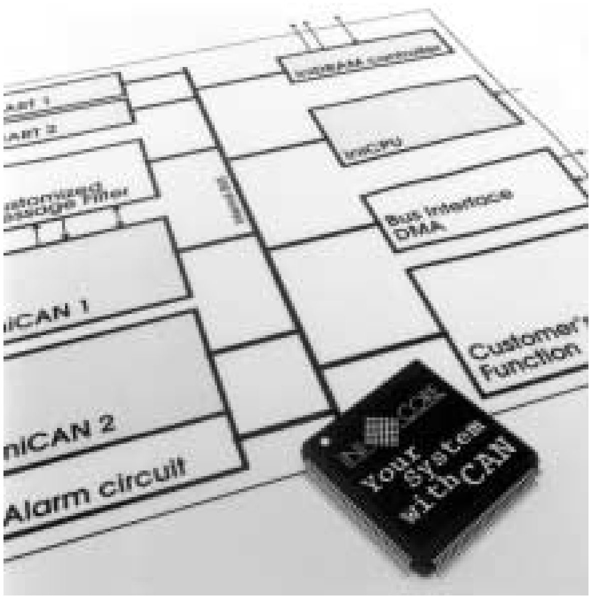
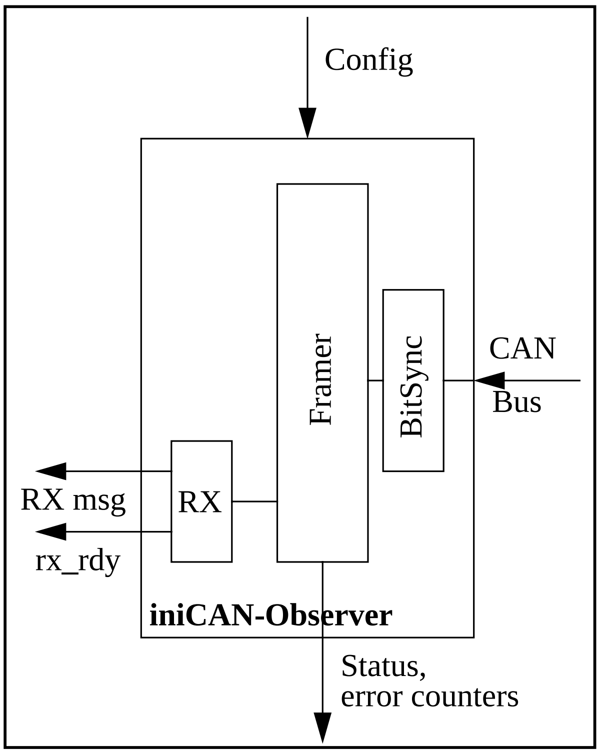
iniCAN-Observer data sheet



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



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

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-Observer model for simulation and synthesis for any target technology. It can be interfaced via any type of microprocessor interface due to the parallel easy to handle message interface. The core contains the complete data link layer, including the framer, receive control, error handling, error reporting, and synchronization. Further, special error handling features are implemented to give best possible robustness in case of local errors. 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.

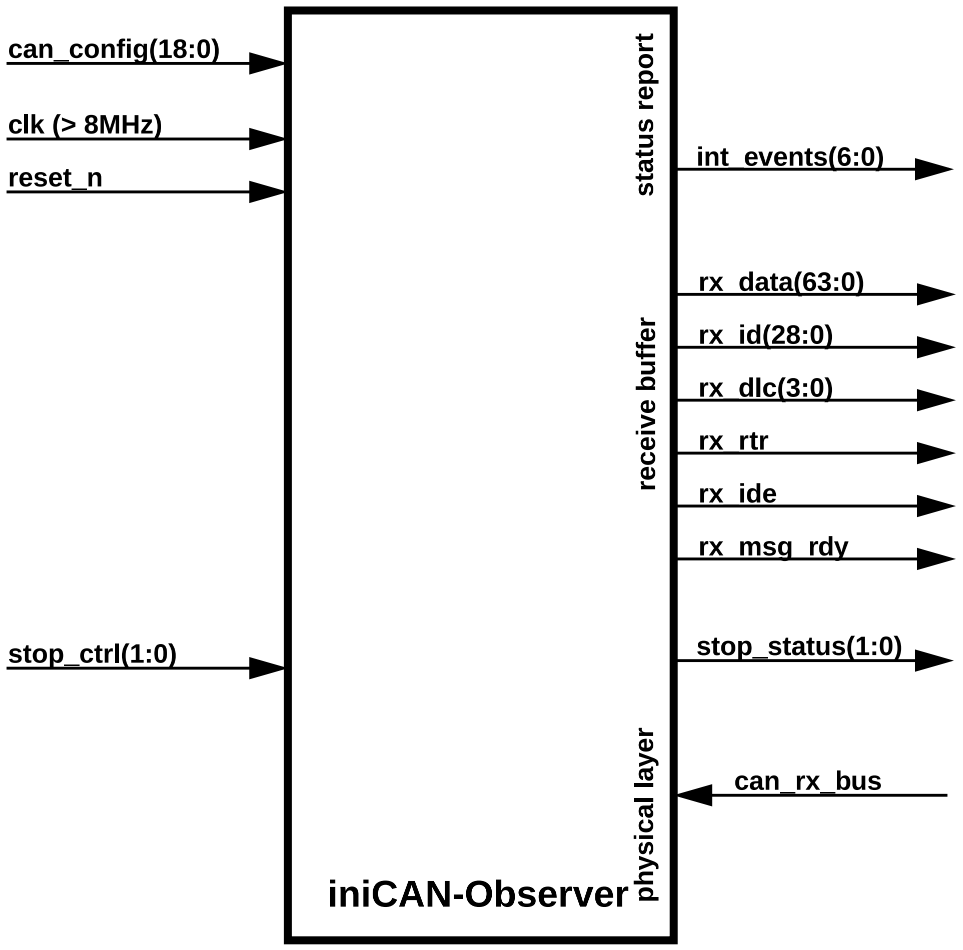
Memec Design Services

Memec Design Services
Memec Inicore AG
Mattenstrasse 6a,CH-2555 Brügg, Switzerland
Tel: ++41 32 374 32 00, Fax: ++41 32 374 32 01
E-mail: ask_us@inicore.ch
http://www.mds.ch.memec.com

Ref.Nr.: 311-DS-30 / 07/99

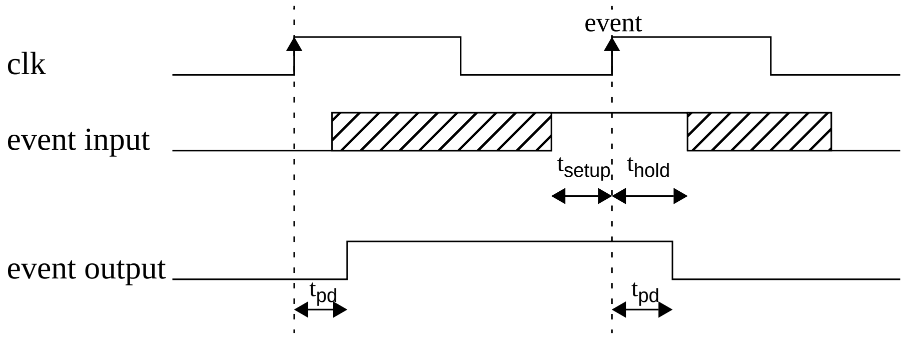
1 Overview

The iniCAN-Observer core is design for data link layer observer with parallel interfaces and event communication. Microprocessor specific interfaces must be built around the iniCAN-Observer, 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 choose technology.

2 IO description

The following part lists the input and output ports of the iniCAN-Observer core and gives a short overview of their functionality.

2.1 General inputs

These pins are used to clock and initialize the whole iniCAN-Observer core. There are no other clocks in this core.

pin name	**type**	**description**
clk	in	system clock, at least 8MHz for 1 Mbit/s transmission speed
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 (bitrate + 1)/clk e.g. for 1Mbit/s and 8MHz clk: bitrate = 0
tseg1[3:0]	in	(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	(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	(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 resyncronisation: '0' : use only R-D edges '1' : use R-D and D-R edges

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 Error Reporting

For tracing the protocol, the following events are reported:

pin name	type	description
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 received message was not acknowledged .stuff_err: stuff error, e.g. while receiving an active error flag .bit_err: when rx pin doesn’t equal (internal) tx pin .frame_err: an error frame is generated after error cond. .overload: when overload flag occurs
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 : not used for iniCAN-Observer

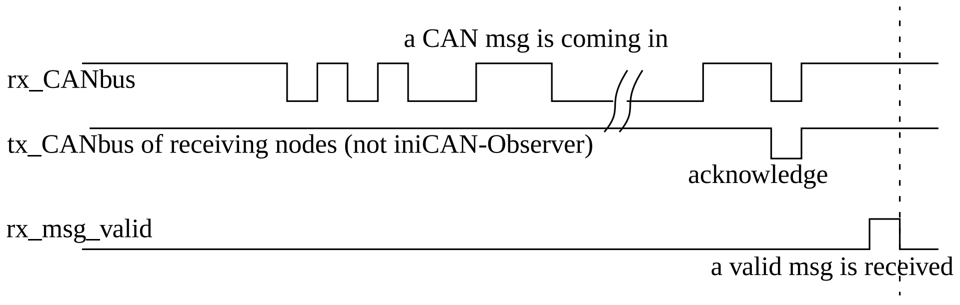
2.5 Receive data signals

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 be placed in bits[28:18], bits[17:0] are ‘1’

pin name	type	description
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:



2.6 CANbus pins

These pins represent the physical layer. Since the iniCAN-Observer is receive only, there is only one pin available:

pin name	type	description

can_rx_bus	in	Receiver pin of CAN bus
		D = low (‘0’)
		R = high (‘1’)

2.7 VHDL Entity

Below you find the VHDL Entity and type definitions of the iniCAN-Observer top module:

```
ENTITY can_obs IS
  PORT
    (
    clk              : IN     t_bit;
    reset_n          : IN     t_bit;
    can_config       : IN     t_can_obs_config;
    can_rx_bus       : IN     t_bit;
    can_stop_ctrl    : IN     t_can_obs_stop_ctrl;
    can_stop_status  : OUT    t_can_obs_stop_status;
    can_rx_msg_data  : OUT    t_can_obs_msg_data;
    can_rx_id        : OUT    t_can_obs_id;
    can_rx_dlc       : OUT    t_can_obs_dlc;
    can_int_events   : OUT    t_can_obs_int_events;
    can_bit_sync     : OUT    t_bit;
    can_frame_ref    : OUT    t_can_obs_frame_ref
    );
END can_obs;



TYPE t_can_obs_config IS RECORD
    sjw              : t_can_obs_cfg_sjw;
    bitrate          : t_can_obs_cfg_bitrate;
    tseg1            : t_can_obs_cfg_tseg1;
    tseg2            : t_can_obs_cfg_tseg2;
    sampling         : t_bit;
    edge_mode        : t_bit;
END RECORD;

SUBTYPE t_can_obs_cfg_sjw     IS t_bit_vector(1 DOWNTO 0);
SUBTYPE t_can_obs_cfg_bitrate IS t_bit_vector(7 DOWNTO 0);
SUBTYPE t_can_obs_cfg_tseg1   IS t_bit_vector(3 DOWNTO 0);
SUBTYPE t_can_obs_cfg_tseg2   IS t_bit_vector(2 DOWNTO 0);


TYPE t_can_obs_stop_ctrl IS RECORD
    set_stop         : t_bit;
    clr_stop         : t_bit;
END RECORD;



TYPE t_can_obs_stop_status IS RECORD
    want_stop        : t_bit;
    grant_stop       : t_bit;
END RECORD;
```

```
SUBTYPE t_can_obs_msg_data    IS t_bit_vector(63 DOWNTO 0);
SUBTYPE t_can_obs_id          IS t_bit_vector(28 DOWNTO 0);
SUBTYPE t_can_obs_dlc         IS t_bit_vector_4;


TYPE t_can_obs_int_events IS RECORD
    crc_err         : t_bit;
    form_err        : t_bit;
    ack_err         : t_bit;
    stuff_err       : t_bit;
    bit_err         : t_bit;
    frame_err       : t_bit;
    overload        : t_bit;
END RECORD;


TYPE t_can_obs_frame_ref IS RECORD
    rx_mode         : t_bit;
    tx_mode         : t_bit;
    field           : t_can_obs_field;
    bit_nr          : t_can_obs_bit_nr;
    stuff_ind       : t_bit;
    remote_ind      : t_bit;
    extended_ind    : t_bit;
    tx_msg_valid    : t_bit;
    rx_msg_valid    : t_bit;
END RECORD;
```