CTU CAN FD IP CORE

System Architecture

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

December 15, 2023

Document version	Corresponding Datasheet version (release)	Date	Change description
0.1	2.2	26-09-2019	Initial version - separated stand-alone architecture document from Datasheet document.
0.2	2.2	29-09-2019	TX Arbitrator loads identifier as part of TXT buffer validation.
0.3	2.2	07-10-2019	Update interfaces
0.4	2.2	21-10-2019	Clarify TXT Buffer will go to TX Failed in Bus-off.
0.5	2.2	07-11-2019	Replace SSP shift register by SSP generator.
0.6	2.2	13-12-2019	Add "Error delimiter too long" state to Protocol control FSM. Clear non-actual TODOs.
0.7	2.2	30-04-2020	Add note about implementation types. Remove form error on EDL/R0. Update Protocol control FSM to handle protocol exception.
0.8	2.2.4	18-05-2020	Correct Expected segment lenght preload values for negative resynchronisation.
0.9	2.2.5	6-10-2020	Update Protocol control FSM diagram.
0.10	2.3.0	6-02-2021	Add notes on clock gating.
0.11	2.3.3	26-04-2021	Add description of memory testability.
0.12	2.3.4	10-05-2021	Keep NBTM counter enabled always even in data bit rate. Fixes bug with improper PH2 lenght if error is detected during data bit rate with BRP=1
0.13	2.3.5 and higher	16-05-2021	Add ***res_n_out*** synchronized reset output.
0.14	2.4 and higher	22-12-2021	Clarify implications of connecting core to 8/16/32 bit buses.
0.15	2.4.1	10-4-2022	Add Parity Check use-case in TXT Buffer. Add ***sup_parity*** generic.
0.16	2.4.2	27-6-2022	Add ***reset_buffer_rams*** and ***active_timestamp_bits*** generic. Remove interfaces of each sub-block.
0.17	2.4.3	18-2-2023	Remove drv_bus and stat_bus.
0.18	2.5	9-12-2023	Move to new release of CTU CAN FD. Bump document version accordingly.

Contents

Format

Throughout this document following notations are kept:

- Common text is written with this font.
- Memory registers are always described with capital letters e.g. REGISTER or REGISTER [BIT_FIELD] to represent register or bit field within a register.
- Signal names and generic names are written by bold lower-case cursive (e.g. ***can_rx***)
- Explicit terms from ISO11898-1 2015 are marked via red color (e.g. SOF bit). Definition of these terms can be found in [1].
- Open issues and TODOs are written in blue font like so TODO: not yet implemented.

1. General Information

1.1 Introduction

This document describes architecture of CTU CAN FD IP Core. It describes interfaces within the core and function of each module. This document is not written in specification format (device shall behave like so), rather in description format (device behaves like so). Nevertheless, this document alogn with CTU CAN FD Datasheet ([2]) serves as reference on how shall CTU CAN FD function and it is supposed to be used as verification reference on how shall the device behave.

1.2 Development tools

To develop CTU CAN FD following tools are used:

- GHDL for RTL simulations.
- Quartus Prime and Xilinx Vivado for Synthesis to Intel and Xilinx FPGAs, Timing analysis and design size benchmarks.
- VUnit for simulation wrappers.
- Kactus2 for definition of register map in IP-XACT format.
- LyX v.2.3.0 to write documentation.
- GitLab of CTU FEE to host source code GIT repository.
- Wavedrom for Timing Diagrams.
- Python for scripting.

1.3 Design automation

Part of CTU CAN FD Core is auto-generated. Register map is implemented in Kactus 2 in IP-XACT format ("spec/CTU/ip/CAN_FD_IP_Core/2.1/CAN_FD_IP_Core.2.1.xml"). The design in IP-XACT format is unified specification of user-interface. Following resources are generated from IP-XACT specification:

- VHDL packages with address, bit-fields and reset values definitions ("src/lib/can_fd_frame_format.vhd", "src/lib/can_fd_register_map.vhd").

- C header file with address map definitions and register descriptions ("driver/ctu_can_fd_regs.h", "driver/ctu_can_fd_frame.h").
- Lyx documentation of register map. Reffer to [2].
- RTL Code of Control Registers module ("src/memory_registers/generated/*).
- Documentation of RTL module interfaces ("doc/core/entity_docs").

To generate these design materials CTU CAN FD IP Core uses IP-XACT register map generator which is accessible at regmap_gen. Register map generator is linked as sub-module of CTU CAN FD repository. Clone all the submodules recursively before using register map generator. All of the generated files are considered as don't touch. Part of this document is also auto-generated. Each section which describes list of Generics and Signals of a module is generated from VHDL RTL code.

1.3.1 Register map generation

When CTU CAN FD GIT repository is clonned, register map can be generated by following script:

```
cd scripts
./update_reg_map
```

1.3.2 Documentation generation

Documentation can be exported from VHDL RTL codes by following script:

```
cd scripts
python gen_lyx_tables.py --configPath vhdl_lyx_interface_cfg.yml
```

"vhdl_lyx_interface_cfg.yml" is YAML configuration file which describes source RTL codes and destination LyX files.

1.3.3 Xilinx Vivado component

CTU CAN FD contains Xilinx Vivado component ("src/component.xml") for integration of CTU CAN FD to Xilinx based FPGAs. Xilinx Vivado component is generated by following script:

```
cd scripts
python gen_vivado_component.py
```

1.4 General coding guidlines

RTL code within CTU CAN FD has following coding rules:

- Underscore is always used to separate words within signal/entity/process/variable/port/generic names (e.g. tx_hw_cmd, can_core).
- Constants are written by capital letters with "C_" prefix (e.g. C_SUSPEND_DURATION).
- Generics are written by capital letters with "G_" prefix (e.g. G_RX_BUFF_SIZE). This rule has an exception on top level interface and wrappers of CTU CAN FD (can_top_level, can_top_ahb).

- Signals are always commented on line before the signal. This must be especially true for port signals. This allows to extract documentation of VHDL entities from RTL code.
- Sections of signals can be defined by surrounding section name by whole line of "-" characters.
- All RTL codes are indented with 4 spaces.
- Line length shall be limited to 80 characters.
- Instance names are suffixed with "_inst", process names are suffixed with "_proc", cover point names are suffixed with "_cov", assertion names are suffixed with "_asrt". DFF names can be suffixed by "_d/_q" depending on whether it is DFF input/output.

1.5 Source code access

CTU CAN FD IP Core source code is available in CTU FEE GitLab repository at:

https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core

1.6 ISO11898-1 2015 compliance

CTU CAN FD is compliant with [1]. With regards to this document, CTU CAN FD supports all implementation options (Classical CAN, CAN FD Tolerant, CAN FD enabled). Compliance to each of these options can be configured via a register (run-time configurable). Reffer to [2] for description of CTU CAN FD configuration.

Support of optional features from [1] is described in Table 1.1 and Table 1.2.

Table 1.1: ISO11989-1 optional features (1)

Feature Name	Status	Notes
FD Frame format	Supported	
Disabling of frame formats	Supported	Reception of CAN FD frames can be disabled by setting MODE[FDE] = '0'.
Limited LLC frames	Not Supported	Only full size (64 byte) frames are supported.
No transmission of frames including padding bytes	Not Supported	No padding is inserted since full sized frames are supported.
LLC Abort Interface	Supported	Issuing Set abort command to TXT buffer which is used for transmission is equal to issuing LData.Abort_Request / LRemote.Abort_Request primitive.
ESI and BRS values	Supported	BRS value can be specified for each transmitted CAN frame. ESI value can't be specified for transmitted CAN frames, it is always derived from current Fault confinement state of CTU CAN FD. ESI value can be read for each received frame.
Method to provide MAC data consistency	Partially Supported	CTU CAN FD implements TXT Buffer RAMs which stores whole CAN frame for transmission before the transmission is started. This corresponds to: "The MAC sub-layer shall store the whole message to be transmitted in a temporary buffer that is filled before the transmission is started." Additionally, CTU CAN FD implements parity protection on each word of TXT Buffer and RX Buffer if ***sup_parity***=true.
Time and time triggering	Partially Supported	Time triggerred transmission is available in TX Arbitrator module. CTU CAN FD does not support time base by itself, it is left up to integrator to provide Time base via ***timestamp*** input. The reason for this, is to share single Time base between multiple instances of CTU CAN FD. ***timestamp*** input is readable from CTU CAN FD. No event generation is provided from ***timestamp*** input.
Time stamping	Supported	Timestamping of RX frames is supported in SOF or EOF bit. Time Base counter must be provided by integrator and must be connected to ***timestamp*** input.
Bus Monitoring mode	Supported	Supported via MODE[LOM].
Handle	Supported	Handle corresponds to TXT Buffer.
Restricted operation	Supported	Supported via MODE[ROM].
Separate prescalers for Nominal and Data Bit Rate	Supported	Prescalers are separate in BTR[BRP] and BTR_FD[BRP_FD] registers.

Table 1.2: ISO11989-1 optional features (1)

Feature Name	Status	Notes
Disabling of automatic retransmission	Supported	Supported via SETTINGS[RTRLE] and SETTINGS[RTRTH] registers.
Maximum number of retransmissions	Supported	
Disabling of protocol exception event on res bit detected recessive	Supported	Protocol exception is configurable via SETTINGS[PEX] register.
PCS_Status	Supported	CTU CAN FD supports both nominal and data bit rate.
Edge filtering during the bus integration state	Not Supported	
Time resolution for SSP placement	Not Supported	Secondary sample point position is always given in minimum time quanta regardless of bit rate prescaler seettings.
FD_T/R message	Supported	

2. Interfaces

2.1 Memory Bus

CTU CAN FD is a slave device accessible via one of three memory buses:

- RAM-like interface,
- APB
- AHB.

Each interface can be used via dedicated wrapper. SW shall not access CTU CAN FD sooner than two clock cycles after external reset was released (due to reset synchronisation) (see Table 3.1). If CTU CAN FD is accessed earlier, writes accesses have no effect and read accesses return zeroes. If external reset is executed via SW driver (e.g. at driver load time), it is recomended to add corresponding delay before driver executes any access to the device (e.g. via usleep, nanosleep, dummy NOPs, or similar mechanism).

2.1.1 RAM-like interface

Wrapper can_top_level.vhd

RAM-like interface is default interface of CTU CAN FD with signals shown in Table 2.1. A typical read/write transcations on RAM-like interface are shown in Figure 2.1. Note that RAM-like interface does not contain any Ready/ACK signal. CTU CAN FD is always able to process written data in one clock cycle (write access) and return read data in the next clock cycle (read access). Accesses on RAM-like interface shall be 4 byte aligned (lower 2 bits of address shall be equal to 0). If access is not 4 byte aligned, lower 2 bits of address are ignored. Therefore, single access spaning more than 1 32 bit memory word is not possible. Each byte is separately writable and readable via byte enable (***sbe***), therefore 8-bit and 16-bit accesses are supported. If ***sbe*** signal is zero, data on corresponding byte are not written during write access, and zeroes are returned during read access. CTU CAN FD is little endian oriented (LSB = Lowest Adress -> ***sbe(0)*** = Byte 0 = ***data_in/out (7:0)***; ***sbe (3)*** = Byte 3 = ***data_in/out(31:24)***).

RAM-like interface supports burst read from RX Buffer (see 3.15). In such case, ***address*** input must be equal to RX_DATA register address during whole read operation ("stationary"/"frozen" burst). During such read, each word must be read by 32-bit access (***sbe***="1111"). This means that read from RX Buffer is always executed by 32-bit word regardless of ***sbe*** value. Such a situation is shown in Figure 2.2.

Table 2.1: RAM-like interface

Signal Name	Direction	Width	Description
data_in	in	32	Write Data
address	in	16	Address
scs	in	1	Chip Select
srd	in	1	Read indication
swr	in	1	Write indication
sbe	in	4	Byte enable (applicable for both reads and writes)
data_out	out	32	Read data

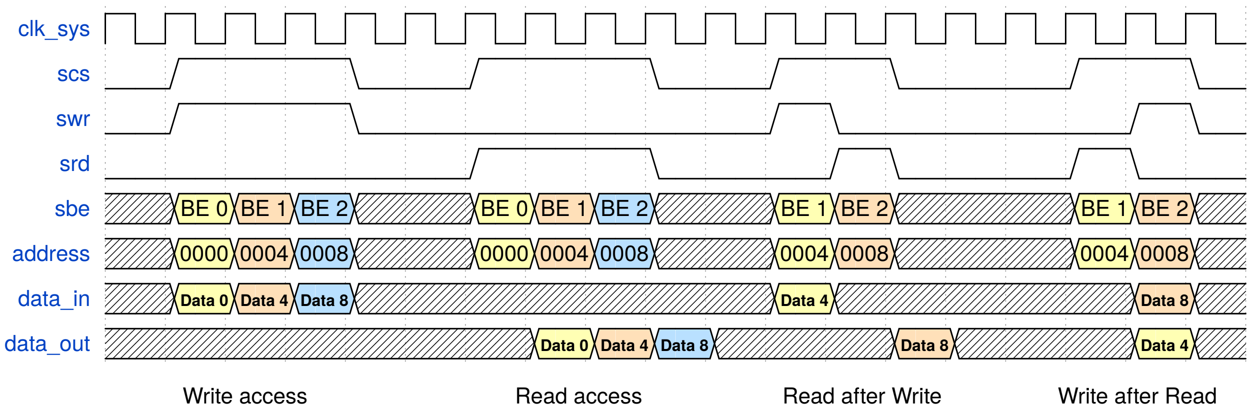

Figure 2.1: RAM-like interface

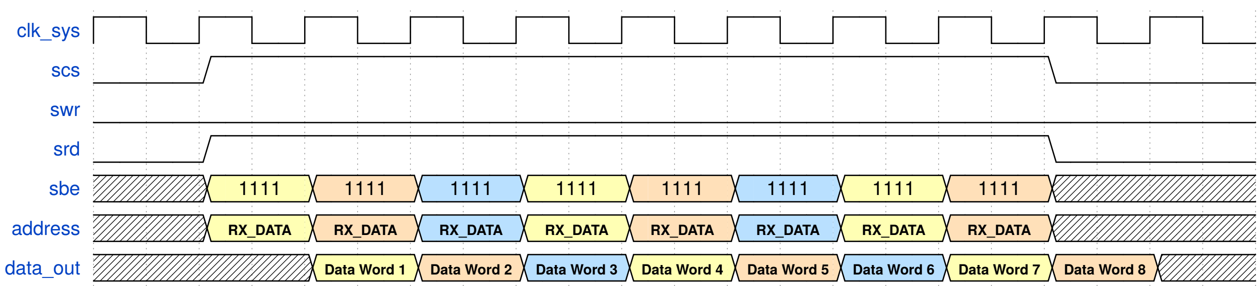

Figure 2.2: RX Buffer burst read

RAM-like interface is Avalon compatible (according to [3]) and mapping of RAM like signals to Avalon Memory-mapped slave signals is shown in Table 2.2. When connected to Avalon MM master, write access to reserved address has no effect and read access returns all zeroes instead of responding with DECODEERROR response. ***response*** signal shall be connected to "00", ***writeresponsevalid*** and ***readdatavalid*** shall be connected to '1'.

2.1.2 APB

Wrapper can_top_apb.vhd

APB Wrapper is compatible with [4]. Signals of CTU CAN FD on APB interface are shown in Table 2.3. Note that every access on APB Interface lasts two clock cycles, no bursts can be executed by nature of this interface. CTU CAN FD

Table 2.2: RAM-like to Avalon mapping

RAM-like signal name	Avalon signal name	Description
data_in	***write_data***	Data written to Avalon MM slave.
address	***address***	Address for read/write of Avalon MM slave.
scs	-	Shall correspond to chip select of slave if more than 1 slave is connected to given bus. If single slave is connected, shall be connected to 1.
srd	***read***	Read indication
swr	***write***	Write indication
sbe	***byteenable***	Byte enable, used for both read and write transfers.
data_out	***readdata***	Data read from Avalon MM slave.

does not stall transfers on APB interface via ***s_apb_pready***, it keeps ***s_apb_pready*** always high. CTU CAN FD does not return error via ***s_apb_pslverr*** on any access. If SW executes access to an invalid location within CTU CAN FD, it is simply ignored. This allows dumping whole CTU CAN FD memory space without memory access errors. Accesses on APB Interface shall be 4 byte aligned. If access is not 4 byte aligned, lowest 2 bits of address are ignored. 8/16 bit write accesses are supported via write strobe signal (***s_apb_pstrb***). Basic accesses on APB are shown in Figure 2.3.

Table 2.3: APB interface

Signal Name	Direction	Width	Description
s_apb_paddr	in	32	Address
s_apb_penable	in	1	Enable. Indicates second cycle of access.
s_apb_prot	in	3	Protection type. Ignored by CTU CAN FD. All access types are treated equally by CTU CAN FD.
s_apb_prdata	out	32	Read data.
s_apb_pready	out	1	Ready. Always asserted.
s_apb_psel	in	1	Slave select.
s_apb_pslverr	out	1	Access error. CTU CAN FD always drives this pin low.
s_apb_pstrb	in	4	Write Strobe. During write access, logic 1 indicates according byte will be written. Ignored during read access.
s_apb_pwdata	in	32	Write data.
s_apb_pwrite	in	1	Access direction.

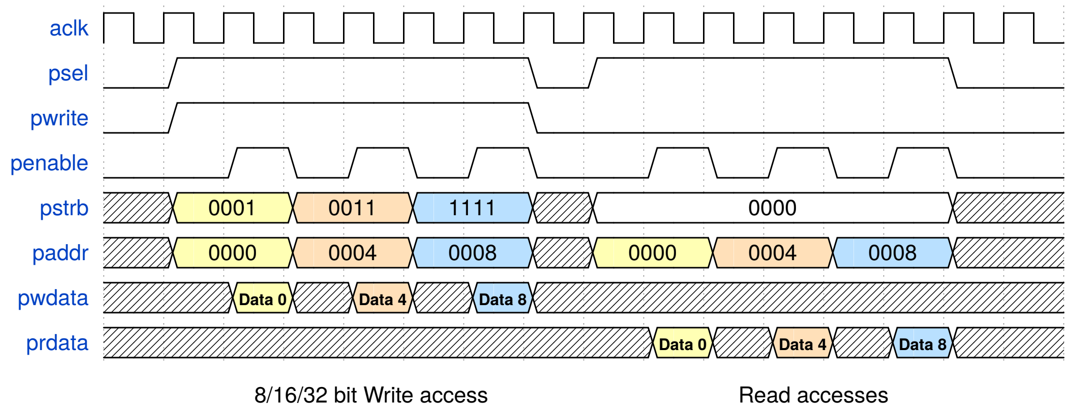

Figure 2.3: APB Interface access

2.1.3 AHB

Wrapper CAN_top_ahb.vhd

AHB Wrapper is compatible with [8]. Signals of CTU CAN FD on AHB interface are shown in Table 2.4. CTU CAN FD accepts all transfer types (Non-sequential, Sequential, Idle, Busy) on AHB bus. CTU CAN FD treats burst accesses equally as regular accesses (no internal caching is done). If read transfer occurs after write transfer (directly one after another), CTU CAN FD inserts one wait cycle into AHB transaction, as is shown in Figure 2.4. CTU CAN FD does not return error via ***hresp*** on any accesses. If SW executes access to an invalid location within CTU CAN FD, it is simply ignored. This allows dumping whole CTU CAN FD memory space without memory access errors. CTU CAN FD does not support unaligned accesses on AHB Bus. Each access shall be aligned to its own size (8-bit access can have arbitrary address, 16 bit access must have address 2-byte aligned, 32-bit access must have address 4-byte aligned). No locked sequences (***hmastlock***) are supported by CTU CAN FD.

Table 2.4: AHB interface

Signal Name	Direction	Width	Description
haddr	in	32	Address
hwdata	in	32	Write Data
hsel	in	1	Write select
hwrite	in	1	Access direction
hsize	in	3	Access size. (8/16/32 bit access sizes are supported).
hburst	in	3	Burst indication, ignored by CTU CAN FD.
hprot	in	3	Protection type, ignored by CTU CAN FD.
htrans	in	2	Transaction type.
hmastlock	in	1	Locked sequence indication.
hready	in	1	Ready indication.
hreadyout	out	1	Ready indication output.
hresp	out	1	Response type.
hrdata	out	32	Read data.

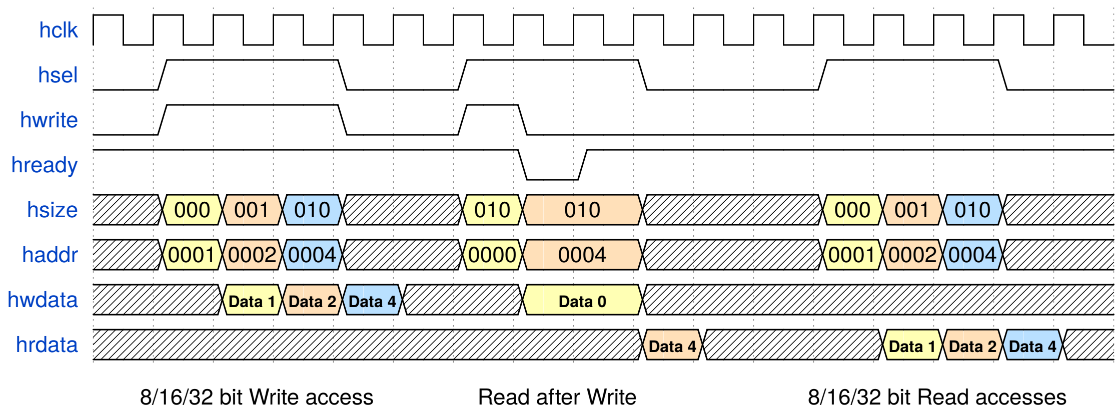

Figure 2.4: AHB Interface access

2.1.4 Limitations on 8/16 bit buses

CTU CAN FD is 32-bit peripheral, however, it is possible to integrate it to systems with 8/16 bit bus thanks to "byte enable" capabilities of each bus interface wrapper. If SW accesses CTU CAN FD via 8/16 bit bus, access to simple 32-bit R/W register can be split into 4/2 consecutive accesses without affecting the functionality. However, due to side-effects on several registers, there are following limitations when accessing CTU CAN FD from 8/16 bit buses:

- CTU CAN FD must be used in RX Buffer manual mode (MODE[RXBAM] = 0). This is necessary since read of single word from RX Buffer can not be done by single read access to RX_DATA register. On 8 bit systems, it will require 4 reads (addresses RX_DATA .. RX_DATA + 0x3), on 16 bit systems it will require 2 reads (addresses RX_DATA and RX_DATA + 0x2). Since each read from RX_DATA register in RX Buffer automated mode (MODE[RXBAM] = 1), will move RX Buffer read pointer, the rest of the memory word would be lost without being read out. Thus it would be impossible to correctly read out received frames. Reading out RX Buffer on 8/16 bit systems thus requires operation in MODE[RXBAM] = 0 and manually moving RX Buffer read pointer by COMMAND[RXRPMV] bit.
- On 8 bit systems, TX_PRIORITY register is only able to change priority of TXT Buffers atomically if number of TXT Buffers is 2. On 16 bit systems, TX_PRIORITY register is only able to change priority of TXT Buffers atomically, if number of TXT Buffers is 2-4. Atomic change of TXT Buffer priorities is required if TXT Buffers are used like a FIFOs by priority rotation (such approach is used by CTU CAN FD Linux driver). Thus, if TXT buffer priorities need to be rotated atomically, following restrictions apply:
  - On 8 bit systems, only 2 TXT Buffers must be used.
  - On 16 bit systems, only up to 4 TXT Buffers must be used.
  - If atomic rotation of priorities is not required, number of TXT Buffers is not restricted.

2.2 CAN Bus

CTU CAN FD interfaces to physical layer transceiver via ***can_rx*** and ***can_tx*** pins. ***can_rx*** input is assumed to be asynchronous to System clock (see 2.4) and it is treated like asynchronous signal. ***can_tx*** output is synchronous to System clock. ***can_tx*** output is glitch-free during operation on CAN bus as long as MODE[LOM] bit is not changed.

2.3 Timestamp

CTU CAN FD interfaces to system level Time base via ***timestamp*** input. ***timestamp*** input is assumed to be synchronous to System clock, and therefore there is no resynchronization on this input. If ***timestamp*** is unused (no Timestamping / Time Triggering capability), it shall be driven to 0xFFFF FFFF FFFF FFFF. If ***timestamp*** is used, it shall be driven by unsigned up-counting counter which measures flow of time within a system to which CTU CAN FD is being integrated. ***timestamp*** does not need to be incremented every clock cycle of System clock, nor there is a constraint on step that it is incremented with, it only needs to be synchronous to System clock. If system level time counter has lower width than 64 bits, integrating system shall connect such counter to lower bits of ***timestamp*** input, and drive unused high bits to zero. Integrating system shall also set ***active_timestamp_bits*** to width of such counter - 1 (e.g. when system has 32 bit timestamp, it shall be connected to ***timestamp[31:0]*** and ***active_timestamp_bits***=31).

2.4 Clock and reset

CTU CAN FD is clocked via single clock input which represents System clock domain. Name of clock signal is different depending on used memory bus wrapper as is shown in Table 2.5. CTU CAN FD has single external reset which is treated as asynchronous reset, and it is internally synchronized by reset synchronizer (see 3.3). Note that AHB bus specifications requires ***hresetn*** to be synchronous to ***hclk***. CTU CAN FD implemenation is more relaxed, and does not require these signals to be synchronous to ***hclk*** (System clock), since it handles reset synchronisation internally. ***res_n_out*** signal output contains synchronized version of ***res_n***/***arstn/hresetn*** input. It can be left unconnected, or it can be used as an indication that reset has been completed and CTU CAN FD can be accessed on its memory bus.

Table 2.5: Clock signal names

Bus type	Clock signal name	Reset signal name
RAM-like	***sys_clk***	***res_n***
APB	***aclk***	***arstn***
AHB	***hclk***	***hresetn***

2.5 Test probe

CTU CAN FD contains ***test_probe*** record output. This signal is used by CTU CAN FD test-bench to peek inside the design of CTU CAN FD. When integrating CTU CAN FD, this output can remain un-connected. Reffer to [8] for description of how to connect test-probe if integrating CTU CAN FD VIP. This signal has no effect on design functionality, and it can remain unconnected in design to which CTU CAN FD is integrated.

2.6 Scan enable

CTU CAN FD is designed to simplify DFT insertion during ASIC design via ***scan_enable*** input. When ***scan_enable*** = 1, CTU CAN FD is in scan mode. In scan mode following is valid:

- All clock gates within CTU CAN FD are un-gated (to make sure that scan chain is always clocked).
- All resets which depend on value of other flip-flops are gated (to avoid reseting part of scan chain during scan operation).

scan_enable input shall be controlled by SoC level DFT controller, and it shall be connected to the same signal which enables scan mode on inserted scan flip-flops. Purpose of scan mode in CTU CAN FD, is to reduce number of violations/warnings during DFT insertion. If CTU CAN FD is used in FPGA (***target_technology*** = 1), ***scan_enable*** shall be tied low. ***scan_enable*** signal shall be driven synchronous to System clock.

2.7 Configuration options

CTU CAN FD is configurable on top level interface via VHDL generics which are explained in Table 2.6.

Table 2.6: CTU CAN FD generic parameters

Name	Type	Default	Range	Description
rx_buf_size	natural	128	32-4096	Size of RX Buffer RAM in 32 bit words. See 3.15.
txt_buffer_count	natural	4	2-8	Number of TXT buffers. See 3.17.
sup_filt_A	boolean	true	true/false	Synthesize filter A. See 3.16.
sup_filt_B	boolean	true	true/false	Synthesize filter B. See 3.16.
sup_filt_C	boolean	true	true/false	Synthesize filter C. See 3.16.
sup_range	boolean	true	true/false	Synthesize range filter. See 3.16.
sup_traffic_counters	boolean	true	true/false	Synthesize traffic counters. See 3.14.8.
target_technology	natural	1	0-1	Target technology (set 0 for ASIC, set 1 for FPGA).
sup_test_registers	boolean	true	true/false	Synthesize test registers.
sup_parity	boolean	false	true/false	Add parity protection to TXT Buffers / RX Buffer.
reset_buffer_rams	boolean	false	true/false	When true, TXT Buffer and RX Buffer RAMs are reset by ***res_n***.
active_timestamp_bits	integer	63	0-63	Number of active timestamp bits minus - 1.

3. System architecture

3.1 Block diagram

Detailed block diagram of CTU CAN FD IP Core is shown in Figure 3.1.

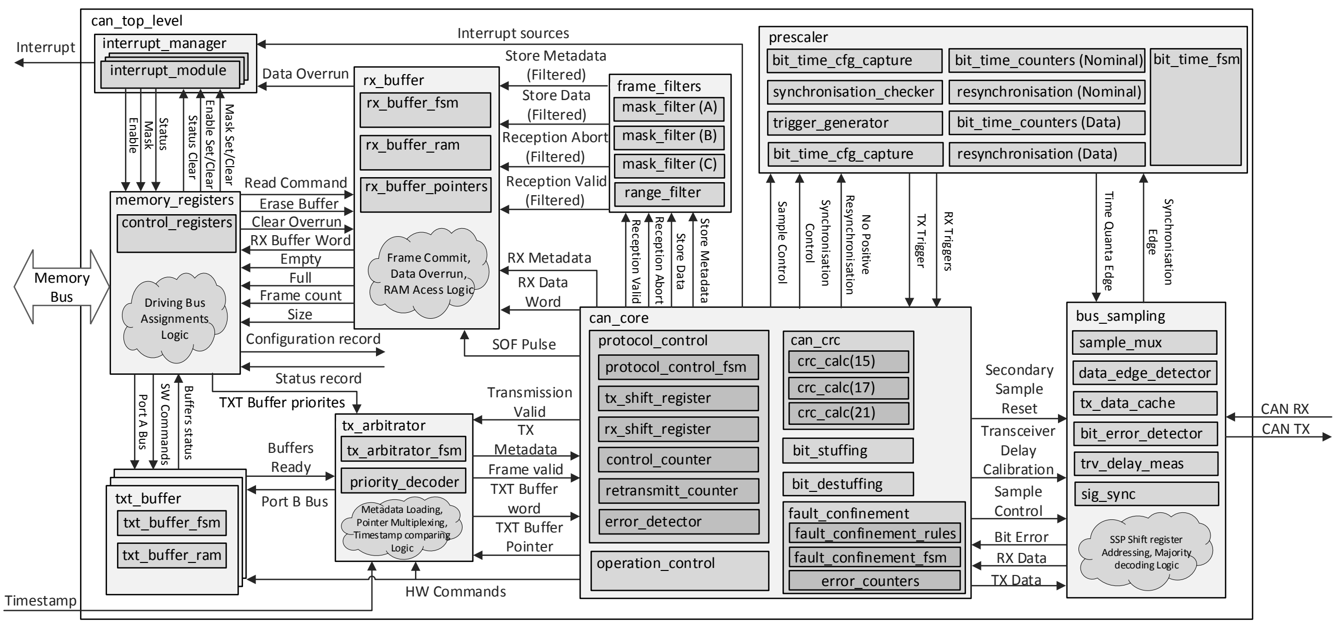

Figure 3.1: CTU CAN FD - Detailed block diagram

3.2 Reset architecture

CTU CAN FD IP Core has two reset sources: External reset and Soft Reset. Both reset sources are described in Table 3.1. Both reset cause assertion of internal System reset which resets whole CTU CAN FD including Memory registers. Reset architecture is shown in Figure 3.2. Note that DFF which pipelines Soft Reset is a DFF without Set and Reset. Reset on this DFF is de-activated on purpose to avoid timing problems between Q output and CLR pin of this DFF. An example of reset sequence by both External as well as Soft reset are shown in Figure 3.3. Note that all DFFs in Figure 3.2 are clocked by System clock.

Table 3.1: Reset description

Reset Name	Asserted by	Reset description
External Reset	RAM like interface: ***res_n*** = 0.	To be used by HW reset structure integrating CTU CAN FD (e.g. POR, System level reset controller). CTU CAN FD shall not be accessed for two System clock periods after External reset was de-asserted (or until ***res_n_out*** = 1). Asserting External reset does not require System clock to be running. De-asserting reset requires System clock to be running.
	AHB interface: ***hresetn*** = 0.	
	APB interface: ***aresetn*** = 0.	
Soft Reset	Writing MODE[RST] = '1'.	To be used by SW for resetting CTU CAN FD. System clock must be running when this reset is asserted (needed for Bus access and pipeline DFF).

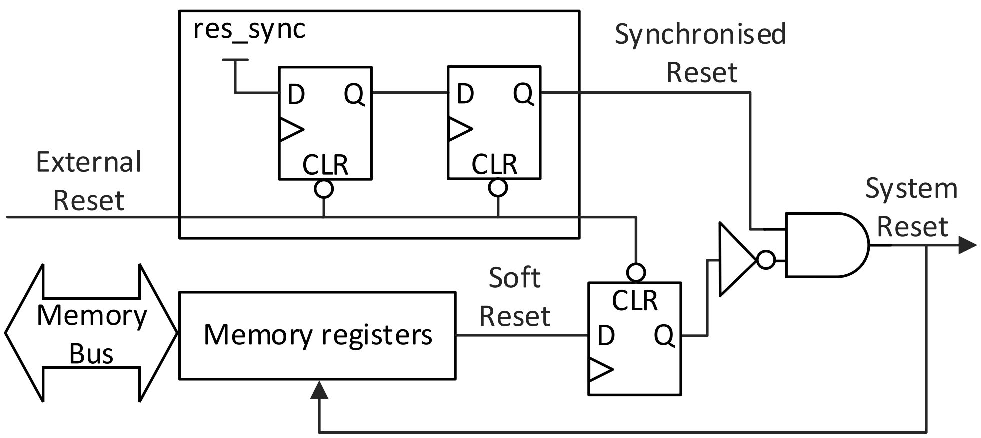

Figure 3.2: Reset structure

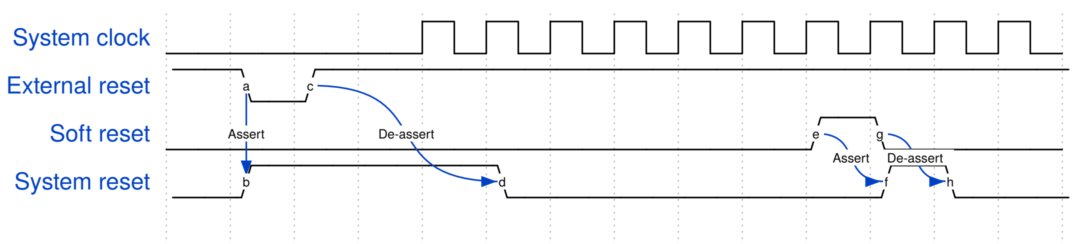

Figure 3.3: Reset operation

3.3 Clock architecture

CTU CAN FD IP Core contains one clock domain, System clock. Each other timing related information (e.g. time quanta) is derived from System clock via clock enable signals. This makes CTU CAN FD fully synchronous design with no clock domain crossing. CTU CAN FD is assumed to be implemented in a single power domain, all parts of CTU CAN FD must be either turned on or off. To reduce dynamic power consumption, majority of registers is written to allow usage of "clock enables" (FPGAs) or inferred "clock gating" (ASIC).

If ***target_technology*** = 0 (ASIC), hand-written clock gating is implemented for Memory registers, RX buffer RAM and TXT Buffer RAMs. If ***target_technology*** = 1 (FPGA), no hand-written clock gating is implemented, clocks for memory registers RX buffer RAMs , and TXT Buffer RAMs are always enabled. There is no functional difference between ASIC/FPGA target technology (even if clocks are always enabled, registers are wrriten only when enabled).

If ***target_technology*** = 0 (ASIC), manually used clock gating cell (clk_gate.vhd) has Latch + AND type. It is recommended to replace clk_gate with with Integrated clock gating cell (e.g. by rewriting internals of clk_gate.vhd by instantiating technological ICG), if such cell is available. If not done, clk_gate.vhd will synthesize into discrete Latch + AND gate. If ***target_technology*** = 1 (FPGA), then clk_gate.vhd does not gate clocks, but only connects input clock to output clock.

If CTU CAN FD is implemented in SoC system, it is recommended to implement configurable clock gating for whole CTU CAN FD peripheral on system level to save power when CTU CAN FD is not clocked. In such situation, CTU CAN FD ignores traffic on CAN Bus and continously transmitts recessive bits to CAN Bus.

3.4 Testability

CTU CAN FD contains following features for manufacturing testability:

1. Memory testability - Allows direct read/write access to TXT Buffer RAMs and RX Buffer RAM. This approach is supported only when Test registers memory region is synthesized (***sup_test_registers*** = true). In general, it is recommended to synthesize Test registers only for ASIC implementations (***target_technology*** = 0), as synthesis for FPGA implementations is usefull only for testing parity protection of RX / TXT Buffer RAMs, since test access bypasses parity encoding mechanism.
2. Scan mode (via ***scan_enable*** input) - In scan mode, all clock gates are enabled, and all reset signals which depend on outputs of combinatorial logic are gated.

3.4.1 Memory testability

Each memory within CTU CAN FD can be tested at production via Test Registers (e.g. executing march pattern test). Any data can be written to any address inside each memory. Memory testability is available only in Test Mode (MODE[TSTM] = 1). If device is not in Test mode, accesses to whole Test registers block are ignored. Memory testability has its own “enable” bit (TSTCTRL[TMENA]), which must be set to enable memory testing via Test registers. An example of memory testing is shown in Table 3.2. Note that this test sequence is only an example. Since Test registers provide independed Read/Write functionality to arbitrary addresses, any known testing approach can be used (any address step, direction or data pattern can be used).

3.5 Sequential logic

CTU CAN FD logic is implemented from DFFs with asynchronous reset. If TXT Buffer and RX Buffer RAMs (see 3.7) are implemented from DFFs (not inferred, nor replaced by hard RAMs) and ***reset_buffer_rams*** = false, DFFs without set and reset are used. All DFFs are active on positive clock edge (to mitigate effects of clock duty-cycle). CTU CAN FD is latch free (apart from latches within clock gate cells). These facts can be used as a sanity check that there should be no DFFs without Set and Reset within CTU CAN FD after synthesis (apart from TXT Buffer / RX Buffer RAMs, if they are synthesized, not inferred, nor replaced by Hard RAM macros).

3.6 Resynchronisers

Resynchronisers within CTU CAN FD IP Core are listed in Table 3.3.

Table 3.2: Memory testing example

Step	Action
1	Set MODE[TSTM] = 1 and TSTCTRL[TMENA] = 1. This enables memory testing.
2	Configure target memory to be tested in TST_DEST[TST_MTGT] register. Set TST_DEST[TST_ADDR] = 0 (initial address).
3	Write test pattern to TST_WDATA register. It is up to user to choose test pattern.
4	Execute write to the memory by writing TSTCTRL[TWRSTB] = 1. Note that TSTCTRL[TMAENA] must remain set.
5	Increment address in TST_DEST[TST_ADDR]. If this is last address within tested memory, then go to Step 6. Otherwise go to Step 3.
6	Set TST_DEST[TST_ADDR] = 0 (initial address).
7	Wait for 1 System clock clock cycle (read from RAMs is pipelined).
8	Read value from TST_RDATA. Check that value read from this register matches what has been written TST_WDATA register in Step 3. If value does not match, test fails.
9	Increment address in TST_DEST[TST_ADDR]. If this is last address within tested memory, then go to Step 10. Otherwise go to Step 7.
10	Test is successfull.

Table 3.3: Resynchronisers

Resynchroniser function	Resynchroniser Type	Resynchroniser path
Resynchronisation of External Reset	Reset Synchroniser	can_top_level\rst_sync_inst
Resynchronisation of CAN RX Data Stream	Signal Synchroniser	can_top_level\ bus_sampling_inst\ can_rx_sig_sync_inst

3.7 Memories

CTU CAN FD contains memories which are used to store CAN FD frames. These memories are parts of RX buffer and TXT buffers (see 3.15 and 3.17). List of memories is shown in Table 3.4. Memories are designed to automatically infer dedicated synchronous RAM resources on FPGA. When integrating CTU CAN FD to ASIC, integrator can either replace these memories by hard macros, or leave memory implementation to synthesis tool. In such case, memory consists of DFFs without set or reset (memory is "uninitalized"). If it is desirable for RAMs to be reset, set ***reset_buffer_rams*** = true. When ***reset_buffer_rams*** = true, ***res_n*** RAMs to zeroes.

Each memory is synchronous memory with one clock cycle latency on data read and one cycle write access latency. Both memories are dual port memories with write-only port A, read-only port B, and the same clock signal is used to clock both ports. If true dual port memories are used, write data/enable of Port B shall be driven to 0. Memory word width is 32 bits, and it must support byte-enable capability. An example of memory access is shown in Figure 3.4. In case of read during write, memories return old data value, there is no "bypassing" implemented.

Table 3.4: RAM memories

Memory location	Write mask	Instance Name	Instances	Depth	Word Width	Address size	Port A Access	Port B Access	Read
RX Buffer RAM	No	rx_buffer_ram	1	32-4096	32	12	CAN Core	Memory Registers	Synchronous
TXT Buffer RAM	No	txt_buffer_ram	2-8	20	32	5	Memory registers	CAN Core	Synchronous

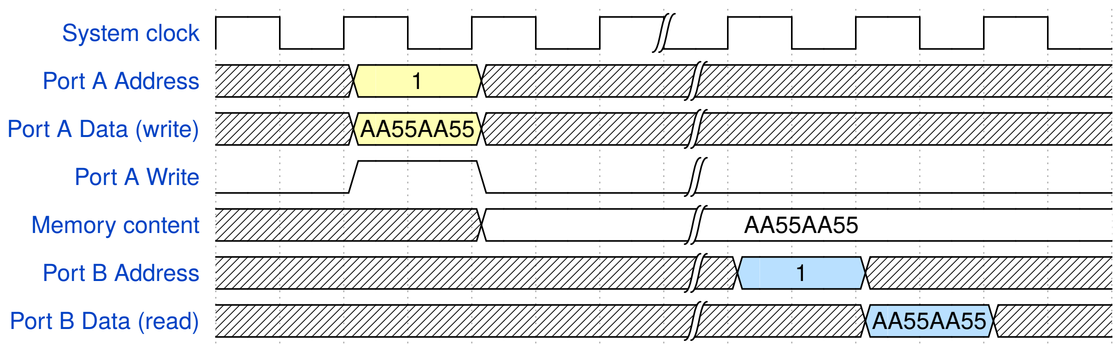

Figure 3.4: Dual port memories access

3.8 Pipeline architecture and triggers

Processing of data on CAN bus in CTU CAN FD is pipelined into three stages which are described in Table 3.5. Pipeline architecture meets maximal information processing time (2 time quanta) when System clock period is equal to time quanta. Since processing takes two clock periods information processing time of CTU CAN FD is 2 . Due to this, minimum time quanta of CTU CAN FD is 1.

Each stage of pipeline processing is controlled by trigger signal which is active for one clock cycle. Trigger signals are used to synchronise data transfer in exact moments to meet bit timing requirements on CAN Bus. Trigger signals are used as clock enable signals for DFF which process data in according pipeline stage. If trigger signal is inactive, processed data remain on DFF output and keep their previous value (data after bit destuffing (RX) and bit stuffing (TX)). An example of pipeline processing is shown in Figure 3.5. Note that Process pipeline stage always occurs one clock cycle after Destuff pipeline stage. Between Process and Stuff pipeline stage there will be number of clock cycles where no data are processed. This gap corresponds to TSEG2 (see 3.20.1 for definition of TSEG2).

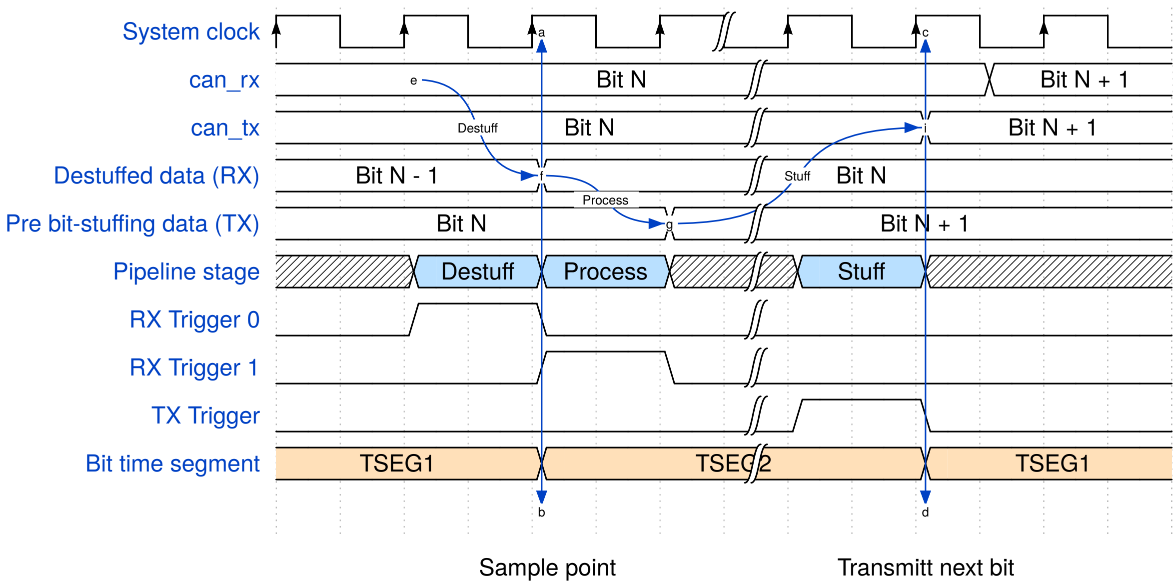

Figure 3.5: Datapath pipeline processing

In case of negative resynchronisation, length of TSEG2 can be shortened to less then 2 clock cycles, in such case following TX Trigger signal is throttled by one clock cycle and overall length of bit remains unaffected. Such situation is further described in 3.20.7. A high level algorithm for processing of data on CAN bus is described in Table 3.7.

Table 3.5: Pipeline stages

Index	Pipeline stage	Trigger signal	Corresponding moment on CAN Bus	Modules which process data in this pipeline stage	Description
1	Destuff	RX Trigger (0)	Sample point	Bus Sampling, Bit Destuffing	Stuff Bits are removed from ***can_rx*** and provided as destuffed data to Protocol control.
2	Process	RX Trigger (1)	One clock cycle after Sample point	Protocol Control	Destuffed data are processed by Protocol control, value of following transmitted bit is determined and provided as TX data before bit stuffing"
3	Stuff	TX Trigger	Start of Bit time	Bit Stuffing	Stuff bit is inserted to TX data before bit stuffing and propagated to ***can_tx***.

Table 3.7: Pipeline stages - algorithm

Step	Step Description	Pipeline Stage	Module
1	***can_rx*** input is synchronised to System clock domain. Delay imposed by synchronisation is treated as wire delay and it is ignored.	-	
2	Bus value is sampled to save information about previous sampled bus value for next edge detection. Synchronisation edges are detected on ***can_rx*** and propagated to Prescaler. ***can_rx*** value is propagated to Bit Destuffing module.	Destuff	Bus Sampling
3	Bit de-stuffing is performed in Sample point, and destuffed data are provided on output of Bit Destuffing module.	Destuff	Bit Destuffing
4	CRC from RX bit value with stuff bits included (***can_rx***) is calculated.	Destuff	CAN CRC
5	Destuffed data are sampled by Protocol control, RX shift register is shifted, TX shift register is preloaded by following bit to be transmitted, Protocol control FSM state is updated.	Process	Protocol Control
6	CRC from destuffed data is calculated.	Process	CAN CRC
7	Stuff bits are inserted to TX bit value on output of TX shift register by Bit Stuffing module. Value on output of Bit Stuffing module is propagated to ***can_tx*** output.	Stuff	Bit Stuffing
8	TX shift register is shifted.	Stuff	Protocol Control
9	CRC from output of TX shift register (TX data before bit stuffing) is calculated.	Stuff	CAN CRC
10	CRC from TX data with bit stuffing is calculated. As this stage does not affect data transmitted on the bus in the actual bit, it is not considered as separate pipeline stage.	Stuff + 1 clock cycle	CAN CRC

3.9 CAN Frame metadata

Through this document, term "frame metadata" is used for description of CAN frame information which are described in Table 3.8. In TXT Buffers and RX Buffer, metadata are stored in Frame Format word as is shown in Chapter 4 of [2].

3.10 CAN Frame format

CAN frame spans multiple 32-bit words in TXT Buffers and within RX Buffer RAMs (see 3.17 and 3.15). One TXT Buffer always contains single frame. RX Buffer contains multiple frames one after another in a RX Buffer RAM. Format of CAN frame within these memories is the same with following exceptions:

- ESI bit in TXT Buffer has no meaning while in RX Buffer ESI has value of received ESI bit on CAN bus
- RWCNT field in TXT Buffer has no meaning while in RX Buffer it contains number of words that current frame takes in RX Buffer without Frame Format word).
- FRAME_TEST_W word is available only in TXT Buffer RAM, not in RX Buffer RAM.

Meaning of memory words within CAN frame is described in Table 3.9. Meaning of individual bits can be found in Chapter 5 of [2].

Table 3.8: CAN frame metadata

Name	Abbreviation	Possible values	Description
Identifier type	ID_TYPE	BASE (0), EXTENDED (1)	Distiguishes frames with base identifier (BASE) only and frames with identifier extension (EXTENDED).
Frame type	FR_TYPE	NORMAL_CAN (0), FD_CAN (1)	Distiguishes CAN 2.0 frames and CAN FD frames.
Remote Transmission Request	RTR	NO_RTR_FRAME (0), RTR_FRAME (1)	Distinguishes between Data Frame and Remote frame. When frame is CAN FD frame, RTR bit has no meaning.
Bit Rate Shift flag	BRS	BR_NO_SHIFT (0), BR_SHIFT (1)	Distinguishes if bit rate will be shifted in CAN FD frame or not. This bit has no meaning in CAN 2.0 frames.
Error State Indicator	ESI	ESI_ERR_ACTIVE (0), ESI_ERR_PASSIVE (1)	Value of received ESI bit. This bit has no meaning in CAN 2.0 frames. This bit has no meaning in TXT buffers. Value of transmitted ESI bit is always given by actual Fault confinement state.
Data length code	DLC	0 - 15 as defined in [1]	Data length code determines length of data field within CAN frame.

Table 3.9: CAN frame format - memory words

Name of memory word	Name in register map (see [2])	Description
Frame Format	FRAME_FORM_W	Contains DLC, ESI, Frame Type, Identifier Type, BRS.
Identifier	IDENTIFIER_W	Contains base identifier base and identifier extension.
Timestamp Low	TIMESTAMP_L_W	Contains lower 32-bits of CAN frame Timestamp (in RX Buffer as sampled during frame reception, in TXT Buffer as inserted by user).
Timestamp High	TIMESTAMP_U_W	Contains upper 32-bits of CAN frame Timestamp (in RX Buffer as sampled during frame reception, in TXT Buffer as inserted by user).
Data words	DATA_X_Y_W	Contain CAN frame data payload transmitted/received during data frame field.
Frame Test	FRAME_TEST_W	Contains metadata for intentional corruption of transmitted CAN frames.

3.11 Test mode

CTU CAN FD is in Test mode when MODE[TSTM] = '1'. Features of test mode are listed in Table 3.10.

Table 3.10: Test mode features

Relevant register	Description
CTR_PRES	In test mode CTR_PRES is writable and allows setting values of transmitt error counter, receive error counter, nominal error counter and data error counter.
EWL	In test mode EWL register is read-write therefore Error warning limit is configurable by SW.
ERP	In test mode ERP register is read-write and Error passive threshold is configurable by SW. When either transmitt error counter or receive error counter reaches Error Passive threshold, unit becomes error passive.
TST_CONTROL, TST_DEST, TST_WDATA, TST_RDATA	In test mode Test registers are writable, therefore it is possible to directly read/write RX buffer RAM and TXT buffer RAMs. This feature is available only when ***sup_test_registers*** = true.
FRAME_TEST_W	CTU CAN FD uses bits in FRAME_TEST_W to intentionally corrupt transmitted CAN frames.

3.12 ISO vs NON-ISO CAN FD

CTU CAN FD supports both types of CAN FD protocol, so called ISO FD (according to [1]) and also non-ISO FD (according to [2]). By default ISO CAN FD is selected. Selection between ISO FD and NON-ISO FD is done by SETTINGS[NISOFD] register. This bit shall be changed only when device is disabled (SETTINGS[ENA] = '0'). Differences between ISO and NON-ISO FD are following:

- Stuff count and Stuff parity bit fields are not transmitted by transmitter, nor received by receiver.
- Stuff count and Stuff parity are not considered as part of CRC Check.
- Highest bit of CRC_17 and CRC_21 CRC_INIT_VECTOR is 0.

3.13 Integration vs. Reintegration

In this document term "Integration" means attempt to detect 11 consecutive recessive bits after logic 1 was written to SETTINGS[ENA] (CTU CAN FD was turned on). Term "Reintegration" means attempt to detect 129 ocurrences of 11 consecutive recessive bits after node went bus off and logic 1 was written to COMMAND[ERCRST] (SW Requests to rejoin the bus).

3.14 CAN Core

File: can_core.vhd

CAN Core implements following functionality:

- Transmission and reception of CAN frame.
- Control of TXT buffers and RX buffer.
- Bit stuffing, bit destuffing, CRC calculation and CRC check.
- Fault confinement and Operation control (transmitter, receiver, idle).
- Bus traffic counters.
- Configuration of bit rate for Prescaler and synchronisation.

CAN core block diagram is shown in Figure 3.6. CAN core is structural entity which instantiates other modules and by itself it implements nearly no logic. An exception to this rule are two multiplexers as shown in Figure 3.6. Multiplexor on TX datapath (green color) multiplexes between transmitted data after bit stuffing or constant recessive value. Constant recessive value is sent to the bus in bus monitoring mode. Multiplexor on RX datapath (red color) multiplexes input data to Bit destuffing module. During normal operation, ***can_rx*** input is used. When secondary sample point is used, data after bit stuffing are taken (transmitted data are looped back to make sure that Protocol control FSM receives proper value as real received value can be delayed by several bits). In bus monitoring mode, data afer bit stuffing logically ORed with ***can_rx*** from input of CAN core (this corresponds to re-routing transmitted bit value internally as defined in 10.14 of [1]).

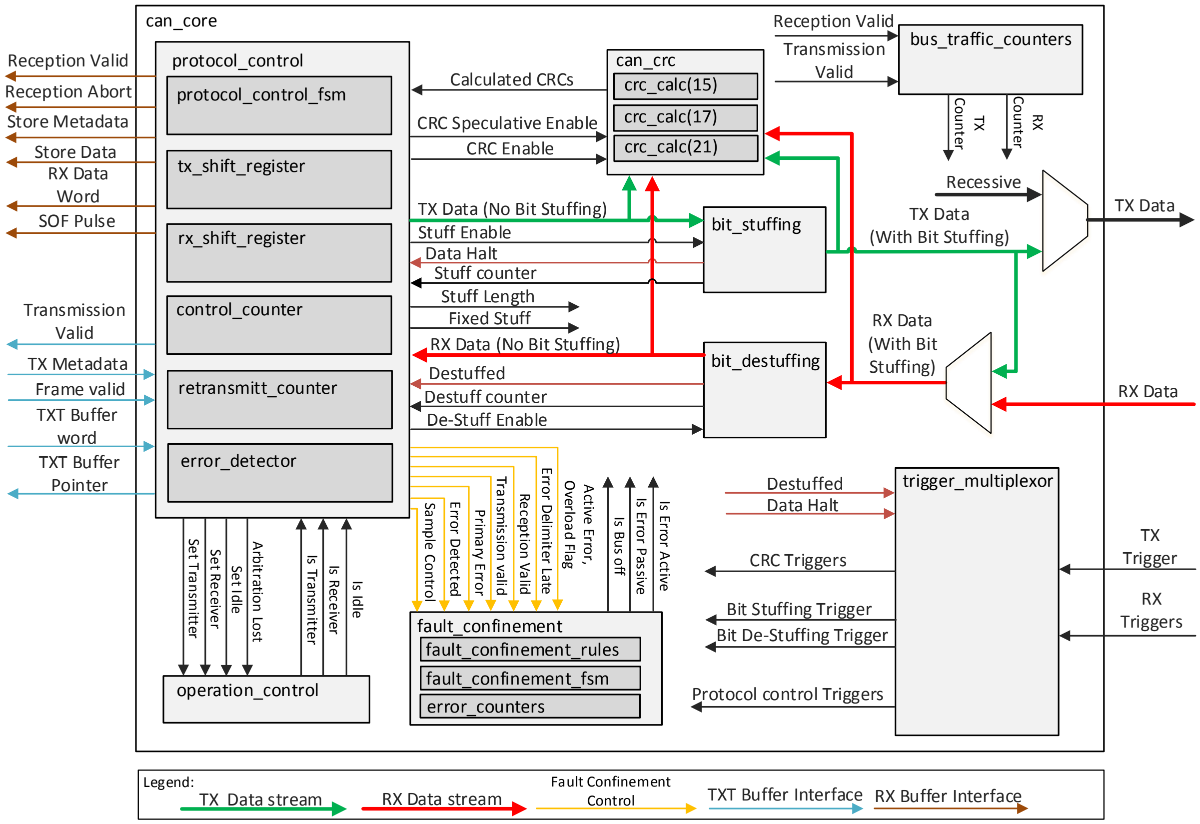

Figure 3.6: CAN Core - Block diagram

3.14.1 Protocol control

File: protocol_control.vhd

Protocol control implements following functionality:

- Transmission and reception of CAN frames.
- Handling of content-based arbitration (further in this document reffered to only as arbitration).
- Handling of bus integration state, error frame and overload frames.
- CRC check and error detection.
- Storing of received CAN frame to RX buffer.
- Reading of transmitted CAN frame from TXT buffers.
- Control of TXT buffers and TX arbitrator via HW commands.
- Counting number of frame retransmissions.
- Control synchronisation (no synchronisation, hard synchronisation, resynchronisation)
- Control bit rate switching (Nominal sample, Data sample, Secondary sample)

Protocol control diagram is shown in Figure 3.7. Protocol control is structural entity which only instantiates other modules and by itself it implements no logic.

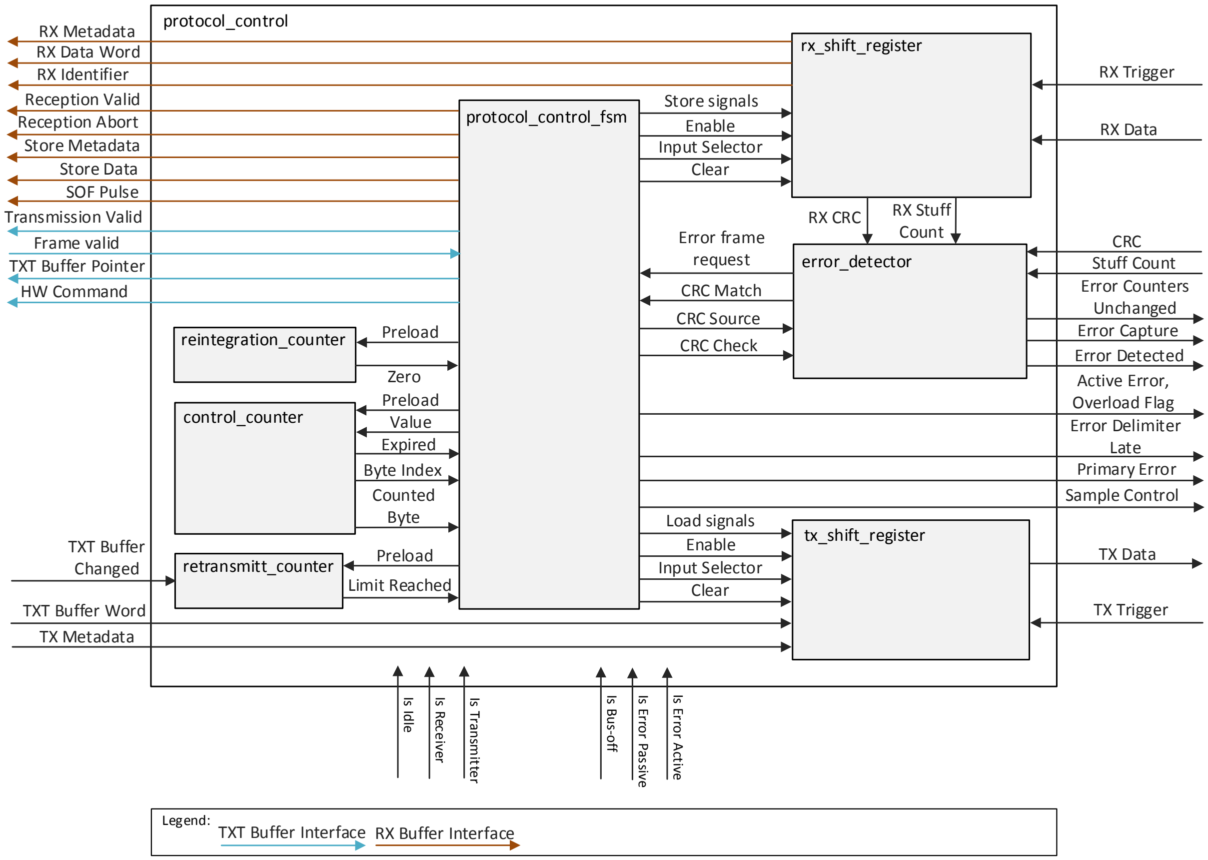

Figure 3.7: Protocol control - Block diagram

Protocol control FSM

File: protocol_control_fsm.vhd

Protocol control FSM implements following functionality:

- Transmission and reception of CAN frames.
- Controls Control counter, Retransmitt Counter, Re-integration counter.
- Controls TX Shift Register.
- Controls RX Shift Register. Storing values from RX Shift register to RX Buffer.
- Reading of transmitted frame from TXT Buffer (addressing and reading data words from TXT Buffer).
- Storing of received frame to RX Buffer.
- Controls measurement of transmitter delay.
- Controls TXT Buffers and TX Arbitrator via HW Commands.
- Controls synchronisation (no synchronisation, hard synchronisation, resynchronisation)
- Controls bit rate switching (Nominal Sample, Data Sample, Secondary Sample).
- Performs form error detection.
- Evaluate results of CRC check.
- Handles arbitration.

Protocol control FSM state transition diagam is shown in Figure 3.8. Rules for Protocol control FSM state transitions are described in Table 3.11. Protocol control FSM does not change its state in any other moment. Note that regular change of Protocol control FSM state corresponding to e.g. transition from control field to data field occurs one clock cycle after sample point (in Process pipeline stage).

Table 3.11: Protocol control state transition rules

Condition of state transition	Pipeline stage when transition occurs.	Description
Regular condition	Process	Transition corresponds to regular change of CAN frame field (e.g. stuff count to CRC).
Error frame request	One clock cycle after Process	Transition corresponds to start of active error flag or passive error flag and can occur from any state of Protocol control FSM.

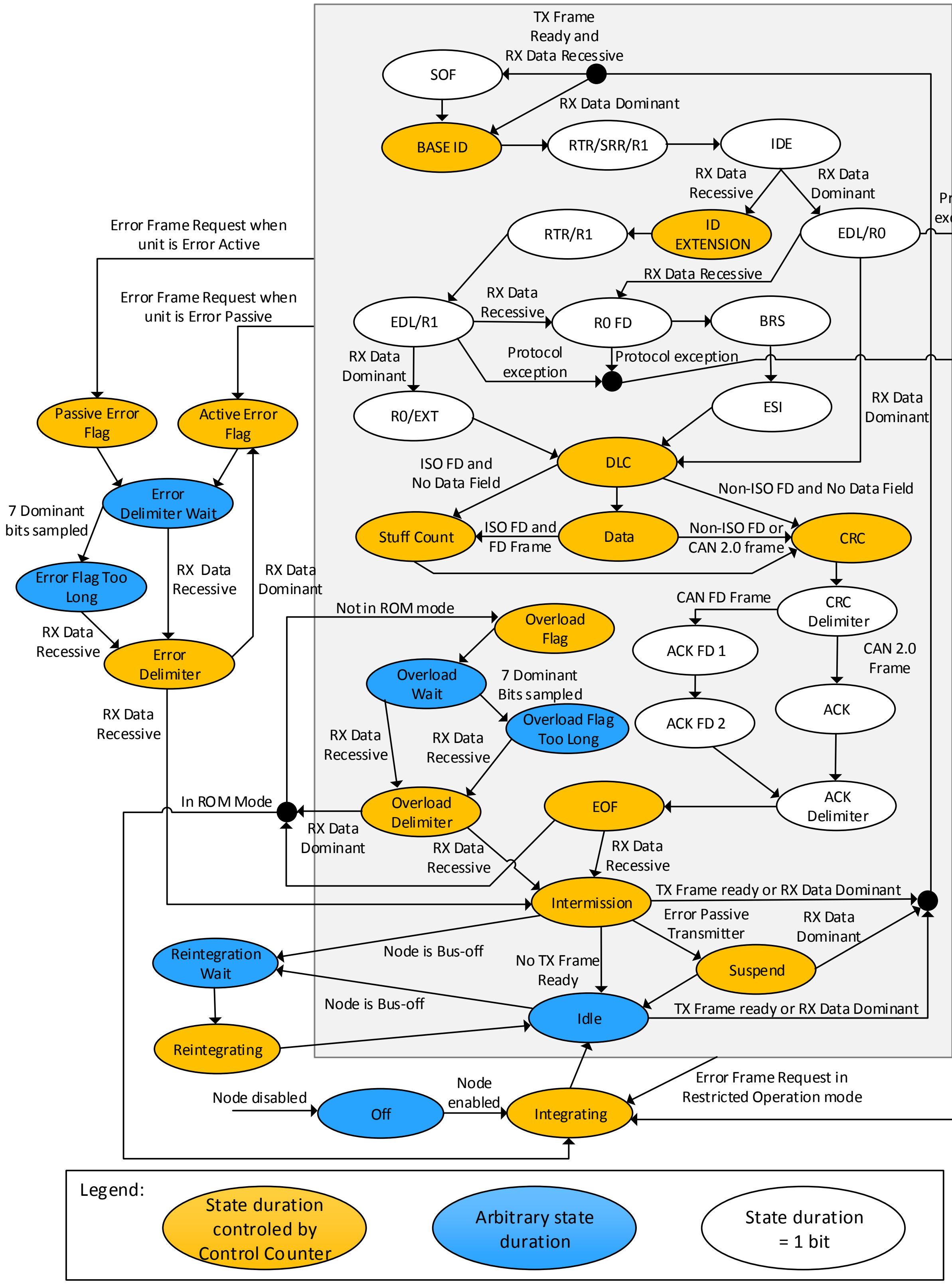

Figure 3.8: Protocol control FSM

Control counter

File: control_counter.vhd

Control counter measures duration of CAN frame fields which last longer than 1 bit. These fields and according configuration of Control counter are shown in Table 3.12. Control counter is preloaded in Process pipeline stage and it counts towards zero. Control counter counting is controlled by Protocol control FSM. It is decremented by 1 in each bit of CAN frame field in Process pipeline stage. When Control counter is equal to 1 and 0, this is signalled to Protocol control FSM. This situation indicates one bit before end of CAN frame field or last bit of CAN frame field. A current CAN frame field ends when Control counter is zero. Control counter is not counting during CAN frame fields which last only 1 bit (e.g. IDE bit), nor during fields which might last arbitrary number of bits (bus idle). An example of Control counter operation during base identifier in CAN frame is shown in Figure 3.9.

Table 3.12: Control counter

CAN Frame field	Control counter preload value
Base identifier	10
Identifier extension	17
Data length code	3
Data	Depends on transmitted / received data field length.
CRC	14, 16, 20 - depends on length of CRC sequence
Stuff count (+ Stuff parity)	3
End of Frame	7
Interframe space	2
Suspend transmission	7
Integration	10
Error flag, overload flag	5
Error delimiter, Overload delimiter	7
Re-integration	11, preloaded 129 times.

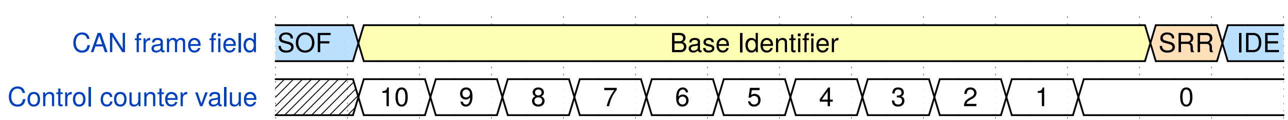

Figure 3.9: Control counter operation

Control counter module contains a complementary counter which counts from 0. Complementary counter is incremented by 1 each bit time in Process pipeline stage and it counts only during data field. Complementary counter provides information that data byte has elapsed (when counter mod 8 == 0), or whole memory word has elapsed (when counter mod 32 == 0). Complementary counter addresses memory words between addresses 4 (DATA_1_4_W) and 19 (DATA_61_64_W) in TXT Buffer. Complementary counter decodes address of Data memory word within TXT Buffer according to following equation:

$$Memory\,word\,index = \left(\frac{Control\,counter}{32}\right) + 4$$

Control counter module implements Arbitration lost capture register. Arbitration lost capture register stores position within CAN frame at which arbitration was lost. Arbitration lost capture register is loaded when arbitration lost is signalled by Protocol Control FSM in Process pipeline stage. Arbitration lost capture saves current value of Control counter (determines bit at which arbitration was lost) and bit field type within arbitration (base identifier, IDE bit, identifier extension, etc.) when arbitration was lost. Arbitration lost capture register is readable by SW via ALC register. Meaning of values in Arbitration lost capture register is described in [2]. An example of Arbitration lost capture register is shown in Figure 3.10.

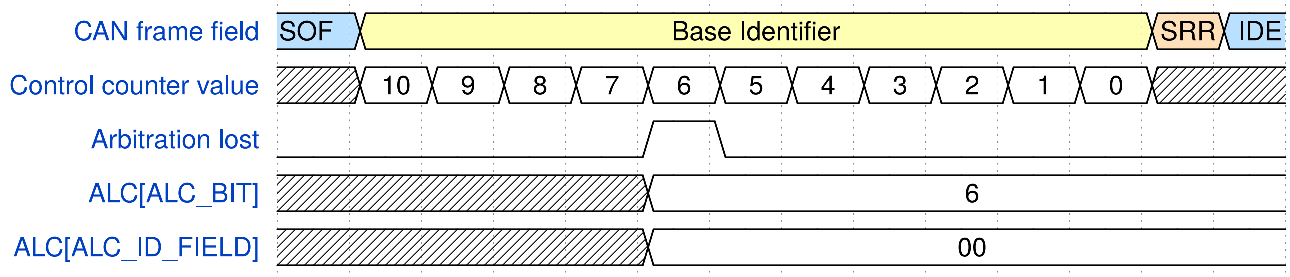

Figure 3.10: Arbitration lost capture

Retransmitt counter

File: retransmitt_counter.vhd

Retransmitt counter controls number of retransmissions of current CAN frame from dedicated TXT Buffer. Retransmitt counter counts from zero, and it is controlled by Protocol control FSM. Retransmitt counter counts only when retransmitt limitation is enabled by user (SETTINGS[RTRLE] = '1'), otherwise it stays at 0. When retransmitt limitation is disabled (SETTINGS[RTRLE] = '0') frame transmission is attempted indefinite amount of times. Retransmitt counter is incremented by 1 when arbitration is lost, or when error frame transmission is requested by Error detector (reffer to 3.14.1).

When error frame and arbitration loss occur in the same frame, retransmitt counter is incremented only once (such a situation is shown in Figure 3.12). When multiple error frames occur in the same frame (e.g. due to error during error frame), retransmitt counter is also incremented only once.

When Retransmitt counter reaches retransmitt limit (SETTINGS[RTRTH]), it signals this to Protocol control FSM. In case of next arbitration loss or error frame request, Protocol control FSM stops transmitting actual frame, signals this to TXT Buffer and TXT Buffer moves to TX Failed state (see Figure 3.30). When unit is a receiver without attempt to transmitt frame (no frame was available during bus idle, intermission), retransmitt counter is not modified during this frame. When unit is error passive and transmission of a frame is not succesfull, unit becomes receiver of next frame (due to suspend transmission field) without attempting to transmitt a frame. If error occurs during next frame, retransmitt counter is not incremented. Possible configurations of retransmitt limit are shown in Table 3.13.

Retransmitt counter is cleared when TXT Buffer used for transmission changes between two consecutive transmissions (another TXT Buffer with another TX Frame selected by TX Arbitrator), as is described in Table 3.54. Retransmitt counter is cleared upon succesfull transmission (TXT Buffer goes to TX OK state) or when transmission fails (TXT Buffer goes to TX Failed state). Retransmitt counter is also cleared when TXT Buffer which is currently used for transmission goes to Aborted state.

Table 3.13: Retransmitt limit configuration

SETTINGS[RTRTH	SETTINGS[RTRLE]	Behaviour
-	0	Frame transmission is attempted without any limitation until unit turns Bus-off.
0	1	Frame transmission is attempted only once, there is no retransmission attempt after first failed transmission (so called one-shot mode).
1 - 15	1	Frame transmission is attempted SETTINGS[RTRTH] times.

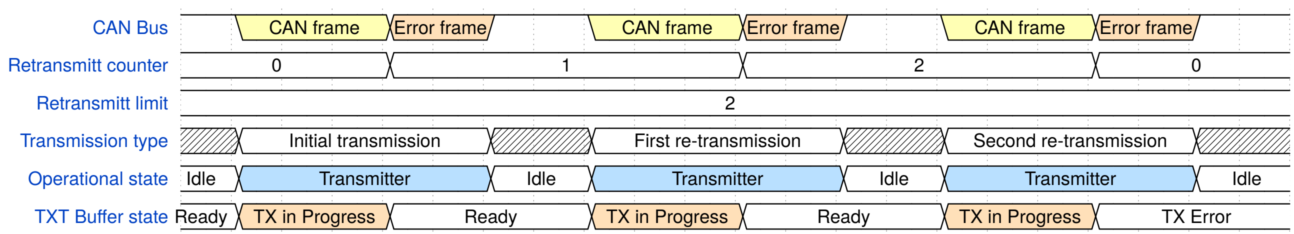

Figure 3.11: Retransmitt counter operation

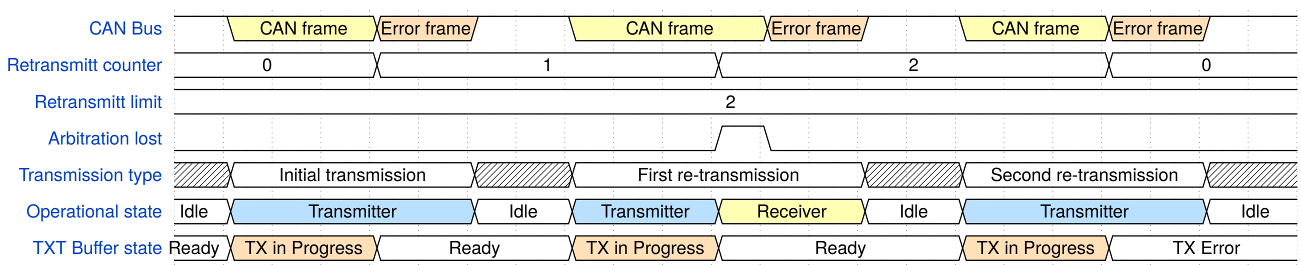

Figure 3.12: Retransmitt counter - arbitration loss and error frame

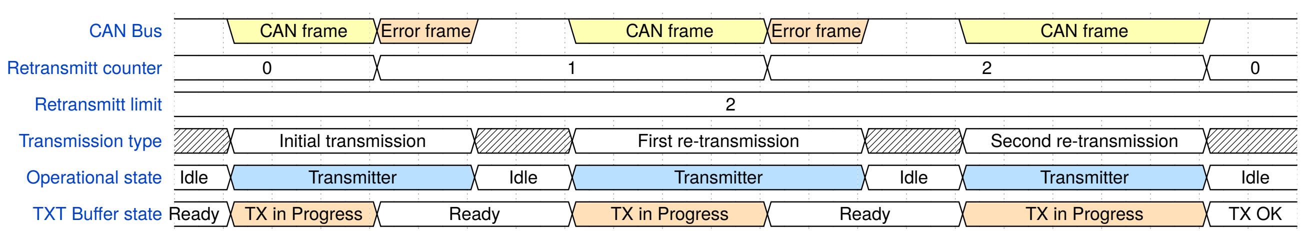

Figure 3.13: Retransmitt counter - second retransmission succesfull

Reintegration counter

File: reintegration_counter.vhd

Reintegration counter counts 129 consecutive ocurrences of 11 consecutive recessive bits after unit turned bus-off. Reintegration counter counts only during reintegration, not during initial bus integration. It is controlled by Protocol control FSM, it counts from zero and it is cleared when unit is bus-off and it receives command to reset error counters (by writing logic 1 to COMMAND[ERCRST] register). Reintegration counter is incremented by 1 after each 11 consecutive recessive bits are received. 11 consecutive recessive bits are measured by Control counter. If during reintegration dominant bit is detected, Control counter is pre-loaded again to 10 (there was dominant bit before 11 consecutive recessive bits were reached). When reintegration counter reaches 128 (0-128 = 129 times), it signals this to Protocol control FSM (Protocol control FSM becomes Idle), unit becomes error active again and operation control state is changed to Idle. An example use case of reintegration counter operation is shown in Table 3.14.

Table 3.14: Reintegration counter - use case

Step	Action
1	CTU CAN FD is enabled by writing SETTINGS[ENA] = '1'. After bus integration is over, unit becomes error active.
2	CTU CAN FD takes part in bus communication. Due to error frames, it turns first error passive and then bus-off.
3	SW is notified of such an event by FCS interrupt, then SW reads FAULT_STATE register and finds out that unit is bus-off.
4	SW decides that it wants the unit to join the network again. SW writes logic 1 to COMMAND[ERCRST] (so called "error counter reset" command or "reintegration request")
5	Reintegration counter is cleared. Control counter is preloaded to 10.
6	Control counter is being decremented by 1 for each recessive bit received by Protocol Control FSM. If dominant bit is detected, Control counter is preloaded to 10 again.
7	After 11 consecutive recessive bits are received, Control counter is 0, it signals this to Protocol control FSM.
8	Protocol control FSM increments Reintegration counter by 1.
9	After 129 repetitions of 11 consecutive recessive bits (note that there can be CAN frames between consecutive sequences of 11 consecutive recessive bits, these frames are ignored by CTU CAN FD), Reintegration counter is 128. Reintegration counter signals this to Protocol Control FSM.
10	Protocol control FSM becomes Idle, CTU CAN FD becomes error active and it is ready to receive/transmitt frames again.

TX shift register

File: tx_shift_reg.vhd

TX shift register is 32 bit shift register which transmitts given bit sequence to the output of Protocol control module. TX shift register is preloaded by Protocol control FSM in Process pipeline stage when new data sequence is about to be transmitted, thus output value is also valid after Process pipeline stage of the same bit. TX shift register is shifted by one position in Stuff pipeline stage of each bit on CAN bus during multi-bit frame fields. When stuff bit is inserted, TX shift register is not shifted (Protocol control is halted for one bit).

TX shift register is preloaded according to Table 3.15. TX shift register is enabled only as long as unit is transmitter, TX shift register is not shifting when unit is receiver, nor during CAN frame fields which last only one bit (SOF, ACK, etc.), nor during fields which transmitt constant sequence (EOF, error flag, etc.). In such case constant value is transmitted on its output. TX shift register shifts from lowest bit index to highest bit index (shifting up). Transmission of single bits (e.g. SOF, ACK) or constant sequences (e.g. active error flag, EOF) is handled by separate logic inside TX shift register, and has higher priority than transmission from TX shift register. Rules for handling of these situations are described in Table 3.16. An example of TX shift register operation during CAN frame is shown in Table 3.17

Table 3.15: TX shift register preload rules

CAN frame fields in which TX shift register is preloaded	Preloaded bit sequence	Where the bit sequence is preloaded from
SOF, suspend transmission, intermission, idle	Base identifier	Identifier capture register in TX Arbitrator.
IDE bit	Identifier extension	Identifier capture register in TX Arbitrator.
r0 bit of CAN 2.0 frame with identifier extension, EDL/r0 bit. ESI bit	Data length code	Metadata capture registers in TX Arbitrator.
Last bit of data length code, in data field when multiple of 32 bits of data field were transmitted.	Data word (4 bytes) for transmission.	TXT Buffer RAM data output on Port B.
Last bit of data length code in ISO CAN FD frames without data field, in last bit of data field in ISO CAN FD frames.	Stuff count and stuff parity.	Counter of stuffed bits in Bit Stuffing module.
Last bit of stuff count, last bit of data field in non-ISO CAN FD frames (no stuff-count), last bit of data length code in non-ISO CAN frames with no data field.	Calculated CRC.	CRC calculation register in CAN CRC module.

Table 3.16: TX Shift register - handling of special cases

Bit value transmitted	Special conditon
Dominant	Error frame request - unit is error active
Recessive	Error frame request - unit is error passive
Dominant	Protocol control FSM requests transmission of dominant bit
Recessive	TX shift register is disabled and none of the above conditions apply. This situation corresponds to transmission of continous stream of recessive bits.

Table 3.17: TX shift register - example of operation

CAN Frame:	Base identifier: 0x123 DLC: 0x1 Data: 0xAB Frame Type: CAN FD Frame Identifier Type: Base Identifier
Bit on CAN bus	TX Shift Register status, left-most bit transmitted on output of Protocol Control, transmitted sequence boldom
SOF	00000000 00000000 00000000 00000000
Base ID - Bit 1	**00100100 011**00000 00000000 00000000 (Base ID: 0x123: 00100100011)
Base ID - Bit 2	**01001000 11**000000 00000000 00000000
Base ID - Bit 3	**10010001 1**0000000 00000000 00000000
Base ID - Bit 4	**00100011** 00000000 00000000 00000000
Base ID - Bit 5	**0100011**0 00000000 00000000 00000000
Base ID - Bit 6	**100011**00 00000000 00000000 00000000
Base ID - Bit 7	**00011**000 00000000 00000000 00000000
Base ID - Bit 8	**0011**0000 00000000 00000000 00000000
Base ID - Bit 9	**011**00000 00000000 00000000 00000000
Base ID - Bit 10	**11**000000 00000000 00000000 00000000
Base ID - Bit 11	**1**0000000 00000000 00000000 00000000
RTR	00000000 00000000 00000000 00000000
IDE	00000000 00000000 00000000 00000000
r0	00000000 00000000 00000000 00000000
DLC - Bit 1	**0001**0000 00000000 00000000 00000000 (DLC: 0x1 0001)
DLC - Bit 2	**001**00000 00000000 00000000 00000000
DLC - Bit 3	**01**000000 00000000 00000000 00000000
DLC - Bit 4	**1**0000000 00000000 00000000 00000000
Data - Bit 1	**10101011** 00000000 00000000 00000000 (Data: 0xAB 10101011)
Data - Bit 2	**0101011**0 00000000 00000000 00000000
Data - Bit 3	**101011**00 00000000 00000000 00000000
Data - Bit 4	**01011**000 00000000 00000000 00000000
Data - Bit 5	**1011**0000 00000000 00000000 00000000
Data - Bit 6	**011**00000 00000000 00000000 00000000
Data - Bit 7	**11**000000 00000000 00000000 00000000
Data - Bit 8	**1**0000000 00000000 00000000 00000000

RX shift register

File: rx_shift_reg.vhd

RX shift register is 32 bit shift register which receives bit sequence and stores parts of this sequence to dedicated capture registers when commanded by Protocol control FSM. RX shift register operates in two basic modes as is described in Table 3.18. Mode of RX shift register determines whether input of each byte in shift register is taken from output of previous byte, or directly from input of RX shift register. Diagram of RX shift register is shown in Figure 3.14. Shifting of each byte of RX shift register is enabled separately and it is controlled by Protocol control FSM. RX Shift register is shifting during multi-bit fields on CAN bus and it shifts by one position each bit in Process pipeline stage. This corresponds to reception of bit on CAN bus. RX shift register shifts up. RX shift register stores part of its content to either a dedicated capture register, or RX Buffer memory when signalled to do so by Protocol control FSM as described in Table 3.19. Received CRC sequence is not stored into any capture register and it is used for CRC check directly from RX shift register (CRC frame field is the last field of CAN frame which is shifted into RX shift register, therefore after CRC frame field, CRC remains in RX shift register).

RX shift register is not used till the end of frame and its content remains stable. Other one bit metadata information are stored to dedicated capture registers directly from input of RX shift register in corresponding fields of CAN frame as described in Table 3.20. An example of RX shift register operation is shown in 3.21

Table 3.18: RX shift register modes

RX Shift register mode	Bit fields on CAN bus when mode is used.	Byte which is enabled.	Description
Linear mode	Base identifier, identifier extension, DLC, CRC sequence, Stuff count	All bytes are enabled.	Shift register forms single 32-bit shift register. Inputs of each next byte are connected to outputs of previous byte. All bits are shifted simultaneously.
Byte mode	Data field	Only one byte is enabled at any time. Enabled byte is given by index of actually received data field byte on CAN bus.	Shift register forms 4 separate 8-bit shift registers. Inputs of each byte are connected to input of RX shift register. Only 1 shift register (one byte) is shifted at any time.

Table 3.19: RX shift register - stored sequences

Bit on CAN bus in which RX shift register stores part of its content.	Meaning of stored sequence	Destination where value is stored.
Last bit of base identifier	Base identifier	Capture register.
Last bit of identifier extension	Extended identifier	Capture register.
Last bit of data length code	Data length code	Capture register.
Last bit of data field or last bit of memory word within data field (after each 32 bits).	4 bytes (single memory word) of data field.	RX Buffer RAM memory.
Last bit of stuff count	Grey coded stuff count + stuff parity	Capture register.

Table 3.20: RX shift register - stored single bits

Protocol control FSM state	Meaning of stored bit	Corresponding metadata signal	Destination where value is stored.
BRS	Value of bit rate switch bit	BRS	Capture register
ESI	Value of error state indicator bit	ESI	Capture register
IDE	Value of identifier extension bit	ID_TYPE	Capture register
RTR/SRR/R1, RTR/R1	Value of remote transmission request Bit	RTR	Capture register
EDL/R0, EDL/R1	Value of extended data length / flexbile data-rate format bit	FR_TYPE	Capture register

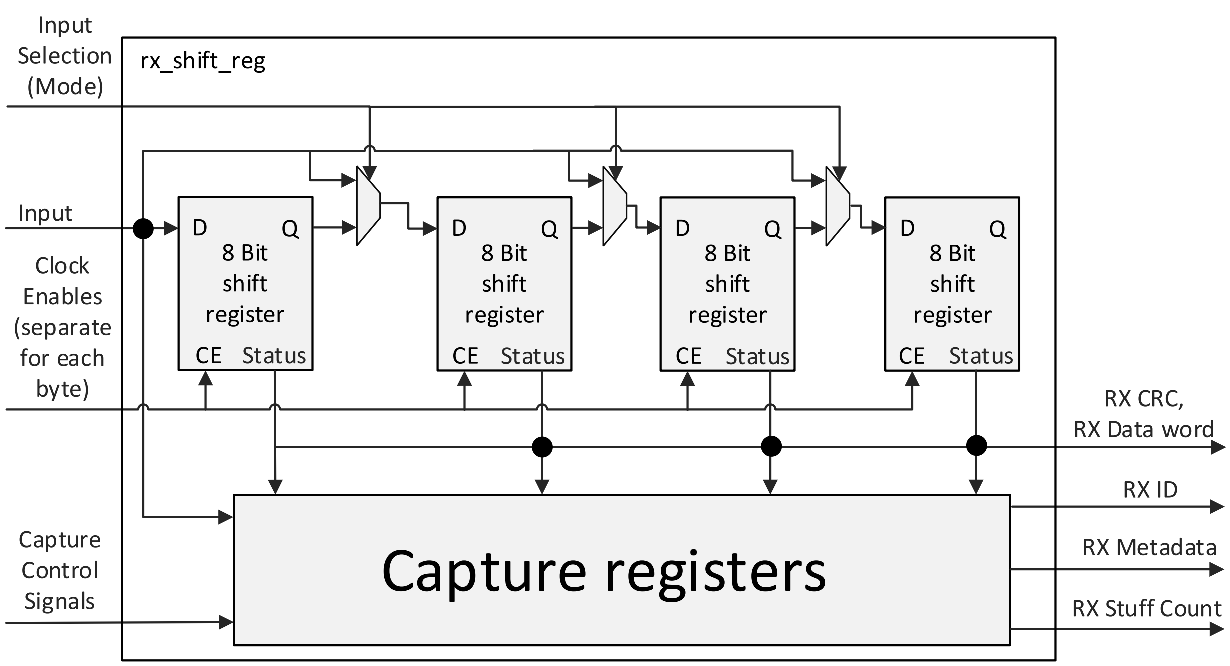

Figure 3.14: RX shift register - Block diagram

Table 3.21: RX shift register operation

CAN Frame:		Base ID: 0x123 DLC: 0x2 Data: 0xAB 0xCD Frame Type: CAN FD Frame Identifier Type: Base Identifier
Bit on CAN bus	Mode	RX shift Register status, right most bit is received on input of Protocol control, received sequence boldom
SOF	-	00000000 00000000 00000000 00000000
Base ID - Bit 1	Linear	00000000 00000000 00000000 0000000**0**
Base ID - Bit 2	Linear	00000000 00000000 00000000 000000**00**
Base ID - Bit 3	Linear	00000000 00000000 00000000 00000**001**
Base ID - Bit 4	Linear	00000000 00000000 00000000 0000**0010**
Base ID - Bit 5	Linear	00000000 00000000 00000000 000**00100**
Base ID - Bit 6	Linear	00000000 00000000 00000000 00**001001**
Base ID - Bit 7	Linear	00000000 00000000 00000000 0**0010010**
Base ID - Bit 8	Linear	00000000 00000000 00000000 **00100100**
Base ID - Bit 9	Linear	00000000 00000000 0000000**0 01001000**
Base ID - Bit 10	Linear	00000000 00000000 000000**00 10010001**
Base ID - Bit 11	Linear	00000000 00000000 00000**001 00100011** (Base ID: 0x123: 00100100011)
RTR	-	00000000 00000000 00000001 00100011
IDE	-	00000000 00000000 00000001 00100011
r0	-	00000000 00000000 00000001 00100011
DLC - Bit 1	Linear	00000000 00000000 00000010 0100011**0**
DLC - Bit 2	Linear	00000000 00000000 00000100 100011**00**
DLC - Bit 3	Linear	00000000 00000000 00001001 00011**001**
DLC - Bit 4	Linear	00000000 00000000 00010010 0011**0010** (DLC: 0x2 0010)
Data Byte 0 - Bit 1	Byte	00000000 00000000 00010010 0011001**1**
Data Byte 0 - Bit 2	Byte	00000000 00000000 00010010 001100**10**
Data Byte 0 - Bit 3	Byte	00000000 00000000 00010010 00110**101**
Data Byte 0 - Bit 4	Byte	00000000 00000000 00010010 0011**1010**
Data Byte 0 - Bit 5	Byte	00000000 00000000 00010010 001**10101**
Data Byte 0 - Bit 6	Byte	00000000 00000000 00010010 00**101010**
Data Byte 0 - Bit 7	Byte	00000000 00000000 00010010 0**1010101**
Data Byte 0 - Bit 8	Byte	00000000 00000000 00010010 **10101011** (Data: 0xAB 10101011)
Data Byte 1- Bit 1	Byte	0000000 00000000 0001001**1** 10101011
Data Byte 1- Bit 2	Byte	0000000 00000000 000100**11** 10101011
Data Byte 1- Bit 3	Byte	0000000 00000000 00010**110** 10101011
Data Byte 1- Bit 4	Byte	0000000 00000000 0001**1100** 10101011
Data Byte 1- Bit 5	Byte	0000000 00000000 000**11001** 10101011
Data Byte 1- Bit 6	Byte	0000000 00000000 00**110011** 10101011
Data Byte 1- Bit 7	Byte	0000000 00000000 0**1100110** 10101011
Data Byte 1- Bit 8	Byte	0000000 00000000 **11001101** 10101011 (Data: 0xCD 1100 1101)

Error detector

File: err_detector.vhd

Error detector processes errors detected by other modules, decides whether these errors are valid and creates error frame request to Protocol control FSM. Errors are detected in Process pipeline stage and error frame request is provided to Protocol control FSM one clock cycle after Process pipeline stage. Error frame request is registered to avoid combinatorial loops between Error detector and Protocol control FSM. Error types and modules of their origin are described in Table 3.22. Error detector containts Error code capture register which stores type and position of last error. Error code capture register is loaded when Error detector creates error frame request to Protocol control FSM. Reffer to [2] for description of Error code capture register. An example of error detection (form error) with details of actions in each pipeline stage is shown in Figure 3.15.

Table 3.22: Error detection rules (part 1)

Error type	CAN frame fields when error is detected	CAN Frame Fields where Error can't occur	Module where error is detected	Description
Bit error	SOF, control, data, stuff count, CRC, CRC delimiter	Can occur anywhere	Bit error detector in Bus sampling module	Bit error is detected when transmitted and received value of bit on CAN bus differs. Reffer to 3.21 for details of bit error detection by Bus sampling module. Bit error detection by Bus sampling module is enabled always, it is only ignored in bit fields as described in 3.26.
	Arbitration field	Can occur anywhere	Protocol control FSM	In arbitration field, bit error detected by Bus sampling is ignored by Error detector. Instead bit error detected by Protocol control FSM is considered. Protocol control FSM detects bit error during arbitration field only when transmitted bit was dominant and received bit is recessive.
Stuff error	Arbitration field, control, data, stuff count, CRC	Intermission, idle, suspend, error frame, overload frame, end of frame, CRC delimiter, ACK, ACK delimiter	Bit destuffing module	Stuff error is detected by Bit destuffing module as described in 3.14.5. If fixed stuff bit does not have oposite value as previous bit, this error is detected as stuff error by Bit destuffing module, but error is stored as form error in Error code capture register.
Form error	SOF, control, stuff count, CRC, EOF	Arbitration, data field, ACK, intermission, suspend transmission	Protocol control FSM, Bit destuffing module for fixed stuff bits.	Form error is detected by Protocol Control FSM by checking received bit during fixed frame fields as described in 3.24. Protocol control signals form error to Error detector and based on this, Error frame request is signalled one clock cycle after Process pipeline stage.

Table 3.23: Error detection rules (part 2)

Error type	CAN frame fields when error is detected.	CAN frame fields where error can't occur.	Module where error is detected	Description
CRC error	ACK delimiter	SOF, Arbitration, Control, Data, Stuff Count, CRC, CRC Delimiter, ACK, End of Frame, Intermission, Bus idle, Error frame, Overload frame	Protocol control FSM	Comparison of RX CRC with calculated CRC is executed in Error detector. Since after CRC field, RX shift register is not shifting and CRC module is not calculating CRC anymore, comparison shows valid result from CRC delimiter further. Based on result of comparison "CRC match" is signalled to Protocol control FSM. If unit is receiver and "CRC match" is not signalled to Protocol control FSM in ACK delimiter, Protocol control FSM detects CRC error (in Process pipeline stage of ACK delimiter) and propagates it back to Error detector. Error detector forms Error frame request for Protocol control FSM. An example of CRC check mechanism and detection of CRC error is shown in Figure 3.16.
ACK error	ACK	SOF, Arbitration, Control, Data, Stuff Count, CRC, CRC Delimiter, ACK Delimiter, End of Frame, Intermission, Bus idle, Error frame, Overload frame	Protocol control FSM	ACK error is detected by Protocol control FSM when unit is transmitter, recessive bit is received and unit is not in Self test mode (frame valid also without ACK dominant).

Table 3.24: Form error detection

CAN frame field	Condition
SOF	If recessive bit is received, form error is detected.
r0 bit after EDL/r1 bit in frame with extended identifier or r0 bit in CAN FD frames	If recessive bit is received, form error is detected when SETTINGS[PEX] = '0'. Recessive bit would mean extending beyond CAN FD standard (CAN XL). When SETTINGS[PEX] = '1', form error is not detected and CTU CAN FD enters integration.
CRC delimiter, ACK delimiter	If dominant bit is received, form error is detected.
EOF	If dominant bit is detected at all but last bit of EOF, form Error is detected. At last bit dominant bit means Error frame only for transmitter. For receiver, it means Overload condition.
All but last bit of error delimiter and overload delimiter	If dominant bit is received, form error is detected.

Table 3.26: Bit error by Bus sampling module exceptions

Frame Field/ Protocol control FSM state	Description
SOF	Dominant bit is transmitted. Bit error would be detected when recessive value was received. Such a situation is treated as form error, and bit error is ignored.
bus integration, reintegration	Recessive value is transmitted, receiving dominant is not detected as bit error since these might represent a frame between other units while CTU CAN FD is integrating.
arbitration field	Bit error is detected by Protocol control FSM, thus bit error detected by Bus sampling module is ignored.
Control, data, stuff count, CRC	Bit error detected by Bus sampling module is ignored if unit is receiver. Receiver in these fields transmitts only recessive bits and reception of dominant bit is not treated as bit error since unit is receiving data from other transmitter.
CRC delimiter	Receiving dominant bit during is interpreted as form error, due to this reason bit error detected by Bus sampling module is ignored.
ACK	Bit error is ignored, as is defined in [1].
ACK delimiter	During ACK delimiter, recessive value is transmitted and reception of dominant value is considered as form error. Due to this reason bit error is ignored.
EOF	Reception of dominant bit during EOF is treated as form error due to this bit error is ignored.
Intermission	Recessive value is sent to the bus. Receiving dominant bit during first or second bit of intermission is interpreted as overload frame. Receiving dominant bit during third bit of intermission is interpreted as SOF of next frame. Due to these reasons, bit error during intermission is ignored.
Suspend transmission, idle	Recessive value is sent to the bus. Receiving dominant bit is interpreted as SOF of next frame. Due to this reason bit error during suspend transmission and idle is ignored.
Reintegration wait	When unit turned bus-off, it is de-facto off the bus, It shall not transmitt anything unless it re-intagrates. Due to this reason bit error is ignored.
Passive error flag	Detecting dominant bit during passive error flag is not interpreted as bit error since it is defined like so in [1].
Error delimiter, Overload delimiter	Recessive bit is sent to the bus. Receiving dominant bit is interpreted as form error. Due to this bit error is ignored.

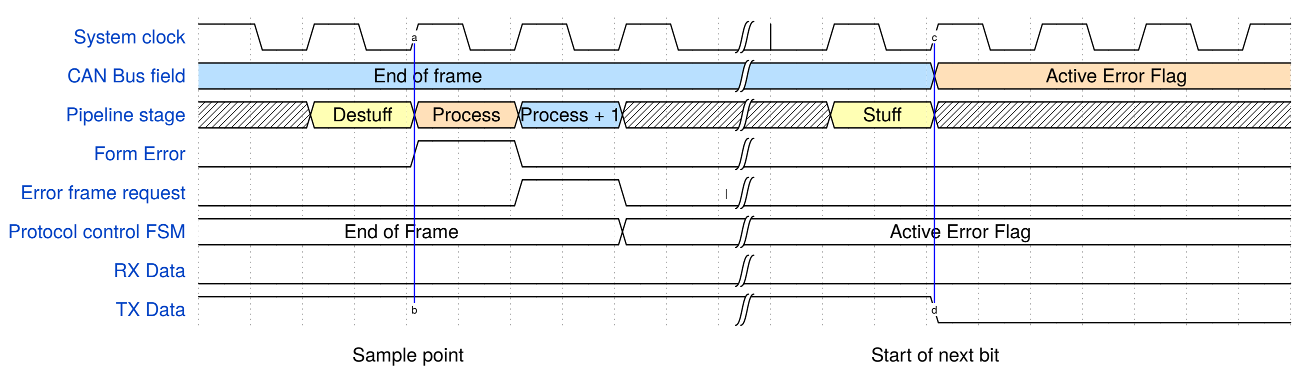

Figure 3.15: Error detection example (form error)

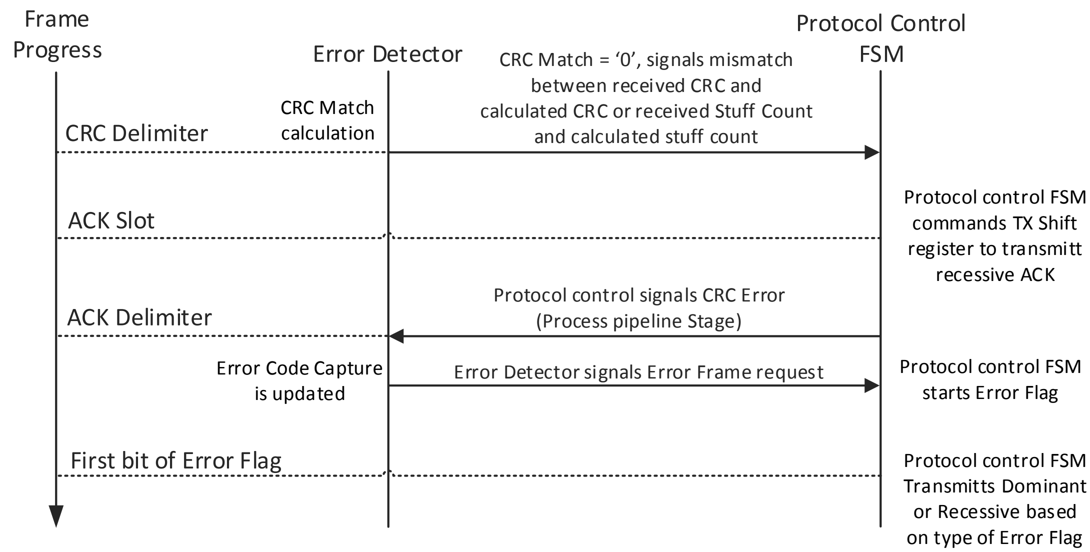

Figure 3.16: CRC check and CRC error signalling

3.14.2 Operation control

File: operation_control.vhd

Operation control implements following functionality:

- Operational state of CTU CAN FD node (transmitter, receiver, idle).

Operation control implements a FSM whose state transition diagram is shown in Figure 3.17. It is controlled by Protocol control FSM and Fault confinement FSM. Rules for control of Operation control FSM are described in Table 3.27.

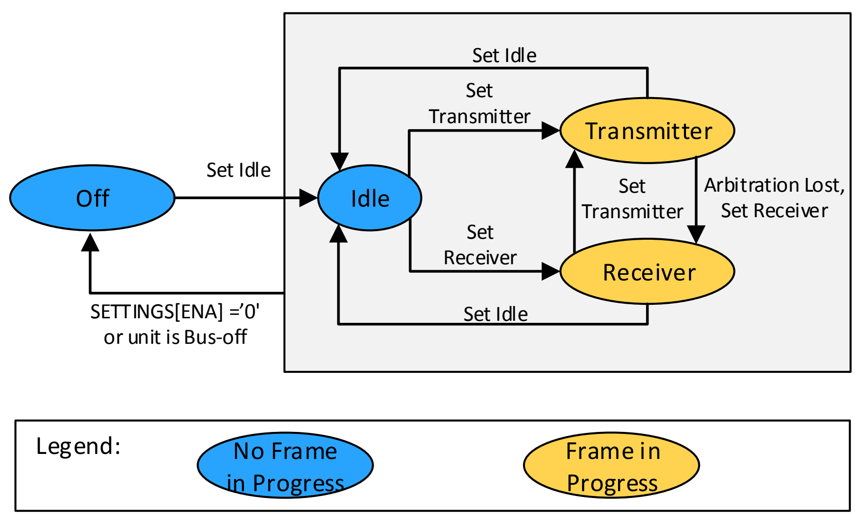

Figure 3.17: Operation control FSM

Table 3.27: Operation control FSM - state transitions

Actual state	Next state	Description
Off	Idle	When unit is turned on (SETTINGS[ENA]='1'), unit integrates to the bus communication. After integration is finished (11 consecutive recessive bits received), Protocol control signals ***set_idle***. Unit becomes idle.
Idle	Transmitter	Unit is idle and in sample point TX arbitrator signals available frame for transmission, Protocol control FSM locks Validated TXT buffer (reffer to 3.49), Protocol control signals ***set_transmitter*** and unit becomes transmitter of frame from Validated TXT buffer.
Idle	Receiver	Unit is idle, there is no available frame for transmission signalled by TX arbitrator. Dominant bit is sampled, Protocol control FSM signals ***set_receiver*** and unit becomes receiver of next frame.
Transmitter	Receiver due to ***set_receiver***	Unit transmitts frame. In last bit of intermission field, unit is still transmitter, unit detects dominant bit and considers this bit as SOF (reffer to [1]). If there is no available frame for transmission signalled by TX arbitrator, Protocol control FSM signals ***set_receiver*** and unit becomes receiver of following frame.
		Unit is error passive and it transmitts a frame. It enters suspend transmission. If during suspend transmission, dominant bit is detected, Protocol control FSM issues ***set_receiver*** and unit becomes receiver of next frame.
Transmitter	Receiver due to ***arbitration_lost***	If during arbitration field recessive bit is sent on the bus, but dominant bit is monitored by Protocol control FSM, ***arbitration_lost*** is signalled and unit becomes receiver.
Transmitter	Idle	Unit transmitts a frame. In last bit of intermission, recessive bit is detected (no other unit is attempting to transmitt frame) and there is no available frame for transmission signalled by TX arbitrator. Protocol control FSM issues ***set_idle*** command and unit becomes idle.
Receiver	Transmitter	Unit receives a frame. In last bit of intermission, available frame for transmission is signalled by TX arbitrator. Protocol control FSM signals ***set_transmitter*** and unit becomes transmitter of frame from Validated TXT buffer.
Receiver	Idle	Unit receives a frame. In last bit of intermission, there is no available frame for transmission signalled by TX arbitrator, recessive bit is monitored (no other unit is attempting to transmitt frame), then Protocol control FSM issues ***set_idle*** command and unit becomes idle.
Idle, Transmitter, Receiver	Off	Fault confinement FSM signals that unit is bus-off or unit is disabled (SETTINGS[ENA] = '0'). In next sample point, unit becomes "Off".

3.14.3 Fault confinement

File: fault_confinement.vhd

Fault confinement module implements following functionality:

- Transmitt error counter (TEC)/ receive error counters (REC) according to [1].
- Rules for manipulation of TEC and REC.
- Fault confinement state of node (error active, error passive, bus-off).
- Set of special error counters to distuinguish between errors in nominal bit rate and data bit rate.

Fault confinement block diagram is shown in Figure 3.18.

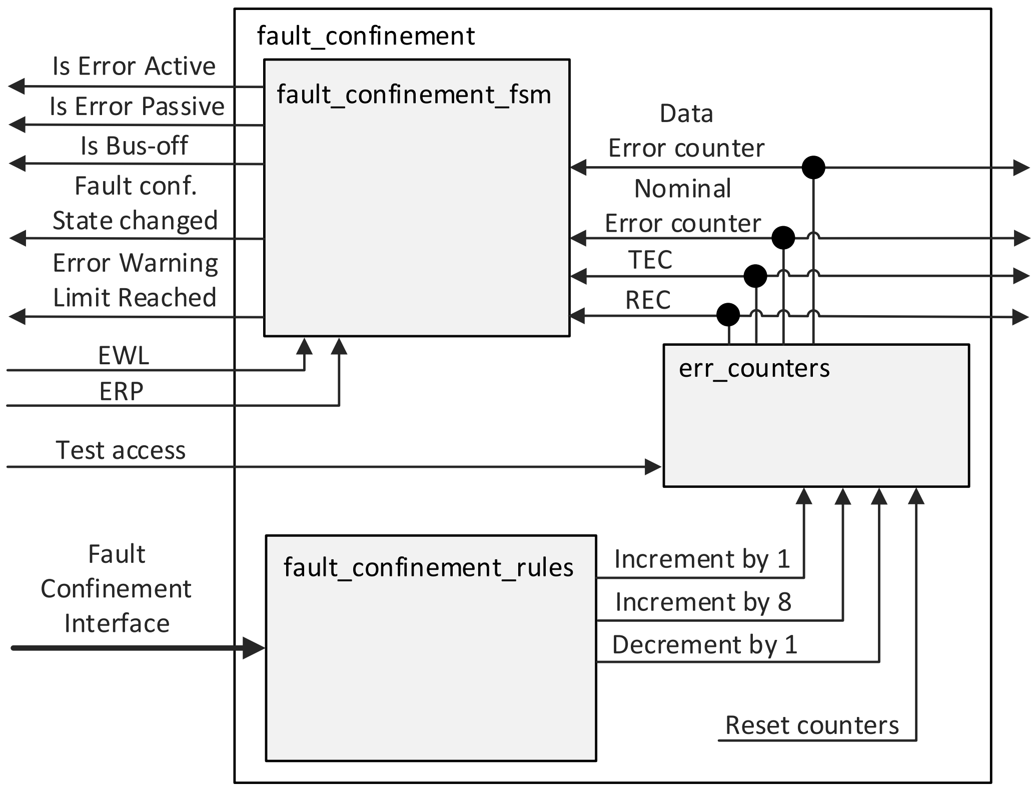

Figure 3.18: Fault confinement block diagram

TEC and REC counters are controlled by Protocol control FSM via interface standardized in 12.1.3.3 of [1]. Detection of special conditions stated in 12.1.4.2 of [1] is realized in Fault confinement rules module. Error counters module implements counters as described in Table 3.28. Counters can be modified from Memory registers via CTR_PRES register when CTU CAN FD is in Test mode (MODE[TSTM] = '1'). Fault confinement state as defined in 12.1.4.1 of [1] is implemented by Fault confinement FSM. State transition diagram of Fault confinement FSM is shown in Figure 3.19. Threshold for Error warning limit detection (EWL) and transition to error passive (ERP) can be configured from Memory registers when device is in Test mode (MODE[TSTM] = '1'). Transition from bus-off to error active is performed after reintegration (***set_err_active*** is signalled by Protocol control FSM). Reffer to 3.14.1 for description of Reintegration counter operation.

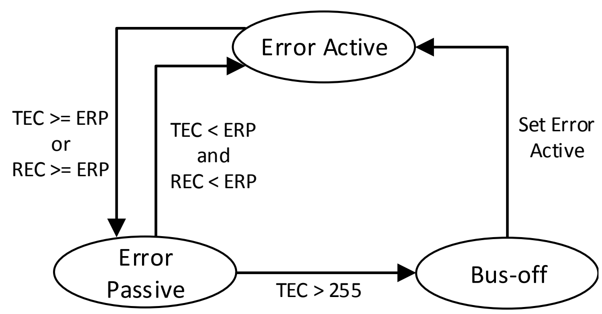

Figure 3.19: Fault confinement FSM

Table 3.28: Error counters

Counter Name	CAN FD standard name	Description
Receive error counter	REC	Incremented, decremented as described below.
Transmitt error counter	TEC	Incremented, decremented as described below.
Nominal error counter	-	Incremented by 1 for each error detected during nominal bit rate. Does not influence fault confinement state of CTU CAN FD.
Data error counter	-	Incremented by 1 for each error detected during data bit rate. Does not influence fault confinement state of CTU CAN FD.

Fault confinement rules

The error counters shall be modified according to the following rules (more than one rule may apply during a given frame transfer):

a) When a receiver detects an error, the receive error counter shall be incremented by 1, except when the detected error was a bit error during the sending of an active error flag or an overload flag.

b) When a receiver detects a dominant bit as a first bit after sending an error flag, the receive error counter shall be incremented by 8.

c) When a transmitter sends an error flag, the transmit error counter shall be incremented by 8.

Exception 1: If the transmitter is error-passive and detects an ACK error because of not detecting a dominant ACK and does not detect a dominant bit while sending its passive error flag.

Exception 2: If the transmitter sends an error flag because a stuff error occurred during arbitration, whereby the stuff bit should have been recessive, and has been sent recessive but is monitored to be dominant. In exception 1 and in exception 2, the transmit error counter remains unchanged.

d) If a transmitter detects a bit error while sending an active error flag or an overload flag, the transmit error counter shall be incremented by 8.

e) If a receiver detects a bit error while sending an active error flag or an overload flag, the receive error counter shall be incremented by 8.

f) Any node shall tolerate up to 7 consecutive dominant bits after sending an active error flag, passive error flag, or overload flag. After detecting 14 consecutive dominant bits (in case of an active error flag or an overload flag) or after detecting 8 consecutive dominant bits following a passive error flag, and after each sequence of additional 8 consecutive dominant bits, every transmitter shall increment its transmit error counter by 8 and every receiver shall increment its receive counter by 8.

g) After the successful transmission of a frame (getting ACK and no error has been detected until EOF is finished), the transmit error counter shall be decremented by 1 unless it was already 0.

h) After the successful reception of a frame (reception without error up to the ACK slot and the successful sending of the ACK bit), the receive error counter shall be decremented by 1, if it was between 1 and 127. If the receive error counter was 0, it shall stay at 0, and if it was greater than 127, then it shall be set to a value between 119 and 127.

3.14.4 Bit stuffing

File: bit_stuffing.vhd

Bit stuffing module implements following functionality:

- Insertion of stuff bits to data transmitted by Protocol control (regular and fixed stuff bits).
- Halting CAN core for one bit time when stuff bit is inserted.
- Counter number of stuff bits modulo 8 for transmission of stuff count field.
- Insertion of stuff bit in the beginning of stuff count field or CRC field of CAN FD Frame.

Bit stuffing module processes transmitted data by Protocol control in Stuff pipeline stage. Bit stuffing module operates in two modes as described in 3.29. When Bit stuffing is enabled, it inserts bit of opposite polarity to transmitted bit stream based on Bit stuffing mode. Data are processed by Bit stuffing module with one clock cycle delay (output is registered). When Bit stuffing module is disabled, it propagates data from input to output without inserting stuff bits (but still with one clock cycle delay). Input data are processed in Stuff pipeline stage regardless of the fact if Bit stuffing module is enabled or disabled (Input is not combinatorially bypassed when Bit stuffing module is disabled). Bit stuffing module is enabled only when unit is transmitter of CAN Frame. When unit is receiver, Bit stuffing module is disabled and only propagates recessive bit values from input to output. Bit stuffing module counts number of inserted stuff bits in Regular Bit stuffing mode in counter of stuff bits (this counter is then used in stuff count frame field). A basic sequence of Bit stuffing module operation is described in Table 3.30.

When bus is idle and transmission of frame starts, SOF bit is the first bit which is processed by Bit stuffing module. If unit samples dominant bit during third bit of intermission, bus idle or suspend transmission, this bit is considered as SOF bit (see 10.4.2.2 of [1]). Such a bit is counted as first dominant bit by Bit stuffing module. Bit stuffing module is disabled when unit reaches CRC delimiter frame field. Bit stuffing module is not disabled in last bit of CRC sequence so that stuff bit can be inserted behind the last bit of CRC sequence. When unit loses arbitration (turns receiver), Bit stuffing module is disabled. An example of Bit stuffing module operation during whole frame is shown in Figure 3.20. If an error is detected (error frame is requested by Error detector), Bit stuffing module is disabled. Bit stuffing module is enabled only during fields which shall be coded by bit stuffing as described in [1].

Table 3.29: Bit stuffing modes

Bit stuffing mode	Stuff rule length	Description
Regular	5	When 5 consecutive bits of equal value are processed, bit of opposite value is inserted. Inserted stuff bit counts as first bit of next sequence of 5 equal consecutive bits (bit stuffing is recursive).
Fixed	4	When 4 bits are processed (regardless of their value), a bit of opposite value than last bit of these 4 bits is inserted on output of Bit stuffing module.

Table 3.30: Bit stuffing module operation

Step	Action
1	Bit stuffing module is disabled, there is no transmission / reception in progress by CTU CAN FD. Counter of equal consecutive bits is 1. Bit stuffing module only propagates recessive value to output in Stuff pipeline stage.
2	Transmission starts (unit becomes transmitter), Bit stuffing module is enabled. Length of Stuff rule is configured to 5 by Protocol control FSM.
3	Bit stuffing module processes bits from Protocol control in Stuff pipeline stage. Counter of equal consecutive bits is incremented by 1 for each processed bit of equal polarity (with respect to previous bit). When bit of opposite polarity is processed, counter of equal consecutive bits is set to 1.
4	Counter of equal consecutive bits reaches length of Stuff rule. Instead of propagating processed bit to output, Bit stuffing inserts bit of opposite polarity on output. Bit stuffing module halts to Protocol control. Protocol control remains halted for one bit. Counter of stuff bits is incremented by 1.
5	After one bit time for which Protocol control was halted, it continues in transmission. Bit stuffing module continues in processing data transmitted by Protocol control. Counter of equal consecutive bits is incremented after insertion of stuff bit to account for recursive behaviour of bit stuffing.
Applies only for CAN FD frames	
6	CAN FD Frame advances to last bit of frame field preceding stuff count frame field. Bit stuffing mode is changed to Fixed. Length of Bit stuffing rule is configured to 4.
7	Stuff bit is inserted by Bit stuffing module in the first bit which is processed in Fixed Bit stuffing mode (First bit of stuff count frame field).
8	Counter of equal consecutive bits is incremented with each processed bit regardless of previous processed bit value. Stuff bit is inserted after each 4 processed bits.

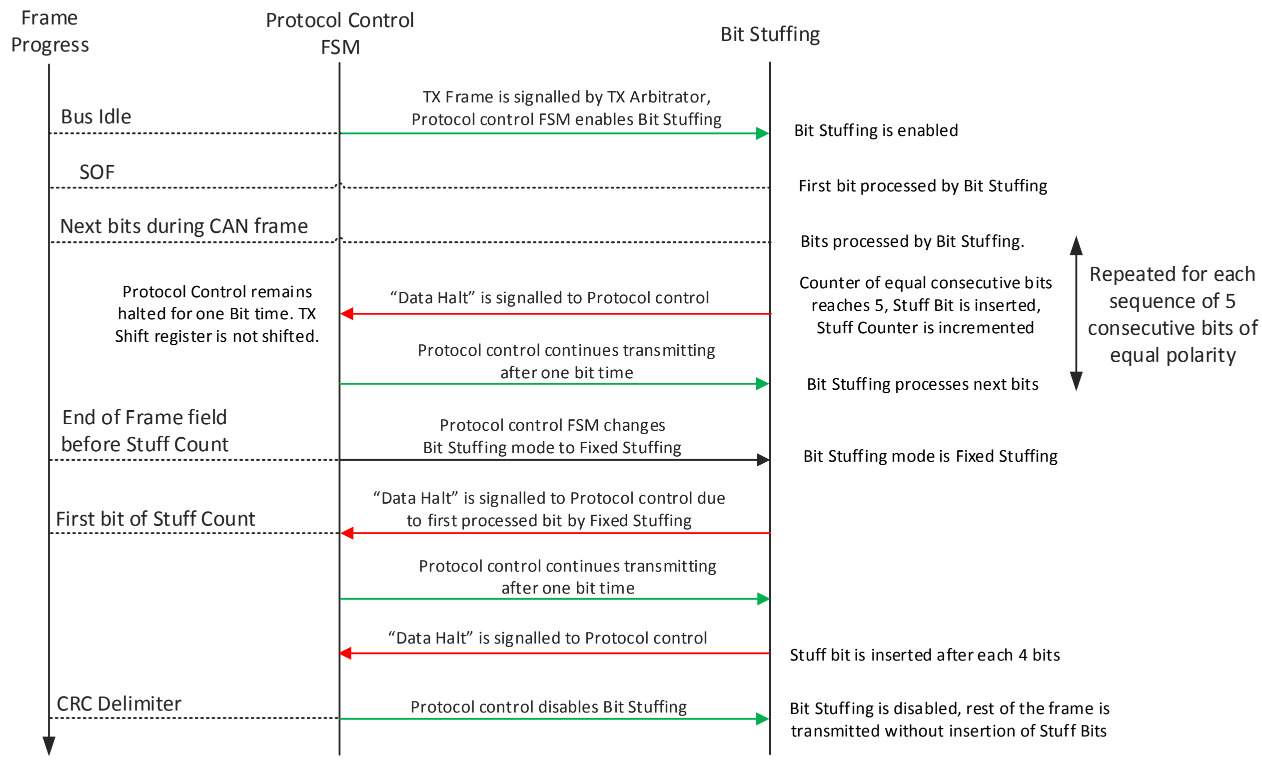

Figure 3.20: Bit stuffing detailed operation

3.14.5 Bit destuffing

File: bit_destuffing.vhd

Bit destuffing module implements following functionality:

- Discard of stuff bits from received data on CAN bus (regular and fixed stuff bits).
- Halting CAN core for one bit time when stuff bit is discarded.
- Holds counter with number of de-stuffed bits modulo 8 for comparison with received stuff count frame field.
- Discarding first fixed stuff bit of CAN FD Frame.
- Detection of stuff error.

Bit destuffing module processes received data on CAN bus as provided by multiplexor in Figure 3.6 in Destuff pipeline stage. Bit destuffing module operates in two modes as described in Table 3.31. Bit destuffing module discards stuff bits according to current Bit destuffing mode. Discarded stuff bit is signalled to Protocol control and it is ignored by Protocol control (not shifted to RX shift register, does not affect Protocol control FSM). Input data are processed with one clock cycle delay (output is registered). When Bit destuffing module is disabled, it only propagates input data to output in Destuff pipeline stage without discarding any bit or detecting stuff error. Bit destuffing module is enabled when unit is transmitter or receiver since transmitter also receives bits transmitted by itself. Bit destuffing module contains counter of discarded stuff bits in Regular mode. This counter is compared with received stuff count field as part of CRC check in CAN FD frames. A basic sequence of operation is shown in Figure 3.32.

When bus is idle, unit is in suspend transmission or third bit of intermission, Bit destuffing module processes dominant bit (which is subsequently evaluated as SOF by Protocol control FSM), then Bit destuffing module considers this bit as first bit in sequence of equal consecutive bits. Bit destuffing module is disabled when unit reaches CRC delimiter frame field. Bit destuffing module is not disabled in last bit of CRC sequence so that stuff bit can be discarded behind the last bit of CRC sequence. When transmission of error frame is requested, Bit destuffing module is disabled. Bit destuffing module is enabled only during fields which shall be coded by bit stuffing as described in [1].

Table 3.31: Bit destuffing modes

Bit destuffing Mode	Destuff rule length	Description
Regular	5	When 5 consecutive bits of equal polarity are processed, next bit is discarded. If value of discarded bit is equal to previous bit, stuff error is detected.
Fixed	4	When 4 bits are processed next bit is discarded, next bit is discarded regardless of values of previous processed bits. If value of discarded bit is equal to previous bit, stuff error is detected.

Table 3.32: Bit destuffing module operation

Step	Action
1	Bit destuffing module is disabled, there is no transmission / reception in progress by CTU CAN FD. Counter of equal consecutive bits is 1. Bit destuffing module only propagates recessive value to output in Destuff pipeline stage.
2	Transmission or reception of frame starts (unit becomes receiver), Bit destuffing module is enabled. Destuff rule length is configured to 5 by Protocol control FSM.
3	Bit destuffing module processes bits in Destuff pipeline stage. Counter of equal consecutive bits is incremented by 1 for each processed bit of equal polarity (with respect to previous bit). When bit of opposite polarity is processed, Counter of equal consecutive bits is set to 1.
4	Counter of equal consecutive bits reaches length of Stuff rule. Following bit is discarded (not processed) and signalled to Protocol control FSM as "Destuffed". Protocol control ignores such a bit and its processing of received data remains halted for one bit time. Number of discarded stuff bits (counter of discarded stuff bits) is incremented by 1.
5	After one bit time for which Protocol control was halted, Bit stuffing module processes next bit. This bit is also processed by Protocol control. Counter of equal consecutive bits is incremented after discarding stuff bit to account for "recursive" behaviour of bit destuffing.
Applies only for CAN FD frames	
8	CAN FD Frame advances to the end of frame field preceding stuff count frame field. Bit destuffing mode is changed to Fixed. Destuff rule length is configured to 4.
9	Stuff bit is discarded by Bit destuffing module in the first bit which is processed in Fixed Bit Stuffing mode (first bit of stuff count frame field).
10	Counter of equal consecutive bits is incremented with each processed bit regardless of previous processed bit value. Stuff bit is discarded after each 4 processed bits.

3.14.6 CAN CRC

File: can_crc.vhd

CAN CRC implements following functionality:

- Calculate CRC sequences according to [1] (for ISO CAN FD) and according to [6] (for non-ISO CAN FD).
- Choose appropriate input and trigger for calculation of CRC sequence.

Block diagram of CAN CRC is shown in Figure 3.21.

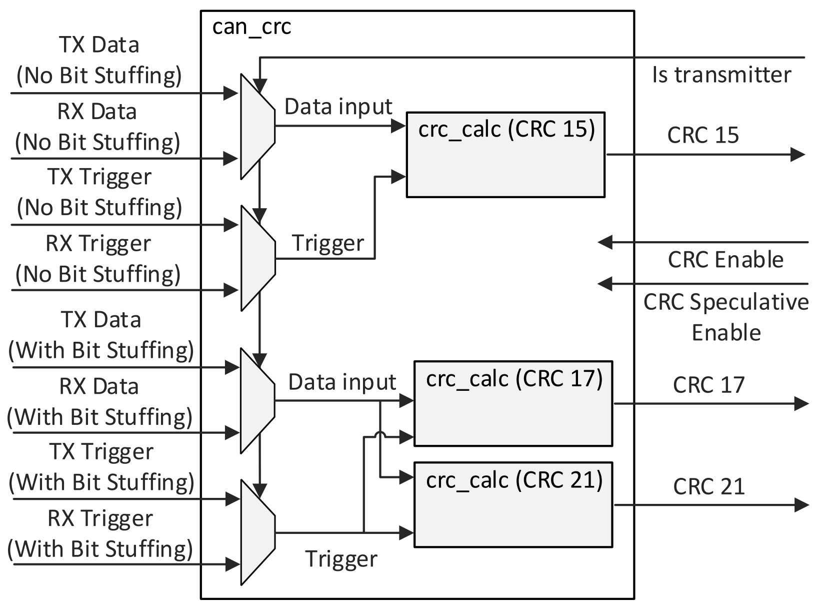

Figure 3.21: CAN CRC block diagram

CAN CRC contains 3 CRC calculation modules (CRC_15, CRC_17, CRC_21). CRC_15 is calculated from data without stuff bits. CRC_17 and CRC_21 are calculated from data with stuff bits inserted. CRC register is preloaded to CRC_INIT_VECTOR upon enabling of CRC calculation (before first bit is processed). Each bit of CAN frame, next step of CRC calculation is executed when according CRC calculation module is enabled. A pseudo-code for CRC calculation is shown in [1].

Data input which is used as input of CRC calculation is different for transmitter/receiver and part of CAN frame when CRC calculation step is executed. During arbitration field, or when speculative enable is used (during bus idle, intermission or suspend transmission), CRC is calculated from received data as there can be multiple units transmitting on the bus at once and correct value (when bus has settled in sample point) must be used for calculation. After arbitration field (when only one unit on the bus remained transmitter), transmitter calculates CRC from transmitted data and receivers calculate CRC from received data. Calculation step from transmitted data is shown in Figure 3.22 and from received data is shown in 3.23.

After arbitration field, source of data for CRC calculation changes from transmitted to received data. Pipeline stage during which next step of CRC calculation is executed differs based on source of input data (if received data are used, input data are not valid before sample point) as described in Table 3.33. When CRC_17/CRC_21 execute CRC calculation step from stuffed/destuffed bit, CRC_15 remains unchagned (according trigger signal is gated). CRC calculation step can be enabled by means of two enable signals: Regular enable and Speculative enable. Meaning of these two signals is explained in Table 3.34.

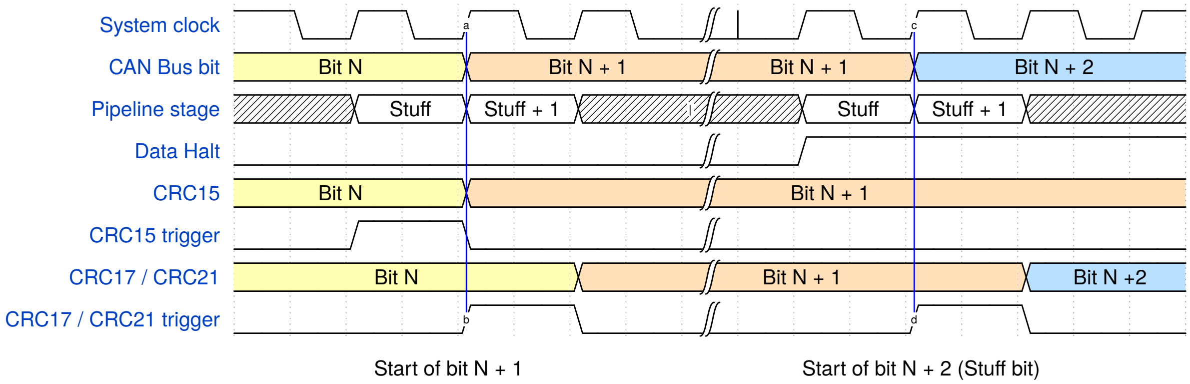

Figure 3.22: CRC calculation - TX Data stream

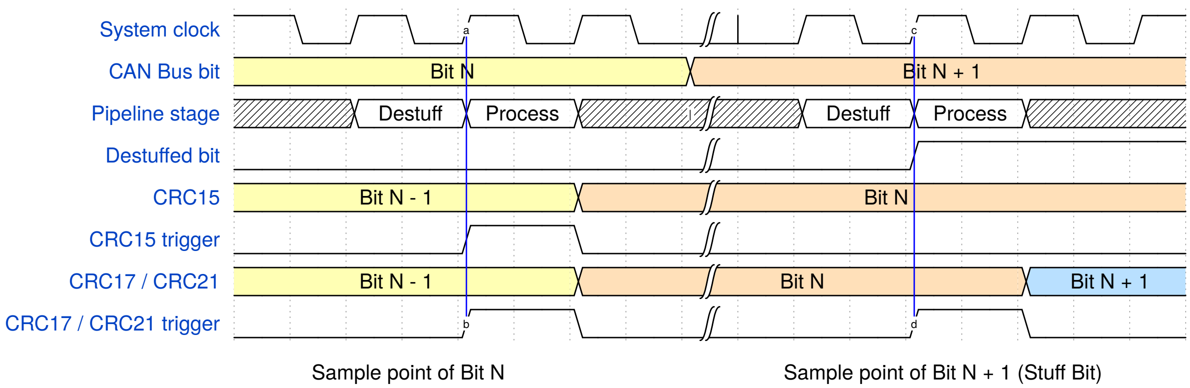

Figure 3.23: CRC calculation - RX Data stream

Table 3.33: CAN CRC calculation

CRC module	Data stream	Data input for CRC calculation	Pipeline stage when calculation step is executed
CRC_15	TX	Transmitted data on output of Protocol control.	Stuff
	RX	Received data on input of Protocol control.	Process
CRC_17	TX	Transmitted data on output of Bit stuffing module.	Stuff + 1 clock cycle
	RX	Received data on input of Bit destuffing module.	Process
CRC_21	TX	Transmitted data on output of Bit stuffing module.	Stuff + 1 clock cycle
	RX	Received data on input of Bit destuffing module.	Process

Table 3.34: CAN CRC enable signals

CAN CRC Enable signal	Description
Regular enable	When CRC module is enabled by regular enable signal, it executes next step of calculation in according pipeline stage regardless of input data value to be processed. This enable signal is used during CAN frame fields from SOF until end of data field.
Speculative enable	When CRC module is enabled by speculative enable signal, it executes next step of calculation in according pipeline stage only when input data value to be processed is dominant (logic 0) and recessive value is ignored. Speculative enable is used in suspend transmission, last bit of intermission and bus idle when dominant value is sampled and this value is interpreted as SOF by Protocol control (as this bit needs to be already taken into account for CRC calculation).

3.14.7 Trigger multiplexor

File: trigger_mux.vhd

Trigger multiplexor implements following functionality:

- Gating of trigger signals (clock enables for pipeline stages)

Trigger multiplexor creates trigger signals for other blocks within CAN core from trigger signals generated by Prescaler as described in Table 3.35. See 3.20.7 on how are trigger signals generated by Prescaler.

Table 3.35: Trigger signals

Trigger Name	Pipeline stage	Description
Protocol control TX Trigger	Stuff	Used to shift TX shift register in Protocol control. Gated when there is stuff bit inserted, this corresponds to halting Protocol control for 1 bit time as described in Table 3.30
Protocol control RX Trigger	Process	Used to shift RX shift register in Protocol control, update of Protocol control FSM state, manipulation of Control counter and Retransmitt Counter. Gated when stuff bit is discarded, this corresponds to halting Protocol control for 1 bit time as described in Table 3.32.
Bit Stuffing Trigger	Stuff	Used for processing of transmitted data by Bit stuffing module.
Bit Destuffing Trigger	Destuff	Used for processing of received data by Bit destuffing module.
CRC TX WBS Trigger	Stuff + 1 clock cycle	Used to enable CRC calculation step for CRC_17 / CRC_21 when CRC calculation step is executed from transmitted data.
CRC TX NBS Trigger	Stuff	Used to enable CRC calculation step for CRC_15 when CRC calculation step is executed from transmitted data.
CRC RX WBS Trigger	Process	Used to enable CRC calculation step for CRC_17 / CRC_21 when CRC calculation step is executed from received data.
CRC RX NBS Trigger	Process	Used to enable CRC calculation step for CRC_15 when CRC calculation step is executed from received data.

3.14.8 Bus traffic counters

File: bus_traffic_counters.vhd

Bus traffic counters contains two 32-bit counters (TX frame counter and RX frame counter). TX frame counter counts succesfully transmitted frames (without error frame or arbitration lost) and is incremented by 1 for each such transmitted frame. RX frame counter counts succesfully received frames (without error frame) and is incremented by 1 for each such a frame. If unit is transmitter in Loopback mode (it also receives frame transmitted by itself), both counters are incremented upon succesfull transmission/reception. In such case, TX frame counter is incremented when transmitted frame is considered valid and RX frame counter is incremented when received is considered valid as defined in 10.7 of [1]).

Both counters can be erased by SW via COMMAND[TXFRCRST] and COMMAND[RXFRCRST] register. Value of traffic counters can be read out from TX_FR_CTR and RX_FR_CTR registers. Bus traffic counters are instantiated only when ***sup_traffic_counters=true***. When Bus traffic counters are not instantiated, access to TX_COUNTER and RX_COUNTER registers are reserved and writes to COMMAND[TXFRCRST] and COMMAND[RXFRCRST] have no effect.

3.15 RX buffer

File: rx_buffer.vhd

RX buffer implements following functionality:

- Storing frame to FIFO memory as CAN frame progresses.
- Count number of stored frames in FIFO.
- Provide read interface for Memory registers.
- Abort storing of CAN frame in case of an error frame request or overrun.

Block diagram of RX Buffer is shown in Figure 3.24.

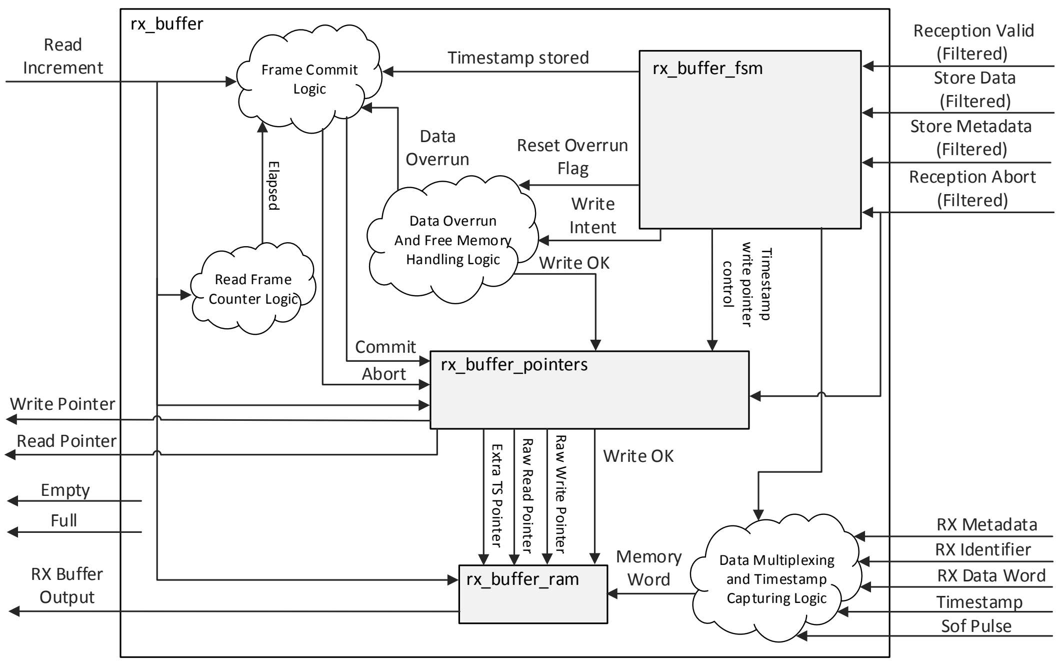

Figure 3.24: RX Buffer block diagram

RX buffer contains FIFO memory (details of actual RAM memory are described in 3.7). Size of RX buffer memory is configurable by ***rx_buffer_size*** generic between 32 and 4096 32-bit memory words. Lower limit on size of RX buffer RAM is imposed to be able to store at least 1 CAN FD frame with 64 byte data payload. Format of CAN FD frame within the memory is described in 3.10 and visualized in Figure 3.25. Size of CAN frame within RX buffer memory spans from 4 to 20 32-bit memory words. Remote frames and frames with no data field span 4 memory words (Metadata, Identifier, Timestamp upper and Timestamp lower). Each next 4 bytes of data field span one memory word. Longest frame with 64 data bytes spans 20 memory words (Metadata, Identifier, Timestamp upper, Timestamp lower and 16 data words).

RX frame is stored to FIFO by means of Storing protocol which is described in 3.15.1. RX Frame is read from FIFO by means of Reading protocol which is described in 3.15.4. RX buffer contains pointers to FIFO which are described in detail in Table 3.36. RX buffer can by flushed by issuing Release receive buffer command (writing logic 1 to COMMAND[RRB]). In such case, all pointers are reset to zero as well as counter of stored frames (see 3.15.4). If Release receive buffer

Table 3.36: RX Buffer pointers

Pointer	Incremented by 1	Pre-loaded	Pre-load value
Raw write pointer	When a word is written to RX buffer RAM (Metadata, Identifier, Timestamp or Data word)	When Reception abort command is issued or, Reception valid command is issued and Overflow occured before in the frame.	Commited write pointer
Commited write pointer	-	When frame is commited.	Raw write pointer
Timestamp write pointer	During storing of Timestamp lower word.	When Raw write pointer points to Lower timestamp word of frame which is actually being stored.	Raw write pointer
Read pointer	When a word is read from RX buffer.	-	-

command is issued by SW during storing of CAN frame, overrun flag is set, and upon the end of actual frame this frame is discarded, and Raw write pointer is reset to value of previous Comited write pointer.

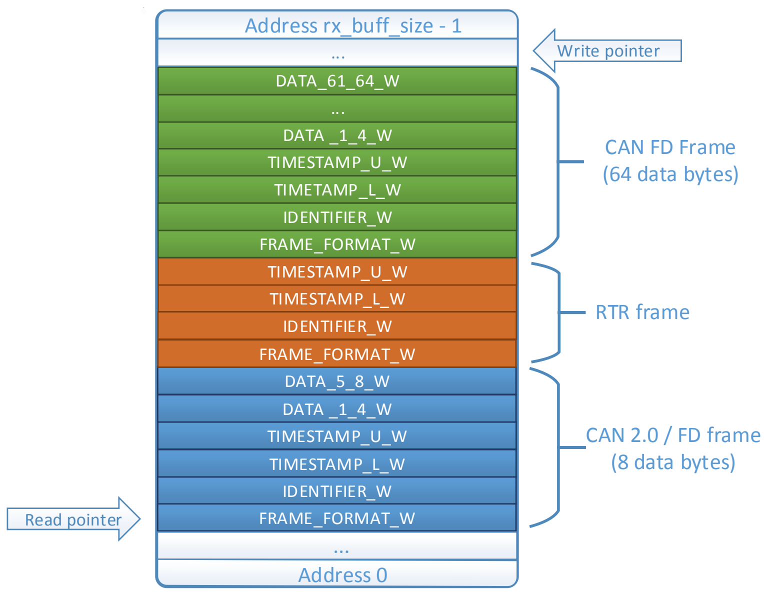

Figure 3.25: RX Buffer memory format

3.15.1 Storing protocol

Protocol control FSM forms “Master” side of Storing protocol and it issues commands which are described in Table 3.37. Commands from Protocol control FSM are filtered by Frame filters before being connected to RX buffer. Commands pass CAN fame filters when RX frame matches CAN frame filters as described in 3.16. If received frame does not match CAN frame filters, commands are gated and does not reach RX buffer within current CAN frame. RX buffer FSM forms “Slave” side of this protocol, it receives commands and reacts upon them. State transition diagram of RX buffer FSM is shown in Figure 3.28. Commands are issued by Protocol control FSM continously as reception of CAN frame progresses. Commands are issued by Protocol control FSM when unit is receiver of a frame, or when Loopback mode

(SETTINGS[ILBP] = '1') is enabled. When unit is transmitter and Loopback mode is disabled, commands are not issued to RX buffer (CAN frame is not being stored). An example of Storing protocol is shown in Figures 3.15.1 and Figure 3.27. Storing protocol is described in Table 3.38.

During storing of CAN frame, this frame can't be read out by SW via Memory registers. When frame is succesfully received without error frame or overrun (last bit of EOF field), it is commited to RX buffer and it becomes available for SW.

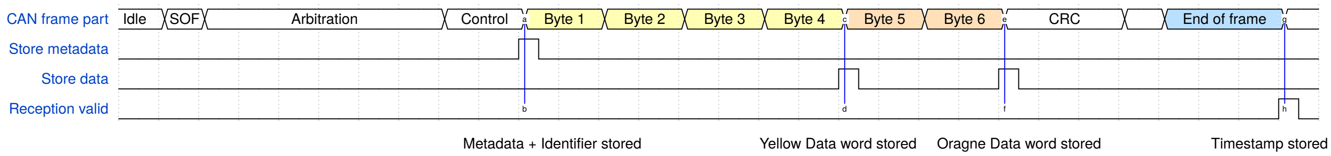

Figure 3.26: RX buffer storing protocol - succesfull reception

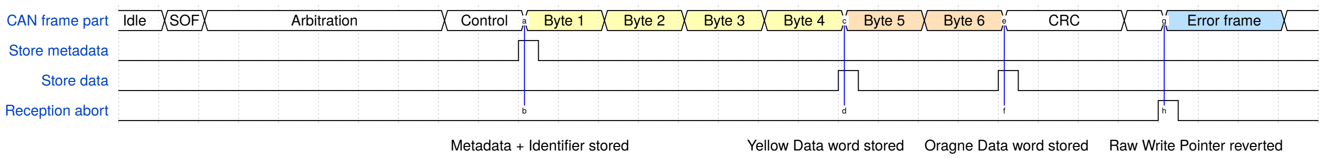

Figure 3.27: RX buffer storing protocol - Error frame

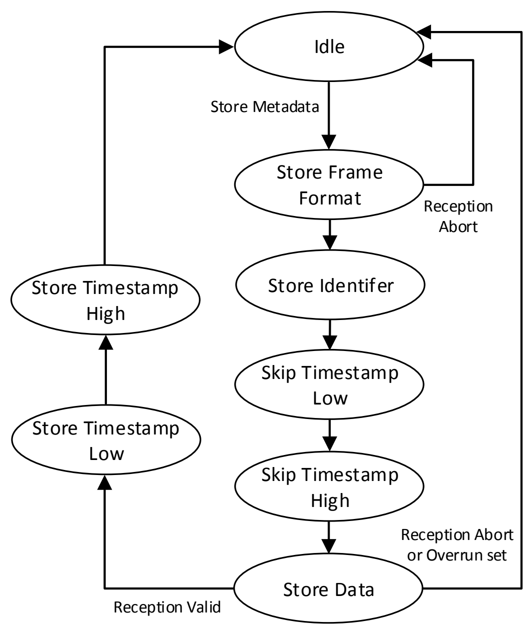

Figure 3.28: RX buffer FSM

Table 3.37: RX buffer commands

Command	Issued in CAN frame part	Action performed	Source of stored information to RX buffer RAM
Store metadata	At the end of data length code field.	Store Metadata word, Identifier word and zeroes to Timestamp words.	Frame metadata and identifier from capture registers in RX shift register in Protocol control.
Store data	After multiple of 4 bytes of data field elapsed and at the end of data field.	Store Data word (4 bytes).	RX shift register in Protocol control.
Reception valid	In sample point of last bit of EOF field.	Timestamp is stored and afterwards CAN Frame is commited to memory.	Timestamp capture register.
Reception abort	When error frame is transmitted.	Frame storing is aborted, Raw write pointer is reverted to last Commited write pointer.	-

Table 3.38: RX buffer storing protocol - detailed description

Step	Action
1	Reception of CAN frame starts. If received frame timestamp is configured to be captured at SOF (RX_SETTINGS[RTSOPT]), it is captured to Timestamp capture register.
2	Identifier is received to RX shift register in Protocol control and stored to dedicated capture register. Metadata are stored to dedicated capture registers in Protocol control. See 3.14.1.
3	At the end of control field, it is already clear whether unit is transmitter or receiver. It can no longer happend that a word will be stored to RX buffer and unit will turn receiver due to losing arbitration. Protocol control FSM issues Store metadata command if unit is receiver or in Looback mode.
4	RX buffer FSM stores Metadata to Frame format word, received CAN identifier to Identifier word and zeroes to Timestamp words during 4 consecutive clock cycles (during each cycle 1 word is stored). Raw write pointer is incremented by 1 during each of these cycles. When Raw write pointer points to Lower Timestamp word, it is captured to Timestamp write pointer. After this step Raw write pointer points to first Data word.
5	Data field of CAN frame starts. After each 4 bytes are received, Protocol control FSM issues Store data command. These 4 bytes are stored to RX buffer RAM in single word and Raw write pointer is incremented.
6	At the end of last bit of data field, Protocol control FSM issues Store data command if the length of data field is not multiple of bytes. Remaining bytes are stored to RX buffer RAM and Raw write pointer is incremented.
7	CAN frame progresses to EOF field. In sample point of EOF field, received frame is considered valid (assuming no error frame). Protocol control FSM issues Reception valid command. If received frame timestamp shall be taken in EOF, it is captured to Timestamp capture register.
8	Timestamp is stored from Timestamp capture register (by means of Timestamp write pointer), to Timestamp low and Timestamp high memory words of RX Buffer.
9	If overrun condition did not occur during storing of current frame, frame is commited to memory, Raw write pointer moves to Commited write pointer and number of frames in RX buffer (Frame counter) is incremented. If overrun condition or Release receiver buffer command did occur during storing of current frame, frame is not commited to memory, Raw Write Pointer is reverted to Commited Write Pointer and number of frames in RX Buffer remains unchanged.

3.15.2 Overrun flags

RX Buffer maintains two overrun flags: User overrun flag and Internal overrun flag. Both overrun flags are set when RX buffer FSM intents to store a word to RX buffer RAM, and RX buffer RAM is full (Overrun condition). Internal overrun flag is reset at the end of CAN frame. User overrun flag is reset by SW writing COMAND[CDO]=1. When frame is error-free (no error frame), but overrun condition occured at some point before in the frame (Internal overrun flag is set), frame is discarded (not commited) and Write pointers are manipulated as if Reception abort command was received.

3.15.3 Received frame timestamp

RX buffer implements Timestamping of received frames. Such a timestamp is created by sampling ***timestamp*** input of CTU CAN FD in sample point of SOF or EOF bits (configured by RX_SETTINGS[RTSOP]). In sample point of these bits, ***timestamp*** is captured to capture register and stored to RX bufer RAM from capture register at the end of CAN frame . As position of Timestamp memory words within RX buffer RAM is lower than Data words, when timestamp is about to be stored (in sample point of EOF), Raw write pointer is pointing one memory word behind last word of CAN frame. Due to this reason, Raw write pointer can't be used to store received frame timestamp and dedicated Timestamp write pointer is used. This pointer is loaded by RX buffer FSM to point to first Timestamp word in RX Buffer RAM.

3.15.4 Reading protocol

CAN frame from RX buffer is read out by SW word by word by reading RX_DATA register. There are two modes (distuiguished by MODE[RXBAM] bit) in which RX buffer can be read:

- Automated mode (default) - SW must read via 32 bit accesses. When RX_DATA register is read, RX buffer read pointer automatically moves to next word.
- Manual mode - SW can read via 8/16/32 bit accesses. When RX_DATA register is read, RX buffer read pointer is NOT moved automatically to next word. To move RX buffer to next word, use must issue COMMAND[RXRPMV]). This mode can be used in systems which are incapable of executing "atomic" 32 bit accesses, and require reading by 8 or 16 bit accesses.

Behavior of RX buffer during reads is described in 3.39. Read pointer is incremented after each word is read, either manually or automatically (an exception to this rule is when FIFO is empty). RX buffer supports single reads (Read indication asserted for one clock cycle) and also continous burst read (Read indication asserted for several consecutive clock cycles). Since RX buffer RAM has one clock cycle delay on data output, RAM read address is speculatively multiplexed between Read pointer and Read pointer + 1 as shown in Figure 3.29. Due to this speculation RX Buffer read pre-feteches data from next memory word instead of memory word given by Read pointer. This speculation is executed to support burst read.

Table 3.39: RX buffer - read protocol

Step	Action
1	Read pointer points to Frame Format word of most recently stored frame in RX buffer. Output of RX buffer RAM contain Frame Format word.
2	SW reads from RX_DATA register (Frame Format word). Auxiliarly counter is loaded to value of RWCNT. Read pointer is incremented by 1.
3	SW now knows value of RWCNT (number of remaining words in currently read frame). SW reads from RX_DATA register RWCNT times. Read Pointer is incremented by 1 and auxiliarly counter is decremented by 1 after each of these reads.
4	During last read (when auxiliarly counter transits from 1 to 0), Frame counter is decremented by 1.

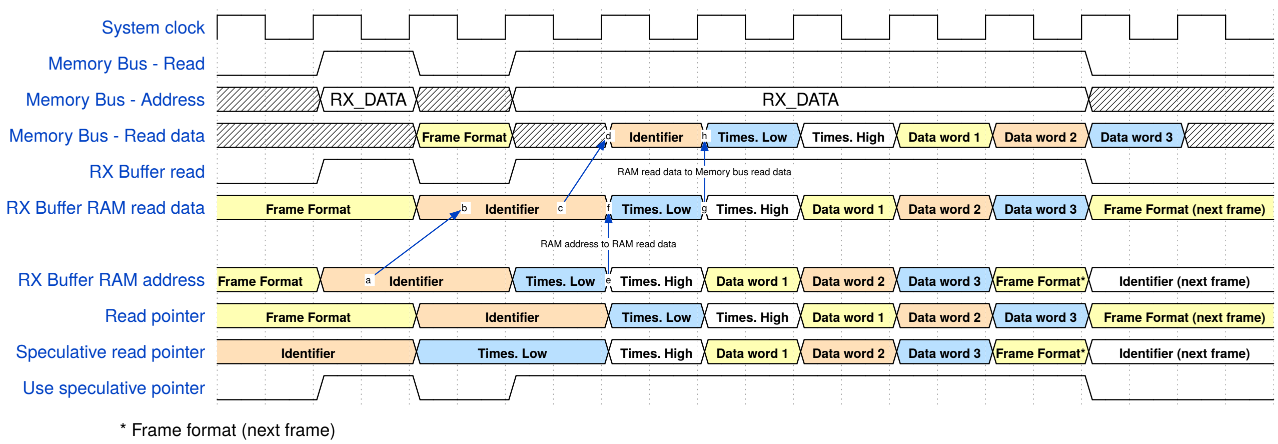

Figure 3.29: RX Buffer - Read pointer speculation

RX buffer contains Frame counter (readable by SW via RX_STATUS[RXFRC]). Frame counter holds amount of CAN frames actually stored in RX buffer. Frame counter is incremented by 1 when a frame is commited to RX buffer. Since RX buffer RAM is read word by word, RX buffer counts each read word from Memory registers and decrements Frame counter only when whole frame was read. If new frame is committed at the same time as last word of another frame is read, Frame counter remains unchanged. Manipulation with Frame counter is described in Table 3.40.

Table 3.40: Frame counter handling

Step	Action
1	Frame counter is 0. CAN frame is being received and stored to RX buffer RAM.
2	Frame ends and it is commited to RX Buffer, Frame Counter is incremented to 1.
3	Read Pointer points to the first word of CAN frame (Frame format word). Memory registers issue a read from RX Buffer. RX Buffer RAM output contains Frame Format word. RX Buffer loads value of RWCNT (Read word count) to an auxiliarly counter. Frame counter remains 1 and Read Pointer increments and points to Identifier word.
4	Memory registers issue RWCNT - 1 number of reads from RX Buffer and Read pointer increments by 1 on each read. Auxiliarly register decrements by 1 each read.
5	Memory registers issue a read from RX Buffer (reading last word of CAN frame). Auxiliarly register indicates that last word of frame is read and Frame counter is decremented by 1.

3.15.5 RX Buffer RAM

If ***target_technology*** = 0 (ASIC), clock for RX buffer RAM are gated if RX buffer RAM is not written nor read.

3.16 Frame Filters

File: frame_filters.vhd

Frame filters implement following functionality:

- Filter RX frames before storing to RX buffer based on CAN Identifier.
- Gate RX buffer commands when identifier does not pass Frame Filters.

Frame filters implement two types of filters: Bit filter and Range filter. There are three instances of Bit filter (A, B, C) and one instance of Range filter. Each instance is selectively synthesizable by ***sup_filt_A/B/C*** or ***sup_range*** generics. If filter is not synthesized, it is not taken into account during frame filtering. When no Frame filter is synthesized, all RX frames are stored to RX buffer and no frame is filtered out.

CAN frame passes Frame filters if received identifier passes at least one filter (logical OR). Filters are considered only when Acceptance filter mode is enabled (MODE[AFM] = '1'). When Acceptance filter mode is disabled, no received frames are filtered out.

Each filter can be configured to accept only given combination of Frame type and Identifier type via FILTER_CONTROL register. If received Frame type and Identifier type does not match accepted Frame type and Identifier type, it does not pass filter even if its identifier is matching. For description of filter operation reffer to 3.41 and 3.42. Note that logic equations within these tables follow C-like syntax with "&" meaning "logical AND" and "&&" meaning "boolean AND". (A,B) means concatenation of vectors A and B where A is MSB. Note that accepted combinations of Accepted Frame types / Identifier are one-hot coded in FILTER_CONTROL register and therefore any combination of these settings can be used.

Table 3.41: Bit filter operation

Accepted Frame types / Identifier types	Received Identifier type	Condition for frame to pass RX_BASE = Received base identifier, RX_EXT = Received identifier extension, FILTER_X_MASK (A,B,C) = Filter mask, FILTER_X_VALUE (A,B,C) = Filter value, FR_TYPE = Received frame type (corresponds to FDF bit), ID_TYPE = Received identifier type (corresponds to IDE bit)
CAN 2.0 / Base	Base	[(RX_BASE & FILTER_MASK(28:18)) == (FILTER_BASE(28:18) & FILTER_MASK(28:18))] && (FR_TYPE == CAN 2.0) && (ID_TYPE == Base)
	Extended	not accepted
CAN FD / Base	Base	[(RX_BASE & FILTER_MASK(28:18)) == (FILTER_BASE(28:18) & FILTER_MASK(28:18))] && (FR_TYPE == CAN FD) && (ID_TYPE == Base)
	Extended	not accepted
CAN 2.0 / Extended	Base	not accepted
	Extended	[(RX_BASE & FILTER_MASK(28:18)) == (FILTER_BASE(28:18) & FILTER_MASK(28:18))] && [(RX_EXT & FILTER_MASK(17:0)) == (FILTER_BASE(17:0) & FILTER_MASK(17:0))] && (FR_TYPE == CAN FD) && (ID_TYPE == Extended)
CAN FD / Extended	Base	not accepted
	Extended	[(RX_BASE & FILTER_MASK(28:18)) == (FILTER_BASE(28:18) & FILTER_MASK(28:18))] && [(RX_EXT & FILTER_MASK(17:0)) == (FILTER_BASE(17:0) & FILTER_MASK(17:0))] && (FR_TYPE == CAN FD) && (ID_TYPE == Extended)

Table 3.42: Range filter operation

Accepted Frame types / Identifier types	Received Identifier type	Condition for frame to pass RX_BASE = Received base identifier, RX_EXT = Received identifier extension, FILTER_RAN_LOW = Lower filter threshold, FILTER_RAN_HIGH = Upper filter threshold, FR_TYPE = Received frame type (corresponds to FDF bit), ID_TYPE = Received identifier type (corresponds to IDE bit)
CAN 2.0 / Base	Base	(RX_BASE >= FILTER_RAN_LOW(28:18)) && (RX_BASE <= FILTER_RAN_LOW(28:18) && (FR_TYPE == CAN 2.0) && (ID_TYPE == Base)
	Extended	not accepted
CAN FD / Base	Base	(RX_BASE >= FILTER_RAN_LOW(28:18)) && (RX_BASE <= FILTER_RAN_LOW(28:18)) && (FR_TYPE == CAN FD) && (ID_TYPE == Base)
	Extended	not accepted
CAN 2.0 / Extended	Base	not accepted
	Extended	((RX_BASE, RX_EXT) >= FILTER_RAN_LOW(28:0)) && ((RX_BASE, RX_EXT) <= FILTER_RAN_LOW(28:0)) && (FR_TYPE == CAN 2.0) && (ID_TYPE == Extended)
CAN FD / Extended	Base	not accepted
	Extended	((RX_BASE, RX_EXT) >= FILTER_RAN_LOW(28:0)) && ((RX_BASE, RX_EXT) <= FILTER_RAN_LOW(28:0)) && (FR_TYPE == CAN FD) && (ID_TYPE == Extended)

3.17 TXT buffer

File: txt_buffer.vhd

TXT buffer implements following functionality:

- Stores single CAN frame for transmission in internal RAM memory.
- Manages access from HW and SW to this RAM memory.
- Provide status of frame transmission for SW.

Number of TXT buffers in CTU CAN FD is configurable at synthesis time via ***txt_buffer_count*** top level generic. Each TXT buffer contains 1 RAM memory. Each TXT buffer RAM is accessed by SW via Memory registers as described in [1]. SW stores CAN frame to TXT buffer. For SW, TXT buffer RAM is write-only. TXT buffer RAM is also accessed by Protocol control FSM and TX arbitrator. TX arbitrator reads parts of CAN frame as part of TXT buffer valiation. Protocol control FSM reads data words from TXT buffer RAM as part of their transmission on CAN bus. For Protocol control and TX arbitrator, TXT buffer is read-only. TXT buffer is managed by FSM which is shown in Figure 3.30. CAN frame format within TXT buffer is the same as within RX buffer and it is described within 3.10. Each TXT buffer in CTU CAN FD has its own priority (configured by SW in TX_PRIORITY register). Based on priority, TX arbitratror selects TXT buffer which will be used for transmission (see 3.18).

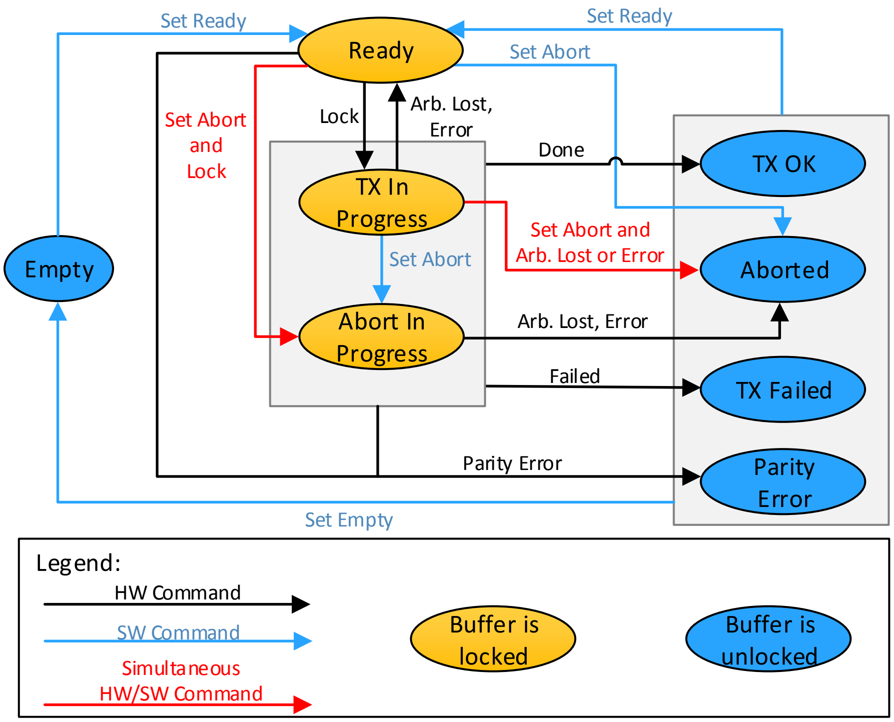

Figure 3.30: TXT buffer FSM

3.17.1 TXT buffer commands

Two types of commands can be issued to TXT buffer: SW commands and HW commands. SW commands are issued by SW access to TX_COMMAND register. HW commands are issued by Protocol control FSM. Both command types are described in Table 3.43. Since operation of SW and Protocol control FSM are not synchronized, HW and SW commands can occur simultaneously. Behavior in such cases is described in Table 3.44. If SW command is applied to TXT buffer FSM in state for which it is not valid, it has no effect. HW command is never applied in TXT buffer FSM state for which it is not valid (there are design assertions to check that).

Table 3.44: TXT buffer simultaneous HW/SW commands

HW Command	SW Command	TXT Buffer state	Result
Lock	Set abort	Ready	TXT buffer becomes "Abort in progress", Protocol control attempts to do do single transmission from thix TXT buffer.
Unlock - done	Set abort	TX in Progress	TXT buffer is unlocked and becomes "TX OK" since transmission is successfull.
Unlock - failed	Set abort	TX in Progress	TXT buffer is unlocked and becomes "TX failed" since transmission failed.
Unlock - arbitration lost, error	Set abort	TX in Progress	TXT buffer is unlocked and becomes "Aborted". No more transmissions are attempted from this TXT buffer. In this case SW command has priority over HW command. Due to this, transmissions will not go on from thix TXT buffer.

Table 3.43: TXT buffer commands

Command name	Command type	Valid TXT buffer States	When is command issued
Set ready	SW	Empty, TX OK, Aborted, TX failed	SW stored CAN frame to TXT buffer RAM and wants to transmit this frame.
Set empty	SW	TX OK, Aborted, TX failed	SW wants to move TXT buffer to its inital state after reset.
Set abort	SW	Ready, TX in progress, Abort in progress	SW wants to abort transmission of a frame whose transmission has been previously requested by Set ready command.
Lock	HW	Ready	Protocol control FSM starts transmitting frame from TXT buffer.
Unlock - done	HW	TX in progress, Abort in progress	Protocol control FSM successfully transmitted frame from TXT buffer.
Unlock - error	HW	TX in progress, Abort in progress	Error frame occurred, Protocol control stops transmitting from TXT buffer.
Unlock - arbitration lost	HW	TX in progress, Abort in progress	Arbitration was lost, Protocol control stops transmitting from TXT buffer.
Unlock failed	HW	TX in Progress, Abort in progress	A frame was re-transmitted number of times unsucesfully (either arbitration was lost or error frame occurred) and Retransmitt counter reached Retransmitt threshold. Frame transmission will not be attempted anymore.

3.17.2 TXT buffer RAM

File: txt_buffer_ram.vhd

TXT buffer RAM is written by SW (port A) and read by Protocol Control FSM (port B). With regards to accessibility, TXT buffer RAM can be in two states: Unlocked and Locked. TXT buffer FSM states corresponding to Locked and Unlocked state of TXT buffer RAM are demonstrated in Figure 3.30. When TXT buffer is unlocked, it is not acessed by Protocol control (nor TX arbitrator) as there is no frame transmission/validation from this TXT buffer and SW can write to TXT buffer via Memory registers. When TXT buffer is Locked, it was either marked as Ready, or validated by TX arbitrator, or transmission is in progress from this TXT buffer. When TXT buffer is locked, SW can not write to TXT buffer RAM and such writes have no effect.

3.17.3 TXT buffer - Transmission availability

When TXT buffer FSM is in Ready state, it is "Available" for transmission from TX arbitrators point of view. However, if TXT buffer receives Set abort command, it become "Unavailable" for transmission in the same clock cycle as Set abort command is active (***txtb_available*** drops low). In this clock cycle, TXT buffer FSM is still in Ready state and it will move to Aborted (or Abort in progress) in following clock cycle. This combinatorial path from Set abort command to output of TXT buffer is necessary to avoid hazards on TXT buffer selection as explained in 3.18.10.

3.17.4 TXT buffer - Use cases

Table 3.45: TXT Buffer - sucessfull transmission

Step	SW Action	HW Action / State
1	SW fills TXT buffer RAM.	TXT buffer is in Empty state.
2	SW issues Set ready command.	TXT buffer moves to Ready state.
3		TX arbitrator validates TXT buffer for transmission and indicates this to Protocol control. On third bit of intermission or when bus is idle, Protocol control issues Lock command, TXT buffer moves to TX inprogress and Protocol control starts transmission from TXT buffer.
4		Frame transmission ends successfully and Protocol control issues Unlock - done command. TXT buffer moves to TX OK state.
5	SW reads state of TXT buffer and finds out that transmission ended succesfully.	

Table 3.46: TXT buffer - Abort

Step	SW Action	HW Action / State
1	SW fills TXT buffer RAM.	TXT buffer is in Empty state.
2	SW issues Set ready command.	TXT buffer moves to Ready state.
3		TX arbitrator validates TXT buffer for transmission and signals to Protocol control there is a valid TXT buffer for transmission. On third bit of Intermission or when bus is idle, Protocol control issues Lock command, TXT buffer moves to TX in progress. Protocol control starts transmission from TXT buffer.
4	During transmission SW issues Set abort command to TXT buffer.	TXT buffer moves to Abort in progress.
5		If error frame occurs or arbitration is lost, TXT buffer moves to Aborted state. If frame transmission finished succesfully, TXT buffer moves to TX OK state.
6	SW reads state of TXT buffer and finds out whether transmission was aborted or it ended succesfully.	

Table 3.47: TXT buffer - transmission failed

Step	SW Action	HW Action / State
1	SW fills TXT buffer RAM. SW configures retransmitt limit to 5 and enables retransmitt limitation.	TXT buffer is in Empty state.
2	SW issues Set ready command.	TXT buffer moves to Ready state.
3		TX arbitrator validates TXT buffer for transmission and indicates available TXT buffer for transmission to Protocol control. On third bit of intermission or when bus is idle, Protocol control issues Lock command, TXT buffer moves to TX in progress and Protocol control starts transmission from TXT buffer.
4		An error frame occurs or arbitration is lost, Protocol control issues Unlock - error or Unlock - arbitration lost command. TXT buffer moves to state Ready. Retransmitt counter is incremented by 1.
Steps 3-4 repeat until retransmitt counter reaches 5		
5		On 5th retransmission (retransmitt counter = 5), error occurs. Protocol control issues Unlock - failed command. TXT buffer FSM moves to TX failed state.
6	SW reads state of TXT buffer and finds out that transmission failed.	

Table 3.48: TXT buffer - Simultaneous Set abort and Lock

Step	SW Action	HW Action / State
1	SW fills TXT buffer RAM.	TXT buffer is in Empty state.
2	SW issues Set ready command.	TXT buffer moves to Ready state.
3	SW decides to abort transmission of this frame and issues Set abort command.	TX arbitrator validates TXT buffer for transmission and indicates available TXT buffer for transmission to Protocol control. On third bit of intermission or when bus is idle, Protocol control issues Lock command. By coincidence, Set abort command (SW) and Lock command (HW) are active in the same clock cycle. TXT buffer moves to Abort in progress and Protocol control starts transmission from TXT buffer.
4		An error frame occurs or arbitration is lost, Protocol control issues Unlock - error or Unlock - arbitration lost command. TXT buffer moves to state Aborted.
5	SW reads state of TXT buffer and finds out that transmission was aborted.	

3.18 TX arbitrator

File: tx_arbitrator.vhd

TX arbitrator implements following functionality:

- Pick TXT buffer for transmission.
- Load CAN frame metadata and Identifier from TXT buffer and provide them to CAN core for transmission.
- Check parity of Metadata, Identifier and Timestamp words read from TXT Buffer, and signal to TXT Buffer that it contains corrupted data.
- Execute comparison of ***timestamp*** input with transmitted frame timestamp and determine moment of CAN frame transmission.
- Signal to CAN core that CAN frame was validated and can be locked for transmission.
- Hold index of TXT buffer from which CAN core is actually transmitting.
- Detect change of TXT buffer between two consecutive transmissions.

TX arbitrator block diagram is shown in Figure 3.31.

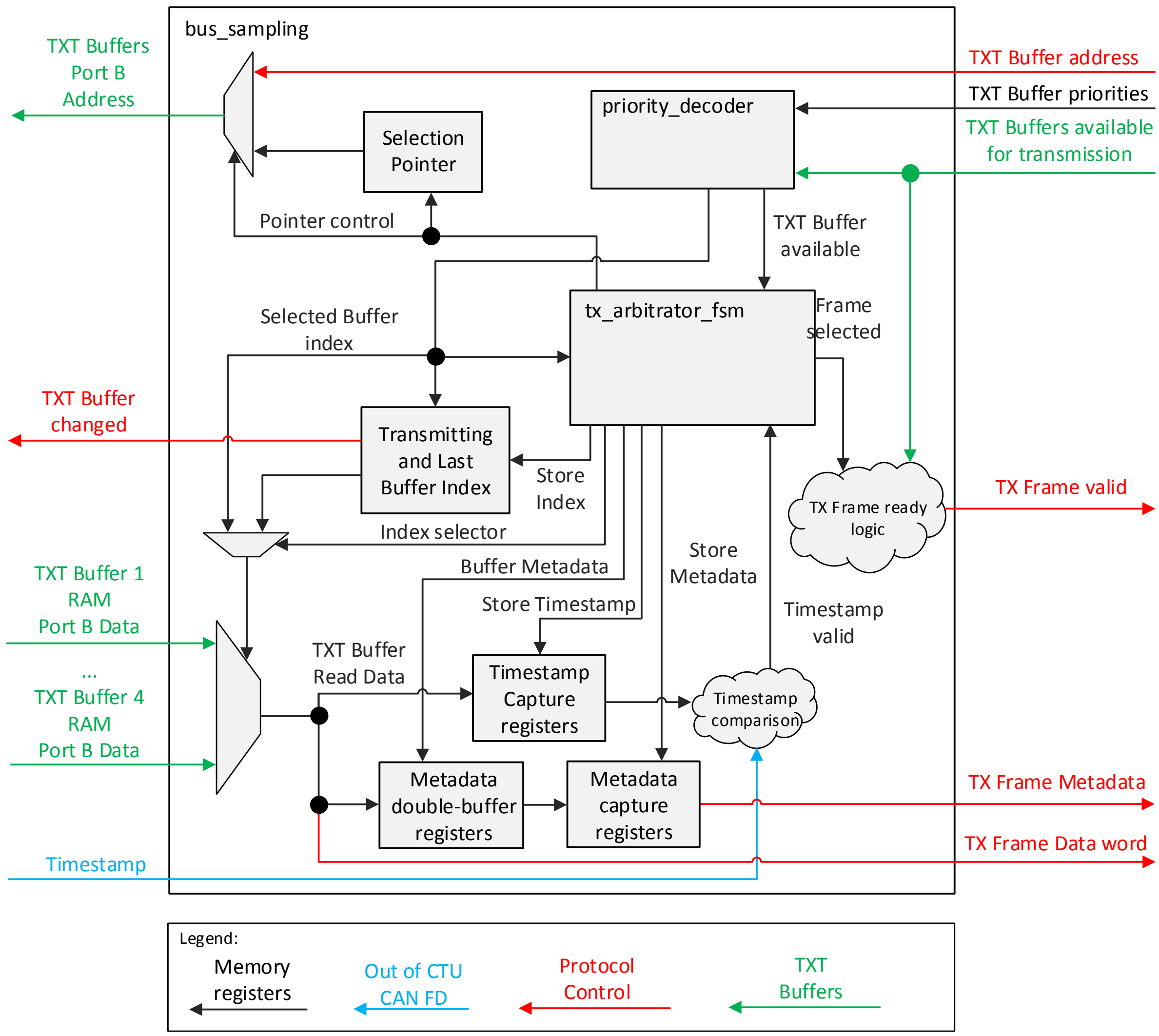

Figure 3.31: TX arbitrator block diagram

3.18.1 TXT buffer validation process

With regards to processing by TX arbitrator, TXT buffer can be in one of states described in Table 3.49. TXT buffer validation process starts when Priority decoder picks highest priority Available TXT buffer (such TXT Buffer becomes "Selected") for transmission (see 3.18.2). Validation process is described in 3.50. An FSM controlling the selection is shown in 3.32. Note each state of TXT buffer FSM which is part of TXT buffer validation lasts for two clock cycles due to wait state. Such wait state is inserted to cover delay of TXT buffer RAM.

If index of Selected TXT buffer changes (due to another higher priority TXT buffer becoming Ready or change of TXT buffers priorities) during validation process or after validation process was finished (TX arbitrator FSM is in Validated state), TXT buffer validation process restarts with newly Selected TXT buffer.

If Validated TXT buffer suddenly becomes Unavailable (due to Set abort SW command), TX arbitrator signals immediately (in the same clock cycle) to Protocol control FSM that there is no Validated TXT buffer (this is done to avoid control hazards on TX frame datapath and it is further explained in 3.18.10) and TX arbitrator FSM moves to Idle state. Several use-cases are explained in 3.51 and 3.52.

When there is Validated TXT buffer, Protocol control FSM issues Lock command during bus idle or third bit of intermission. In such case TX arbitrator goes to Locked state and TXT buffer becomes Used from TX arbitrators point of view (TXT buffer FSM itself goes to TX in progress). Protocol control then transmitts frame from this TXT buffer and upon its end it issues Unlock command. TXT buffer then becomes either Available or Unavailable (see 3.17.1).

If during TXT buffer validation process, TX Arbitrator detects parity error in Metadata, Identifier or Timestamp words, it immediately aborts validation of such TXT buffer, and signals this to TXT Buffer. If TXT Buffer is "Used" (transmission is being executed from it), and TX Arbitrator detects that parity is corrupted on a data word which is being transmitted, it also signals this to TXT Buffer.

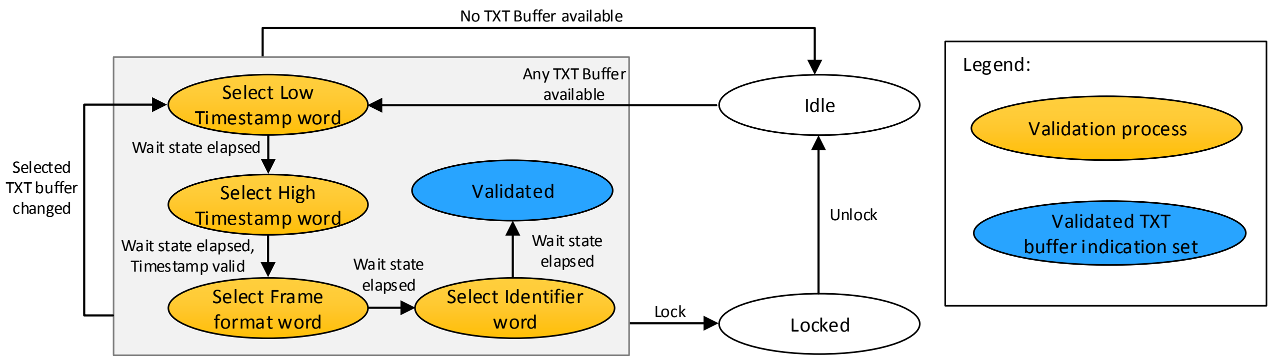

Figure 3.32: TX arbitrator FSM

Table 3.49: TX arbitrator - TXT buffer processing

Filter name	Description
Unavailable	TXT buffer is Unavailable when it is not Available for transmission as defined in 3.17.3. Such a TXT buffer is ignored by TX arbitrator.
Available	TXT buffer is Available when it is Available for transmission as defined in 3.17.3.
Selected	TXT buffer is Selected when it is Available with highest priority out of all Available TXT buffers.
Validated	TXT buffer is Validated when it is Available for transmission, its Timestamp comparison has been executed and Metadata from TXT buffer RAM (Frame format word) has been loaded to capture registers for CAN core.
Used	TXT buffer is Used after CAN core issues Lock command on Validated TXT buffer and is transmitting from this TXT buffer.

Table 3.50: TX arbitrator operation

Step	External action (SW or external components)	HW action
1		No TXT buffer is Available.
2	SW fills TXT buffer 1 and issues Set ready command to this TXT buffer.	TXT buffer 1 FSM goes to Ready state is and therefore Available for TX arbitrator. As this is only TXT buffer which is Available, Priority decoder selects it as highest priority Available TXT buffer.
3		TX arbitrator FSM loads Lower timestamp word from TXT buffer 1 RAM and stores it to auxiliarly register.
4		TX arbitrator FSM loads Upper timestamp word from TXT buffer 1 RAM and executes comparison of ***timestamp*** input and timestamp of CAN frame in TXT buffer 1 (Lower word is in auxiliarly register and Upper word is on output of TXT buffer 1 RAM). When ***timestamp*** is lower than timestamp of CAN frame in TXT buffer 1, TX arbitrator waits, otherwise it proceeds to step 5.
5	***timestamp*** is incrementing (as it counts running time within a system) and it reaches value of CAN frame timestamp in TXT buffer 1.	TX arbitrator notices ***timestamp*** input is now higher than timestamp of CAN frame in selected TXT buffer. At this moment TX arbitrator proceeds with frame validation.
6		TX arbitrator FSM loads TX frame metadata from TXT buffer 1 RAM (Frame format word) to double-buffer registers. These are not visible to CAN Core, they hold metadata internally.
7		TX arbitrator FSM loads TX frame identifier from TXT buffer 1 RAM (Identifier word) to Identifier capture register. At the same clock cycle, TX arbitrator FSM loads metadata from double-buffer registers to capture registers on output of TX Arbitrator. Reffer to 3.18.9 for explanation.
		TXT buffer 1 becomes "validated" and TXT arbitrator signals that there is a valid TX frame for transmission to CAN core.
8		When Protocol control FSM is in sample point of third bit of intermission or bus idle, it issues Lock command to TXT buffer 1 (TXT buffer 1 becomes Used, TXT buffer FSM moves to TX in progress state) and TX arbitrator becomes Locked.
9		TX arbitrator is Locked and it is waiting for Unlock command. No TXT buffer validation is in progress. If another higher priority TXT buffer became Available this has no effect as frame transmission is already in progress.
10		Protocol control transmitts frame from TXT buffer 1, and issues Unlock - done command to TXT buffer 1 (TXT buffer 1 becomes Unavailable and TXT buffer FSM moves to TX OK). Since TXT buffer 1 was only TXT buffer which was Available before the transmission, now there is no TXT buffer which is Available. Therefore no TXT buffer is Selected, and no TXT buffer validation is in progress. TX arbitrator signals there is no Validated TXT buffer to CAN Core.
11	SW reads state of TXT buffer 1 and finds out whether transmission was aborted or it ended succesfully.	

Table 3.51: TX arbitrator - use-case 1

Step	External action (SW or external components)	HW Action
1	SW configures priority 1 to TXT buffer 1 and priority 2 to TXT buffer 2. SW fills TXT buffer 1 and TXT buffer 2 by CAN frames. SW issues Set ready command to TXT buffer 1.	TXT buffer 1 FSM goes to Ready state and therefore TXT buffer 1 becomes Available from TX arbitrators point of view. Since this is only Available TXT buffer, it becomes Selected.
2		TX arbitrator performs validation process (loads timestamp words, executes timestamp comparison, loads metadata and identifier) and TXT buffer 1 becomes Validated. TX arbitrator signals to CAN core that there is validated TXT buffer for transmission.
3	SW Issues Set ready command to TXT buffer 2.	TXT buffer 2 FSM goes to Ready state and therefore TXT buffer 2 becomes Available from TX arbitrators point of view. As TXT buffer 2 has higher priority than TXT buffer 1, TXT buffer 2 becomes Selected by Priority decoder.
4		TXT buffer validation process restarts with TXT buffer 2. During this time TXT buffer 1 remains Validated (TXT buffer 1 is still Available). If during validation process of TXT buffer 2, Protocol control issued HW Lock command, transmission would still be started from TXT buffer 1.
5		TX arbitrator finishes validation process (loads timestamp words, executes timestamp comparison, loads metadata) of TXT buffer 2. At the end, TXT buffer 2 becomes Validated and TXT buffer 1 (which was Validated till now) becomes Available.
6		Protocol control issues Lock command and since now TXT bufer 2 is Validated, transmission starts from TXT buffer 2. TX arbitrator becomes Locked.

Note: This allows performing validation of another TXT buffer while previous TXT buffer is still Validated. Only when validation process is finished, index of Validated TXT buffer will be changed to new TXT buffer. The reason behind this is following: If TXT buffer is validated and SW decides to issue Set ready to another TXT buffer which is higher priority, Lock command might arrive just slightly after this moment (SW and Protocol control have no synchronisation). If first TXT buffer did not remain validated during validation process of new TXT buffer, ***tran_frame_valid*** would need to drop low before the validation process of second TXT buffer is finished. This would cause that for some short time, Protocol control would not have any TXT buffer available for transmission, while actually two TXT buffers are in Ready state. This effect is undesirable.

Note: Due to meta-data double buffering, validated TXT buffer is swapped atomically (TXT buffer index, identifer and loaded metadata) from Protocol control point of view. It can never occur that e.g. data will be transmitted from TXT buffer 1 with incorrect metadata or identifier, this would be a bug.

Note: This behaviour is necessary, since TXT buffer which is Validated suddenly becomes Unavailable due to Set Abort command. If ***tran_frame_valid*** did not drop low immediately, it could happend that Protocol control would issue Lock command on a TXT buffer which was Unavailable (in Aborted state).

Table 3.52: TX arbitrator - use-case 2

Step	External action (SW or external components)	HW Action
1	SW configures TXT buffer 1 priority to 1 and TXT buffer 2 priority to 2. SW fills TXT buffer 1 and TXT buffer 2 RAMs by CAN frames. SW Issues Set ready command to TXT buffer 1 and TXT buffer 2.	TXT buffers 1 and 2 become Available and TXT buffer 2 becomes Selected because it has higher priority than TXT buffer 1.
2		TX arbitrator performs TXT buffer 2 validation process (loads timestamp words, executes timestamp comparison, loads metadata and identifier) and TXT buffer 2 becomes Validated. TX arbitrator signals to CAN core that there is Validated TXT buffer for transmission.
3	SW Issues Set abort command to TXT buffer 2.	TXT buffer 2 which is now Validated becomes Unavailable. TX arbitrator immediately (in the same clock cycle) signals to CAN core that no TXT buffer is available for transmission (***tran_frame_valid*** drops low).
4		As TXT buffer 1 is now only Available TXT buffer and thus it becomes Selected. TX arbitrator proceeds with validation of TXT buffer 1 and upon its end when TXT buffer 1 becomes Validated, it signals that there is available frame for transmission.

Table 3.53: TX arbitrator - use-case 3

Step	External action (SW or external components)	HW Action
1	SW stores a frame to a TXT Buffer 1 and issues Set ready command.	TXT Buffer 1 becomes available from TX Arbitrator point of view.
2		TX Arbitrator starts validating TXT Buffer 1. It reads out Metadata, Identifier, Timestamp Low/High words. During each of these words, it checks that parity of word being read is correct. If not, it stops validation of this TXT Buffer, and it signals this to TXT Buffer 1.
3		TXT Buffer 1 moves to Parity Error state.

3.18.2 Priority decoder

File: priority_decoder.vhd

Priority decoder selects highest priority TXT buffer from all Available TXT buffers combinatorially. Such TXT buffer becomes Selected. Priority of TXT buffers is given by SW (TX_PRIORITY register). If no TXT buffer is Available, Priority decoder signals it on its output and no TXT buffer is Selected (and TXT buffer validation will not be started). If two Available TXT buffers have equal priority, TXT buffer with lower index is selected. Priority decoder provides index of Selected TXT buffer on its output.

Priority decoder is implemented as comparator tree with 3 levels (see Figure 3.33). Each level contains so called "decoder cells". Decoder cell selects higher priority TXT buffer from two TXT buffers. Each decoder cell behaves like so:

- When only one of the two TXT buffers is Available it is automatically selected, its index is propagated as winner of comparison and "Available" output of this decoder cell is high.
- When no TXT buffer input is Available, ***output_valid*** is low.
- When both TXT buffer inputs are Available, ***output_valid*** is high and index TXT Buffer with higher priority is propagated as winner.

Priority decoder supports up to 8 input TXT buffers. If less than 8 TXT buffers are configured (see ***txt_buffer_count***), unused inputs are driven to zeroes.

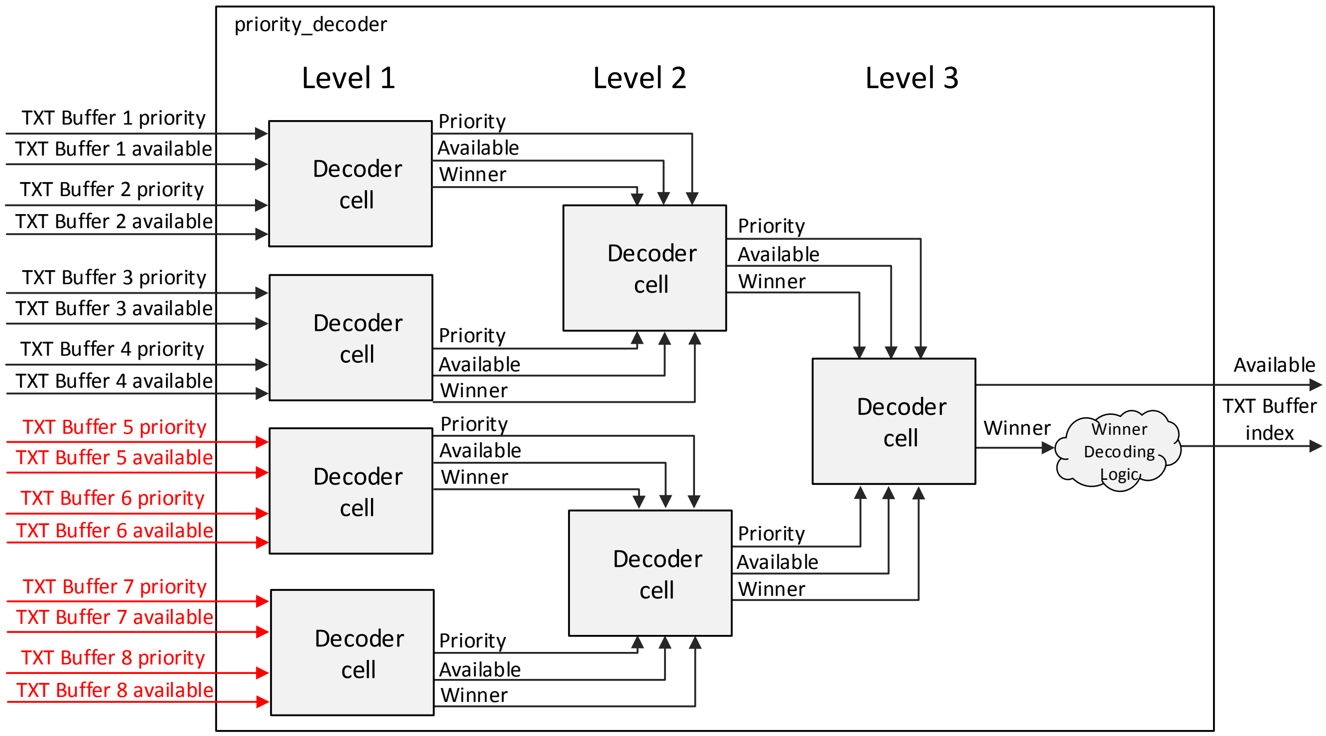

Figure 3.33: Priority decoder block diagram

3.18.3 TXT buffer change between transmissions

Table 3.54: Selected TXT buffer changed between transmissions

Step	SW action	HW action / State
1	SW fills TXT buffer 1 RAM. SW enables retransmitt limitation and configures Retransmitt limit to 5.	TXT buffer 1 FSM is in Empty State.
2	SW issues Set ready command to TXT buffer 1.	TXT buffer 1 FSM moves to Ready state. TXT buffer 1 becomes Available from TX arbitrators point of view.
3		TX arbitrator performs validatation and TXT buffer 1 becomes Validated, TX arbitrator signals this to CAN core.
		CAN core issues Lock command and starts transmitting from TXT buffer 1. TXT buffer 1 becomes Used and TXT buffer 1 FSM goes to TX in progress state
4		An error frame occurs or arbitration is lost. Protocol control signals Unlock - arbitration lost or Unlock - error frame" commands. TXT buffer 1 becomes Unavailable , TXT buffer 1 FSM moves to Ready and Retransmitt counter is incremented to 1.
5	SW fills TXT buffer RAM 2. SW Issues Set ready command to TXT Buffer 2.	TXT buffer 2 moves to Ready state. Lets assume TXT buffer 2 has higher priority than TXT buffer 1.
6		Now there are two Available TXT buffers (1 and 2). TXT buffer 2 becomes Selected by Priority decoder because it has higher priority.
7		TX arbitrator performs validation and TXT buffer 2 becomes Validated, TX arbitrator signals this to CAN core.
8		CAN core issues Lock command, TXT buffer 2 becomes Used (transmission starts by CAN core). At this moment Retransmitt counter is cleared because TXT buffer used for current transmission (TXT buffer 2) is different from the one for previous transmission (TXT buffer 1). (Logically, counting retransmissions on TXT buffer 2 shall not include one previous failed transmission from TXT buffer 1, because it is different CAN frame which is being transmitted).

3.18.4 TX Arbitrator corner-cases

TX arbitrator must react on following events which are all not synchronized:

- Change of TXT buffer priorities by SW -> possibly change of selected TXT buffer.
- Change of TXT buffer state (due to SW commands) -> possibly change of selected TXT buffer.
- Lock command from Protocol control.

Handling of these events is resolved like so:

- Lock command shall never occur when TX Arbitrator FSM is Idle.
- Unlock command shall never occur when TX Arbitrator FSM is not Locked.

- Lock command shall only occur when there is TXT buffer available for transmission, or when it was available for transmission in previous clock cycle. It might happend, that Lock command and Set abort command are active simultaneously. Due to Set abort command, it might be that only Available TXT buffer becomes immediately unavailable, therefore Lock command is active when no Available TXT buffer is signalled. This is OK since TXT buffer FSM resolves simultaneous Set abort and Lock command.
- Lock command occurs at the same time as Selected TXT Buffer is changed. Lock command shall have priority and TX Arbitrator FSM shall become Locked.
- TXT Buffer validation process is about to be finished, but Lock command occurs. Lock command shall have priority, TX Arbitrator FSM shall become Locked and Metadata, Identifier capture registers shall not be preloaded!

3.18.5 TXT buffer addressing

During TXT buffer validation process, TX arbitrator is accessing TXT buffer memories and loads Frame format, Identifier, Timestamp low and Timestamp High words, therefore TXT buffer RAM address on port B is given by TX arbitrator FSM.

During transmission when TX arbitrator is Locked, TX arbitrator holds index of Used TXT buffer. During this time, Protocol control FSM provides address of memory word from which it reads relevant data word for transmission. TX arbitrator uses this address to drive TXT buffer address and index of Used TXT buffer to multiplex read data. Data memory words (see 3.10) are addressed during transmission of data field and Protocol control transmitts value of data field from these memory words. Each next 4 bytes of data field correspond to one memory word in TXT buffer RAM. From output of TXT buffer RAM, this memory word is loaded to TX shift register and transmitted from there (see 3.14.1). Therefore Protocol control provides address of data word with sufficient reserve to cover latency of TXT buffer RAM as is shown in Table 3.55. Metadata and Identifier for transmission are available from capture registers in TX arbitrator which were loaded during TXT buffer validation process.

Table 3.55: TXT buffer RAM adressing during transmission

CAN frame field	Memory word in TXT buffer addressed by Protocol control	Meaning of data loaded to TX shift register
DLC	Data word 0	data field bytes 0 .. 3
data field byte N * 4 - 1	Data word N + 1	data field bytes (N * 4) to (N + 1) * 4

3.18.6 TXT buffer RAM access

TXT Buffer RAM has clock gating implemented if ***target_technology*** = 0 (ASIC). In such case, clocks are enabled only when there are write (by user) or read accesses (by TX Arbitrator or Protocol control FSM) to RAM. If TX Arbitrator is performing TXT buffer validation process, the clocks are ungated during this process since TX Arbitrator is reading metadata words from TXT buffer RAM. If Protocol control FSM is reading data words (during transmission of data field), TXT buffer RAM clocks are ungated when new word shall be read (when read pointer is updated by Protocol control FSM).

3.18.7 TX frame timestamp comparison

Part of TXT buffer validation process is comparison of ***timestamp*** input with timestamp of CAN frame in TXT buffer which is currently being validated. If ***timestamp*** input is lower than timestamp of CAN frame in currently validated

TXT buffer, validation process is paused. When ***timestamp*** input is equal to or higher than timestamp of CAN frame in currently validated TXT buffer, TXT buffer validation proceeds. If during this time index of Selected TXT buffer changes, validation process is restarted.

Comparison of timestamps realizes Time triggered transmission functionality as is described in 9.2 of [1]. Only when ***timestamp*** input passes (desired moment of transmission passes), TXT buffer is admited for transmission to CAN core. This does not mean that CAN core will transmit the frame immediately! CAN core will transmitt such frame in nearest bus idle or when it samples dominant bit during third bit of intermission. Since TXT buffer validation process takes 6 clock cycles, ***timestamp*** input must reach TX frame timestamp at latest 6 clock cycles of System clock before sample point of a bit to be considered for transmission from following bit. Mismatch between the time when frame validation finishes due to transmitted frame timestamp passing and sample point of SOF bit can be up to two bit times as is demonstrated in Figure 3.34. This situation can be avoided if change of time is synchronized in a system with sufficiently large period of counting on ***timestamp*** input (close to Bit time period).

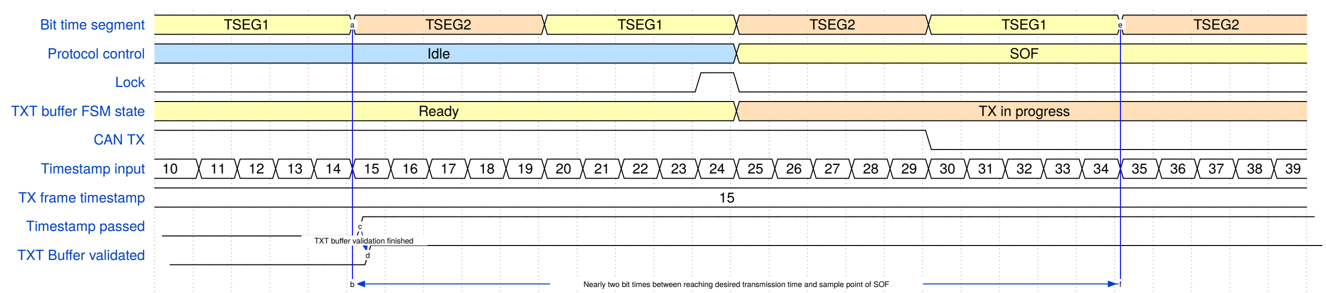

Figure 3.34: Time triggered transmission

Consider having two TX frames with timestamps 10 (in TXT buffer 1) and 50 (in TXT buffer 2). Lets assume that TXT buffer 2 has higher priority and it is therefore Selected and validation process is in progress. It finishes its validation wehn ***timestamp*** input reaches 50. Although CAN frame in TXT buffer 1 has lower timestamp, it is transmitted after frame from TXT buffer 2 because TXT buffer 2 has higher priority! Therefore TXT buffer priority is at any moment considered first during TXT buffer selection and CAN frame timestamp is considered only from Selected TXT buffer.

3.18.8 Lock and Unlock commands

Protocol control FSM issues Lock command in third bit of intermission (when it samples dominant bit) or during bus idle when there is Validated TXT buffer available. In such case CTU CAN FD becomes transmitter of following CAN frame. After Lock command, TX arbitrator becomes Locked and signalling of Validated TXT buffer remains high during whole frame. If there is no TXT buffer Validated so far and TXT buffer becomes Validated just slightly after Protocol control samples dominant bit during third bit of intermission or bus idle, unit becomes receiver and frame from Validated TXT buffer is not transmitted. If suspend transmission field is transmitted and Protocol control samples dominant bit, it does not issue Lock command and becomes receiver of following frame.

3.18.9 Metadata double-buffering

During TXT buffer validation process, TX arbitrator first reads Frame format word from TXT buffer RAM and stores it in internal registers which are invisible to CAN core. In the next step TX arbitrator reads Identifier word from TXT buffer RAM and stores it to capture register which is available to CAN core. At the same time internal registers with metadata are moved to capture registers for metadata. Therefore, reading of metadata from TXT buffer RAM is double-buffered. Both identifier and metadata available for CAN core are changed at once (atomically), therefore it will never happend that

Identifier in capture registers corresponds to different CAN frame than metadata in capture registers. This is necessary as when there is Validated TXT buffer, another TXT buffer validation process can be in progress. In change was not atomic, CAN core could issue Lock command and transmitt e.g. ID from TXT buffer 1 and metadata from TXT buffer 2.

3.18.10 TX datapath hazard protection

TX frame datapath (TX arbitrator + TXT buffers) are both manipulated by SW and HW commands simultaneously. This fact opens question of hazards susceptibility. Such a hazard would occur, when e.g. TXT buffer FSM moved to Aborted state after Set abort command, but Protocol control FSM still managed to issue Lock command and start transmission from this TXT buffer. In such case, Protocol control FSM would transmitt from TXT buffer which is Aborted (and therefore content of its RAM can be modified by SW). Due to combinatorial path between Set abort and indication of Validated TXT buffer, it never happends that when Set abort command is issued to a TXT buffer, Protocol control FSM would issue Lock command, therefore this situation will never occur. The relevant combinatorial path is shown in Figure 3.35.

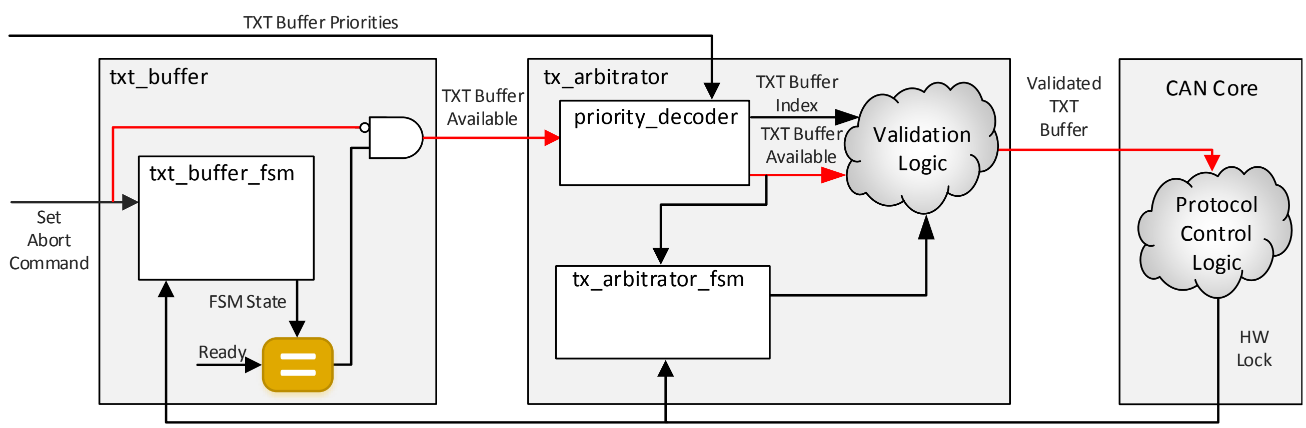

Figure 3.35: TX datapath hazard protection

3.18.11 TX Abort + Retransmitt clear

TODO: This feature is not yet designed! If TXT Buffer which is currently Validated or Used becomes "Aborted", then retransmitt counter should be also cleared. It can happen that user will abort buffer, replace CAN frame within this buffer and put ready again. In such a case, retransmitt counter should count only retransmissions of new frame! This would become epecially important if we went for generic amount of TXT buffers! If config with only 1 TXT buffer was used, then any abort in actual implementation leaves retransmitt counter untouched and any new frame would start with this value of retransmitt counter... This could be implemented like so: If TXT Buffer FSM moves to Aborted, it gives a signal. If last TXT Buffer that was used for transmission (not Selected one because when abort is applied on TXT buffer, it will not be Selected!), is equal to index of TXT Buffer that just moved to Abort, then retransmitt counter will be cleared. This still needs to be evaluated.

3.19 Interrupt Manager

File: int_manager.vhd

Interrupt manager implements following functionality:

- Capture occurence of events/conditions within CTU CAN FD to Interrupt status register.
- Interrupt masking and enabling.
- Generation of level-based Interrupt output.

Occurence of events within CTU CAN FD is captured to Interrupt status register (INT_STAT) register when corresponding interrupt is unmasked. When Interrupt is masked, correponding event is ignored. Interrupt mask is set by writing logic 1 to corresponding bit of INT_MASK_SET register. Interrupt mask is cleared by writing logic 1 to corresponding bit of INT_MASK_CLR register. When a bit in Interrupt status register is set, it causes ***int*** output of CTU CAN FD to go high when this interrupt is enabled. A bit in Interrupt status register is cleared by writing logic 1 to corresponding bit in INT_STAT register. Value of ***int*** output is given by logical OR of all enabled interrupts which have Interrupt status equal to logic 1. Interrupt output is registered to be glitch free. Interrupt is enabled by writing logic 1 to corresponding bit of INT_ENA_SET register. Interrupt is disabled by writing logic 1 to corresponding bit of INT_ENA_CLR register. When Interrupt status shall be set at the same clock cycle by an internal event of CTU CAN FD and cleared by write to INT_STAT register, Interrupt will be set (set has priority over clear). Block diagram of single interrupt datapath is shown in Figure 3.36. Available types of Interrupts are described in [2].

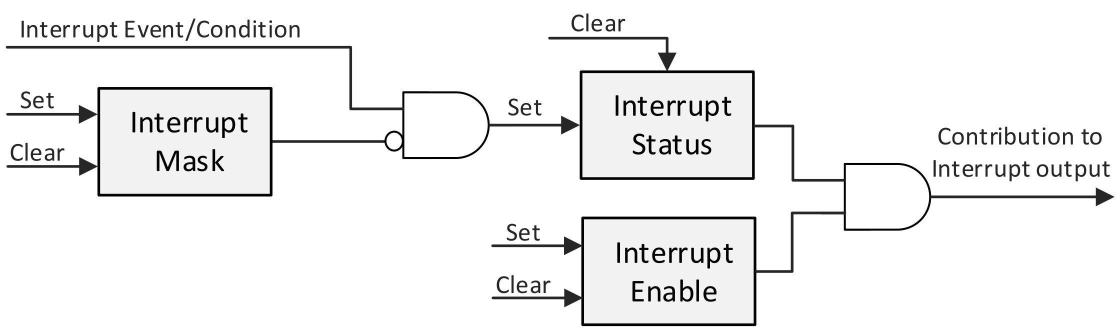

Figure 3.36: Single interrupt datapath

3.20 Prescaler

File: prescaler.vhd

Prescaler implements following functionality:

- Time quanta measurement (for both nominal and data bit rates).
- Bit segments measurement (Sync_Seg, Prop_Seg, Phase_Seg1 and Phase_Seg2).
- Hard synchronisation and resynchronisation as defined in [1].
- Check if edge is valid for synchronisation (only one edge between two sample points).
- Generate TX trigger and RX triggers for each stage of pipeline.
- Switch between nominal and data bit rates.

Prescaler block diagram is shown in Figure 3.37.

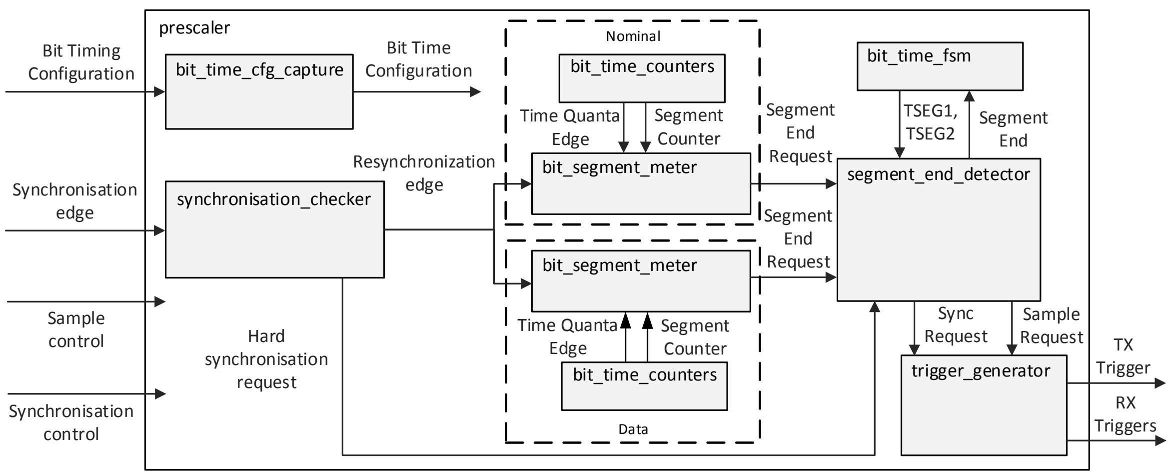

Figure 3.37: Prescaler block diagram

CAN FD standard ([1]) distuiguishes two bit rates: nominal and data. CTU CAN FD implementation distuighushes 3 bit rate types as shown in Table 3.56. Protocol Control FSM configures correct bit rate in according parts of CAN frame as explained in [1].

Table 3.56: Bit-Rate types

Bit rate type	Corresponding [1] bit rate	Description
Nominal	Nominal	Nominal bit rate for both transmitter and receiver.
Data	Data	Data bit rate for receiver of CAN FD frame.
Secondary	Data	Data bit rate for transmitter of CAN FD frame. Secondary sampling point is used to detect bit error.

Prescaler contains separate logic for both bit rates (nominal and data). Logic for Secondary is the same as for Data. Logic for single bit rate consist of Bit time counters module and Bit segment meter module. Doubled logic for nominal and data bit rates is implemented to achieve better timing performance (shorter combinatorial paths) with slightly higher resource usage when compared to common logic for nominal and data bit rates. During bits where bit rate is switched, logic for both is functioning simultaneously, otherwise only logic for actual bit rate is functioning.

3.20.1 Bit rate configuration

Bit rates (nominal and data) are configured by SW when CTU CAN FD is disabled (SETTINGS[ENA] = '0') in registers BTR (nominal) and BTR_FD (data). BTR and BTR_FD registers are writable only when SETTINGS[ENA]='0', otherwise write access to these registers has no effect. Timing parameters for each bit rate are listed in Table 3.57.

Table 3.57: CTU CAN FD bit rate configuration

Parameter name	Abbreviation	Description
Bit rate prescaler	BRP	Time quanta = Bit rate prescaler * System clock period
Synchronisation segment length	SYNC	Length of Synchronisation segment is always 1 time quanta.
Propagation segment length	PROP	Configured in multiples of time quanta.
Phase 1 segment length	PH1	Configured in multiples of time quanta.
Phase 2 segment length	PH2	Configured in multiples of time quanta.
Synchronisation jump width	SJW	Configured in multiples of time quanta.

3.20.2 Bit time counters

File: bit_time_counters.vhd

Bit time counters module contains two counters: Time quanta counter and Segment counter. There are two intstances of Bit time counters module, nominal (NBTCM) and data (DBTCM).

Time quanta counter measures length of time quanta and provides information that time quanta has elapsed (***tq_edge_nbt/dbt***=1). Time quanta has elapsed when Time quanta counter is equal to Bit rate prescaler (therefore dividing the frequency of System clock by Bit rate prescaler). ***tq_edge_nbt/dbt*** is either active continously (when Bit rate prescaler is 1), or always for one clock cycle at the end of time quanta. When Bit rate prescaler is 1, time quanta is equal to System clock period and Time quanta counter is not running.

Segment counter counts number of time quanta of actual bit segment (counts only when ***tq_edge_nbt/dbt***=1). Prescaler distuiguishes two bit segments: TSEG1 (Sync_Seg + Prop_Seg + Phase_Seg1 parts of bit) and TSEG2 (Phase_Seg2 part of bit). Segment counter counts from 0 and it is restarted upon the end of previous segment or upon hard synchronisation. Segment counter for nominal(data) bit rate shall never overflow during nominal(data) bit rate. Segment counter for nominal bit rate may overflow during data bit rate and Segment counter for data bit rate may overflow during nominal bit rate. Current bit rate is determined by Protocol control FSM based on current field of CAN frame and its type (see [1]).

NBTCM is enabled always, apart from situations when CTU CAN FD is disabled. This is to make sure, that if error is detected during data bit rate (DBTCM is being used), Nominal bit time counter will be available for measuring duration of Ph2 ASAP after error was detected. DBTCM is enabled only during data bit rate. During bits of CAN frame where bit rate is switched, both NBTCM and DBTCM are running. When NBTCM or DBTCM are disabled, none of its both counters are running (to save power). Both counters are erased when bit time segment ends to force alignment of nominal and data time quanta in the moment of bit rate switch.

3.20.3 Bit segment meter

File: bit_segment_meter.vhd

Bit segment meter module measures length of bit time segments (TSEG1 and TSEG2). Bit segment meter module maintains Expected segment length register. Expected segment length register contains number of time quanta that current bit segment shall last. When current bit segment ends, Expected segment length register is loaded with length of following bit segment. Loading of Expected segment length register is shown in Figure 3.38 for TSEG1 = 10 time quanta, TSEG2 = 5 time quanta and BRP = 2. When positive resynchronisation occurs (see [1]), Expected segment length register is increased (TSEG1 segment is lengthed) as in Figure 3.39. When negative resynchronisation occurs (see [1]), Expected segment length register is decreased (TSEG2 is shortened). All rules for loading Expected segment length registre are described in 3.58.

Table 3.58: Expected segment length register

Occurs when	Loaded to value	Description
End of segment TSEG1 due to Segment counter equal to Expected segment length register - 1.	PH2	Regular end of segment, no synchronisation.
End of segment TSEG2 due to Segment counter equal to Expected segment length register - 1.	SYNC + PROP + PH1	Regular end of segment, no synchronisation.
Positive resynchronisation with phase error <= SJW.	SYNC + PROP + PH1 + Segment counter	Segment counter = phase error in this case, therefore overall efect is as if TSEG1 was re-started with SYNC completed as in [1].
Positive resynchronisation with phase error > SJW.	SYNC + PROP + PH1 + SJW	Lengthening of TSEG1 by SJW.
Negative resynchronisation with phase error <= SJW.	SYNC + PROP + PH1 - 1	Immediate end of segment. TSEG2 ends, therefore Expected segment length register is preloaded with length of TSEG1 - 1 (the same effect as hard synchronisation).
Negative resynchronisation with phase error = SJW + 1.	SYNC + PROP + PH1	Immediate end of segment. TSEG2 ends since magnitude of phase error is equal to amount of SJW. Length of enxt segment is preloaded.
Negative resynchronisation with phase error > SJW.	PH2 - SJW	Shortening TSEG2 by SJW.
Hard synchronisation	SYNC + PROP + PH1 - 1	TSEG1 length is subtracted by 1 since hard synchronisation shall restart Bit with SYNC segment completed according to 11.3.2.3 of [1].

When Segment counter is equal to or higher than Expected segment length register - 1, Bit segment meter module issues

End of segment request. Overally, End of segment request from Bit segment meter can be caused by following means:

- Segment counter equals Expected segment length - 1. Such a situation is shown in Figure 3.38.
- Immediate end of segment occurs. See Figure 3.40 (SJW = 3).

Immediate end of segment is signalled when there is negative resynchronisation during TSEG2 and phase error <= SJW. Immediate resynchronisation causes Segment end request in the same clock cycle when resynchronisation edge occured. In this situation, TSEG2 segment ends immediately, not one clock cycle later when updated Expected segment length register would be equal to Segment counter + 1! This special case covers negative resynchronisation with BRP=1 and phase error <= SJW. The extra clock cycle needed to update Expected segment length register is undesirable, therefore immediate end of segment was introduced.

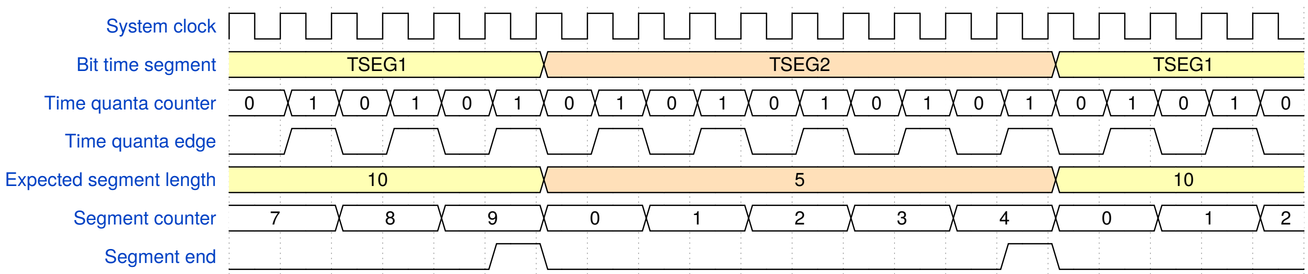

Figure 3.38: Segment end - regular

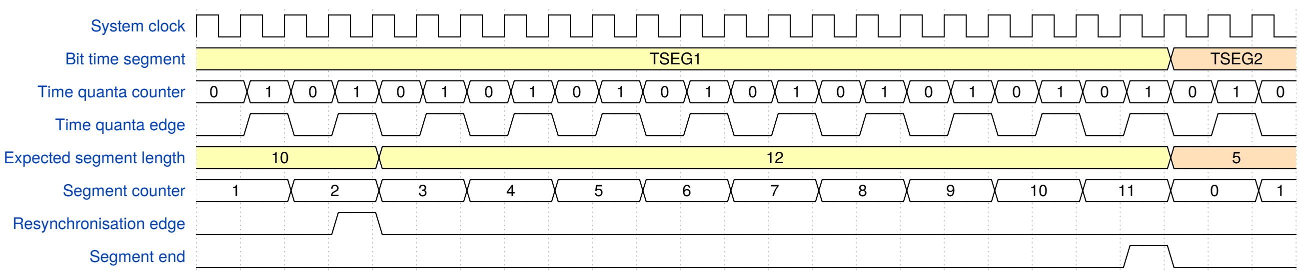

Figure 3.39: Positive resynchronisation

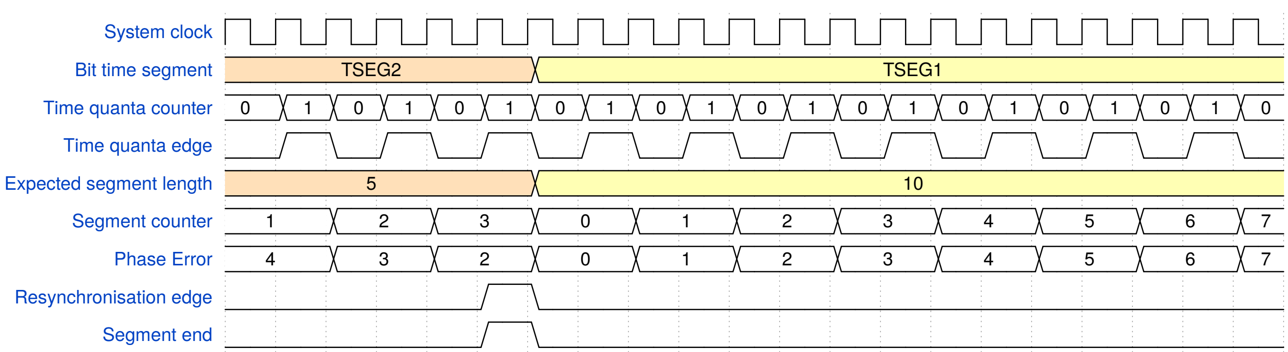

Figure 3.40: Immediate segment end

3.20.4 Segment end detector

File: segment_end_detector.vhd

Segment end detector determines when segment ends based on requests as shown in Table 3.59. Segment end detector captures these requests and processes them when time quanta has elapsed (***tq_edge_nbt***/***dbt***=1). If request arrives in the same clock cycle as time quanta has elapsed, it is processed immediately and not captured.

Table 3.59: Segment end causes

Request type	Issued by	Description
Segment end request (Nominal).	Bit segment meter (Nominal)	Considered only during nominal bit rate.
Segment end request (Data).	Bit segment meter (Data)	Considered only during data bit rate.
Hard synchronisation	Synchronisation checker.	Considered only during nominal bit rate. Shall not occur during data bit rate.

3.20.5 Bit rate switch

Since both Bit time counters (nominal and data) are running in bits where bit rate is switched (BRS and CRC Delimiter), length of TSEG2 of these bits is measured by both counters and both Bit segment meter modules can provide Segment end request. Segment end detector only considers requests from resynchronisation module of actual bit rate as given by Protocol control FSM (**sp_control** signal). Bit rate switch is shown in Figure 3.41 (BRP nominal = 2, BRP data = 1, TSEG1 nominal = 10, TSEG1 data = 7, TSEG2 data = 6). Note that in this Figure Time quanta counter, Time quanta edge, Segment counter and Expected segment length register are different signals for nominal / data bit rate but "Nominal" version are shown in nominal bit rate and "Data" versions are shown in data bit rate.

Note that in the moment of bit rate switch, Protocol control FSM provides actualized **sp_control** (bit rate) already in Process pipeline stage. Sample control is driven by DFF which is bypassed in this moment so that first time quanta of TSEG2 after bit rate switch is measured with proper bit rate selected!

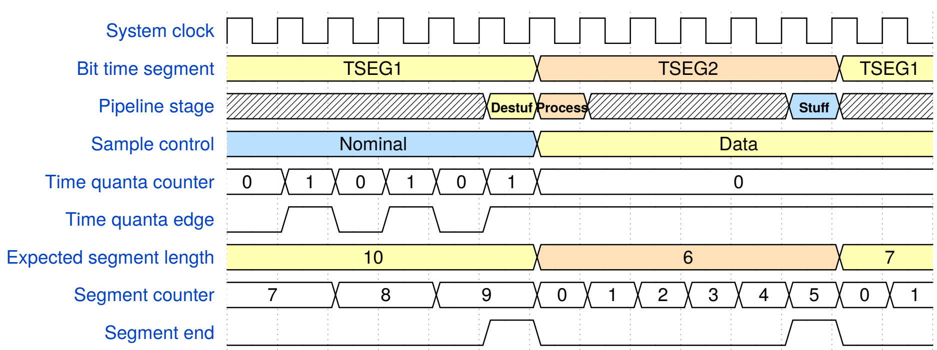

Figure 3.41: Bit rate switch

3.20.6 Prescaler FSM

File: bit_time_fsm.vhd

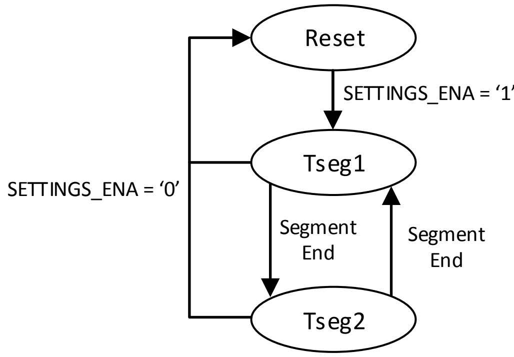

Figure 3.42: Prescaler FSM

Prescaler FSM determines actual bit time segment (TSEG1, TSEG2). Its state transition diagram is shown in Figure 3.42. Prescaler FSM issues requests to generate TX trigger and RX triggers to Trigger generator. TX trigger is requested upon the end of TSEG2 segment (start of new bit, bit value is transmitted). RX trigger is requested upon the end of TSEG1 segment (sample point, bit value is sampled).

3.20.7 Trigger generator

File: trigger_generator.vhd

Trigger generator processes requests to generate TX trigger (used to process data in Stuff pipeline stage) and RX triggers (used to process data in Destuff and Process pipeline stages). Typical scenario is shown in Figure 3.43. As there is no lower limit on length of TSEG2 from [1], resynchronisation which shortens length of TSEG2 to just one clock cycle can occur (assuming BRP=1). In such case, RX trigger for Process pipeline stage and TX trigger for Stuff pipeline stage would overlap. This is not acceptable since Stuff pipeline stage needs Process pipeline stage to be finished before it can proceed (new transmitted data must be provided by Protocol control FSM before being "stuffed"). To avoid this situation, TX trigger is shifted by one clock cycle as it is shown in Figure 3.44. Stuff pipeline stage is also shifted by one clock cycle (from last clock cycle of TSEG2 to first clock cycle of TSEG1). As value of information processing time of CTU CAN FD is 2, this situation corresponds to shortening length of TSEG2 to less than information processing time. Shifting of TX trigger corresponds to delaying calculation of following bit value after information processing time from sample point as defined in 11.3.2.4 of [1].

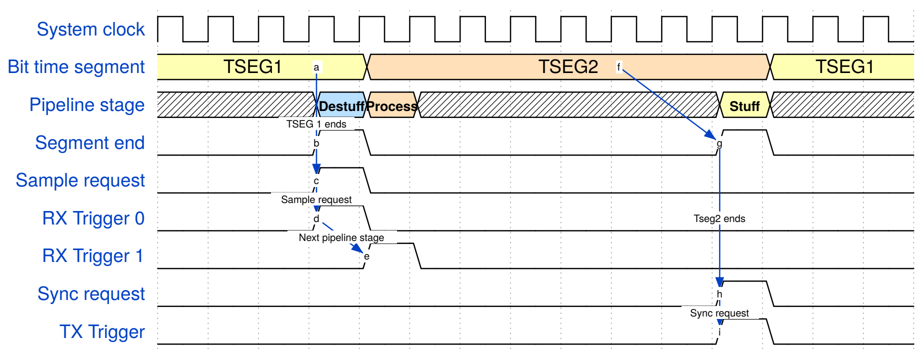

Figure 3.43: TX, RX triggers

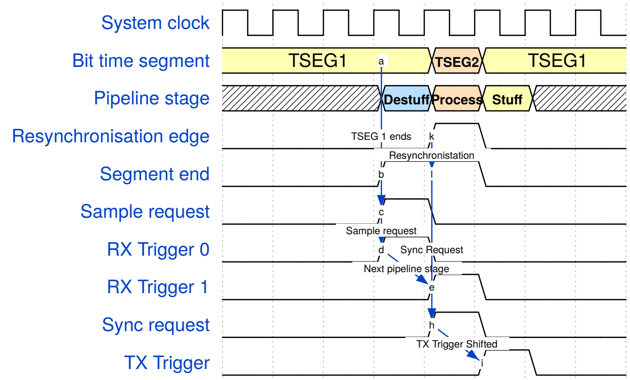

Figure 3.44: TX trigger shift

3.20.8 Synchronisation control

Type of synchronisation is controlled by Protocol control FSM based on current part of CAN frame as is shown in Table 3.60.

Table 3.60: Synchronisation control

Synchronisation type	Used during Protocol control FSM state	Description
Hard synchronisation	Suspend transmission, 2nd or 3rd bit of intermission, bus idle, integration, reintegration, FDF/res bit edge in CAN FD Frame.	TSEG1 is started with SYNC segment complement.
No synchronisation	All other parts	Transmitter operating in data bit rate does not synchronise.
No synchronisation for phase error > 0	All other parts	Node sending dominant bit does not perform resynchronisation or hard synchronisation as a result of positive phase error.
Resynchronisation	All other parts	All other recessive to dominant edges are used for resynchronisation.

3.20.9 Synchronisation checker

File: synchronisation_checker.vhd

Synchronisation checker determines if synchronisation edge (detected by Bus sampling, see 3.21) is valid for synchronisation according to 11.3.2.1 [1]. Synchronisation checker maintains Synchronisation edge flag. This flag is set when synchronisation edge occurs, and cleared when TSEG1 ends (sample point of bit). If this flag is set and another synchronisation edge occurs before the flag is cleared, such an edge is ignored and prescaler does not synchronize on this edge. Therefore, if there is more than one synchronisation edge between two consecutive sample points, only first edge is detected as valid edge and other edges are ignored. A situation where two synchronisation edges are detected (and second one is filtered out) is shown in Figure 3.45. When synchronisation edge is valid for synchronisation, it causes resynchronisation, hard synchronisation or no synchronisation according to rules in Table 3.60.

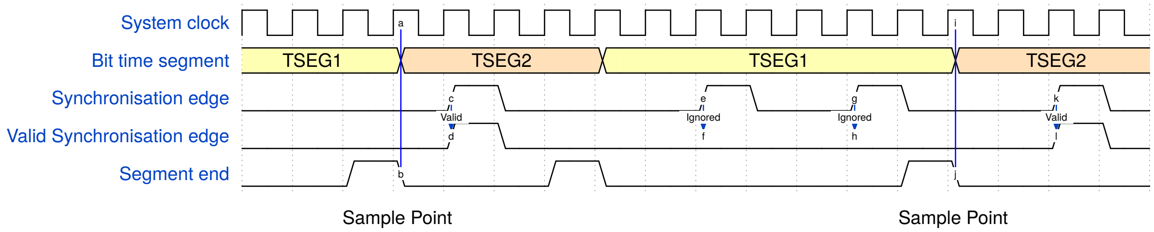

Figure 3.45: Synchronisation edge filtration

3.21 Bus sampling

File: bus_sampling.vhd

Bus sampling module implements following functionality:

- Synchronize **can_rx** input to System clock domain.
- Sample bus in sample point (Destuff pipeline stage).
- Detect edges on sampled **can_rx** and **can_tx**. Detect synchronisation edges.
- Measure transmitter delay and calculate secondary sample point offset.
- Create secondary sample point (SSP).
- Detect bit error.

Block diagram of Bus sampling is shown in Figure 3.46.

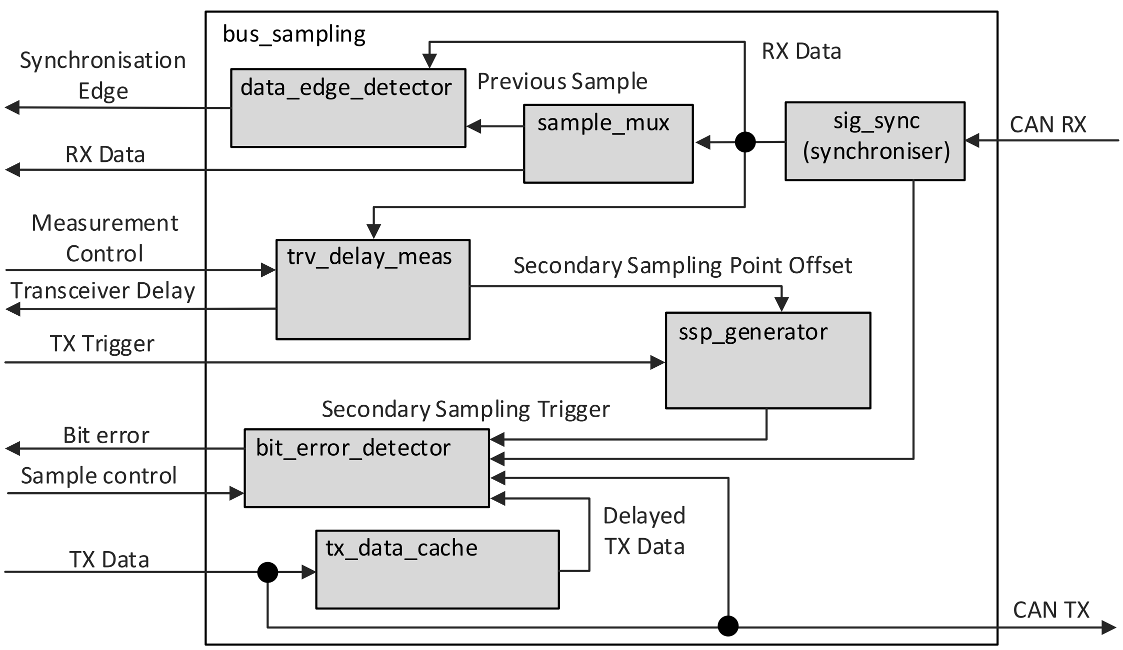

Figure 3.46: Bus sampling block diagram

Bus sampling implements 2 DFF synchronizer to synchronize asynchronous **can_rx** input. Output of this synchronizer is sampled in sample point and stored to Previous bus value register. Output of this synchronizer is also connected as data input to Bit destuffing module, therefore bus is sampled in the same moment as input serial data from CAN bus are processed by Bit destuffing. This synchronizer is clocked with System clock and it is enabled always.

Bus sampling detects edges on **can_rx** and **can_tx**. Edges on **can_tx** are detected with granularity of System clock period. Edges on **can_rx** are detected with granularity of time quanta (Edges are gated by Time quanta edge provided by Prescaler). When CTU CAN FD is running in nominal bit rate, nominal time quanta is used. When CTU CAN FD is running in data bit rate data time quanta is used. Only recessive to dominant edges are detected on **can_rx**.

Furthermore, edge on **can_rx** is detected only when bus value (synchronizer output) has opposite value than bus value sampled in previous sample point (Therefore previous sampled bus value must be recessive). Detected edge on **can_rx** is propagated as synchronization edge to Prescaler. Edge on **can_tx** is detected regardless of previous sampled bus value, but only recessive to dominant edges are detected. A typical scenario of edge detection on **can_tx/can_rx** is shown in Figure 3.47 (with BRP=2).

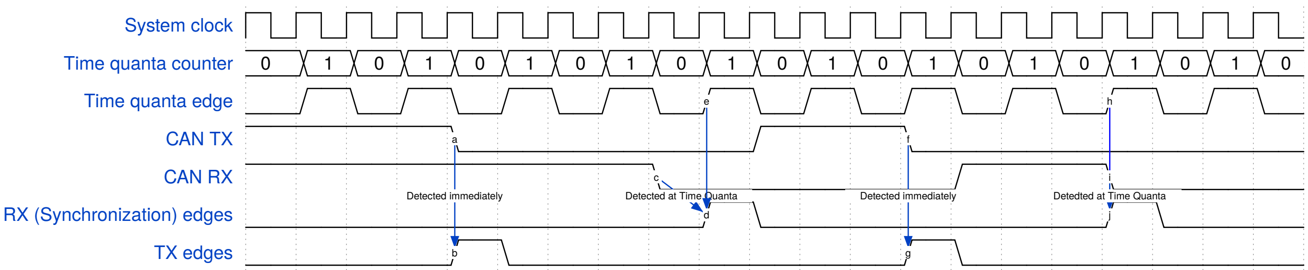

Figure 3.47: Edge detection

3.21.1 Transmitter delay measurement

File: trv_delay_meas.vhd

Transmitter delay is roundtrip delay from **can_tx** to **can_rx** upon transmission of dominant bit. This delay includes propagation of signal to physical layer transceiver, delay of transceiver and propagation of signal back. Transmitter delay is measured in CAN FD frames on falling edge between FDF (EDL) bit and following r0 bit. In CAN 2.0 frames, Transmitter delay is not measured. Transmitter delay is measured in multiples of System clock (not time quanta) and its measurement is controlled by Protocol control FSM. Measurement is described in Table 3.61 and shown in Figure 3.48.

Measured transmitter delay can be read out from TRV_DELAY register via SW. Transmitter delay readable from TRV_DELAY register is shadowed and this shadowed value is changed upon the end of transmitter delay measurement. Therefore if SW reads TRV_DELAY during measurement, it will read previous measured value. New value will be read only after the end of current measurement. To read proper value of transmitter delay from TRV_DELAY, at least one CAN FD frame must have been transmitted since previous reset, otherwise 0 will be read from TRV_DELAY register.

Table 3.61: Transmitter delay measurement

Step	Action
1	Transmitter of CAN FD frame reaches sample point of FDF (EDL) bit. It enables measurement of transmitter delay.
2	At start of next bit (Stuff pipeline stage, r0 bit), Protocol control transmits dominant bit.
3	An edge on **can_tx** is detected by Bus sampling. Transmitter delay counter is erased.
4	Transmitter delay counter is incremented by 1 each clock cycle.
5	The dominant value which was transmitted in Step 2, propagates to physical layer transceiver and back to **can_tx** input of CTU CAN FD.
6	**can_rx** input is synchronized by 2 DFF synchronizer to System clock domain. Delay of synchronizer is included in measured transmitter delay.
7	Bus sampling detects edge on **can_rx**. Measurement is finished, new value can bea read from TRV_DELAY register.

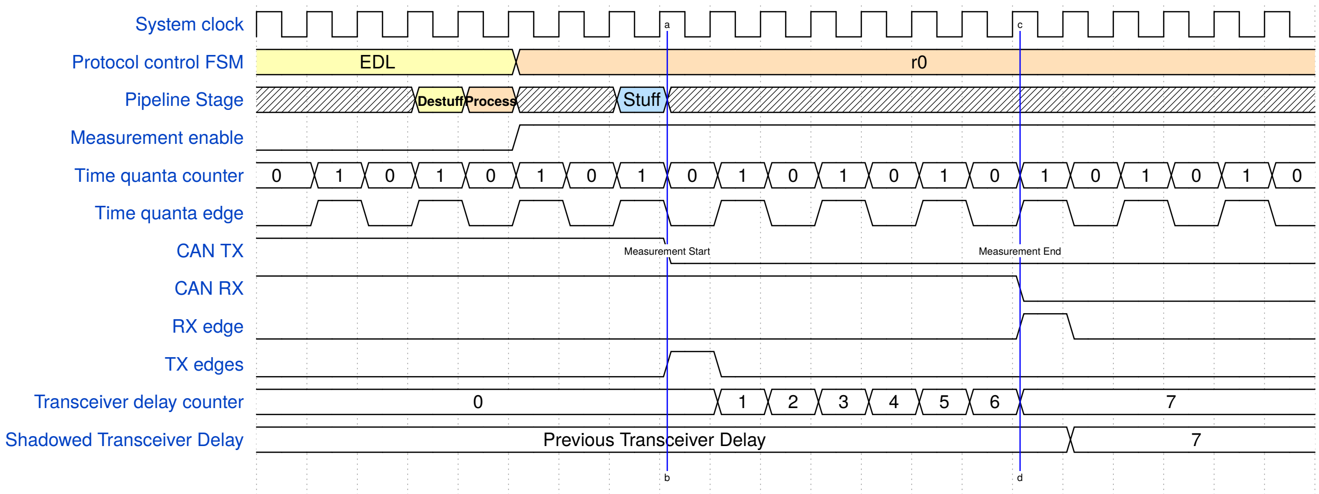

Figure 3.48: Transmitter delay measurement

3.21.2 Secondary sampling point offset

Secondary sampling point offset is calculated as offset from start of bit (SyncSeg field) in multiples of System clock. Secondary sampling point offset can be configured by SW from SSP_CFG register according to Table 3.62. Secondary sampling point Offset can have values between 0 and 127. If secondary sampling point offset is 0, secondary sampling point is active in the same clock cycle as TX trigger. If secondary sampling point offset is higher than 127 (e.g. measured transmitter delay + offset > 127), it is saturated to 127.

Table 3.62: Secondary sampling point configuration

Configuraton name	Description
Offset	Position of secondary sampling point is fixed at SSP_CFG[SSP_OFFSET]. Measured transmitter delay is not taken into account.
Offset + transmitter delay	Position of secondary sampling point is given as SSP_CFG[SSP_OFFSET] + Measured transmitter delay.
No SSP	Bit rate within Prescaler is never changed to "Secondary", it only changes to "Data" even for transmitter of CAN FD frame and bus is sampled at moment of data bit rate sample point.

3.21.3 Secondary sampling point generator

File: ssp_generator.vhd

Secondary sampling point (SSP) is created by delaying TX trigger by the amount of SSP offset as is shown in Figure 3.49. When bit rate is switched from Nominal to Data, first SSP is delayed from TX trigger by the amount of SSP offset. As SSP is used to detect bit errors by Transmitters of CAN FD frames during data bit rate, each next SSP is located whole data bit time later from previous SSP (there is no resynchronisation by Transmitters in data bit rate, so bit time is not shortened nor lengthened for them). The position of first three SSPs is shown in Figure 3.50. The relationship between first and next SSPs is used by SSP generator module which creates SSP and provides it to Bit error detector. Operation of SSP generator is described in Table 3.63.

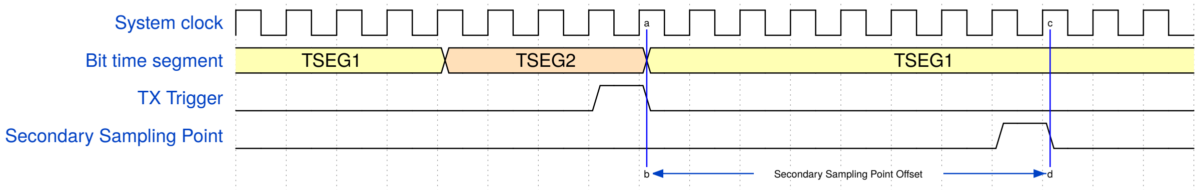

Figure 3.49: Secondary sampling point

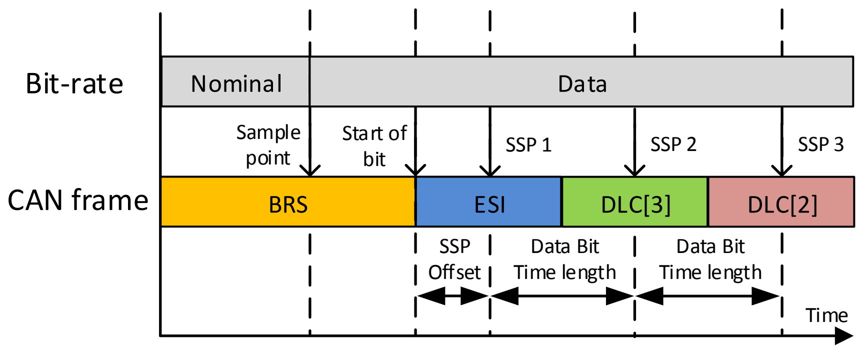

Figure 3.50: Secondary sampling point positions

Table 3.63: SSP generator operation

Step	Action
1	CTU CAN FD is transmitter of CAN FD frame where bit rate will be switched.
2	Protocol control switched bit rate in sample point of BRS bit. Protocol control configures SSP generator to measure length of data bit time and to create first SSP.
3	SSP generator waits for first TX trigger in data bit rate and starts measurement of data bit time length when TX trigger is active (by means of so called SSP counter (SSPC)). SSP generator starts measuring delay of SSP offset from TX trigger (by means of so called Bit time measurement counter (BTMC)).
4	When next TX trigger occurs (at start of next bit), SSP generator stops measurement of data bit time in SSPC. Now SSP generator knows distance between each next SSP (SSPC value).
5	When BTMC reaches value of SSP offset, SSP generator creates first SSP.
6	SSPC is restarted, and position of next sample point starts to be calculated by SSPC. Now the delay of each next SSP is given by data bit time length (value of BTMC).
7	Step 5 is repeated for each SSP until the end of data phase of CAN FD frame. Note that SSPC can reach value of SSP offset for first SSP sooner than BTMC measurement will finish (This position occurs when SSP position is located within the same bit time). This does not mind, since value of BTMC will always be higher than SSPC, therefore SSPC can count when BTMC is still running.

3.21.4 Bit error detection

File: bit_err_detector.vhd

Bit error detection differs for nominal bit rate, data bit rate and Secondary sampling as is shown in Table 3.64. Note that bit error is detected by Bus sampling always when CTU CAN FD is enabled (SETTINGS[ENA] = 1). Bit error is

Table 3.64: Bit Error detectiron

Bit-Rate	Detected when	Description
Nominal bit rate	RX trigger 1 is active	Detected when actual **can_tx** value (transmitted value in actual bit) is not equal to**can_rx** value (sampled bus value).
Data bit rate	RX trigger 1 is active	Detected when actual **can_tx** value (transmitted value in actual bit) is not equal to **can_rx** value (sampled bus value).
Secondary sample	Secondary sample point	Detected when **can_tx** value on the output of TX data cache is not equal to **can_rx** value (sampled bus value).

only ignored by Error detector module when it is irrelevant as shown in Table 3.26. Bit error detection in nominal bit rate is shown in Figure 3.51.

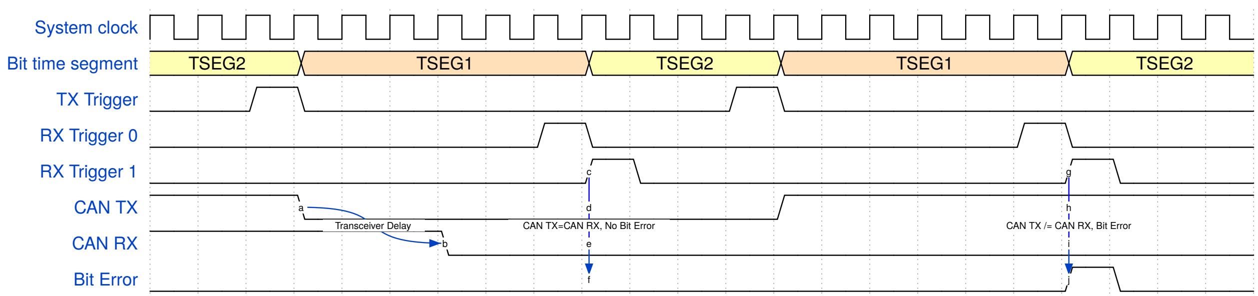

Figure 3.51: Bit error detection

3.21.5 TX data cache

File: tx_data_cache.vhd

To detect bit error in Secondary sampling, CTU CAN FD needs to remember **can_tx** values of several bits transmitted on CAN bus (secondary sample point can be so late, that it does not fit within the bit itself, and may occurs in following bits, therefore, a transmitted bit value must be rememebered until secondary sample point). This functionality is implemented by TX data cache. TX data cache is FIFO memory with each entry containing single bit. **can_tx** value is stored to TX data cache directly after a bit was transmitted to the bus (SYNC segment, One clock cycle after Stuff pipeline stage). TX data cache can store up to 8 bit values (therefore allowing 8 bits on the fly). A value is read from TX data cache when secondary sampling point is active. TX data cache operation together with bit error detection during Secondary Sampling is shown in Figure 3.52.

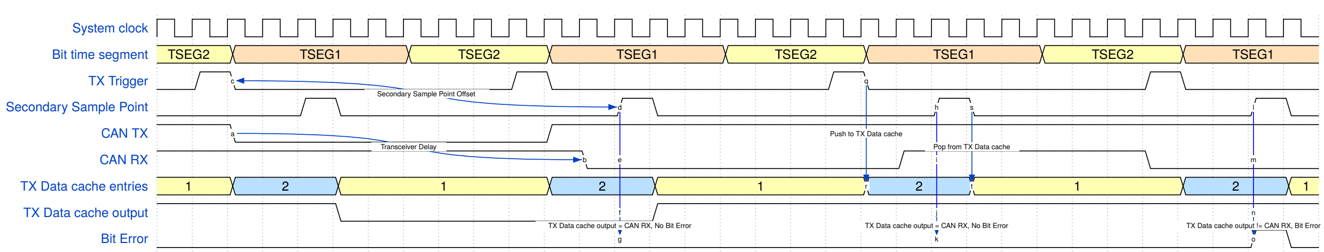

Figure 3.52: TX data cache operation

3.22 Memory registers

File: memory_registers.vhd

Memory registers implement following functionality:

- Contains configuration and status registers of CTU CAN FD (accessed by SW).
- Issue commands to CTU CAN FD by SW.
- Read received CAN frame from RX buffer RAM.
- Write CAN frame to be transmitted to TXT buffer RAMs.

Block diagram of Memory registers is shown in Figure 3.53.

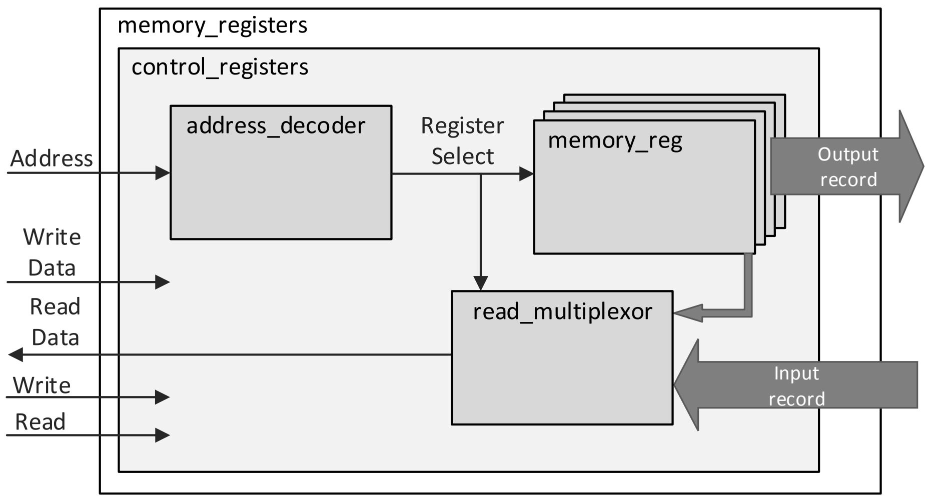

Figure 3.53: Memory registers block diagram

Memory registers contain Control registers module which is generated by [7]. Control registers module and format of CAN frame as is stored in TXT buffers and RX buffer are described in IP-XACT format with slight modifications as explained in 3.65. Memory map is edited via Kactus2 tool.

From one side, Control registers module is accessed via simple RAM-like memory interface which is described in 2.1.1. From other side, Control registers module is accessible via two records: Output record (signals going from Control registers module to rest of CTU CAN FD) and Input record (signals going from rest of CTU CAN FD to Control registers module).

Memory registers block decodes write accesses to TXT buffers (via TXT buffer 1 to TXT buffer 8 memory locations) and maps these accesses to access TXT buffer RAMs.

3.22.1 Register types

Control registers module contains following types of registers:

Read/Write register

A DFF is instantiated and connected to output record (write value). When register is read, value in this DFF is returned.

Read only register

No DFF is instantiated. When register is read, value from Input record is returned.

Write only register

A DFF is instantiated and connected to output record (write value). When register is read, all zeroes are returned.

Read/Write Once register

A DFF is instantiated and connected to output record (write value). When register is read, value from Input record is used. This type of register is used when write value has different meaning than read value.

3.22.2 Register attributes

Registers within Control registers module use additional IP-XACT attributes as is shown in Table 3.65.

Table 3.65: IP-XACT register attributes

IP XACT attribute	Attribute value	Applied on	Used on registers	Description
Modified write value	clear	Register field	COMMAND, MODE[RST], INT_STAT, INT_ENA_CLR, INT_ENA_SET, INT_MASK_CLR, INT_MASK_SET, TX_COMMAND, CTR_PRES	No DFF is instantiated in the register, but written value is only combinatorially decoded and connected to Output record.
Is present	IP_XACT parameter name	Register	FILTER_*_MASK, FILTER_*_VAL	Register is instantiated only when VHDL generic with the same name as IP-XACT parameter is set to "true". When generic is "false", register is not instantiated and its reset value is returned upon read (if it is readable). Value of this generic is added to generics of Control registers module.
Read action	modify	Register field	RX_DATA	Read signaller module is instantiated. This module combinatorially decodes when register field is being read and provides this information in Output record. Used to signal to RX buffer that there is a read from RX_DATA register.
Vendor extension - regLocks/ regLock	name= register name	Register	EWL/ ERP/ CTR_PRES	If specified, register is writable only when ***lock*** = 0. If not specified, ***lock*** input has no effect. This is used to prevent user from writing EWL/ERP/CTR_PRES unless CTU CAN FD is in test mode.

Bibliography

[1] ISO11898-1 2015 - Road vehicles, Controller area network, Part 1, Data link layer and signalling

[2] CTU CAN FD - Datasheet

[3] Avalon@ Interface specification, 2018-09-26, Intel

[4] AMBA 3 APB Protocol, v1.0, Specification, ARM

[5] AMBA 3 AHB-Lite Protocol, v1.0, Specification, ARM

[6] CAN with Flexible Data-Rate, Specification, Version 1.0, April 2012, BOSCH

[7] Register map generation tool, https://github.com/Blebowski/Reg_Map_Gen

[8] CTU CAN FD - Testbench architecture