INICORE

***Datasheet***

# CANmodule-IIx

## Version 2.7.0

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

**1 Overview.....5**

**1.1 Features.....5**

**1.2 Implementation Options.....6**

**1.3 Block Diagram .....7**

1.3.1 On-Chip RAM.....7

**2 IO Description.....8**

**2.1 Inputs – Outputs.....8**

**2.2 General Inputs.....8**

**2.3 APB Bus Interface.....9**

**2.4 CAN Bus Interface.....9**

**3 Memory Map.....11**

**3.1 Memory map of all internal registers:.....11**

**3.2 Internal Register Description.....12**

3.2.1 Transmit Message Registers.....12

3.2.2 Rx Message Buffers.....18

3.2.3 Acceptance Filter and Acceptance Code Mask.....21

3.2.4 Error Status Indicators.....24

3.2.5 Interrupt Controller.....25

3.2.6 CAN Controller Operating Mode.....27

3.2.7 CAN Controller Configuration Register.....28

**4 Configuring The CAN Controller.....30**

**4.1 Setting proper bit rate, tseg1 and tseg2.....30**

# Figure Index

Figure 1: Block Diagram.....7
Figure 2: Input and Outputs.....8
Figure 3: 3 Pin CANbus Interface.....10
Figure 4: 2 Pin CANbus Interface.....10
Figure 5: Transmit Path.....12
Figure 6: Receive Path.....18
Figure 7: Bit-rate and bit-time settings.....30

# 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* |
|---|---|
| 2.7.0 | • Added configuration option to select endianes of CAN data field<br>• 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)<br>• Removed readback on transmit buffers (pages 11, 14 & 15)<br>• 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<br>• General document update |
| 2.6.0 | • Added cclk<br>• 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.

- Configuration register read-back enable
  To minimize gate count, the configuration register read-back path can be disabled
- Receive and transmit FIFO size can be adapted to system requirements
- 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.
- Different Data Bus Interfaces
  - Supports 32-bit APB, 16-bit, and 8-bit system bus interfaces

## 1.3 Block Diagram

The main building blocks are shown in the following figure:

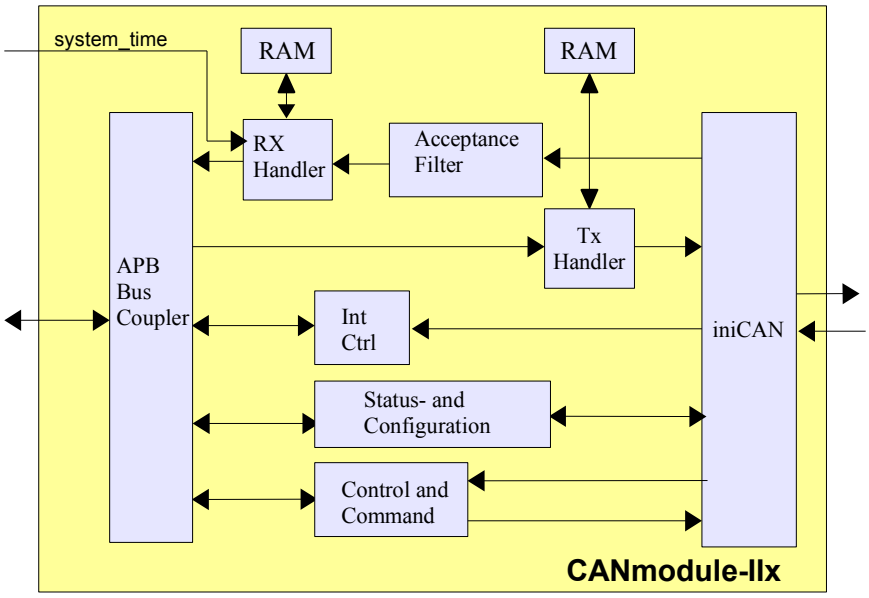



*Figure 1: Block Diagram*

### 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 used memories must support synchronous write and asynchronous read operation.

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.

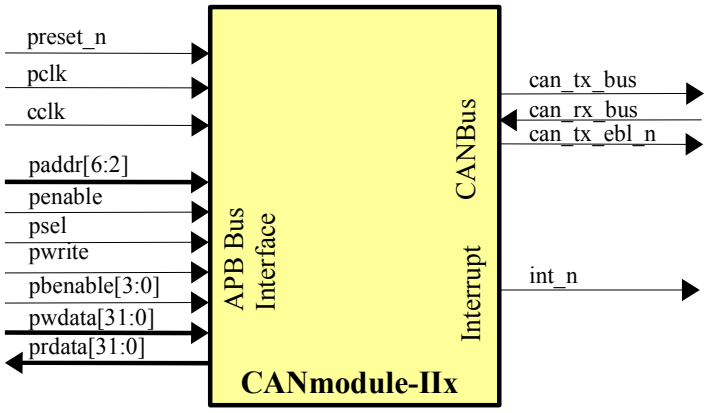



*Figure 2: Input and Outputs*

## 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 system reset, active low |

Two different clock domains are available to help FPGA systems where the main clock is much faster than what is support on the CAN side.

## 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<br>'0': read operation<br>'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 an external Phillips CAN driver:

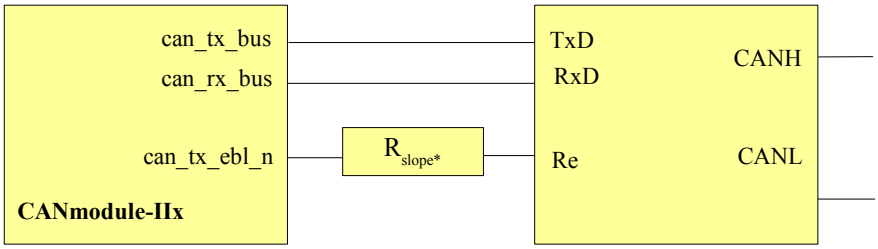



*Figure 3: 3 Pin CANbus Interface*

*) See specification of third party CAN transceiver for definition of $R_{slope}$.

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

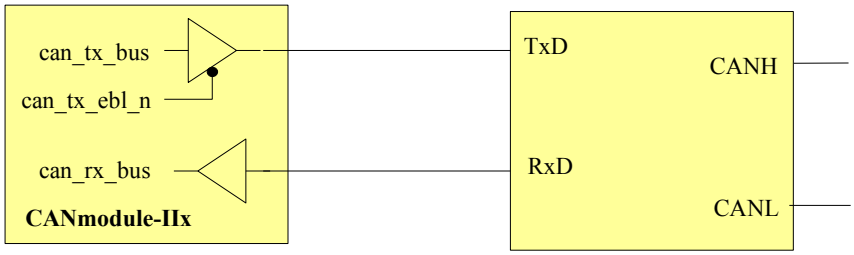



*Figure 4: 2 Pin CANbus Interface*

# 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.

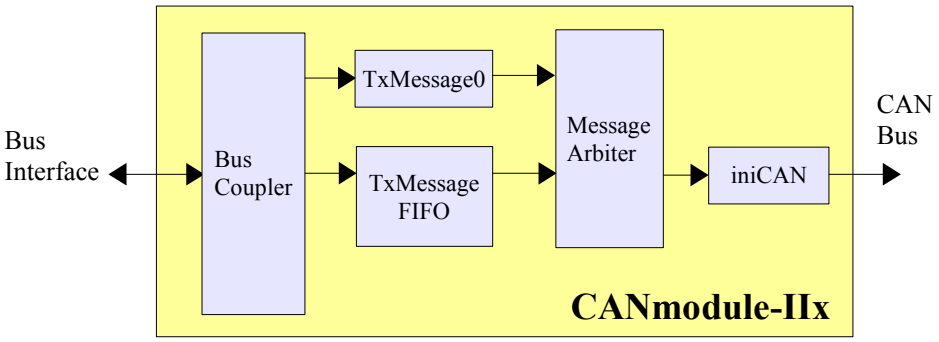



*Figure 5: Transmit Path*

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

| Address | R/W | Name | Comment |
|---|---|---|---|
| | | | [7:0]: CAN data byte 8<br>swap_endian = 1:<br>[31:24]: CAN data byte 8<br>[23:16]: CAN data byte 7<br>[15:8]: CAN data byte 6<br>[7:0]: CAN data byte 5 |
| 0x0C | W | | TxMessage0 Buffer: Control Flags<br>[23]: WPN, Write Protect Not<br>'0': Bit [21:16] remain unchanged<br>'1': Bit [21:16] are modified. This bit is zero for readback<br>[21]: RTR, Remote Bit<br>'1': This is an RTR message<br>'0': This is a standard message<br>[20]: IDE, Extended Identifier Bit<br>'1': This is an extended format message<br>'0': This is a standard format message |
| 0x0C | W | | TxMessage0 Buffer: Control Flags<br>[19:16]: DLC, Data Length Code.<br>Invalid values are transmitted as they are set, but the number of data bytes is limited to eight.<br>0x0: Data length is 0 byte<br>0x1: Data length is 1 byte, data[63:56] is used<br>...<br>0x8: Data length is 8 bytes, data[63:0] is used<br>0x9-0xF: Data length is 8 bytes<br>TxMessage0 Buffer: Command Flags<br>[7]: WPN: Write protect not.<br>'0': Bit [0] remains unchanged<br>'1': Bit [0] is modified. This bit is always zero for readback<br>[0]: TxReq, Transmit Request<br>'1': Transmit request<br>'0': Idle |
| | R | | TxMessage0 Buffer: Control Flags<br>[0]: TxReq, Transmit Request |

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

| Address | R/W | Name | Comment |
|---|---|---|---|
| | | | '0': Bit [21:16] remain unchanged<br>'1': Bit [21:16] are modified. This bit is zero for readback<br><br>[21]: RTR, Remote Bit<br>'1': This is an RTR message<br>'0': This is a standard message<br><br>[20]: IDE, Extended Identifier Bit<br>'1': This is an extended format message<br>'0': This is a standard format message<br><br>[19:16]: DLC, Data Length Code.<br>Invalid values are transmitted as they are set, but the number of data bytes is limited to eight.<br>0x0: Data length is 0 byte<br>0x1: Data length is 1 byte, data[63:56] is used<br>...<br>0x8: Data length is 8 bytes, data[63:0] is used<br>0x9-0xF: Data length is 8 bytes<br><br>[7]: WPN: Write protect not.<br>'0': Bit [0] remains unchanged<br>'1': Bit [0] is modified. This bit is always zero for readback<br><br>[0]: TxReq, Transmit Request<br>'1': Transmit request<br>'0': Idle |
| | R | | TxMessageFIFO Buffer: Control Flags<br><br>[0]: TxReq, Transmit Request<br>'1': Transmit request pending<br>'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 = '1'.
- 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 = '1'.
- 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.

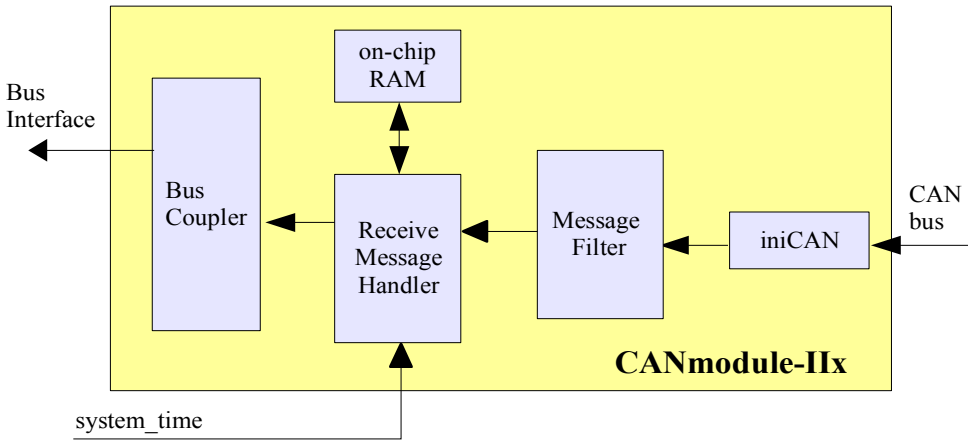



*Figure 6: Receive Path*

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

| Address | R/W | Name | Comment |
|---|---|---|---|
| | | | [7:0]: CAN data byte 5 |
| 0x3C | R | | RxMessage Buffer: Status<br>[31:16]: Captured receive time.<br>Once a message is received, the system time is captured and stored in this location.<br>[10:8]: Acceptance Filter Indicator<br>“xx1”: Filter 0 accepted message<br>“x1x”: Filter 1 accepted message<br>“1xx”: Filter 2 accepted message<br>[6]: RTR, Remote Bit<br>'1': This is an RTR message<br>'0': This is a regular message<br>[5]: IDE, Extended Identifier Bit<br>'1': This is an extended format message<br>'0': This is a standard format message<br>[4:1]: DLC, Data Length Code. Invalid values are shown as received.<br>0x0: Message length is 0. data[63:0] is not valid<br>0x1: Message length is 1. data[63:56] is valid<br>...<br>0x8: Message length is 8. data[63:0] is valid<br>[0]: MsgValid, Message Valid<br>'0': Current message is not valid<br>'1': Current message is valid |
| 0x3C | W | *RxMessage* | RxMessage Buffer: Control<br>[31:1]: N/A<br>[0]: MsgAck, Message Acknowledge<br>'0': idle<br>'1': Acknowledges receipt of new message<br>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<br>[31:3]: Identifier<br>[2]: IDE<br>[1]: RTR<br>[0]: N/A |
| 0x44 | r/W | ACR0 | Acceptance Code Register 0<br>[31:3]: Identifier<br>[2]: IDE<br>[1]: RTR<br>[0]: N/A |
| 0x48 | r/W | ACMR_DATA0 | Acceptance Code / Mask Data Register 0<br>[31:16] : Acceptance Code Data Register<br>[15:0] : Acceptance Mask Data Register |
| 0x4C | r/W | AMR1 | Acceptance Mask Register 1 |
| 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<br>[2]: AFE2 : Acceptance Filter 2 Enable bit<br>‘0’: Acceptance filter 2 is disabled<br>‘1’: Acceptance filter 2 is enabled<br>[1]: AFE1 : Acceptance Filter 1 Enable bit<br>‘0’: Acceptance filter 1 is disabled<br>‘1’: Acceptance filter 1 is enabled<br>[0]: AFE0 : Acceptance Filter 0 Enable bit<br>‘0’: Acceptance filter 0 is disabled<br>‘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<br>[19]: rxgte96<br>The receiver error counter is greater or equal 96(dec)<br>[18]: txgte96<br>The transmitter error counter is greater or equal 96(dec)<br>[17:16]: error_stat[1:0]<br>The error state of the CAN node:<br>"00": Error active (normal operation)<br>"01": Error passive<br>"1x": Bus off<br>[15:8]: rx_err_cnt[7:0]<br>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.<br>[7:0]: tx_err_cnt[7:0]<br>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<br>[3:2]: rx_level[1:0]: Sets the rx_msg interrupt threshold<br>0: Receive FIFO not empty<br>1: Receive FIFO at least ¼ full<br>2: Receive FIFO at least ½ full<br>3: Receive FIFO at least ¾ full<br>[1:0]: tx_level[1:0]: Sets the tx_msg interrupt threshold<br>0: Transmit FIFO at least ¼ empty<br>1: Transmit FIFO at least ½ empty<br>2: Transmit FIFO at least ¾<br>3: Transmit FIFO empty |
| 0x70 | r/W | IntEbl | Interrupt Enable Register<br>An interrupt source is enabled by setting its respective flag to ‘1’.<br>[15]: rx_msg<br>[14]: tx_msg<br>[12]: tx_xmit_fifo<br>[11]: tx_xmit0<br>[10]: bus_off<br>[9]: crc_err<br>[8]: form_err<br>[7]: ack_err<br>[6]: stuff_err<br>[5]: bit_err<br>[4]: rx_ovr<br>[3]: ovr_load<br>[2]: arb_loss<br>[0]: int_ebl, global interrupt enable flag<br>‘0’: All interrupts are disabled<br>‘1’: Enabled interrupt sources are available<br>[*others*]: fixed to '0' |

| Address | R/W | Name | Comment |
|---|---|---|---|
| 0x74 | R/W | IntStatus | Interrupt Status Register<br>A pending interrupt is indicated that its respective flag is set to '1'. To acknowledge an interrupt, set its flag to '1'<br>[15]: rx_msg<br>[14]: tx_msg<br>[12]: tx_xmit_fifo<br>[11]: tx_xmit0<br>[10]: bus_off<br>[9]: crc_err<br>[8]: form_err<br>[7]: ack_err<br>[6]: stuff_errr<br>[5]: bit_err<br>[4]: rx_ovr<br>[3]: ovr_load<br>[2]: arb_loss<br>[*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. This Flag is set if either the incoming message overwrites an existing one or if it is discarded. (see also overwrite_new_message).
- 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 automatic bit rate detection.

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

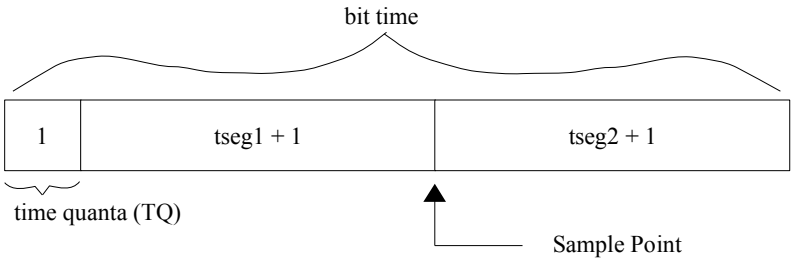
| *Address* | *R/W* | *Name* | *Comment* |
|---|---|---|---|
| 0x7C | r/W | can_cfg | CAN Controller Configuration<br>[24]: swap_endian<br>The byte position of the CAN receive and transmit data fields can be modified to match the endian setting of the processor or the CAN protocol.<br>0: CAN data byte position is not swapped (big endian)<br>1: CAN data byte position is swapped (little endian)<br>[23:16]: cfg_bitrate[7:0]:<br>Prescaler for generating the time quantum<br>“0x00”: 1 TQ = 1 clock cycle<br>“0x01”: 1 TQ = 2 clock cycles<br>...<br>“0xFF”: 1 TQ = 256 clock cycles<br>[11:8]: cfg_tseg1:<br>Length - 1 of the first time segment (bit timing) It includes the propagation time segment.<br>Cfg_tseg1=0 and cfg_tseg1=1 are not allowed<br>[7:5]: cfg_tseg2:<br>Length -1 of the second time segment.<br>Cfg_tseg2=0 is not allowed; cfg_tseg2=1 is only allowed in direct sampling mode<br>[4]: auto_restart:<br>‘0’: After bus off, the CAN controller must be restarted ‘by hand’. This is the recommended setting.<br>‘1’: After bus off, the CAN controller restarts automatically after 128 groups of 11 recessive bits.<br>[3:2]: cfg_sjw: Synchronization jump width – 1<br>sjw ≤ tseg1 and sjw ≤ tseg2 |

| Address | R/W | Name | Comment |
|---|---|---|---|
| 0x7C | r/W | can_cfg<br>*continued* | [1]: sampling_mode:<br>'0': One sampling point is used in the receive path<br>'1': 3 sampling points with majority decision are used<br>[0]: edge_mode:<br>'0': Edge from 'R' to 'D' is used for synchronization<br>'1': Both edges are used |

# 4 Configuring The CAN Controller

## 4.1 Setting proper bit rate, tseg1 and tseg2

Following relations exist for bit time, time quanta, time segments 1/2 and the data sampling point.




$$bittime = ( 1 + ( tseg1 + 1 ) + ( tseg2 + 1 ) ) * timequanta$$

$$timequanta = \frac{bitrate + 1}{f_{clk}}$$

*Figure 7: Bit-rate and bit-time settings*

e.g.,

for 1 Mbps with fclk = 8 Mhz, set bitrate = 0, tseg1 = 3 and tseg2 = 2

for 1 Mbps with fclk = 40 Mhz, set bitrate = 4, tseg1 = 3 and tseg2 = 2

Please observe following conditions for setting tseg1 and tseg2:

- tseg1=0 or tseg1=1 are not allowed
- tseg2=0 is not allowed; tseg2=1 is only allowed in direct sampling mode.


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**