iAP-CAN 16f

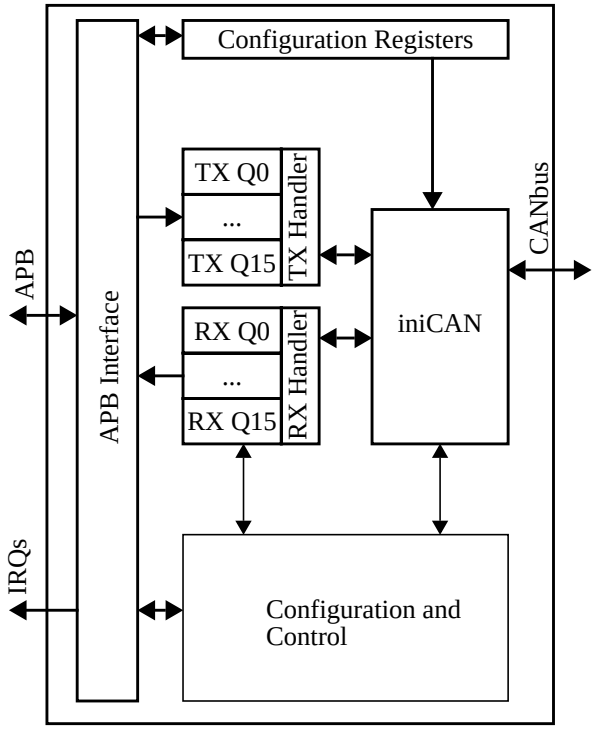
AMBA (APB) compliant

data sheet

Features:

- AMBA (APB) compliant Interface
- CAN2.0B compliant
- 1Mbit/s with > 8MHz clock
- 16 transmit buffers
- 16 messages deep receive buffer
- Message transmission timeout
- Advanced message priority handling
- Full access to internal status
- Structured, fully synchronous VHDL
- Designed for test, ready for SCAN
- Available on evaluation platform
- Utility library in C

iAP-CAN 16f Architecture



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

CAN2.0B, developed for the European car industry, became famous as a high security, fast and cost effective data link layer for multi master and real time applications.

The iAP-CAN 16f is a full CAN controller, which provides message queueing for both transmission and reception.

16 independent transmission queues allow the user to prepare several messages in advance. The intelligent priority handler will analyse the identifier of all pending messages to send and transmit the highest priority message first.

For reception message buffering, the iAP-CAN 16f provides a 16 messages deep fifo. A low overhead handshake scheme is used in order to guarantee consistency of the data structure in case of overflow. The receive path is fully interrupt driven. The iAP-CAN 16f provides status information for message polling, too.

Its structured, synchronous VHDL implementation gives you reliable results on ASIC and FPGA technolgies.

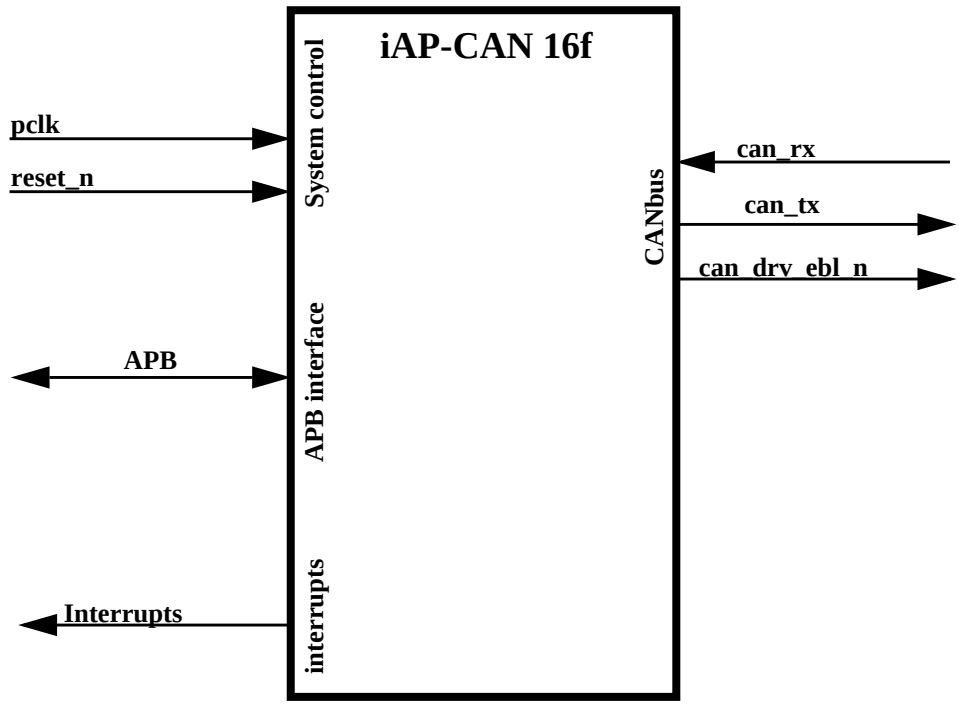
INICORE Inc. (USA)
5600 Mowry School Road, Suite 180,
Newark, CA 94560
Tel: 510 445 1529 Fax: 510 656 0995
E-mail: ask_us@inicore.com
Web: www.inicore.com

INICORE AG (Europe)
Mattenstrasse 6a,CH-2555 Brügg, Switzerland
Tel: ++41 32 374 32 00, Fax: ++41 32 374 32 01
E-mail: ask_us@inicore.ch
Web: www.inicore.ch

Ref.Nr.: V1.0 10/22/01

1 Overview

This picture gives a short overview of the iAP-CAN 16f.



The iAP-CAN 16f is an AMBA (APB) compliant full CAN controller, based on INICORE's iniCAN. It delivers a receive fifo of 16 messages and 16 transmit queues with priority handling. Interrupts allow an efficient message handling. The iAP-CAN 16f is fully configurable and can be directly used in an AMBA system. Further, the easy, synchronous APB protocol allows the usage in any system without or with minor modifications only.

1.1 General input

This pin is used to initialize the whole iAP-CAN 16f circuit.

pin name	type	size	description
reset_n	in	1	asynchronous system reset, active low

1.2 APB Interface

The APB (Advanced Peripheral Bus) is a synchronous easy to use bus architecture for peripherals from ARM. It provides the clock for the core.

pin name	type	size	description
pclk	in	1	APB clock and clock for the iAP-CAN 16f, no other clock sources needed
paddr	in	32	APB Address input, bits [6:2] used only
pwdata	in	32	APB Write data
psel	in	1	APB Select signal
penable	in	1	APB Enable signal
pwrite	in	1	APB Write signal
prdata	out	32	APB read data

1.3 Interrupts

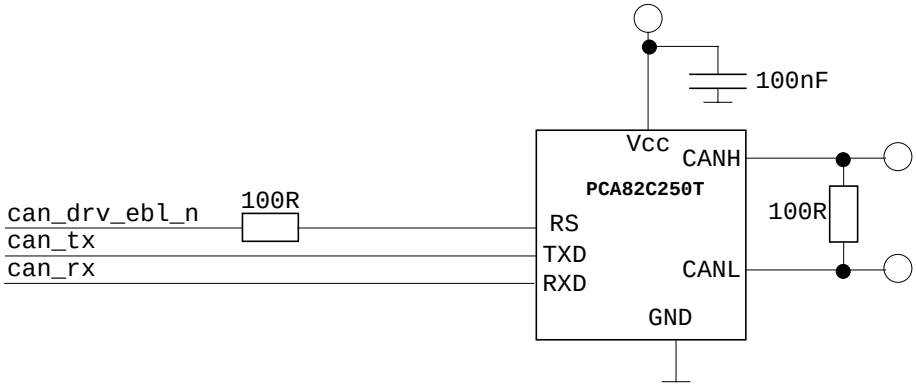
The 3 interrupt pins are active high and remain high until they are acknowledged:

pin name	type	size	description
can_int_tx	out	1	interrupt for transmit queue: active when a CAN message has been sent
can_int_rx	out	1	receive interrupt, active when a CAN message has been received
can_int_err	out	1	diagnostics interrupt for errors, etc.

1.4 CAN bus

These signals may be used to directly drive a physical bus line or an external driver. The following picture shows how to connect an external Phillips CAN driver to iAP-CAN 16f:

pin name	type	size	description
can_rx	in	1	local receive signal (connect to can_rx of external driver)
can_tx	out	1	CAN transmit signal (connect to can_tx of external driver)
can_drv_ebl_n	out	1	external driver control signal



2 Functional Description

This section explains the use of each register and block within the iAP-CAN 16f.

2.1 iniCAN

This is the actual CAN controller circuitry. It contains the complete data link layer, including the framer, transmit and receive control, error handling and reporting and synchronization logic. Please refer to the iniCAN datasheet for a more detailed description.

2.2 TX Handler / Queue

For transmission the iAP-CAN 16f provides 16 transmit queues which work as an intelligent fifo. The fifo generates status information about the number of free queues, busy/idle etc.

The TX handler is responsible for the message scheduling and the message prioritization logic. After every arbitration loss or bus idle, all pending messages in the 16 queues are analyzed and the highest priority message is sent first. This guarantees that urgent messages are sent as fast as possible and are not locked by preceding low priority messages. The TX handler compares the message ID's of all valid TX queues, selects the one with the highest priority and forwards it to the CAN controller for transmission.

2.3 RX Handler / FIFO

The RX path of the iAP-CAN 16f contains an RX handler and a 16-message deep RX FIFO with an additional message read register.

The RX FIFO is preceded by a message filter that allows to select or inhibit (depending on the comparison mode) certain messages or groups of messages. Two ID compare registers and two mask registers make this filter mechanism very simple.

The RX FIFO always holds the 16 latest messages. When the FIFO is overrun, an RX FIFO overrun bit marks this situation. The message that caused the overrun is stored in the FIFO and the oldest message is pushed out of the FIFO.

To read out a message one has to acknowledge the message first. The acknowledge causes to push the oldest message in the RX FIFO out and store it in the message read register. The pushing out of the oldest message has the advantage to release the FIFO to receive a new message. And the message read register allows to read the message or parts of it until the next acknowledge is done.

2.4 Register Description

The following table lists all registers in the iAP-CAN 16f.

Register	Size	Access	Description
CAN_CFG	[7:0]	R/W	CAN configuration: (see iniCAN datasheet) Bitrate
	[11:8]	R/W	Tseg1
	[14:12]	R/W	Tseg2
	[17:16]	R/W	sync jump width (sjw)
	[18]	R/W	Sampling
	[19]	R/W	Edge Mode
	[20]	R/W	Auto restart

Register	Size	Access	Description
CAN_CTRL	[0]	R	Can start/stop control: (see iniCAN datasheet) Stop Indicator: ‘0’: CAN is not stopped ‘1’: CAN is stopped
		W	Clear Stop: (see iniCAN datasheet) ‘0’: No effect ‘1’: Exit stop mode, start CAN synchronization
	[1]	R	Stop announcement ‘0’: Normal operation ‘1’: Stop command asserted, CAN will stop as soon as possible
		W	Set Stop: ‘0’: No effect ‘1’: Enter stop mode, stop CAN as soon as possible
CAN_STATUS	[7:0]	R	CAN status Rx error counter
	[16:8]	R	Tx error counter
	[21:20]	R	CAN error status “00”: error active (normal state) “01”: error passive “1x”: bus off
CAN_RX_FILT1	[28:0]	R/W	Filter value 1 for incoming messages. ID compare value
	[29]	R/W	IDE compare value 1
	[30]	R/W	RTR compare value 1
	[31]	R/W	Compare functionality ‘0’ compare to equal ‘1’ compare to NOT equal
CAN_RX_FILT2	[28:0]	R/W	Filter value 2 for incoming messages. ID compare value
	[29]	R/W	IDE compare value 2
	[30]	R/W	RTR compare value 2
	[31]	R/W	Compare functionality ‘0’ compare to equal ‘1’ compare to NOT equal
CAN_RX_MASK1	[28:0]	R/W	Mask register 1 for incoming message ID ‘0’ bit is don’t care ‘1’ bit is compared
	[29]	R/W	IDE mask 1
	[30]	R/W	RTR mask 1
CAN_RX_MASK2	[28:0]	R/W	Mask register 2 for incoming message ID ‘0’ bit is don’t care ‘1’ bit is compared
	[29]	R/W	IDE mask 2
	[30]	R/W	RTR mask 2

Register	Size	Access	Description
CAN_MSG_STATUS	[7:0]	R	RX FIFO fill level: 0x00: RX queue is empty 0x01: RX queue contains 1 message 0x10: RX queue contains 16 messages
	[8]	R	Message status register: RX FIFO overflow flag
		W	clear RX FIFO overflow flag ‘0’: no action ‘1’: clear flag
	[9]	R	TX queue full ‘0’: queue is not full ‘1’: queue is full
	[10]	R	TX message pending ‘0’: No message pending in queue or CAN ‘1’: Message pending in queue or sending
CAN_INT_EBL	[0]	R/W	Interrupt enable flags: TX message sent interrupt enable
	[1]	R/W	RX interrupt enable
	[2]	R/W	RX error counter > 96 interrupt enable
	[3]	R/W	TX error counter > 96 interrupt enable
	[4]	R/W	crc error interrupt enable
	[5]	R/W	form error interrupt enable
	[6]	R/W	ack error interrupt enable
	[7]	R/W	stuff error interrupt enable
	[8]	R/W	bit error interrupt enable
	[9]	R/W	arbitration loss interrupt enable
	[10]	R/W	overload interrupt enable

Register	Size	Access	Description
CAN_INT_STATUS	[0]	R	Interrupt flags: ‘0’: no interrupt active ‘1’: ‘TX message sent’ interrupt active
		W	Clear flag: ‘0’: no action ‘1’: clear flag
	[1]	R	RX interrupt enable
		W	Clear flag
	[2]	R	RX error counter > 96 interrupt enable
		W	Clear flag
	[3]	R	TX error counter > 96 interrupt enable
		W	Clear flag
	[4]	R	crc error interrupt enable
		W	Clear flag
	[5]	R	form error interrupt enable
		W	Clear flag
	[6]	R	ack error interrupt enable
		W	Clear flag
	[7]	R	stuff error interrupt enable
		W	Clear flag
	[8]	R	bit error interrupt enable
		W	Clear flag
	[9]	R	arbitration loss interrupt enable
		W	Clear flag
	[10]	R	overload interrupt enable
		W	Clear flag
CAN_RX_D0	[7:0]	R	Data bytes 1-4 of received CAN message Data 1
	[15:8]	R	Data 2
	[23:16]	R	Data 3
	[31:24]	R	Data 4
CAN_RX_D1	[7:0]	R	Data bytes 5-8 of received CAN message Data 5
	[15:8]	R	Data 6
	[23:16]	R	Data 7
	[31:24]	R	Data 8
CAN_RX_ID	[28:0]	R	Identifier of received CAN message Extended identifier
	[28:18]	R	Standard identifier

Register	Size	Access	Description
CAN_RX_CTRL	[3:0]	R	DLC and control information of received CAN message. DLC (data length code) “0000“: 0 data bytes “0001”: 1 data byte “0010”: 2 data bytes ... “1000”: 8 data bytes (maximum) other values n/a
	[4]	R	IDE (identifier extension bit) ‘0’: standard format data frame ‘1’: extended format data frame
	[5]	R	RTR (remote transmission request bit) ‘0’: data frame ‘1’: remote frame
	n/a	W	write any value to this register to move the next RX message into the RX message read register This also acknowledges the RX interrupt

Register	Size	Access	Description
CAN_TX_Q_D0	[7:0]	R/W	Transmit queue data bytes 1-4 of CAN message Data 1
	[15:8]	R/W	Data 2
	[23:16]	R/W	Data 3
	[31:24]	R/W	Data 4
CAN_TX_Q_D1	[7:0]	R/W	Transmit queue data bytes 5-8 of CAN message Data 5
	[15:8]	R/W	Data 6
	[23:16]	R/W	Data 7
	[31:24]	R/W	Data 8
CAN_TX_Q_ID	[28:0]	R/W	Transmit queue identifier Extended identifier
	[28:18]	R/W	Standard identifier
CAN_TX_Q_CTRL	[3:0]	R/W	Transmit queue DLC and control information (write access starts transmission) DLC
	[4]	R/W	IDE
	[5]	R/W	RTR

3 Programming Model

This section describes the configuration and programming of the iAP-CAN 16f.

3.1 Configuration

The iAP-CAN 16f has to be configured correctly before the CAN is started. The register CAN_CFG contains all necessary information to define baud rate, synchronization modes etc. of the CAN bus. Please refer to the iniCAN datasheet for more information.

After configuration, the CAN is started by setting the ‘Clear stop’ flag in the CAN_CTRL register. This will force the CAN core to leave stop mode and start with bus synchronization. The status of the synchronization can be polled by reading the CAN_CTRL register: Bits [1:0] indicate whether the CAN core is in transition to start/stop or in steady state:

CAN_CTRL[1:0]	**Activity**
“00”	CAN running
“01”	CAN synchronization
“10”	CAN will stop
“11”	CAN stopped

Register CAN_INT_EBL contains enable flags for the following internal interrupt flags:

Interrupt enable flag	**Routed to interrupt pin:**
‘TX message sent’ interrupt enable	can_int_tx
RX interrupt enable	can_int_rx
CAN error interrupt enable: RX error counter > 96 indicator TX error counter > 96 indicator crc error indicator form error indicator ack error indicator stuff error indicator bit error indicator arbitration loss indicator overload indicator	can_int_err

Per default, all interrupts are disabled.

Interrupts which are mapped to a common interrupt signal are ORed together.

3.2 Message Handling

3.2.1 Transmitting messages

For transmission the iAP-CAN 16f provides 16 transmit queues which can be used in a fifo mode. Status information for the queue is available through register CAN_MSG_STATUS or CAN_INT_STATUS.

For a complete CAN message, max. 4 cycles are used to fill the queue. To activate the transmission process, register CAN_TX_Q_CTRL serves as activator. Write accesses to the other tx queue registers do not affect the send process. The transmit interrupt is activated after successful transmission of a CAN message.

After every arbitration loss or bus idle, all pending messages in the 16 queues are analyzed and the highest priority identifier is sent first. This guarantees that urgent messages are sent as fast as possible and aren't locked by preceding low priority messages.

The typical send process looks as follows (example for 4-byte data message):

1. Check if the queue is not full in the CAN_MSG_STATUS. When a 'full' flag is '0', then the queue is ready and can be used.
2. Write CAN data bytes 1-4 into CAN_TX_Q_D0
3. Write CAN identifier into CAN_TX_Q_ID
4. Write CAN DLC, RTR and IDE into CAN_TX_Q_CTRL -> this also starts the transmission.
5. Wait for a transmit interrupt or poll the interrupt flag
6. Acknowledging of the transmission interrupt is done by writing a '1' to the flag in the register CAN_INT_STATUS.

To automate the transmission procedure one can set up an interrupt routine that is triggered by the TX message sent successfully interrupts. The routine acknowledges the interrupt by writing a '1' to the corresponding interrupt flag and then writes the next message into an empty TX queue and so on. Of course the first message cannot be written into a TX queue by the interrupt routine.

3.2.2 Receiving Messages

The receive path features a configurable message filter mechanism, that can select or inhibit (depending on the compare operation selected) certain messages. Two filter registers are included that are compared against the incoming message ID. The two mask registers can be used to extend the filtering in order to select or inhibit groups of messages. If the comparison is passed then the message is stored in the RX FIFO, otherwise the message is discarded.

To read out a message one has to acknowledge the message first by writing an arbitrary value into register CAN_RX_CTRL. This write access causes to push the next RX message out of the RX FIFO and store it in the RX read register. This register can be read out via CAN_RX_D0, CAN_RX_D1, CAN_RX_ID and CAN_RX_CTRL. The pushing out of the next message has the advantage to release the FIFO to receive a new message. And the RX read register allows to read the message or parts of it until the next acknowledge is done.

When the FIFO is overrun, the RX FIFO overrun bit in register CAN_MSG_STATUS is set to '1'. The message that caused the overrun is stored in the FIFO and the oldest message is pushed out of the FIFO. Therefore the FIFO holds always the latest 16 messages. Clearing the RX FIFO overrun flag takes place through a write to the corresponding flag in register CAN_MSG_STATUS.

Writing several times to register CAN_RX_CTRL flushes the RX FIFO.

Reading out of the RX FIFO can be done on a polling basis (checking the fill level via register CAN_MSG_STATUS) or it can be triggered by the RX interrupt.

3.3 Exception Handling

This section describes how CAN errors can be traced. The iAP-CAN 16f provides sticky flags for all types of CAN errors, which are:

- RX error counter > 96
- TX error counter > 96
- crc error
- form error
- ack error
- stuff error
- bit error
- arbitration loss
- overload

All of these flags have individual interrupt enable bits and are ORed together to form the CAN error interrupt, which allows to report only relevant events/errors to the software.

These flags are set to ‘1’ when such an error/event occurs and can be cleared by writing a ‘1’ to the corresponding bits in the CAN_INT_STATUS register. When a ‘1’ is written to the interrupt status flag (CAN error interrupt) in the CAN_INT_STATUS register, all sticky bits in the CAN_INT_STATUS register and the interrupt itself are cleared. Please note that the CAN error interrupt also depends on both “error counter > 96’ interrupts. The two ‘error counter > 96’ flags are set when the corresponding counter value becomes bigger than 96. These flags remain set until they are cleared again (sticky bits).

4 Implementation Guidelines

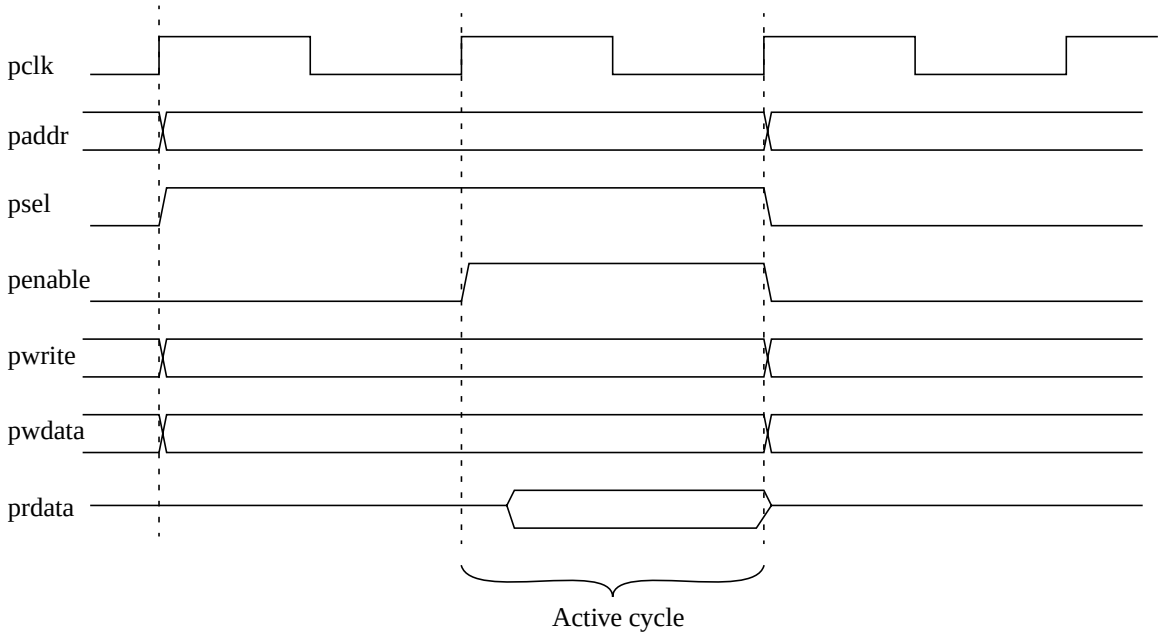
This section describes how the iAP-CAN 16f is embedded into a system.

4.1 FIFO Memory

The fifos for rx and tx message buffering are normally synthesized and does not need further attention. For FPGA and ASIC implementations, it can be replaced by a memory hardmacro, which may result in better utilization results. The fifo has synchronous write access and asynchronous read access features.

4.2 APB Bus

This bus is directly connected to the APB Bridge. Since the protocol is very basic, it can be connected to any other synchronous bus. The following timing diagram shows the timing of the iAP-CAN 16f bus, which is compatible to APB:



The iAP-CAN 16f does not count on the 2 cycle access of the APB protocol, the signals are only relevant during the high time of penable.

4.3 Register Map This table shows the (byte) address offset for the registers.

Register	Address offset
CAN_CFG	0x00
CAN_CTRL	0x04
CAN_STATUS	0x08
CAN_RX_FIFO_CFG	0x0c
CAN_RX_FILT1	0x10
CAN_RX_FILT2	0x14
CAN_RX_MASK1	0x18
CAN_RX_MASK2	0x1c
CAN_MSG_STATUS	0x20
CAN_INT_EBL	0x24
CAN_INT_STATUS	0x28
CAN_RX_D0	0x2c
CAN_RX_D1	0x30
CAN_RX_ID	0x34
CAN_RX_CTRL	0x38
CAN_TX_Q_D0	0x3c
CAN_TX_Q_D1	0x40
CAN_TX_Q_ID	0x44
CAN_TX_Q_CTRL	0x48