INICORE

Datasheet

CANmodule-III

Version 3.0.3

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:

- CAN Data Order
  - CAN Data[63:56] is the 1st data byte of a CAN message
  - CAN Data[55:48] is the 2nd data byte of a CAN message
  - CAN Data[47:40] is the 3rd data byte of a CAN message
  - CAN Data[39:32] is the 4th data byte of a CAN message
  - CAN Data[31:24] is the 5th data byte of a CAN message
  - CAN Data[23:16] is the 6th data byte of a CAN message
  - CAN Data[15:8] is the 7th data byte of a CAN message
  - CAN Data[7:0] is the 8th data byte of a CAN message
- All default values are ‘0’ unless otherwise noted
- Undefined bits in read back are read as ‘0’
- Following nomenclature is used register mapping
  R : Read operation
  W : Write operation
- Signals ending with ‘_n’ are active low

Revision History

Version	*Comment*
3.0.3	• Changed behavior: canbus_tx_ebl_n is asserted when core is in listen only mode
3.0.2	• Corrected Rx/Tx mode bit location in error capture register • Updated test mode operation description • Updated overload interrupt description • Added description of CAN field and bit number to error capture register • Stuck-at-dominant interrupt description updated
3.0.1	• Updated bit rate calculation definition • Merged control and command register description of the Receive Message Handler • Merged control and command register description of the Transmit Message Handler • Fixed typos
3.0.0	• Added configuration option to select endianness of CAN data field • Added single-shot transmission • Added error capture register • Added revision control register • Added stuck-at-dominant interrupt • Updated APB interface to support AMBA 3 APB Protocol • Update description of interrupt flags
2.6.0	• Added SRAM test mode • Refined CAN data field and length description • Added automatic bitrate detection flowchart • Added clarification operation of canbus_tx_ebl_n
2.5.2	• Added option to generate interrupt upon transmission of an RTR auto-reply message
2.5.0	• Added internal and external loopback test mode • Corrected readback value of WPN flags

Version	*Comment*
2.2.7	• Added CAN message filter example (page 25)
2.2.6	• rx_err_cnt: changed counter description, more expressive
2.2.5	• Error Status Registers are read only (page 18)
2.2.4	• tseg1 and tseg2, corrected bits assignments
2.2.3	• Corrected and refined Rx and Tx command and control buffer description
2.2.2	• ABP Bus address, bits assignment corrected (page 9)
2.2.1	• Refined description of operation • Updated signal names • Document restructured

1 Overview

CANmodule-III is a full functional CAN controller module that supports the concept of mailboxes. It is compliant to the international CAN standard defined in ISO 11898-1.

It contains 16 receive buffers, each one with its own message filter, and 8 transmit buffers with prioritized arbitration scheme. For optimal support of Higher Layer Protocols (HLP) such as DeviceNet or SDC, the message filter covers the first two data bytes as well.

The design is written in technology independent HDL and can be mapped to ASIC and FPGA architectures and makes use of on-chip SRAM structures. An AMBA 3 Advanced Peripheral Bus (APB) interface enables smooth integration into ARM based SOC's. This full synchronous bus interface can easily be connect to other system buses.

1.1 Features

The CANmodule-III is designed for system-on-chip integrations.

Standard Compliant

- Full CAN 2.0B compliant
- Conforms to ISO 11898-1
- Maximum baudrate of 1 Mbps with 8 MHz system clock

Receive Path

- 16 receive buffers
- Each buffer has its own message filter
- Message filter covers: ID, IDE, RTR, Data byte 1 and Data byte 2
- Message buffers can be linked together to build a bigger message array
- Automatic remote transmission request (RTR) response handler with optional generation of RTR interrupt

Transmit Path

- 8 Tx message holding registers with programmable priority arbitration
- Message abort command
- Single-shot transmission (no automatic retransmission upon error or arbitration loss)

System Bus Interface

- AMBA 3 Advanced Peripheral Bus (APB) Interface
- Full synchronous zero wait-states interface
- Status and configuration interface

Programmable Interrupt Controller

- Local interrupt controller covering message and CAN error sources

Test and Debugging Support

- Listen only mode
- Internal loopback mode
- External loopback mode
- SRAM test mode
- Error capture register
  Provides option to either
  - show current bit position within CAN message
  - show bit position and type of last captured CAN error

SRAM Based Message Buffers

- Optimized for low gate-count implementation
- Single port, synchronous memory based
- 100% Synchronous Design

1.2 Block Diagram

The main building blocks are shown in the following figure:

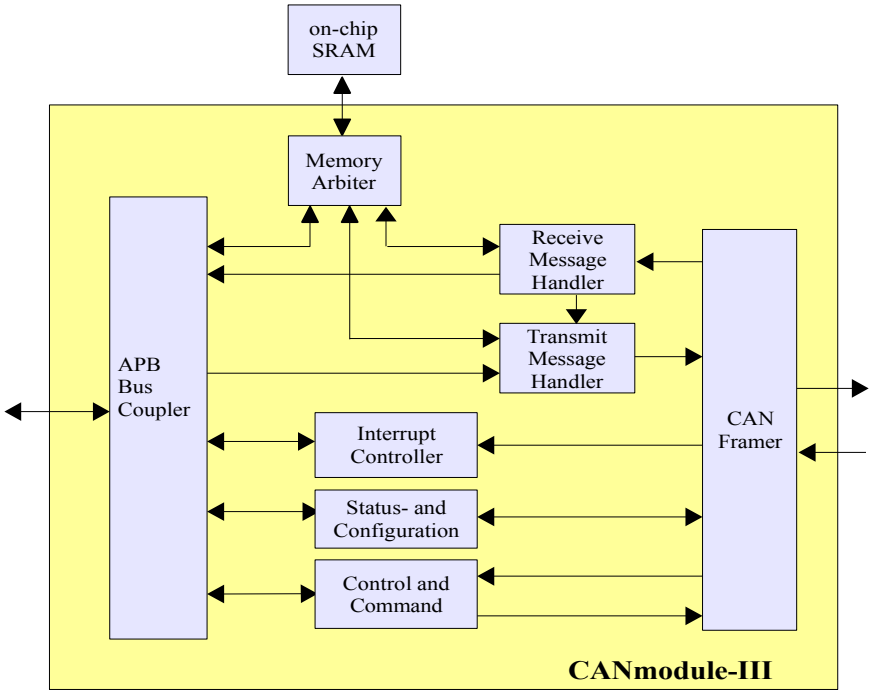


Figure 1: Block Diagram

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 figure shows the main inputs and outputs.

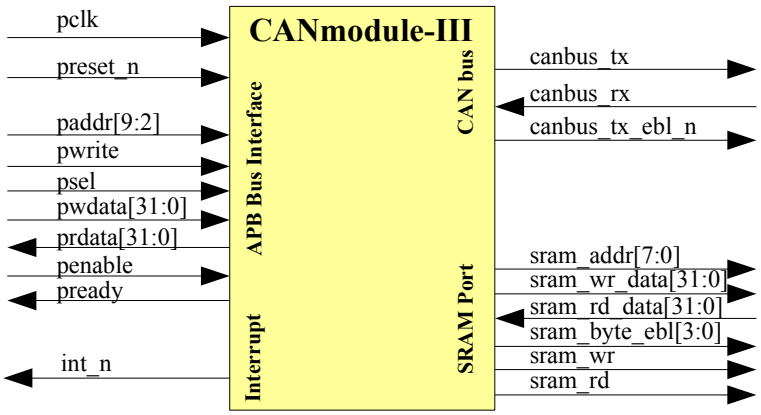


Figure 2: Inputs and Outputs

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
preset_n	in	Asynchronous system reset, active low

2.3 APB Bus Interface

The on-chip bus interface is compliant to the AMBA 3 APB bus specification[1]. The interface is full synchronous to the system clock.

Pin Name	*Type*	*Description*
psel	in	Module select signal
penable	in	Bus transfer enable signal
paddr[9:2]	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
pready	out	Ready indicator
int_n	out	Interrupt request, active low

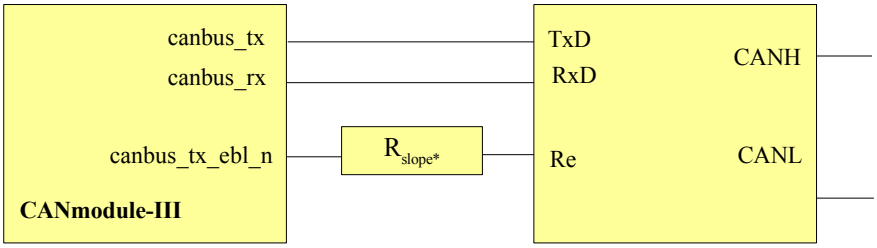
2.4 CAN Bus Interface

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

Pin Name	*Type*	*Description*
canbus_rx	in	Local receive signal (connect to can_rx_bus of external driver)
canbus_tx	out	CANbus transmit signal, connected to external driver
canbus_tx_ebl_n	out	External driver control signal This is used to enable an external CAN transceiver.

1 For AMBA 2 APB implementations, the pready signal can be ignored as the core does not generate any user wait-states.

Standard CANbus transceiver chips can directly be connected to the CAN interface pins. The following figure shows how to connect an external Phillips CAN driver.



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

Figure 3: 3 Pin CANbus Interface

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

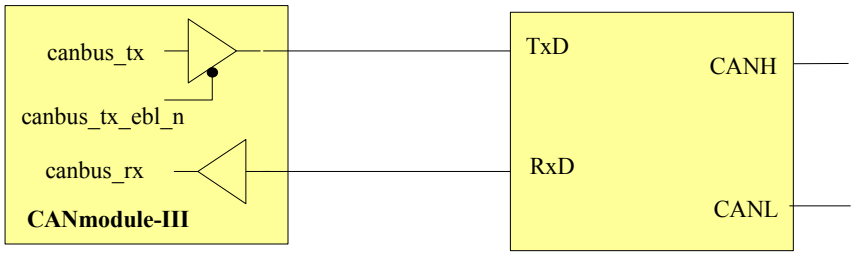


Figure 4: 2 Pin CANbus Interface

2.5 SRAM Port

A synchronous on-chip SRAM is used to store Rx and Tx messages. The SRAM is external to the core so technology adaption and test strategies don't require any modification of the core logic. The following diagram shows how to connect the core to the SRAM module:

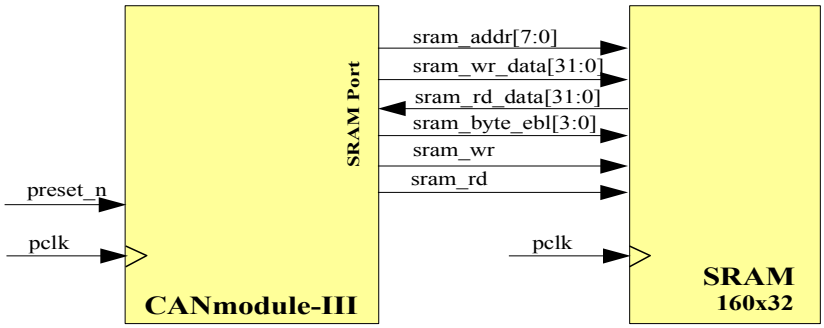


Figure 5: SRAM Connection

SRAM Interface Pins

Pin Name	*Type*	*Description*
sram_addr[7:0]	out	SRAM address bus
sram_wr_data[31:0]	in	Write data bus
sram_rd_data[31:0]	out	Read data bus
sram_byte_ebl[3:0]	out	Byte valid indicator, active high [0]: Byte valid for sram_wr_data[7:0] [1]: Byte valid for sram_wr_data[15:8] [2]: Byte valid for sram_wr_data[23:16] [3]: Byte valid for sram_wr_data[31:24]
sram_wr	out	Write enable, active high
sram_rd	out	Read enable, active high

3 Programmer's Model

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

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	*Type*	*Description*
0x000	R/W	Interrupt Control
0x008	R/W	Buffer Status Indicators
0x00C	R/W	Error Status Indicators
0x010	R/W	CAN Operating Mode
0x014	R/W	CAN Configuration
0x018	R/W	Error Capture Register
0x020	R/W	TxMessage0 Buffer
0x030	R/W	TxMessage1 Buffer
0x040	R/W	TxMessage2 Buffer
0x050	R/W	TxMessage3 Buffer
0x060	R/W	TxMessage4 Buffer
0x070	R/W	TxMessage5 Buffer
0x080	R/W	TxMessage6 Buffer
0x090	R/W	TxMessage7 Buffer
0x0A0	R/W	RxMessage0 Buffer

Address	*Type*	*Description*
0x0C0	R/W	RxMessage1 Buffer
0x0E0	R/W	RxMessage2 Buffer
0x100	R/W	RxMessage3 Buffer
0x120	R/W	RxMessage4 Buffer
0x140	R/W	RxMessage5 Buffer
0x160	R/W	RxMessage6 Buffer
0x180	R/W	RxMessage7 Buffer
0x1A0	R/W	RxMessage8 Buffer
0x1C0	R/W	RxMessage9 Buffer
0x1E0	R/W	RxMessage10 Buffer
0x200	R/W	RxMessage11 Buffer
0x220	R/W	RxMessage12 Buffer
0x240	R/W	RxMessage13 Buffer
0x260	R/W	RxMessage14 Buffer
0x280	R/W	RxMessage15 Buffer

3.2 Internal Register Description

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

3.2.1 Interrupt Controller

The interrupt controller contains an interrupt status and an interrupt enable register. The interrupt status register stores internal interrupt events. Once a bit is set it remains set until it is cleared by writing a 1 to it. The interrupt enable register has no effect on the interrupt status register.

The interrupt enable register controls which particular bits from the interrupt status register are used to assert the interrupt output int_n. int_n is asserted if a particular interrupt status bit and the respective enable bit are set.

Address	*Name*	*R/W*	*Comment*
0x000	IntStatus	R/W	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]: sst_failure: Single shot transmission failure 1: A buffer set for single shot transmission experienced an arbitration loss or a bus error during transmission. The sst_failure interrupt is set as well when an SST message is in the transmit buffer while the CAN controller is being stopped. 0: Normal operation [14]: stuck_at_0: Stuck at dominant error 1: Indicates if the rx input remained stuck at 0 (dominant level) for 16 consecutive bit times. This detection is active when the CAN controller is running. 0: Normal operation [13]: rtr_msg: RTR auto-reply message sent 1: Indicates that a RTR auto-reply message was sent. 0: Normal operation [12]: rx_msg: Receive message available 1: A new message was successfully received and stored in a receive buffer which had its RxIntEbl flag asserted. 0: Normal operation

Address	Name	R/W	Comment
0x000	IntStatus *continued*	R/W	Interrupt Status Register *(continued)* [11]: tx_msg: Message transmitted 1: A message was successfully sent from a transmit buffer which had its TxIntEbl flag asserted. 0: Normal operation [10]: rx_msg_loss: Received message lost 1: A newly received message couldn't be stored because the target message buffer was full (eg, its MsgAv flag was set). 0: Normal operation [9]: bus_off: Bus-off 1: The CAN controller entered the bus-off error state. 0: Normal operation [8]: crc_err: CRC error 1: A CAN CRC error was detected [7]: form_err: Format error 1: A CAN format error was detected 0: Normal operation [6]: ack_err: Acknowledge error 1: A CAN message acknowledgment error was detected 0: Normal operation [5]: stuff_err: Bit stuffing error 1: A CAN bit stuffing error was detected 0: Normal operation [4]: bit_err: Bit error 1: A CAN bit error was detected 0: Normal operation [3]: ovr_load: Overload condition detected 1: A CAN overload condition was detected 0: Normal operation

Address	Name	R/W	Comment
0x000	IntStatus *continued*	R/W	[2]: arb_loss: Arbitration loss 1: The message arbitration was lost while sending a message. The message transmission will be retried once the CAN bus is idle again. 0: Normal operation [1:0]: N/A
0x004	IntEbl	R/W	Interrupt Enable Register A particular interrupt source is enabled by setting its respective flag to '1'. [15]: sst_failure interrupt enable [14]: stuck_at_0 interrupt enable [13]: rtr_msg interrupt enable [12]: rx_msg interrupt enable [11]: tx_msg interrupt enable [10]: rx_msg_loss interrupt enable [9]: bus_off interrupt enable [8]: crc_err interrupt enable [7]: form_err interrupt enable [6]: ack_err interrupt enable [5]: stuff_err interrupt enable [4]: bit_err interrupt enable [3]: ovr_load interrupt enable [2]: arb_loss interrupt enable [1]: N/A [0]: int_ebl, global interrupt enable flag '1': Enabled interrupt sources are available '0': All interrupts are disabled

Interrupt Generation

Following figure shows how the system interrupt is generated:

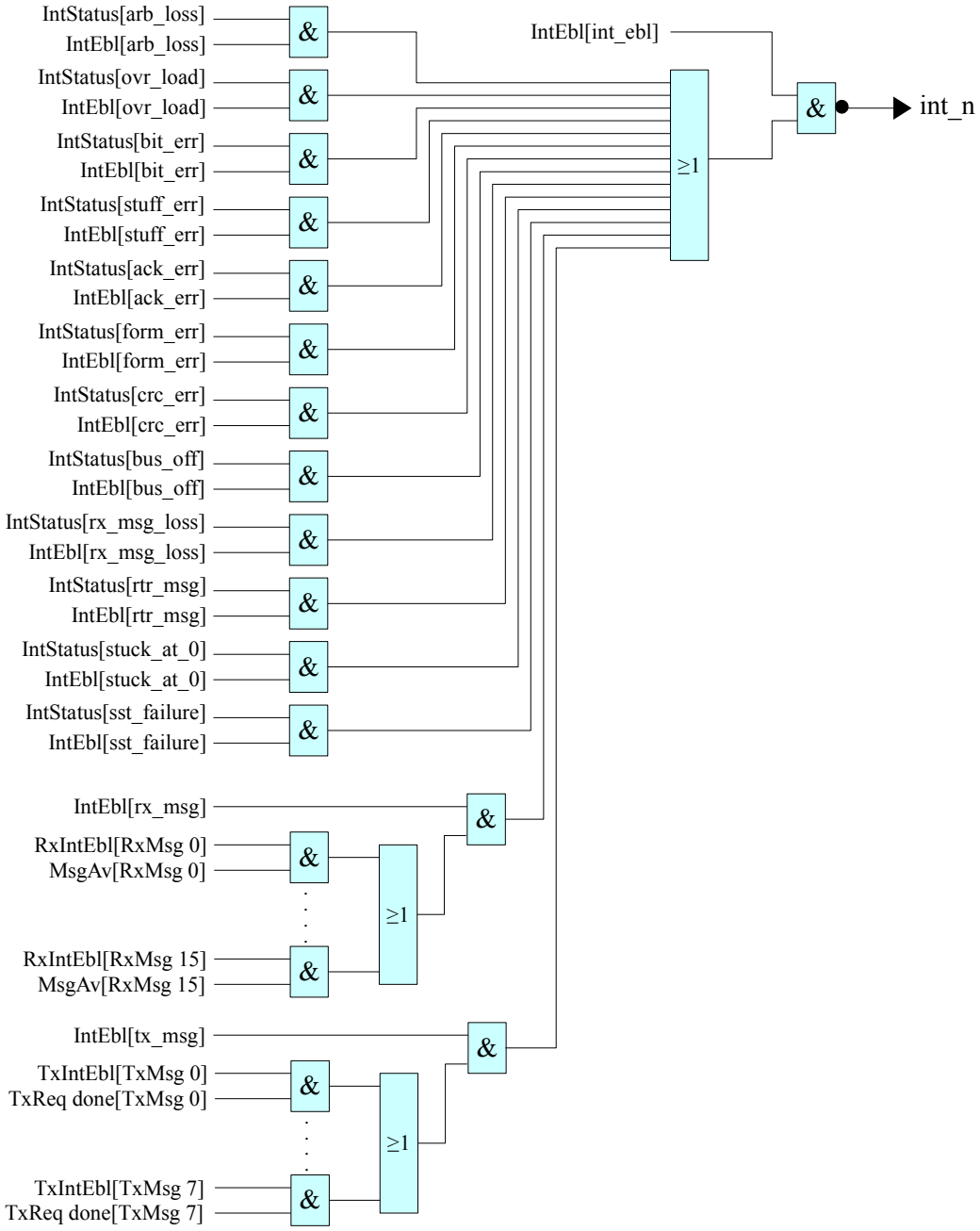


Figure 6: Interrupt generation

3.2.2 Buffer Status Indicators

These status indicators bundle the respective flags from all RxMessage and TxMessage buffers.

Address	*Name*	*R/W*	*Comment*
0x008	BufferStatus	R	RxMessage and TxMessage Buffer Status [23]: TxMessage7 - TxReq pending .. [16]: TxMessage0 - TxReq pending [15]: RxMessage15 – MsgAv .. [0]: RxMessage0 - MsgAv

Note: All flags are read only! E.g., to acknowledge a MsgAv flag, the CPU has to directly write to the respective RxMessage buffer.

3.2.3 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	*Name*	*R/W*	*Comment*
0x00C	ErrorStatus	R	CAN Error Status [19]: rxgte96 The Rx error counter is greater or equal 96_{dec} [18]: txgte96 The Tx error counter is greater or equal 96_{dec} [17:16]: error_state[1:0] The error state of the CAN node: “00”: error active (normal operation) “01”: error passive “1x”: bus off

Address	Name	R/W	Comment
0x00C	ErrorStatus *continued*	R	CAN Error Status (*continued*) [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 transmitter error counter according to the CAN standard. When it is greater than 255_{dec}, it is fixed at 255_{dec}.

3.2.4 Operating Modes

The CANmodule-III 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.

Before starting the CAN controller, all the CAN configuration registers have to be set according to the target application.

Address	Name	R/W	Comment
0x010	Command	R	Revision Control Register The following bits show the version of the CAN core in the format [major version].[minor version].[revision number] [31:28]: Major version [27:24]: Minor version [23:16]: Revision number
		R/W	CAN Command Register [3]: SRAM test mode 0: Normal operation 1: Enable SRAM test mode [2:1]: Test Mode 0: Normal operation 1: Listen-only mode

Address	Name	R/W	Comment
0x010	Command *continued*	R/W	CAN Command Register *(continued)* 2: External loopback mode 3: Internal loopback mode [0]: Run/Stop mode 0: Sets the CAN controller into stop mode. Returns 0 when stopped. 1: Sets the CAN controller into run mode. Returns 1 when running.

Test modes overview

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

Test Mode	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 bit-rate detection. The output is kept at recessive ('R') level. The transmitter remains active.
2	External loop back The CAN controller participates in the regular CAN transmission and reception. Further, 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 its own data. No data is sent to the network and no data form the CANbus is received. The output is kept at recessive ('R') level.

SRAM Test Mode

To support software based memory testing, the CANmodule-III core can be put into a SRAM test mode. When this SRAM test mode is active, the CAN controller operation is disabled and transparent access from the host interface to all SRAM memory locations is available.

When in SRAM test mode,

- Transparent read and write access to all SRAM memory locations is supported
- All message buffer write protect features are disabled
- Access to receive and transmit message buffer control registers is disabled

The SRAM test mode and the CAN controller operation are mutually exclusive:

- SRAM test mode can only be enabled when the CAN controller is stopped
- The CAN controller can only be started when the SRAM test mode is not active

Following table provides the address mapping between the APB host interface and the SRAM:

APB Address	*SRAM Address*	*Description*
0x020	0x000	TxObject0: Control Bits
0x024	0x001	TxObject0: Identifier Bits
0x028	0x002	TxObject0: Data High Bits
0x02C	0x003	TxObject0: Data Low Bits
0x030-0x03C	0x004-0x007	TxObject1
0x040-0x04C	0x008-0x00B	TxObject2
0x050-0x05C	0x00C-0x00F	TxObject3
0x060-0x06C	0x010-0x013	TxObject4
0x070-0x07C	0x014-0x017	TxObject5
0x080-0x08C	0x018-0x01B	TxObject6
0x090-0x09C	0x01C-0x01F	TxObject7
0x0A0	0x020	RxObject0: Control Bits
0x0A4	0x021	RxObject0: Identifier Bits
0x0A8	0x022	RxObject0: Data High Bits
0x0AC	0x023	RxObject0: Data Low Bits
0x0B0	0x024	RxObject0: AMR – ID
0x0B4	0x025	RxObject0: ACR – ID
0x0B8	0x026	RxObject0: AMR – Data
0x0BC	0x027	RxObject0: ACR – Data
0x0C0-0x0DC	0x028-0x02F	Receive Message Object 1
0x0E0-0x0FC	0x030-0x037	Receive Message Object 2
0x100-0x11C	0x038-0x03F	Receive Message Object 3
0x120-0x13C	0x040-0x047	Receive Message Object 4

APB Address	*SRAM Address*	*Description*
0x140-0x15C	0x048-0x04F	Receive Message Object 5
0x160-0x17C	0x050-0x057	Receive Message Object 6
0x180-0x19C	0x058-0x05F	Receive Message Object 7
0x1A0-0x1BC	0x060-0x067	Receive Message Object 8
0x1C0-0x1DC	0x068-0x06F	Receive Message Object 9
0x1E0-0x1FC	0x070-0x077	Receive Message Object 10
0x200-0x21C	0x078-0x07F	Receive Message Object 11
0x220-0x23C	0x080-0x087	Receive Message Object 12
0x240-0x25C	0x088-0x08F	Receive Message Object 13
0x260-0x27C	0x090-0x097	Receive Message Object 14
0x280-0x29C	0x098-0x09F	Receive Message Object 15

3.2.5 CAN Configuration Register

The CANmodule-III has to be configured prior to its use. Following registers define the effective CAN data rate[2], CAN data synchronization, and message buffer arbitration. These registers have to be set before the CAN controller is started.

Address	*Name*	*R/W*	*Comment*
0x014	Config	R/W	CAN Configuration [30:16]: cfg_bitrate[14:0]: Prescaler for generating the time quantum which defines the TQ: 0: One time quantum equals 1 clock cycle 1: One time quantum equals 2 clock cycles ... 32767: One time quantum equals 32768 clock cycles [14]: ecr_mode: Error Capture Mode 0: Free running: The ecr register shows the current bit position within the CAN frame. 1: Capture mode: The ecr register shows the bit position and type of the last captured CAN error. [13]: 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 used CAN protocol. 0: CAN data byte position is not swapped (big endian) 1: CAN data byte position is swapped (little endian) [12]: cfg_arbiter: Transmit Buffer Arbiter 0: Round robin arbitration 1: Fixed priority arbitration

2 Additional information on the CAN data rate settings using time segment 1 (tseg1), time segment 2 (tseg2), and bit rate are given in chapter 4.

Address	Name	R/W	Comment
0x014	Config *continued*	R/W	CAN Configuration (*continued*) [11:8]: cfg_tseg1: Time Segment 1 Length of the first time segment: tseg1 = cfg_tseg1 + 1 Time segment 1 includes the propagation time. cfg_tseg1=0 and cfg_tseg1=1 are not allowed. [7:5]: cfg_tseg2: Time Segment 2 Length of the second time segment: tseg2 = cfg_tseg2 + 1 cfg_tseg2=0 is not allowed; cfg_tseg2=1 is only allowed in direct sampling mode. [4]: auto_restart 0: After bus-off, the CAN must be restarted 'by hand'. This is the recommended setting. 1: After bus-off, the CAN is restarting automatically after 128 groups of 11 recessive bits [3:2]: cfg_sjw: Synchronization Jump Width Length of the synchronization jump width: sjw = cfg_sjw + 1 Note: sjw ≤ tseg1 and sjw ≤ tseg2 [1]: sampling_mode: CAN Bus Bit Sampling 0: One sampling point is used in the receiver path 1: 3 sampling points with majority decision are used [0]:edge_mode: CAN Bus Synchronization Logic 0: Edge from 'R' to 'D' is used for synchronization 1: Both edges are used

CAN Bit-Timing Configuration

Using cfg_tseg1 and cfg_tseg2, the effective sampling point within a bit-time and the length of the bit-time field 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 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)

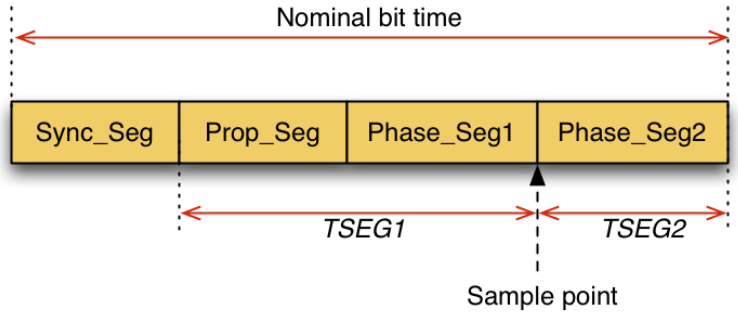


Figure 7: Bit-timing configuration

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

3.2.6 Tx Message Registers

Eight transmit message holding buffers are provided. An internal priority arbiter selects the message according to the chosen arbitration scheme. Upon transmission of a message or message arbitration loss, the priority arbiter re-evaluates the message priority of the next message.

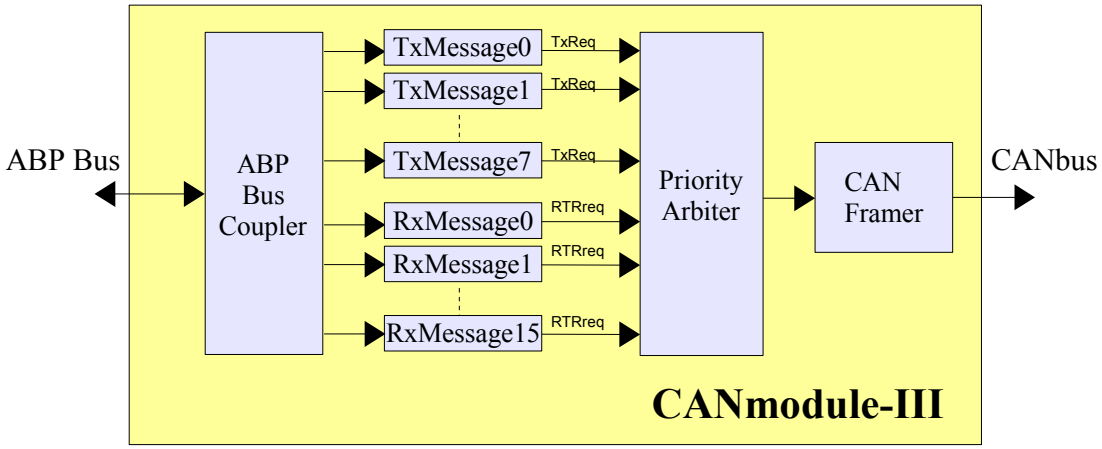


Figure 8: Message Arbitration

Message Arbitration

The priority arbiter supports round robin and fixed priority arbitration. The arbitration mode is selected using the configuration register.

- Round Robin: Buffers are served in a defined order: 0-1-2..7-0-1... A particular buffer is only selected if its TxReq flag is set. This scheme guarantees that all buffers receive the same probability to send a message.
- Fixed Priority: Buffer 0 has the highest priority. This way it is possible to designate buffer 0 as the buffer for error messages and it is guaranteed that they are sent first.

Note: RTR message requests are served before TxMessage buffers are handled. E.g., RTRreq0, ... RTRreq15, TxMessage0, TxMessage1, ... TxMessage7

Register Mapping Transmit Buffers

The register mapping of the transmit buffers is shown in the table below.

Address	*Name*	*R/W*	*Comment*
0x020	TxMessage0. Control	R/W	TxMessage0 Buffer: Control Flags [23]: WPNH, Write Protect Not High[3] 0: Bit [21:16] remain unchanged 1: Bit [21:16] are modified, default. The readback value of this bit is undefined. [21]: RTR, Remote Bit 0: This is a standard message 1: This is an RTR message [20]: IDE, Extended Identifier Bit 0: This is a standard format message 1: This is an extended format message [19:16]: DLC, Data Length Code Invalid values are transmitted as they are, but the number of data bytes is limited to eight. 0: Message has 0 data bytes 1: Message has 1 data byte ... 8: Message has 8 data bytes 9-15: Message has 8 data bytes [3]: WPNL: Write Protect Not Low. 0: Bit [2] remains unchanged 1: Bit [2] is modified, default. This bit is always zero for readback [2]: TxIntEbl, Tx Interrupt Enable 0: Interrupt disabled 1: Interrupt enabled, successful message transmission sets the TxMsg flag in the interrupt controller. [1]: TxAbort, Transmit Abort Request 0: idle 1: Requests removal of a pending message. The message is removed the next time an arbitration loss happened. The flag is cleared when the message was removed or when the message won arbitration. The TxReq flag is released at the same time.

3 Using the WPN flag enables simple retransmission of the same message by only having

Address	Name	R/W	Comment
0x020	TxMessage0. Control *continued*	R/W	[0]: TxReq, Transmit Request (continued) Write: 0: idle 1: Message Transmit Request[4] Read: 0: TxReq completed 1: TxReq pending
0x024	TxMessage0. ID	R/W	TxMessage0 Buffer: Identifier [31:3]: ID[28:0] [2:0]: N/A
0x028	TxMessage0. DataHigh	R/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 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

to set the TRX flag without taking care of the special flags

4 The Tx message buffer must not be changed while TxReq is 1!

Address	Name	R/W	Comment
0x02C	TxMessage0. DataLow	r/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 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
0x030-0x03C	TxMessage1 Buffer, see TxMessage0 Buffer for description		
0x040-0x04C	TxMessage2 Buffer, see TxMessage0 Buffer for description		
0x050-0x05C	TxMessage3 Buffer, see TxMessage0 Buffer for description		
0x060-0x06C	TxMessage4 Buffer, see TxMessage0 Buffer for description		
0x070-0x07C	TxMessage5 Buffer, see TxMessage0 Buffer for description		
0x080-0x08C	TxMessage6 Buffer, see TxMessage0 Buffer for description		
0x090-0x09C	TxMessage7 Buffer, see TxMessage0 Buffer for description		

Procedure for sending a message

- Write message into an empty transmit message holding buffer. An empty buffer is indicated by TxReq is equal to zero.
- Request transmission by setting the respective TxReq flag to one.
- The TxReq flag remains set as long as the message transmit request is pending. The content of the message buffer must not be changed while the TxReq flag is set!
- The internal message priority arbiter selects the message according to the chosen arbitration scheme
- Once the message was transmitted, the TxReq flag is set to zero and the TxMsg interrupt status bit is asserted.

Procedure for removing a message from a transmit holding register

A message can be removed from a transmit holding buffer by asserting the TxAbort flag. Use following procedure to remove the contents of a particular TxMessage buffer:

- Set TxAbort to one to request the message removal.
- This flag remains set as long as the message abort request is pending. It is cleared when either the message won arbitration (TxMsg interrupt active) or the message was removed (TxMsg interrupt inactive)

Single Shot Transmission (SST)

The single-shot transmission mode is used in systems where the retransmission of a CAN message due to an arbitration loss or a bus error must be prevented.

A single-shot transmission request is set by asserting TxReq and TxAbort at the same time. Upon a successful message transmission, both flags are cleared.

If an arbitration loss or a bus error happened during the transmission, the TxReq flag is cleared, but the TxAbort flag remains asserted. At the same time, the sst_failure interrupt is asserted.

An SST message can be aborted by setting the TxAbort=1 and TxReq=0.

3.2.7 Rx Message Buffers

The CANmodule-III provides 16 individual receive message buffers. Each one has its own message filter mask. Automatic reply to RTR messages is supported.

If a message is accepted in a receive buffer, its MsgAv flag is set. The message remains valid as long as MsgAv flag is set. The host CPU has to reset the MsgAv flag to enable receipt of a new message.

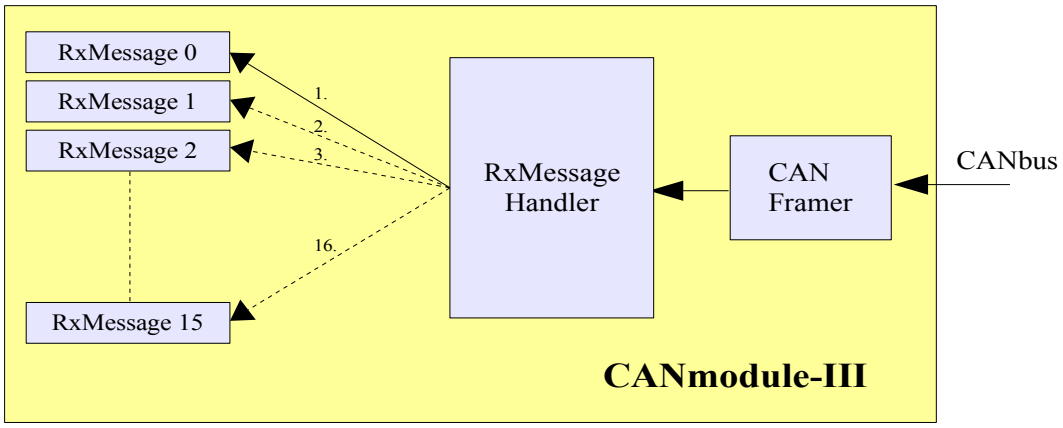


Figure 9: Receive Message Handler

Rx Message Processing

After receipt of a new message, the RxMessageHandler searches all receive buffer starting from RxMessage0 until it finds a valid buffer.

A valid buffer is indicated by:

- Receive buffer is enabled indicated by RxBufferEbl = 1
- Acceptance Filter of receive buffer matches incoming message

If the RxMessageHandler finds a valid buffer that is empty, then the message is stored and the MsgAv flag of this buffer is set to '1'. If the RxIntEbl flag is set, than the RxMsg flag of the interrupt controller is asserted. If the receive buffer already contains a message indicated by MsgAv = 1 and the Link Flag is not set, then the RxMsgLoss interrupt flag is asserted.

If an incoming message has its RTR flag set and the RTRreply flag of the matching buffer is set, then the message is not stored but an RTR auto-reply request is issued. See paragraph 'RTR Auto-Reply' for more details.

Acceptance Filter

Each receive buffer has its own acceptance filter that is used to filter incoming messages. An acceptance filter consists of Acceptance Mask Register (AMR) and Acceptance Code Register (ACR) pair. The AMR defines which bits of the incoming CAN message have to match the respective ACR bits.

Following message fields are covered:

- ID
- IDE
- RTR
- Data byte 1 and data byte 2 (DATA[63:56])[5]

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 not accepted when the incoming bit doesn't match respective ACR flag

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

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 Layer 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						

5 Some CAN High Level Protocols such as SDS or Device Net carry additional protocol related information in the first or first two data bytes that are used for message acceptance and selection. Having the capability to filter on these fields provides a more efficient implementation of the protocol stack running on the CPU.

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

RTR Auto-Reply

The CANmodule-III supports automatic answering of RTR message requests. All 16 receive buffers support this feature.

If an RTR message is accepted in a receive buffer where the RTRreply flag is set, then this buffer automatically replies to this message with the content of this receive buffer. The RTRreply_pending flag is set when the RTR message request is received. It is cleared when the message was sent or when the message buffer is disabled. To abort a pending RTRreply message, use the RTRabort command.

If the RTR auto-reply option is selected, the RTRsent flag is asserted when the RTR auto-reply message was successfully sent. It is cleared by writing a 1 to it.

An RTR message interrupt is generated if the RTRsent flag and the RxIntEbl are set. This interrupt is cleared by clearing the RTRsent flag.

RxBuffer Linking

Several receive buffers may be linked together to form a receive buffer array which acts almost like a receive FIFO.

Requirements:

- All buffers of the same array must have the same message filter setting (AMR and ACR are identical)
- The last buffer of an array may not have its link flag set

When a receive buffer already contains a message (MsgAv=1) and a new message arrives for this buffer, then this message would be discarded (RxMsgLoss Interrupt). To avoid this situation several receive buffers can be linked together. When the CANmodule-III receives a new message, the RxMessage handler searches for a valid receive buffer. If one is found that is already full (MsgAv=1) and the link flag is set (BufferLink=1), the search for a valid receive buffer continues. If no other buffer is found, than the RxMsgLoss interrupt is set and the message discarded.

It is possible to build several message arrays. Each of these arrays must use the same AMR and ACR.

Register Mapping Receive Buffers

The register mapping of the receive buffers is shown in the table below.

Address	*Name*	*R/W*	*Comment*
0x0A0	RxMessage0. Command	R/W	RxMessage0: Control Flags [23]: WPNH, Write Protect Not High 0: Bits [21:16] remain unchanged 1: Bits [21:16] are modified The readback value of this bit is undefined. [21]: RTR, Remote Bit 1: This is an RTR message 0: This is a regular 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 0: Message has 0 data bytes 1: Message has 1 data byte … 8: Message has 8 data bytes 9-15: Message has 8 data bytes [7]: WPNL, Write Protect Not Low 0: Bits [6:3] remain unchanged 1: Bits [6:3] are modified This bit is always zero for readback [6]: Link Flag 0: This buffer is not linked to the next 1: This buffer is linked with next buffer [5]: RxIntEbl, Receive Interrupt Enable 0: Interrupt generation is disabled 1: Interrupt generation is enabled [4]: RTRreply, automatic message reply upon receipt of an RTR message 0: Automatic RTR message handling disabled 1: Automatic RTR message handling enabled

Address	Name	R/W	Comment
0x0A0	RxMessage0. Control *(continued)*	R/W	RxMessage0: Control (*continued*) [3]: Buffer Enable 0: Buffer is disabled 1: Buffer is enabled [2]: RTRabort, RTR Abort Request 0: Idle 1: Requests removal of a pending RTR message reply. The flag is cleared when the message was removed or when the message won arbitration. The TxReq flag is released at the same time. [1]: RTReply_pending 0: No RTR reply request pending 1: RTR reply request pending [0]: MsgAv/RTRsent, Message Available/RTR sent If RTRreply flag is set, this bit shows if an RTR auto-reply message has been sent, otherwise it indicates if the buffer contains a valid message. Read: 0: idle 1: New message available (RTReply=0), RTR auto-reply message sent (RTRreply=1) Write: 0: idle 1: Acknowledges receipt of new message or transmission of RTR auto-reply message [1]
0x0A4	RxMessage0. ID	R/W	RxMessage: Identifier [31:3]: ID[28:0] [2:0]: zeros

1 Before acknowledging receipt of a new message, the message content must be copied into system memory. Acknowledging a message clears the MsgAv flag.

Address	Name	R/W	Comment
0x0A8	RxMessage0. DataHigh	R/W	RxMessage Data high 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
0x0AC	RxMessage0. DataLow	R/W	RxMessage Data low 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
0x0B0	RxMessage0. AMR	R/W	Acceptance Mask Register [31:3]: Identifier [2]: IDE [1]: RTR [0]: N/A

Address	Name	R/W	Comment
0x0B4	RxMessage0. ACR	R/W	Acceptance Code Register [31:3]: Identifier [2]:IDE [1]: RTR [0]:N/A
0x0B8	RxMessage0. AMR_Data	R/W	Acceptance Mask Register – Data [15:8]: CAN data byte 1 [7:0]: CAN data byte 2
0x0BC	RxMessage0. ACR_Data	R/W	Acceptance Code Register – Data [15:8]: CAN data byte 1 [7:0]: CAN data byte 2
0x0C0-0x0DC	RxMessage1 Buffer, see RxMessage0 Buffer for description		
0x0E0-0x0FC	RxMessage2 Buffer, see RxMessage0 Buffer for description		
0x100-0x11C	RxMessage3 Buffer, see RxMessage0 Buffer for description		
0x120-0x13C	RxMessage4 Buffer, see RxMessage0 Buffer for description		
0x140-0x15C	RxMessage5 Buffer, see RxMessage0 Buffer for description		
0x160-0x17C	RxMessage6 Buffer, see RxMessage0 Buffer for description		
0x180-019C	RxMessage7 Buffer, see RxMessage0 Buffer for description		
0x1A0-0x1BC	RxMessage8 Buffer, see RxMessage0 Buffer for description		
0x1C0-0x1DC	RxMessage9 Buffer, see RxMessage0 Buffer for description		
0x1E0-0x1FC	RxMessage10 Buffer, see RxMessage0 Buffer for description		
0x200-0x21C	RxMessage11 Buffer, see RxMessage0 Buffer for description		
0x220-0x23C	RxMessage12 Buffer, see RxMessage0 Buffer for description		
0x240-0x25C	RxMessage13 Buffer, see RxMessage0 Buffer for description		

Address	Name	R/W	Comment
0x260-0x27C	RxMessage14 Buffer, see RxMessage0 Buffer for description		
0x280-0x29C	RxMessage15 Buffer, see RxMessage0 Buffer for description		

3.3 Error Capture Register

The CANmodule-III core contains a dedicated error capture register that can be used to perform additional CAN bus diagnostics.

Two different modes of operation are supported:

- Free running mode
  In free-running mode, the ECR displays the field and bit position within the current CAN frame.
- Error capture mode
  In error capture mode, the ECR samples the field and bit position when a CAN error is detected. In order to sample such an event, the ECR needs to be armed by performing a write access to it. When armed, the ECR only captures one error event. For successive error captures, the ECR needs to be armed again.

Address	*Name*	*R/W*	*Comment*
0x018	ECR	R	Error capture register [16:12]: Field 0x00: Stopped 0x01: Synchronize 0x05: Interframe 0x06: Bus idle 0x07: Start of frame 0x08: Arbitration 0x09: Control 0x0A: Data 0x0B: CRC 0x0C: ACK 0x0D: End of frame 0x10: Error flag 0x11: Error echo 0x12: Error delimiter 0x18: Overload flag 0x19: Overload echo 0x1A: Overload delimiter Others: n/a [11:6]: Bit number Bit number inside of Field

Address	Name	R/W	Comment
0x018	ECR *continued*	R	Error Capture Register (*continued*) [5]: Tx mode When asserted, the CAN controller is transmitter [4]: Rx mode When asserted, the CAN controller is receiver [3:1]: Error type 0: Arbitration loss 1: Bit error 2: Bit stuffing error 3: Acknowledge error 4: Form error 5: CRC error Others: n/a [0]: Status 0: ECR register captured an error or is in free running mode. 1: ECR register is armed
		W	Arm Error Capture Register When in error capture mode, writing to the ECR register will arm the error capture register. This means that the error type and position is captured upon detection of a CAN error. Once an error is captured, the register will hold the value until it is armed again.

Figure 10 shows the bit mapping reported by the Error Capture Register. Please note that the IDE bit in the standard frame is reported as bit 12 of the Arbitration field instead of bit 0 of the Control field.

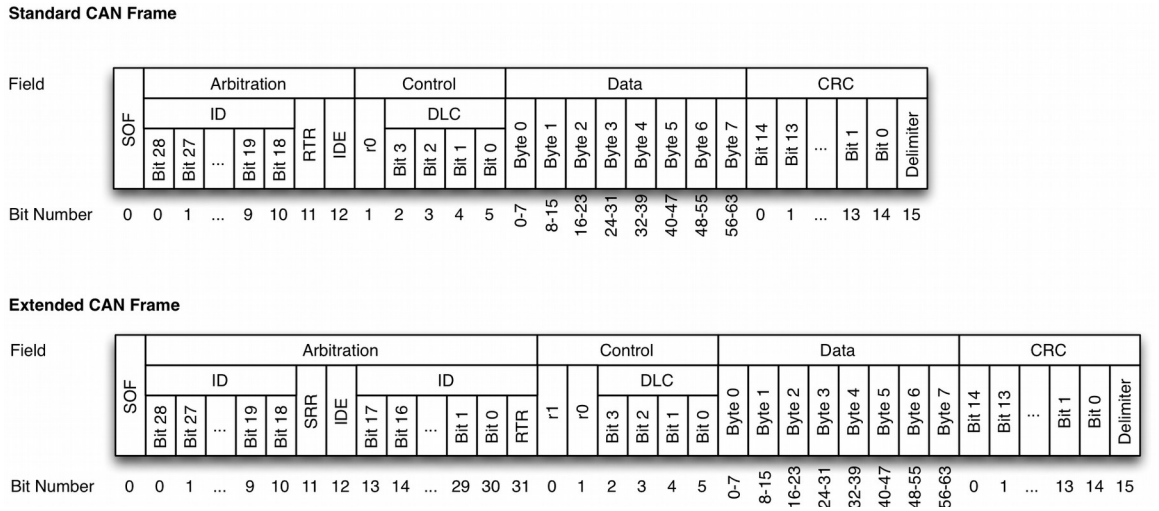


Figure 10: ECR CAN Frame Bit Mapping

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:

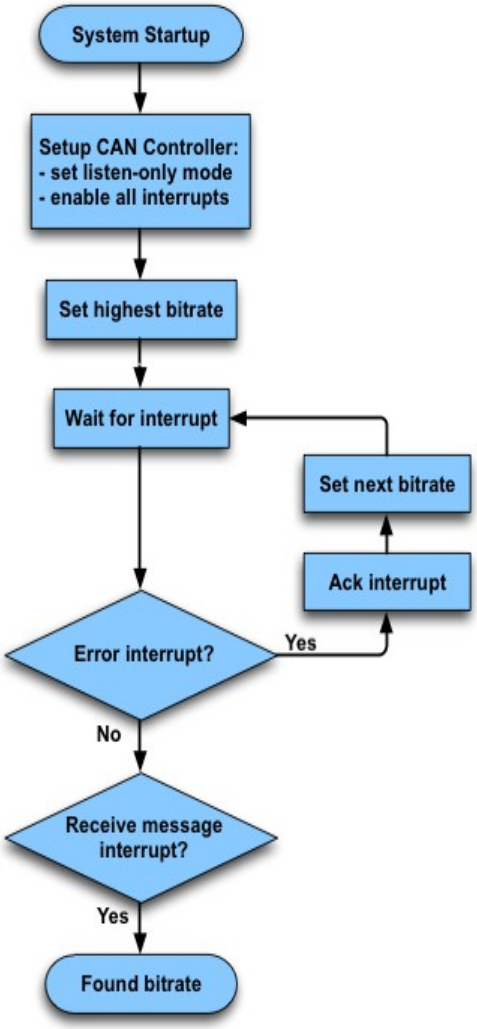


Figure 11: Automatic bitrate detection flowchart

About Inicore

- FPGA and ASIC Design
- Easy-to-use IP Cores
- System-on-Chip Solutions
- Consulting Services
- ASIC to FPGA Migration
- Obsolete ASIC Replacements

Inicore is an experienced system design house providing FPGA / ASIC and SoC design services. The company's expertise in architecture, intellectual property, methodology and tool handling provides a complete design environment that helps customers shorten their design cycle and speed time to market. Our offering covers feasibility study, concept analysis, architecture definition, code generation and implementation. When ready, we deliver you a FPGA or take your design to an ASIC provider, whatever is more suitable for your unique solution.

Customer Advantages

We offer one-stop shopping for everything from the specifications to the chip or module solution. Our experience and fast turnaround time reduces your development costs and increases your returns from the market. Your system is not limited by the level of expertise and standard chip solutions you happen to have in-house. Achieve market success by differentiating and optimizing your product. Reusability builds the basis for further developments in the ever-decreasing product life cycle.

Visit us @ www.inicore.com