INICORE

Datasheet

CANmodule-IIx

Version 2.7.4

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

Figure Index

Definition of Terms

Following conventions are used in this document:

- Message Byte 1..8 -> D_63..D_0
- All default values are ‘0’ unless otherwise noted
- Undefined bits in read back are read as ‘0’
- Following nomenclature is used register mapping
  - r : Readback operation
  - R : Read operation
  - W : Write operation
- Signals ending with ‘_n’ are active low

Revision History

Version	*Comment*
2.7.4	• Added separate reset for CAN clock domain
2.7.3	• Fixed typos • Clarified description of rx_ovr interrupt
2.7.2	• Updated CAN transceiver diagram • Added new section parameter • Updated description of configuration register • Added automatic bitrate detection flowchart
2.7.1	• Removed pbenable[3:0] signal from I/O diagram
2.7.0	• Added configuration option to select endianness of CAN data field • Refined CAN data field description
2.6.5	• Corrected interrupt control register address (page 12)
2.6.4	• Added CAN message filter example, (page 20) • Removed readback on transmit buffers (pages 11, 14 & 15) • Added identifier bit mapping for standard ID frames (p 13 & 18)
2.6.3	• rx_err_cnt: changed counter description, more expressive
2.6.2	• Corrected typing errors
2.6.1	• Refined DLC, RTR, and IDE bit description • General document update
2.6.0	• Added cclk • Corrected rx_fill_level configuration description

1 Overview

CANmodule-IIx is a full functional CAN controller module that contains advanced message filtering, and receive-, and transmit buffers. It is designed to provide a low gate-count CAN interface for FPGA and ASIC based system-on-chip (SOC) integrations.

Full message filtering together with a transmit FIFO and a high priority transmit message buffer support a wide range of applications. An AMBA Advanced Peripheral Bus (APB) interface enables smooth integration into ARM based SOC's.

1.1 Features

The CANmodule-IIx is designed for a system-on-chip design.

Standard Compliant

- Full CAN 2.0B compliant
- Supports standard CAN baud rates including 1 Mbps

3 Programmable Acceptance Filters

- Message filter covers: ID, IDE, RTR, Data byte 1 and Data byte 2
- User selectable number of filters

Receive Path

- 32 messages deep receive FIFO
- FIFO status indicator
- System time-stamp

Transmit Path

- 16 messages deep transmit FIFO
- 1 message buffer for high priority messages to bypass transmit FIFO
- Message Arbiter

System Bus Interface

- AMBA 2.0 Advanced Peripheral Bus Interface
- 8-bit, 16-bit, or 32-bit wide data path
- Status and configuration interface

Programmable Interrupt Controller

- Local interrupt controller covering message and CAN error sources

Supports FPGA systems with two clock domains

- System clock (fast clock)
- CAN clock (slow clock)

Test and Debugging Support

- Listen only mode
- Internal loopback mode
- External loopback mode

SRAM Based Message Buffers

- Optimized for low gate-count implementation
- 100% Synchronous Design

1.2 Implementation Options

Several special implementation options are available for gate count optimized implementations. These options have to be configured prior to synthesizing the design using parameters/generics.

- Configuration register read-back enable
  To minimize gate count, the configuration register read-back path can be disabled
- Two separate clock domains
  A dedicated CAN clock is available when the system clock is too high for the CAN core. This feature can be disabled by a configuration entry.
- Fixed configuration
  For gate count optimized FPGA implementations, it might be desirable to set the configuration register to a fixed value.
- Message filter support
  3 local message filters can individually be selected. This provides the option of having 0, 1, 2, or 3 CAN message filters available for the target application. Additionally, filtering on the data field portion of the CAN message can be disabled.
- High-priority transmit message buffer
  The high-priority transmit message buffer TxMessage0 can be disabled if not needed in the system.

Apart of the 32-bit APB interface, the core can be used in a 8-bit and 16-bit APB bus system too. Dedicated wrappers are provided as a standard deliverable.

1.3 Block Diagram

The main building blocks are shown in the following figure:

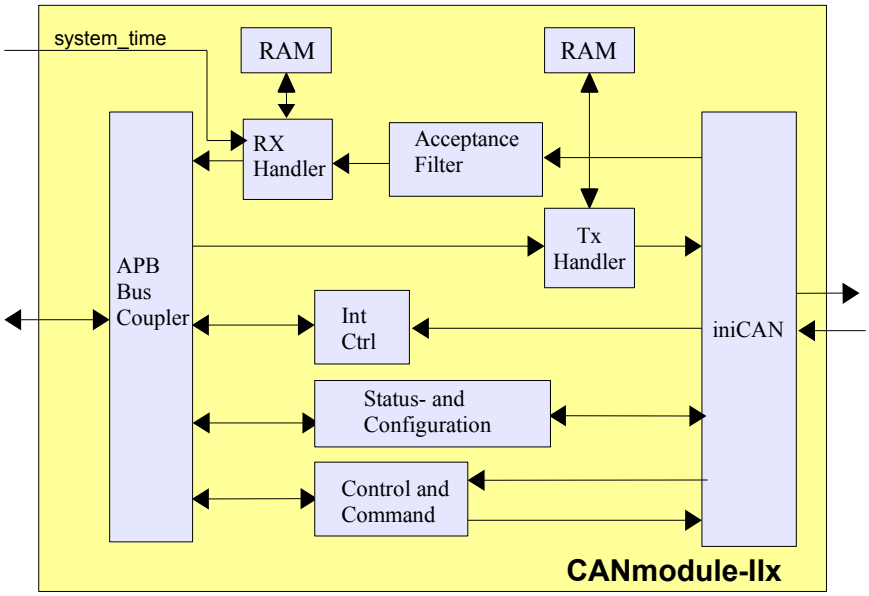

Figure 1.1: Block Diagram

1.3.1 On-Chip RAM

To optimize system performance and on-chip resources, the size of the FIFOs can be selected. Register based implementations are supported as well.

The memory resource requirements for the default configuration are as follows:

- With 32-bit wide data path: 2 x SRAM 128x32
- With 16-bit wide data path: 2 x SRAM 256x16
- With 8-bit wide data path: 2 x SRAM 512x8

2 IO Description

The following paragraph lists the input and output ports of this core and explains their respective functionality.

2.1 Inputs – Outputs

This picture shows the main inputs and outputs.

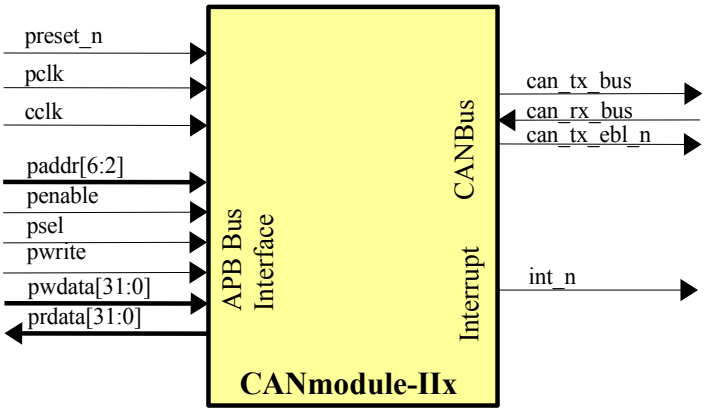

Figure 2.1: Input and Outputs

2.2 General Inputs

These pins are used to clock and initialize the whole core. There are no internally generated clocks or resets.

Pin Name	*Type*	*Description*
pclk	in	System clock
cclk	in	CAN clock
preset_n	in	Asynchronous reset of system clock domain, active low
creset_n	in	Asynchronous reset for CAN clock domain, active low

Two different clock domains are available to help FPGA systems where the main clock is much faster than the CAN clock. Each clock domain has its own asynchronous reset.

It is recommended to generate the reset for the CAN clock domain using a circuit such as shown in figure 2.2.

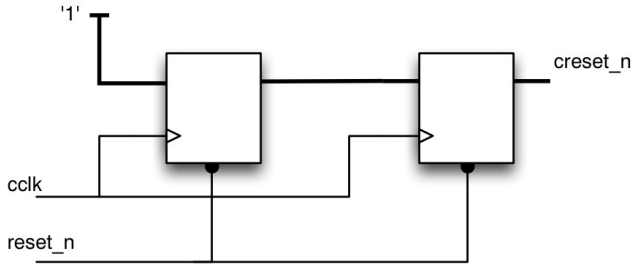

Figure 2.2: CAN reset generation

2.3 APB Bus Interface

The on-chip bus interface is compliant to the AMBA 2.0 APB bus specification. The interface is full synchronous to the system clock. The interface supports true 32-bit access with zero wait-states. 8-bit and 16-bit access are supported through separate byte enable signals.

Pin Name	*Type*	*Description*
psel	in	Module select signal
penable	in	Bus transfer enable signal
paddr[6:2]	in	Address bus
pwrite	in	Read/write signal ‘0’: read operation ‘1’: write operation
pwdata[31:0]	in	Write data bus
prdata[31:0]	out	Read data bus
int_n	out	Interrupt request, active low

2.4 CAN Bus Interface

Three signals are provided to directly connect to a CANbus transceiver.

Pin Name	*Type*	*Description*
can_rx_bus	in	Local receive signal (connect to can_rx_bus of external driver)
can_tx_bus	out	CANbus transmit signal, connected to external driver
can_tx_ebl_n	out	External driver control signal

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

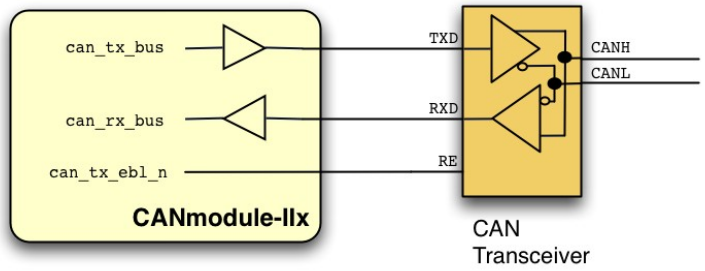

Figure 2.3: 3 Pin CANbus Interface

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

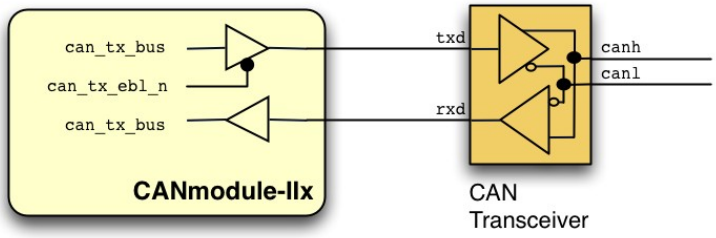

Figure 2.4: 2 Pin CANbus Interface

3 Memory Map

The table below shows the entire memory map of the CANmodule-IIx function. All registers are 32-bit wide. Following nomenclature is used to differentiate different bus access:

r : Data read back operation. Read back of configuration registers
R : Read operation
W : Write operation

Default value for all register if not otherwise noted is 0x00.

3.1 Memory map of all internal registers:

Address	*R/W*	*Description*
0x00	W	TxMessage0 Buffer
0x04		
0x08		
0x0C		
0x10	W	TxMessage FIFO
0x14		
0x18		
0x1C		
0x30	R	RxMessage Buffer
0x34		
0x38		
0x3C		
0x40	r/W	Acceptance Register 0
0x44		
0x48		
0x4C	r/W	Acceptance Register 1
0x50		
0x54		
0x58	r/W	Acceptance Register 2
0x5C		
0x60		

Address	R/W	Description
0x64	r/W	Acceptance Configuration Register
0x68	R/W	Error Status Indicator
0x6C	R/W	Interrupt Control
0x70		
0x74		
0x78	R/W	CAN Controller Operating Mode
0x7C	r/W	CAN Controller Configuration

3.2 Internal Register Description

This paragraph shows all internal registers and describes how the CANmodule-IIx can be used and programmed.

3.2.1 Transmit Message Registers

This CAN controller provides two different transmit paths. One message buffer (TxMessage0) is dedicated for high priority messages, while a second 16 message deep buffer is organized as a FIFO.

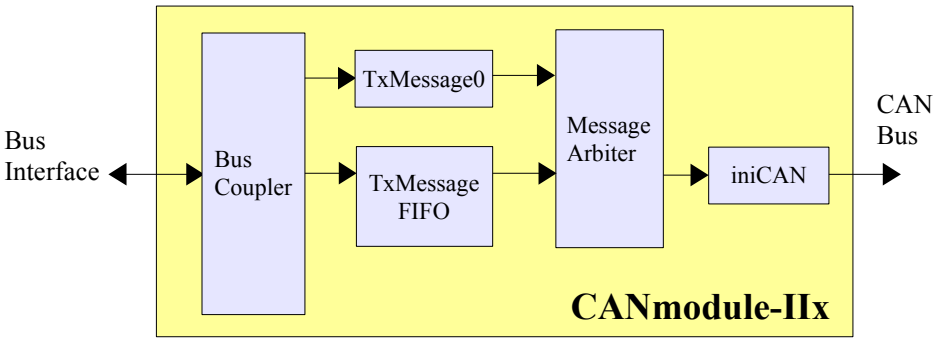

Figure 3.1: Transmit Path

Message Arbitration

A message residing in TxMessage0 is always sent prior to sending a message from the TxMessage FIFO.

- High priority messages can always be sent
- Low priority messages can be queued to reduce CPU overhead

Register Description:

Address	*R/W*	*Name*	*Comment*
0x00	W	TxMessage0	TxMessage0 Buffer: Message Identifier Field For extended identifier: [31:3]: ID bits [28:0] [2:0]: don't care For standard identifier: [31:21]: ID bits [11:0] [20:0]: don't care
0x04	W		TxMessage0 Buffer: Data low The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 1 [23:16]: CAN data byte 2 [15:8]: CAN data byte 3 [7:0]: CAN data byte 4 swap_endian = 1: [31:24]: CAN data byte 4 [23:16]: CAN data byte 3 [15:8]: CAN data byte 2 [7:0]: CAN data byte 1
0x08	W		TxMessage0 Buffer: Data high The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 5 [23:16]: CAN data byte 6 [15:8]: CAN data byte 7

Address	R/W	Name	Comment
			[7:0]: CAN data byte 8 swap_endian = 1: [31:24]: CAN data byte 8 [23:16]: CAN data byte 7 [15:8]: CAN data byte 6 [7:0]: CAN data byte 5
0x0C	W		TxMessage0 Buffer: Control Flags [23]: WPN, Write Protect Not '0': Bit [21:16] remain unchanged '1': Bit [21:16] are modified. This bit is zero for readback [21]: RTR, Remote Bit '1': This is an RTR message '0': This is a standard message [20]: IDE, Extended Identifier Bit '1': This is an extended format message '0': This is a standard format message
0x0C	W		TxMessage0 Buffer: Control Flags [19:16]: DLC, 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 TxMessage0 Buffer: Command Flags [7]: WPN: Write protect not. '0': Bit [0] remains unchanged '1': Bit [0] is modified. This bit is always zero for readback [0]: TxReq, Transmit Request '1': Transmit request '0': Idle
	R		TxMessage0 Buffer: Control Flags [0]: TxReq, Transmit Request

Address	R/W	Name	Comment
			'1': Transmit request pending '0': Idle
0x10	W	TxMessage-FIFO	TxMessageFIFO Buffer: Identifier [31:0]: Message Identifier, ID bits [28:3] [2:0]: N/A
0x14	W		TxMessageFIFO Buffer: Data low The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 1 [23:16]: CAN data byte 2 [15:8]: CAN data byte 3 [7:0]: CAN data byte 4 swap_endian = 1: [31:24]: CAN data byte 4 [23:16]: CAN data byte 3 [15:8]: CAN data byte 2 [7:0]: CAN data byte 1
0x18	W		TxMessageFIFO Buffer: Data high The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 5 [23:16]: CAN data byte 6 [15:8]: CAN data byte 7 [7:0]: CAN data byte 8 swap_endian = 1: [31:24]: CAN data byte 8 [23:16]: CAN data byte 7 [15:8]: CAN data byte 6 [7:0]: CAN data byte 5
0x1C	W		TxMessageFIFO Buffer: Control Flags [23]: WPN, Write Protect Not

Address	R/W	Name	Comment
			'0': Bit [21:16] remain unchanged '1': Bit [21:16] are modified. This bit is zero for readback [21]: RTR, Remote Bit '1': This is an RTR message '0': This is a standard message [20]: IDE, Extended Identifier Bit '1': This is an extended format message '0': This is a standard format message [19:16]: DLC, 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 [7]: WPN: Write protect not. '0': Bit [0] remains unchanged '1': Bit [0] is modified. This bit is always zero for readback [0]: TxReq, Transmit Request '1': Transmit request '0': Idle
	R		TxMessageFIFO Buffer: Control Flags [0]: TxReq, Transmit Request '1': Transmit request pending '0': Idle

1. Byte 1 is Data[63:56], Byte 2 is Data[55:48], etc.

Procedure for sending a message using TxMessage0

- First check that the transmit message buffer is empty. This is indicated by TxReq = '0'.
- Write message into the transmit message holding register.
- Request transmission by setting the TxReq flag. This flag remains set as long as the message holding registers contains this message. The content of the message buffer must not be changed while the TxReq flag is set!
- The TxReq flag remains set as long as the message transmit request is pending
- The internal message priority arbiter selects the message with the highest priority to be sent next.
- The successful transfer of a message is indicated by the respective tx_xmit interrupt and by releasing the TxReq flag.

Procedure for sending a message using TxMessageFIFO

- First check that the transmit message FIFO is empty. This is indicated by TxReq = '0'.
- Write message into the transmit message holding FIFO.
- Request transmission by setting the TxReq flag. The content of the message buffer must not be changed while the TxReq flag is set!
- The TxReq flag is released once a new TxMessage FIFO buffer becomes available.
- The successful transfer of a message is indicated by the tx_xmit_fifo interrupt. Depending on the tx_level configuration settings, an additional interrupt source tx_msg is available to indicate that the transmit FIFO is empty or below a certain level.

3.2.2 Rx Message Buffers

Received messages are stored in a 32 messages deep FIFO. Status indicators are provided to show how many messages are available.

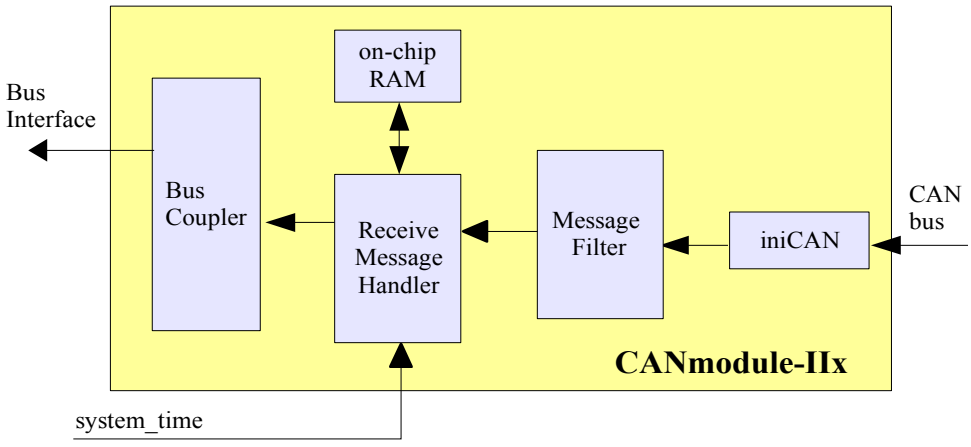

Figure 3.2: Receive Path

Time Stamp

Using an external system time reference, all incoming messages are time-stamped. This enables the system to keep track of when a particular message arrived.

Procedure for reading received messages

The following sequence outlines the recommended Rx message handling:

- Wait for rx_msg interrupt
- MessageReadLoop:
  - read message
  - acknowledge ‘message read’ by writing a ‘1’ to MsgAck register
  - read MsgValid; reading a ‘1’ means a new message is available
  - IF MsgValid=1 THEN jump to MessageReadLoop
- Acknowledge rx_msg interrupt by writing a ‘1’ to this register location

Register Description:

Address	*R/W*	*Name*	*Comment*
0x30	R	RxMessage	RxMessage Buffer: Identifier For standard identifier: [31:21]: Identifier bits ID[11:0] [20:0]: fixed at one For extended identifier: [31:3]: ID bits [28:0] [2:0]: fixed at one
0x34	R		RxMessage Buffer: Data low The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 1 [23:16]: CAN data byte 2 [15:8]: CAN data byte 3 [7:0]: CAN data byte 4 swap_endian = 1: [31:24]: CAN data byte 4 [23:16]: CAN data byte 3 [15:8]: CAN data byte 2 [7:0]: CAN data byte 1
0x38	R		RxMessage Buffer: Data high The byte mapping can be set using the CAN swap_endian configuration bit. swap_endian = 0, default: [31:24]: CAN data byte 5 [23:16]: CAN data byte 6 [15:8]: CAN data byte 7 [7:0]: CAN data byte 8 swap_endian = 1: [31:24]: CAN data byte 8 [23:16]: CAN data byte 7 [15:8]: CAN data byte 6

Address	R/W	Name	Comment
			[7:0]: CAN data byte 5
0x3C	R		RxMessage Buffer: Status [31:16]: Captured receive time. Once a message is received, the system time is captured and stored in this location. [10:8]: Acceptance Filter Indicator “xx1”: Filter 0 accepted message “x1x”: Filter 1 accepted message “1xx”: Filter 2 accepted message [6]: RTR, Remote Bit '1': This is an RTR message '0': This is a regular message [5]: IDE, Extended Identifier Bit '1': This is an extended format message '0': This is a standard format message [4:1]: DLC, Data Length Code. Invalid values are shown as received. 0x0: Message length is 0. data[63:0] is not valid 0x1: Message length is 1. data[63:56] is valid ... 0x8: Message length is 8. data[63:0] is valid [0]: MsgValid, Message Valid '0': Current message is not valid '1': Current message is valid
0x3C	W	*RxMessage*	RxMessage Buffer: Control [31:1]: N/A [0]: MsgAck, Message Acknowledge '0': idle '1': Acknowledges receipt of new message Acknowledging a message clears the MsgValid flag. If a new message is directly available from the receive FIFO then this flag remains set to indicate that an additional message is available.

2. Byte 1 is Data[63:56], Byte 2 is Data[55:48], etc.

3.2.3 Acceptance Filter and Acceptance Code Mask

Three programmable Acceptance Mask Register (AMR) and Acceptance Code Register (ACR) pairs are available to filter incoming messages. Each pair can be individually enabled through an Acceptance Filter Enable (AFE) register.

Following fields are covered:

- ID
- IDE
- RTC
- DATA[63:48]

The acceptance mask register (AMR) defines whether the incoming bit is checked against the acceptance code register (ACR).

AMR: '0': The incoming bit is checked against the respective ACR. The message is discarded when the incoming bit doesn't match the respective ACR flag.

'1': The incoming bit is don't care.

Register Description:

Address	*R/W*	*Name*	*Comment*
0x40	r/W	AMR0	Acceptance Mask Register 0 [31:3]: Identifier [2]: IDE [1]: RTR [0]: N/A
0x44	r/W	ACR0	Acceptance Code Register 0 [31:3]: Identifier [2]: IDE [1]: RTR [0]: N/A
0x48	r/W	ACMR_DATA0	Acceptance Code / Mask Data Register 0 [31:16] : Acceptance Code Data Register [15:0] : Acceptance Mask Data Register
0x4C	r/W	AMR1	Acceptance Mask Register 1

Address	*R/W*	*Name*	*Comment*
0x50	r/W	ACR1	Acceptance Code Register 1
0x54	r/W	ACMR_DATA1	Acceptance Code / Mask Data Register 1
0x58	r/W	AMR2	Acceptance Mask Register 2
0x5C	r/W	ACR2	Acceptance Code Register 2
0x60	r/W	ACMR_DATA2	Acceptance Code / Mask Data Register 2
0x64	r/W	AFE	Acceptance Filter Enable Register [2]: AFE2 : Acceptance Filter 2 Enable bit ‘0’: Acceptance filter 2 is disabled ‘1’: Acceptance filter 2 is enabled [1]: AFE1 : Acceptance Filter 1 Enable bit ‘0’: Acceptance filter 1 is disabled ‘1’: Acceptance filter 1 is enabled [0]: AFE0 : Acceptance Filter 0 Enable bit ‘0’: Acceptance filter 0 is disabled ‘1’: Acceptance filter 0 is enabled

If all three message filters are disabled then no messages will be received! To receive all messages then at least one message filter must be enabled and programmed with all its fields set as “don’t care”.

CAN Message Filter Example:

The following example shows the acceptance register settings used to support receipt of a CANopen TPDO1 (Transmit Process Data Object) message. In CANopen, a widely used CAN Higher Level Protocol (HLP), the ID bits are used to select the message type. The bit assignment is shown in following table:

CANopen Identifier										
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Function Code				Node-ID						

Identifier fields:

- Function Code: The function code for a TDPO1 message is 3h
- Node-ID: In our example, we use 02h as the Node ID
- IDE = '0', CANopen uses the short format message

- RTR = '0', this is a regular message

To accept this message, the acceptance filter settings would look like

AMR settings:

- ID[28:18] = 0
- ID[17:0] = all ones
- IDE = 0
- RTR = 0
- DATA[63:56] = all ones

ACR settings:

- ID[28:18] = 182h
- ID[17:0] = don't care
- IDE = 0
- RTR = 0
- DATA[63:56] = don't care

3.2.4 Error Status Indicators

Status indicators are provided to report the CAN controller error state, receive error count, and transmit error count. Special flags to report error counter values equal to or in excess of 96 errors are available to indicate heavily disturbed bus situations.

Address	*R/W*	*Name*	*Comment*
0x68	R/W	ErrorStatus	CAN Controller Error Status [19]: rxgte96 The receiver error counter is greater or equal 96(dec) [18]: txgte96 The transmitter error counter is greater or equal 96(dec) [17:16]: error_stat[1:0] The error state of the CAN node: “00”: Error active (normal operation) “01”: Error passive “1x”: Bus off

Address	R/W	Name	Comment
0x68	R/W	ErrorStatus *continued*	CAN Controller Error Status [15:8]: rx_err_cnt[7:0] The receive error counter according to the CAN 2.0 specification. When in bus-off state, this counter is used to count 128 groups of 11 recessive bits. [7:0]: tx_err_cnt[7:0] The transmit error counter according to the CAN specification. When it is greater than 255_{dec}, it is fixed at 255_{dec}.

3.2.5 Interrupt Controller

An interrupt enable register is provided to enable a particular interrupt source. This is done by setting this flag to '1'. Interrupt sources are available for regular traffic, error, and diagnostic information.

Address	R/W	Name	Comment
0x6C	r/W	Cfg_MsgTh	Cfg Message Threshold [3:2]: rx_level[1:0]: Sets the rx_msg interrupt threshold 0: Receive FIFO not empty 1: Receive FIFO at least ¼ full 2: Receive FIFO at least ½ full 3: Receive FIFO at least ¾ full [1:0]: tx_level[1:0]: Sets the tx_msg interrupt threshold 0: Transmit FIFO at least ¼ empty 1: Transmit FIFO at least ½ empty 2: Transmit FIFO at least ¾ 3: Transmit FIFO empty

Address	R/W	Name	Comment
0x70	r/W	IntEbl	Interrupt Enable Register An interrupt source is enabled by setting its respective flag to ‘1’. [15]: rx_msg [14]: tx_msg [12]: tx_xmit_fifo [11]: tx_xmit0 [10]: bus_off [9]: crc_err [8]: form_err [7]: ack_err [6]: stuff_err [5]: bit_err [4]: rx_ovr [3]: ovr_load [2]: arb_loss [0]: int_ebl, global interrupt enable flag ‘0’: All interrupts are disabled ‘1’: Enabled interrupt sources are available [*others*]: fixed to '0'
0x74	R/W	IntStatus	Interrupt Status Register A pending interrupt is indicated that its respective flag is set to ‘1’. To acknowledge an interrupt, set its flag to ‘1’ [15]: rx_msg [14]: tx_msg [12]: tx_xmit_fifo [11]: tx_xmit0 [10]: bus_off [9]: crc_err [8]: form_err [7]: ack_err [6]: stuff_errr [5]: bit_err [4]: rx_ovr [3]: ovr_load [2]: arb_loss [*others*]: N/A

Explanation of interrupt sources:

- rx_msg: Depending on rx_level, the selected number of messages are available in the receive FIFO.
- tx_msg: Depending on tx_level, the selected number of transmit buffers are empty.
- tx_xmit_fifo: Indicates that one message from the FIFO was successfully sent.
- tx_xmit0: Indicates that a message from TxMessage0 buffer was successfully sent.
- bus_off: The CAN controller has reached the bus off state.
- crc_err, form_err, ack_err, stuff_err, bit_err: Any of the mentioned error occurred while receiving or transmitting a message.
- rx_ovr: Receiver overrun. This Flag indicates that a new message arrived while the receive buffer is full. The new message is discarded.
- ovr_load: The CAN controller received an overload message.
- arb_loss: An arbitration loss happened while trying to send a message.

3.2.6 CAN Controller Operating Mode

The CANmodule can be used in different operating modes. By disabling transmitting data, it is possible to use the CAN in listen only mode enabling features such as auto - matic bit rate detection.

Address	*R/W*	*Name*	*Comment*
0x78	R/W	Command	CAN Command Register [2]: Loopback Test Mode: '0': Normal operation '1': Active [1]: Listen only mode: '0': Active '1': CAN listen only: The output is held at 'R' level. The CAN controller is only listening. [0]: Run/Stop mode: '0': Sets the CAN controller into stop mode. Read '0' when stopped '1': Sets the CAN controller into run mode. Read '1' when running.

Test modes overview

Using the loop back and the listen only flag, the CAN controller can perform certain test operation:

Loop back	*Listen only*	*Comment*
'0'	'0'	Normal operation
'0'	'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.
'1'	'1'	Internal loop back: The CAN controller receives the sending data. No data is sent to the network and no data is received.
'1'	'0'	External loop back: The CAN controller participates in the regular CAN transmission and reception. Further, a copy of all sending messages is received. This mode works only if at least one additional CAN node is on the network.

3.2.7 CAN Controller Configuration Register

See chapter 4 for additional information on setting time segment 1, time segment 2, and the bit rate.

Address	*R/W*	*Name*	*Comment*
0x7C	r/W	can_cfg	CAN Controller Configuration [24]: swap_endian The byte position of the CAN receive and transmit data fields can be modified to match the endian setting of the processor or the CAN protocol. 0: CAN data byte position is not swapped (big endian) 1: CAN data byte position is swapped (little endian) [23:16]: cfg_bitrate[7:0]: 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 The effective value is the programmed value plus one. [11:8]: cfg_tseg1: Time segment 1 Length of the first time segment. cfg_tseg1 = 0 and cfg_tseg1 = 1 are not allowed! The effective value is the programmed value plus one. [7:5]: cfg_tseg2: 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. The effective value is the programmed value plus one.

Address	R/W	Name	Comment
0x7C	r/W	can_cfg *continued*	CAN Controller Configuration [4]: auto_restart: 0: After bus off, the CAN controller must be restarted 'by hand'. This is the recommended setting. 1: After bus off, the CAN controller restarts automatically after 128 groups of 11 recessive bits. [3:2]: cfg_sjw: Synchronization jump width Please note: sjw ≤ TSEG1 *and* sjw ≤ TSEG2 The effective value is the programmed value plus one. [1]: sampling_mode: 0: One sampling point is used in the receive path 1: 3 sampling points with majority decision are used [0]: edge_mode: 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'

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.

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

1 R: Recessive level; D: Dominant level

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)

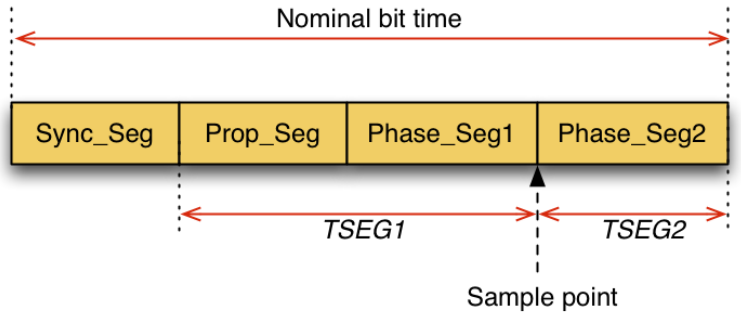

Figure 3.3: Bit-timing configuration

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

4 Parameters

Some of the CANmodule-IIx features can be controlled by top-level parameters/generics. These values are permanently set when creating the target design and can't be modified by software.

Parameter	*Default Value*	*Description*
MSG_FILTER_0_AV	1	Receive message filter 0 available If this message filter is not needed, it can be automatically removed. 0: Filter is not available 1: Filter is available
MSG_FILTER_1_AV	1	Receive message filter 1 available If this message filter is not needed, it can be automatically removed. 0: Filter is not available 1: Filter is available
MSG_FILTER_2_AV	1	Receive message filter 2 available If this message filter is not needed, it can be automatically removed. 0: Filter is not available 1: Filter is available
DATA_FILTER_AV	1	Message filtering on data field If the message filters don't use the CAN data field bytes 0 and 1, then the message filter can be reduced to just use the ID field. 0: Data filter is not available 1: Data filter is available This parameter applies to all message filters.
TX_MSG0_AV	1	High-priority buffer TxMessage0 The high-priority buffer can be disabled if the system doesn't use it. 0: TxMsg0 buffer is not available 1: TxMsg0 buffer is available

Parameter	*Default Value*	*Description*
ENABLE_CLK_SYNCH_LOGIC	1	Clock domain synchronization logic If the system clock is running higher than what the CAN clock can do or if the system clock can't be used to generate an accurate CAN bitrate, then two separate clocks may be used. In this case, the core contains proper clock synchronization logic to accommodate the clock domain crossing. 0: Both clocks are on same network 1: Enable clock domain synchronization The synchronization logic uses dual-stage resynchronization registers.
FIXED_CONFIG_EBL	0	Fixed configuration enable If the core always uses the same CAN bus configuration, a fixed setting can be used. 0: Configuration can be set by user 1: Use fixed configuration
FIXED_CONFIG_SJW	0	Fixed configuration: Synchronization jump width[2] Valid range: 0-3
FIXED_CONFIG_BITRATE	0	Fixed configuration: Bitrate prescaler[2] Valid range: 0-255
FIXED_CONFIG_TSEG1	0	Fixed configuration: Time segment 1[2] Valid range: 0-15
FIXED_CONFIG_TSEG2	0	Fixed configuration: Time segment 2[2] Valid range: 0-7
FIXED_CONFIG_AUTO_RESTART	0	Fixed configuration: Automatic restart control[2] Valid settings: 0/1

2 See paragraph CAN Controller Configuration Register on page 28 for a detailed description of this setting.

Parameter	*Default Value*	*Description*
FIXED_CONFIG_SAMPLING	0	Fixed configuration: Sampling mode[2] Valid settings: 0/1
FIXED_CONFIG_EDGE_MODE	0	Fixed configuration: Edge mode[2] Valid settings: 0/1
FIXED_CONFIG_SWAP_ENDIAN	0	Fixed configuration: Swap endian on CAN data fields[2] Valid settings: 0/1
DATA_READBACK_EBL	1	Data readback from configuration registers The readback data path from the configuration registers may be disabled. In this case, the respective configuration bits read 0. 0: Data readback is not enabled 1: Data readback is enabled

Note:

- The FIXED_CONFIG_XXX parameters control the reset state of their respective configuration registers.

5 Application Notes

5.1 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:

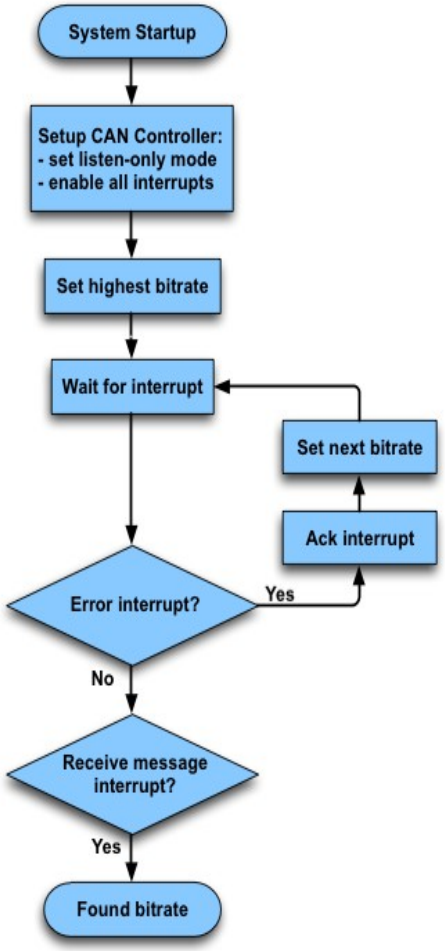

Figure 5.1: Automatic bitrate detection flowchart

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