
rx_write_pointer_pos	8	in	std_logic_vector	Write pointer position in Recieve buffer
rx_message_disc	1	in	std_logic	Recieved Message is being discarded
rx_data_overrun	1	in	std_logic	Any data overun occured before
tran_data_in	640	out	std_logic_vector	Data to commit into TXT buffers
txt1_empty	1	in	std_logic	TXT Buffer 1 empty
txt1_disc	1	in	std_logic	Frame was not stored into TXT Buffer 1
txt2_empty	1	in	std_logic	TXT Buffer 2 empty
txt2_disc	1	in	std_logic	Frame was not stored into TXT Buffer 2
trv_delay_out	16	in	std_logic_vector	Transciever delay mesured by Bus sampling
loger_act_data	64	in	std_logic_vector	Actual data word read by logger
log_write_pointer	8	in	std_logic_vector	Logger memory write pointer
log_read_pointer	8	in	std_logic_vector	Logger memory read pointer
log_size	8	in	std_logic_vector	Logger size
log_state_out	-	in	logger_state_type	Actual logger state
int_vector	11	in	std_logic_vector	Interrupt vector

2.5.10 Event logger (optional)

Source file	logger.vhd
Instanced in	CAN_top_level.vhd
Entity name	CAN_logger

Description

Event logger is module capturing events on the CAN bus with its timestamps. FIFO memory is implemented with configurable size (generic "memory_size"). Event logger implements the state machine with "CONFIG" state which is dedicated to reading out previously logged data, configuring triggering and capturing event types. When command is given (via Driving bus, see register LOG_COMMAND description) state machine is moved to "READY" state where it is waiting for triggering condition to move to "RUNNING" state. In "RUNNING" state, events are being captured along with its time stamp and additional information (see register EVENT_TYPE description). When the memory is full circuit automatically moves to "CONFIG" state and sets "logger_finished" output. From "READY" and "RUNNING" state, circuit operation can be aborted via driving bus.

Event logger implements so called "Event harvesting" mechanism. Since there are many events sources in the Status bus, several events can occur at once! Any events that occur are stored in an internal register in the first clock cycle. In next up to N clock cycles (N is the amount of simultaneous events) events are written to the logger memory. If any other event occurs (of the same type) during these N cycles it is not logged. Due to the nature of the system, it is very improbable that repeated event will happend in e.g. 4 clock cycles.

This circuit provides additional testing capability beyond the CAN FD specification. Events can be read out from memory from EVENT_INFO_1 and EVENT_INFO_2 register. The event at the position of "read_pointer" is read out. Read_pointer position can be manipulated via LOG_CMD register.

FIFO Memory is implemented in the same way as RX Buffer memory. It is un-initialized during reset which allows ALTERA Synthesis tools to infer native RAM memory. Additional vector "memory_valid" is stored which contains information about the state of the memory row, so it is possible to achieve erase-on-reset behaviour from user perspective.

Ports

Name	Width	Direction	Type	Description
clk_sys	1	in	std_logic	System clock
res_n	1	in	std_logic	Asynchronous reset
drv_bus	1024	in	std_logic_vector	Driving bus from memory registers
stat_bus	512	in	std_logic_vector	Status bus from CAN Core
sync_edge	1	in	std_logic	Synchronization edge occured
data_overrun	1	in	std_logic	Data overrun occurred
timestamp	64	in	std_logic_vector	Timestamp
bt_FSM	-	in	bit_time_type	Bit time state
loger_finished	1	out	std_logic	Logger finished logging
loger_act_data	64	out	std_logic_vector	Actual Logger data row to be read
log_write_pointer	8	out	std_logic_vector	Logger memory write pointer position
log_read_pointer	8	out	std_logic_vector	Logger memory read pointer position
log_size	8	out	std_logic_vector	Logger size
log_state_out	-	out	logger_state_type	Logger state

2.6 Information processing time

CAN standard defines information processing time (IPT) which can be different for every implementation of CAN controller. IPT determines how many clock cycles controller needs after sampling bus value before knowing following bit value. Implementation of this IP core is based on so-called "triggering signals". For sampling, three triggering signals are used. These signals are mutually delayed by one clock cycle (first for sampling, second for bit de-stuffing, third for processing by CAN Core). This gives the minimal duration of PH2 to 4 clock cycles (one more to actualize states of Fault Confinement and Protocol Control) . Table 4 shows the minimal settings of Bit time in relation to PRESCALER (see section 3.5 ALC_PRESC) settings. Triggering signals (Sync and Sample) in this corner case are shown in Figure 5.

Prescaler value	Min. Prop+Ph1	Min Ph2
1	2	4
2	1	2
4	1	1

Table 4: Information processing time requirements

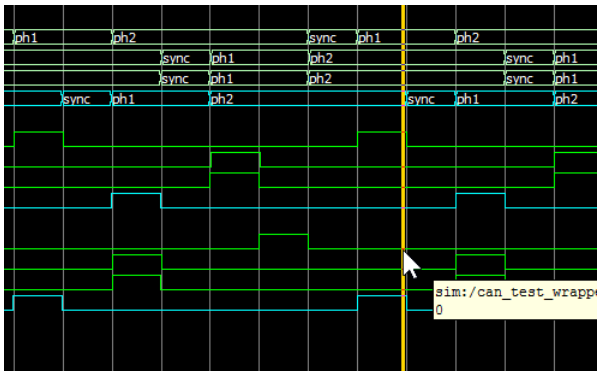

Figure 5: Minimal bit time length

From Table 4 theoretical maximum bandwidth in Data phase of CAN FD IP Core can be calculated. Assuming we have Prescaler set to 1, length of all bit segments together is 7 and clk_sys is 100 Mhz CAN FD IP Core reaches 14,7 Mbit bit rate in Data phase. Note that this situation is theoretical maximum and since timing conditions are very tight, synchronization errors can appear. This configuration was tested (see - Sanity test) and one such a frame is shown in Figure 6.

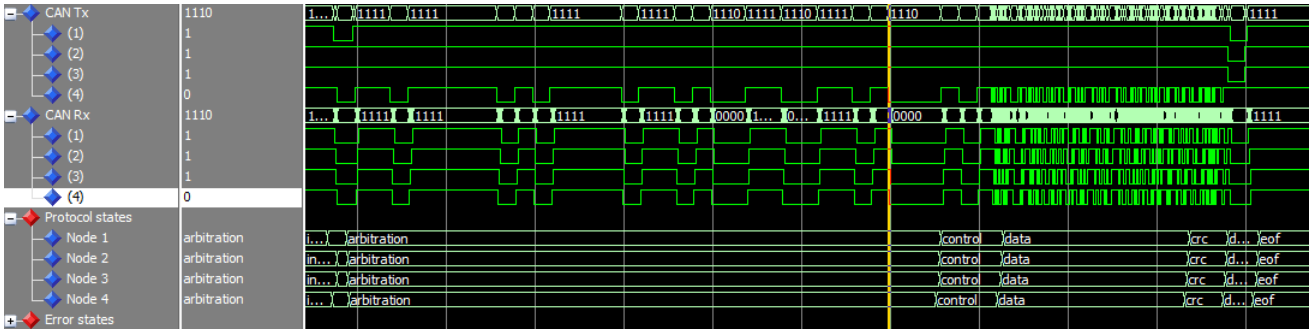

Figure 6: Frame with maximum bit rate

3. Register map

Register structure is selected to be formally similar to SJA1000 controller [8]. Since SJA1000 has 8-bit addresses, to fit into 32-bit address 8-bit registers were concatenated (to save address space). If a register is read or written all 32 bits are read or written at once. Therefore by one read/write also other registers can be affected. Therefore it is necessary to read the register and then write back just partially modified value! Reading or writing just 8 or 16 bits is not possible. Several registers which are read-only or write-only are concatenated together. Read of the write-only register has no effect and always return 0. Write into the read-only register also doesn't have any effect. Some registers (e.g. INT in INTERRUPT REG) are erased by reading. Register map is shown in Table 5.

3.1 DEVICE_ID

Type: Read

The register contains an identifier of CAN FD IP function. It is used to determine whether CAN IP function is mapped correctly on its base address.

Bit offset	31-0
Name	DEVICE_ID
Default value	0x0000CAFD

3.2 MODE_REG

3.2.1 MODE

Type: Read / Write

MODE register (part of MODE_REG) sets special operating modes of the controller. All bits are active in logic 1.

Bit offset	7	6	5	4	3	2	1	0
	ACF	TSM	RTRP	FDE	AFM	STM	LOM	RST
Default value	0	0	1	1	0	0	0	0

RST Activating this bit resets the controller. It has the same effect as logic 0 on res_n input of controller

LOM Listen only mode. In this mode controller only receives data and sends only recessive bits on the bus. When a dominant bus is sent it is rerouted internally so that bus value remains the same. Note that when this mode is enabled controller will not transmit any inserted frame!

Address offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0	Register name
0x0	DEVICE_ID				DEVICE_ID
0x4	SETTINGS	STATUS	COMMAND	MODE	MODE_REG
0x8	INT_ENA		INT		INTERRUPT_REG
0xC	BTR_FD		BTR		TIMING_REG
0x10	BRP_FD	BRP	SJW	ALC	ALC_PRESC
0x14	FAULT_STATE		ERP	EWL	ERROR_TH
0x18	TXC		RXC/CTR_PRES		ERR_COUNTERS
0x1C	ERR_FD		ERR_NORM/CTR_PRES_S		ERR_COUNTERS_SP
0x20	FILTER_A _MASK				FILTER_A_MASK
0x24	FILTER_A _VALUE				FILTER_A_VAL
0x28	FILTER_B _MASK				FILTER_B_MASK
0x2C	FILTER_B _VALUE				FILTER_B_VAL
0x30	FILTER_C _MASK				FILTER_C_MASK
0x34	FILTER_C _VALUE				FILTER_C_VAL
0x38	FILTER_RAN _LOW				FILTER_RAN_LOW
0x3C	FILTER_RAN _HIGH				FILTER_RAN_HIGH
0x40	Reserved		FILTER_CONTROL		FILTER_CONTROL
0x44	Reserved	RX_MF	RX_MC	RX_STATUS	RX_INFO_1
0x48	Reserved	RX_RPP	RX_WPP	RX_BUFF _SIZE	RX_INFO_2
0x4C	RX_DATA				RX_DATA
0x50	Reserved		TRV_DELAY		TRV_DELAY
0x54	Reserved			TX_STAT	TX_STATUS
0x58	Reserved			TX_SET	TX_SETTINGS
0x5C	TX_DATA_1				TX_DATA_1
0x60	TX_DATA_2				TX_DATA_2
...	...				
0xA8	TX_DATA_20				TX_DATA_20
0xAC	RX_COUNTER				RX_COUNTER
0xB0	TX_COUNTER				TX_COUNTER
0xB4	Reserved				
0xB8	LOG_TRIG_CONFIG				LOG_TRIG_CONFIG
0xBC	Reserved				
0xC0	LOG_CAPT_CONFIG				LOG_CAPT_CONFIG
0xC4	LOG_RPP	LOG_WPP	LOG_STAT		LOG_STATUS
0xC8	Reserved			LOG_CMD	LOG_COMMAND
0xCC	EVENT_TIME_STAMP(47:16)				LOG_CAPT_EVENT_1
0xD0	EVENT_TIME_STAMP(15:0)		EVENT_INFO		LOG_CAPT_EVENT_2
0xD4	DEBUG_REGISTER				
0xD8	YOLO_REGISTER				

Table 5: Register map

STM Self test mode. In this mode transmitted frame is considered valid even if acknowledge is not received.

AFM Acceptance filters mode. Activating this bit enables usage of acceptance filters. If disabled, every received frame is stored in the RX buffer.

FDE Enable flexible data rate support. When this bit is inactive, receiving recessive EDL bit (Flexible data rate frame) causes Form error.

RTRP RTR Frame preferred behavior. When RTR frame is sent non-zero dlc code can be inserted. This bit specifies the behavior of controller when this situation happens. If the bit is active then all zeros are sent and inserted dlc is ignored. If a bit is inactive then inserted dlc is sent.

TSM Triple sampling mode. Bus value is sampled three times when this bit is active. Even if this bit is set, triple sampling is used only during Nominal data rate. It is recommended to use triple sampling only at low Bit rates.

ACF When this bit is active acknowledge is not sent even when received CRC matches the calculated one.

3.2.2 COMMAND

Type: Write

Writing logic 1 gives a command to the controller. The meaning of command is different for every bit. This register is automatically erased when a command is finished.

Bit offset	15-12	11	10	9	8
Name	Reserved	CDO	RRB	AT	Reserved
Default value	-	0	0	0	-

AT Writing logic 1 into this bit aborts actual transmission or reception.

RRB Release Receive buffer. Writing logic 1 deletes all data from the Receive buffer.

CDO Clear data overrun flag. Writing logic 1 will clear overrun flag.

3.2.3 STATUS

Type: Read

Register signals various states of CAN controller which are not mutually exclusive. Every bit is active in logic 1.

Bit offset	23	22	21	20	19	18	17	16
	BS	ES	TS	RS	ET	TBS	DOS	RBS
Default value	1	0	0	0	0	0	0	0

RBS Active value of this bit means Receive buffer is not empty.

DOS Data overrun status (flag). An active value of this bit signals frame was lost due to not enough space in the Receive buffer.

TBS TX buffer status. Active value of this bit means both TXT buffers are full and frame can not be inserted for transmission

ET Active value of this bit means that Error frame is beeing transmitted.

RS Active value of this bit signals that controller is receiving a frame.

TS Active value of this bit signals that controller is transmitting a frame.

ES Error status. Active value signals that error warning limit was reached at any of error counters

BS Bus status. Active value signals that bus is idle, the controller is integrating or bus off. Therefore this bit is active when there is no activity on the bus.

3.2.4 SETTINGS

Type: Read / Write

Register with enable bit of the controller.The configuration of retransmission limit for failed frames is also located in this register. Furthermore configuration of ISO FD CAN or CAN FD 1.0 is done here. Every bit is active in logic 1.

Bit offset	31	30	29	28-25	24
	FD_TYPE	ENA	INT_LOOP	RTR_TH	RTRLE
Default value	0	0	0	0	0

RTRLE Active value means that limit of retransmission is enabled.

RTR_TH The maximal amount of retransmission attempts.

INT_LOOP Active value in this bit means that internal loop-back option is permanently enabled (used only for testing).

ENA Active value means that the CAN FD controller is enabled.

FD_TYPE Selection between two possible CAN FD frame formats (0 ISO CAN FD, 1 CAN FD 1.0)

3.3 INTERRUPT_REG

3.3.1 INT

Type: Read - automatically erased after read

This register contains interrupt vector of interrupts that were generated since the last read.

Bit offset	15-11	10	9	8	7	6	5	4	3	2	1	0
Name	Reserved	BSI	RFI	LFI	BEI	ALI	EPI	Reserved	DOI	EI	TI	RI
Default value	-	0	0	0	0	0	0	-	0	0	0	0

BSI Bit-rate shifted interrupt

RFI Receive buffer full interrupt

LFI Event logging finished interrupt

BEI Bus Error interrupt

ALI Arbitration lost interrupt

EPI Node became error passive or bus off interrupt

DOI Data Overrun interrupt

EI Error warning limit interrupt reached

TI Frame successfully transmitted interrupt

RI Frame successfully received interrupt

3.3.2 INT_ENA

Type: Read / Write

Register enables interrupts by different sources. Logic 1 in each bit means interrupt is allowed

Bit offset	31-27	26	25	24	23	22	21	20	19	18	17	16
Name	Reserved	BSIE	RFIE	LFIE	BEIE	ALIE	EPIE	Reserved	DOIE	EIE	TIE	RIE
Default value	-	0	0	0	0	0	1	-	1	1	0	0

BSIE Bit-rate shifted interrupt enable

RFIE Receive buffer full interrupt enable

LFIE Event logging interrupt enable

BEIE Bus Error interrupt enable

ALIE Arbitration lost interrupt enable

EPIE Node became error passive or bus off interrupt enable

DOIE Data Overrun interrupt enable

EIE Error warning limit reached interrupt enable

TIE Frame successfully transmitted interrupt enable

RIE Frame successfully received interrupt enable

3.4 TIMING_REG

3.4.1 BTR

Type: Read / Write

The length of bit time segments for Nominal bit time in Time quanta. Note that SYNC segment always lasts one Time quanta.

Bit offset	15-11	10-6	5-0
Name	PH2	PH1	PROP
Default value	5	3	5

PROP Propagation segment

PH1 Phase 1 segment

PH2 Phase 2 segment

3.4.2 BTR_FD

Type: Read / Write

Length of bit time segments for Data bit time in Time quanta. Note that SYNC segment always lasts one Time quanta.

Bit offset	31	30-27	26	25-22	21-20	19-16
Name	Reserved	PH2_FD	Reserved	PH1_FD	Reserved	PROP_FD
Default value	-	3	-	3	-	3

PROP_FD Propagation segment

PH1_FD Phase 1 segment

PH2_FD Phase 2 segment

3.5 ALC_PRESC

3.5.1 ALC

Type: Read

Bit offset	7-5	4-0
Name	Reserved	ALC_VALUE
Default value	-	0

ALC_VALUE Arbitration lost capture value as defined in[8].

3.5.2 SJW

Type: Read/Write

Synchronisation jump width registers for both Nominal and Data bit times.

Bit offset	15-12	11-8
Name	SJW_FD	SJW
Default value	2	2

SJW Synchronisation jump width in Nominal bit time.

SJW_FD Synchronisation jump widh in Data bit time.

3.5.3 BRP

Type: Read / Write

Baud rate Prescaler register for Nominal bit time. Specifies time quanta duration and synchronization jump width

Bit offset	23-22	21-16
Name	Reserved	BRP
Default value	-	10

BRP Baud-Rate Prescaler

3.5.4 BRP_FD

Type: Read / Write

Baud rate Prescaler register for Data bit time. Specifies time quanta duration and synchronization jump width.

Bit offset	31-30	29-24
Name	Reserved	BRP_FD
Default value	-	4

BRP_FD Baud rate Prescaler

3.6 ERROR_TH

3.6.1 EWL

Type: Read / Write

Error warning limit register. If an error warning limit is reached interrupt can be called. Error warning limit indicates heavily disturbed bus. Note that according to CAN specification this value is fixed at 96 and should not be configurable! The configuration of this value is one of the extra features of this IP Core.

Bit offset	7-0
Name	EWL
Default value	96

EWL Error warning limit

3.6.2 ERP

Type: Read / Write

Error passive limit. When one of error counters (RXC/TXC) exceeds this value, it changes Fault confinement state to error passive. Note that according to CAN specification this value is fixed at 128 and should not be configurable! The configuration of this value is one of the extra features of this IP Core. Note that IP Core always turns to bus_off state once any error counter reaches 255!

Bit offset	15-8
Name	ERP_LIMIT
Default value	128

ERP_LIMIT Error passive limit

3.6.3 FAULT STATE

Type: Read

Fault confinement state of the node. This state can be manipulated by writing into registers RXC/TXC and ERP_LIMIT of ERP register. When these counters are set Fault confinement state changes automatically.

Bit offset	31-19	18	17	16
Name	Reserved	BOF	ERP	ERA
Default value	-	0	0	1

ERA Error active

ERP Error passive

BOF Bus off

3.7 ERROR_COUNTERS

3.7.1 RXC/CTR_PRES

Type: Read / Write

During read the bit meaning is according to RXC register. During write the bit meaning is according to CTR_PRES.
RXC:

Bit offset	15-0
Name	RXC_VAL
Default value	0

RXC_VAL Receive error counter. This register determines fault confinement state of the device.

CTR_PRES:

Bit offset	15-11	10	9	8-0
Name	Reserved	PRX	PTX	CTR_PRES_VAL
Default value	-	0	0	0

CTR_PRES_VAL Value to set the error counter.

PTX Set Transmit error counter into value in CTR_PRES_VAL. This bit is automatically erased after write.

PRX Set Receive error counter into value in CTR_PRES_VAL. This bit is automatically erased after write.

3.7.2 TXC

Type: Read

Bit offset	31-16
Name	TXC_VAL
Default value	0

TXC_VAL Transmit error counter. This register determines fault confinement state of the device.

3.8 ERROR_COUNTERS_SP

Special error counters do not influence fault confinement state of CAN node but enable comparison of error rates in both Bit times. Theses registers can be set to zero. These register are increased by one when any errors appear.

3.8.1 ERR_NORM/CTR_PRES_S

Type: Read/Write

When reading from register bit meaning is according to ERR_NORM register. When writing to this register bit meaning is according to CTR_PRES_SPECIAL.

ERR_NORM:

Bit offset	15-0
Name	ERR_NORM_VAL
Default value	0

ERR_NORM_VAL Number of errors in the Nominal bit time.

CTR_PRES_SPECIAL:

Bit offset	15-13	12	11	10-0
Name	Reserved	EFD	ENORM	Reserved
Default value	-	0	0	-

ENORM Erase error counter for Nominal bit time. This bit is set to zero after the counter is erased.

EFD Erase error counter for Data bit time. This bit is set to zero after the counter is erased.

3.8.2 ERR_FD

Type: Read

Bit offset	31-16
Name	ERR_FD_VAL
Default value	0

ERR_NORM_VAL

Number of errors in the Data bit time.

3.9 FILTER_X_MASK

Type: Read / Write

Bit mask for acceptance filters. Filters A, B, C are available. The identifier format is the same as transmitted and received identifier format. BASE Identifier is 11 LSB and Identifier extension are bits 28-12!

Bit offset	31-29	28-0
Name	Reserved	BIT_MASK_X_VAL
Default value	-	0

BIT_MASK_X_VAL Bit mask for acceptance filters. Logic 1 indicates this bit of Income identifier is compared with the same bit in FILTER_X_VALUE. Logic 0 indicates this bit is not compared.

3.10 FILTER_X_VALUE

Type: Read / Write

The bit value for acceptance filters. Filters A,B,C are available. The identifier format is same as transmitted and received identifier format. BASE Identifier is 11 LSB, Identifier extension is on bits 28-12!

Bit offset	31-29	28-0
Name	Reserved	BIT_VAL_X_VAL
Default value	-	0

BIT_VAL_X_VAL Bit Value for acceptance filters to be compared with income identifier. Only bits set in according to FILTER_X_MASK register are compared.

3.11 FILTER_RAN_LOW

Type: Read / Write

Lower threshold of the Range filter. Note that 29-bit value of range threshold is not the same format as transmitted and received identifier! In TX_DATA_4 (transmitted identifier) BASE Identifier is at 11 LSB bits and Extension at bits 28-12. However, actual decimal value of the Identifier is that BASE identifier is at MSB bits and 18 LSB bits is identifier extension. The unsigned binary value of the identifier must be written into this register!

Bit offset	31-29	28-0
Name	Reserved	BIT_RAN_LOW_VAL
Default value	-	0

BIT_RAN_LOW_VAL Low threshold value

3.12 FILTER_RAN_HIGH

Type: Read / Write

Upper threshold of the Range filter. Note that 29-bit value of range threshold is not the same format as transmitted and received identifier! In TX_DATA_4 (transmitted identifier) BASE Identifier is at 11 LSB bits and Extension at bits 28-12. However, actual decimal value of identifier is that BASE Identifier is at MSB bits and 18 LSB bits is identifier extension. The unsigned binary value of actual value of identifier must be written into this register!

Bit offset	31-29	28-0
Name	Reserved	BIT_RAN_HIGH_VAL
Default value	-	0

BIT_RAN_LOW_VAL High threshold value

3.13 FILTER_CONTROL

Type: Read / Write

Every filter can be set to accept only selected frame types. Every bit active in logic 1.

Bit offset	3,7,11,15	2,6,10,14	1,5,9,13	0,4,8,12
Name	FD_EXT_FRAME	FD_FRAME	EXT_FRAME	BASIC_FRAME
Default value	1,0,0,0	1,0,0,0	1,0,0,0	1,0,0,0

BASIC_FRAME If CAN Basic Frame should be accepted by filter (Bit 0 - Filter A, Bit 4 - Filter B, Bit 8 - Filter C, Bit 12 - Range filter)

EXT_FRAME If CAN Extended Frame should be accepted by filter (Bit 1 - Filter A, Bit 5 - Filter B, Bit 9 - Filter C, Bit 13 - Range filter)

FD_FRAME If FD CAN Basic Frame should be accepted by filter (Bit 2 - Filter A, Bit 6 - Filter B, Bit 10 - Filter C, Bit 14 - Range filter)

FD_EXT_FRAME If CAN FD Extended Frame should be accepted by filter (Bit 3 - Filter A, Bit 7 - Filter B, Bit 11 - Filter C, Bit 15 - Range filter)

3.14 RX_INFO_1

Information register one about FIFO Receive buffer.

Type: Read

3.14.1 RX_STATUS

Bit offset	7-2	1	0
Name	Reserved	RX_FULL	RX_EMPTY
Default value	-	0	1

RX_FULL Receive buffer is full.

RX_EMPTY Receive buffer is empty.

3.14.2 RX_MC

Register with the number of CAN Frames in RX buffer. Note that one frame takes 5 to 20 32-bit words in buffer memory.

Bit offset	15-8
Name	RX_MC_VALUE
Default value	0

RX_MC_VALUE Receive buffer frame count value.

3.14.3 RX_MF

Bit offset	23-16
Name	RX_MF_VALUE
Default value	Buffer size

RX_MF_VALUE Number of free (32 bit) words in RX Buffer

3.15 RX_INFO_2

Type: Read

3.15.1 RX_BUFF_SIZE

Bit offset	7-0
Name	RX_BUFF_SIZE_VALUE
Default value	Depends on the buffer size set before synthesis

RX_BUFF_SIZE_VALUE Size of th Receive buffer. This parameter is configurable before synthesis.

3.15.2 RX_WPP

Bit offset	15-8
Name	RX_WPP_VALUE
Default value	0

RX_WPP_VALUE Write pointer position in th Receive buffer. When a new frame is stored write pointer is increased accordingly.

3.15.3 RX_RPP

Bit offset	23-16
Name	RX_RPP_VALUE
Default value	0

RX_RPP_VALUE Read pointer position in the Receive buffer. When RX_DATA register is read, read pointer is increased accordingly.

3.16 RX_DATA

Type: Read

The recieve buffer data at read pointer position in FIFO. CAN Frame layout in RX buffer is described in Figure 7. By reading data from this register read_pointer is automatically increased, as long as there is next data word stored in the buffer. Next Read from this register returns next word of CAN frame. First stored word in the buffer is "FRAME_FORM", next "TIMESTAMP_U" etc. In detail bits of each word have following meaning:

Word name	Bit offset											
	31-11	10	9	8	7	6	5	4	3	2	1	0
FRAME_FORM	Reserved	ESI	BRS	TBF	FR_TYPE	ID_TYPE	RTR	Reserved	DLC			
TIMESTAMP_U	TS_VAL(63:32)											
TIMESTAMP_L	TS_VAL(31:0)											
IDENTIFIER	ID_EXT	ID_BASE										
DATA_1_4	DATA_1 to DATA_4											
DATA_5_8	DATA_5 to DATA_8											
...	...											
DATA_61_64	DATA_61 to DATA_64											

DLC The Data length code of frame according to [2]. Note that when RTR frame is received this sequence is either zero or non-zero. It depends on RTR preferred behavior of transmitting controller. Note that when DLC is zero or RTR frame was received then there are no DATA words.

RTR Remote transmission request. Logic 1 in this bit means that RTR frame was received. Note that RTR frames are valid only CAN format message. If FR_TYPE=1 then this bit is always zero.

ID_TYPE Logic 1 in this bit means that frame with extended identifier was received. Otherwise, base identifier was recieved.

FR_TYPE Logic 1 in this bit means that CAN FD frame was recieved. Otherwise, CAN frame was received.

TBF Time based format. This bit is always set to logic 1.

BRS Bit-rate shift. Logic 1 in this bit means that bit rate was shifted for DATA and CRC field during transfer of this frame. This bit is valid only for CAN FD frames and it is always zero in CAN frames.

TS_VAL THe timestamp value when controller received the frame. The timestamp value is sampled from timestamp input of IP Core after acknowledge was recieved.

ID_BASE Identifier base.

ID_EXT Identifier extension. It is up to driver to convert ID_BASE and ID_EXT values to unsigned value of identifier

DATA_X Received Data words. Note that in the RX buffer there is minimal amount of DATA words stored that can fit the data bytes. E.g 1-4 bytes -> Only one word, 5-8 bytes -> Two words, and so on.

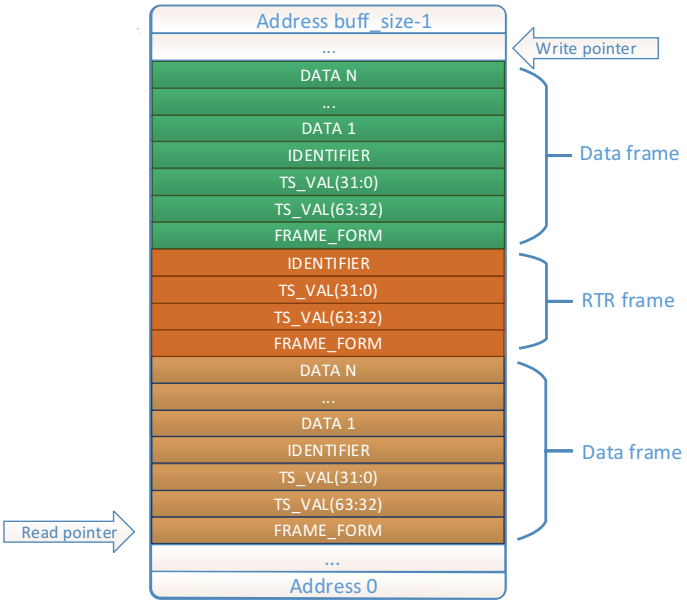

Figure 7: RX Buffer memory layout

3.17 TRV_DELAY

Type: Read

Bit offset	15-0
Name	TRV_DELAY_VALUE
Default value	0

TRV_DELAY_VALUE When sending CAN FD Frame with bit rate shift, transceiver delay is measured to apply secondary sampling point for bit error detection during transmission. After the measurement is done (after EDL bit) it can be read from this register. The value in this register is valid until the start of the next frame. It is recommended to set bit rate shift interrupt and read this value directly after bit rate is shifted in interrupt handling. This register can be used for transceiver TXD to RXD delay verification.

3.18 TX_STATUS

Type: Read

Bit offset	31-2	1	0
Name	Reserved	TXT_2_EMPTY	TXT_1_EMPTY
Default value	-	1	1

TXT_1_EMPTY Active when Transmit buffer 1 is empty.

TXT_2_EMPTY Active when Transmit buffer 2 is empty.

3.19 TX_SETTINGS

Type: Read / Write - Partially automatically erased

This register controls the frame inserton into the TX buffers (The frame to be transmitted in registers TX_DATA_1 to TX_DATA_20). All bits are active in logic 1. Once a frame is committed into the buffer it is transmitted as soon as it passes TX Arbitrator.

Bit offset	31-4	3	2	1	0
Name	Reserved	TXT_2_COMMIT	TXT_1_COMMIT	TXT_2_ALLOW	TXT_1_ALLOW
Default value	-	0	0	1	1

TXT_2_ALLOW Allow transmitting frames from TXT buffer 2.

TXT_1_ALLOW Allow transmitting frames from TXT buffer 1.

TXT_2_COMMIT When active value is written into this bit, frame in registers TX_DATA_1 to TX_DATA_20 are inserted into Transmit buffer 2. Afterward, this value is automatically erased.

TXT_1_COMMIT When active value is written into this bit, frame in registers TX_DATA_1 to TX_DATA_20 are inserted into Transmit buffer 1. Afterward, this value is automatically erased.

3.20 TX_DATA_X

Type: Write

Registers TX_DATA_1 to TX_DATA_20 contain frame to be inserted into transmit buffers. The data in these registers must have the following format in order to be properly sent:

Register name	Bit offset											
	31-11	10	9	8	7	9	5	4	3	2	1	0
TX_DATA_1	Reserved	Reserved	BRS	TBF	FR_TYPE	ID_TYPE	RTR	Reserved	DLC			
TX_DATA_2	TS_VAL(63:32)											
TX_DATA_3	TS_VAL(31:0)											
TX_DATA_4	ID_EXT	ID_BASE										
TX_DATA_5	DATA_1 to DATA_4											
TX_DATA_6	DATA_5 to DATA_8											
...	...											
TX_DATA_20	DATA_61 to DATA_64											

DLC Data length code of frame to transceive according to [2].

RTR Remote transmission request. Logic 1 in this bit means that RTR frame will be sent. DLC value is then sent only when RTRP bit of MODE register is set. Otherwise, DLC value has no meaning when this bit is set and zero length DLC is sent. Note that RTR frames are valid only CAN format frame. If FR_TYPE=1 then this bit has no meaning since there are no RTR frames in CAN FD Frames.

ID_TYPE Logic 1 in this bit means that frame with extended identifier will be sent. Otherwise, base identifier will be sent.

FR_TYPE Logic 1 in this bit means that CAN FD frame will be sent. Otherwise, CAN frame will be sent.

TBF Time based format. This bit should be always set to logic 1.

BRS Bit-rate shift. Logic 1 in this bit means that bit rate will be shifted for DATA and CRC field during frame transfer. This bit is valid only for CAN FD frames, it has no meaning for CAN frames.

TS_VAL An External timestamp value when controller should attempt to start frame transmission. If bus is Idle then transmission will start within next bit time. Otherwise, it will start as soon as bus is idle. If the frame should be transmitted immediately all zeroes must be written into these two registers.

ID_BASE Identifier base.

ID_EXT Identifier extension. Note that MSB bit is sent first. If an extended frame is sent then value in register TX_DATA_4 is not unsigned value of identifier. Identifier should be converted to an unsigned number and 11 MSBs should be written to bits 10:0. 18 LSBs should be written to bits 28:11!! Other bits should be zero.

DATA_X Data to be transmitted. The amount of transceived data is given by DLC code. If the data length is shorter than 64 than remaining data bytes have no meaning. Data are transmitted MSB first.

3.21 RX_COUNTER

Type: Read / Write

Bit offset	31-0
Name	RX_COUNTER_VALUE
Default value	0

RX_COUNTER_VALUE Counter for received frames to enable bus traffic measurement.

3.22 TX_COUNTER

Type: Read / Write

Bit offset	31-0
Name	TX_COUNTER_VALUE
Default value	0

TX_COUNTER_VALUE Counter for transcieved frames to enable bus traffic measurement.

3.23 LOG_TRIG_CONFIG

Type: Read / Write

Register for configuration of event logging triggering conditions. If Event logger is in Ready state and any of triggering conditions appear it starts recording the events on the bus (moves to Running state). Logic 1 in each bit means this triggering condition is valid.

Bit offset	7	6	5	4	3	2	1	0
Name	T_USRW	T_BRS	T_ERR	T_OVL	T_TRV	T_REV	T_ARBL	T_SOF
Default value	0	0	0	0	0	0	0	0

Bit offset	15	14	13	12	11	10	9	8
Name	T_ERPC	T_EWLR	T_ACKNR	T_ACKR	T_CRCS	T_DATS	T_CTRS	T_ARBS
Default value	0	0	0	0	0	0	0	0

Bit offset	31-18	17	16
Name	Reserved	T_RES	T_TRS
Default value	0	0	0

T_SOF Trigger on Start of frame field appears

T_ARBL Trigger on arbitration was lost

T_REV Trigger on valid frame received

T_TRV Trigger on valid frame transmitted.

T_ERR Trigger on error appeared

T_USR When logic 1 is written into this bit event logging is triggered immediately

T_BRS Trigger when bit rate is shifted

T_OVL Trigger when Overload frame is transmitted

T_ARBS Trigger on Arbitration filed starts

T_CTRS Trigger on Control field starts

T_DATS Trigger on Data field starts

T_CRCS Trigger on CRC field starts

T_ACKR Trigger on acknowledge received in ACK slot

T_ACKNR Trigger on acknowledge not received in ACK slot

T_EWLR Trigger on Error warning limit reached

T_ERPC Trigger on Fault confinement state changed

T_TRS Trigger when the unit starts transmitting a new frame.

T_RES Trigger when the unit starts receiving a new frame.

3.24 LOG_CAPT_CONFIG

Type: Read / Write

Register for configuring which events to capture by event logger into the ogger FIFO memory when event logger is running.

Bit offset	7	6	5	4	3	2	1	0
Name	C_ARBS	C_BRS	C_ERR	C_OVL	C_TRV	C_REV	C_ARBL	C_SOF
Default value	0	0	0	0	0	0	0	0

Bit offset	15	14	13	12	11	10	9	8
Name	C_TRS	C_ERC	C_EWR	C_ACKNR	C_ACKR	C_CRCS	C_DATS	C_CTRS
Default value	0	0	0	0	0	0	0	0

Bit offset	31-18	20	19	18	17	16
Name	Reserved	C_OVR	C_DESTUFF	C_STUFF	C_SYNE	C_RES
Default value	0	0	0	0	0	0

C_SOF Capture when Start of frame field appears.

C_ARBL Capture when arbitration was lost.

C_REV Capture when valid frame received.

C_TRV Capture when valid frame transmitted.

C_ERR Capture when error appeared.

C_OVL Capture when error appeared.

C_BRS Capture when bit rate is shifted.

C_ARBS Capture when Overload frame is transmitted.

C_CTRS Capture when Control field starts.

C_DATS Capture when Data field starts.

C_CRCS Capture when CRC field starts.

C_ACKR Capture when Acknowledge was received in ACK Slot.

C_ACKNR Capture when Acknowledge was not received in ACK Slot.

C_EWR Capture when Error warning limit is reached.

C_ERC Capture when Fault confinement state is changed.

C_TRS Capture when the unit starts transmitting.

C_TRS Capture when frame transmission started.

C_RES Capture when receive of frame started.

C_SYNE Capture when synchronization edge was detected (recessive to dominant edge).

C_STUFF Capture when Stuff bit was inserted (transceiver only, one fixed stuff bit before CRC sequence is not captured).

C_DESTUFF Capture when received bit is de-stuffed (receiver and transceiver, one fixed stuff bit before CRC sequence is not captured).

C_OVR Capture data overrun

3.25 LOG_STATUS

Type: Read

3.25.1 LOG_STAT

Bit offset	15-8	7	6-3	2	1	0
Name	LOG_SIZE	LOG_EXIST	Reserved	LOG_RUN	LOG_RDY	LOG_CFG
Default value	Configurable	Configurable	-	0	0	1

LOG_CFG Event logger is in Config state

LOG_RDY Event logger is in Ready state

LOG_RUN Event logger is in Running state

LOG_SIZE Size of event logger. This information is valid only if logger is synthesized! (generic "use_logger")

LOG_EXIST Information whether event logger is synthesized in the circuit (0-no, 1-yes).

3.25.2 LOG_WPP

Bit offset	23-16
Name	LOG_WPP_VAL
Default value	0

LOG_WPP_VAL Logger write pointer position

3.25.3 LOG_RPP

Bit offset	31-24
Name	LOG_RPP_VAL
Default value	0

LOG_RPP_VAL Logger read pointer position

3.26 LOG_COMMAND

Type: Write - Automatically erased

Register for controlling the state machine of Event logger and read pointer position. Every bit is active in logic 1.

Bit offset	7-4	3	2	1	0
Name	Reserved	LOG_DOWN	LOG_UP	LOG_ABT	LOG_STR
Default value	-	0	0	0	0

LOG_STR Start event logging. Move from Config State to Ready state. Has no effect in Ready state or Running state

LOG_ABT Abort event logging. Move from Ready State or Running State to Config State.

LOG_UP Move read pointer one position up.

LOG_DOWN Move read pointer one position down.

3.27 LOG_CAPT_EVENT_1

Type: Read

Upper 32 bits of Event logger memory at read pointer position.

Bit offset	31-0
Name	EVENT_TIME_STAMP(47:16)
Default value	0

EVENT_TIME_STAMP(47:16) Bits 47 to 16 of timestamp when the event occurred. MSB 16 bits from 64-bit time stamp are not captured due to saving capacity after synthesis. 48 bits create enough of resolution to distinguish between events.

3.28 LOG_CAPT_EVENT_2

Type: Read

3.28.1 EVENT_TS(15:0)

Bit offset	31-16
Name	EVENT_TIME_STAMP(15:0)
Default value	0

EVENT_TIME_STAMP(15:0) Bits 15 to 0 of timestamp when the event occurred.

3.28.2 EVENT_INFO

Bit offset	15-8	7-0
Name	EVENT_DETAILS	EVENT_TYPE
Default value	0	0

EVENT_DETAILS Details of the event which appeared. Refer to 6 for bit meanings.

EVENT_TYPE Type of the event which was captured. Refer to 6 for bit meanings

Bit offset	15	14	13	12	11	10	9	8
ERR_DATA	Reserved			FRM_ERR	ACK_ERR	CRC_ERR	ST_ERR	BIT_ERR
BRS_DATA	Reserved						S_DOWN	S_UP
SYNC_DATA	Reserved		SYNC_TYPE		IS_PH2	IS_PH1	IS_PROP	IS_SYNC
STUFF_DATA	Reserved				F_STUFF	STUFF_LENGTH		
DESTUFF_DATA	Reserved				F_DESTUFF	DESTUFF_LENGTH		

FRM_ERR Form error was captured

ACK_ERR Acknowledge error was captured

CRC_ERR CRC error was captured

ST_ERR Bit stuffing error was captured

BIT_ERR Bit error was captured

S_UP Bit rate was shifted from Nominal to Data

S_DOWN Bit rate was shifted from Data to Nominal

IS_SYNC Synchronization edge appeared during Synchronization segment of Bit time

IS_PROP Synchronization edge appeared during Propagation segment of Bit time

Event Type Name	EVENT_DETAILS value	EVENT_TYPE
Start of frame	0x00	0x01
Arbitration lost	0x00	0x02
framereception valid	0x00	0x03
frametransmission valid	0x00	0x04
Overload frame transcieved	0x00	0x05
Error appeared	ERR_DATA	0x06
Bit rate was shifted	BRS_DATA	0x07
Arbitration field started	0x00	0x08
Control field started	0x00	0x09
Data field started	0x00	0x0A
CRC field started	0x00	0x0B
Acknowledge recieved	0x00	0x0C
Acknowledge not recieved	0x00	0x0D
Error warning limit reached	0x00	0x0E
Fault confinement state changed	0x00	0x0F
Transcieve started	0x00	0x10
Recieve started	0x00	0x11
Synchronisation edge appeared	SYNC_DATA	0x12
Stuff bit was inserted	STUFF_DATA	0x13
Bit was destuffed	DESTUFF_DATA	0x14
Data overrun appeared	0x00	0x15

Table 6: Event details register

IS_PH1 Synchronization edge appeared during Phase 1 segment of Bit time

IS_PH2 Synchronization edge appeared during Phase 2 segment of Bit time

STUFF_LENGTH Amount of same consecutive bits after which stuff bit is inserted

DESTUFF_LENGTH Amount of same consecutive bits after which stuff bit is discarded

F_STUFF Whenever Fixed bit stuffing method was used (CRC field of FD Frames)

F_DESTUFF Whenever Fixed bit de-stuffing method was used (CRC field of FD Frames)

3.29 DEBUG_REGISTER

Type: Read

Register for reading out state of the controller. This register is only for debugging purposes!

Bit offset	8	7	6	5-3	2-0
Name	P_DAT	P_CON	PC_ARB	DESTUFF_COUNT	STUFF_COUNT
Default value	0	0	0	0	0

Bit offset	12	11	10	9
Name	P_INT	P_OVR	P_EOF	P_CRC
Default value	0	0	0	0

STUFF_COUNT Actual stuff count modulo 8 as defined in ISO FD protocol. Stuff count is erased in the beginning of the frame and increased by one with each stuff bit until STUFF count field in ISO FD CRC. Then it stays fixed until the beginning of next frame. In non-ISO FD or normal CAN stuff bits are counted until the end of a frame. Note that this field is NOT gray encoded as defined in ISO FD standard. Stuff count is calculated only as long as controller is transceiving on the bus. During the reception this value remains fixed!

DESTUFF_COUNT Actual de-stuff count modulo 8 as defined in ISO FD protocol. De-Stuff count is erased in the beginning of the frame and increased by one with each de-stuffed bit until STUFF count field in ISO FD CRC. Then it stays fixed until beginning of next frame. In non-ISO FD or normal CAN de-stuff bits are counted until the end of the frame. Note that this field is NOT grey encoded as defined in ISO FD standard. De-stuff count is calculated in both. Transceiver as well as receiver.

P_ARB Protocol control is in Arbitration field

P_CON Protocol control is in Control field

P_DAT Protocol control is in Data field

P_CRC Protocol control is in CRC field

P_EOF Protocol control is in End of file field

P_OVR Protocol control is in Overload field

P_INT Protocol control is in Interframe space field

3.29 YOLO_REG

Type: Read

Register for fun :)

Bit offset	31-0
Name	YOLO_VAL
Default value	0xDEADBEEF

YOLO_VAL What else could be in this register??

4. Testbench

CAN FD IP Core is tested in the dedicated testing framework (CANTest) implemented in VHDL and TCL. It provides automated, randomized and reproducible RTL tests. CANTest always follows a simple rule: Each test must have **exact result** whether the test passed or failed! Modelsim ALTERA Edition 6.5 is used to run the framework. All source codes of CANTest are located in **testbench** directory. CANTest can be started from TCL command line by running script **Run_test_framework.tcl**. CANTest contains several basic commands.

```
help          To print the help menu with the command list
test          To configure and execute automated tests
exit          To exit the test framework
```

Note that commands are only top level commands and need to be executed with additional arguments. Individual description of the commands is available in the details of each test.

CANTest framework implements **3 types of tests**. Unit tests test each circuit towards its expected functionality. Feature tests test the functionality of controller as whole in specific situations. Sanity test implements various bus topologies and simulates bus traffic with more controllers.

4.1 Test libraries

4.1.1 CANTestlib.vhd

CANtestLib.vhd provides set of types and procedures which aim to ease the test implementation and debugging. An important feature of CANTest is logging mechanism. It is recommended that each test uses **log** function. Log function reports the events with equal or higher severity than logging level. Severity levels are: info, warning, error. This approach provides an easy way to reduce the unnecessary reports, which are usually introduced during test prototyping.

4.1.2 randomLib.vhd

Since freeware version of Modelsim has limited support for random number generation custom random number generation is used. File **randomlib.vhd** contains pre genreated array and set of procedures for returning random elements. This array of numbers was generated by Matlab and has **Uniform distribution** of values from the interval (0,1). Functions from the library provide values from generated array in a form of std_logic or std_logic vector. Additional function (rand_gaus) provides random numbers with a Gaussian distribution. To obtain random element (vector, bit, etc.) signal **rand_ctr** needs to be provided. This signal represents an index of the returned element (from generated array) and is automatically increased when the procedure ends. Note that this approach provides only pseudo-random number generation, but it is sufficient for the purpose of CANTest framework. By increasing the size of generated array quality of random number generation is also increased. Actual size of the generated array is 3000 elements in range 0 to 1. There is one major

disadvantage of this approach. When random number generation is used from several processes, more rand_ctr signals are needed. Basic test entity has one implicit signal rand_ctr defined for this purpose.

4.1.3 test_lib.tcl

Contains basic TCL routines for handling the test execution like: adding waveforms, starting simulator, extracting results, executing script from a given directory etc.

4.2 Test entities

In order to unify all tests of CANTest two entities are implemented. **CAN_feature_test** and **CAN_test_wrapper**. Both entities are described in the following table:

Source file				CANtestlib.vhd
Instanced in				every test
Entity name				CAN_test
Description				
Basic test entity. Provides common test parameters for all tests. Each test is intended to implement architecture of this entity.				
Ports				
Name	Width	Direction	Type	Description
run	-	in	boolean	Starting signal for the test
iterations	-	in	natural	Number of iterations to be executed in one test run
log_level	-	in	log_lvl_type	Logging severity which should be reported.
error_beh	-	in	err_beh_type	Specifies how does the test behave when error occurs
error_tol	-	in	natural	Maximal amount of errors which does not cause test to fail
status	-	out	test_status_type	Actual status of the test.
errors	-	out	natural	Error counter of the test

Source file	CANtestlib.vhd
Instanced in	every test
Entity name	CAN_test_wrapper

Description

Test wrapper enitity. If several tests are grouped together then one "wrapper" architecture instantiates all the tests. Wrapper architecture is then responsible for generating run signal and observing the results of the test. It it intended that even individual tests will have its own wrapper architecture. Wrapper architectures are intended to be simulator inputs, not test architectures itself.

Ports

Name	Width	Direction	Type	Description
errors	-	out	natural	Error counter of the test

Generics

Name	Width	Default	Type	Description
iterations	-	1000	natural	Number of iterations to be executed in one test run
log_level	-	warning_l	log_lvl_type	Logging severity which should be reported.
error_beh	-	go_on	err_beh_type	Specifies how does the test behave when error occurs
error_tol	-	0	natural	Maximal amount of errors which does not cause test to fail

4.3 Environment variables

Environment variables in Modelsim TCL environment are used to set test parameters. Variables are automatically introduced when CANTest is started. Environment variables are connected to testbench generics and create the connection between TCL and VHDL testbench. CANTest has following environment variables defined:

ITERATIONS Number of iterations that test run should have.

ERR_BEH Test behaviour when error occurs. Possible values are : "go_on" - test runs on, "quit" - test exits immediately. Test implementation must support this feature.

LOG_LEVEL Logging level. Only reports with severity equal or higher than logging level are reported . Possible values are: "info_l", "warning_l", "error_l".

ERR_TOL Error tolerance.The maximal amount of errors that can occur in the test so that test is still passing.

WAIT_ON_END When a test is executed from CANTest framework after test finishes (goes to either "passed" or "failed"), simulation loop is interrupted by "evaluate_test" procedure. Here program control is given back to the TCL script. Based on this variable TCL script is either stalled , or runs on. When running a single test it is good to set this variable to true. Otherwise, simulation will be exited immediately and it will not be possible to observe waveforms of the testbench. On the other hand, when running multiple tests in the row (e.g. test unit all), it is good to set this variable to false. Some test options set this variable to false by itself.

TEST_RESULT This variable is set automatically by TCL script after test finishes. It samples the value of the status signal from an underlying testbench. Do not modify this value manually.

4.4 Simulator settings

CANTest framework is using some simulator settings which are not default. These settings are required for proper functionality:

- default time unit is "fs" instead of "ns". This setting is needed to simulate precise oscillator tolerance!
- "Wave" window under "View" menu must be displayed.

4.5 Unit tests

Unit tests test each circuits functionality on its own and compare it with the expected beahviour. Expected behaviour is implemented in behavioral manner (as circuit model or verification function). It is important that circuit implementation and behavioral model of the circuit are very well separated and share as little source code as possible! Basic architecture of each unit test is in Figure 8 and can be summed up to following points:

1. Generate random inputs to DUT (Device under test).
2. Observe circuit function.
3. Compare outputs of the circuit and outputs of the model, or evaluation function.
4. If a mismatch occurs increase error counter.
5. Repeat steps 1 to 4 until a certain number of iterations was reached.

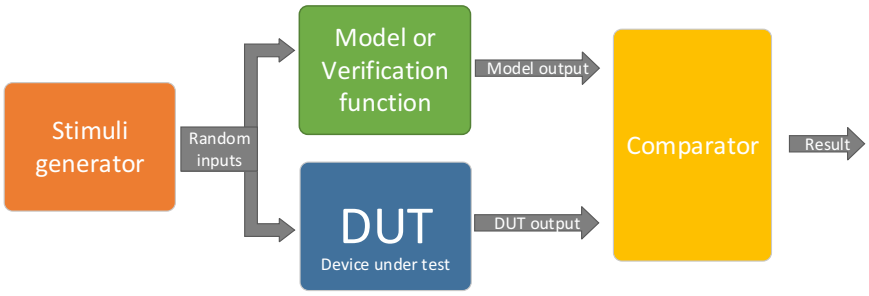

Figure 8: Unit test principle

Each unit test has its own TCL script which adds the most important waveforms into the Wave window of Modelsim simulator. Each unit test must be located in a separate directory under **unit** directory. The test directory (e.g CRC) must contain one **VHDL file** which implements test wrapper architecture and one **TCL file** which adds the waveforms and runs the simulation. Each unit test requires a separate run of the simulator since each unit test implements different DUT, thus simulated architecture is distinct in each unit test.

4.5.1 How to run unit tests?

After configuration of environment variables (in the case of unit tests mainly ITERATIONS, LOG_LEVEL and WAIT_ON_END. If single unit test is executed configure WAIT_ON_END to true), unit tests can be run from command line interpreter of CANTest by typing in the following command:

```
test unit <test_name>
```

Where <test_name> is the name of unit test directory (e.g. CRC, Evnt_logger). Complete run (and print of the results) of all unit tests can be executed via command:

```
test unit all
```

Either command will execute the unit test ITERATIONS times.

4.5.2 How to add a new unit test?

Following steps are recommended when creating new unit test in CANTest:

1. Create a new folder in unit folder.
2. Implement test as an architecture of CAN_test and its wrapper as an architecture of CAN_test_wrapper.
3. Copy existing TCL script from some unit test and modify the waveforms to be observed.

New test can be run from CANTest by **test** command as explained in the previous sub-section.

4.5.3 Existing unit tests

Bit_Stuffing

This unit test includes the unit test of Bit stuffing and bit de-stuffing circuit as well. Both circuits are connected in a chain. As first, random bit is generated then stuffed, then de-stuffed and received. Testbench generates random setting on the input of both circuits. Note that both circuits always have the same settings. Stuff error detection is verified by forcing the value of stuffed bits to be constant for more bit times than length of the stuff rule. De-stuffed data and data before stuffing are compared and when a mismatch occurs error is detected.

Bus_Sampling

Testbench generates random data sequences to be transmitted (as if coming from CAN Core) and appropriate triggering signals (with random bit time settings). Transmitted data on DUT output are inserted into the shift register that is realizing transceiver delay (also randomized). Random errors are inserted into the shift register.The test has 3 parts. In the first part, nominal sampling is used and sampled data are compared with transmitted data. Also, bit error output is observed. If transmitted and sampled data are mismatching and bit_error output is an inactive error is detected. In the second part data sampling is used, but test sequence remains the same as in first part. In third part, secondary sampling is used and additional shift register is implemented in the test which stores the transmitted data. In the beginning of the third part transceiver delay is measured by command (as commin from protocol control in EDL bit). Then received data and delayed transmitted data are compared. Transmitted data are delayed by the length of measured transceiver delay. If a mismatch occurs and bit_error output is not set by DUT error is detected. Note that in the beginning of the third part no bit errors are generated during the time when transceiver delay is being measured. Also, only RECESSIVE bits are generated for a short period at the beginning of the third part. This is to simulate the condition that during EDL bit nominal bit time is still used and transceiver delay is shorter than a nominal bit time. Refer to source code of the test for more detailed explanation.

CRC

CRC unit test generates random bit sequence 10 to 620 bits long. It also generates random drv_fd_type setting. According to ISO FD standard, there is different starting value in CRC shift register in ISO CAN protocol as in non-ISO CAN protocol. Signal "drv_fd_type" switches between these two options. Then CRC is calculated in software by

"calc_crc" procedure. Afterwards, bit sequence is applied to DUT input. Triggering signals are generated by testbench and serial data are processed by CRC circuit. At the end software calculated CRC and DUT output are compared. If these two are not equal, an error is detected.

Evnt_Logger

Testbench generates random event inputs on Status bus in "ev_gen" process. Main test process then generates random setting of event logger (random event type of recording) and starts logging the events (starts DUT as an user would do). After recording has finished it reads all events and checks whether only desired event types were recorded. If any other event is recorded error is detected.

Int_Manager

This testbench is generating random interrupt sources at any time. Periodically it generates random DUT settings on Driving bus. After settings are set, testbench waits for constrained random time. During this time int_counter process is counting how many interrupts are fired by DUT. int_emul process implements interrupt generation in a behavioral way and counts the expected interrupts in another counter. After testbench waits it evaluates whether two counters (acutal interrupts and expected interrupts) are equal and whether interrupt vector on output of DUT is equal to expected interrupt vector. If any of these two pairs are not equal, error is detected.

Message_filter

Random input frame type and identifier are generated on the input of DUT. These inputs emulate received frame from CAN Core. Also, random filter settings are generated on a Driving bus. Testbench waits for one clock cycle since there is one Flip-Flop on the output of message filter circuit. Then validate function is called which compares DUT output and expected output. Validate function implements the behavioral functionality of Message filters. If false is returned then an error is detected.

Prescaler

Prescaler testbench generates random bit time setting on the input of DUT. Afterwards, synchronization and sampling signals are observed on the output of DUT. It is verified that triggering signals are always interleaving (sync,sample,sync...). Minimal information processing time (4 clock cycles) is also verified by "ipt_proc_check" process. Length between generated triggers is observed and compared to a reference value. If any mismatch occurs error is detected. This unit test implements signal generation without synchronization.

Protocol_Control

Protocol control unit test is so far the most complex unit test in CANTest. Its architecture is shown in a header of testbench itself. Random frame is generated on a input of DUT. From this frame expected bit sequence is calculated by "gen_sw_CAN" procedure. This procedure de-facto creates software implementation of CAN FD protocol. After generated frame is applied on the input of DUT,a frame is transmitted by DUT1. CAN bus level is monitored and bit sequence is recorded. Another DUT (DUT2) receives the frame and sends the acknowledge. After the transmission expected and recorded bit sequence are compared. If there is any mismatch error is detected. Also, recieved frame on the output of DUT2 is compared with an original generated frame and mismatch also indicates an error. Note that transmitted bit sequence does not include stuff bits since stuff bits are inserted outside of Protocol control! Fixed value of CRC is transmitted by DUT1 as well as Stuff count field in ISO FD frames.

RX_Buffer

RX Buffer unit test simulates incoming frames from CAN Core. Stimuli generator (stim_gen process) generates random CAN frames on the input of RX Buffer as if coming from CAN Core. Any frame that is stored into the RX Buffer is also stored to testbench memory in_mem. It is observed whether the frame was not discarded by the RX buffer due to lack of free memory in RX Buffer. Another process data_reader reads the data on the output of the frame as an user would do by memory access. Frames which are read on output are stored into testbench memory out_mem. When both memories (in_mem and out_mem) are filled its content is compared and error is detected at any mismatch. Furthermore, testbench keeps it own statistics of the buffer size and free memory based on stored frame size. If a frame is stored but buffer should not have enough space or frame is discarded but buffer should have enough space, an error is also detected. Additionally if Buffer is empty for too long then an error is detected. The testbench is designed that buffer is at constant usage and buffer should never be empty for too long.

TX_Arbitrator

TX_Arbitrator testbench independently generates inputs to the TX Arbitrator in input_gen process. Since the whole circuit is only combinational it provides immediate (in RTL) result. Main test process loop compares the DUT result with expected output via "check_output" procedure. "Check_output" procedure implements a behavioural version of TX arbitrator circuit. If any difference between DUT output and expected output occurs, an error is detected.

TX_Buffer

TXT buffer unit test instantiates two DUTs with IDs 1 and 2 as in CAN FD IP. The difference is that one TXT buffer is set to support FD frames, the other is set not to support FD frames (up to 512 bits of data). Testbench generates inputs of DUTs in data_gen_proc process. Once any buffer is empty random input is applied to DUTs and stored to the buffer which is empty. Testbench also stores the frame into two auxiliarly signals small_mem and big_mem (one for each frame). After data are stored testbench waits for short random time. Main test process periodically reads out the contents of DUTs as long as any of DUTs is signalling that it is not empty. When data are read out, output of DUT is compared with either small_mem or big_mem memory. When any mismatch occurs error is detected.

4.6 Feature tests

Feature tests always test a functionality of CAN controller which is created by cooperation of more circuits and thus can not be tested in a unit test. An architecture of feature tests is different than in unit tests. First of all feature tests have only one test environment, located in feature/feature_env.vhd.The architecture of this test is shown in Figure 9. Two separate CAN controllers are instantiated and connected to the bus. Fixed transceiver delay is emulated (20 clock cycles). Feature tests contain two separate clock sources with 0 ppm and 100 ppm oscillator tolerance. Each clock source is connected to one CAN controller. Test interface to this environment is via two Avalon buses, one for each CAN controller. Note that feature tests implement only one architecture of CAN_test (more exactly CAN_feature test with some additional signals) and one architecture of CAN_test_wrapper. Due to this architecture, it is not necessary to restart the simulator at the start of the new test. All feature tests share two DUTs (two CAN controllers). Both nodes in feature test environment are configured with default values of Bit time registers. It is recommended not to modify this configuration. Several existing feature tests rely on default configuration for their proper function.

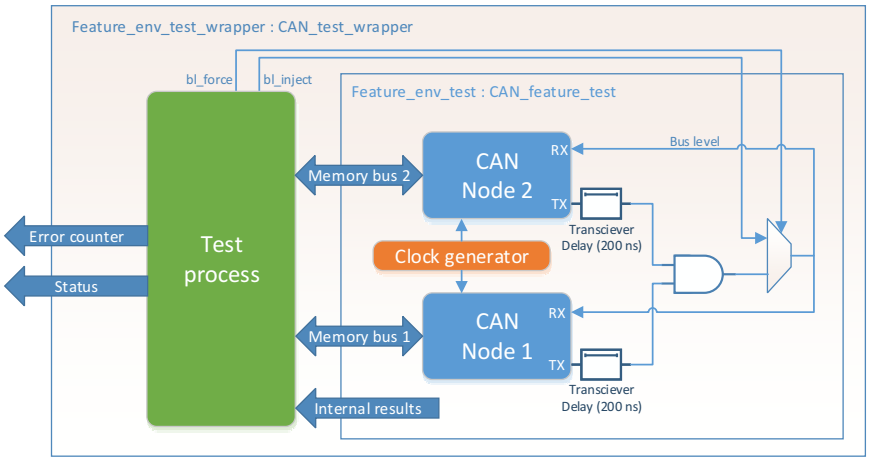

Figure 9: Feature test environment

Each feature test is implemented in a standalone package. This package contains procedure with assignments to memory buses and observations of bus values or internal signals. Each procedure is called from **exec_feature_test** procedure in main test loop. Correct feature test procedure is selected based on the value in signal "test_name". This signal is set in the beginning of each feature test execution.

4.6.1 How to run feature tests?

Since it is possible to execute several feature tests in the row without exiting simulation it is possible to cascade feature tests. Configuration file **feature_config.tcl** contains feature tests which will be run during each run. This file defines TCL list **"FEATURE_FIFO"**. Each entry of the list is another list. Entries of sub-lists have following meaning:

```
list <test_name> <iterations>
```

Where **<test_name>** is the name of the feature test which will be executed as part of feature test run. Name of the feature test must be corresponding to the name of VHDL file where feature test package is implemented without "_feature_tb.vhd" suffix. Name of the feature test is also used in test implementation. Name of the test right-padded to 20 characters is used as unique identification string for the test in "exec_feature_test function. List of all available tests (possible <test_name> values) is in following subsections. **<iterations>** is parameter stating how many times should this feature test be executed. Example of such a configuration file is in Figure 10.

Following command prints the content of configuration file to the command line

```
test feature print_config
```

Simulator with feature test environment can be started either manually or by command:

```
test feature start
```

After simulator is started test only needs to get signal to run via the following command:

```
test feature run
```

Always the most actual value of the config will be executed. After modification of configuration file it might be necessary to restart CANTest framework (type exit in CANTest and re-run Run_test_framework.tcl).

```
global FEATURE_FIFO

quietly set FEATURE_FIFO [list [list forbid_fd 50] \
                        [list abort_transmittion 50] \
                        [list rtr_pref 25] \
                        [list tx_arb_time_tran 100] \
                        [list traf_measure 80] \
                        [list spec_mode 50] \
                        [list soft_reset 5] \
                        [list trv_delay 50] \
                        [list invalid_configs 30] \
                        [list retr_limit 60] \
                        [list overload 20] \
                        [list traf_measure 25]
                ]
```

Figure 10: Feature test example configuration

4.6.2 How to add new feature test?

Following steps must be taken when new feature test is added:

1. Create a new VHDL package (based on existing feature test) in feature directory.
2. Rename the package to have a unique name. Rename the procedure in the package accordingly.
3. Implement the feature test procedure by executing behavioral accesses into the Controller 1 and Controller 2 of feature test environment. The variable "outcome" must be set to true if the test passed and to false if the test failed. One execution of the procedure corresponds to one iteration of the test.
4. Include the package in feature_env.vhd.
5. Add new "elsif" branch into procedure exec_feature_test. The new branch will execute your function. Do not forget to change the string to be compared with test_name signal. This string must be 20 characters right padded unique name. It is the same name as is used in **feature_config.tcl** configuration file. The implementation of CANtest framework uses **force** TCL command to set the test_name signal when execution of next feature test starts. From now on new feature test can be executed as any other feature test.

4.6.3 Existing feature tests

abort_transmittion

Abort transmission tests the option to abort frame transmission by a write to user registers. A frame is generated and inserted for transmission. Abort command is generated and Protocol control state machine is checked. It should

immediately move to Interframe state. If no then an error is detected.

arbitration

Arbitration test tests the arbitration between two nodes in feature test environment. Two random frames with random identifiers are generated. Both are inserted into two nodes simultaneously (nearly simultaneously). Then the expected winner of arbitration process is calculated. Afterwards, frame transmission is observed and real loser of arbitration is determined. Real loser and expected loser are compared and any mismatch causes an error in the test. In order to test also special situations, several constraints were added to frame generation. Arbitration of BASE vs EXTENDED frames with the same BASE identifier, arbitration of RTR vs non-RTR frames with the same identifier and collision during transmission of two identical identifiers are also tested in this test.

fault_confinement

Fault confinement test generates random treshold of Error passive state. Then it sets the Error counters to random values and checks whether Fault confinement state has expected value. If not then an error is detected.

interrupt

Interrupt generation on different events on CAN bus is generated. Firstly, transmit frame and receive frame interrupts are tested. One Node is always transmitter, other one is always a receiver. A frame is generated and transmitted. Test waits for interrupt and then reads interrupt vector from both nodes and checks whether expected bits are in logic 1. If not then an error is detected. Secondly, two conflicting frames (with the same identifier) are sent on CAN bus and an error interrupt is observed. If an interrupt is not generated or marked in interrupt vector then an error is detected. As third part, interrupt on filling RX Buffer and Data overrun is tested. RX buffer is cleared and its size is read. Then CAN frames are generated by other node and sent on the bus. Frames are not read from RX_Buffer. The number of words in RX_Buffer is calculated in test and test checks whether an interrupt is generated once the buffer is full. It also checks if interrupt is generated on the first received frame after buffer was filled. This frame is discarded since there is no space in RX_Buffer. Generation of bit-rate shift interrupt is tested in part 4. CAN FD frame is sent on the bus and two bit-rate shifts are observed during one frame. If interrupt vector does not have expected value error is detected. In the next part, two frames are generated on the bus. These two frames arbitrate. The test is aware of expected winner and checks on interrupt output and interrupt vector of the losing node. An error is detected as in previous parts of the test. In the last part of the test interrupt generation when reaching error warning limit or changing to error passive state is tested.

invalid_configs

This test intends to test the behaviour of the controller when an invalid frame is inserted for transmission. Two situations are covered in this test: 1. CAN Frame (not FD frame) with BRS bit set to '1'. 2. CAN FD Frame with RTR flag set to '1'. FD Frames have no RTR frames according to [1]. In both cases, bits (BRS,RTR) should be ignored and standard frames should be transmitted (CAN Frame without any bit rate shifting, CAN FD frame without any RTR flag). After the transmission it is verified that proper frames were received in the other node. If any of invalid bits were recieved active (BRS='1' or RTR='1') error is detected.

overload

Overload test intends to test transmission of two consecutive overload frames. Since CAN FD implementation is fast enough, it never transmits overload frame due to condition 1 from [1] (Overload frame description). In this test transmission of overload flag due to condition 2 is verified. Random frame is generated and inserted for transmission. Afterwards,

bus value is forced to be DOMINANT in first two bits of intermission. Debug register is used to read out if Protocol control is in Overload state. If overload frame is not transmitted error in the test is detected.

retr_limit

Retransmitt limit test intends to test the feature of limiting the number of frame retransmittions after detection of any error condition. Test generates random frame to be transmitted and inserts it into node 1. The other node has "acknowledge forbidden" bit set in Control register. Random number retransmitt limit is generated and set in node 1. Transmitt error counter in node 1 is erased and frame transmittion is observed. Test waits for expected amount of retransmittions and afterwards checks the Transmitt error counter. If it does not have expected value error in the test is detected.

rtr_pref

RTR Preffered behaviour test intends to test the special feature of CAN FD IP Core which can be set by RTRP bit in MODE register. CAN FD Specification does not clearly define what value of DLC should be sent within RTR frame. Since there is no data field present in RTR frame logically DLC should be "0000". However, it is not required by the protocol. CAN FD IP Core enables sending either inserted DLC or "0000" to be transmitted. Test first sets the behaviour of the Node to send zeroes only. Then it generates RTR frame with non-zero DLC, sends it and verifies that "0000" was transmitted.The second part of the test sets the behaviour to send the actual DLC. Then RTR frame is generated and transmitted. Received DLC in the other node is checked towards the transmitted one. If any mismatch occurs error is detected.

rx_status

RX Buffer status test aims to test proper functionality of RX Buffer from the user perspective. First RX Buffer size is read from the node and RX buffer is cleared. Then free memory and message count status is checked. If any unexpected value is present then error in the test is detected. Random frames are sent on the bus and received by Node 1. After each frame is sent a free memory in the buffer and message count are verified again. During the frame generation test sums the expected number of words in the buffer. Once the buffer should be full, after the transmission of next frame data overrun flag is checked and cleared. If at any point free memory in the buffer or data overrun flag is not matching error in the test is detected.

soft_reset

Soft reset test writes logic 1 into the reset bit of MODE register. Afterwards, it reads back each readable register and compares the value with expected value from this document. If any mismatch occurs an error in the test is detected.

spec_mode

Special modes test tests Self test mode (STM) , Listen only mode (LOM), Acceptance filter mode (AFM) and Acknowledge forbidden mode (ACF). In the first part, node 1 is set to STM and node 2 is set to STM and ACF. A frame is transmitted on the bus by node 1. It is verified that no acknowledge is sent by the node 2. It is also verified that frame was succesfully sent although no acknowledge was transmitted. This is achieved by comparing the transmit error counter before and after transmittion. In the second part node 1 is in STM and node 2 is in LOM. With this setting, the node 2 reroutes the dominant bits internally and node 1 does not expect the acknowledge. No acknowledge is observed on the bus but acknowledge is observed in the protocol control RX_data signal. If acknowledge or error frame would appear on

the bus error in the test is detected. The third part of the test tests AFM mode. It is verified that frame is first stored in the received buffer with AFM turned off. Then the same frame is transmitted with AFM mode turned on, not matching the filters. It is verified that frame is not stored in the RX buffer.

traf_measure

Traffic measurement first reads traffic counters from both nodes. Then it sends 1 to 5 frames on the bus and verifies that traffic counters were increased accordingly in both frames. If counters are not matching its previous value plus the amount of transmitted frames error in the test is detected.

tran_delay

Transceiver delay test generates CAN FD frame with bit-rate shift. Then it sends the frame on the bus. After the transmission frame verifies the content of TRV_DELAY register. This register contains a value of transceiver delay measured during the EDL bit. In feature test environment transciever delay value is constant (200 ns). Considering the two synchronization chain flip-flops (10 ns clock) expected value is 22. If the content of TRV_DELAY is not matching the expected value, an error is detected.

tx_arb_time_tran

TX arbitration and time transmission test tests the functionality of sending the frame in certain time. It also tests the selection between frames in both Transmit buffers. In the first part of the test timestamp is measured and a single frame is transmitted in random time from the acutal value of the timestamp. It is verified that a transmittion starts within first bit time from the expected value of the timestamp. In the second part, two frames are inserted to TXT Buffer 1 and TXT Buffer 2. One of the frames has a higher time of transmission than the other one. Test waits until both frames are sent and verifies that the one with lower time of transmission was transmitted first. In the thrid part two frames with equal transmission time are inserted into both buffers. Frames differ only in the identifier. It is verified that frame with a decimally lower identifier is transmitted.

4.7 Sanity test

Sanity test simulates the behaviour of CAN FD IP Core in the real environment. Sanity test consists of 4 controllers (it can be easily extended) connected on CAN Bus. Block diagram of sanity test is shown in Figure 11. Sanity test has following features:

- Transceiver delay modeling
- Real bus topology
- Oscillator tolerance simulation
- Noise generation
- Traffic emulation and data consitency evaluation

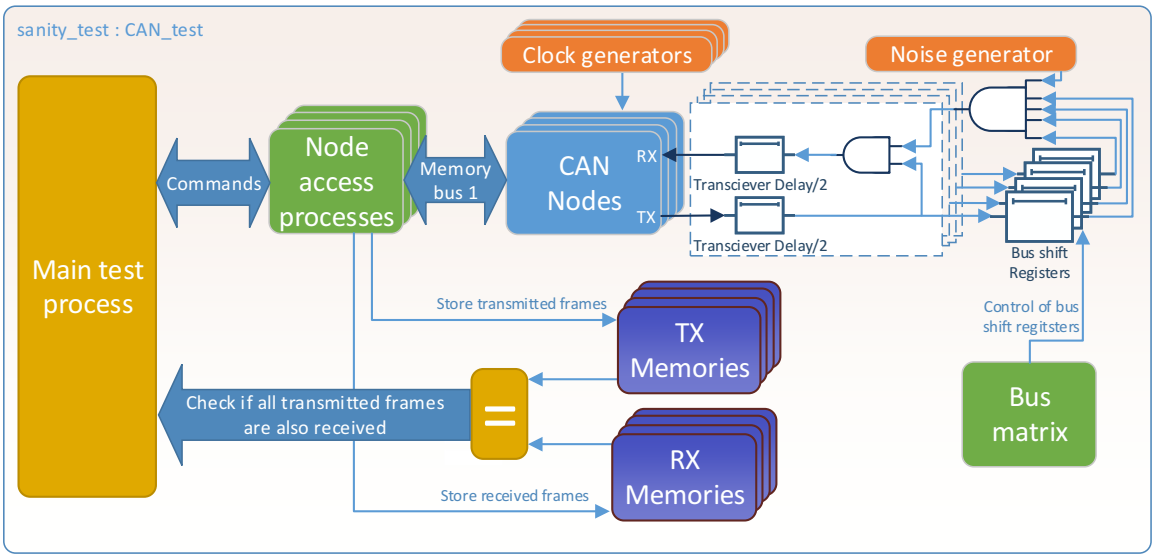

Figure 11: Sanity test block diagram

4.7.1 Transciever delay

TX and RX ports of each CAN FD IP Core are connected to shift registers (signal **transceiver**). Shift registers simulate delay from TX port to RX port. This feature is especially important in fast Data bit-rates where bit time is shorter than transceiver delay. Note that half of the delay is realized in each shift register. In order to find out how to configure transceiver delays refer to

4.7.2 Real bus topology

Topology of CAN Bus is implemented via shift registers, **bus_delay_sr**. Output of each transciever is connected to input of one shift register. Each shift register has capacity of 5000 entries and is shifted each 500 ps. Assuming 1 ns equals approximately 20 cm of electric signal transport delay, bus lengths have resolution 10 cm. Delayed signal from each transciever is connected via AND gate to each reciever. Length of the delay is given by **Bus matrix**:

$$M_B = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} \\ d_{21} & 0 & d_{23} & d_{24} \\ d_{31} & d_{32} & 0 & d_{34} \\ d_{41} & d_{42} & d_{43} & 0 \end{bmatrix}$$

Where d_{ab} is the delay from the transceiver "a" to the receiver "b". Each row of the matrix represents delay from one node to rest of the bus. In general case Bus matrix is n*n Matrix where N is the amount of CAN FD Controllers on the bus. Note that diagonal of the bus is equal to zero. It is assumed it takes infinitely short time from transceivers TX pin to its RX pin. The transciever delay is already modeled before the node is connected to the bus. Bus matrix is symetrical along its main diagonal. Delay from Node A to Node B is equal as Delay from Node B to Node A. Bus Matrix can describe every configuration of the Bus with N controllers. Bus topology in sanity tests does not simulate any analog effects, its purpose is to simulate only bus delay cause by bus cable. In order to simplify the Bus matrix calculation most common topologies are implemented in the sanity test.

Bus

Bus topology is shown in Figure 12. Bus matrix for Bus topology is:

$$M_B = \begin{bmatrix} 0 & l_1 & l_1+l_2 & l_1+l_2+l_3 \\ l_1 & 0 & l_2 & l_2+l_3 \\ l_1+l_2 & l_2 & 0 & l_3 \\ l_1+l_2+l_3 & l_2+l_3 & l_3 & 0 \end{bmatrix}$$

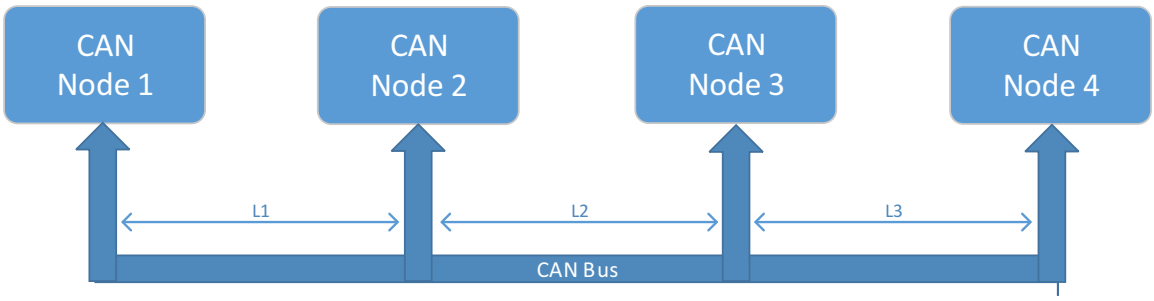

Figure 12: Bus topology

Star

Star topology is shown in Figure 13. Bus matrix for Star topology is :

$$M_B = \begin{bmatrix} 0 & l_1+l_2 & l_1+l_3 & l_1+l_4 \\ l_1+l_2 & 0 & l_2+l_3 & l_3+l_4 \\ l_1+l_3 & l_2+l_3 & 0 & l_3+l_4 \\ l_1+l_4 & l_2+l_4 & l_3+l_4 & 0 \end{bmatrix}$$

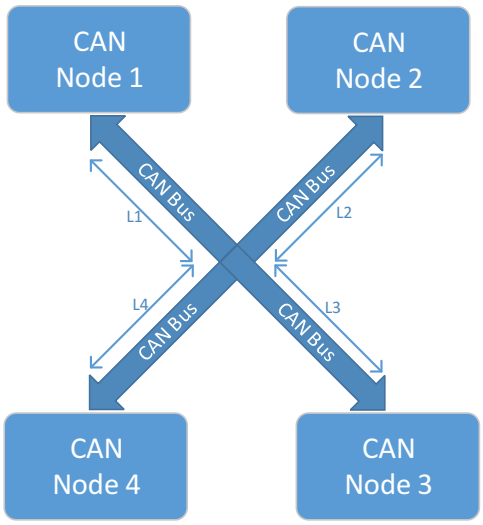

Figure 13: Star topology

Tree

Tree topology is shown in Figure 14. Bus matrix for Tree topology is :

$$M_B = \begin{bmatrix} 0 & l_1+l_2 & l_1+l_3+l_5 & l_1+l_4+l_5 \\ l_1+l_2 & 0 & l_2+l_3+l_5 & l_2+l_4+l_5 \\ l_1+l_3+l_5 & l_2+l_3+l_5 & 0 & l_3+l_4 \\ l_1+l_4+l_5 & l_2+l_4+l_5 & l_3+l_4 & 0 \end{bmatrix}$$

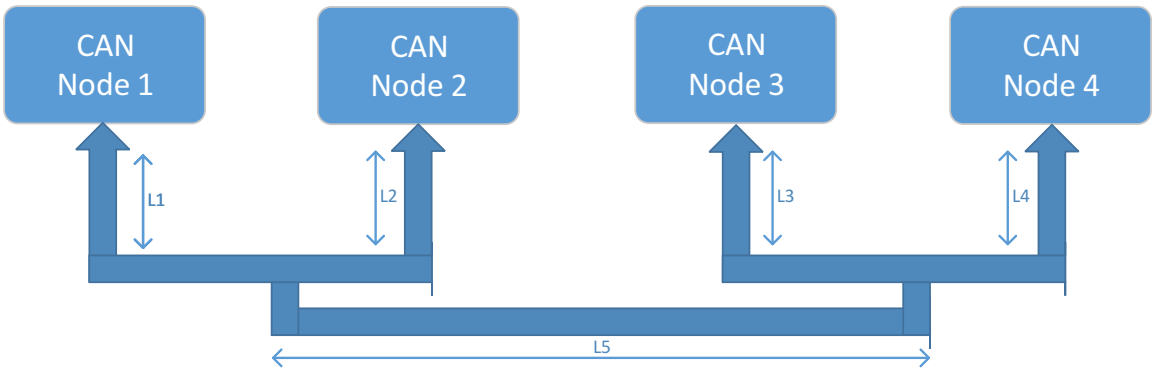

Figure 14: Tree topology

Ring

Ring topology is shown in Figure 15. Bus matrix for Ring topology is :

$$M_B = \begin{bmatrix} 0 & min(l_1, l_2+l_3+l_4) & min(l_1+l_2, l_3+l_4) & min(l_4, l_1+l_2+l_3) \\ min(l_1, l_2+l_3+l_4) & 0 & min(l_2, l_1+l_3+l_4) & min(l_2+l_3, l_1+l_4) \\ min(l_1+l_2, l_3+l_4) & min(l_2, l_1+l_3+l_4) & 0 & min(l_3, l_1+l_2+l_4) \\ min(l_4, l_1+l_2+l_3) & min(l_2+l_3, l_1+l_4) & min(l_3, l_1+l_2+l_4) & 0 \end{bmatrix}$$

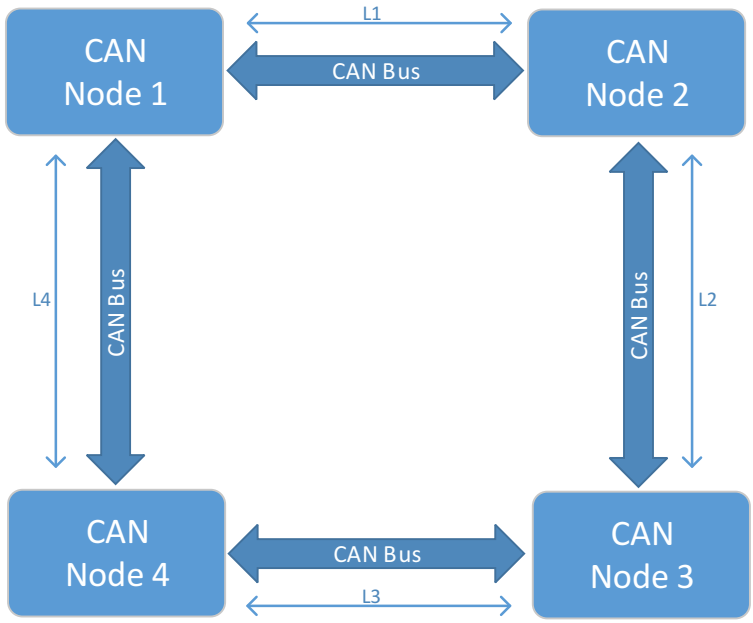

Figure 15: Ring topology

Custom

In custom topolgy user/tester can set arbitrary values to Bus matrix. Bus matrix in custom topology is:

$$M_B = \begin{bmatrix} 0 & l_1 & l_2 & l_3 \\ l_1 & 0 & l_4 & l_5 \\ l_2 & l_4 & 0 & l_6 \\ l_3 & l_5 & l_6 & 0 \end{bmatrix}$$

4.7.3 Oscillator tolerance

Each controller in the sanity test environment has separate clock source with 100 Mhz frequency. Since oscillators in real devices are never absolutely exact, oscillator tolerance needs to be simulated. Each controllers clock has its own tolerance in ppm. Each tolerance is maximal possible tolerance, that means the clock is generated with this error. This approach covers the worst possible case that can happen. In order to properly simulate oscillator tolerance time resolution of Modelsim needs to be set to "fs". If set to "ps" or "ns" error in each clock cycle is lower than shortest time unit and will be rounded to zero and clk_sys of all controllers will have frequency exactly 100 Mhz. During the simulation, it is possible to check whether oscillator tolerance is used properly. If two nodes have different oscillator tollerance then according input timestamp signals should gain different values after simulation runs for some time. In order to find out exactly how to configure oscillator tolerance refer to .

4.7.4 Noise generation

In order to simulate glitches on the bus random binary nosie is generated on RX pins of each transceiver. When noise glitch is generated there is 50 % probability it will be received on the RX pin. Once the glitch is received (signal "noise_force") unit receives only the generated value of noise, thus it is possible to receive recessive bit when a dominant bit is on the

bus. Glitches are not received at all nodes uniquely in order to distinguish between global (all nodes receive glitch) and local (only some nodes receive glitch) errors. Noise generator has two parameters : Noise width and Noise gap. Noise width is the length of generated glitch. Noise gap is time between two generated glitches. Both of these parameters have Gaussian distribution and it is possible to configure them over following parameters:

nw_mean Mean value of noise width (in nanoseconds)

nw_var Variance of noise width (in nanoseconds)

ng_mean Mean value of noise gap (in nanoseconds)

ng_var Variance of noise gap (in nanoseconds)

Each configuration parameter corresponds to VHDL signal. To find out exactly how to configure these parameters refer to .

4.7.5 Traffic emulation and data consitency evaluation

Sanity test generates random CAN frames and inserts these frames to be transmitted on the CAN bus. Each frame that is inserted into Node, is also inserted into according Transceive memory (tx_mems signal). The content of transceive memories monitors what was sent on the bus. Sanity test reads received frames from CAN Nodes and stores them into Recieve memories (rx_mems signal). After TX_memories are filled traffic generation is finished. In order for sanity test to pass the data consistency must be kept. That means that each frame which was transmitted by any node must be received by all other nodes! Contents of Transceive and Recieve memories is compared and data consistency is evaluated. One iteration of sanity test (with given configuration) can be summed up into following steps:

1. Generate frames and insert them to CAN Nodes.
2. Wait until all frames are received or unit turned error passive (indicates heavily disturbed bus).
3. Compare TX Memories and RX Memories contents for data consistency. If any frame was not properly received or dropped error is detected and test fails.

4.7.6 Sanity test configuration

Sanity test has its own configuration file which contains bus configurations for sanity test execution. This file is located in **sanity/sanity_config.tcl**. Each configuration can be iterated different amount of times. Each iteration consists of sequence from . Sanity configuration file is a TCL file. it defines list SANITY_CFG. Each element of the list represents one configuration and it is another list. Entries of the sublist have following meaning:

```
list
    <topology> <l1> ... <l6>
    <td1> ... <td4>
    <e1> ... <e4>
    <nw_mean> <nw_var> <ng_mean> <ng_var>
    <brp> <brp_fd> <prop> <ph1> <ph2> <sjw> <prop_fd> <ph1_fd> <ph2_fd> <sjw_fd>
    <desc> <iterations>
```

Each element of the sublist have following meaning:

topology Bus topology used in this configuration. Valid values are : "bus" , "star" , "tree", "ring". For more information about topologies refer to .

l1 ... l6 Length of the bus segments. These parameters have a different meaning for each topology. Please refer to to find out the exact meaning of these parameters. Please note that if a parameter is not used in Bus Matrix then it can have an arbitrary value.

td1 ... td4 Transciever delays of each node in multiples of 10 ns (e.g. transciever delay = 220 ns , then td(n)=22). Refer to .

e1 ... e4 Oscillator errors (maximal tollerance) of each node in ppm. Refer to.

<**nw_mean**> Mean value of noise width (ns)

<**nw_var**> Variance of noise width (ns)

<**ng_mean**> Mean value of gap between two noise glitches (ns)

<**ng_var**> Variance of between two noise glitches (ns)

<**brp**> Baud rate prescaler in nominal bit time

<**brp_fd**> Baud rate prescaler in data bit time

<**prop**> Length of PROP segment in nominal bit time

<**ph1**> Length of PH1 segment in nominal bit time

<**ph2**> Length of PH2 segment in nominal bit time

<**sjw**> Length of Synchronization jump width in nominal bit time

<**prop_fd**> Length of PROP segment in data bit time

<**ph1_fd**> Length of PH1 segment in data bit time

<**ph2_fd**> Length of PH2 segment in data bit time

<**sjw_fd**> Length of Synchronization jump width in data bit time

<**desc**> Describtion of this configuration

<**iterations**> Number of iterations this configuration should be repeated

Example of one configuration (one sub-list) is shown in figure 16.

```
[ list star 10 10 10 10 0.0 0.0 \
        10 10 10 10 \
        0 5 10 15 \
        10.0 5.0 100000.0 10000.0 \
        4 1 8 8 8 3 3 3 3 2 \
        "1Mb/10Mb 20 m Star" 10
] \
```

Figure 16: Sanity test example configuration

4.7.7 How to run sanity test?

Sanity test can be run from CANTest framework. Once the TCL configuration file contains valid configurations sanity simulation can be opened (most important waveforms are also added) via the following command:

```
test sanity start
```

Since sanity test can run several hours or days simulation waveform file in Modelsim project directory can reach extreme sizes (from 10s up to 100s of GBs). In order to avoid this situation sanity test can be started in "silent" mode. In silent mode only the most important waveforms are added. Sanity test can be started in silent mode via the following command:

```
test sanity start silent
```

Afterwards,test execution is started with command:

```
test sanity run
```

After sanity test runs with all configurations it prints out the results of each configuration run to the command line.

5. Synthesis

CAN FD IP Core is synthesized as part of system developed in [5]. ALTERA FPGA EP4CE55F23C8N with 55856 LUTs is used as target device. Synthesis is performed in Quartus II v 9.1 software. Design size and performance depends on configuration constants (size of buffers and event logger). Synthesis was performed several times and various results were obtained. Settings for balanced Synthesis and Fitting were used. Table 7 shows these results. Data shown in the table exclude the rest of the design from [5] and show only resources used by CAN FD IP Core. For timing analysis TimeQuest Timing analyzer is used. Note that results of Timing analysis which are shown in last column use 85 °C Slow timing model (most critical model).

RX Buffer size [words]	Logger size [events]	TX Buffer size	LC Combinationals	LC Registers	Memory bits	Maximal frequency [Mhz]
32	0(unused)	Basic size	6738	2683	2048	105.12
64	0(unused)	Basic size	6907	2751	4096	112.65
128	0(unused)	Basic size	7391	2883	8192	101.88
256	0(unused)	Basic size	9298	3143	16384	101.85
512	0(unused)	Basic size	9375	3659	32768	97.62
32	0(unused)	FD size	8340	4475	2048	106.38
64	0(unused)	FD size	8500	4543	4096	105.95
128	0(unused)	FD size	8885	4675	8192	101.9
256	0(unused)	FD size	9571	4953	16384	103.78
512	0(unused)	FD size	10968	5451	32768	82.01
32	8	FD size	8718	4677	2520	97.63
32	16	FD size	8767	4688	2992	97.83
32	64	FD size	8912	4742	5824	89.63
64	8	FD size	8894	4745	4680	99.95
64	16	FD size	8914	4756	5040	95.17
64	64	FD size	9072	4810	7872	89.45
128	8	FD size	9282	4877	8864	87.99
128	16	FD size	9307	4888	9136	92.3
128	64	FD size	9456	4942	11968	86.55

Table 7: Design size in FPGA

6. FPGA Verification

The functionality of CAN FD IP Core is verified in real hardware. The device developed in [5] is used for this purpose. CAN FD IP Core is synthesized into FPGA and accessed over EMIF interface from memory of Texas Insruments automotive MCU (ARM) via driver described in previous chapter. TJA1041 is used as transceiver of physical layer.

The main purpose of this testing is to reveal errors which were not covered by CANTest framework. In order to achieve this reference controllers are used for communication. Without reference controller there is always a chance that error will be compensated by receiving node thus error remains undetected.

In the first part, reference controller CANoe program with CAN card was used. Since CANoe program did not support the FD interface it was used only for CAN frames. Both transmission and reception are verified. Basic Identifiers, Extended Identifiers, RTR frames and Arbitration mechanism were verified via logic analyzer. Transceived and received data were compared to make sure data consistency was not corrupted. 1 Mbit/s bit-rate was used.

In the second part CAN FD frames were verified. Atmel SAMV71 Ultra development board ([15]) was used as reference controller. Both transmission and reception were verified. Maximum payload length of 64 bytes was used. Several Bit-rates were tested during this testing:

- 250 kbit/s Nominal bit-rate, 1 Mbit/s Data bit-rate
- 500 kbit/s Nominal bit-rate, 2 Mbit/s Data bit-rate
- 1 Mbit/s Nominal bit-rate, 2 Mbit/s Data bit-rate

Faster bit rates were not verified in real hardware since used transceivers did not support bit-rates over 2 Mbit/s (actually TJA1041 supports only 1 Mbit/s but works fine for 2 Mbit/s). Basic identifier, Extended identifiers, Bit-rate shit, no Bit-rate shift were verified as part of this test. The test setup is shown in Figure 18. An example of frame sampled by logic analyzer is shown in Figure 17. Note that SAMV71 MCU from Atmel contains only non-ISO version of CAN FD protocol (it was released before the CRC issue appeared), thus the testing covers only non-ISO FD protocol.

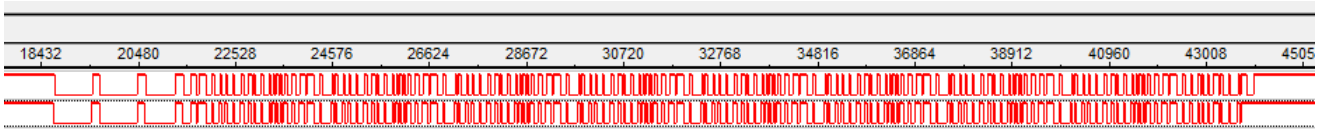

Figure 17: Sampled 64 byte frame

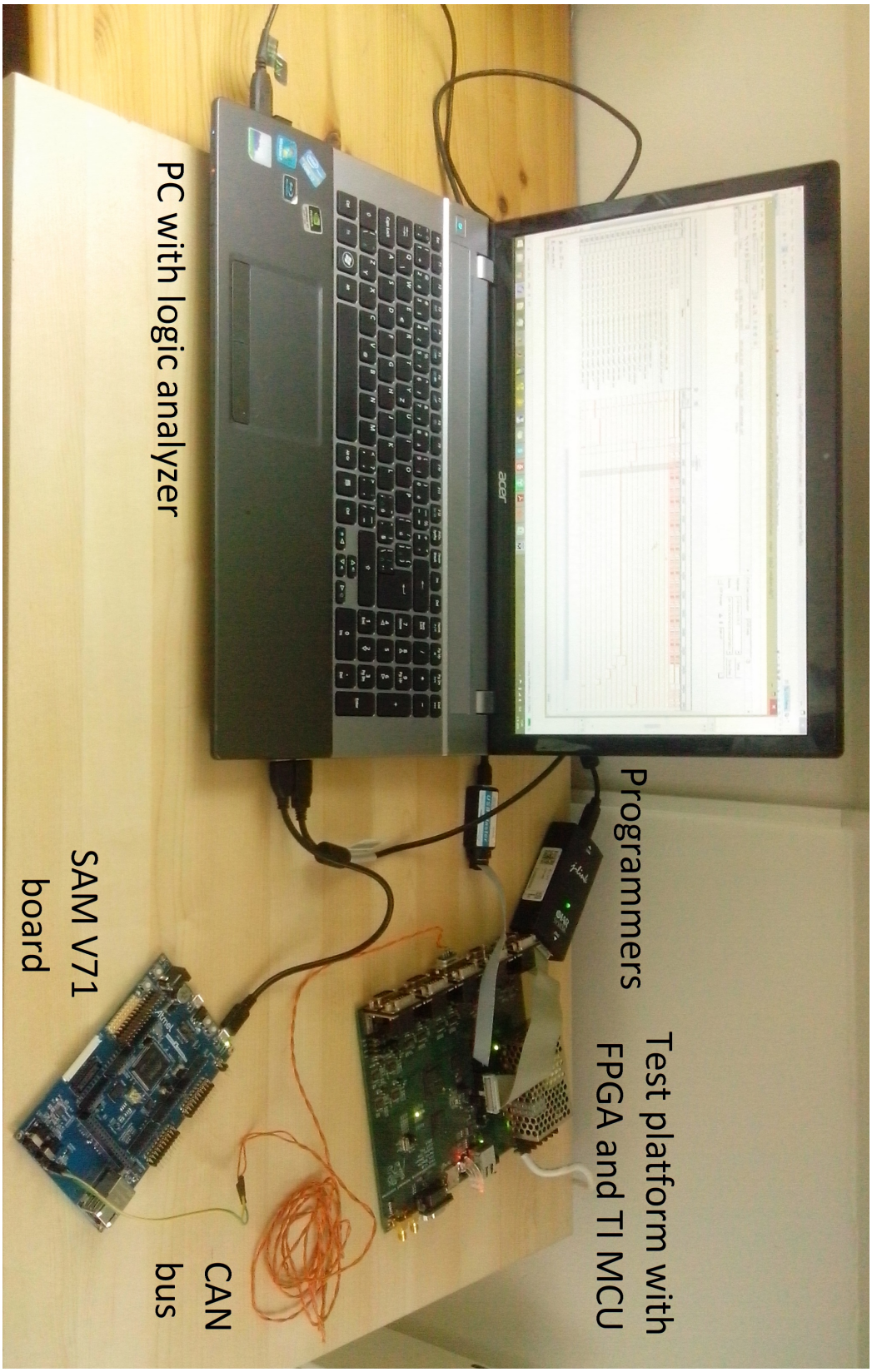

Figure 18: Test setup

7. Known issues and future work

CAN FD IP Core has many features which are not required by CAN FD specification. However there is still room for improvement. Following list names few of the possible improvements:

- Implement automatic baud rate detection, either as part of CAN Core or separate module. Once the unit would be turned on, it would measure bus timing (e.g. over several frames) and provide the results, or even set the bus timing registers.
- Implement additional state machine which would decide about usage of Data bit rate based on relative error rate of two bit rates.
- Optimize the "tranBuffer" and TXT_Buffer to be synthesized into RAM elements, not to LUTs.
- Actual implementation stores Transmitted frame in 3 following pipeline stages: Memory registers, TXT Buffers and TranBuffer. This requires up to 3*640 memory bits! It might be beneficial to optimize the architecture in such a manner that one of the stages will be ommited! E.g user would be directly writing into TXT_Buffer and not to the registers.
- Add hard synchronisation and resynchronisation part into Prescaler unit test
- Implement event logger feature test
- Implement error detection feature test

Appendix A - Driving bus signals

Index	Width	Name	Destination unit	Signal description
0-5	6	drv_tq_nbt	Prescaler	Time quantum length, Nominal bit time
6-11	6	drv_tq_dbt	Prescaler	Time quantum length, Nominal bit time
17-12	8	drv_prs_nbt	Prescaler	Propagation segment length , Nominal bit time
18-23	6	drv_ph1_nbt	Prescaler	Phase 1 segment length, Nominal bit time
24-29	6	drv_ph2_nbt	Prescaler	Phase 2 segment length, Nominal bit time
30-33	4	drv_prs_dbt	Prescaler	Propagation segment length , Data bit time
34-37	4	drv_ph1_dbt	Prescaler	Phase 1 segment length, Data bit time
38-41	4	drv_ph2_dbt	Prescaler	Phase 2 segment length, Data bit time
42-45	4	drv_sjw_nbt	Prescaler	Synchronisation jump width, Nominal bit time
46-49	4	drv_sjw_dbt	Prescaler	Synchronisation jump width, Data bit time
50-60	11	reserved	-	-
61-80	20	reserved	-	-
81-109	29	drv_filter_A_mask	Message filter	Mask for filter A
110-113	4	drv_filter_A_ctrl	Message filter	Allowed frames for filter A
114-142	29	drv_filter_A_bits	Message filter	Bits to compare for filter A
171-143	29	drv_filter_B_mask	Message filter	Mask for filter B
175-172	4	drv_filter_B_ctrl	Message filter	Allowed frames for filter B
204-176	29	drv_filter_B_bits	Message filter	Bits to compare for filter B
205-233	29	drv_filter_C_mask	Message filter	Mask for filter C
234-237	4	drv_filter_C_ctrl	Message filter	Allowed frames for filter C
238-266	29	drv_filter_C_bits	Message filter	Bits to compare for filter C
267-270	4	drv_filter_ran_ctrl	Message filter	Allowed frames for range filter
271-299	29	drv_filter_ran_lo_th	Message filter	Low range treshold for range filter
300-328	29	drv_filter_ran_hi_th	Message filter	High range treshold for range filter
329	1	drv_filter_ena	Message filter	Enable applying message filters.
330-349	29	reserved	-	
350	1	drv_erase_rx	RX Buffer	Erase recieved buffer
351	1	reserved	-	
352	1	drv_read_start	RX Buffer	Move to next word in recieve buffer
353	1	drv_clr_ovr	RX Buffer	Clear Overrun flag
351-355	3	reserved	-	
356	1	drv_erase_txt1	TXT Buffer 1	Erase message in TXT 1 buffer
357	1	drv_store_txt1	TXT Buffer 1	Store message in registers into TXT 1 buffer

Index	Width	Name	Destination unit	Signal description
358	1	drv_erase_txt2	TXT Buffer 2	Erase message in TXT 2 buffer
359	1	drv_store_txt2	TXT Buffer 2	Store message in registers into TXT2 buffer
360	1	reserved	-	
361	1	drv_allow_txt1	TXT Arbitrator	Allow sending messages from TXT1 buffer
362	1	drv_allow_txt2	TXT Arbitrator	Allow sending messages from TXT2 buffer
363-365	3	reserved	-	
366	1	drv_write_tx	TX Buffer	Signal not used
367	1	drv_write_rx	TX Buffer	Signal not used
368-371	4	reserved	-	
372	1	drv_sam	Bus Synchron.	Tripple sampling for slow speeds
373-375	4	reserved	-	
376	1	drv_bus_err_int_ena	Interrupt manager	Enable Bus error interrupt
377	1	drv_arb_lst_int_ena	Interrupt manager	Enable Arbitration lost interrupt
378	1	drv_err_pas_int_ena	Interrupt manager	Enable Fault confinement state changed interrupt
379	1	drv_wake_int_ena	Interrupt manager	Signal not used
380	1	drv_dov_int_ena	Interrupt manager	Enable Data overrun interrupt
381	1	drv_err_war_int_ena	Interrupt manager	Enable Error warning limit reached interrupt
382	1	drv_tx_int_ena	Interrupt manager	Enable sucessfull transcieve interrupt
383	1	drv_rx_int_ena	Interrupt manager	Enable logging finished interrupt
384	1	drv_log_fin_int_ena	Interrupt manager	Enable sucessfull recieve interrupt
385	1	drv_brs_int_ena	Interrupt manager	Enable bit rate shift interrutpt
386	1	drv_rx_full_int_ena	Interrupt manager	Enable interrupt when recieve buffer is full
387	1	drv_int_vect_erase	Interrupt manager	Command to erase interrupt vector
388-399	13	reserved	-	-
400-407	8	drv_ewl	Fault confinement	Error warning limit (by standard 96)
408-415	8	drv_erp	Fault confinement	Error passive treshold (by standard 128)
424-416	8	drv_ctr_val	Fault confinement	Value for presetting error counter
428-425	8	drv_ctr_sel	Fault confinement	Control signals, which counters to preset
459-429	31	reserved	-	-
460	1	drv_CAN_fd_ena	Protocol control	Enable recieve of CAN FD frames
461	1	drv_rtr_pref	Protocol control	RTR preffered behaviour
462-464	3	reserved	-	-
465	1	drv_retr_lim_ena	Protocol control	Retransmittion limit of errornous frames is enabled
466-469	4	drv_retr_th	Protocol control	Retransmittion treshold
470	1	drv_bus_mon_ena	Protocol control	Bus monitoring mode
471	1	drv_self_test_ena	Protocol control	Self Test mode
472	1	drv_abort_tran	Protocol control	Immediately abort actual transmittion
473	1	drv_set_rx_ctr	CAN Core	Preset sucessfully recieved messages counter
474	1	drv_set_tx_ctr	CAN Core	Preset sucessfully transcieved messages counter
475-506	32	drv_set_ctr_val	CAN Core	Value for presetting RX and TX counter
507	1	drv_ack_forb	Protocol control	Acknowledge sending is forbidden
508	1	drv_int_loopback_ena	CAN Core	Internal loopback is enabled
509-519	11	reserved	-	-
520-551	32	drv_trig_config_data	Event logger	Signal is not used

Index	Width	Name	Destination unit	Signal description
552	1	drv_trig_sof	Event logger	Trigger on Start of frame
553	1	drv_trig_arb_lost	Event logger	Trigger on Arbitration lost
554	1	drv_trig_rec_valid	Event logger	Trigger on sucesfull recieve
555	1	drv_trig_tran_valid	Event logger	Trigger on sucesfull transcieve
556	1	drv_trig_ovl	Event logger	Trigger on overload frame transcieved
557	1	drv_trig_error	Event logger	Trigger on error appeared
558	1	drv_trig_brs	Event logger	Trigger on bit rate shifted
559	1	drv_trig_user_write	Event logger	Trigger by logic 1 in this signal
560	1	drv_trig_arb_start	Event logger	Trigger on Arbitration field start
561	1	drv_trig_contr_start	Event logger	Trigger on Control field start
562	1	drv_trig_data_start	Event logger	Trigger on Data field start
563	1	drv_trig_crc_start	Event logger	Trigger on CRC field start
564	1	drv_trig_ack_rec	Event logger	Trigger on acknowledge recieved
565	1	drv_trig_ack_n_rec	Event logger	Trigger on acknowledge was not recieved
566	1	drv_trig_ewl_reached	Event logger	Trigger on error warning limit was reached
567	1	drv_trig_erp_changed	Event logger	Trigger on error passive state changed
568	1	drv_trig_tran_start	Event logger	Trigger on transmittion started
569	1	drv_trig_rec_start	Event logger	Trigger on reception started
570-579	10	reserved	-	-
580	1	drv_cap_sof	Event logger	Capture Start of Frame
581	1	drv_cap_arb_lost	Event logger	Capture Arbitration lost
582	1	drv_cap_rec_valid	Event logger	Capture that message was recieved valid
583	1	drv_cap_tran_valid	Event logger	Capture that message was transcieved valid
584	1	drv_cap_ovl	Event logger	Capture when overload frame is transmitted
585	1	crv_cap_error	Event logger	Capture when error appears
586	1	drv_cap_brs	Event logger	Capture when bit rate is shifted
587	1	drv_cap_arb_start	Event logger	Capture when Arbitration field starts
588	1	drv_cap_contr_start	Event logger	Capture when Control field starts
589	1	drv_cap_data_start	Event logger	Capture when Data field starts
590	1	drv_cap_crc_start	Event logger	Capture when CRC field starts
591	1	drv_cap_ack_rec	Event logger	Capture when Acknowledge was recieved
592	1	drv_cap_ack_n_rec	Event logger	Capture when Acknowledge was not recieved
593	1	drv_cap_ewl_reached	Event logger	Capture when Error warning limit was reached
594	1	drv_cap_erp_changed	Event logger	Capture when Fault confinement state has changed
595	1	drv_cap_tran_start	Event logger	Capture when Transmittion starts
596	1	drv_sap_rec_start	Event logger	Capture when reception starts
597	1	drv_cap_sync_edge	Event logger	Capture that Synchronisation edge appeared
598	1	drv_cap_stuffed	Event logger	Capture that stuff bit was inserted
599	1	drv_cap_destuffed	Event logger	Capture that bit was destuffed from stream
600	1	drv_cap_ovr	Event logger	Capture that data overrun appeared
601-609	9	reserved	-	-
610	1	drv_log_cmd_str	Event logger	Command to start capturing
611	1	drv_log_cmd_abt	Event logger	Command to abort capturing
612	1	drv_log_cmd_up	Event logger	Command to move read pointer up
613	1	drv_log_cmd_down	Event logger	Command to move read pointer down

Appendix B - Status bus signals

Index	Width	Name	Signal description
0-1	2	stat_OP_State	Operation state
2-5	4	stat_PC_State	Protocol Control state
6	1	stat_arb_lost	Arbitration was lost
7	1	stat_set_trans	Unit is set as transciever from next clock
8	1	stat_set_rec	Unit is set as reciever from next clock
9	1	stat_is_idle	Unit is idle
10-11	2	stat_sp_control	Sample point control
12	1	stat_ssp_reset	Secondary sample point reset
13	1	stat_trv_delay_calib	Transciever delay calibration enabled
14-15	2	stat_sync_control	Synchronisation control
16	1	stat_data_tx	Transcieved data
17	1	stat_data_rx	Recieved data
18	1	stat_bs_enable	Bit Stuffing enable
19	1	stat_fixed_stuff	Fixed stuffing method is applied
20	1	stat_data_halt	Bit was stuffed, transmitting should be halted
21-23	3	stat_bs_length	Bit stuffing length
24	1	stat_stuff_error	Stuff Error appeared
25	1	stat_destuffed	Bit is destufffed, shouldntbe recorded by Protocol control
26	1	stat_bds_ena	Bit destuffing is enabled
27	1	stat_stuff_error_ena	Bit Stuffing error detection enabled
28	1	stat_fixed_destuff	Fixed destuffing method should be used
29-31	3	stat_bds_length	Bit destuffing length
32-60	29	stat_tran_ident	Transcieved identifier
61-64	4	stat_tran_dlc	Transcieved dlc
65	1	stat_tran_is_rtr	Transcieved frame is rtr
66	1	stat_tran_frame_type	Transcieved frame type (normal or FD frame)
67	1	stat_tran_ident_type	Transcieved identifier type (basic or extended)
68	1	stat_tran_data_ack	Acknowledge for TXT buffers, TX data are stored in internal buffer
69	1	stat_tran_brs	Transcieved message should shift bitrate
70	1	stat_frame_store	Command to store input frame for transcieve

Index	Width	Name	Signal description
71-79	9	stat_tx_counter	TX error counter
80	1	reserved	-
81-89	9	stat_rx_counter	RX error counter
90-98	9	stat_error_counter_norm	Error counter for errors appeared in nominal bit rate
99-107	9	stat_error_counter_fd	Error counter for errors appeared in data bit rate
108-109	2	stat_error_state	Fault confinement state
110	1	stat_form_error	Form error appeared
111	1	stat_crc_error	CRC error appeared
112	1	stat_ack_error	Acknowledge error appeared
113	1	stat_unknown_state_error	Protocol control is in undefined state
114	1	stat_bit_stuff_error	Bit or Stuff error appeared
115	1	stat_first_bit_after	Signal not used
116	1	stat_rec_valid	Message was recieved valid
117	1	stat_tran_valid	Message was transcieved valid
118	1	stat_const7	Signal not used
119	1	stat_const14	Signal not used
120	1	stat_transm_error	Signal not used
121-149	29	stat_rec_ident_type	Recieved identifier
150-153	4	stat_rec_dlc	Recieved data length code
154	1	stat_rec_is_rtr	Recieved frame is rtr
155	1	stat_rec_frame_type	Recieved frame type (normal or FD)
156	1	stat_rec_ident_type	Recieved identifier type (basic or extended)
157	1	stat_rec_brs	Recieved frame with bit rate shift
158-178	21	stat_rec_crc	Recieved CRC value
179	1	stat_rec_esi	Recieverd Error state indicator
180	1	stat_crc_ena	CRC calculation is enabled
181	1	stat_tran_trig	Transcieve trigger (in sync segment)
182	1	stat_rec_trig	Recieve trigger (in sample point)
183-187	5	stat_alc	Arbitration lost capture
188-219	32	stat_rx_ctr	Sucesfully recieved message counter
220-251	32	stat_tx_ctr	Sucesfully transcieved message counter
252	1	stat_erp_changed	Error passive state has changed
253	1	stat_ewl_reached	Error warning limit was reached
254	1	stat_err_valid	Error is valid
255	1	stat_ack_recieved_out	Acknowledge was recieved
256	1	stat_bit_error_valid	Bit Error appeared

Bibliography

[1] CAN with Flexible Data-Rate Specification v 1.0, Robert Bosch GmbH, April 2012

[2] CAN 2.0 Protocol standard, Robert Bosch GmbH, Rev 3.0

[3] Controller Area Network - Basics, protocols, chips and applications, Prof. Dr.-Ing. K. Etschberger, 2001

[4] CRC for CAN with flexible data rate (CAN FD) - Whitepaper

[5] Software for Test Platform, DataSheet, Ille Ondrej, Czech Technical University, July 2015

[6] Implementation of unconventional CAN controller in VHDL, Diploma Thesis, Dušan Hamza, Czech technical university, 2013

[7] Robustness of a CAN FD Bus System – About Oscillator Tolerance and Edge Deviations, Dr. Arthur Mutter, Robert Bosch GmbH, 2013

[8] SJA1000 Standalone CAN Controller,Philips Semiconductors, January 2000

[9] TJA1041 High speed CAN transceiver Rev. 06, December 2007, NXP Semiconductors

[10] ModelSim Advanced verification and debugging SE Command Reference, Mentor Graphics, v 6.0 November 2004

[11] VHDL guidlines for synthesis, Siemens semiconductor group

[12] FPGA prakticky - Realizace číslicových systémů pro hradlová pole, Jakub Šťastný, BEN Technická literatura 2011

[13] Číslicové systémy a jazyk VHDL , Jiří Pinker, Martin Poupa, BEN Technická literatura 2006

[14] Understanding Metastability in FPGAs, ALTERA

[15] SAM V71, SMART ARM-based Flash MCU, Preeliminary datasheet, Atmel, February 2015

[16] VHDL Guides , Dr. Jayram, Moorkanikara Nageswaran, Department of Computer Science University of California, http://www.ics.uci.edu/~jmoorkan/vhdlref/

[17] Avalon Interface Specifications,ALTERA March 2015

[18] Methods for Testing the FlexRay Start-up Mechanism, Diploma thesis, Martin Paták ,Czech Technical University in Prague, 2012

[19] TimeQuest Timing Analyzer - Quick Start Tutorial, ALTERA, December 2009

[20] Implementing Inferred RAM, http://quartushelp.altera.com/14.0/mergedProjects/hdl/vhdl/vhdl_pro_ram_inferred.htm