INICORE

Datasheet

iniCAN

CONTROLLER AREA NETWORK

PROTOCOL CONTROLLER CORE

Revision 2.1.0

INICORE INC.
5600 Mowry School Road
Suite 180
Newark, CA 94560
t: 510 445 1529 *f:* 510 656 0995 *e:* info@inicore.com
www.inicore.com

Table of Contents

Table of Figures

Revision History

Version	Comment
2.1.0	• Added top-level generics G_DSYNCH_EBL and G_ERROR_COUNTER_RESET
2.0.3	• Updated bitrate configuration signal description
2.0.2	• Naming inconsistencies fixed
2.0.1	• Corrected bit-mapping for cfg_testmode
2.0	• Global datasheet update • Added test mode feature

1 Overview

The Controller Area Network (CAN) bus, originally developed for the car industry, is a fast, reliable and cost-effective data bus for multi-master and real-time applications. In addition to automotive applications, it is widely used in applications such as factory automation, machine control, building automation, maritime, medical, railway and avionics. The iniCAN core first was introduced to the market in 1994 and since then is used in a lot of different applications.

The iniCAN core contains all the low-level CAN protocol handling. The core contains the complete data link layer, including the framer, transmit and receive control, error handling, error reporting and bit synchronization. Simple message level transmit and receive interfaces facilitate smooth system integration. The core provides status on error counts and events as well as a low-level frame reference pointer which identifies the current bit position within a CAN frame. This feature comes in handy when developing CAN protocol analyzers or if detailed reporting on the bit-level is required.

1.1 Applications

- Automotive
- Avionics and aerospace
- Building automation
- Entertainment
- Factory automation
- Machine control
- Science

1.2 Features

- Implementation of CAN protocol version 2.0A/B, ISO-118980-1
- Supports standard and extended identifiers
- Maximum bus speed of 1 Mbps
  - Programmable pre-scaler (1-256)
  - Programmable bit sampling settings according to CAN standard
- Access to internal frame reference pointer
  - Indicates which bit of a CAN frame is currently on the bus
- Built-in CAN error handling
  - Access to receive and transmit error counters
  - Bus state: Error active, error passive, bus-off

- Interrupts for CRC error, bit stuffing error, bit error, format error, arbitration loss, and overload frame

- Parallel message level interface
  - Simplifies system integration
- Test modes
  - Listen only mode (controller doesn't send any messages to the bus)
  - Internal loop-back (controller receives only its own messages)
  - External loop-back (controller receives a copy of sent messages)
- Register based design
  - Technology independent
  - Full synchronous design

1.3 Block Diagram

The iniCAN core contains the low-level protocol handler. Parallel receive and transmit message interfaces simplify system integration.

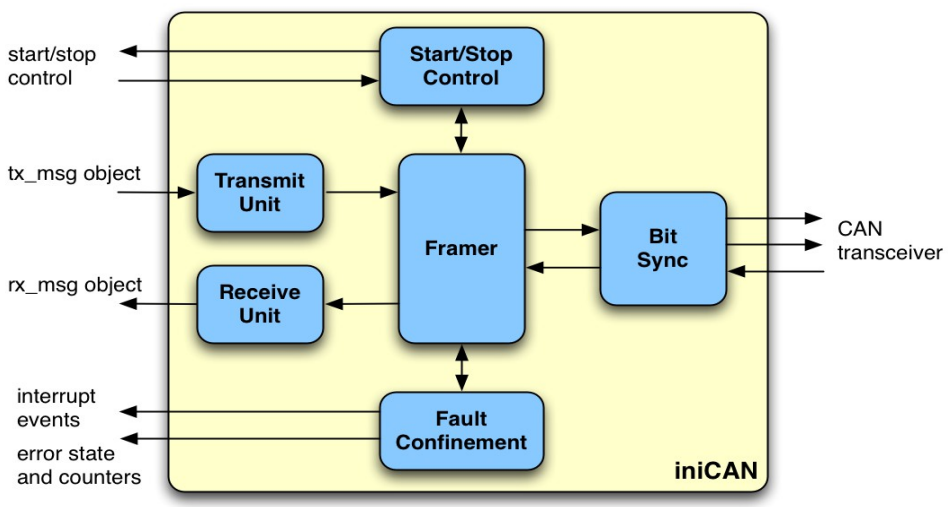

Figure 1: Block Diagram

2 Signal Descriptions

The following paragraph lists the input and output ports of the iniCAN core and provides a detailed description of their functionality.

2.1 I/O Ports

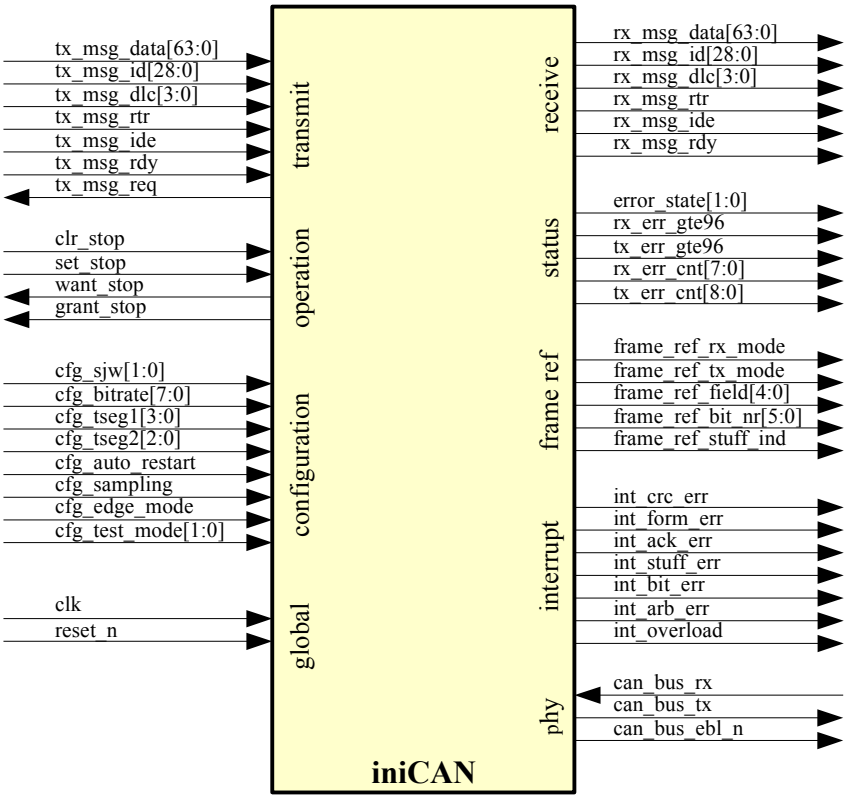

Figure 2: Inputs and Outputs

2.2 I/O Description

The following paragraphs list the inputs and outputs of the CAN controller and provides an overview of their functionality.

2.2.1 Global Signals

The module is initialized with one asynchronous, active low reset input. All registers are clocked with the system clock.

Pin Name	Type	Description
clk	in	System clock
reset_n	in	Asynchronous system reset, active low

2.2.2 CAN Controller Configuration

Configuration settings are static and may only be changed when the CAN controller is stopped.

Pin Name	Type	Description
cfg_bitrate[7:0]	in	Bitrate prescaler cfg_bitrate defines how many clock cycles a time quantum (TQ) lasts. 00h: 1 clock cycle per TQ 01h: 2 clock cycles per TQ ..... FFh: 256 clock cycles per TQ
cfg_tseg1[3:0]	in	Time segment 1 Length of the first time segment. cfg_tseg1 = 0 and cfg_tseg1 = 1 are not allowed!
cfg_tseg2[2:0]	in	Time segment 2 Length of the second time segment. cfg_tseg2 = 0 is not allowed, cfg_tseg2 = 1 is only allowed for direct sampling mode.

Pin Name	Type	Description
cfg_sjw[1:0]	in	Synchronization jump width Please note: sjw ≤ TSEG1 *and* sjw ≤ TSEG2 The effective value is the programmed value plus one.
cfg_sampling	in	Defines the sampling mode for the incoming message 0: One sampling point is used in the receive path 1: 3 sampling points with majority decision are used
cfg_edge_mode	in	Defines which edges of the incoming message are used for resynchronization: 0: Edge from 'R' to 'D' is used for synchronization[1] 1: Both edges are used 'R' to 'D' and 'D' to 'R'
cfg_auto_restart	in	Defines if the iniCAN should automatically restart after a bus-off 0: After bus-off, the CAN controller must be restarted 'by hand' using the clr_stop signal. This is the recommended setting. 1: After bus-off, the CAN controller restarts automatically after 128 groups of 11 recessive bits.
cfg_testmode[1:0]	in	Test Mode Operation 0: Normal Operation 1: Listen only mode 2: External loop back 3: Internal loop back

CAN Bit-Timing Configuration

Using cfg_tseg1 and cfg_tseg2, the effective sampling point within a bit-time can be selected. It is important that within a CAN network, all nodes use the same bit-rate and therefore the same bit-timing.

1 R: Recessive level; D: Dominant level

A bit-time consist of following four fields:

- Sync_Seg
  The synchronization segment of the bit-time is used to synchronize the various CAN nodes on the bus. An edge is expected within this segment. It is always one time quantum (TQ).
- Prop_Seg
  The propagation time segment is used to compensate physical delay times within the network. These delay times consist of the signal propagation time on the bus and the internal delay time of the CAN nodes. This is programmable from 1 to 8 time quanta (TQ)
- Phase_Seg1, Phase_Seg2
  The phase buffer segment 1 and 2 are used to compensate for edge phase errors. These segments may be lengthened or shortened by resynchronization. These segments are programmable from 1 to 8 time quanta (TQ)

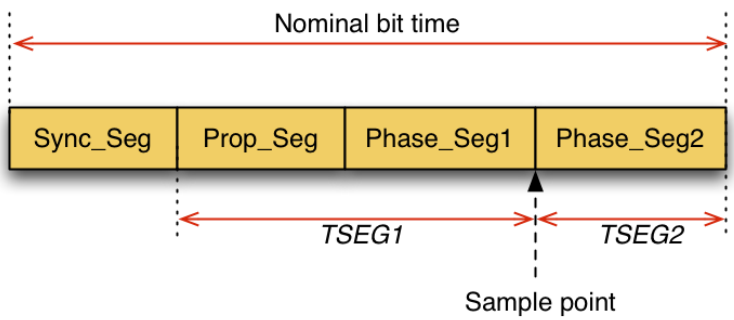

Figure 3: Bit-timing configuration

The nominal bit-time is the number of time quanta (TQ) per bit:

$$bit\ time = 1 + TSEG1 + TSEG2$$

The configured value is always the effective value minus one:

cfg_tseg1 = TSEG1 – 1; cfg_tseg2 = TSEG2 – 1

Following restrictions need to be observed

- cfg_tseg1 = 0 and cfg_tseg1 = 1 are not allowed
- cfg_tseg2 = 0 is not allowed
- cfg_tseg2 = 1 may only be used in direct sampling mode

CAN Bit-Rate

The time quantum TQ is derived from the system clock using the programmable bit-rate prescaler:

$$TQ = \frac{cfg\_bitrate + 1}{f_{clk}}$$

The effective bit rate is

$$f_{bit\,rate} = \frac{1}{TQ\ x\ bit\,time} = \frac{f_{clk}}{(cfg\_bitrate + 1)\ x\ bit\,time}$$

Example: For a 1Mbps CAN system running at 16MHz, the bit timing parameters are:

cfg_tseg1 = 3
cfg_tseg2 = 2
cfg_bitrate = 1

Test Modes Overview

A special test mode is available for diagnostic purposes.

cfg_testmode	Comment
0	Normal operation
1	Listen only mode The CAN controller receives all bus traffic but doesn't send any information to the bus. This feature is useful for automatic bus speed detection.
2	External loop back The CAN controller participates in the regular CAN transmission and reception. Additionally, a copy of all sent messages is received. This mode works only if at least one additional CAN node is on the network.
3	Internal loop back The CAN controller receives the sending data. No data is sent to the network and no data is received.

2.2.3 Start – Stop Control

The operating state of the iniCAN core is controlled using the clr_stop and set_stop inputs. The stop status is reported using want_stop and grant_stop.

Pin Name	Type	Description
clr_stop	in	Clear stop mode Event[2] sets the iniCAN in the 'run' mode. After reset, the CAN changes to 'stop' mode after the synchronization phase.
set_stop	in	Set stop mode Event sets the iniCAN in the 'stop' mode, as soon as the protocol allows it (bus idle). So no protocol errors are generated when the CAN is stopped.
want_stop	out	Stop mode request pending 1: A user stop request is pending. The CAN controller will stop as soon as possible (eg, when the bus becomes idle) 0: Not stop request is pending
grant_stop	out	Stop mode request granted 1: CAN controller is in stop mode 0: CAN controller is running

2.2.4 Status and Error Counters

These pins are used to check the status and to trace the protocol.

Pin Name	Type	Description
error_state[1:0]	out	Informs about the CAN controller error state: "00": error active (normal operation) "01": error passive "1x": bus off
rx_err_gte96	out	Receiver error count is greater or equal to 96_{dec} When the receive error counter is greater or equal 96_{dec}, this signal is activated (= '1') to indicate a highly disturbed bus.
tx_err_gte96	out	Transmit error count is greater or equal to 96_{dec} When the transmit error counter is greater or equal 96_{dec}, this signal is activated (= '1') to indicate a highly disturbed bus.

2 An event is considered a signal that is high for one clock cycle.

Pin Name	Type	Description
rx_err_cnt[7:0]	out	Receive error count The receive error counter represents the error value according to the CAN standard. When in bus-off state, the counter is used to count 128 times 11 recessive bits upon which the CAN controller may be come error active again, if enabled, by setting cfg_auto_restart.
tx_err_cnt[8:0]	out	Transmit error count The transmit error counter represents the transmit error value according to the CAN standard.

2.2.5 Interrupt Events

Interrupt events are used to inform the system of certain low-level CAN activities:

Pin Name	Type	Description
int_crc_err	out	A CRC error was detected.
int_form_err	out	A CAN message form error was detected.
int_ack_err	out	A CAN message acknowledgment error was detected.
int_stuff_err	out	A bit stuffing error was detected.
int_bit_err	out	A bit error was detected.
int_arb_loss	out	An arbitration loss happened while sending a message.
int_overload	out	An overload frame was received.

Note: An interrupt event is valid when sampled high with the rising edge of the clock.

2.2.6 CAN Frame Reference

The CAN frame reference provides additional information about the current operation of the CAN controller. The frame reference points to the current bit in a CAN frame and indicates whether the controller is in receive mode or in transmit mode.

These information can be used for CAN debugging and bus analysis.

Pin Name	Type	Description
frame_ref_rx_mode	out	Active “1” when in receive mode
frame_ref_tx_mode	out	Active “1” when in transmit mode
frame_ref_field[4:0]	out	Current CAN Frame Field 00h: Stopped 01h: Synchronize 05h: Interframe 06h: Bus idle 07h: Start of frame 08h: Arbitration 09h: Control 0Ah: Data 0Bh: CRC 0Ch: Acknowledge 0Dh: End of frame 10h: Error flag 11h: Error echo 12h: Error delimiter 18h: Overload flag 19h: Overload echo 1Ah: Overload delimiter Others: Reserved
frame_ref_bit_nr[5:0]	out	Actual bit number in the message field
frame_ref_stuff_ind	out	Active “1”when a stuff bit is inserted

2.2.7 Transmit Interface

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 tx_msg_req high and wait until the tx_msg_rdy event indicates, that the message has been sent completely and error free. All applied data must remain stable as long as tx_msg_req is active!

Pin Name	Type	Description
tx_msg_data[63:0]	in	Transmit data field [63:56]: CAN byte 1 [55:48]: CAN byte 2 [47:40]: CAN byte 3 [39:32]: CAN byte 4 [31:24]: CAN byte 5 [23:16]: CAN byte 6 [15:8]: CAN byte 7 [7:0]: CAN byte 8
tx_msg_id[28:0]	in	Transmit identifier For extended identifier: [28:0]: ID bits For standard identifier: [28:18]: ID bits [10:0] [17:0]: don't care
tx_msg_dlc[3:0]	in	Transmit Data Length Code. Invalid values are transmitted as they are set, but the number of data bytes is limited to eight. 0x0: Data length is 0 byte 0x1: Data length is 1 byte, data[63:56] is used ... 0x8: Data length is 8 bytes, data[63:0] is used 0x9-0xF: Data length is 8 bytes
tx_msg_rtr	in	Remote transmission request bit 0: Send a data frame 1: Send a remote frame
tx_msg_ide	in	The transmit extended identifier bit 0: Send a standard frame (11-bit identifier) 1: Send an extended frame (32-bit identifier)
tx_msg_rdy	out	Transmit message ready event 0: Transmit message not sent 1: Transmit message was sent

Pin Name	Type	Description
tx_msg_req	in	Transmit message request 0: No transmit request is pending 1: Tx message is valid and requested to be sent Note: While a transmit message request is pending, the message itself may not be changed. Use tx_msg_rdy to clear the transmit request.

Message transmit procedure

1) Apply tx_msg object (tx_msg_data, tx_msg_id, tx_msg_ide, tx_msg_dlc, and tx_msg_rtr).
2) Asserts tx_msg_req to request transmission of tx_msg object. While tx_msg_req is asserted, the tx_msg object may not be changed.
3) Once the bus is idle, the CAN controller starts to send the message.[3]
4) Upon successful transmission of the message, tx_msg_rdy is asserted for one clock cycle.
5) The user must release tx_msg_req once tx_msg_rdy is sampled high.

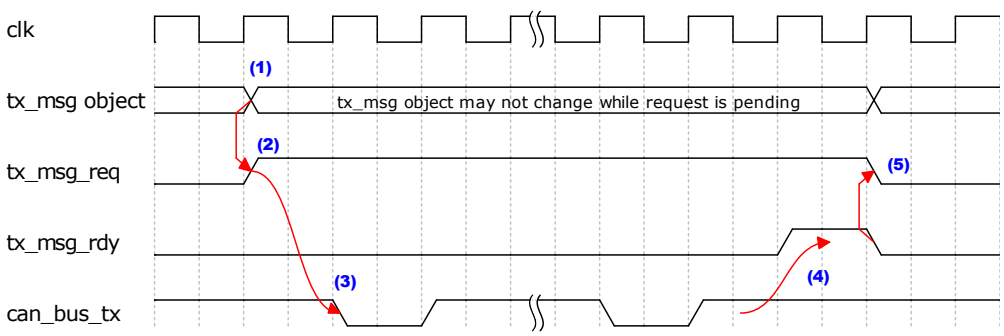

Figure 4: Transmission control

Message abort procedure

As shown in previous paragraph, once the CAN message transmit request is asserted, the message may not be modified and the message transmit request must remain asserted until the end of the message transmission.

3 Please note that the can_bus_tx signal is only shown as illustration and the message bits are not properly scaled to the clock signal.

If the currently pending message needs to be changed (e.g., an alarm message with a higher priority needs to be sent), then the transmit request can be released upon detection of one of the following interrupt events:

- Arbitration loss
- Bus error
  - CRC error
  - Format error
  - Acknowledgment error
  - Bit stuffing error
  - Bit error

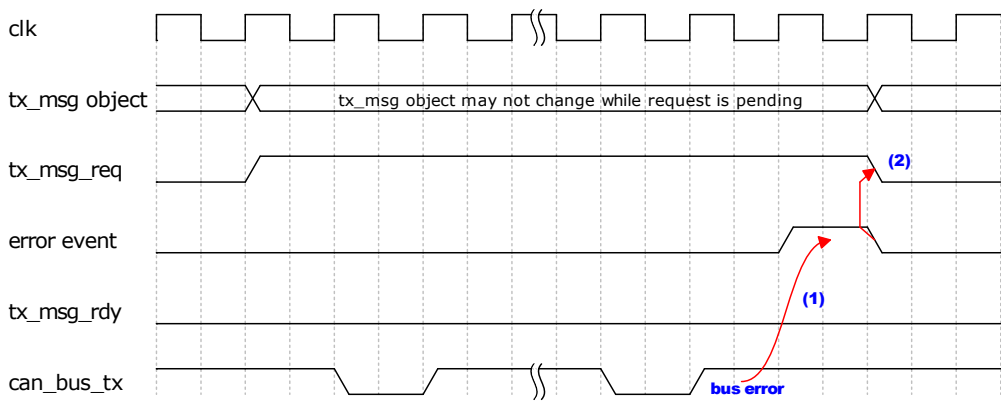

Figure 5: Transmit message abort

1) If the CAN controller detects a bus error the respective interrupt event flag is asserted.
2) If such an error event is detected, the tx_msg_req may be released

2.2.8 Receive Interface

The following list contains all needed signals for receiving messages.

Pin Name	Type	Description
rx_msg_data[63:0]	out	Receive data [63:56]: CAN byte 1 [55:48]: CAN byte 2 [47:40]: CAN byte 3 [39:32]: CAN byte 4 [31:24]: CAN byte 5 [23:16]: CAN byte 6 [15:8]: CAN byte 7 [7:0]: CAN byte 8
rx_msg_id[28:0]	out	Receive identifier For extended identifier: [28:0]: ID bits For standard identifier: [28:18]: ID bits [10:0] [17:0]: all ones
rx_msg_dlc[3:0]	out	Receive data length code: 0x0: Data length is 0 byte 0x1: Data length is 1 byte, data[63:56] is valid ... 0x8: Data length is 8 bytes, data[63:0] is valid 0x9-0xF: Data length is 8 bytes
rx_msg_rtr	out	Receive remote transmission request bit: The RTR signal is valid when rx_msg_rdy = 1 0: Received regular message 1: Received RTR message (rx_msg_data not valid)
rx_msg_ide	out	Receive extended identifier The IDE signal is valid when rx_msg_rdy = 1‘ 0: Received standard format message (11-bit identifier) 1: Received extended format message (29-bit identifier)
rx_msg_rdy	out	Receive message ready 0: rx_msg object is not valid 1: rx_msg object is valid An event for communicating that a new message has arrived. Use it for storing the RTR, IDE, DLC, ID and DATA fields!

Message Reception

The following figure shows how a message is received.

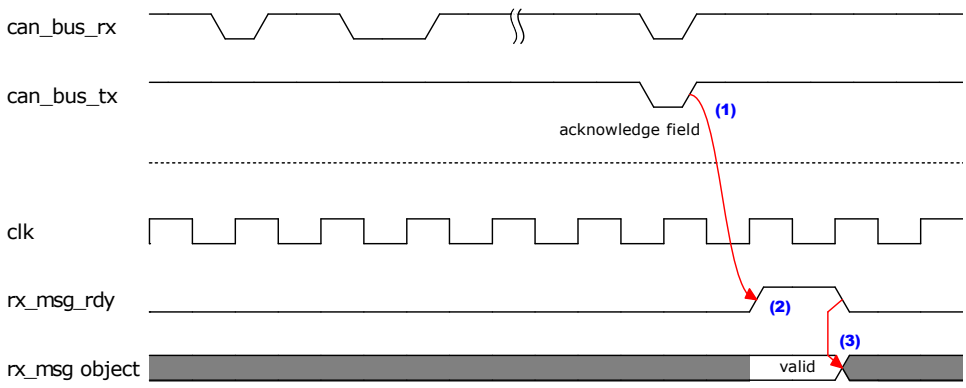

Figure 6: Message reception

1) The end of a CAN message is indicated by the acknowledgment bit
2) The CAN controller asserts rx_msg_rdy to indicate that a valid message has been received
3) The user must register the rx_msg object upon sampling rx_msg_rdy = 1. Afterwards, it is not guaranteed that the rx_msg object is still valid!

2.3 CANbus

Two or three I/Os are required to connect the CAN core to an external CAN transceiver.

Pin Name	Type	Description
can_bus_rx	in	CANbus receive signal Connect to RXD output of external driver
can_bus_tx	out	CANbus transmit signal Connect to TXD input of external driver
can_bus_ebl_n	out	CANbus transmit enable for external driver control 0: CAN controller is operational 1: CAN controller is stopped or bus-off

The following picture shows how to connect the three pins ton an CAN transceiver chip:

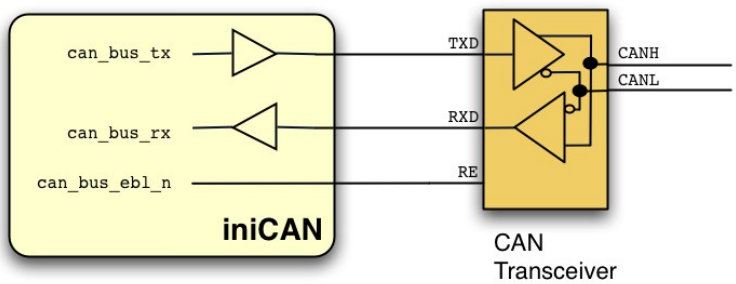

Figure 7: 3 Pin CANbus Interface

To minimize the number of pins used, a two port configuration is also possible:

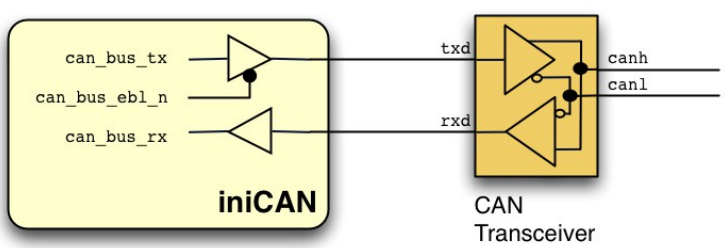

Figure 8: 2 Pin CANbus Interface

3 Top-Level Generics/Parameters

The behavior of the core can be customized to a particular application using a set of top-level generics/parameters.

Name	Default	Description
G_ERROR_COUNTER_RESET	1	Error Counter Reset When the core is started by asserting `clr_stop`, the receive and transmit error counters are automatically reset. 0: The error counters preserve their value (original implementation) 1: The error counters are reset
G_DSYNCH_EBL	0	Double Synchronization Enable The core can be configured to use double-synchronization on the `can_bus_rx` pin by setting `G_DSYNCH_EBL = 1`. *This feature may only be used with `cfg_bitrate >= 1`!*

4 Application Notes

4.1 Automatic bitrate detection

Using the CAN controller's listen-only mode, non intrusive bus observation can be used to determine the actual bitrate. During the bitrate detection, the CAN controller will listen to the on-going CAN bus communication using a set of given bitrates and eventually will detect the actual bitrate.

The procedure to detect the bitrate is shown in following flowchart:

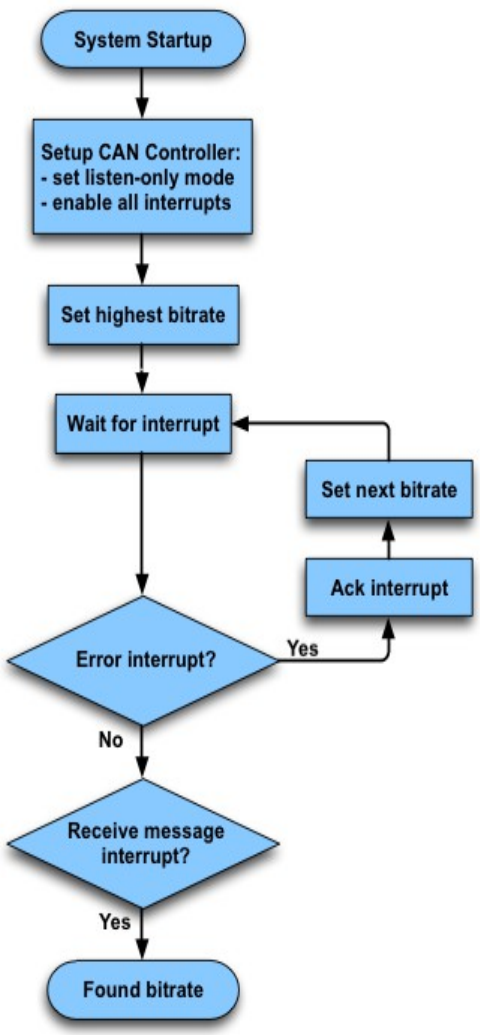

Figure 9: Automatic bitrate detection flowchart

Inicore is a leading Intellectual Property (IP) core and design solution provider. Our mission is to supply pre-verified, technology neutral, and reusable IP cores for a wide range of target markets from consumer goods to avionics and aerospace.

Our IP cores are complemented by comprehensive design service offerings:

- FPGA and ASIC Turn-Key Solutions
- Embedded System Design
- IP Core Design and Integration
- Consulting Services
- ASIC to FPGA Migration Service
- Obsolete Part Replacement

We can quickly provide you with an FPGA-, SoC- or Embedded System solution, leveraging our IP know-how and broad application-specific expertise. Our experience in microelectronic system integration allows us to guide you through the entire design flow from concept to final products. We help you with feasibility studies, concept analysis, system specification, design implementation and verification. Additionally, we do custom IP and low-level software development. We also handle everything from board design through fabrication and assembly.

Our development process is based on Structured Analysis & Structured Design (SA/SD) methodology that we apply to FPGA as well as ASIC projects. Verification testbenches rely on Transaction Based Verification (TBV) methods. Both these methodologies lead to reusable design and verification components. By planning for reusability, we set a solid base for further developments in the ever-decreasing product design - and life cycle.

Customer Advantages

We offer one-stop shopping for everything from the specifications to the chip or module implementation. It is our aim to engage with your engineering team and complement them in order to create your FPGA based system-on-chip solutions. This assistance, added to the ability to reuse our pre-designed and pre-verified IP cores, dramatically reduces design risks and execution time, and helps to successfully bring your product to the market.

Visit us @ www.inicore.com