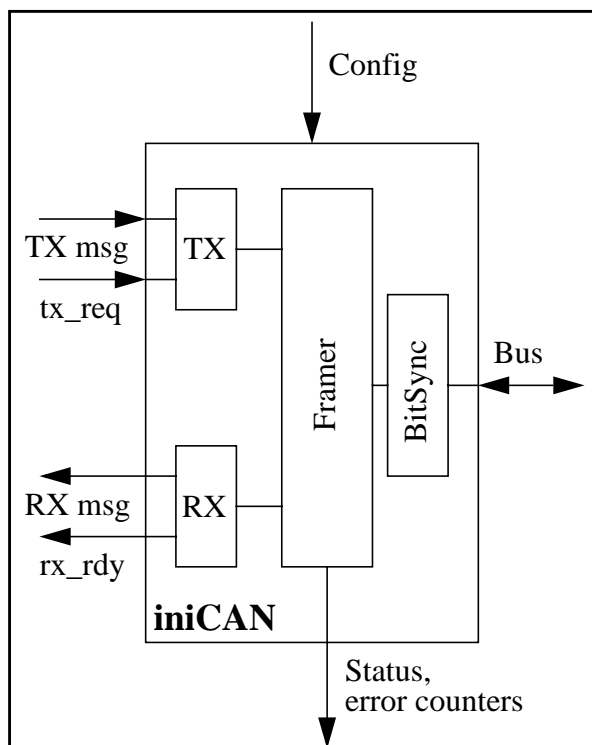


Features:

- CAN 2.0B, 1Mbit/s (and faster)
- Structured Model Description (SD)
- Technology Independent (ASIC and FPGA)
- Synthesizable VHDL Model
- Fully Synchronous Design
- Parallel Interfaces for Configuration and Message Transfer
- Access to All Internal Status
- Error Reporting
- Customizable for Special Requirements



CAN2.0B, originally developed for the European car industry, is a fast, secure, and cost-effective data bus for multi-master and real-time applications. In addition to automotive applications, it is suitable as a general data bus for industrial control functions. Example applications of the CANbus are in the service automation and textile machine industries.

INICORE created the structured VHDL CAN model for simulation and synthesis for any target technology. It can be interfaced via a message filter to various system functions such as sensor/activator control, or embedded into a system application interfacing with the microprocessor and various peripheral functions. The core contains the complete data link layer, including the framer, transmit and receive control, error handling, error reporting, and synchronization. Its structured core design and flexible interface enables access to each internal status, error counter, and frame reference.

INICORE delivered CAN cores for car manufacturers, textile machines, service automation etc. Newer applications will also use CAN as a general bus medium in smaller systems.

INICORE - the reliable Core and System Provider.
We provide high quality IP, design expertise and leading edge silicon to the industry.



US Sales Office:

INICORE INC.

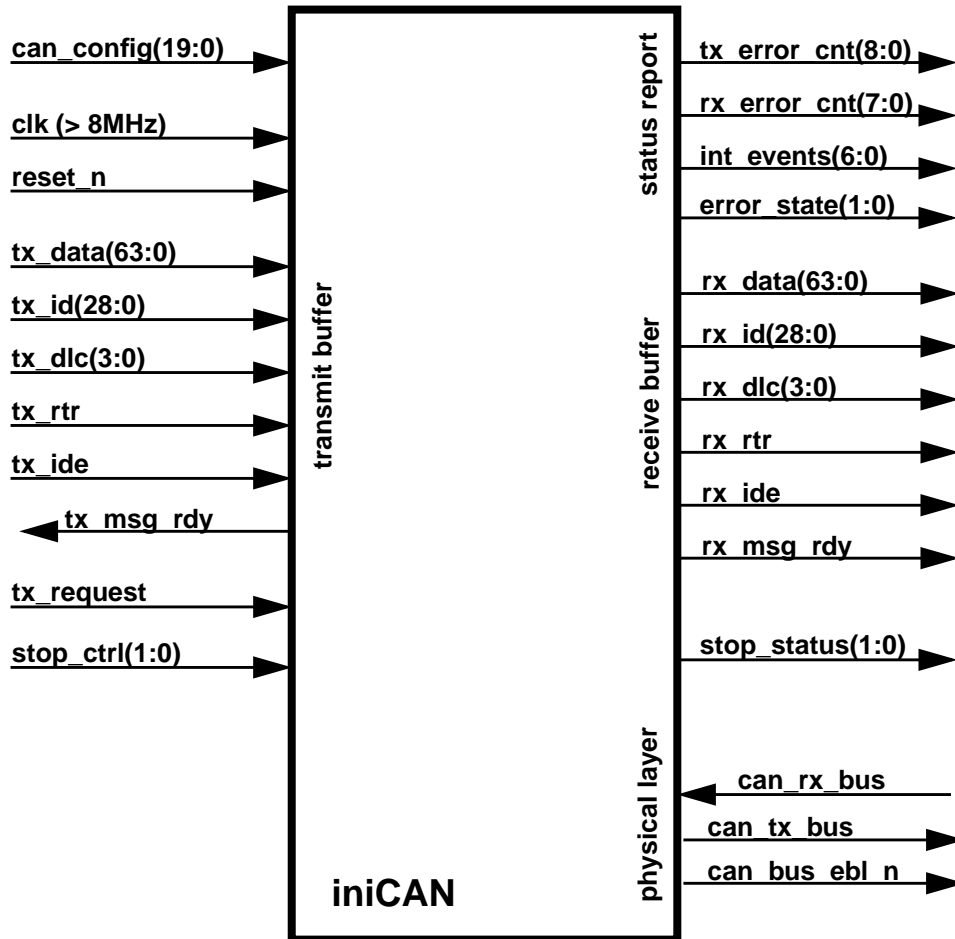
5600 Mowry School Road, Suite 180,
Newark, CA 94560
Tel: 510 445 1529 Fax: 510 656 0995
E-mail: ask_us@inicare.com
Web: www.inicare.com

INICORE AG

Mattenstrasse 6a, CH-2555 Brügg, Switzerland
Tel: ++41 32 374 32 00, Fax: ++41 32 374 32 01
E-mail: ask_us@inicare.ch
Web: www.inicare.ch

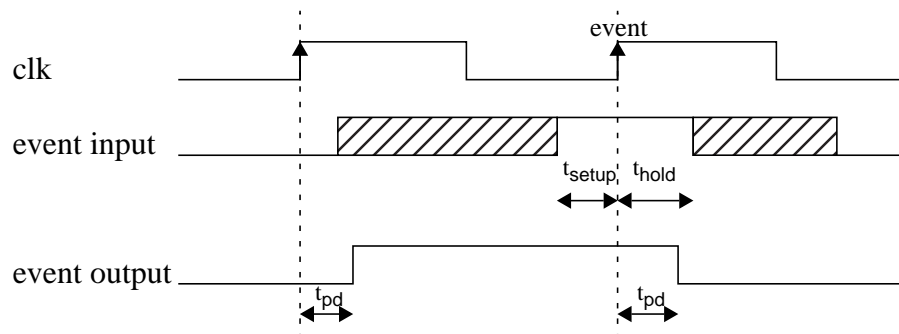
1 Overview

The iniCAN core may be used as a data link layer with parallel interfaces and event communication. Microprocessor specific interfaces must be built around the iniCAN, as well as message filters, interrupt controllers and status reporting circuits. The following picture shows all inputs and outputs:



1.1 Event communication

For communicating events, the iniCAN core uses or produces always active '1' pulses, which are activated for only one clk cycle. In the inactive state, they remain low with respect to the rising clk edge, so glitches may occur. For communicating over clock domains, these events must be synchronized first!



The parameters t_{setup} , t_{hold} and t_{pd} are technology dependent and must be determined according to the chosen technology.

2 IO description The following part lists the input and output ports of the iniCAN core and gives a short overview of their functionality.

2.1 General inputs These pins are used to clock and initialize the whole iniCAN core. There are no other clocks in this core.

| pin name | type | description |
|----------|------|---------------------------------------|
| clk | in | system clock, at least 8MHz |
| reset_n | in | asynchronous system reset, active low |

2.2 Configuration The configuration pins are used to set the bitrate, bit timing and output format. They're static inputs.

| pin name | type | description |
|--------------|------|---|
| bitrate[7:0] | in | defines the time quantum (TQ); one TQ is $(\text{bitrate} + 1)/\text{clk}$ e.g. for 1Mbit/s and 8MHz clk: bitrate = 0 |
| tseg1[3:0] | in | $(\text{tseg} + 1)$ = number of TQ in the first bit time segment: tseg1 = 0 and tseg1 = 1 are not allowed! e.g. for 1Mbit/s and 8MHz clk: tseg1 = 3 |
| tseg2[2:0] | in | $(\text{tseg} + 1)$ = number of TQ in the second bit time segment: tseg2 = 0 is not allowed, tseg2 = 1 is only allowed for direct sampling mode. e.g. for 1Mbit/s and 8MHz clk: tseg2 = 2 |
| sjw[1:0] | in | $(\text{sjw} + 1)$ = sync jump width (TQ) in case of resynchronisation |
| sampling | in | defines the sampling mode of the incoming message: '0' : direct sampling (1 point) '1' : 3 point sampling with majority decision |
| edge_mode | in | defines, which edges on the incoming messages are used for resynchronisation: '0' : use only R-D edges '1' : use R-D and D-R edges |
| auto_restart | in | defines, if the iniCAN should restart after a bus off: '0' : restart by user command (event on "stop.ctrl.clr_stop") '1' : restart automatically protocol allows it (128 x 11 R bits) |

2.3 Start - stop controlling

For controlling the iniCAN core, there are two event inputs for starting and user stop control and two outputs for start - stop status information.

| pin name | type | description |
|----------------------------|------|---|
| stop_ctrl .clr_stop | in | Event sets the iniCAN in the 'run' mode After reset, the CAN will go in 'run' mode after synchronization phase (default = 'run' mode). |
| stop_ctrl .set_stop | in | Event sets the iniCAN in the 'stop' mode, as soon the protocol allows it (bus idle). So no protocol errors are generated when the can is stopped. |
| stop_status .want_stop | out | '1' means, that the iniCAN will stop as soon as possible (when in bus idle) |
| stop_status .grant_stop | out | '1' means, that the iniCAN is in the user stop mode. |

2.4 Status and error counters

For checking the status and tracing the protocol, the following status information is available:

| pin name | type | description |
|-----------------------|------|---|
| rx_error_cnt [7:0] | out | The receive error counter represents the error value according to the CAN2.0B specification. When in bus off, the counter will count up from 1(dec) to 128(dec) for counting the 128 x 11 recessive bits, before it will be allowed again to go error active. |
| tx_error_cnt [8:0] | out | The transmit error counter represents the transmit error value according to the CAN2.0B specification. |
| rx_err_gte96 | out | When the receive error counter is greater or equal 96(dec), this signal is activated (= '1') to signal a highly disturbed bus. |
| tx_err_gte96 | out | When the transmit error counter is greater or equal 96(dec), this signal is activated (= '1') to signal a highly disturbed bus. |
| error_state[1:0] | out | Informs about the error state: "00" : error active (normal case) "01" : error passive "1x" : bus off |
| int_events[6:0] | out | Error events are generated in following situations: .crc_err : crc value doesn't match .form_err : format (delimiters etc.) is not correct .ack_err : a transmitted message was not acknowledged .stuff_err: stuff error, e.g. while receiving an active error flag .bit_err: when rx pin doesn't equal tx pin while transmitting .arb_loss: when arbitration is lost against other node .overload: when overload occurs |

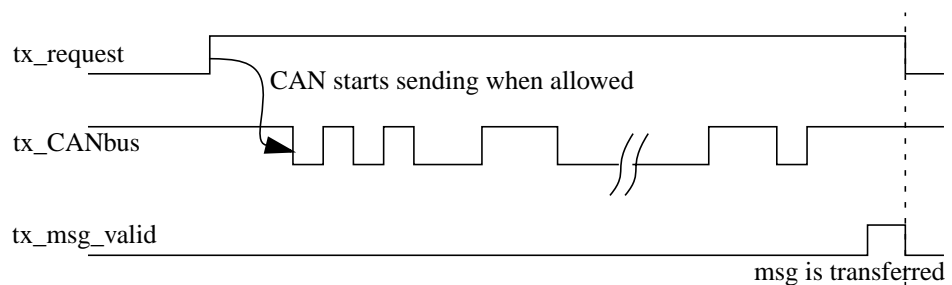
| pin name | type | description |
|-----------|------|--|
| frame_ref | out | The whole internal framer status is available on the frame reference record. It contains the following signals: .field[4:0] : actual message field (coding see below) .bit_nr[6:0] : actual bit number in the message field .rx_mode : active '1' when in receive mode .tx_mode : active '1' when in transmit mode .stuff_ind : active '1' when a stuff bit is inserted .remote_ind : the RTR bit, valid at the end of frame .extended_ind : the IDE bit, valid at the end of rame .rx_msg_valid : event for a successfully received message .tx_msg_valid : event for a successfully transmitted message |

2.5 Transmit data signals

The following list contains all needed signals for transmitting messages. For sending a message, just apply the ID, DLC, RTR, IDE and DATA. Then set the tx_request high until the tx_msg_valid event signals, that the message has been sent completely and error free. All applied data must remain stable as long as tx_request is active!

| pin name | type | description |
|--------------------|------|--|
| tx_msg_data [63:0] | in | The data to transmit. The first data byte is represented in bits [63:56], the second in [55:48] etc. Shorter messages, the unused tx_msg_data may stay undefined. Bit 63 is the first one to be transmitted. |
| tx_id[28:0] | in | The identifier to transmit. When a standard message is sent, the 11bit identifier must be placed in bits[28:18] while an extended frame will use 29 bits. Bit 28 is the first one to be transmitted. |
| tx_dlc[3:0] | in | The data length code. Values between 0 and 8 are valid and determine, how many data bytes will be transmitted. Wrong values above 8 will be transmitted as they are! |
| tx_ide | in | The extended identifier bit: '0' : send standard frame '1' : send extended frame |
| tx_rtr | in | The remote indicate bit: '0' : send data frame '1' : send remote frame |
| tx_request | in | Active high signals to the core that a message is ready to be sent. All data and tx_request must be stable until tx_msg_valid is active. Use also the tx_msg_valid signal to clear tx_request! |
| tx_msg_valid | out | Event for communicating that the message is sent successfully. |

This figure shows schematically the transmission control:

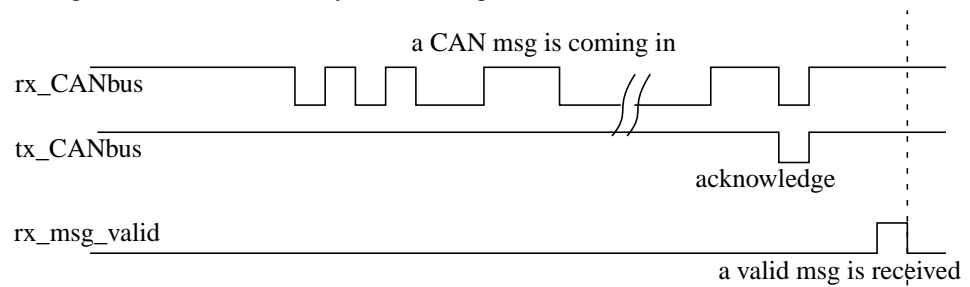


2.6 Receive data signals

The following list contains all needed signals for receiving messages.

| pin name | type | description |
|----------------------------|------|---|
| rx_msg_data [63:0] | out | The received data. The first data byte is represented in bits [63:56], the second in [55:48] etc. In shorter messages, the unused bits contain invalid data. |
| rx_id[28:0] | out | The received identifier. When a standard message is received, the 11bit identifier is placed in bits[28:18], bits[17:0] are '1' |
| rx_dlc[3:0] | out | The data length code. Values between 0 and 8 are valid and determine, how many data bytes have been received. Wrong values above 8 means that 8 data bytes are available! |
| frame_ref. extended_ind | out | The extended identifier bit is valid when rx_msg_valid = '1' '0' : received standard frame '1' : received extended frame |
| frame_ref. remote_ind | out | The remote indicate bit is valid when rx_msg_valid = '1': '0' : received data frame '1' : received remote frame (rx_msg_data not valid) |
| rx_msg_valid | out | Event for communicating, that a new message has arrived. Use it for storing the RTR, IDE, DLC, ID and DATA fields! |

This figure shows schematically how messages are received:



2.7 CANbus pins These pins represent the physical layer.

| pin name | type | description |
|---------------|------|--|
| can_tx_bus | out | Transmitter pin of CAN bus D = low ('0') R = high ('1') |
| can_rx_bus | in | Receiver pin of CAN bus D = low ('0') R = high ('1') |
| can_bus_ebl_n | out | CAN bus driver enable not '0': active '1': passive This signal is passive when the CAN controller is stopped or when reset_n = '0'. |

The following pictures show how they have to be connected:

External driver – To connect an external driver as e.g. a Phillips Chip, use the following connecting scheme:

