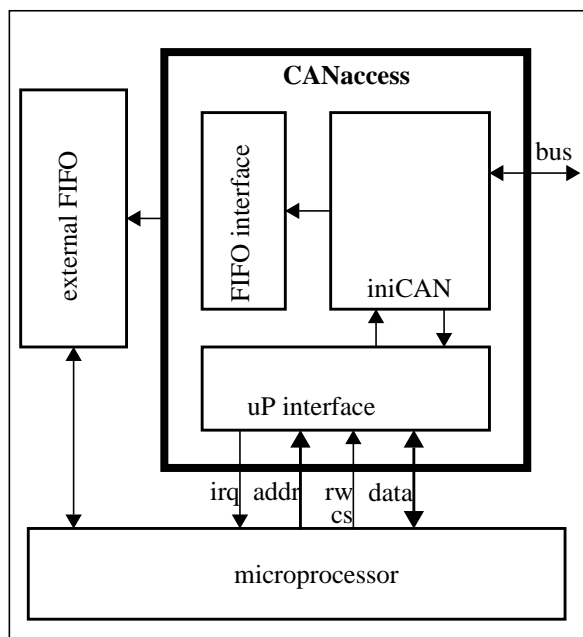




Features:

- CAN 2.0B, up to 1Mbit/s
- Trace Capability on Bit Level
- MC683xx compatible Interface (300 ns)
- Access to all Internal Status, Error Counters
- Frame Reference, TX Bus, RX Bus internally and externally available (FIFO Interface)
- Interrupt for All Errors, Message Traffic, Receiver Overrun
- Fully Synchronous Design
- ACTEL A32140DX PQFP160 Device
- Adaptable to Your Needs
- Available now!

Typical application of CANaccess:



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 multimaster and real time applications.

Since standard CAN controllers only provide functions for the main traffic, it is difficult to use them for CAN analyser tools.

For such purposes, INICORE provides a FPGA which is able to trace the whole CAN protocol on a bit level.

For high bitrates, the **CANaccess** provides a FIFO interface, where the needed information is stored and read then by the microprocessor. In slower cases, all that may directly be done by interrupt on each bit on the CAN bus. And - by the way: **CANaccess** is also a fully working CAN2.0B!

Build your next generation CAN analyser tool with **CANaccess** and talk with us about new ideas on CAN. Future products will have internal memory and will use DMA transfer.

INICORE is a senior system integrator, and migrating FPGA solutions to ASICs is a well known part of our business.



US Contact:

INICORE INC.

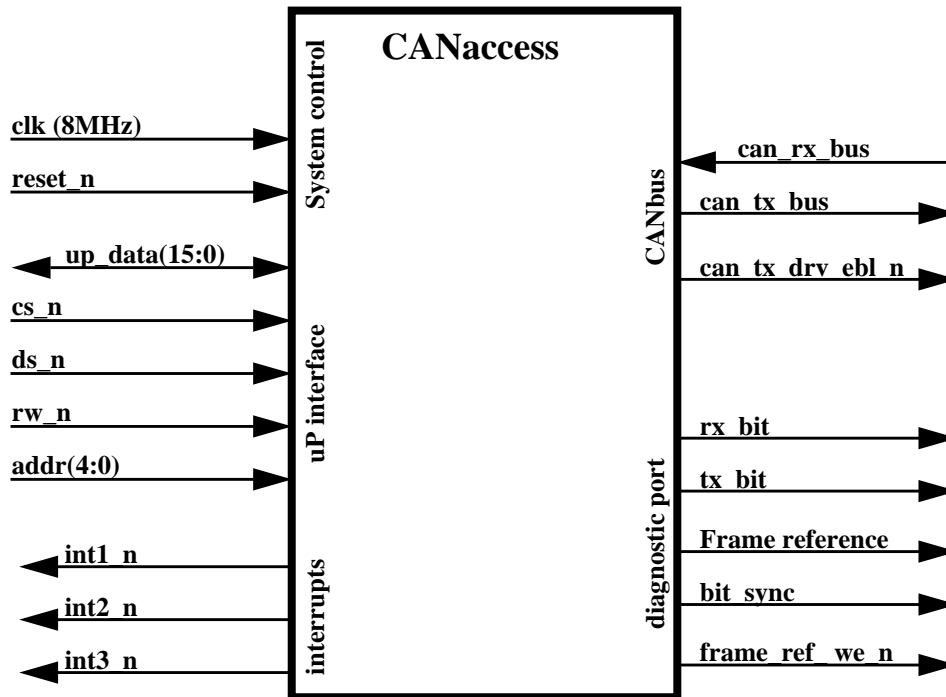
5600 Mowry School Road, Suite 180,
Newark, CA 94560
Tel: 510 445 1529 Fax: 510 656 0995
E-mail: ask_us@inicore.com

INICORE AG

Mattenstrasse 6a, CH-2555 Brugg, Switzerland
Tel: ++41 32 374 32 00, Fax: ++41 32 374 32 01
E-mail: ask_us@inicore.sme.ch
<http://www.inicore.com>

1 Overview The following picture describe the concept of the CANAccess FPGA and give an overview about its functionality.

1.1 Inputs - outputs This picture show the main inputs and outputs:



1.2 Concept

Standard CAN controllers are useful for using CAN in applications, where this field bus is used to communicate data. But they aren't very useful when you debug systems based on CAN or you want to analyze performance, traffic, errors etc., especially when bit level diagnostic features are needed.

Since INICORE has its own CAN2.0B controller (called iniCAN), we have all necessary know how to build a device that gives your microprocessor access to the CANbus on a bit level!

The CANAccess is at one side a CAN controller, but without message filter. But it has all necessary controlling structure like receive and transmit buffer, receive message overrun protection and an interrupt (int1_n) for that kind of traffic on the CANbus.

Further, there is a fully equipped error reporting circuit with error counters, error flags (crc-, bit-, form-, ack- and stuff error), as well as bus off reporting (int2_n). Through the microprocessor you have access to all this flags and status information.

At least, what makes CANAccess really different, you have fully access to the frame reference and the actual bit on the bus. This feature enables you tracing every event on the bus, analyzing errors and bus load. Also there are several diagnostic interrupts (int3_n) for bit_sync, arbitration loss, and overload conditions. For faster bitrates, there is a hardware interface to a FIFO, where the frame reference and the bus state can be stored at high speed. This interface is also configurable, that only active states are logged. For listening to the bus only, you can also set CANAccess in a passive state (it never places a dominant bit on the bus).

2 IO description The following part lists the input and output ports of the INI_CAN core and gives a short overview of their functionality.

2.1 General inputs These pins are used to clock and initialize the whole CANaccess circuit.

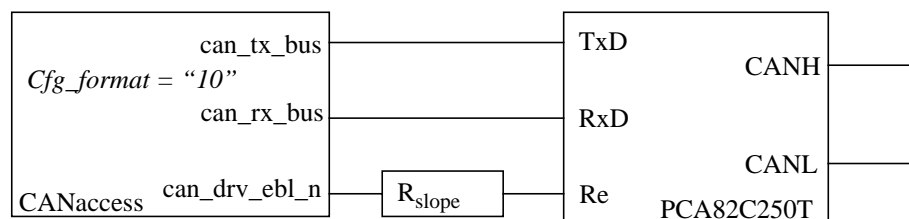
pin name	type	description
clk	in	system clock, 8MHz
reset_n	in	asynchronous system reset, active low

2.2 uP Interface The following signals are used to control the CANaccess FPGA. Through this interface, you have access to all internal registers and to the CAN bus.

pin name	type	description
cs_n	in	chip select, active low
ds_n	in	data strobe, rising edge latches data
rw_n	in	read/ write; '1' = read, '0' = write
data_drv_n	out	Data bus buffer enable signal: (see data(15:0)) '0': CAN_ACCESS drives data bus out '1': Data bus is input
data(15:0)	bi-dir	bidirectional data bus, output drive: cs_n = 0, ds_n = 0, rw_n = 1
addr(4:0)	in	address inputs for selecting internal registers
int1_n ¹	out	open drain interrupt for message traffic
int2_n	out	open drain interrupt for error reporting
int3_n	out	open drain interrupt for diagnostics

1. All int_n signals can be wire-OR connected.

2.3 CAN bus This signals may be used to directly drive a physical busline or an external driver. The following picture shows how to connect an external Phillips CAN driver to CANaccess:



pin name	type	description
can_rx_bus	in	local receive signal (connect to can_rx_bus of external driver)
can_tx_bus	in/ out	three state output, transmit signal, is CANbus or connected to external driver
can_drv_ebl_n	out	external driver control signal (unused without external driver)

2.4 CAN diagnostic

This signals may be used to directly trace what happens on the CAN bus. In the pinout, the signals appear with an index starting with ‘_’.

E.g. field(4:0 is field_4 ... field_0.

pin name	type	description (page 10)
field[4:0]	out	field pointer on actual CAN message
bit_nr[5:0]	out	bit number pointer
rx_mode	out	indicates receiving mode
tx_mode	out	indicates transmitting mode
stuff_ind	out	indicates stuff bit
remote_ind	out	indicates a remote message
extended_ind	out	indicates an extended identifier message
tx_msg_valid	out	indicates that a message has been transmitted completely
rx_msg_valid	out	indicates that a message has been received completely
bit_sync	out	timing indicator for the next bit on the CAN bus
fr_we_n	out	active low write enable for FIFO write (config see page 12)
rx_loc_msg	out	actual bit received on the CAN bus
tx_xmit_msg	out	actual bit transmitted by CANaccess controller

2.5 Power

This signals may be used to directly drive a physical busline or an external driver.

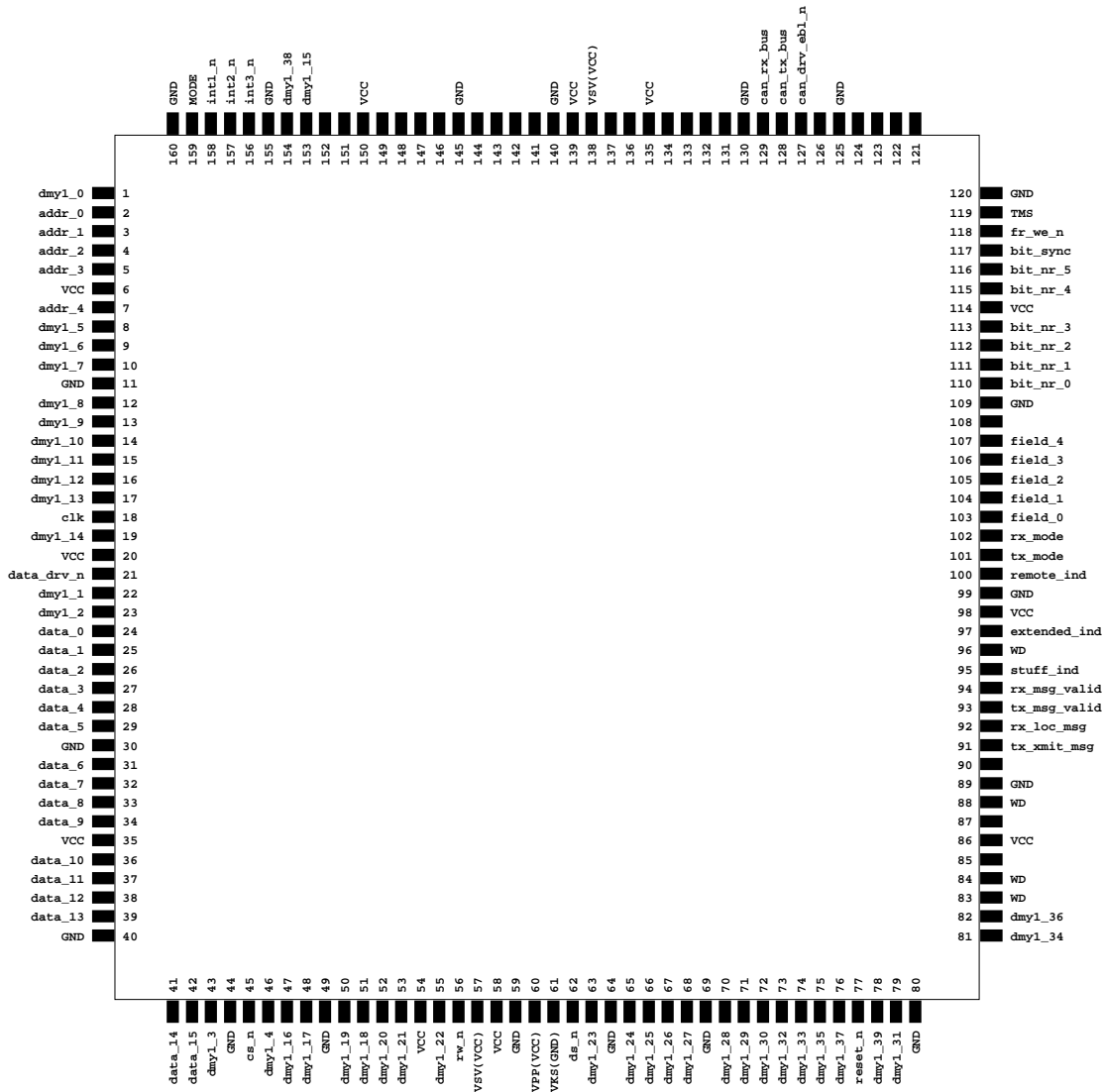
pin name	type	description (page 10)
VCC	in	5V power supply
GND	in	Ground power supply

3 Pinout

The pinout of CANAccess is shown on the ACTEL A32140DX FPGA. All power and ground pins must be connected. Unused pins remain unconnected. The package is a standard PQFP 160 pin device.

Date: Wed Dec 3 17:10:05 1997 Pinchecksum: ad5a08b8_1f83f1ad

Design Name: can_access_top Family: 3200DX Die: A32140DX Package: 160 PQFP



MODE: set to GND
 TMS: left open
 WD : left open
 dmy1_xx: set to GND
 Unnamed: left open

4 Port Map for internal registers: Port Map

Address	Type	Register
0x00	R	Transmit identifier
0x02	R	
0x04	R	Transmit data
0x06	R	
0x08	R	
0x0A	R	
0x0C	R	Transmit RTR, IDE, DLC
0x0E	R/W	Transmit control flag (TRX)
0x10	R	Receive identifier
0x12	R	
0x14	R	Receive data
0x16	R	
0x18	R	
0x1A	R	
0x1C	R	Receive RTR, IDE, DLC
0x1E	R/W	Receive control flag (RCVD)
0x20	R	Receiver/Transmitter error counter
0x22	R/W	Status and error flags
0x24	R	CAN frame reference
0x26	R/W	CAN start/stop control
0x28	R/W	Interrupt enable register
0x2A	R/W	CAN configuration bitrate
0x2C	R/W	CAN configuration mode
0x2E	R/W	CAN diagnostic configuration

Since it is a true 16bit interface, the value for the address is interpreted as a word16 address and is always even!

4.1 The following tables show the content of the internal registers:

Internal register description

Address	content: TX Message: write only							
0x00: bit[15:8]	ID_28	ID_21
0x00: bit[7:0]	ID_20	ID_13
0x02: bit[15:8]	ID_12	ID_5
0x02: bit[7:0]	ID_4	ID_0	--	--	--
0x04: bit[15:8]	D_63	D_56
0x04: bit[7:0]	D_55	D_48
0x06: bit[15:8]	D_47	D_40
0x06: bit[7:0]	D_39	D_32
0x08: bit[15:8]	D_31	D_24
0x08: bit[7:0]	D_23	D_16
0x0A: bit[15:8]	D_15	D_8
0x0A: bit[7:0]	D_7	D_0
0x0C: bit[7:0]	--	--	RTR	IDE	DLC_3	DLC_2	DLC_1	DCL_0
Address	content: TX Message control: readback							
0x0E: bit[7:0]	--	--	--	--	--	--	--	TRX

- ID_28 .. ID_0: Message identifier for both standard and extended messages. Standard messages use ID_28 .. ID_18
- D_63 .. D_0: Message data: Byte 1 is D_63 .. D_56, Byte 2 is D_55 .. D_48 etc.
- RTR: Remote bit
- IDE: Extended identifier bit
- DLC_3:DLC_0: Data length code, invalid values are transmitted as they are, but only 8 data bytes.
- TRX: Writing a '1' will send the message stored in the buffer. The buffer must not be changed while TRX is '1'! When the whole message is transferred, TRX will go low.

Address	content: RX Message: read only							
0x10: bit[15:8]	ID_28	ID_21

Address	content: RX Message: read only							
0x10: bit[7:0]	ID_20	ID_13
0x12: bit[15:8]	ID_12	ID_5
0x12: bit[7:0]	ID_4	ID_0	--	--	--
0x14: bit[15:8]	D_63	D_56
0x14: bit[7:0]	D_55	D_48
0x16: bit[15:8]	D_47	D_40
0x16: bit[7:0]	D_39	D_32
0x18: bit[15:8]	D_31	D_24
0x18: bit[7:0]	D_23	D_16
0x1A: bit[15:8]	D_15	D_8
0x1A: bit[7:0]	D_7	D_0
0x1C: bit[7:0]	--	--	RTR	IDE	DLC_3	DLC_2	DLC_1	DCL_0
Address	content: RX Message flag: write '1' to clear							
0x1E: bit[7:0]	--	--	--	--	--	--	--	RCVD ¹

1.This is a copy of the rx_msg flag in the interrupt status register

- ID_28 .. ID_0: Message identifier for both standard and extended messages. Standard messages use ID_28 .. ID_18
- D_63 .. D_0: Message data: Byte 1 is D_63 .. D_56, Byte 2 is D_55 .. D_48 etc.
- RTR: Remote bit
- IDE: Extended identifier bit
- DLC_3:DLC_0: Data length code, invalid values are received as they are!
- RCVD: RCVD will go high when a new message is received. The buffer may be overwritten by a new message (if overwrite_new_message = '1'). Writing a '1' to RCVD will clear this flag.

Address	content: CAN status, interrupt status							
0x20: bit[15:8]	rx_err_cnt[7:0]							
0x20: bit[7:0]	tx_err_cnt[7:0]							
0x22: bit[15:8]	rx_msg	tx_msg	bus_off	crc_err	form_err	ack_err	stuff_err	bit_err

Address	content: CAN status, interrupt status						
0x22: bit[7:0]	rx_ovr	bit_sync	ovr_load	arb_loss	rx > 96	tx > 96	error_state[1:0]

rx_err_cnt: The receiver error counter according the Bosch CAN specification. When in bus off, this counter is used to count the idle states.

tx_err_cnt: The transmitter error counter according the Bosch CAN specification. When it is greater than 255(dec), it is fixed at 255.

The following flags are set on internal events (they activate an interrupt line when enabled). They are cleared by writing a '1' to the according flag.

rx_msg: A message has arrived. This flag is also set, when an overrun occurred (except if overwrite_new_message = '1').

tx_msg: The message has successfully been transmitted.

bus_off: The CAN 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.

tx_ovr: Receiver overrun: Set when a message has not been acknowledged (by clearing rx_msg or RCVD) and a new message has arrived.
(see also overwrite_new_message)

bit_sync: An interrupt is generated when bit_sync is active. This makes only sense for slow bitrates!

ovr_load: An overload condition has occurred.

arb_loss: The arbitration was lost while sending a message.

The following status information is read only, there is no set/reset possible.

rx > 96: The receiver error counter is greater or equal 96(dec)

tx > 96: The transmitter error counter is greater or equal 96(dec)

error_state: The error state of the CAN node:
"00": error active (normal operation)
"01": error passive
"1x": bus off

Address	content: CAN frame reference			
0x24: bit[15:8]	Stuff_ind	TX_mod	RX_mod	frame_ref_field[4:0]
0x24: bit[7:0]	RX bit	TX bit	frame_ref_bit_nr[5:0]	

Stuff_ind: '1' means a stuff bit is inserted

TX_mod: '0': not in TX mode (receiving or idle)
 '1': transmitting data

RX_mod: '0': not in RX mode (transmitting or idle)
 '1': receiving data

frame_ref_field: This is the frame reference a incoming or outgoing CAN message. It is coded like this:
 stopped : "00000";
 synchronize : "00001";
 interframe : "00101";
 bus_idle : "00110";
 start_of_frame : "00111";
 arbitration : "01000";
 control : "01001";
 data : "01010";
 crc : "01011";
 ack : "01100";
 end_of_frame : "01101";
 error_flag : "10000";
 error_echo : "10001";
 error_del : "10010";
 overload_flag : "11000";
 overload_echo : "11001";
 overload_del : "11010";
 other codes are not used!

RX bit: The bit state on the receiver line

TX bit: The bit state on the transmitter line

frame_ref_bit_nr: A 6 bit vector used for counting the bit numbers in one field.

For example:
 When field = "data" = "01010" and "bit_nr" = "000000" and "tx_mode" = '1' that means transmitting the first data bit.

Address	content: CAN start-stop control							
0x26: bit[7:0]	--	--	--	--	--	--	set_stop	clr_stop

set_stop: Writing a '1' sets the CAN in the stop mode
 Read: '1' when stopped.

clr_stop: Writing a '1' sets the CAN in the run mode
 Read: '1' when CAN is running

Address	content: interrupt enable register							
0x28: bit[15:8]	rx_msg	tx_msg	bus_off	crc_err	form_err	ack_err	stuff_err	bit_err_
0x28: bit[7:0]	rx_ovr	bit_sync	ovr_load	arb_loss	--	--	--	int_ebl

These configuration parameters are the enable bits for the different interrupt sources. A '1' enables the interrupt.

rx_msg, tx_msg and rx_ovr:
 int1_n (traffic interrupts)

crc_err, form_err, ack_err, stuff_err, bit_err, bus_off:
 int2_n (error interrupts)

ovr_load, bit_sync, arb_loss:
 int3_n (diagnostic interrupts)

int_ebl: general interrupt enable.

Address	content: Configuration: readback					
0x2A: bit[15:8]	--	Cfg_tseg2[2:0]			Cfg_tseg1[3:0]	
0x2A: bit[7:0]	Cfg_bitrate[7:0]					
0x2C: bit[7:0]	auto restart	Cfg_sjw[1:0]	sampling mode	edge mode	Cfg_format[1:0]	tx0 polarity

Cfg_tseg1: Length - 1 of the first time segment (bit timing). It includes the propagation time segment.

Cfg_tseg2: Length -1 of the second time segment.

Cfg_bitrate: Prescaler for generating the time quantum:
 "00000000": Maximum speed (1 TQ = 1 clksys cycle)
 "00000001": 1 TQ = 2 clksys cycles
 ...
 "11111111": 1 TQ = 256 clksys cycles

auto_restart: '0': After bus off, the CAN must be started 'by hand'
 '1': After bus off, the CAN is restarting automatically after 128 groups of 11 recessive bits.

Cfg_sjw: Synchronization jump width - 1

sampling_mode: '0': One sampling point is used in the receiver path
 '1': 3 sampling points with majority decision are used

edge_mode: '0': Edge from 'R' to 'D' is used for synchronization
 '1': Both edges are used

Cfg_format: Output format of the tx pin:
 "00": open drain, 'D': driving low, 'R': tri state
 "01": open source, 'D': driving high, 'R': tri state
 "10": push-pull, driving 'R' = high and 'D' = low
 "11": off, always tri state

tx0_polarity: Output polarity of the tx pin:
 '0': tx pin is normal polarity '0' is 'D'
 '1': tx pin is inverted: '0' is 'R'

Address	content: External RAM write control: readback							
0x2E: bit[7:0]	--	--	--	--	CAN passive	overwrite new msg	write bus_idle states	write enable

- CAN passive: The output is hold at the 'R' level. The CAN is only 'listening' at the bus.
- overwrite_new_msg: '0': When a message is still in the receive buffer (RCVD = '1'), a new message will overwrite the old one and set the rx_msg flag.
'1': Under the same conditions, a new message will be discard and no rx_msg flag will be set.
- write_bus_idle_states: For the external memory and the bit_sync interrupt, this flag disables writing resp. the interrupt.
'0': Every bit time, the CAN status is written to the external RAM resp. a bit_sync interrupt is generated.
'1': if the CAN is at least 1 cycle in the bus idle mode, write to external RAM resp. bit_sync interrupt is disabled (until new activities on the bus are detected)

5
Timing

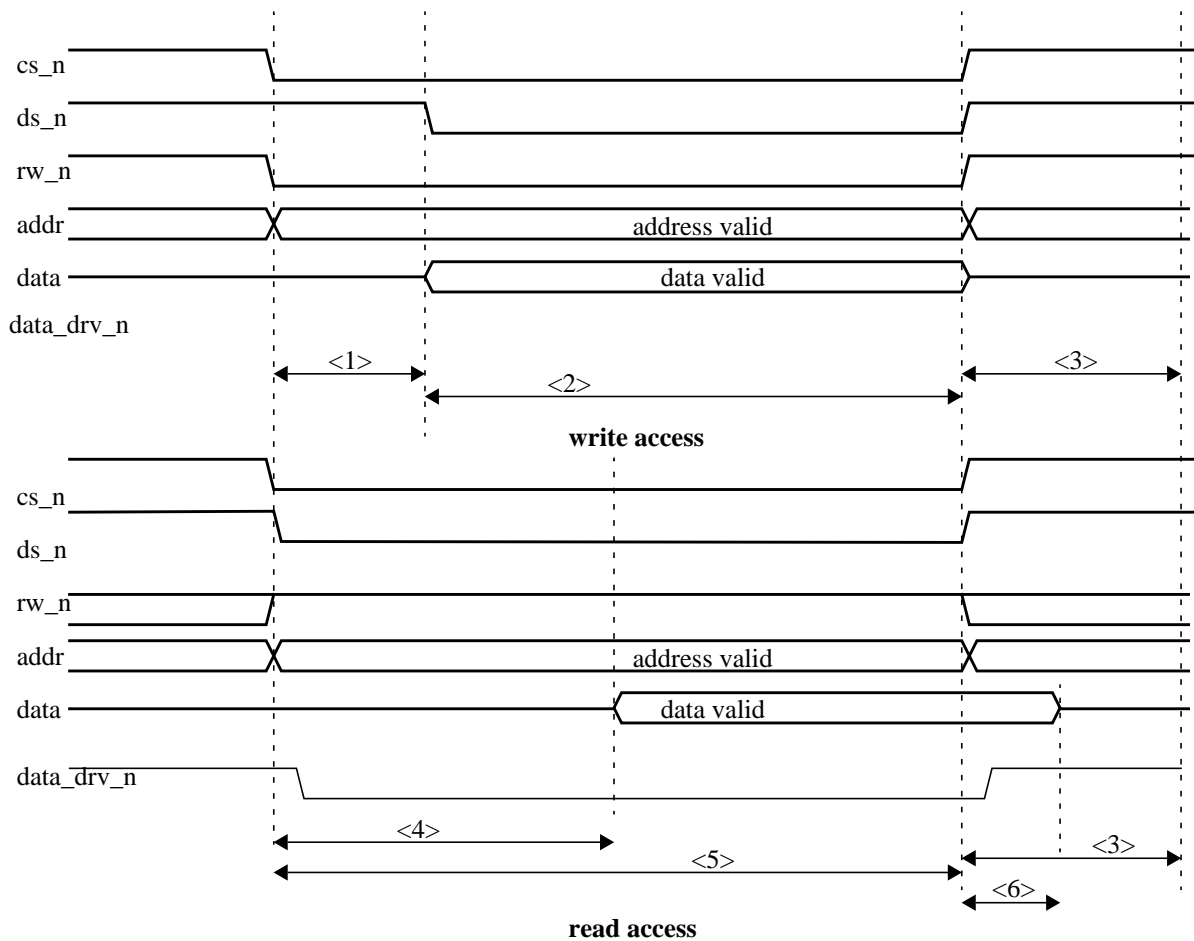
This section describes the external timing of CANaccess.

5.1
uP interface

The microprocessor interface works with signals for chip select, data strobe and read/write indication. 5 address lines select the internal registers and data is transferred via 16 bidirectional data bits. Following figures show a write and read access. Delay times are valid for 8MHz system clock.

Special remark: Synchronous type interfaces like this kind are not sensible to data setup/hold times in write cycles. But the minimum low time for ds_n is essential for a successful cycle.

Time nr	t _{min} [ns]	t _{max} [ns]	comment
<1>	0	-	ds_n falling after cs_n falling
<2>	250	-	ds_n low time (write)
<3>	125	-	cs_n high time after access
<4>	0	200	data valid after ds_n, cs_n, rw_n, addr valid
<5>	250	-	cs_n, ds_n low time (read)
<6>	10	30	data hold time after ds_n, cs_n rising



5.2
FIFO interface

The FIFO interface updates every bittime the outputs and is generating a bitsync pulse (active high) and a write enable pulse (active low) for the FIFO. The timing is shown in the following diagram: (for 8MHz cycle time)

Time nr	t _{min} [ns]	t _{max} [ns]	comment
<1>	125		bit_sync high time
<2>	125		we_n falling afer bit_sync frame_ref toggling
<3>	125		we_n low time
<4>	> 1000		1 CAN bit time

