VME - CAN4

CAN Controller for 4 CAN Nets

Hardware Manual

to Product V.1408.xx

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
www.esd-electronics.com • Fax: 0511/37 29 8-68
Phone: 0511/37 29 80 • International: +49-5 11-37 29 80

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada:
esd electronics Inc.
525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Manual File:	I:\texte\Doku\MANUALS\VME\CAN4\CAN4-15H.en9
Date of Print:	27.06.2006

Described PCB Version:	from serial no.≥ BAxxxx

Changes in the chapters

The changes in the user's manual listed below affect changes in the hardware, as well as changes in the description of the facts only.

Chapter	Alternations versus previous revision 1.0
-	General revision.
4.7	New chapter ‘4.7 Wake-Up Timer’ inserted.
6.	Chapter ‘6. User Clock’ deleted. Function is no longer available.
6.1.2	P2 pin assignment description changed.
7.3	Circuit diagrams deleted. (If required, please ask our support team for the ‘Engineering Manual’.)

Technical details are subject to change without notice.

This page is intentionally left blank.

Content

Page

This page is intentionally left blank.

1. Overview

This manual describes the hardware properties of the VME-CAN4 board. The VME-CAN4 offers four complete, independent CAN interfaces for the VMEbus in 6 U high with a width of only 4 HP.

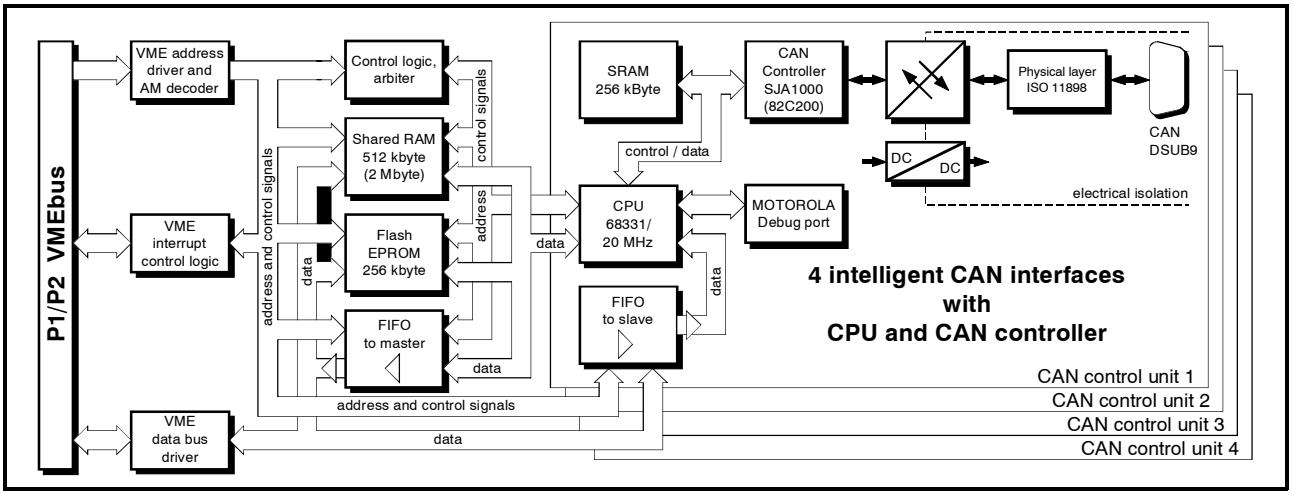


Fig. 1.1.1: Block diagram of the VME-CAN4

Each interface has its own 68331 microcontroller with a clock frequency of 20 MHz. Each microcontroller has a scratchpad memory of 256 kbytes. The program code of the microcontrollers is loaded from the local flash EPROM after a RESET. Program updates can be programmed into the flash EPROM via the VMEbus.

CAN data are transferred to and from the VMEbus via a shared RAM with a capacity of 512 kbytes and FIFOs. Received CAN-Rx data triggers a local interrupt which is handled by the respective microcontroller. It stores the data in the shared RAM and the identifier in the FIFO (FIFO to master) and triggers a VMEbus interrupt. For each of the four CAN channels their own interrupt vector is generated.

For the transmission of CAN data the VMEbus master enters the data into the shared RAM. Doing this a pointer is automatically stored on the identifier in the FIFO (FIFO to slave). The according microcontroller reads-out FIFO and RAM and initiates the transmission of data.

The CAN controller used - model SJA1000 - supports the standard CAN protocol (11-bit identifier) as well as the extended CAN protocol (29-bit identifier). The physical layer corresponds to ISO 11898 for a maximum baudrate of up to 1 Mbit/s. The baudrate can be programmed from 10 kbit/s to 1 Mbit/s.

The CAN channels are electrically isolated from each other as well as from the VME side. The local supply of the four channels is guaranteed by four DC/DC converters. Even with an electrically isolation, esd guarantees a bus length of 37 m at a transmission rate of 1 Mbit/s, provided that the other CAN users have equivalent CAN interfaces.

Overview

The CAN nets are connected by means of 9-pin DSUB female connectors in the front panel. A piggyback interface is available for other layers or connector standards (e.g. DeviceNet).

The states of the CAN controllers and the microcontrollers of the individual channels as well as the general board status are indicated by LEDs in the front panel.

The remarkable fault-tolerant CAN protocol (Hamming distance = 6) excellently suits the construction of decentralized I/O nets in which a secure and manufacturer-independent communication protocol is required because of the variety of the applications. The CAN protocol has a standardized transmission frame and a high transmission speed.

An important advantage of the CAN protocol is its ability of self-arbitration and its multimaster ability, so that the user can interconnect various sensors, actuators, CAN-I/O modules, but also VMEbus computers, PCs, PLC assemblies or stand-alone controllers in a CAN net.

1.1 Summary of the Technical Data

1.1.1 General

Plug-in dimensions	6 U high / 4 HP wide		
PCB dimensions	160 mm x 233 mm		
Weight	ca. 380 g		
Component design	SMD (components equipped on both PCB layers)		
Connectors	P1 - DIN 41612-C96 (VMEbus) P2 - DIN 41612-C96 (VMEbus and PGM signals) P680, P780, P880, P980 - DSUB9/male (CAN interfaces)		
Voltage supply via VMEbus	+5 V ±5%	Current consumption (idle, typical):	I_{1_CAN} = 1,9 A I_{2_CAN} = 2,1 A I_{4_CAN} = 2,5 A
Temperature range	max. permissible ambient temperature: 0...60 ˚C		
Humidity	max. 90%, not condensing		

1.1.2 VMEbus Interface

VMEbus interface	IEEE 1014 Rev. D
Address modifier	standard supervisory and non privileged data access, extended supervisory and non privileged data access, short supervisory and non privileged data access
Access modes	A32, A24: D8, D16, D32, ADO, UAT, RMW
Basic address	A16 - basic address selectable via geographical address (like VME64) or coding switches, A24/A32-basic addresses programmable by A16/D16 access, the board needs 2 Mbytes

1.1.3 Four CAN Units

Each one of the following assemblies is available once for each CAN channel (therefore four times all together), except the shared SRAM, the flash EPROM and the FIFO to master.

1.1.3.1 Microprocessor Assemblies

1x	FIFO to master	Memory capacity: Access:	1 kwords VMEbus read (IRQ service, word access +2 bits FIFO parity for the IACK vector), MC68331 write (word access + 2 bits FIFO parity as channel ID code)
	Shared RAM (only one)	Memory capacity: Access:	512 kbytes, optional 2 Mbytes, 68331 and VMEbus with 32-bits data width
	Flash EPROM	Memory capacity: Access:	256 kbytes, optional 1 Mbyte 68331 (word access)

4x	CPU	68331/ 20 MHz (CPU can be set to 'RESET' by means of the VMEbus.)	
	SRAM	Memory capacity: Access:	256 kbytes only via 68331
	FIFO to slave	Memory capacity: Access:	4 kbytes 68331 read (IRQ service, byte access), VMEbus write (word access)

1.1.3.2 CAN-Interface Assembly (one each per CAN channel)

CAN controller	SJA1000 (CAN2.0A/CAN2.0B)
Identifier	11-bits and 29-bits identifiers with SJA1000
Baudrate	max. 1 Mbit/s, with a maximum bus length of 37 m (provided that the other CAN users have an equivalent interface)
Physical layer	differential, ISO 11898
Connection	9-pin DSUB connectors (DIN41625) in front panel
Electrical isolation	from VMEbus and channels from each other. Isolation is made by means of optocouplers and DC/DC converters.
Voltage supply	one DC/DC converter each for the voltage supply of a CAN channel

1.1.4 Software

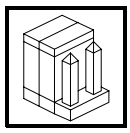
The integrated firmware offers communication on the OSI-layer 2 level or by superordinate CAN protocols.

The software will be described in a separate manual.

1.2 Order Information

Model	Properties	Order no.
VME-CAN4	Intelligent CAN-interface board with 4 independent CAN channels, 11- and 29-bits CAN identifiers (SJA1000)	V.1408.02
VME-CAN4-2	as V.1408.02, but 2 CAN channels	V.1408.04
VME-CAN4-1	as V.1408.02, but 1 CAN channel	V.1408.06
VME-CAN4-64	Option: P1 and P2 designed as 160-pin VMEbus connector (like VME64)	V.1408.12
VME-CAN4-VxW	VxWorks object licence	P.1408.15
VME-CAN4-LynxOS	LynxOS object licence	P.1408.53
VME-CAN4-LINUX	Linux object licence	P.1408.58
VME-CAN4-MD	English user's manual [1*)]	M.1408.21
VME-CAN4-ENG	Engineering manual in English [2*)] Content: Circuit diagrams, PCB top overlay drawing, data sheets of significant components	M.1408.25

1*) If module and manual are ordered together, the manual is free of charge.
2*) This manual is liable for costs, please contact our support.



2. PCB View

The coding switch SW210, the diode D210 and the resistors RA211, RA212 and RA213 are designed to set the A16-basic address. They are described in chapter ‘Accesses to the VME-CAN4 in A16 Mode’ on page 12.

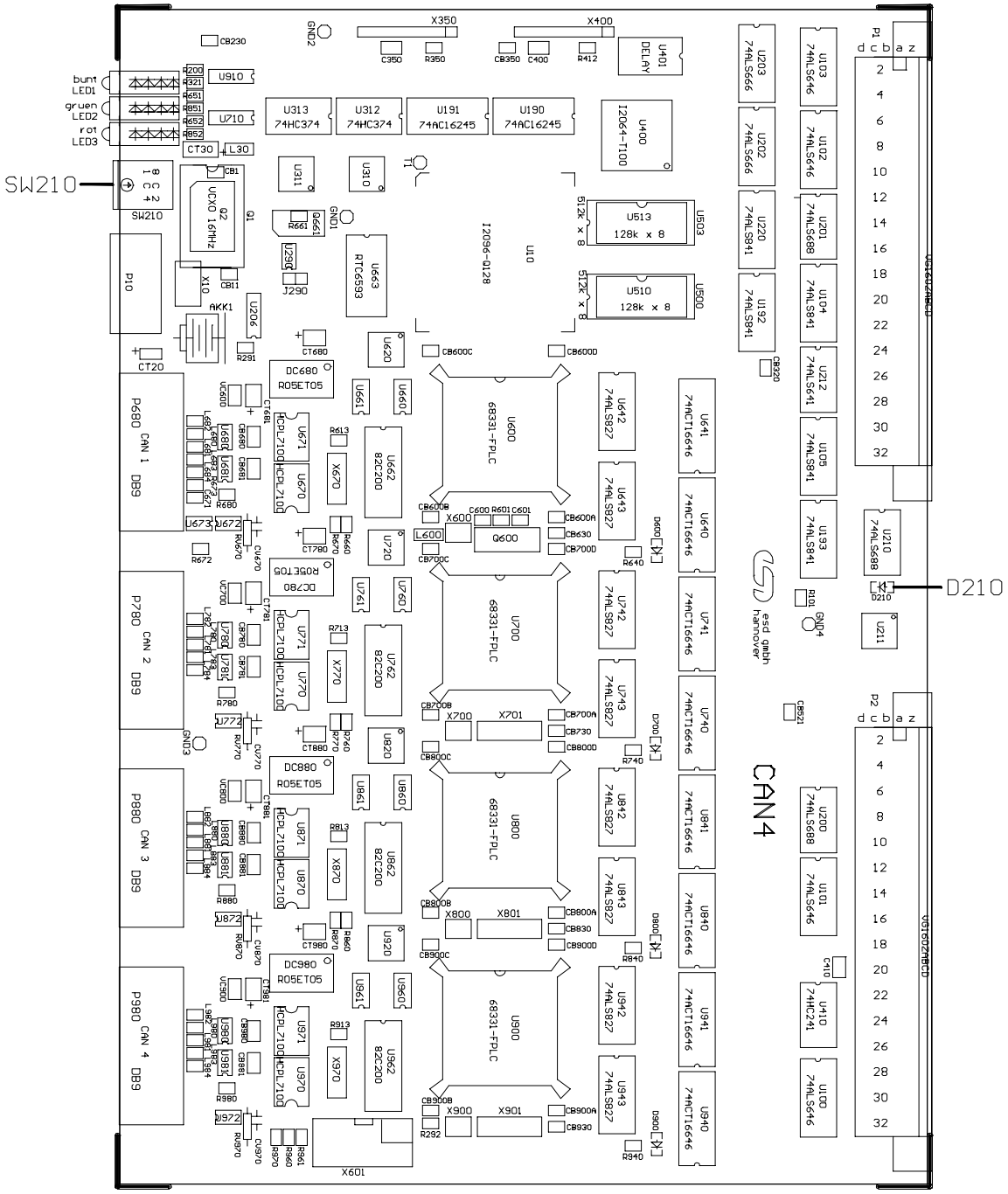


Fig. 2.1.1: Position of the components on the top layer of the PCB

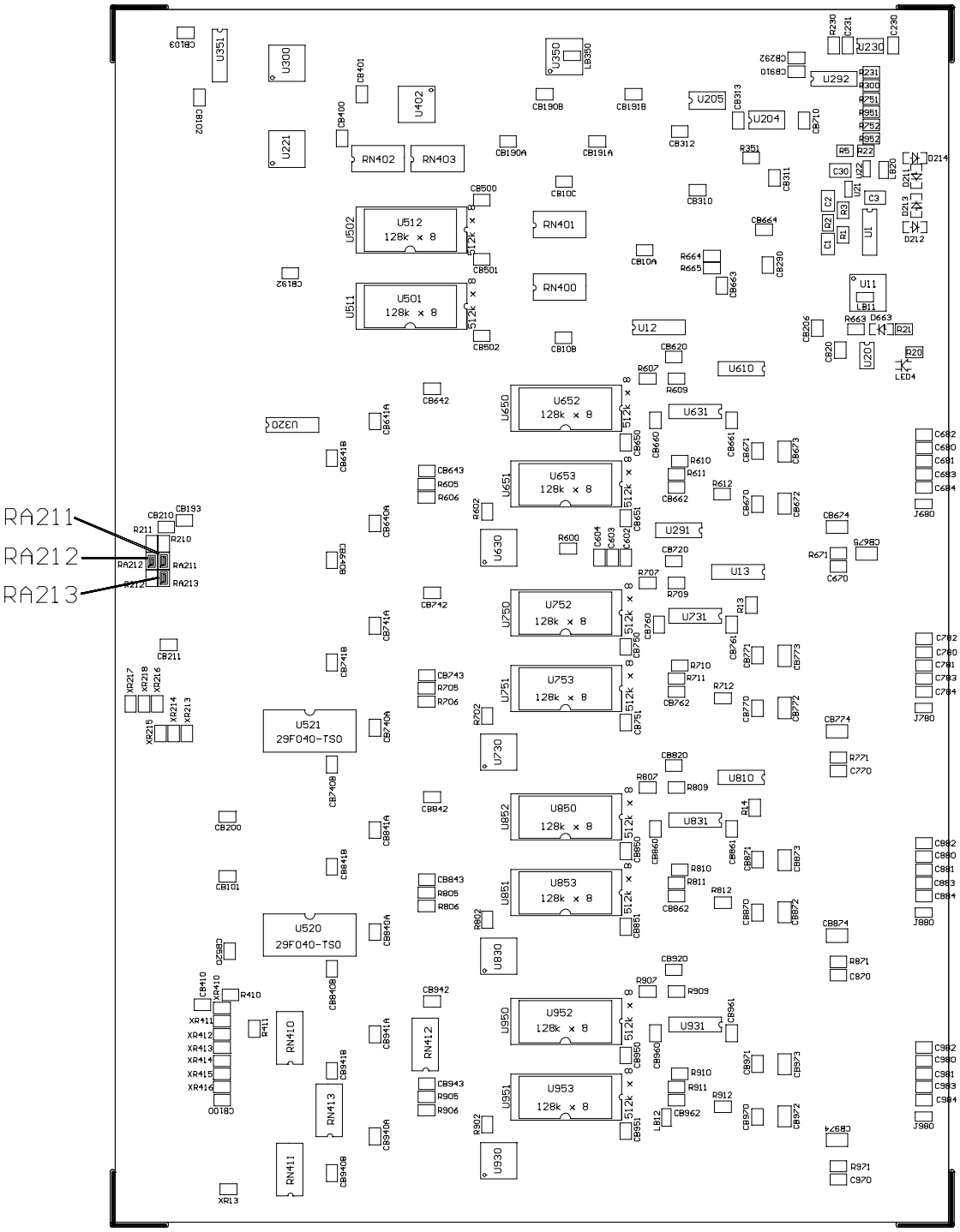


Fig. 2.1.2: Position of the components on the bottom layer of the PCB

3. Address Assignment

3.1 Address Modifier

The VME-CAN4 does not evaluate the address-modifier signal AM2. The following address-modifier combinations are permitted:

Address Modifier						permitted VMEbus access modes
AM5	AM4	AM3	AM2	AM1	AM0	
0	0	1	x	0	1	Selectable via configuration: $0D - extended supervisory data access $09 - extended non-privileged data access
1	0					Configuration area (always accessible): $2D - short supervisory access $29 - short non-privileged access
1	1					Selectable via configuration: $3D - standard supervisory data access $39 - standard non-privileged data access

Table 3.1.1: Selection of VMEbus accesses via address modifier

Short accesses (A16 mode) via the VMEbus are always possible, because the configuration of the board has to be done in this access mode (e.g. determination of the A24- or A32-VME basic address).

Depending on the configuration, standard (A24 mode) or extended accesses can *alternatively* be permitted, i.e. the two access modes are not permitted at the same time.

3.2 Accesses to the VME-CAN4 in A16 Mode

The A16 mode is always active. After a RESET the VME-CAN4 can only be operated in A16 mode, in which the board needs 256 bytes.

By means of A16 accesses the basic address of the board is programmed for A24 and A32 accesses. In addition fundamental control functions and status enquiries can be executed.

3.2.1 A16-Basic Address

The A16-basic address of the board is determined by the geographical address of the plug-in position at P1 (see also VME64 Specification, GAx (x=0...4)). If the board is used in standard-VMEbus systems, the signals GAx (x=0...4) can be simulated at the user-definable pins of the connector P2.

If no valid geographical address is recognized (parity error), the settings of a coding switch and a diode are used for setting the basic address (standard: diode equipped, i.e. A12 = '0').

The more significant addresses A13...A15 are always compared to the values set by three SMD resistors (standard: no resistor equipped, i.e. A15=A14=A13='1').

<table>
<tr><td colspan="8">VMEbus A16-basic address</td><td></td></tr>
<tr><td>A15</td><td>A14</td><td>A13</td><td>A12</td><td>A11</td><td>A10</td><td>A9</td><td>A8</td><td>A7...A0</td></tr>
<tr><td colspan="3">null ohm resistors (default: open)</td><td colspan="5">bits of the geographical address (GAx):</td><td rowspan="4">These bits select the local registers (CAR, etc.).</td></tr>
<tr><td rowspan="3">RA213</td><td rowspan="3">RA212</td><td rowspan="3">RA211</td><td>GA4</td><td>GA3</td><td>GA2</td><td>GA1</td><td>GA0</td></tr>
<tr><td>diode D210</td><td colspan="4">HEX-coding-switch bits (Hx) (only if the geographical address is invalid):</td></tr>
<tr><td>'0'</td><td>H3</td><td>H2</td><td>H1</td><td>H0</td></tr>
</table>

Table 3.2.1: Geographical address and VMEbus address

Status of the resistors RA211, RA212 and RA213 and the diodes D210	Evaluation of the assigned address bit
not equipped	‘1'
equipped	‘0'

Table 3.2.2: Coding the address by means of resistors and diodes

The resulting A16-address area of the VME-CAN4 therefore is between
`$xxxxE0yy` (default settings) and
`$xxxxFFyy`

Here `xxxx` is the VMEbus-A16 address area of the used master CPU and *`yy`* selects the local registers.

3.2.2 A16 Address Area Assignment

Address bits							relative address	Assembly	Access mode
A7	A6	A5	A4	A3	A2	A1			
0	0	x	x	x	x	0	$0000	register CAR (A24- and A32-address)	R/W
0	0	x	x	x	x	1	$0002	register CSTAT (status messages)	R
								Wake-Up Timer:	
0	1	x	x	x	x	0	$0040	write IRQ counter H, ack. IRQ	W
0	1	x	x	x	x	1	$0042	write IRQ counter L, start timer	W
1	0	x	x	a	b	0	$0081	CS-IRQ vector	R/W
1	0	x	x	x	x	1	$0082	CS-FIFO latch	R

x... address bit is not evaluated.
a, b... access by various addresses (see interrupt-vector description below).

Table 3.2.3: Address assignment in the A16-address area

The registers will be explained in detail in the chapter 'Registers of the VME-CAN4'.

3.3 Accesses to the VME-CAN4 in A24 or A32 Mode

The VME-CAN4 only reacts to the A24 or A32 accesses, if an A24 or A32-basic address had been entered into the register 'CAR' and the address has been enabled in the register.
The register will be explained in detail in the chapter 'Registers of the VME-CAN4'.

The permissible address modifiers have already been described in the chapter of the same name.

4. Registers of the VME-CAN4

4.1 Register 'CAR'

Relative Address: $00
Data: 16 bits
Access: read/write

Assignment:

Register bit (assembler)	Name	Status after RESET	Description	Level assignment	
D15 : D8	XA31 : XA24	0 : 0	determine A32-basic address	0 1	address bit = 0 address bit = 1
D7 D6 D5	XA23 XA22 XA21	0	determine A24-basic address	0 1	address bit = 0 address bit = 1
D4	XEN	0	enabling the A24 or A32-basic address	0 1	no enabling enabling
D3	X32	0	selection between A24 and A32-basic address	0 1	A32 A24
D2	XRES	0	RESET of all local CPUs 68331 (RESET active as long as '1' applies)	0 1	no RESET RESET
D1	WDOG	0	MAX813L-watchdog trigger (level has to be switched over within watchdog time of 1.6 s)	0 1	switch to '0' switch to '1'
D0	FLSHEN	0	providing the FLASH-EPROM memory area for VMEbus accesses	0 1	access to RAM access to FLASH

Table 4.1.1: Assignment of register CAR

Register

4.2 Status Register 'CSTAT'

Relative address: $02
Data: 16 bits
Access: read

Assignment:

Register bit (assembler)	Name	Description	Level assignment	
D15 : D13	REV2 REV1 REV0	current software-revision number	0 1	LOW HIGH
D12	GEOADD	geographical address is valid (if not, the HEX-coding switch is evaluated)	0 1	no yes
D11 D10 D9 D8	HVAL3 HVAL2 HVAL1 HVAL0	value set at HEX-coding switch	0 1	LOW HIGH
D7 : D4	CANERR3 : CANERR0	error messages of the four CAN channels	0 1	no error error
D3 : D0	CPUSTAT3 : CPUSTAT0	status messages of the four CPU 68331	0 1	status OK error

Table 4.2.1: Assignment of register CSTAT

4.3 IRQ-Counters HIGH and LOW

Relative address:	counter HIGH	$40
	counter LOW	$42
Data:	16 bits each	
Access:	write	

Assignment of the register Counter HIGH:

Register bit (assembler)	Name	Description
D15 : D3	-	these bits are not evaluated
D2 D1 D0	Counter D18 Counter D17 Counter D16	3 MSB of the 19-bits interrupt counter In addition to setting the counter bits a write access to this register acknowledges the interrupt.

Table 4.3.1: Assignment of register Counter HIGH

Assignment of the register Counter LOW

Register bit (assembler)	Name	Description
D15 : D0	Counter D15 : Counter D0	16 LSB of the 19-bits interrupt counter In addition to setting the counter a write access to this register starts the counting process.

Table 4.3.2: Assignment of register Counter LOW

4.4 Interrupt-Vector Register

4.4.1 Determine Interrupt Level and Vector (IRLEV)

Register name: IRLEV
Relative address: $81 (for write access)
Data: 8 bits
Access: write

Register bit (assembler)	Name	Status after RESET	Description	Level assignment	
D7 : D3	VEC7 : VEC3	x : x	programmable and readable 5 most significant bits of the VMEbus-interrupt vector	0 1	LOW HIGH
D2 D1 D0	IRLEV2 IRLEV1 IRLEV0	0	definition of the VMEbus-interrupt level	see the following table	

Table 4.4.1: Writing the interrupt-vector register IRLEV

IRLEV2	IRLEV1	IRLEV0	VMEbus-interrupt level
0	0	0	no VMEbus IRQ
0	0	1	IRQ 1
0	1	0	IRQ 2
0	1	1	IRQ 3
1	0	0	IRQ 4
1	0	1	IRQ 5
1	1	0	IRQ 6
1	1	1	IRQ 7

Table 4.4.2 Meaning of bits IRLEV0...2

4.4.2 Reading Interrupt-Vector Register

There are two possibilities on separate addresses for reading the interrupt-vector register: The 'interrupt-vector register with general status message (ISTAT)' and the 'interrupt-vector register with acknowledging the interrupt source (FSTAT)'.

The registers described below are only needed, if a polling on IRQ level is necessary.

The use of the 5 MSB is the same in both cases. They can be freely programmed by the user.

The registers differ in the 3 LSB and in function. The 'interrupt-vector register with general status message (ISTAT)' only returns in these bits, whether a FIFO interrupt applies or not. It is needed to poll the FIFO and does not change the FIFO status (no data is read back). It does not say anything about the triggering CAN channel.

If a read access is made to the 'interrupt-vector register with acknowledging the interrupt source (FSTAT)', and a FIFO IRQ is available, the current FIFO contents is automatically read back and written into the FIFO-latch register. In that case the FIFO latch *must* be read back *immediately afterwards* so that the data is not lost (e.g. by a read access of another master to the vector register)! The address distributor has been chosen in a way that the interrupt-vector register and the latch can be read by the CAN data by means of a longword read access to **$88**. This guarantees that no data is lost.
The 3 LSBs of the interrupt-vector register contain information about the channel that triggered the interrupt.

Register

4.4.2.1 Reading the Interrupt-Vector Register with Acknowledging the Interrupt Source

Register name: FSTAT
Relative address: $89
Data: 8 bits
Access: read/write

Assignment of the interrupt-vector register FSTAT:

Register bit (assembler)	Name	Status after RESET	Description	Level assignment	
D7 : D3	VEC7 : VEC3	x : x	programmable and acknowledgeable 5 most significant data bits of the interrupt vector	0 1	LOW HIGH
D2 D1 D0	FSTATQ2 FSTATQ1 FSTATQ0	x x x	acknowledgement of the interrupt source	see the following table	

Table 4.4.4: Assignment of register FSTAT

FSTATQ2 (D2)	FSTATQ1 (D1)	FSTATQ0 (D0)	Vector	Interrupt source
0	0	0	0	FIFO CAN channel 0
0	0	1	1	FIFO CAN channel 1
0	1	0	2	IRQ counter
0	1	1	3	68331 interrupt (collect IRQ)
1	0	0	4	FIFO CAN channel 2
1	0	1	5	FIFO CAN channel 3
1	1	0	6	not used
1	1	1	7	no interrupt -> hardware error !

Table 4.4.5: Acknowledgement of the interrupt source

Reading this register initiates the transmission of the FIFO data and the FIFO latch, if the interrupt has been triggered by the FIFO.
Furthermore in the interrupt-acknowledge cycle this register corresponds to the returned IRQ vector.

4.4.2.2 Reading the Interrupt-Vector Register with General Status Message

Register name: ISTAT
Relative address: $8D
Data: 8 bits
Access: read/write

Assignment of the interrupt-vector register ISTAT:

Register bit (assembler)	Name	Status after RESET	Description	Level assignment	
D7 : D3	VEC7 : VEC3	x : x	programmable and readable most significant 5 data bits of the interrupt vector	0 1	LOW HIGH
D2 D1 D0	FIFO-IRQ CNT-IRQ 332-IRQ	x x x	acknowledgement of the interrupt source, each one bit per IRQ source	0 1	no IRQ IRQ applies

Table 4.4.4: Assignment of register ISTAT

Reading this register has no influence on the FIFO latch.

Register

4.4.2.3 Reading the Interrupt-Vector Register without Status Message

Only to be used for testing!

Register name: IRLEV
Relative address: $85
Data: 8 bits
Access: read

Assignment of the interrupt-vector register IRLEV:

Register bit (assembler)	Name	Status after RESET	Description	Level assignment	
D7 : D3	VEC7 : VEC3	x : x	return of the programmed 5 most significant data bits of the interrupt vector	0 1	LOW HIGH
D2 D1 D0	IRLEV2 IRLEV1 IRLEV0	x x x	return of the programmed interrupt level	see the table on page 18	

Table 4.4.4: Assignment of register IRLEV

Reading this registers always transmits the FIFO data into the FIFO latch. It is impossible however, to find out from which channel the data are. The register should only be read for testing matters!

4.5 FIFO Latch

Relative address: $82, $86, $8A, $8E (reflected)
Data: 16 bits
Access: read

Register bit (assembler)	Name	Status after RESET	Description
D15 : D0	DATA15 : DATA0	x : x	CAN-FIFO data

x... not defined

Table 4.5.1: Assignment of register FIFO latch

4.6 Interrupt Priority

If more than one local interrupt occur at the same time, they are handled in the interrupt-acknowledge cycle in the following priority:

Prio IRQ

High: counter IRQ
Mid: 331-IRQ
Low: FIFO-IRQ

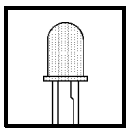
Register

4.7 Wake-Up Timer

The VME-CAN4 can trigger interrupts at the VMEbus to execute independent scheduling by means of an independent 16-bit counter with a resolution of 1 µs. This Wake-Up timer can only be accessed by the VMEbus, not by the local CPUs of the VME-CAN4.

The generated VMEbus interrupt is managed by a local interrupt handler and has a specific interrupt vector.

The Wake-Up timer is accessed via the A16-address range of the board. The addresses of the timer are printed at page 13.



5. LED Description

5.1 Position of the LEDS in the Front Panel

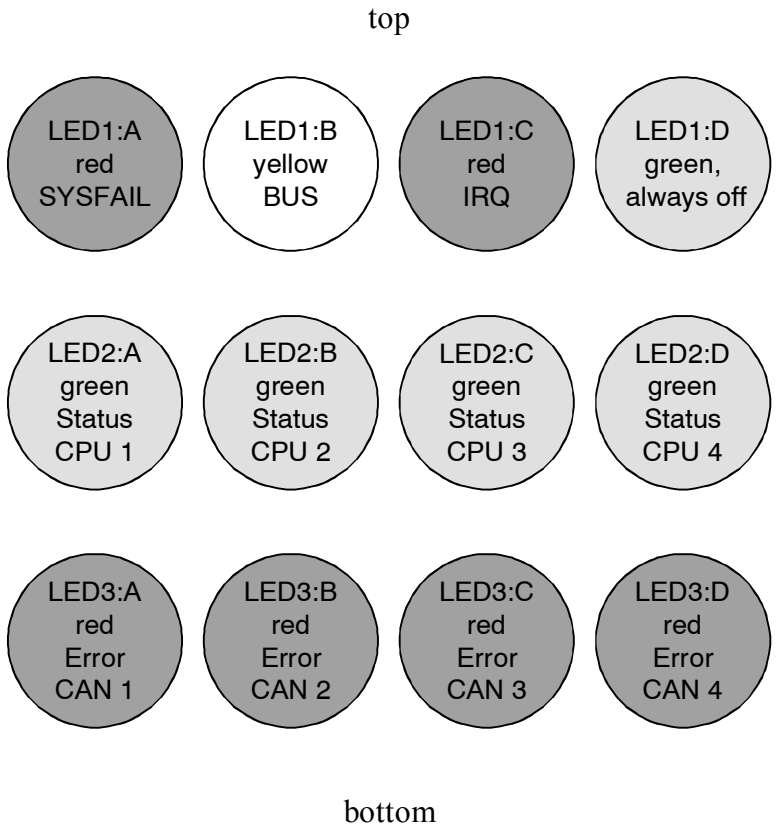
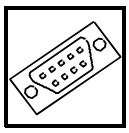


Fig. 5.1.1: Assignment of LEDs in the front panel

5.2 Function of the LEDs

LED	Function	
	LED off	LED on
LED 1:A red SYSFAIL	SYSFAIL not active.	SYSFAIL active The signal is active, if at least one CPU is not initialized. The SYSFAIL signal is guided to the VMEbus.
LED 1:B yellow BUS	No VMEbus access to the VME-CAN4.	Access from VMEbus to the VME-CAN4.
LED 1:C red IRQ	No VMEbus interrupt of the VME-CAN4 is active.	The VMEbus-interrupt signal of the VME-CAN4 is active.
LED 1:D green User	From PCB version ≥ BA*xxxx*. on, this LED is no longer in use. The LED is always off.	
LED2:A...D green Status CPU1...4	The respective CPU is not initialized.	The initialization of the CPU has been successfully completed.
LED 3:A...D red Error CAN 1...4	The error flag of the respective CAN controller is inactive.	The error flag of the respective CAN controller is active.



6. Appendix

6.1 Connector Assignments

6.1.1 VMEbus P1

Pin	Row z	Row a	Row b	Row c	Row d
1	-	D00	-	D08	-
2	GND	D01	-	D09	GND
3	-	D02	-	D10	-
4	GND	D02	BG0IN*]	D11	-
5	-	D04	BG0OUT*]	D12	-
6	GND	D05	BG1IN*]	D13	-
7	-	D06	BG1OUT*]	D14	-
8	GND	D07	BG2IN*]	D15	-
9	-	GND	BG2OUT*]	GND	BGAP*
10	GND	-	BG3IN*]	SYSFAIL*	BGA0*
11	-	GND	BG3OUT*]	-	BGA1*
12	GND	DS1*	-	SYSRESET*	-
13	-	DS0*	-	LWORD*	BGA2*
14	GND	WRITE*	-	AM5	-
15	-	GND	-	A23	BGA3*
16	GND	DTACK*	AM0	A22	-
17	-	GND	AM1	A21	BGA4*
18	GND	AS*	-	A20	-
19	-	GND	AM3	A19	-
20	GND	IACK*	GND	A18	-
21	-	IACKIN*	-	A17	-
22	GND	IACKOUT*	-	A16	-
23	-	AM4	GND	A15	-
24	GND	A07	IRQ7*	A14	-
25	-	A06	IRQ6*	A13	-
26	GND	A05	IRQ5*	A12	-
27	-	A04	IRQ4*	A11	-
28	GND	A03	IRQ3*	A10	-
29	-	A02	IRQ2*	A09	-
30	GND	A01	IRQ1*	A08	-
31	-	-12 V	-	+12 V	-
32	GND	+5 V	+5 V	+5 V	-

male multipoint connector according to DIN41612, I_{max} per pin : 1.0 A

The connector rows ‘z’ and ‘d’ are only available, if the 160-pin connector design (option ‘VME-CAN4-64') is equipped.
]... Signals bridged on board
- ... Signal not connected on board

6.1.2 Signals on P2

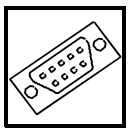
Pin	Row z	Row a	Row b	Row c	Row d
1	-	-	+5 V	-	-
2	-	-	GND	-	-
3	-	-	-	-	-
4	-	-	A24	-	-
5	-	PGA0*	A25	PGA1*	-
6	-	PGA2*	A26	PGA3*	-
7	-	PGA4*-	A27	PGAP*	-
8	-	-	A28	-	-
9	-	-	A29	-	-
10	-	-	A30	-	-
11	-	-	A31	-	-
12	-	-	GND	-	-
13	-	-	+5 V	-	-
14	-	-	D16	-	-
15	-	-	D17	-	-
16	-	-	D18	-	-
17	-	-	D19	-	-
18	-	-	D20	-	-
19	-	-	D21	-	-
20	-	-	D22	-	-
21	-	-	D23	-	-
22	-	-	GND	-	-
23	-	-	D24	-	-
24	-	-	D25	-	-
25	-	-	D26	-	-
26	-	-	D27	-	-
27	-	-	D28	-	-
28	-		D29	-	-
29	-	-	D30	-	-
30	-	-	D31	-	-
31	-	-	GND	-	-
32	-	-	+5 V	-	-

male multipoint connector according to DIN41612

The connector rows ‘z’ and ‘d’ are only available, if the 160-pin connector design (option ‘VME-CAN4-64') is equipped.

I_{max} per pin : 1.0 A

See the following page for signal descriptions.

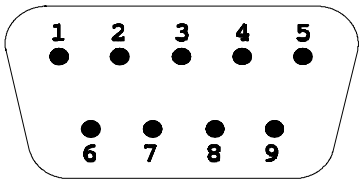


6.1.3 Meaning of the I/O Signals on P2

PGA0*... PGA4*, PGAP*...	Signals for simulating the geographical address on P2.

6.1.4 Connector of the CAN Interfaces P680...P980 (9-pin DSUB male)

Pin Location:

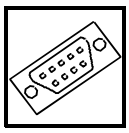


Pin Assignment:

Signal	Pin		Signal
		1	reserved
CAN_GND	6	2	CAN_L
CAN_H	7	3	CAN_GND
reserved	8	4	reserved
reserved	9	5	Shield

9-pole DSUB male

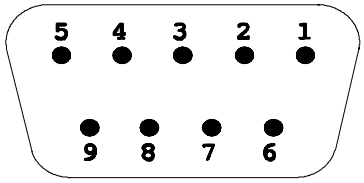
CAN_L, CAN_H...	CAN signal lines
CAN_GND ...	reference potential of the local CAN-physical layer
Shield...	can be connected to the potential of the connector case by means of a soldering bridge
reserved ...	reserved for future applications



6.1.5 Optional Mini-DSUB Connector in the Front Panel

This connector is not equipped in the standard version of the VME-CAN4.

Pin Location:

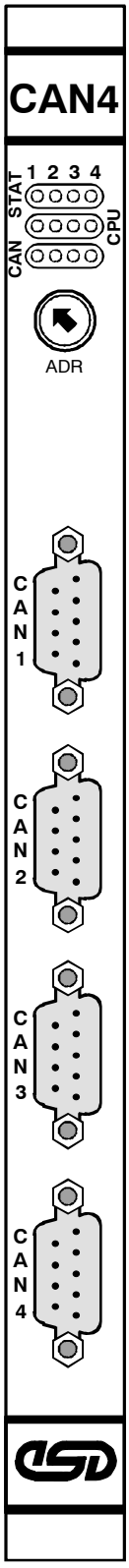


Pin Assignment:

Signal	Pin		Signal
+5V	1	6	CA0
CA1	2	7	CA2
CA3	3	8	CA4
CA5	4	9	CA6
GND	5		

CA0...CA6... The function of these signals is not yet defined.

6.2 Front Panel



7. Correctly Wiring Electrically Insulated CAN Networks

Generally all instructions applying for wiring regarding an electromagnetic compatible installation, wiring, cross sections of wires, material to be used, minimum distances, lightning protection, etc. have to be followed.

The following **general rules** for the CAN wiring must be followed:

1.	A CAN net must not branch (exception: short dead-end feeders) and has to be terminated by the wave impedance of the wire (generally 120 W ±10%) at both ends (between the signals CAN_L and CAN_H and **not** at GND)!
2.	A CAN data wire requires **two twisted** wires and a wire to conduct the reference potential (CAN_GND)! For this the shield of the wire should be used!
3.	The reference potential CAN_GND has to be connected to the earth potential (PE) at **one** point. Exactly **one** connection to earth has to be established!
4.	The bit rate has to be adapted to the wire length.
5.	Dead-end feeders have to kept as short as possible (l < 0.3 m)!
6.	When using double shielded wires the external shield has to be connected to the earth potential (PE) at **one** point. There must be not more than **one** connection to earth.
7.	A suitable type of wire (wave impedance ca. 120 Ω ±10%) has to be used and the voltage loss in the wire has to be considered!
8.	CAN wires should not be laid directly next to disturbing sources. If this cannot be avoided, double shielded wires are preferable.

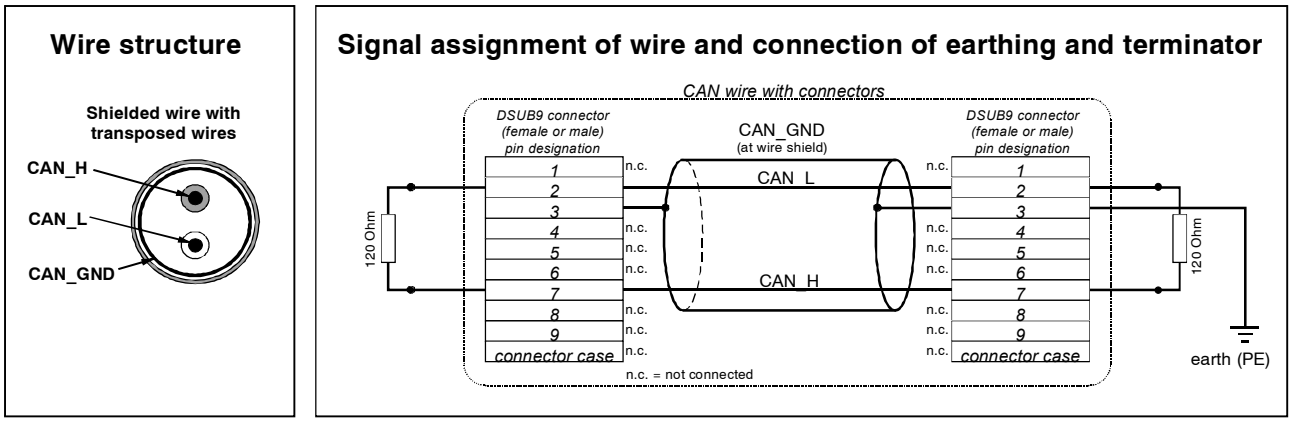


Figure: Structure and connection of wire

Cabling

- for devices which have only one CAN connector per net use T-connector and dead-end feeder (shorter than 0.3 m) (available as accessory)

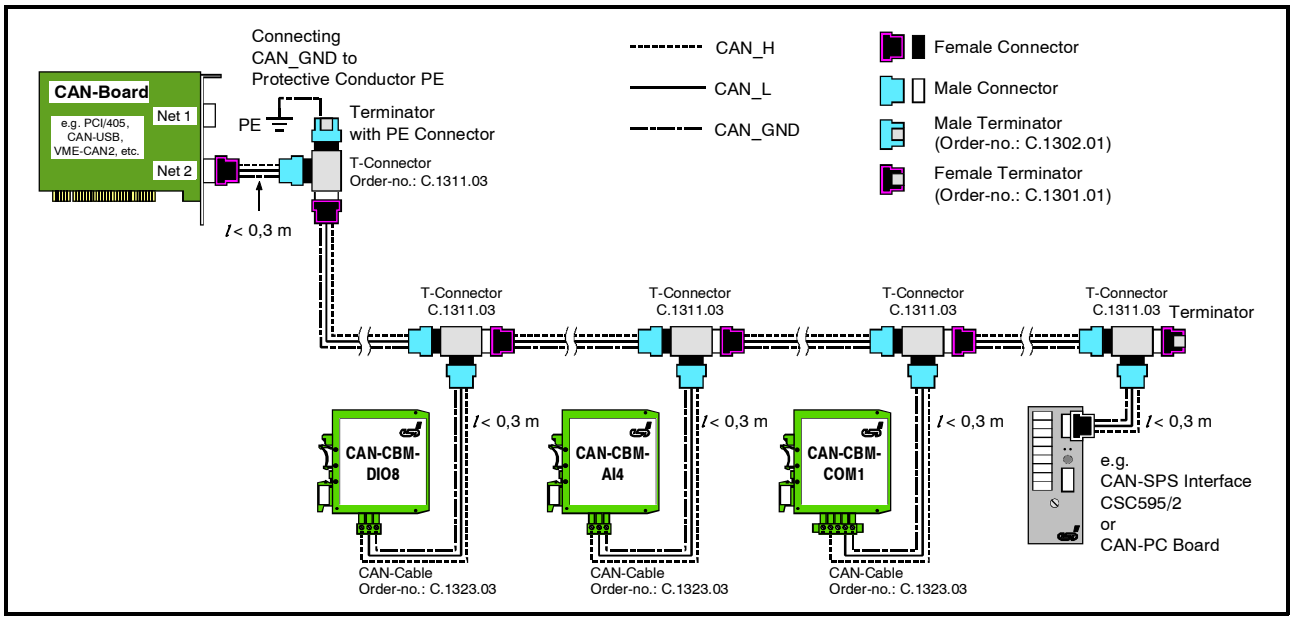


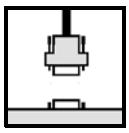
Figure: Example for correct wiring (when using single shielded wires)

Terminal Resistance

- use **external** terminator, because this can later be found again more easily!
- 9-pin DSUB-terminator with male and female contacts and earth terminal are available as accessories

Earthing

- CAN_GND has to be conducted in the CAN wire, because the individual esd modules are electrically isolated from each other!
- CAN_GND has to be connected to the earth potential (PE) at **exactly one** point in the net!
- each CAN user without electrically isolated interface works as an earthing, therefore: do not connect more than one user without potential separation!
- Earthing CAN e.g. be made at a connector



Wire Length

❍ Optical couplers are delaying the CAN signals. By using fast optical couplers and testing each board at 1 Mbit/s, however, esd CAN guarantee a reachable length of 37 m at 1 Mbit/s for most esd CAN modules within a closed net without impedance disturbances like e.g. longer dead-end feeders. (Exception: CAN-CBM-DIO8, -AI4 and AO4 (these modules work only up to 10 m with 1 Mbit/s))

Bit rate [Kbit/s]	Typical values of reachable wire length **with esd interface** l_{max} [m]	**CiA recommendations** (07/95) for reachable wire lengths l_{min} [m]
1000	37	25
800	59	50
666.6	80	-
500	130	100
333.3	180	-
250	270	250
166	420	-
125	570	500
100	710	650
66.6	1000	-
50	1400	1000
33.3	2000	-
20	3600	2500
12.5	5400	-
10	7300	5000

Table: Reachable wire lengths depending on the bit rate when using esd-CAN interfaces

Examples for CAN Wires

Manufacturer	Type of wire
U.I. LAPP GmbH Schulze-Delitzsch-Straße 25 70565 Stuttgart Germany www.lappkabel.de	e.g. UNITRONIC ®-BUS CAN UL/CSA (UL/CSA approved) UNITRONIC ®-BUS-FD P CAN UL/CSA (UL/CSA approved)
ConCab GmbH Äußerer Eichwald 74535 Mainhardt Germany www.concab.de	e.g. BUS-PVC-C (1 x 2 x 0.22 mm²) Order No.: 93 022 016 (UL appr.) BUS-Schleppflex-PUR-C (1 x 2 x 0.25 mm²) Order No.: 94 025 016 (UL appr.)
SAB Bröckskes GmbH&Co. KG Grefrather Straße 204-212b 41749 Viersen Germany www.sab-brockskes.de	e.g. SABIX® CB 620 (1 x 2 x 0.25 mm²) Order No.: 56202251 CB 627 (1 x 2 x 0.25 mm²) Order No.: 06272251 (UL appr.)

Note: Completely configured CAN wires can be ordered from **esd**.

VME - CAN4

CAN Controller for 4 CAN Networks

Manual of the local Firmware

VME-CAN4 Software Rev. 1.2

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 205
D-30165 Hannover
Germany

Tel: +49-511-37298-0
Fax: +49-511-37298-68
Email: info@esd-electronics.com
Internet: http://www.esd-electronics.com

Manual file	I:\TEXTE\DOKU\MANUALS\VME\CAN4\CAN4-12S.EN6
Date of print	04.10.1999

Software version described	

Changes in the chapters

The changes in the users' manual listed below affect changes in the **software** as well as changes in the **description** of facts only.

Chapter	Changes versus previous version
-	First English version.

Technical details are subject to change without further notice.

Content

1. Overview

1.1 About this Manual

This manual describes the local firmware of the CAN4. The firmware has been stored in the Flash EPROM.

2. Initializing the CAN4

During initialization of the CAN4 by the VMEbus master the VMEbus address A24 or A32 is assigned to the board. **The registers for setting the VMEbus basis addresses are described in the hardware manual.**

Initialization sequence (example):

1. Searching for CAN4 in A16 address range

2. Evaluating VMEbus structure

2.1 Initializing the basis address of the board in A32/D32 range:

- check, whether 32-bits data transfer (aligned) is successful
- if yes -> OK, exit
- if no -> P2 not is not equipped -> BUSERROR caused by missing address A24...A32
  -> Read/Write check faulty -> 2.2

2.2 Initializing basis address of board in A24/D16 range:

- check, whether Read/Write OK
- if yes -> OK, exit
- if no -> hardware error?

3. Determine basis address of board

Further information on determining the address can be taken from the description of registers in the hardware manual.

3. Memory Structure of CAN4

3.1 Shared RAM Assignment

The following table shows the assignment of the Shared RAM range accessible via VMEbus. The address offset is shown in relative to the programmed VMEbus basis address.

All addresses (except for two cells of the link structure) are assigned dynamically. The following table is as at 19.03.97. The absolute values of the addresses may change in future upgrades! When programming you have to use the according pointer cells instead of the absolute address values, therefore!
The link structure of the CAN4 will be described in the following chapter.

Address range from... [HEX]	Assignment	Notes
000.000 ...	Link structure read only	This memory area must only be read accessed by the user.
001.000 ... 001.100 ... 001.200 ... 001.300 ...	iobuff 1 iobuff 2 iobuff 3 iobuff 4	Command buffer of the 4 CAN channels
001.400 ...	free memory	free memory under buffer management (direct access by user is not permissible)
020.000 ... 030.000 ... 040.000 ... 050.000 ...	CAN #1 CAN #2 CAN #3 CAN #4	data and control buffer of the 4 CAN channels
060.000 ...	free memory	free memory under buffer management (direct access by user is not permissible)
07F.F00 ...	Irq_Trigger_Cells	interrupt trigger cells
080.000 ... ...1FF.FFE	free memory	free memory under buffer management (direct access by user is not permissible) (only available, if option 4x 512 kbyte SRAM is equipped)
1FF.FF0	Read_Area_Clock	register of user-defined cycle handling

Table 3.1.1: Assignment of Shared RAM 1

3.2 Local Link Structure

Except for cells `c4ioff` and `c4iolen`, which are on relative addresses +$0 and +$2, and the three long cells for identification, all addresses are managed dynamically. The address values are stored into memory cells.
All cells can only be read by the user and cannot be changed!

The following table shows the basis link structure. The first cell `c4ioff` shows the relative address of the link structure of the first CAN channel. Structure and length of link structures are the same for all four channels. The length is specified in `c4iolen`.

Address offset [HEX]	Data width	Contents	Notes	Default value after RESET [HEX] (as at 20.03.97)
+0	word	`ctoff`	offset address of link structure of first CAN channel	10
+2	word	`ctlen`	length of link structures of CAN channels (in bytes)	10
+4	long	`c4id`	ASCII-ID	‘CAN4'
+8	long	`swrev`	ASCII-ID	‘5X__’
+C	long	`swcms`	ASCII-ID	‘_CMS’

Table 3.2.1: Cells with pointers

The following table shows the structure of link structures **by means of the first CAN channel**. The structure starts from the address which has been specified in c4ioff. The link structures of the other three CAN channels directly follow this structure. From the length of the structure (iolen), therefore, the addresses of the link structures of the other channels can be determined.

Address offset c4ioff + [HEX]	Data width	Contents	Notes	Default value after RESET [HEX] (as at 20.03.97)
+0	long	c4cdat	offset address of first CAN-data structure	20000
+4	long	c4iobf	offset address of first CAN-command channel	1000
+8	long	c4ctrg	interrupt trigger address for local interrupt (CAN1)	0007.FF00
+C	long	c4mtrg	interrupt trigger address for interrupt to VMEbus (CAN1)	0007.FF20

Table 3.3.1: Structure of link structures by means of CAN channel 1

With the current default settings the following link structures were to be found under the following addresses, therefore:

CAN-link structure 2: $20
CAN-link structure 3: $30
CAN-link structure 4: $40

4. Description of the Various Memory Ranges

4.1 Data Structures CAN-Data 1...CAN-Data 4

4.1.1 Function and Structure

Data is exchanged between VMEbus and CAN via data-structure fields.

A structure field is divided into 2048 elements.

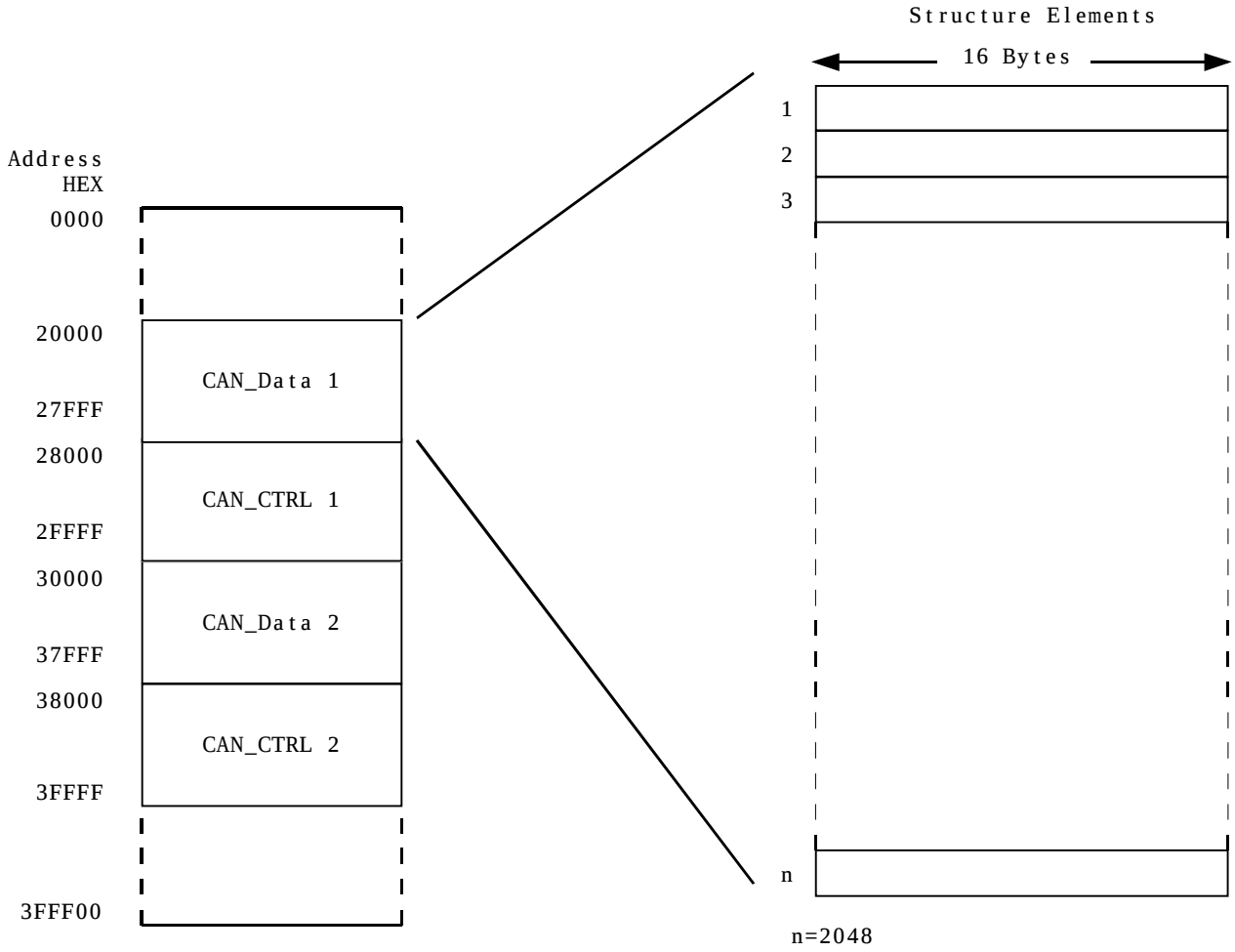


Fig. 4.1.1: Structure of a CAN_Data_Structure_Field (example: CAN-data 1, CAN-data 2)

A structure element is assigned to each possible CAN identifier of a CAN channel (max. 2048 Ids are possible).

A structure element is 16 bytes long and contains 8 bytes monitoring and control parameters apart from the data transmitted (max. 8 bytes).

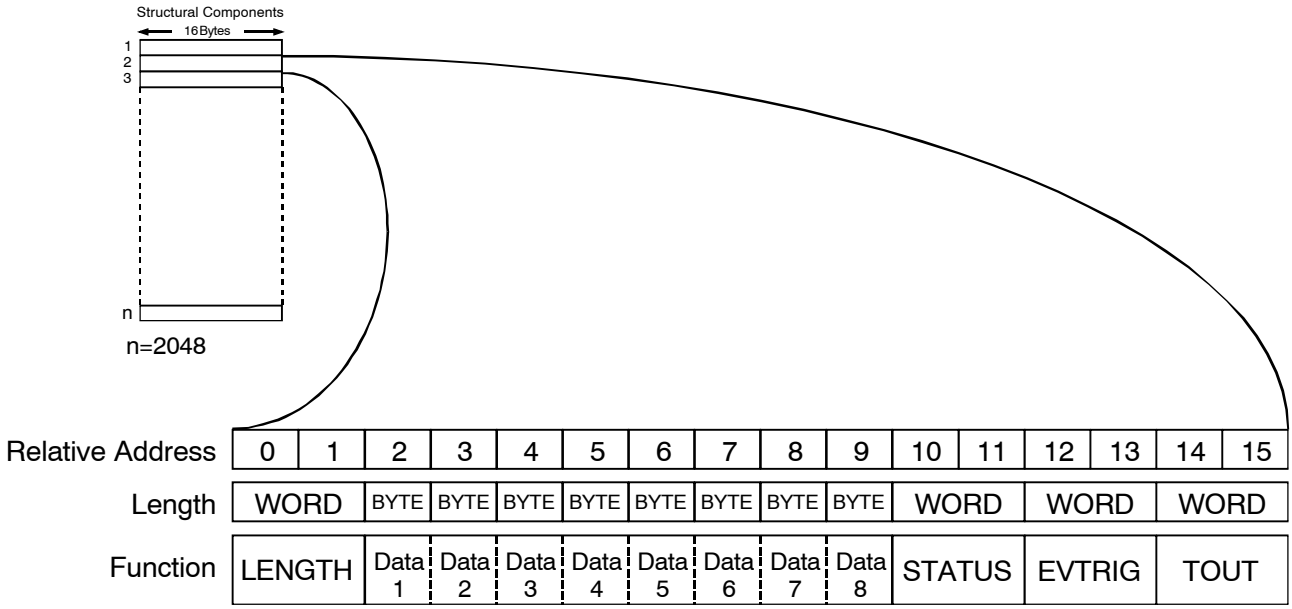


Fig. 4.1.2: Contents of a CAN_Data_Structure_Element

4.1.2 Bytes of a Data Structure Element

Below, the bytes of a data structure element is described in ascending order.

LENGTH... shows the number of valid data bytes and by means of cell 'MODE' determines the operating mode for this identifier in the according control-structure element (see chapter 'Parameter und Command Specification via Parameter Buffer').

Possible values [HEX]:
0000...0008 (only number of valid bytes)
0020...0028 (LENGTH in mode RTR-transmit)
0040...0048 (LENGTH in mode Rx-transfer with time out)
0060...0068 (number of valid bytes and starting Tx-transfer)

Data1..Data8... contain the data transmitted.
You can only transmit 0 to eight bytes.

STATUS... has information about the current status of transmission.
Values of STATUS are generally:

STATUS = 0	everything OK
STATUS < 0	some kind of 'Busy', data invalid, transfer not finished
0 < STATUS <$100	crucial error during transfer
STATUS ≥ $100	special status messages: RTR, ...

Especially with values under 0 you must not request via 'EQUAL $FFxx' in program sequence, but only via negative values to make sure that new error-status values, which still have to be defined, will be recognized!

STATUS messages $0013...$0019 trigger the transmission of an Abort-Domain-Frame.

The following message have been implemented so far:

STATUS [HEX]	Label of local firmware	Note
FFFF	ERMBSY	Tx is busy, active (transmission not finished)
FFFE	ERMWRX	waiting for Rx after remote request
FFFD	ERMTXQ	Tx is busy, in queue
FFFC	ERMBS1	Tx is busy, from queue to active
FFFB	ERMTLK	Tx is busy by node link
0000		OK, transmission successfully done
0001	ERMRTO	Rx-timeout
0002	ERMTTO	Tx-timeout of active element
0003	ERMres	released due to 'off-bus' (similar to $005)
0004	ERMTER	Tx-error
0005	ERMOFB	controller 'off-bus'
0006	ERMBUS	controller 'busy'
0007		release due to INIBIT
0008	ERMTUL	Tx-timeout of element in Tx-chain
0010		CMS: domain timeout
0013	ERMbfe	forced error (VME sets bfcmmd=0, while domain -stat≠0)
0014	ERMcd1	client: wrong ack at 'init_download'
0015	ERMcd2	client: wrong ack at 'download_segment'
0016	ERMcu1	client: wrong ack at 'init_upload'
0017	ERMcu2	client: wrong ack at 'upload_segment'
0018	ERMton	client/server: Rx-timeout
0019	ERMsws	server: 'wrong_state'
0020	ERMWST	CMS: compare error of CMS-watchdog
001C	ERMtto	Tx-timeout on 'domain_transmit'
001D	ERMabx	'abort_domain' received
001E	ERMwst	no server or client active
001F	ERMcmd	wrong VME-command

STATUS [HEX]	Label of local firmware	Note
0101 0104	RTRRX ERMRTR	RTR-frame received error on transmit RTR

Table 4.1.1: Meaning of values of STATUS word

Description of status messages:

Rx-timeout... (error)

No data has been received in this structure element within the time specified in cell ‘TOUT’.

Time-out monitoring is activated after entering the according operating mode into the CTRL structure element of the identifier (enter ‘MODE’ via parameter buffer) by triggering cell ‘LENGTH’ of the CTRL structure element. After triggering the value specified in cell ‘TOUT’ is entered into the ‘TIME COUNTER’ of the CTRL element, the element is integrated into the ‘Rx-Time Out chain’ and the ‘TIME COUNTER’ is count down.

After the time out has expired this error status is always set - FIFO ‘Data To VME’ can be loaded by a pointer to the structure element and an interrupt can be triggered (see chapter ‘Interrupts’).

Tx-timeout... (error)

The structure element could not be transmitted within the time specified in cell ‘TOUT’. The transmission attempt is aborted.

Time-out monitoring is activated after entering the according operating mode into the CTRL structure element of the identifier (enter ‘MODE’ via parameter buffer) by triggering cell ‘LENGTH’ of the CTRL structure element. After triggering the value specified in cell ‘TOUT’ is entered into the ‘TIME COUNTER’ of the CTRL element, the element is integrated into the ‘Tx-Time Out chain’ and the ‘TIME COUNTER’ is count down.

After the time out has expired this error status is always set - FIFO ‘Data To VME’ can be loaded by a pointer to the structure element and an interrupt can be triggered (see chapter ‘Interrupts’).

Tx-error...

This error applies, if CAN controller 82C200 requests new data for transmission from the firmware via its Tx-interrupt, even though it had not yet set its ‘Tx-complete’ signal. The error message can trigger an entry into FIFO ‘Data To VME’, which can trigger a VMEbus interrupt.

controller 'off bus'... (error)	If crucial errors occur during transmission attempts of CAN controller 82C200 (such as bit rate controller ≠ bit rate bus), the controller ‘exits’ the bus, because further transmission attempts were useless. If this error status occurs during active Tx-transmissions, this error message is not only entered into the current structure element, but also in all other Tx-structures already taken into the local priority chain. If the error occurs during the first transmission of a Tx-element, the message is only entered into this structure element, the transmission is aborted and the transmission of the following Tx-structure element is started. The error message can trigger an entry into FIFO ‘Data To VME’, which can trigger a VMEbus interrupt.
controller 'busy'... (error)	This error occurs, if the local Tx-priority chain is empty, a Tx-command is transmitted via VMEbus, and the firmware receives the message ‘Busy’ for the usual request of the controller of its Tx-status, even though the controller should not be busy at this point (-> crucial error!). The error message might trigger an entry into FIFO ‘Data to VME’, if cell ‘EVTRIG’ is set. This entry might trigger a VMEbus interrupt.
domain-timeout...	A time out error has occurred in CMS domain transfer.
compare error...	CMS watchdog has found an error while comparing the set point value with the actual value of toggle bit or status.
RTR-frame... received	An RTR-frame has been received for this structure element. This frame requests the transmission of data contained in this element. The status message only remains active as long as no transmission has been made.
waiting for Rx...	A ‘Remote Request’ has been transmitted on this identifier. The firmware now waits for the reception of a message for this structure element. The status is reset after a message has been received.
Tx is busy, active...	The transfer of this structure element has not yet been finished. After it has been finished, the status is reset and removed from the local priority chain.

.....

EVTRIG... determines, whether an entry into FIFO ‘Data to VME’ is to be made. This entry might trigger a user-VME interrupt. EVTRIG is only an enable status for triggering an interrupt. The interrupt has to be enabled by the hardware, i.e. the interrupt vector register must have been set (See hardware manual, chapter ‘Interrupt-Vector Register’ for register description).

If these enable statuses have been complied with, an interrupt is triggered on the VMEbus with the level ‘iolev’ and the vector specified via ‘iovec’, when the interrupt status described below occurs.

Value ‘EVTRIG’ [HEX]	Function
0	no FIFO entry after an end status has occurred
0001 : 00FF	FIFO entry and possible triggering of a user IRQ after end status has occurred

Table 4.1.2: Function of EVTRIG cell

Valid end statuses are successful transmissions or transmission which have been terminated by an error:

1. transmission successful
2. Rx-time out (error)
3. Tx-time out (")
4. Tx-error (")
5. Controller 'Off Bus' (")
6. Controller 'Busy' (error)

Errors in CMS transfer always trigger the entry of a pointer into the FIFO, regardless of the value of cell EVTRIG!

TOUT... specifies the time out value in msec (operation without CMS protocol) or the watchdog guard time (operation with CMS protocol) for values greater than 0 ($0001...$7FFF).

Operation without CMS protocol

Time out monitoring is activated after the according operation mode has been specified in the CTRL structure element of the identifier (enter 'MODE' via parameter buffer) by triggering cell 'LENGTH' of the CTRL structure element. After triggering the value specified in cell 'TOUT' is entered into 'TIME COUNTER' of the CTRL element, the element is integrated into a time out chain and 'TIME COUNTER' is count down.
The time out option can be used for Tx- as well as Rx-transfers:
In Tx-mode it triggers a Tx-time out error, if no transmission had been made within the time specified. The Tx-time out is shown in cell 'STATUS' and might trigger an interrupt. Furthermore it is possible to start Tx-transfers of up to 64 different structure elements in periodical intervals via the parameter buffer.
In Rx-mode an Rx-time out is shown in cell 'STATUS', if no messages have been received on this identifier within the time out time. 'TIME COUNTER' is reset with each reception of a message.

> If the parameter buffer cells mentioned above have not been set, the time out is disabled by negative TOUT values.
> This is to be strictly avoided, because the system might crash, if the Tx-command cannot be handled!

> 'TOUT' is never to be set to '0' and values smaller than 5 msec in any operating mode, because the resolution of the timer is within this range!

'TOUT' [HEX]	Function
0000...7FFF	depending on implementation: without CMS: time out in msec 0005 => 5 msec (*) 7FFF => 32.767 sec with CMS: watchdog guard time 0005 => 5 msec 7FFE =>32.767 sec

Table 4.1.3: Function of 'TOUT' cell

Operation with CMS protocol

If the CAN4 is equipped with CMS implementation, positive TOUT values specify the CMS watchdog. The CMS watchdog will be described in more detail in its own chapter.

4.2 Control Structures CAN_CTRL 1... CAN_CTRL 4

4.2.1 Function and Structures

Control structures are used by the firmware to assign transmission parameters, store status information and variables and document the order of active data structure elements.

Each CAN_Data structure has a CAN-CTRL structure. A control structure element contains 16 bytes.

> The direct setting of parameters of control structures is mainly limited to the firmware. **The user is only permitted to set cell XTTID directly!**

Structure elements are structured as follows:

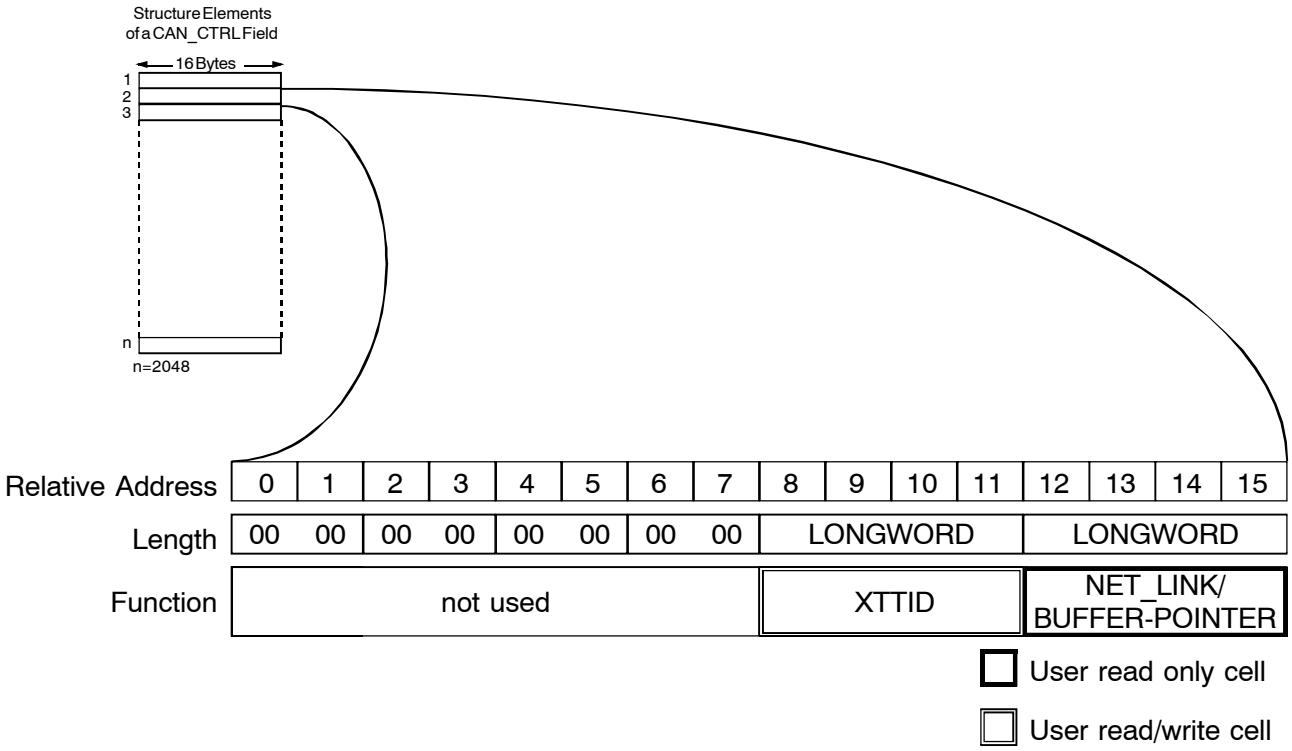


Fig. 4.2.1: Contents of a CAN_CTRL structure element

4.2.2 Bytes of a CTRL-Structure Element

Below, the bytes of a CTRL-structure element will be described in ascending order.

XTTID... This cell is not assigned by the local firmware. Users can store their own, freely selectable, parameters in four bytes. The cell can be used, for example, for storing the program counter of a user program into the cell.

In modes ‘Rx-transfers disabled (Tx-only)’, ‘Rx-transfer with time-out option’ and ‘Transmit RTR’, cell ‘XTTID’ can be used instead of cell ‘EVTRIG’ as required status for writing into FIFO ‘Data to VME’ and can therefore be used as interrupt status: If cells ‘iolev’ and ‘iovec’ of the ‘CARD-Interrupt Enable’ command are set in the parameter buffer, and cell ‘XTTID’≠ 0, a VMEbus interrupt is triggered as soon as a valid end status occurs.

NET_LINK/
BUFFER-POINTER...

If the identifier of this control structure element is used for CMS-transfer, the initial address of the CMS-buffer can be read in this cell after the buffer has been initialized by ‘Init Domain’ (see also command specification). By means of this initial address the addresses of relevant buffer cells can be determined.

5. Special Functions

5.1 Rx-Buffer

5.1.1 Function

Data of selected Rx-identifiers can be stored into Rx-buffers in order of their chronological reception.

If the command ‘Get Rx-Buffer’ is called via parameter channel, the Rx-buffer is installed by the local firmware. By means of command ‘Set Mode’ identifiers can be selected which are to be stored in this buffer. The commands of the parameter buffer will be explained in detail in a special chapter.

The data of identifiers selected for the Rx-buffer are not stored in the data-structure element, but in a memory area whose basis address will be returned as return parameter when executing the command.

	Rx-buffer
Buffer capacity	programmable (max. 4096 messages) (buffer is always overwritten again: ‘Wrap Around’)
Trigger for storing	no trigger, storing starts immediately after mode has been selected
Time stamp	1024 µs
VMEbus interrupt	VME-IRQ possible when first data arrive

Table 5.1.1: Summary of features of Rx-buffer

5.1.2 Structure of Rx-Buffer

The Rx-buffer can be divided into a header and a data area. The header contains the pointers and the number of valid elements in the buffer.

The data area contains, apart from received data, further parameters such as identifiers and network No..

Relative address [HEX]	Length	Access	Cell contents
0	WORD	read only	Rx-Write-Pointer (WRP)
2	WORD	read/write	Rx-Read-Pointer (RDP)
4	WORD	read only	Rx-Mask (number of elements contained)
6 : E	-	-	reserved
10 (BuffStart) : ??	is programmed via command 'Get Rx-Buffer' in parameter buffer	read only	data area

Table 5.1.2: Structure of Rx-buffer

Rx-Write-Pointer... This pointer points to the buffer cell (relative to 'BuffStart') which will be overwritten after the next RxId has been received.
Example for counting mode of pointers:
$0000, $0010, $0020,... ..., $0FF0, $0000
After a RESET the pointer is set to '0'.

Rx-Read-Pointer... The Read-Pointer points to the next cell which will be read out. After a RESET the pointer is set to '0'.

The contents of Rx-Write-Pointer and Rx-Read-Pointer are used as status for writing FIFO 'Data To VME' and therefore as status for triggering a VMEbus interrupt as well: After new data has been received in status 'Rx-Read-Pointer=Rx-Write-Pointer', the FIFO will be set.

Rx-Mask... This parameter contains the capacity of the data area of the Rx-buffer. Rx-Mask is determined from value 'Buffer Length', which can be specified in the parameter buffer by the user during buffer initialization (see also description of parameter buffer): The specified value is rounded by the firmware to values of 2^n and is entered into Rx-Mask (this means 2, 4, 8, 16, 32...).

Data area... In the data area the data which has been received is stored. Each Rx-identifier can use a line with 16 bytes to store status information and data.

The following figure represents the structure of a line:

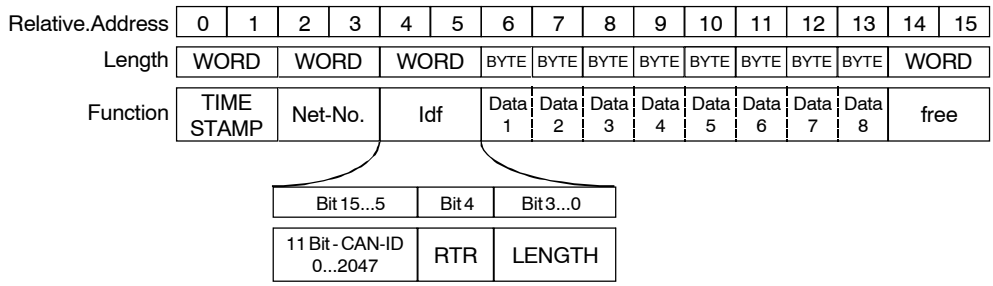


Fig. 5.1.1: Structure of the data area of an identifier

TIME STAMP... time (in [1024µs] steps) which has passed since the last system start (RESET or power-on).

Net-No. ... number of CAN network on which the RxId has been received (local CAN interface 0, 1, 2, 3).

Idf ... contains the 11-bit long CAN-Id, the RTR-bit (Remote Transmission Request) and the number of bytes transmitted (LENGTH):

CAN-Id...	possible values: 0...2047
RTR-Bit...	indicates, whether a data transfer has been requested under this identifier
RTR = '0'...	data transfer
RTR = '1'...	remote transmission request
LENGTH...	indicates the number of data bytes transmitted. Possible values: 0...8

Data1..Data8... contain the data received.

6. The Parameter Buffer

6.1 Function and Structure

By means of the parameter buffer various commands and parameters of CAN interfaces are specified. Each CAN channel has got its own parameter buffer.

The following figure represents the structure of the parameter buffer and the parameters relevant to the user.

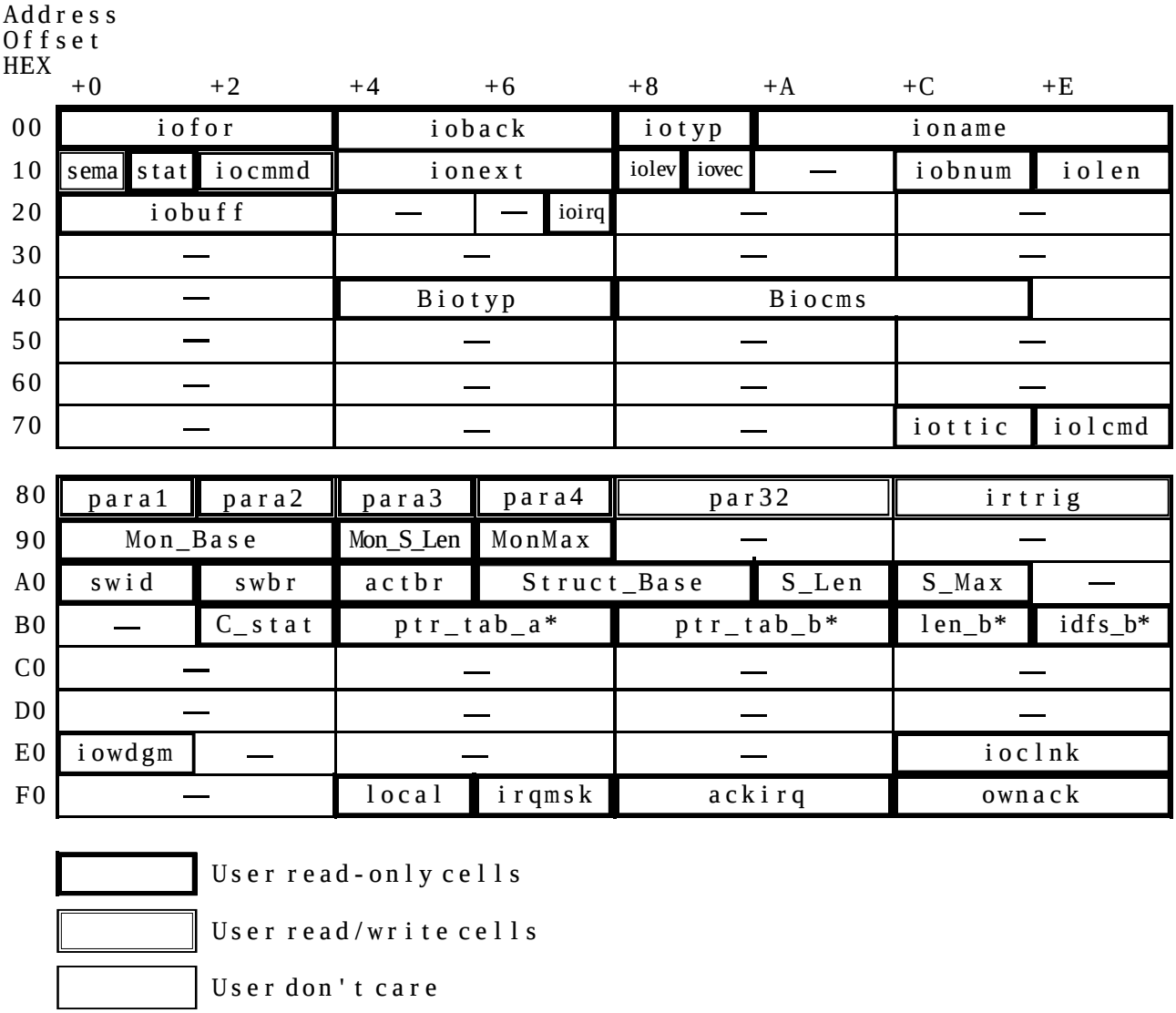
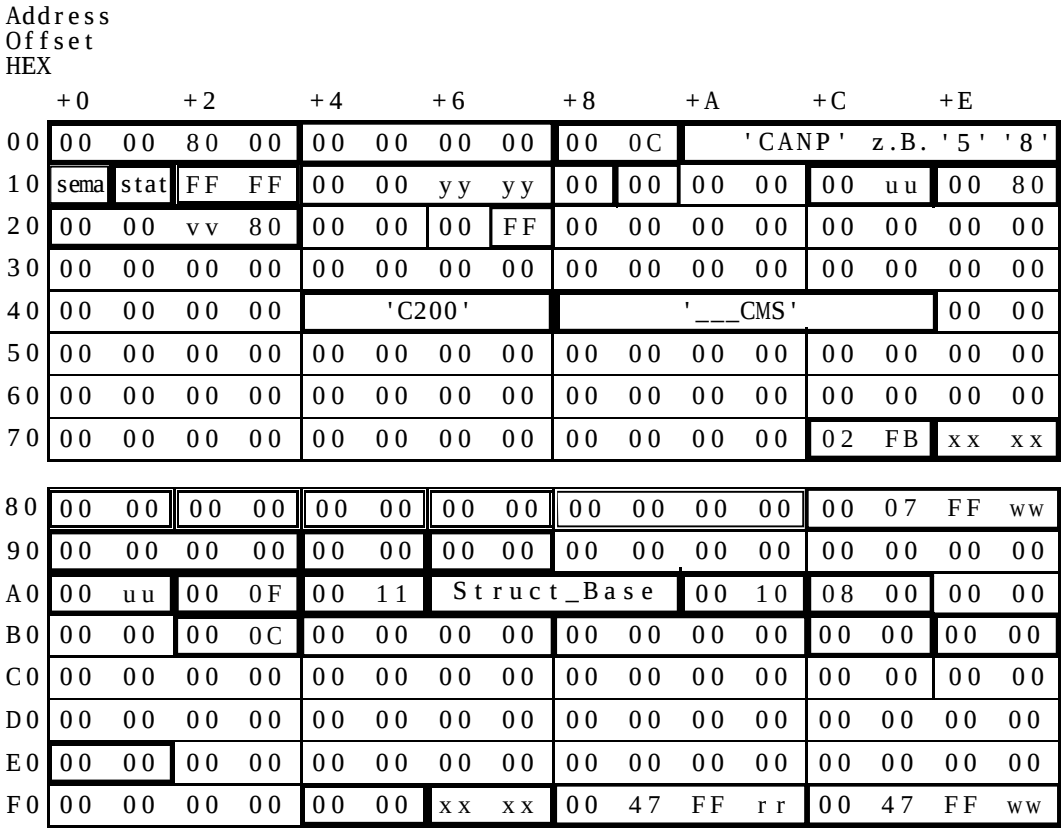


Fig. 6.1.1: Structure of parameter buffer

The following figure shows the parameter buffer with its default settings after a RESET on CAN4.



CAN channel	Default setting of channel-dependent variables (as at 19.03.97)					
	rr	uu	vv	ww	xx xx	yy yy
1	80	00	01	00	81 01	01 00
2	90	01	02	08	82 02	02 00
3	A0	02	03	10	84 04	03 00
4	B0	03	04	18	88 08	00 00

Fig. 6.1.2: Default setting of parameter buffer after a RESET

6.2 Specifying Parameters and Commands

Parameters and commands of a VMEbus MASTER-CPU of CAN4 have to be set in the following order:

1. - Evaluating cell ‘sema’--> is parameter buffer available?
   (via TAS iosema (multiprocessor) or BSET=7, iosema)
   - If available, assigning parameter buffer by setting cell ‘sema’ (‘TAS’ command)

2. - Setting parameters and commands by writing into desired cells.

3. - Transmitting parameters and commands to CAN4 slave server by triggering command IRQ (pointer in ‘irtrig’)

3. - Waiting for end of command processing: 'Cmmd' = FFFF

5. - Evaluating cell ‘stat’:
   00 --> OK
   FF --> ERROR

6. - Enabling cell ‘sema’.

6.3 Description of Parameters

Parameters are divided into ‘only readable’ and ‘writeable and readable’ parameters. In the figures above the different parameter types have been marked by different frames.

Apart from configuring the CAN4, the parameter buffer also transmits commands. The transmission of commands is explained in detail in chapter ‘Transmission of Commands via Parameter Channel’).

The parameters will be describes as follows:

parameter...	(relative local address, length of parameter) explanations about parameter

6.3.1 Writeable and Readable Parameters

sema...	($10, byte) assigned with semaphore: bit 7 - semaphore : ‘0' -> channel is available ‘1' -> channel is not available bit 6..1 - reserved bit 0 - at ‘1' -> triggering interrupt ‘CMD_to_VME’
iocmmd...	($12, word) is only required for transmission of commands (see the following chapter ‘Commands of Parameter Buffer’). Default value after RESET: $FFFF (= ‘last command has been handled’)
para1... to para4...	($80, word) ($86, word) contain the parameters required for transmission of commands (see the following chapter ‘Commands of Parameter Buffer’). Default value after RESET: $0000
par32...	($88, long word) contains return parameters after commands have been executed. Return parameters depend on the command selected. Exception: In some commands par32 is also used to set parameters by the user.
irtrig...	($8C, long word) Here the local CPU enters the VMEbus interrupt trigger address relative to the absolute board address. The user can/should then enter the absolute address here. After the address has been entered this cell is not required by the local software anymore.

6.3.2 Only Readable Parameters

iofor, ioback... ($00, $04, long words) internal management pointers

iotyp... ($08, word) buffer identifier (always $000C)

ioname... ($0A, 6 byte ASCII) contains buffer identifier as ASCII string and is serially numbered (always ‘CANP’).

stat... ($11, byte) contains information about the correct transmission of a command (see also the following chapter ‘Commands of Parameter Buffer’).

'stat' = $00 --> no error
'stat' P $00 --> error
'stat' = $FF --> Cmmd-Error
'stat' = $FE --> Para-Error

Commands should generally be transmitted in the following order:
1. assign ‘sema’
2. transmit parameters and commands
3. trigger CMD-IRQ
3. wait for ‘Cmmd’ = FFFF
or enable interrupt (iosema_0=’1')
5. request ‘stat’:
= 00 --> OK
P 00 --> ERROR
6. enable ‘sema’

ionext... ($14, long word) contains offset address of the parameter buffer and the following CAN channel. ionext of the last buffer points to the first buffer again.

iolev... ($18, byte) is not assigned by CAN4 firmware

iovec... ($19, byte) is not assigned by CAN4 firmware

iobnum... ($1C, word) contains buffer No.
CAN channel 1 -> 0
CAN channel 2 -> 1
CAN channel 3 -> 2
CAN channel 4 -> 3

iolen... ($1E, word) shows length of the data area of the parameter buffer.
Default: 'iolen' = $80 == 128 bytes.

iobuff...	($20, long word) pointer to the data area assigned to the buffer. In default setting ‘iobuff’ points to cell ‘para 1'.
ioirq...	($27, byte) = ‘00' -> ‘Cmd_to_VME_IRQ’ is NOT supported ≠ ‘00' -> ‘Cmd_to_VME_IRQ’ is supported
Biotyp...	($44, 4 bytes) returns the type of CAN controller used in ASCII code: ‘C200' -> 82C200
Biocms...	($48, 6 bytes) shows in ASCII code, whether the CMS protocol is implemented on board: ‘___CMS’ -> CMS protocol implemented
iottic....	($7C, word) determines resolution interval of time stamp. The cell contains the smallest unit of time in ns. Default: ‘timetic’ = $02FB (DEC 763 ns) -> hardware/CPU-clock-dependent = $0000 -> not supplied = = 4000 nsec (CAN2)
iolcmd....	($7E, word) informs the programmer and contains the command which has been handled last.
Mon_Base....	($90, long word) contains the local address of the monitor buffer. Default: ‘Mon_Base’ - $0 (no monitor)
Mon_S_Len....	($94, word) contains the length of a monitor structure element. Default: ‘Mon_S_Len’ - $0
MonMax...	($96, word) contains the maximum number of structure elements in the monitor buffer. Default: ‘MonMax’ - $0
swid....	($A0, word)
swbr...	($A2, word) always $0F
actbr...	($A4, word) contains the topical bit rate index. The bit rate index can be modified via a command entry (see also chapter ‘Transmission of Commands via Parameter Channel’)

The following table shows the values of the bit rate depending on the register contents (bit rate index) of the controller, and each physically maximum attainable line length.

Index [HEX]	Bit rate [kbit/s]	actbr 82C200 or SJA1000 register		typical values of attainable line length $_{lax}$ [m]	minimum attainable line length $_{Lein}$ [m]
		BTR0 [HEX]	BTR1 [HEX]		
0	1000	00	14	37	20
1	666.6	00	18	80	65
2	500	00	1C	130	110
3	333.3	01	18	180	160
4	250	01	1C	270	250
5	166	02	1C	420	400
6	125	03	1C	570	550
7	100	04	1C	710	700
8	66.6	45	2F	1000	980
9	50	09	1C	1400	1400
A	33.3	4B	2F	2000	2000
B	20	18	1C	3600	3600
C	12.5	5F	2F	5400	5400
D	10	31	1C	7300	7300
E	800	00	16	59	42

The specifications in the table are based on limit values of the bit timing of the CAN protocol, the delays of the local CAN interface and the delays of the cable. The delay of the cable is assumed at about 5.5 ns/m. Further influences, such as by terminating impedances, resistivity, cable geometry or external disturbances have not been considered in these specifications!

Table 6.3.1: Setting of bit rate

Struct_Base...	($A6, long word) contains local basis address of CAN data structure (CAN_Data). The basis address of the control structure (CAN_CTRL) is always 'Struct_Base'+ $8000.
S_Len...	($AA, word) contains the length of a data structure element: Default: 'S_Len' = $0010 (16 bytes)
S_Max...	($AC, word) contains the number of data structure elements of the data structure field (CAN_Data) Default: 'S_Max' = $0800 (DEC 2048)
C_stat...	($B2, word) contains status of CAN controller components 82C200 or SJA1000 in bit 0...7, according to the corresponding manual.
ptr_tab_a*...	($B4, long word) reserved for future applications
ptr_tab_b*...	($B8, long word) reserved for future applications
len_b*...	($BC, word) reserved for future applications
idfs_b*...	($BE, word) reserved for future applications
iowdgm...	($E0, word) CAN4 special: Mode 'node_guarding' when VME watchdog fails: $0000 - (default) - ignore watchdog $0001 - abort 'node_guarding'
ioclnk...	($EC, long word) local link pointer for clock server
local...	($F4, word) is only assigned by firmware
irqmsk...	($F6, word) reserved for future applications
ackirq...	($F8, long word) contains local address for triggering 'Cmd_to_VME_IRQ'
ownack...	($FC, long word) contains local address for resetting 'Cmd_IRQ' (points to same cell as 'irtrig')

6.4 Commands of Parameter Buffer

6.4.1 Overview of Implemented Commands

The sequence of command transmissions has already been explained in chapter ‘Specifying Parameters and Commands’.

Cell ‘s t a t’ in the parameter buffer acknowledges the transmission of commands:

‘s t a t’	=	$00	-->	no error in command transmission
‘s t a t’	P	$00	-->	error in transmission
‘s t a t’	=	$FF	-->	Cmmd-Error
‘s t a t’	=	$FE	-->	Para-Error

The desired command is entered in cell ‘i o c m m d’. Cells ‘p a r a 1' to ‘p a r a 4' and ‘ p a r 32' store the parameters required for the according command.

Cells of parameter buffer							Command
iocmmd [HEX]	Program Label	para1	para2	para3	para4	par32	
0 or 1 (equal)	svini0	bit rate $0000.. $000F or $0011.. $7F7F	-	-	-	-	init CAN net 1 with bit rate
(4) or (5) (equal)	svmon0	-	M_code1	M_mask1	-	-	trigger monitor 1 or monitor 2 **(without function at the moment)**
6	nodlk0	Net_A 0,1	Idf_A 0...2047	Net_B 0,1	Idf_B 0...2047	-	link Net_A/Idf_A to Net_B/Idf_B **(not to be used for new applications!)**
7	nodlk1	Net_A 0,1	Idf_A 0...2047	Net_B 0,1	Idf_B 0...2047	-	unlink Net_A/Idf_A from Net_B/Idf_B **(not to be used for new applications!)**
8	autlk0	-	Idf 0...2047	-	-	-	activate periodical Tx-transfers (or CMS Watchdog TxId)
9	autlk1	-	Idf 0...2047	-	-	-	stop periodical repeat of periodical Tx-transfers (or CMS RTR)
(A)	iniirq	iolev $0001... $0007	iovec $0000..$00FC	-	-	-	CARD interrupt enable **(not suitable for new applications - command triggers error message)**
B	svirq	0	Idf_start 0...2047	Idf_end 0...2047	MODE	-	set MODE
C	domwsp	-	client TxId 0...2047	client RxId 0...2047	buffer length 1...32 kbytes	*returns* buffer address	init domain (only CMS transfers)
D	domrbf	-	client TxId 0...2047	client RxId 0...2047	-	*returns* buffer address	release CMS buffer
E	grxbuf	handle-no.	buffer length	-	-	*returns* buffer address	get Rx buffer
F	rrxbuf	handle-no.	-	-	-	-	release Rx buffer
-10	idfins	-	handle 0...2047	ext_mode 0,1	dyn_mode 0,1,2,3	idf 29/11 0...2947 or $0...2^{29}-1$	only if 29-bit identifier required: Insert Identifier **(is not yet being supported)**
11	idfdel	-	handle 0...2047	-	-	-	only if 29-bit identifier required: Release Identifier **(is not yet being supported)**
12	gdlbuf	buffer length	-	-	-	*returns* download buffer address	get code download buffer **(is not yet being supported)**
13	coddld	-	-	-	-	*set* buffer address	copy downloaded program in Flash-EPROM and RESET board **(is not yet being supported)**
14	gtxbuf	handle-no.	buffer length	-	-	*returns* buffer address	get Tx buffer
15	rtxbuf	handle-no.	-	-	-	-	release Tx buffer

Cells of parameter buffer							Command
iocmmd [HEX]	Program Label	para1	para2	para3	para4	par32	
16	ndlink	idfnet_1	idfnet_2	idfnet_3	idfnet_4	-	4-channel net link
17	ndulnk	idfnet_1	idfnet_2	idfnet_3	idfnet_4	-	4-channel net unlink
FFFF		-	-	-	-	-	'last command done' (Default setting after RESET)

Parameter values marked by '-' are not evaluated.

Table 6.4.1: General commands of parameter buffer

6.4.2 Description of Commands

In this chapter the individual commands and the corresponding parameters will be described in ascending order.

6.4.2.1 Setting the Bit Rate (iocmmd = 0)

The bit rate of the CAN channel is set again. A table with the assignment of parameter ‘bit rate’ to the bit rate can be found under the description of buffer cell ‘a c t b r’.

The bit rate is specified by directly entering it into the 16-bit wide register BTR0 + BTR1 of the CAN controller.

In active (unfinished) transmissions changing the bit rate would probably cause errors. Therefore, the handling of all structure elements which are in waiting status is aborted with the according error message.

6.4.2.2 Triggering the Monitor Function (iocmmd = 4)

This function is not being supported at the moment.

6.4.2.3 Linking CAN Identifiers of Different Networks (iocmmd = 6, 7)

This command is not to be used for new applications! It is only supported for reasons of compatibility to VME-CAN2.

By means of this command the CAN-Net_A-Identifier Idf_A is bidirectionally linked to CAN-Net_B (Idf_B) (or the link is removed again). This means that all data with identifier ‘A’ received in Net_A are transmitted to Net_B with Idf. ‘B’, and all data received in Net_B are transmitted to Net_A.

For reasons of compatibility to VME-CAN2 the networks can only be linked between channels 1 and 2 or 3 and 4.

Parameters ‘Net_A’ = 0/1 and
‘Net_B’ = 0/1

channels 1, 3 -->'0'
channels 2, 4 -->'1'

In addition to simply pass on CAN frames from one network to another it is possible to transmit and receive message in the networks. For this the two identifiers of the networks are seen as one identifier, that means if, e.g., a message is transmitted via identifier A/Net_A, it is simultaneously also transmitted on identifier B/Net_B.

6.4.2.4 Periodical Transmission of Tx-Messages (iocmmd = 8, 9)

This command repeats Tx-transfers periodically or resets this mode when using standard EPROMs without CMS-implementation.

CMS-implementation
If CMS has been implemented into the EPROMs used, the function ‘Periodical Transmission’ does not apply. Instead, CMS-watchdog Tx-identifier and network number are transmitted via this command. The watchdog identifier is specified in cell ‘para2' and the network number in cell ‘para1'.

Initialization
This mode is initialized by transmitting parameters ‘iocmmd’, ‘Net’ and ‘Idf’ to the parameter buffer.

Activation
The operating mode becomes active immediately after initialization.

The period time of Tx-frames on the CAN, as specified in ‘TOUT’, might be incorrect in the range of ±4 ms.

End
The periodical Tx-transfers are ended by resetting the command in the parameter buffer or setting the time out to negative values.

6.4.2.5 Select Operating Mode (MODE) (iocmmd = $B)

By means of this command operating modes can be selected for identifier groups or individual identifiers. If the operating mode is to be selected for only one identifier, the same value has to be entered in cells ‘Idf_start’ and ‘Idf_end’.

The command ‘Set Mode’ is called via the parameter buffer by entering the value $B in cell iocmmd. Cells para1...para4 have to be set as in the previous chapter ‘Overview of Commands Implemented’. In cell para4 the value for MODE is entered.

In the operating modes selected various transfers can be made via the identifiers. Sometimes further action is required, as for instance setting cell ‘length’ in the assigned data structure element when transmitting. Chapter 7 describes the correct way of executing all possible transfers.

The following table gives an overview of the possible transfers in the operating modes selected:

Entry for MODE	Rx-transfers possible	Tx-transfers possible	Automatic Tx on RTR possible	Notes
0	no	yes	no	basic Rx/Tx-transfers
1	yes	yes	no	
2	yes	yes	yes	
4	(yes)	no	no	special case: monitor MODE **(not yet implemented)**
5	(yes)	no	no	special case: CAN serial MODE **(not yet implemented)**
000F	This entry resets the identifier/s into the previous operating MODE (no operating MODE).			
80xy	?	?	?	special case: handle assignment for Rx-buffer **(not yet implemented)**
81xy	?	?	?	special case: handle assignment for Tx-buffer **(not yet implemented)**

Table 6.4.2: Overview of implemented modes

6.4.2.6 CMS-Initialization (iocmmd = $C, $D)

For the upload or download via CMS-protocol the local firmware requires a memory area (CMS-buffer) in order to be able to transmit parameters and store data.

By means of command ‘init domain’ the CMS-buffer is installed and the Rx- and Tx-identifiers which have been selected for transfers are specified.
The buffer length specified regards to the length of the CMS-data buffer which is to be made available. Values of up to 32kbytes can be selected for the length.

After the command has been executed, the absolute local initial address of the CMS-buffer can be read in parameter ‘par32'.

If the CMS-buffer is not required anymore, the memory area can be released again (iocmmd = $D). Only the identifiers and the network number are transmitted as parameters.

6.4.2.7 Rx-Buffer (iocmmd = $E, $F)

Only when using the Rx-buffer functionality:
Before assigning the identifiers you have to execute command ‘Get Rx-buffer’. The local firmware then installs an Rx-buffer. Each Rx-buffer receives a so-called handle No.when the command ‘Get Rx-buffer’ is executed.

Then the identifiers are selected which are to be stored in the buffer. Identifiers and buffers are assigned by means of the handle numbers: handle numbers can be assigned to individual identifiers or identifier blocks by means of the MODE assignment.

Identifiers with the same handle No. are therefore stored into the same Rx-buffer.

The identifiers are stored in the Rx-buffer by executing this command. After the command has been executed the absolute initial address of the Rx-buffer can be read in parameter ‘par32'.

By means of command ‘Release Rx-buffer’ (iocmmd = $F) the memory area is released again.

6.4.2.8 Insert/Release Identifier (only for 29-bit Identifiers) (iocmmd = $10, $11)

The 29-bit identifier assignment is not yet being supported.

6.4.2.9 Handle Code Download Buffer (iocmmd = $12, $13)

This function is used to load program updates. It is not yet being supported.

6.4.2.10 Tx-Buffer (iocmmd = $14, $15)

The documentation of the Tx-buffer has yet to be worked out.

6.4.2.11 Net Link Function for up to Four Networks (iocmmd = $16, $17)

By means of this command identifiers of different networks can be linked. If two identifiers are to be linked in two different CAN networks, they have to entered in the parameter buffers of *both* networks. The same applies for four networks. The parameter structure is as follows:

Parameters of parameter buffer:	para1	para2	para3	para4
Net link identifiers:	idfnet_1	idfnet_1	idfnet_1	idfnet_1
Link with CAN channel...	network 1	network 2	network 3	network 4

Examples:

1. Identifier A in CAN network 2 is to be linked bidirectionally to identifier B in CAN network 4.

Entry in parameter buffer of network 2:			
para1	para2	para3	para4
idfnet_1	**idfnet_2**	idfnet_3	idfnet_4
0	identifier A	0	identifier B

Entry in parameter buffer of network 4:			
para1	para2	para3	para4
idfnet_1	idfnet_2	idfnet_3	**idfnet_4**
0	identifier A	0	identifier B

2. Identifier A in CAN network 1 is to be linked bidirectionally to identifier B in CAN network 2, identifier C in CAN network 3 and identifier D in CAN network 4. The identifiers of networks 2, 3 and 4, however, are *not* to be linked with each other.

Entry in parameter buffer of network 1:			
para1	para2	para3	para4
idfnet_1	idfnet_2	idfnet_3	idfnet_4
identifier A	identifier B	identifier C	identifier D

Entry in parameter buffer of network 2:			
para1	para2	para3	para4
idfnet_1	**idfnet_2**	idfnet_3	idfnet_4
identifier A	identifier B	0	0

Entry in parameter buffer of network 3:			
para1	para2	para3	para4
idfnet_1	idfnet_2	**idfnet_3**	idfnet_4
identifier A	0	identifier C	0

Entry in parameter buffer of network 4:			
para1	para2	para3	para4
idfnet_1	idfnet_2	idfnet_3	**idfnet_4**
identifier A	0	0	identifier D

If an RTR were received on identifier D / network 4 in this constellation, the RTR is passed on to identifier A /network 1. Identifier A / network 1 then responds with a Tx-frame which is again transmitted via identifier D / network 4, but also via identifier B / network 2 and identifier C / network 3.

7. User System Clock

7.1 Overview

The user clock is a custom development for synchronizing several CAN4 boards. It is available as an option only. The CAN4 can either be operated as system clock master or as system clock slave. The cycle line can be connected via special signals on the CAN connector or via P2.

A detailed description of the system clock functionality can be taken from the hardware manual.

The system clock is configured via the VMEbus master.

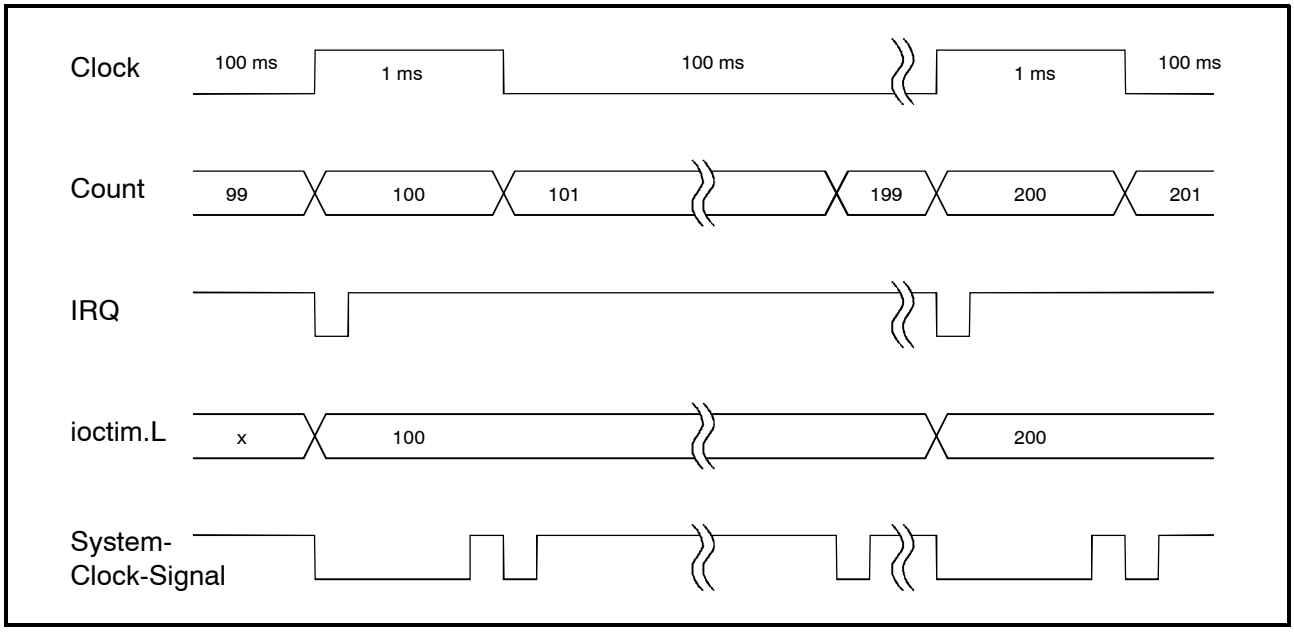


Fig. 7.1.1: Generating system clock

7.2 Configuring and Activating the System Clock

The system clock is configured and activated by means of the parameter buffer of the first CAN channel.

Cells of parameter buffer							Command
iocmmd [HEX]	Program Label	para1	para2	para3	para4	par32	
1F	-	clock_ MODE	count_mode	ref_mode	irq_mode	time	configure system clock

Table 7.2.1: System clock commands in the parameter buffer of the first CAN channel

Description of Command Parameters:

clk_mode... 0 - disable all (all other parameters will not be evaluated in this case)
1 - master MODE
2 - slave MODE

count_mode... 0 - count binary
1 - count msec/day

ref_mode... 0 - free run
1 - reference is local RTC (Real Time Clock)

irq_mode... 0 - no interrupt
1 - interrupt on 100 ms pulse

time... <0 - clear msec counter and start counting
≥ 0 - load counter and start synchronisation to reference

Status Cells:

The status cells are in the parameter buffer of the first CAN channel.

Name of cell	Address (relative to basis address of 1. parameter buffer - at the moment $01.000)	Access	Description
iocirs	+$E4	word	interrupt status, here: always ‘0’
iocmod	+$E6	word	interrupt MODE = parameter 4 of command, such as 001
ioctim	+$E8	long	here: msec count with 100 ms edge

Table 7.2.2: Status cells of system clock

CAN API for CAN2 / CAN4

VxWorks

Software Manual

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 205
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd-electronics.com
Internet: www.esd-electronics.com

USA / Canada
7667 W. Sample Road
Suite 127
Coral Springs, FL 33065
USA

Phone: +1-800-504-9856
Fax: +1-800-288-8235
E-mail: sales@esd-electronics.com

Manual file:	I:\texte\Doku\MANUALS\PROGRAM\CAN\Schicht2\ENGLISCH\VxWorks\CAN2VX11.en6
Date of print:	26.10.2000

Described software:	VME-CAN2/VME-CAN4 driver for VxWorks
Revision/date:	CAN-Interface V1.2x canTools (CAN2: V03x, CAN4: V01x) cmsTools (CAN2: V03x, CAN4: V01x)
Order number:	CAN2: P.1405.56, CAN4: P.1408.56

Changes in the software and/or documentation

Changes in this manual from previous version	Changes in the software	Changes in the documentation
First English version.	-	-

Contents

Page

1. Reference

/1/: electronic system design gmbh, CAL-Slave Manual, July 1998

/2/: electronic system design gmbh, CANopen-Slave Manual, March 1998

/3/: electronic system design gmbh, CANopen-Master Manual, August 2000

/4/: CiA DS-102, CAN Physical Layer for Industrial Applications, April 1994

/5/: CiA DS-201, CAN Reference Model, February 1996

/6/: CiA DS-202/1, CMS Service Specification, February 1996

/7/: CiA DS-202/2, CMS Protocol Specification, February 1996

/8/: CiA DS-203/3, CMS Data Types and Encoding Rules, February 1996

2. Introduction

In this document the C application programmers interface (API) for the driver of the CAN controller will be explained. The CAL-/CANopen-Masters /3/ respectively the CAL-/CANopen-Slaves /1/, /2/ from *esd gmbh* are based on this interface.

Concerning the OSI-layer model the driver contains functions from layer 2 which serve the blocking and non-blocking transmission and reception of data, and also functions from the layer-7-CMS-domain-transfer protocol of the *CAN-application layer* (CAL) /6/-/7/.

> The C-interface is implemented for several operating systems at several esd CAN boards. If there are limitations for the functions due to the operating system or the board type, this is marked in the text.

The prototypes of all functions and the *communication handles* are combined in the header file *can.h*.

3. Program Interface

This chapter describes the C-API for access to the CAN controller hardware. The meaning of returned values in case of error is described in chapter 4.

3.1 Initialization

The service described below serves as initialization of the CAN-controller hardware and has to be executed before every further function call.

---

canCalSetBaudrate()

Name: **canCalSetBaudrate()** - initializing CAN interface

Call:

```
int    canCalSetBaudrate
       (
       int          net,        /* number of CAN interface */
       int          baud        /* bit rate */
       )
```

Description: This function initializes the CAN interface and sets the bit rate for net *net* to the bit rate *baud* corresponding to following table. The number of the supported nets depends on the implementation and is described in the appendix.

baud [HEX]	Bit rate [kbit/s]
0	1000
1	6666
2	500
3	3333
4	250
5	166
6	125
7	100
8	666
9	50
A	333
B	20
C	125
D	10

Return: 0 or an error code described at page 15.

3.2 Reading and Writing of Data

The API calls described below serve the reading and writing of data on the CAN. Each function exists in a non-blocking and a blocking variant with timeout default. Data can be requested by giving out an RTR frame on an Rx identifier and it is possible to wait for the reception of an RTR frame by means of a Tx identifier.

---

canCalOpen()

Name: **canCalOpen()** - generating a handle for read/write operations

Call:

```
HCAN    canCalOpen
        (
        int             net,    /* number of CAN interface */
        int             txid,   /* Tx identifier of handle */
        int             rxid,   /* Rx identifier of handle */
        unsigned short  typ,    /* properties */
        int             txtout, /* timeout of Tx identifier */
        int             rxtout  /* timeout of Rx identifier */
        )
```

Description: This function returns a handle for following I/O requests.

Net transmits the net number if several nets are available.

txid and *rxid* are the CAN identifiers in the range of 0-2047 assigned to this handle. For writing API calls the Tx identifier is used and for reading API calls the RX identifier of the handle is used. If only one Tx identifier or one Rx identifier is to be assigned to the handle, the unused identifier has to be set to a negative value.

The table below shows additional Tx- and/or Rx-identifier properties which are determined by means of parameter *typ*. Moreover, some drivers support additional options which are described in the appendix.

Parameter	Description
TXRX_DOMAIN	This identifier couple is to be used for the CMS-domain transfer described in section 3.4.
TX_RECV_RTR	An RTR frame can be received on the Tx identifier.
TX_AUTOANSWER	After receiving an RTR frame on this Tx identifier, the actual data is automatically transmitted.
RX_SEND_RTR	An RTR frame can be transmitted on the Rx identifier.

txtout and *rxtout* determine the timeout interval in ms in blocking read/write operations.

Return: In case of error NULLHANDLE, otherwise a valid handle.

canCalClose()

Name: **canCalClose()** - closing a handle for read/write operations

call:

```
int     canCalClose
        (
        HCAN            hcan        /* CAN handle */
        )
```

Description: This function frees a CAN handle and all assigned resources.

Return: 0 or an error code described at page 15.

canCalUpdate()

Name: **canCalUpdate()** - updating the Tx data

call:

```
int     canCalUpdate
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to current data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function updates the internal Tx-data structure of the driver of the Tx identifier designated by *hcan*. For this matter *len* bytes are copied from memory address *buf*.

Return: 0 or an error code described at page 15.

canCalSend()

Name: **canCalSend()** - non-blocking transmission of a message

Call:

```
int     canCalSend
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to current data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function transmits *len* bytes from memory address *buf* on the Tx identifier of *hcan*. At the same time the internal Tx-data structure of the driver is updated. The successful termination of the transmission command is not being waited for.

Return: 0 or an error code described at page 15.

canCalWrite()

Name: **canCalWrite()** - blocking transmission of a message

call:

```
int     canCalWrite
        (
        HCAN            hcan,       /* CAN handle */
        const void *    buf,        /* pointer to transmission data */
        int             len         /* number of bytes (1..8) */
        )
```

Description: This function transmits *len* bytes from the memory address to the *buf* on the Tx identifier of *hcan*. At the same time the internal Tx data structure of the driver is updated. The function only returns after the transmission request has been successfully terminated or the Tx-timeout interval which had been specified when opening the handle has been exceeded.

Return: 0 or an error code described at page 15.

canCalWaitRTR()

Name: **canCalWaitRTR()** - waiting for the reception of an RTR frame

Call:

```
int     canCalWaitRTR
        (
        HCAN        hcan            /* CAN handle */
        )
```

Description: This function waits for the reception of an RTR frame on the Tx identifier of *hcan*. Flag `TX_RECV_RTR` has to be set when opening the handle. The function only returns after an RTR frame has been received or the Tx-timeout interval specified when opening the handle has been exceeded.

Return: 0 or an error code described at page 15.

---

canCalTake()

Name: **canCalTake()** - non-blocking reading of updated Rx data

Call:

```
int     canCalTake
        (
        HCAN        hcan,       /* CAN handle */
        void *      buf,        /* pointer to free memory area */
        int *       len         /* number of copied bytes (1..8) */
        )
```

Description: This function copies the last-valid Rx data of the Rx identifier from *hcan* to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*.

Return: 0 or an error code described at page 15.

canCalRead()

Name: **canCalRead()** - blocking reading of updated Rx data

Call:

```
int     canCalRead
        (
        HCAN        hcan,       /* CAN handle */
        void *      buf,        /* pointer to free memory area */
        int *       len         /* number of copied bytes (1..8) */
        )
```

Description: This function copies the updated data of the Rx identifier from *hcan* to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*. If the data in the Rx buffer of the driver are valid when the function is requested, the function returns immediately. Otherwise the function only returns, if the Rx timeout interval which has been specified when opening the handle has been exceeded or, if new data has been received within this interval.

Return: 0 or an error code described at page 15.

---

canCalSendRTR()

Name: **canCalSendRTR()** - non-blocking request of data

Call:

```
int     canCalSendRTR
        (
        HCAN        hcan        /* CAN handle */
        )
```

Description: This function requests data by transmitting an RTR frame on the Rx identifier of handle *hcan*. This is only possible, if the flag `RX_SEND_RTR` is set when opening the handle.

Return: 0 or an error code described at page 15.

canCalReadRem()

Name: **canCalReadRem()** - blocking request of data

Call:

```
int     canCalRead
        (
        HCAN            hcan,       /* CAN handle */
        void *          buf,        /* pointer to a free memory area */
        int *           len         /* number of copied bytes (1..8) */
        )
```

Description: This function requests data by transmitting an RTR frame on the Rx identifier of handle *hcan*. This is only possible, if flag `RX_SEND_RTR` is set when opening the handle. If valid data are available at the time of the request, these data are marked as invalid first. The function only returns after new data has been received or the timeout interval specified when opening the handle has been exceeded. Valid data are copied to the memory area specified by *buf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *len*.

Return: 0 or an error code described at page 15.

canCalWriteRead()

Name: **canCalWriteRead()** - succeeding write/read operations

Call:

```
int     canCalWrite
        (
        HCAN            hcan,       /* CAN handle */
        const void *    tbuf,       /* pointer to transmission data */
        int             tlen,       /* number of transmission bytes (1..8) */
        void *          rbuf,       /* pointer to memory for reception data */
        int *           len         /* number of received bytes (1..8) */
        )
```

Description: This function transmits *tlen* bytes of the transmission data to which *tbuf* points at the Tx identifier of handle *hcan*. At the same time the internal Tx-data structure of the driver is updated. Afterwards a reading request on the Rx identifier of *hcan* is executed. If valid Rx data are available at the time of the request, these are aborted first. The function only returns after new data has been received or the Rx-timeout interval specified when opening the handle has been exceeded. Valid data is copied to the memory area specified by means of *rbuf*. This memory area must have a size of at least 8 bytes. The number of copied bytes is stored in *rlen*.

Return: 0 or an error code described at page 15.

3.3 Simultaneous Reading of Several Rx Identifiers (*Multi read)*

By means of the services described below it is possible to wait for the reception of several Rx identifiers by linking all identifiers into a waiting queue. An identifier used this way cannot be used for an individual reading request anymore. The received data of all Rx identifiers to be waited for are stored into a message queue.

---

canCalMultiOpen()

Name: **canCalMultiOpen()** - generating a multi read handle

Call:

```
SCAN     canCalMultiOpen
         (
         int                net,          /* number of CAN interface */
         unsigned short     typ,          /* reserved */
         int                queusize,     /* size of waiting queue */
         int                timeout       /* timeout of reading order */
         )
```

Description: This function returns a handle for following multi read requests for net *net*.

Parameter *typ* is reserved at the moment and has to be set to zero.

queuesize determines the size of the waiting queue in buffered CAN telegrams. The greater this parameter is selected the more improbable is a data loss and the greater is the resource need of the driver.

timeout determines the timeout interval in ms in reading order.

Return: In case of error `NULLHANDLE`, otherwise a valid handle.

---

canCalMultiClose()

Name: **canCalMultiClose()** - closing a multi read handle

Call:

```
int    canCalMultiClose
       (
       SCAN               scan          /* multi read handle */
       )
```

Description: This function frees a multi read handle and all assigned resources. All Rx identifiers have to be removed from the waiting queue before.

Return: 0 or an error code described at page 15.

canCalMultiAdd()

Name: **canCalMultiAdd()** - adding an Rx identifier to a multi read queue

Call:

```
int     canCalMultiAdd
        (
        SCAN            scan,       /* multi read handle */
        int             idf         /* CAN identifier */
        )
```

Description: This function adds the Rx identifier *idf* to the waiting queue of the multi read handle *scan*.

Return: 0 or an error code described at page 15.

canCalMultiDelete()

Name: **canCalMultiDelete()** - deleting an Rx identifier from a multi read queue

Call:

```
int     canCalMultiDelete
        (
        SCAN            scan,       /* multi read handle */
        int             idf         /* CAN identifier */
        )
```

Description: This function deletes the Rx identifier *idf* from the waiting queue of the multi read handle *scan*.

Return: 0 or an error code described at page 15.

canCalMultiRead()

Name: **canCalMultiRead()** - waiting for the reception of Rx data

Call:

```
int    canCalMultiRead
       (
       SCAN        scan,      /* multi read handle */
       int *       idf,       /* received CAN identifier */
       int *       len,       /* number of received bytes */
       char *      data       /* pointer to the receive buffer  */
       )
```

Description: This function copies *len* Bytes received data from the message queue of the multi read handle *scan* to the memory area given by data. The according identifier is returned in *idf*. If no valid data are available, the function only returns after the reception of valid data or after exceeding the timeout interval specified when opening the multi read handle.

Return: 0 or an error code described at page 15.

3.4 Auxiliary Functions

The API requests described below are auxiliary functions which cannot be allocated to any of the previous sections.

---

canCalCopy()

Name: **canCalCopy()** - copying with possibly swapping the byte order

Call:

```
int    canCalCopy
       (
       const char *      src,      /* pointer to memory with source data */
       char *            dest,     /* pointer to memory for destination data */
       const char *      datatype  /* data type designator */
       )
```

Description: This function copies data from the memory area pointed to by *src* into the memory area pointed to by *dest*. At the same time the required byte- swapping is initiated in processors which manage data internally different as the byte order on the CAN required in /5/. For this the datatype designator is transmitted to the routine in *datatype*. This datatype designator consists of a zero-terminated string in which every component of the source data is represented by a single `CHAR` which is the length of the component in bytes. The total length of source data accordingly results from *datatype* and the memory area for source and destination data has to be accordingly sized.

Example:

Following structure is to be transmitted on the CAN:

```
struct myStruct
   {
   long      a;
   char      b;
   short     c;
   };
```

A datatype designation for this example would have to be:

```
const char myDatatype = {4, 1, 2, 0};
```

The number of copied data bytes in this example is 7 and the memory area for source and destination data has to be selected accordingly large.

Return: Always 0.

canCalGetVersion()

Name: **canCalGetVersion()** - determining driver version

Call:

```
int     canCalGetVersion
        (
        unsigned short *    version,    /* memory cell for version number */
        char *              name        /* memory cell for driver name */
        )
```

Description: By means of this function it is possible to determine the version number and driver name for the respective net. In *version* the net number has to be given.

The version number is returned in *version* as 16-bit value in which the individual bits have following meaning:

Bit 15...12	Bit 11...8	Bit 7...0
level	revision	modification

A textual description of the driver is copied to *name*. The buffer size made available for this by the application has to have at least 32 bytes. If `NULL` is given as argument, no copy operation is executed.

Return: 0 or an error code described at page 15.

4. Returned Values

All functions which are called for the request of a handle return NULLHANDLE in case of error.

A function with a returned value of type *int* returns an error code corresponding to following table in which not all codes are useful for all API functions.

Error	Description
CAN_OK	no error
CAN_NO_MEMORY	not enough memory for internal resource need
CAN_NO_DEVICE	CAN controller not available or not initialized
CAN_PARA_ERROR	invalid parameter in function request
CAN_SYS_ERROR	internal error
CAN_RX_TIMEOUT	timeout in reception
CAN_TX_TIMEOUT	timeout in transmission
CAN_TX_ERROR	error during transmission
CAN_CONTR_OFF_BUS	error on CAN. Controller not at bus anymore
CAN_CONTR_BUSY	access to CAN controller not possible
CAN_CONTR_WARN	error on CAN
CAN_OLD_DATA	the read data have already been read
CAN_NO_ID_ENABLED	multi read handle without CAN identifier
CAN_INVALID_HANDLE	the transmitted handle is invalid

5. Programming Interface for the CMS Domain Protocol

By means of the following calls the protocol described in /6/ - /8/ is realised. This can also be achieved by means of the I/O-operations described in 3.1, however, task changes will be reduced on driver level and performance will increase. The protocol distinguishes between a client, which initiates the upload and download of data, and a server, which waits for upload and download requests.

canCalClientDownload()

Name: **canCalClientDownload()** - initiating a downloading transfer

Call:

```
int   canCalClientDownload
      (
      HCAN          hcan,     /* CAN-handle */
      long          mux,      /* CMS multiplexor */
      const void *  buf,      /* pointer to transmit data */
      long          len       /* number of bytes to be transmitted */
      )
```

Description: This function initiates the transmission of *len* bytes of the data pointed at by *buf*, according to the CMS domain protocol with the multiplexor *mux* for the handle *hcan*. When opening the handle, `RXTX_DOMAIN` has to be set.

Return: 0 or an error code described in the appendix.

canCalClientUpload()

Name: **canCalClientUpload()** - initiating an uploading transfer

Call:

```
int   canCalClientUpload
      (
      HCAN          hcan,     /* CAN-handle */
      long          mux,      /* CMS-multiplexor */
      void *        buf,      /* pointer to buffer for receive data */
      long *        len       /* length of receive buffer */
      )
```

Description: This function requests the transmission of data downloaded by *mux* from a server in accordance with the domain protocol on handle *hcan*. When opening the handle, `RXTX_DOMAIN` has to be set. The received data is copied into the memory *buf*, whose length is determined in *len* when called. The number of received bytes is stored in *len*.

Return: 0 or an error code described in the appendix.

canCalServerWait()

Name: **canCalServerWait()** - waiting for a transfer operation

Call:

```
int   canCalServerWait
      (
      HCAN          hcan,     /* CAN-handle */
      long *        mux,      /* CMS-multiplexor */
      int *         up        /* transmission direction */
      )
```

Description: This function waits for the handle *hcan* as a server for a transfer operation to be initiated by a client in accordance with the CMS domain protocol. When opening the handle, the flag `RXTX_DOMAIN` has to be set. The function only returns after a client has initiated a domain transfer or when the value for Rx-timeout, as specified when the handle was opened, has been exceeded.
In *mux* the multiplexor of the data is stored which the client wants to upload or download.

The desired direction of transmission is returned in *up* (0 = download, 1 = upload). In order to continue the domain transfer depending on *up* the application has to call the routines **canCalServerUpload()** or **canCalServerDownload()**.

Return: 0 or an error code described in the appendix.

---

canCalServerUpload()

Name: **canCalServerUpload()** - continuing a receive transfer

Call:

```
int   canCalServerUpload
      (
      HCAN          hcan,     /* CAN-handle */
      void *        buf,      /* pointer to memory for receive data */
      long *        len       /* number of transmitted bytes */
      )
```

Description: This function has to be called after **canCalServerWait()** has returned successfully, if the client initiated a download. When opening the handle *hcan*, the flag `RXTX_DOMAIN` has to be set. The received data is copied into the memory determined by *buf*, whose length is specified in *len* when called. After the transmission has been finished, the number of actually received bytes is stored in *len*.

Return: 0 or an error code described in the appendix.

canCalServerDownload()

Name: **canCalServerDownload()** - continuing a transmission transfer

Call:

```
int     canCalServerDownload
        (
        HCAN            hcan,       /* CAN-handle */
        const void *    buf,        /* pointer to transmission data */
        long            len         /* number of bytes to be transmitted */
        )
```

Description: This function has to be called after **canCalServerWait()** has successfully returned, if the client initiated an upload. When opening the handle *hcan* , the flag `RXTX_DOMAIN` has to be set.
The transmission data is downloaded from the buffer *buf*, and the number of the bytes to be transmitted is determined in *len*.

Return: 0 or an error code described in the appendix.

---

canCalServerWaitAndUpload()

Name: **canCalServerWaitAndUpload()** - waiting and executing an upload

Call:

```
int     canCalServerWaitAndUpload
        (
        HCAN            hcan,       /* CAN-handle */
        long *          mux,        /* CMS-multiplexor */
        void *          buf,        /* pointer to buffer for receive data */
        long *          len         /* number of transmitted bytes */
        )
```

Description: This function waits for the handle *hcan* as server for a download to be initiated by a client in accordance with the CMS-domain transfer protocol and copies the received data into the buffer determined in *buf*, whose length is determined in *len* when called. After the transmission has been finished the number of actually received bytes is returned in *len* and the multiplexor is returned in *mux*. When opening the handle, the flag `RXTX_DOMAIN` has to be set. The function only returns, when a client initiated a domain upload or when the value for the Rx-timeout, specified when the handle was opened, has been exceed.

The advantage of this call compared to a combination of **canCalServerWait()** and **canCalServerUpload()** is in avoiding a task change, because the initiation is acknowledged on interrupt level and the calling task only has to be considered again after the complete data transfer has been finished.

Return: 0 or an error code described in the appendix.

Appendix Implementation

Contents

Page

A 1.1 Number and Assignment of Networks

The driver for the VME-CAN2 supports up to sixteen networks. The network number for both channels is set by means of the coding switches in the front panel (see hardware manual). The network number cannot be configured by means of software.

The driver for the VME-CAN4 supports up to 32 networks. The network number for the four channels ca be configured by means of software (see canTools manual).

A 1.2 Interaction and Multi-Processing Ability

Interaction means the ability of a driver to pass on transmission commands on a certain CAN-identifier in a certain network via the CAN-bus as well as internally to other processes which have acknowledged reception for this CAN-identifier. A useful consequence of this property is that a CAL-master and a CAL-slave can be initiated in the same network, for example.

Multi-processing ability means the property of the driver that transmission and receive commands can be given in the same CAN-identifiers. A useful consequence of this property is that two CAL-slaves can be initiated in the same network, for example.

The CAN2/CAN4-driver supports interaction as well as multi-processing ability.

If the driver has not been adapted to using interaction, the functionality to wait with simultaneous upload without additional task change (canCalServerWaitAndUpload) will not be supported.

A 1.3 Configuring the Bit Rate

After a RESET the VME-CAN2 has got the bit rate which has been set by means of coding switches SW2 and SW4 (see hardware manual) and can be overwritten by **canCalSetBaudrate()**. If these coding switches are set to 0xF, the CAN2 performs passively on the bus until the bit rate is set accordingly by means of software.

The VME-CAN4 is always passive on board after a RESET so that the bit rate must always be set by **canCalSetBaudrate()**.

If a value between 0x0000 and 0x000E is specified for **canCalSetBaudrate()** for the parameter *baud*, the bit rate will be set according to the following table. Each value between 0x0011 and 0x7F7F will be directly entered into the *Bit Timing Register* of the CAN-controller.

Index [HEX]	Bit rate [kbit/s]	82C200- or SJA1000-register		typical values of the attainable line length $_{lax}$ [m]	minimum attainable line length $_{Lein}$ [m]
		BTR0 [HEX]	BTR1 [HEX]		
0	1000	00	14	37	20
1	800	00	18	59	42
2	666.6	00	1C	80	65
3	500	01	18	130	110
4	333.3	01	1C	180	160
5	250	02	1C	270	250
6	166	03	1C	420	400
7	125	04	1C	570	550
8	100	45	2F	710	700
9	66.6	09	1C	1000	980
A	50	4B	2F	1400	1400
B	33.3	18	1C	2000	2000
C	20	5F	2F	3600	3600
D	12.5	31	1C	5400	5400
E	10	00	16	7300	7300

The specifications in the table are based on limit values of the bit timing of the CAN-protocol, the delays of the local CAN-interface and the delays of the cable. The delay of the cable has been assumed with about 5.5 ns/m. Further influences, such as the terminal resistor, the resistivity, the cable geometry or external disturbances during transmission have not been considered in these specifications!

Target

DeviceNet API
for
esd-CAN-Boards

Software Manual

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
http://www.esd.eu
Phone: +49 (0) 511 3 72 98-0 • Fax: +49 (0) 511 3 72 98-68

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. In particular descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

esd reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd.eu
Internet: www.esd.eu

Manual File:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\DeviceN\TrgetUni\DeviceNet_API-Manual_en_17.wpd
Date of Print:	2016-01-22

Described Software	**Revision/Date**
Local DeviceNet API-Library	Windows 32/64-Bit: V2.2.0 / 2011-08-23
	VxWorks: V2.2.2 / 2004-05-28

Implementation at the Following Boards	**Order no.**
DN-ISA/331-11x DeviceNet DN-ISA/331-22x DeviceNet	C.2016.02 C.2016.04
DN-PCI/331-11x DeviceNet DN-PCI/331-22x DeviceNet	C.2017.06 C.2017.07
DN-PC-104/331 1x DeviceNet	C.2014.02
DN-VME-CAN/4 4x DeviceNet	V.1408.08

Changes in the chapters

The changes in the user's manual listed below affect changes in the software, as well as changes in the description of the facts only.

Version	Chapter	Alterations versus previous revisions	Alternations in software	Alternations in documentation
1.6	-	Description of function *dnetDataEvMask ( )* added.	-	x
1.7	-	Classification of notes inserted, notes revised	-	x
	6.	New chapter: “DeviceNet Errata DN-PCI/331" inserted	-	x

Technical details are subject to change without notice.

Classification of Notice Statements and Information

NOTICE
Notice statements are used to notify people on hazards that could result in things other than personal injury, like property damage or software error.

NOTICE
This NOTICE statement contains the general mandatory sign and gives information that must be heeded and complied with for a safe and error-free use.

INFORMATION

INFORMATION
Notes to point out something important or useful.

Contents

Page

1. Overview

The DeviceNet library is available for Windows and VxVorks. Further operating systems will be supported in the future. You will find the actual list of supported operating systems at the history page at the beginning of this manual.

The DeviceNet library supports as many DeviceNet channels as the underlying CAN driver provides to the DeviceNet library layer.

The following figure shows the structure of the software using the esd DN-ISA/331 as an example. The DeviceNet protocol handler program runs **locally** on the esd-CAN-board.

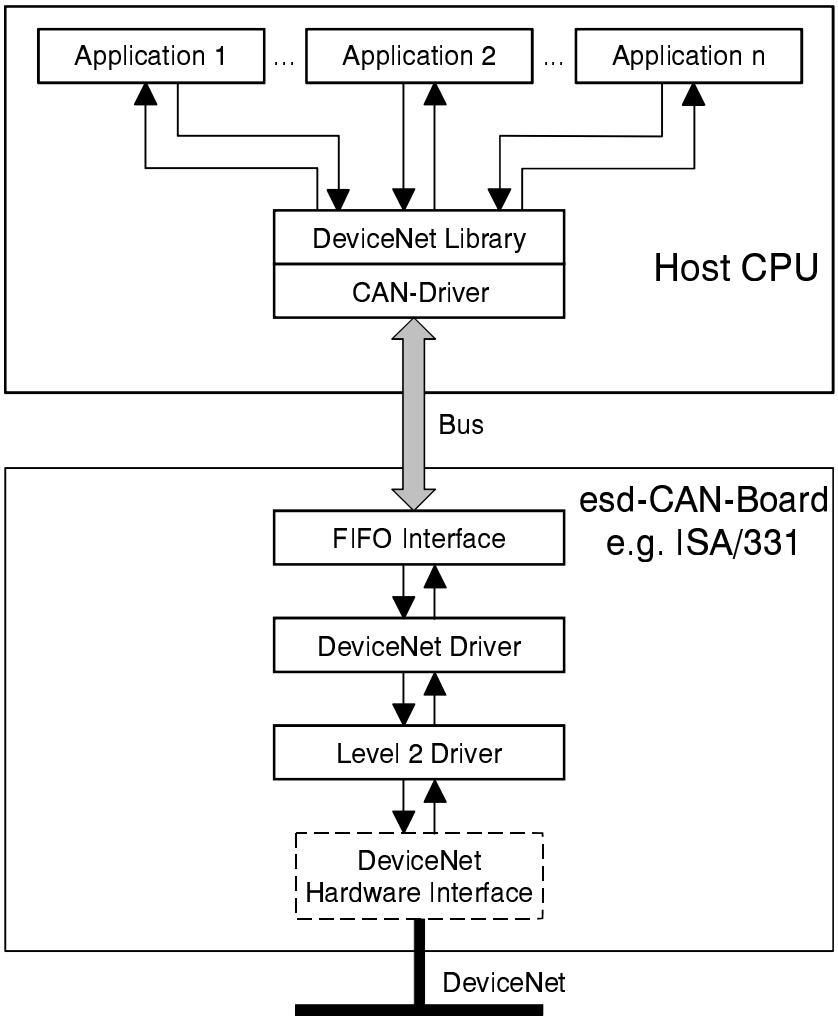


Fig. 1.1.1: Software structure

2. Function Description

2.1 Get Access to the Library Functions

dnetOpen( )

Name: **dnetOpen()** - Generates a handle to access the DeviceNet library

Synopsis:

```
STATUS    dnetOpen
          (
          int         net,
          HANDLE *handle
          )
```

Description: This function returns a handle that gives access to the other DeviceNet library functions.

Parameter:

net: logical net-no. (same number as CAN driver, defined in installation sequence)

handle*: handle initialized by ***dnetOpen ( )*

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

NOTICE

- To ensure the proper operation of the library every thread of execution (task) has to use its own handle. Or as a rule of thumb: only one thread per handle is allowed to enter the library at a time!
- Use a different handle at least for each independent running thread/task!
- If you are using a handle as event handle then you should use this handle only for calling ***dnetEvRead( )*** or ***dnetEvDisable( )***. Do **NOT** use it for any other calls!
- Only **ONE** event handle (***dnetEvEnable( )*** called with this handle) can exist for each network!
- The CAN driver allows at the moment a maximum of **8** DeviceNet handles per network. This is a compilation parameter and can easily be changed on demand.

dnetClose( )

Name: **dnetClose()** - Closing a handle of the DeviceNet library

Synopsis:

```
STATUS          dnetClose
(
HANDLE          handle
)
```

Description: This function closes the opened handle.

Parameter: *handle*: handle returned from ***dnetOpen ( )***

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.2 Hardware and DeviceNet Initialisation

dnetStart( )

Name: **dnetStart( )** - Initialize and start DeviceNet protocol

Synopsis:

```
STATUS   dnetStart
         (
         HANDLE handle,
         int    MACID,
         int    baudrate,
         int    rxLen,
         int    txLen,
         byte   option
         )
```

Description: This function initializes and starts the DeviceNet protocol on the network selected by *handle*. For the protocol the following parameters are necessary:

Parameter:

handle:	specifies the network to operate on	
MACID:	MACID of the local scanner and slave interface (0...63)	
baudrate:	bit rate	0 = 125 kbit/s
		1 = 250 kbit/s
		2 = 500 kbit/s
rxLen:	consumed data length of the slave interface (0...440)	
txLen:	produced data length of the slave interface (0...440)	
option:	bit 0: Event on receiving I/O data (local slave)	
	bit 1: Event on receiving changed I/O data (local slave)	

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetStop( )

Name: **dnetStop( )** - Remove remote slaves from scanner table

Synopsis:

```
STATUS    dnetStop
          (
          HANDLE handle
          )
```

Description: This function removes all remote slaves from the scanner table (its list of slaves to scan) and stops all CAN actions.

Parameter: *handle*: specifies the network to stop

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetInstRemSlave( )

Name: **dnetInstRemSlave( )** - Install a remote slave

Synopsis:

```
STATUS   dnetInstRemSlave
         (
HANDLE   handle,
int      MACID,
int      rxLen,
int      txLen,
int      EPR,
byte     allocation,
byte     option
         )
```

Description: This function installs a remote slave in the local scanner table. The firmware's local data image is initialized to zero. After that the scanner begins immediately to talk to this slave and maintains the requested connection to the slave.

Parameter:

handle: specifies the network to operate on

MACID: MACID of the remote slave interface (0...63)

rxLen: produced data length of the remote slave interface (0...440)

txLen: consumed data length of the remote slave interface (0...440)

EPR: The **e**xpected **p**ackage **r**ate of the slave (0...32767 ms). The scanner polls the slave with the half time of the EPR.
Limits: 0x3FFF (COS/Cyclic), 0x7FFF (Polling, BitStrobe)

allocation: Meaning is as described in the DeviceNet specification. An allocation value of 0 (zero) means to deinstall the remote slave!

Bit no.	7	6	5	4	3	2	1	0
Content	Reserved	Acknowledge Suppression	Cyclic	Change of State	Reserved	Bit Strobed	Polled	Explicit Message

Table 2.1.1: Allocation choice byte contents

The following choices for the allocation byte are possible:
- cyclic and/or change of state (COS) (0x20/0x10)
- bit strobed (0x04)
- polled (0x02)

Specify only one of these three. Additional selection of explicit message connection (0x01) is allowed

option: Mask 0x01: Event on receiving I/O data (remote slave)
Mask 0x02: Event on receiving changed I/O data (remote slave)
Mask 0x04: Generate event if receiving data changed in the valid bits area of the event mask (see ***dnetDataEvMask( )***)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetDataEvMask( )

Name: **dnetDataEvMask( )** -Modify data change event mask

Synopsis:

```
STATUS    dnetDataEvMask
          (
HANDLE    handle,
int       MACID,
int       offset,
int       len,
void      *dataEvMask
          )
```

Description: This function modifies the data change event mask for a remote slave. After a call of ***dnetInstRemSlave( )*** with *option* set to mask 0x04 (event on receiving changed data) a data event mask is established and initialised to all bytes 0xff. That means all bytes of the slave's incoming data are checked for changes.

You can set a new data event mask with bitwise granularity using this function. Any cleared bit of the mask means that the correspondent bit of the slave's incoming data is ignored when the firmware checks for changes.

Parameter:

handle:	specifies the network to operate on
MACID:	MACID of the remote slave interface
offset:	position of the data inside the change mask
len:	length of data to modify
**dataEvMask*:	points to buffer with new event mask

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetIdent( )

Name: **dnetIdent( )** -Read board information

Synopsis:

```
STATUS    dnetIdent
          (
          HANDLE      handle,
          DN_IDENT    *ident
          )
```

Description: This function gets information out of the Identity class of the DeviceNet board. Also it gets the firmware release number. Look at the typedef for DN_IDENT in the `dnet.h` file for further information.

Parameter:

handle: specifies the network to operate on

**ident*: Information about the Identity class of DeviceNet board providing this DeviceNet node and the board's firmware release number

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.3 Common I/O Data Transfer

dnetRead( )

Name: **dnetRead( )** - Read input data from a DeviceNet module

Synopsis:

```
STATUS    dnetRead
          (
HANDLE    handle,
int       MACID,
int       offset,
int       *rlen,
void      *buffer
          )
```

Description: This function reads input data from firmware's local data image of the DeviceNet module with *MACID*. Specify the same MACID as in the ***dnetStart( )*** call to access the input data of the local slave interface.

Parameter:

handle:	specifies the network to operate on
MACID:	MACID of the remote slave or local slave interface (0...63)
offset:	position of the data inside the stream of this device
**rlen*:	in: # of bytes to read or max. *buffer* space, respectively out:# of bytes really transferred
**buffer*:	points to a buffer to store data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetWrite( )

Name: **dnetWrite( )** - Write output data to a DeviceNet module

Synopsis:

```
STATUS   dnetWrite
         (
HANDLE   handle,
int      MACID,
int      offset,
int      len,
void     *buffer
)
```

Description: This function writes output data to the firmware's local data image of the DeviceNet module with *MACID*. This new data is then sent immediately to a remote slave. Specify the same MACID as in the ***dnetStart( )*** call to access the output data of the local slave interface.

Parameter:

handle:	specifies the network to operate on
MACID:	MACID of the remote slave or local slave interface (0...63)
offset:	position of the data inside the stream of this device
len:	length of data to write
**buffer*:	points to buffer with the data to write

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.4 Special: BitStrobe From Slave to Scanner

dnetWriteBitStrobe( )

Name: **dnetWriteBitStrobe( )** -Write bit strobe data to the DeviceNet scanner

Synopsis:

```
STATUS      dnetWriteBitStrobe
            (
            HANDLE      handle,
            int         flags,
            void        *buffer
            )
```

Description: This function writes 8 bytes of data to the scanner that are used as bit strobe data for any remote slave configured with a bit strobe connection. Or you may configure the data length of the bitstrobe frame to zero or eight bytes. In any case *buffer* must point to a 8 byte memory space.

Parameter:

handle: specifies the network to operate on

flags: Windows:

flags are reserved, please preset to (0x08) for compatibility with future releases.

VxWorks:

mask 0x08/0x00 selects 8 or 0 bytes bitstrobe frame, mask 0x01/0x00 selects immediate update or in next bitstrobe cycle.

**buffer*: pointer to a buffer for the data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetReadBitStrobe( )

Name: **dnetReadBitStrobe( )** - Read data from the bit strobe connection

Synopsis:

```
STATUS    dnetReadBitStrobe
          (
          HANDLE      handle,
          int         MACID,
          void        *buffer
          )
```

Description: If the local slave is scanned by a remote scanner via a bitstrobe connection then you can read the data received from the bitstrobe connection with this function. You always get eight bytes of data.

Parameter:

handle: specifies the network to operate on
MACID: MACID of the local slave interface (0...63)
**buffer*: pointer to a buffer for the data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.5 Direct Data/Explicit Transfer via Scanner

dnetWriteRead( )

Name: **dnetWriteRead( )** - Write data and read answer

Synopsis:

```
STATUS    dnetWriteRead
          (
          HANDLE handle,
          int    MACID,
          int    tlen,
          int    *rlen,
          void   *buffer
          )
```

Description: This function writes the data to a remote DeviceNet slave and reads after reception the answer.

Parameter:

handle:	specifies the network to operate on
MACID:	MACID of the remote slave (0...63)
tlen:	length of data to be transmitted
**rlen*:	max. length to receive, returns received length
**buffer*:	pointer to buffer to store data (Tx and Rx)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetExplRequest( )

Name: **dnetExplRequest( )** - Send an explicit request and read response

Synopsis:

```
STATUS    dnetExplRequest
          (
HANDLE    handle,
int       MACID,
int       service,
int       dnClass,
int       inst,
int       attr,
int       tlen,
int       *rlen,
void      *buffer
          )
```

Description: This function sends an explicit request to a remote module and returns the response after the reception.

Parameter:

handle:	specifies the network to operate on
MACID:	MACID of the remote module (0...63)
service:	DeviceNet service
dnClass:	DeviceNet class ID
inst:	DeviceNet instance ID
attr:	DeviceNet attribute ID (if zero, no attribute ID)
tlen:	length of data to be transmitted
**rlen*:	max. length to receive, returns received length
**buffer*:	points to buffer to store data (Tx and Rx) The buffer holds after return the explicit message body of the response, means the first byte is the service&response byte. For the format of the explicit message body please refer to the DeviceNet specification.

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.6 State Information

dnetReadFail( )

Name: **dnetReadFail( )** - Read status information of the DeviceNet interface

Synopsis:

```
STATUS    dnetReadFail
          (
HANDLE    handle,
void      *buffer
          )
```

Description: This function reads status information of the DeviceNet interface. The function reads a 64 bit = 8 byte array. Each set bit represents a failed or absent module (bit position = *MACID*), but bits of not installed modules are cleared.

The LSB of buffer [0] is bit 0 [-> MACID0].
The MSB of buffer [7] is bit 63 [-> MACID63].

Parameter:

handle:	specifies the network to operate on
**buffer*:	points to buffer to store data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetReadState( )

Name: **dnetReadState( )** - Read status of the DeviceNet module

Synopsis:

```
STATUS    dnetReadState
          (
HANDLE    handle,
int       MACID,
int       *state
          )
```

Description: This function reads the status of DeviceNet module with *MACID*. Specify the same MACID as in the ***dnetStart( )*** call to access the status of the local slave interface.

Parameter:

handle: specifies the network to operate on
MACID: MACID of the remote module or local slave interface (0...63)
**state*: variable that returns the state of the selected module as described below

States of Remote/Internal Slaves	
0	STATE_NOT_EXISTENT
1	STATE_WAIT_CONNECT
2	STATE_CONFIGURING
3	STATE_OPERATIONAL
4	STATE_TIMEOUT
5	STATE_ERROR

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.7 Event Handling

dnetEvEnable( )

Name: **dnetEvEnable( )** - Enable events for a network and a handle

Synopsis:

```
STATUS    dnetEvEnable
          (
          HANDLE          handle
          )
```

Description: This function enables the possibility to read events on the handle *handle*. Event generation is enabled for the network *handle* is opened for.

Parameter: *handle*: specifies the network to operate on

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

NOTICE You must NOT use a handle with events enabled for other purposes than reading events. This means it can only be used for ***dnetEvRead( )*** or ***dnetEvDisable( )***.

Header: `dnet.h`

dnetEvDisable( )

Name: **dnetEvDisable( )** - Disable events for a network and a handle

Synopsis:

```
STATUS    dnetEvDisable
          (
          HANDLE          handle
          )
```

Description: This function stops event generation for the net *handle* is valid for.
The last generated event on this net is:
EventNr == SERVICE_CLOSE
AddInfo == EVENT_SRV
This event may be used to exit from the event read loop.

Parameter: *handle*: specifies the network to operate on

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetEvRead( )

Name: **dnetEvRead( )** - Read one event of a DeviceNet network

Synopsis:

```
STATUS    dnetEvRead
          (
HANDLE    handle,
int       *EventNr,
int       *AddInfo,
int       *MACID
)
```

Description: This function reads one event from the network specified by *handle*.

Parameter:

handle:	handle with events enabled by ***dnetEvOpen ( )***
**EventNr*:	returns event number
**AddInfo*:	returns additional information value
**MACID*:	returns the MACID of the DeviceNet module that triggered the event

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.8 Host as Server for Explicit Messages

dnetExplSrvEnable( )

Name: **dnetExplSrvEnable( )** - Enable transfers between remote scanner and local slave

Synopsis:

```
STATUS    dnetExplSrvEnable
          (
          HANDLE    handle,
          int       MACID
          )
```

Description: This function enables the direct exchange of explicit messages between a remote scanner and the local slave through the vendor specific class 100. After this call explicit requests of a remote scanner are forwarded to the local application.

Parameter:
handle: handle of the specified net
MACID: MACID of the local slave interface (0...63)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetExplSrvDisable( )

Name: **dnetExplSrvDisable( )** - Disable transfers between remote scanner and local slave

Synopsis:

```
STATUS    dnetExplSrvDisable
          (
          HANDLE    handle,
          int       MACID
          )
```

Description: This function disables the forwarding of explicit messages from a remote scanner to the local slave and then to the local application program.

Parameter:

handle: handle returned by ***dnetOpen ( )***
MACID: MACID of the internal slave interface (0...63)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetExplSrvRead( )

Name: **dnetExplSrvRead( )** - Wait for a request from remote client and return its service request type

Synopsis:

```
STATUS    dnetExplSrvRead
          (
          HANDLE      handle,
          int         *cnxn,
          int         *service,
          int         *dnClass,
          int         *inst,
          int         *attr,
          int         *rlen,
          void        *buffer
          )
```

Description: This function waits for an explicit request from a remote client and returns it.

Parameter:

handle:	handle on which ***dnetExplSrvEnable ( )*** was done
**cnxn*:	connection number, return to card on response
**service*:	service number, return to card on response
**dnClass*:	client request to class
**inst*:	client request to instance
**attr*:	client request to attribute
**rlen*:	size of buffer space, returns received data length
**buffer*:	pointer to buffer to store data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetExplSrvResponse( )

Name: **dnetExplSrvResponse( )** - Response on request from remote client

Synopsis:

```
STATUS    dnetExplSrvResponse
          (
          HANDLE      handle,
          int         cnxn,
          int         service,
          int         len,
          void        *buffer
          )
```

Description: ***dnetExplSrvResponse ( )*** is the response of an explicit request from a remote client.

Parameter:

handle:	handle returned by ***dnetOpen ( )***
cnxn:	connection number, returned to card on response
service:	service number, returned to card on response
len:	size of buffer space, returns received data length
**buffer*:	pointer to buffer of data to transmit

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

2.9 Host as Server for Polled I/O-Connections

dnetPollSrvEnable( )

Name: **dnetPollSrvEnable ( ) -** Enable I/O-transfers between remote scanner and local slave

Synopsis:

```
STATUS    dnetPollSrvEnable
          (
          HANDLE          handle,
          int             MACID
          )
```

Description: This function enables the direct reception and reply of I/O-messages from a remote scanner.

Parameter:
handle: opened by ***dnetOpen ( )***
MACID: MACID of the local slave interface (0...63)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetPollSrvDisable( )

Name: **dnetPollSrvDisable( ) -** Disable I/O-transfers between remote scanner and local slave

Synopsis:

```
STATUS    dnetPollSrvDisable
          (
          HANDLE          handle,
          int             MACID
          )
```

Description: This function disables the direct exchange of I/O-polling data between a remote scanner and the local slave.

Parameter:
handle: handle returned by ***dnetOpen ( )***
MACID: MACID of the internal scanner and slave interface (0...63)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetPollSrvRead( )

Name: **dnetPollSrvRead( )** - Wait and read I/O-message from remote server

Synopsis:

```
STATUS    dnetPollSrvRead
          (
          HANDLE handle,
          int                MACID,
          int                *rlen,
          void               *buffer
          )
```

Description: This function waits for an I/O-message from a remote scanner and reads it.

Parameter:

handle:	handle on which ***dnetPollSrvEnable ( )*** was done
**rlen*:	size of buffer space, returns received data length
**buffer*:	pointer to buffer to store data
MACID:	MACID of the internal slave interface (0...63)

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

dnetPollSrvResp( )

Name: **dnetPollSrvResp( )** - Response I/O-poll request from remote scanner

Synopsis:

```
STATUS    dnetPollSrvResp
          (
HANDLE    handle,
int       MACID,
int       len,
void      *buffer
          )
```

Description: ***dnetPollSrvResp ( )*** is the response of an I/O-poll request of a remote scanner.

Parameter:

handle:	handle returned by ***dnetOpen ( )***
MACID:	MACID of the internal slave interface (0...63).
len:	size of data
**buffer*:	pointer to buffer to store data

Return: After successful execution `DNET_OK` is returned, otherwise an error code as described in chapter 3 is returned.

Header: `dnet.h`

3. Error Codes

Error code		Description
-1	DNET_INTERNAL_ERROR	other errors, e.g. DeviceNet scanner has stopped
0	DNET_OK	no error
1	DNET_WRONG_COMMAND	command not implemented
2	DNET_WRONG_PARA	general parameter error
3	DNET_WRONG_LENGTH	wrong length selected
4	DNET_WRONG_NET	wrong net-no selected
5	DNET_WRONG_MACID	wrong MACID
6	DNET_INV_DATA_ACCESS	data access out of range
7	DNET_WRONG_BAUDRATE	wrong bit rate selected
8	DNET_ALLREADY_EXIST	handle already exists
9	DNET_WRONG_STATE	wrong status
10	DNET_DISCONNECTED	status = disconnected
11	DNET_ON_TEST	-
12	DNET_NOT_STARTED	module not started
13	DNET_SRV_PENDING	service already pending

4. Events

The following events can occur:

Event	Event_no
NET_EVENT	1
TIMEOUT_EVENT	2
ERROR_EVENT	3
STATE_CHANGE	4
DATA_EVENT	5
CHANGE_EVENT	6
EXPLICIT_REQUEST	7
EXPLICIT_RESPONSE	8
POLL_REQUEST	9
POLL_RESPONSE	10
NET_REMOVED	11
SERVICE_CLOSE	12

Further additional information are returned with the events:

Additional Info	Info Code
BUS_OK_STATE	1
BUS_WARN_STATE	2
BUS_OFF_STATE	3
EXPLICIT_CNX	4
POLLING_CNX	5
BIT_STROBE_CNX	6
COS_CYCLIC_CNX	7
DUPLICATE_MACID	8
CONFIGURATION	9
FRAG_RESPONSE	10
ABORTED	11
PROT_ERROR	12
ERROR_ACKN	13
NO_RESOURCE	14
TIME_OUT	15
EVENT_SRV	16
EXPL_SRV	17
POLL_SRV	18

5. Using the DeviceNet Software

5.1 How to Write or Fit an Application

1. Open a handle using ***dnetOpen ( )***

2. Start the interface using ***dnetStart ( )***

> **NOTICE**
> The MACID is freely selectable, but must be unique inside the network (duplicate MACID check).

The *rxlen* and *txlen* means the consumed and produced data length of the I/O-connection to the internal slave interface. If you do not want to build this I/O-connection, set these parameters to zero.

3. Define the remote slaves. The scanner should access ***dnetInstRemSlave ( )***

> **NOTICE**
> The scanner checks *rxlen* and *txlen* against a found remote slave. If this check fails, the connection is not established! So you have to define the exact data length of the I/O-polling connection.

After this command the scanner tries immediately to connect this slave.

4. Check successful connection using ***dnetReadFail ( )***
Every not established or failed connection is shown by this command.

5. Read/Write data using ***dnetRead ( )*** and
dnetWrite ( )

6. Stop the application using ***dnetClose ( )***
This command closes only the handle opened by ***dnetOpen ( )***.The DeviceNet interface is not affected. A following application can directly access the DeviceNet data after ***dnetOpen ( )***.

5.2 Access Example

You have connect a flex I/O-module (AB) with a single digital input module IB16 (16-bit input) to your network.

The produced data length is 4 (= *rxlen* in ***dnetInstRemSlave ( )***):
2 bytes state information from the flex I/O-adapter and 2 bytes data from the input module.

The consumed data length is 2 (= *txlen* in ***dnetInstRemSlave ( )***):
2 bytes configuration for the input module.

If you want to read the input-data word by ***dnetRead ( )***, the resulting offset is 2 (data word after state information) and the length is 2 (word).

6. DeviceNet Errata DN-PCI/331

The DN-PCI/331 board has not been presented to an official ODVA test lab to apply for a conformance certificate. However the board's protocol implementation of the DeviceNet slave has been checked with the conformance test tool (CTT) that official ODVA test labs use while doing the conformance test for a DeviceNet appliance. The used release of the CTT was “DeviceNet Protocol Conformance Revision A22, November 2010”.
During the tests some deviating behaviour was observed that is described in the following paragraphs:

6.1 Master /Scanner: The UCMM loses Message IDs

The scanner needs to allocate a message ID for the UCMM connection when it starts talking to an external slave. If the external slave does not answer and in that state the connection is closed using *dnetInstRemSlave(allocation=0)* then the UCMM Connection ID may be locked and the connection ID is lost for later connection attempts. There is only the limited amount of 16 connection IDs available. This is a severe problem if you try to scan the DeviceNet network for any slaves present and you possibly lose a connection ID on every slave not present.
It is only a minor problem if you only need to talk to a fixed list of slaves and do not use *dnetInstRemSlave(allocation=0)* because as soon as a slave answers, its connection ID is returned to the pool and reused.
This error does only affect the master/scanner side and is not tested in ODVA's conformance test.

6.2 Master/Scanner: CoS Allocation is incorrect

When using the CoS connection only, conflicts with the required behaviour of the slave occur. In particular a single CoS connection can not be chosen.
The allocation byte is determined by the configuration of the external DeviceNet slave via the function *dnetInstRemSlave()*, see table 2.1.1 on page 11.
The behaviour for the allocation of a CoS/Cyclic connection together with a Polled connection is described for the slave side in the DeviceNet specification Volume 3, Chapter 3.13 "Change of State/Cyclic Messages". In particular the behaviour, in the case that both connections (CoS/Cyclic & Polled) are not enabled using a single allocation request, is precisely specified.

The following deviations from the specification occur:

- If you use an allocation byte of 0x11 (CoS only & Explicit) in *dnetInstRemSlave()*, the scanner will implicitly switch on the Polled connection if the consumed length of the external slave is unequal zero (0). According to the standard, the scanner should only use the CoS connection and leave the Polled connection unused. Therefore the board's scanner can not use the CoS connection as an input only connection.
- If you use an allocation byte of 0 x13 (CoS & Polled & Explicit) in *dnetInstRemSlave()*, the scanner will wrongly send an allocation request with a choice byte of 0x01 (Explicit) and then 0x02 (Polled Only).

NOTICE
To avoid these errors do not use the allocation bytes 0x11 (CoS only & Explicit) or 0x13 (CoS & Polled & Explicit) when calling *dnetInstRemSlave()* for an external slave.

6.3 Slave/Device: Network Status is not always indicated correctly

During the conformance test the network status is not always indicated correctly by the Status LED (especially if only UCMM connections do exist).
The DeviceNet object test of the CTT performs a Network Status LED test, because a combined Module/Network Status LED is provided. The Network Status LED test applies also to the combined LED, because the CTT does only test the interaction on the network. The states of the hardware, which are to be additionally indicated by the combined LED, do not occur during the test and consequently do not affect the display of the combined LED.
When the state machine is run though, the blinking codes of the combined DeviceNet Module/Network Status LED do not indicate the DeviceNet network states correctly.
For example, the slave should switch the LED to light permanently green already if a connection is dynamically established via UCMM, but the LED remains blinking green.

6.4 Network Access State Machine after Communication Failure

This problem occurs in the communication failure state (for example after a CAN bus-off). When the 24V power supply voltage of the DeviceNet interface is removed and then supplied again in that state, the Network Access State Machine should restart, carry out a Duplicate MACID Check and leave the communication failure state on Duplicate MACID Check success. But this does not happen.

INFORMATION
To leave the communication failure state you may power down the DN-PCI/331 or reconfigure all slaves and then call *dnetStop()* followed by *dnetStart()*. This should reset this condition.

6.5 Network Status on DeviceNet Power Loss

The network status at loss of the 24V DeviceNet power supply voltage is not indicated correctly.
The LED is not switched to blinking red if the power supply voltage on the DeviceNet bus is not supplied.

6.6 Restriction in the EDS File for the DeviceNet Slave Part of the Device

The EDS file currently does not reflect, that the I/O lengths of the DeviceNet slave are completely configurable. The EDS itself declares only fixed input and output lengths for the slave part of the device.

INFORMATION
To work around this problem enter values of your configuration for the amount of input and output bytes into the EDS file of the DN-PCI/331 manually at the appropriate places.