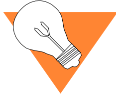
CANpie FD

User manual

Version 3.00

Document conventions

For better handling of this manual the following icons and headlines are used:



This symbol marks a paragraph containing useful information about the software operation or giving hints on configuration.

This symbol marks a paragraph which describes actions to be executed by the user of the source code package.

Keywords

Important keywords appear in the border column to help the reader when browsing through this document.

Syntax, Examples

`For function syntax and code examples the font face Source Code Pro is used.`

MicroControl GmbH & Co. KG
Junkersring 23
D-53844 Troisdorf
Fon: +49 / 2241 / 25 65 9 - 0
Fax: +49 / 2241 / 25 65 9 - 11
http://www.microcontrol.net

1. Scope 5
1.1 References 5
1.2 Abbreviations 6
1.3 Introduction to CAN 7
1.4 License 8
1.5 Document History 8

2. Driver Principle 9
2.1 Message Distribution 10
2.2 Data Types 10
2.3 Naming Conventions 11
2.4 File Structure 12
2.5 Configuration Options 13
2.6 Initialization Process 14

3. API Overview 15
3.1 Physical CAN Interface 15
3.2 Hardware Description Interface 16
3.3 Structure of a CAN message 18
3.4 Bittiming 20
3.5 CAN Statistic Information 21
3.6 Error Codes 22

4. Core Functions 23
4.1 Deprecated Functions 24
4.2 CpCoreBitrate 25
4.3 CpCoreBufferConfig 26
4.4 CpCoreBufferGetData 28
4.5 CpCoreBufferGetDlc 29
4.6 CpCoreBufferRelease 30
4.7 CpCoreBufferSend 31
4.8 CpCoreBufferSetData 32
4.9 CpCoreBufferSetDlc 33
4.10 CpCoreCanMode 34
4.11 CpCoreCanState 36
4.12 CpCoreDriverInit 37
4.13 CpCoreDriverRelease 39

4.14 CpCoreFifoConfig 40
4.15 CpCoreFifoRead 41
4.16 CpCoreFifoRelease 42
4.17 CpCoreFifoWrite 43
4.18 CpCoreHDI 44
4.19 CpCoreIntFunctions 45
4.20 CpCoreStatistic 47

5. CAN Message Functions 49

5.1 CpMsgGetData 50
5.2 CpMsgGetDlc 51
5.3 CpMsgGetIdentifier 52
5.4 CpMsgInit 53
5.5 CpMsgIsExtended 54
5.6 CpMsgIsRemote 55
5.7 CpMsgSetData 56
5.8 CpMsgSetDlc 57
5.9 CpMsgSetIdentifier 58
5.10 CpMsgSetRemote 59

A LGPL LICENSE 61

B Index 65

1. Scope

CANpie (CAN Programming Interface Environment) is an open source project and pursues the objective of creating and establishing an open and standardized software API for access to the CAN bus.

The current version of the CANpie API covers both classic CAN frames as well as ISO CAN FD frames. The API is designed for embedded control applications as well as for PC interface boards: embedded microcontrollers are programmed in C, a C++ API is provided for OS independent access to interface boards. The API provides ISO/OSI Layer-2 (Data Link Layer) functionality. It is not the intention of CANpie to incorporate higher layer functionality (e.g. CANopen, J1939).

CANpie provides a method to gather information about the features of the CAN hardware. This is especially important for an application designer, who wants to write the code only once.

1.1 References

/1/ ISO 11898-1:2015, Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling

/2/ ISO 11898-2:2016, Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit

/3/ ISO 11898-3:2006, Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium access unit

/4/ CANpie users guide, Version 2.0, MicroControl GmbH & Co. KG www.microcontrol.net/en/products/protocol-stacks/canpie-fd/

1.2 Abbreviations

CAN	Controller area network
CAN-ID	CAN identifier
CBFF	Classical base frame format
CEFF	Classical extended frame format
CRC	Cyclic redundancy check
FBFF	FD base frame format
FEFF	FD extended frame format
LSB	Least significant bit/byte
MSB	Most significant bit/byte
OSI	Open systems interconnection
PLS	Physical layer signalling
PMA	Physical medium attachment
RTR	Remote transmission request

1.3 Introduction to CAN

The CAN (Controller Area Network) protocol is an international standard defined in the ISO 11898 standard /1/.

CAN is based on a broadcast communication mechanism. This broadcast communication is achieved by using a message oriented transmission protocol. These messages are identified by using a message identifier. The message identifier has to be unique within the whole network and it defines not only the content but also the priority of the message.

The priority at which a message is transmitted compared to another less urgent message is specified by the identifier of each message. The priorities are laid down during system design in the form of corresponding binary values and cannot be changed dynamically. The identifier with the lowest binary number has the highest priority. Bus access conflicts are resolved by bit-wise arbitration on the identifiers involved by each node observing the bus level bit for bit. This happens in accordance with the "wired and" mechanism, by which the dominant state overwrites the recessive state. The competition for bus allocation is lost by all nodes with recessive transmission and dominant observation. All the "losers" automatically become receivers of the message with the highest priority and do not re-attempt transmission until the bus is available again.

The CAN protocol supports four message frame formats:

- Classical base frame format (CBFF):

  message that contains up to 8 byte and is identified by 11 bits

- Classical extended frame format (CEFF):

  message that contains up to 8 byte and is identified by 29 bits

- FD base frame format (FBFF):

  message that contains up to 64 byte and is identified by 11 bits

- FD extended frame format (FEFF):

  message that contains up to 64 byte and is identified by 29 bits

1

1.4 License

CANpie is **free software**; you can redistribute it and/or modify it under the terms of the **GNU Lesser General Public License** as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This code / library is distributed hoping that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

The complete license text can be found as appendix to this manual.

1.5 Document History

Version	*Date*	*Description*
3.00 WD	01.12.2016	Work draft
3.00	13.04.2017	Release version

Table 1: Document history

2. Driver Principle

One of the ideas of CANpie is to keep it independent from the hardware. CANpie uses a message buffer (mailbox) model for hardware abstraction.

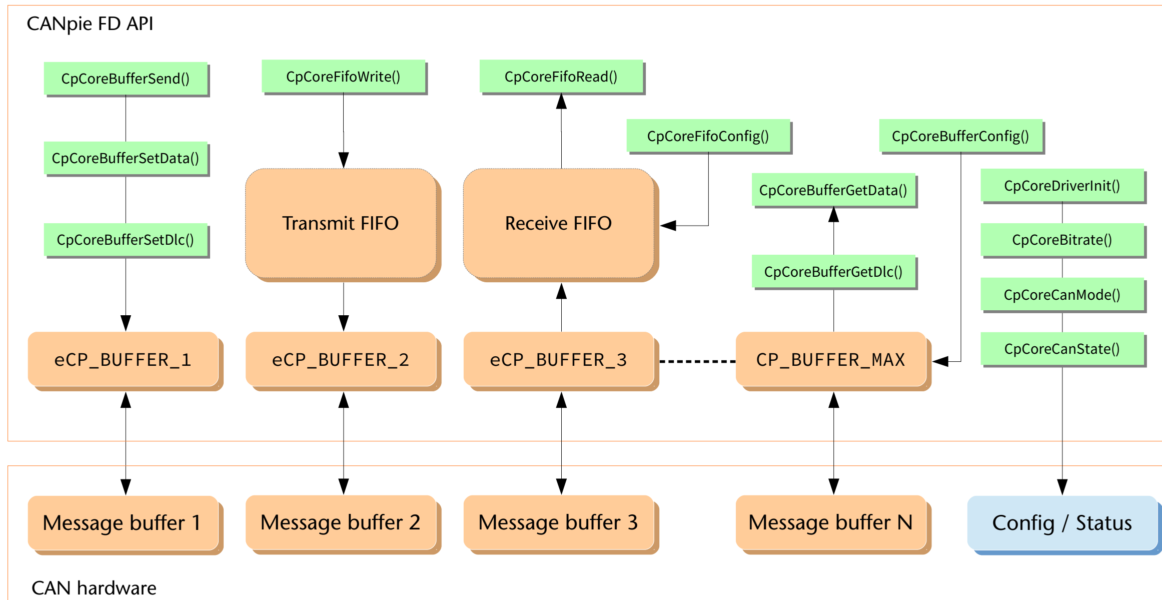


Figure 1: CANpie Structure

Core Functions

The core functions access the hardware directly, so an adaption is necessary when implementing on a piece of hardware.

A message buffer has a unique direction (receive or transmit), the initial setup is accomplished via `CpCoreBufferConfig()`. As an option it is possible to connect a FIFO with arbitrary size to a message buffer.

CANpie supports more than one CAN channel on the hardware. The actual number of CAN channels can be gathered via the Hardware Description Interface (refer to “Hardware Description Interface” on page 16).

2

2.1 Message Distribution

The message distribution is responsible for reading and writing CAN messages. The key component for message distribution is the Interrupt Handler. The Interrupt Handler is started by a hardware interrupt from the CAN controller. The Interrupt Handler has to determine the interrupt type (receive / transmit / status change).

Interrupt Handler

In case of a receive interrupt the handler uses the Receive Message routine to get the CAN message from the controller and put it into the Receive FIFO (First-In-First-Out). If the Receive FIFO is full, no further messages will be queued and an error-signal will be submitted.

In case of a transmit interrupt the Transmit FIFO is checked. If there are messages in this queue, the Transmit Message routine will write the next waiting message to the CAN controller. If the Transmit FIFO is empty and the application puts a CAN message into the queue, the Transmit Message routine will be called automatically.

Callback Functions

The occurrence of an interrupt may call a user defined handler function. Handler functions are possible for the following conditions:

- Receive interrupt
- Transmit interrupt
- Error / Status interrupt

2.2 Data Types

Due to different implementations of data types in the world of C compilers, the following data types are used for CANpie API. The data types are defined in the header file `compiler.h`.

Data Type	*Definition*
bool_t	Boolean value, True or False
uint8_t	1 Byte value, value range 0 .. 2^{8} - 1 (0 .. 255)
int8_t	1 Byte value, value range -2^{7} .. 2^{7} - 1 (-128 .. 127)
uint16_t	2 Byte value, value range 0 .. 2^{16} - 1 (0 .. 65535)
int16_t	2 Byte value, value range -2^{15} .. 2^{15} - 1
uint32_t	4 Byte value, value range 0 .. 2^{32} - 1
int32_t	4 Byte value, value range -2^{31} .. 2^{31} - 1

Table 2: Data Type definitions

2.3 Naming Conventions

All functions, structures, defines and constants in CANpie have the prefix `Cp`. Refer to table 3 for the used nomenclature:

CANpie	*Prefix*
Core functions	CpCore<name>
Message access functions	CpMsg<name>
Structures	Cp<name>_s
Constants / Defines	CP_<name>
Error Codes	CpErr<name>

Table 3: Naming conventions

All constants, definitions and error codes can be found in the header file `canpie.h`.

2

2.4 File Structure

The include dependency graph of the header files is show in figure 2, the contents of the files is described by table 4.

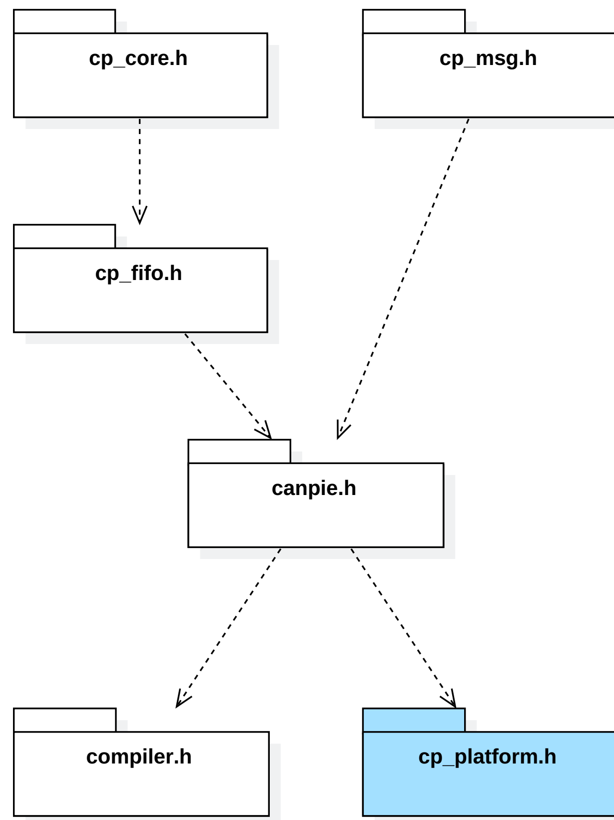


Figure 2: Include dependency graph

The header file `cp_platform.h` is unique for every target (CAN interface) and is located in the directory of the CAN driver.

File	*Description*
canpie.h	Definitions, structures and enumerations
compiler.h	Compiler independent data types
cp_core.h	Core functions
cp_fifo.c. / h	FIFO support
cp_msg.c / h	CAN message access
cp_platform.h	Configuration options for target

Table 4: CANpie files

2.5 Configuration Options

Configuration options for a specific target are defined inside the file `cp_platform.h`.

Symbol	*Default value*	*Description*
CP_AUTOBAUD	0	Automatic bit-rate detection
CP_BUFFER_MAX	8	Number of message buffers
CP_CAN_FD	1	Support of ISO CAN FD
CP_CAN_MSG_MACRO	1	CAN message access via macros
CP_CAN_MSG_TIME	1	Support of time-stamp field
CP_CAN_MSG_USER	1	Support of user-defined field
CP_CHANNEL_MAX	1	Number of physical CAN channels
CP_SMALL_CODE	0	Omit CAN port parameter
CP_STATISTIC	0	Support statistic information

Table 5: Configuration options

2

2.6 Initialization Process

The CAN driver is initialized with the function CpCoreDriverInit(). This routine will setup the CAN controller, but not configure a certain bit-rate nor switch the CAN controller to active operation. The following core functions must be called in that order:

- CpCoreDriverInit()
- CpCoreBitrate()
- CpCoreCanMode()

```
void MyCanInit(void)
{
   CpPort_ts  tsCanPortT;   // logical CAN port

   //---------------------------------------------------
   // setup the CAN controller / open a physical CAN
   // port
   //
   memset(&tsCanPortT, 0, sizeof(CpPort_ts));

   CpCoreDriverInit(eCP_CHANNEL_1, &tsCanPortT, 0);

   //---------------------------------------------------
   // setup 500 kBit/s
   //
   CpCoreBitrate(&tsCanPortT,
                 eCP_BITRATE_500K,
                 eCP_BITRATE_NONE);

   //---------------------------------------------------
   // start CAN operation
   //
   CpCoreCanMode(&tsCanPortT, eCP_MODE_OPERATION);


   //.. now we are operational

}
```

Example 1: Initialization process of the CAN interface

The function **CpCoreDriverInit()** must be called before any other core function in order to have a valid handle / pointer to the open CAN interface.

3. API Overview

This chapter gives an overview of the CANpie API. It also discusses the used structures in detail.

3.1 Physical CAN Interface

A target system may have more than one physical CAN interface. The physical CAN interfaces are numbered from 1 .. N (N: total number of physical CAN interfaces on the target system, defined by the symbol `CP_CHANNEL_MAX`). The header file `canpie.h` provides an enumeration for the physical CAN interface (the first CAN interface is `eCP_CHANNEL_1`). A physical CAN interface is opened via the function `CpCoreDriverInit()`. The function will setup a pointer to the structure `CpPort_ts` for further operations. The elements of the structure `CpPort_ts` depend on the used target system and are defined in the header file `cp_platform.h` (which also defines configuration options for the target).

```
/*----------------------------------------------------------*/
/*!
** \struct  CpPortEmbedded_s
** \brief   Port structure for embedded target
**
*/
struct CpPortEmbedded_s {

   /*!   Physical CAN interface number,
   **    first CAN channel (index) is eCP_CHANNEL_1
   */
   uint8_t     ubPhyIf;

   /*!   Private driver information
   */
   uint8_t     ubDrvInfo;

};
.....

typedef struct CpPortEmbedded_s     CpPort_ts;
```

Example 2: Example CAN port structure for an embedded target

For an embedded application with only one physical CAN interface the parameter to the CAN port can be omitted. This reduces the code size and also increases execution speed. This option is configured via the symbol `CP_SMALL_CODE` during the compilation process.

3

3.2 Hardware Description Interface

The Hardware Description Interface provides a method to gather information about the CAN hardware and the functionality of the driver. For this purpose the following structure is defined:

```
typedef struct CpHdi_s{
    uint8_t    ubVersionMajor;
    uint8_t    ubVersionMinor;
    uint8_t    ubCanFeatures;
    uint8_t    ubDriverFeatures;
    uint8_t    ubBufferMax;
    uint8_t    uwReserved[3];
    uint32_t   ulTimeStampRes;
    uint32_t   ulCanClock;
    uint32_t   ulBitRateMin;
    uint32_t   ulBitRateMax;
    uint32_t   ulNomBitRate;
    uint32_t   ulDatBitRate;
} CpHdi_ts;
```

The hardware description structure is available for each physical CAN channel.

Version Major

The element `ubVersionMajor` defines the major version number of the CANpie driver release.

Version Minor

The element `ubVersionMinor` defines the minor version number of the CANpie driver release.

CAN Features

The element `ubCanFeatures` defines the capabilities of the CAN controller.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
res.	res.	res.	res.	res.q	res.	CAN FD	Ext. Frame

Driver Features

The element `ubDriverFeatures` defines the capabilities of the software driver.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
res.						User Data	Timestamp

Message Buffer

The element `ubBufferMax` defines the total number of CAN message buffers (mailboxes).

Time-stamp The element `ulTimeStampRes` defines the resolution in nano-seconds (ns) of the optional time-stamp.

CAN Clock The element `ulCanClock` defines the clock rate of the CAN controller in Hertz (Hz).

Bit-rate Limits The elements `ulBitRateMin` and `ulBitRateMax` define the lower and upper limit values of the bit-rate. These values also respect the specified values of the used CAN transceiver.

CAN Bit-rate The element `ulNomBitRate` defines the actual configured bit-rate of the CAN controller in bits-per-second (bps). For ISO CAN FD the value defines the bit-rate of the arbitration phase.

CAN FD Bit-rate The element `ulDatBitRate` is only valid for ISO CAN FD controller. The value defines the actual configured bit-rate of the data phase in bits-per-second (bps).

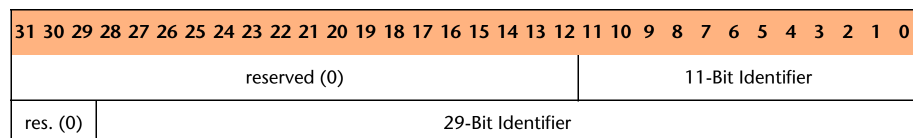
3.3 Structure of a CAN message

For transmission and reception of CAN messages a structure which holds all necessary informations is used (`CpCanMsg_ts`). The structure is defined in the header file canpie.h and has the following data fields:

```
typedef struct CpCanMsg_s {
    // identifier field (11/29 bit)
    uint32_t   ulIdentifier;

    // data field: 8 bytes (CAN) or 64 bytes (CAN FD)
    uint8_t    aubData[CP_DATA_SIZE];

    // Data length code
    uint8_t    ubMsgDLC;

    // frame type
    uint8_t    ubMsgCtrl;

    #if CP_CAN_MSG_TIME == 1
    CpTime_ts  tsMsgTime;
    #endif

    #if CP_CAN_MSG_USER == 1
    uint32_t   ulMsgUser;
    #endif

    #if CP_CAN_MSG_MARKER == 1
    uint32_t   ulMsgMarker;
    #endif
} CpCanMsg_ts;
```

Identifier

The identifier field (`ulIdentifier`) may have 11 bits for standard frames or 29 bits for extended frames. The three most significant bits are reserved (always 0).



31 30 29	28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
reserved (0)		11-Bit Identifier
res. (0)	29-Bit Identifier	

Data Field

The data field (`aubData[]`) may contain up to 8 bytes for a CAN message or up to 64 bytes for a ISO CAN FD message. If the data length code is less than the maximum size, the value of the unused data bytes will be undefined.

Data Length Code

The data length code field (`ubMsgDLC`) holds the number of valid bytes in the data field array. The allowed range is 0 to 8 for CAN frames and 0 to 15 for ISO CAN FD frames.

DLC value	Payload size [byte]	Frame type
0 .. 8	0 ..8	CAN / ISO CAN FD
9	12	ISO CAN FD only
10	16	ISO CAN FD only
11	20	ISO CAN FD only
12	24	ISO CAN FD only
13	32	ISO CAN FD only
14	48	ISO CAN FD only
15	64	ISO CAN FD only

Table 6: DLC conversion for CAN / ISO CAN FD frames

Message Control

The message control field (`ubMsgCtrl`) contains detailed information about the CAN frame.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ESI	BRS	reserved	reserved	OVR	RTR	FDF	EXT

The `EXT` bit defines an *Extended Frame Format* if set to 1. It is allowed for classical CAN frames and FD CAN Frames.

The `FDF` bit defines a *Fast Data Frame* if set to 1 (i.e. FD CAN frame).

The `RTR` bit defines a *Remote Transmission Request* if set. It is only defined for classical CAN frames.

The `OVR` bit defines a *Overrun* during message reception if set.

The `BRS` bit defines a *Bit Rate Switch* if set to 1. It is only defined for FD CAN frames.

The `ESI` bit defines a *Error State Indicator* if set to 1. It is only defined for FD CAN frames.

Time Stamp

The time stamp field (`tsMsgTime`) defines the time when a CAN message was received by the CAN controller. The lowest possible resolution is one nanosecond (1 ns). This is an optional field.

The time stamp field is defined similar to the EtherCAT® distributed clocks time value.

User Data

The field user data (`ulUserData`) can hold a 32 bit value, which is defined by the user. This is an optional field.

3

3.4 Bittiming

To ensure correct sampling up to the last bit, a CAN node needs to re-synchronize throughout the entire frame. This is done at the beginning of each message with the falling edge SOF and on each recessive to dominant edge.

One CAN bit time is specified as four non-overlapping time segments. Each segment is constructed from an integer multiple of the Time Quantum. The Time Quantum or TQ is the smallest discrete timing resolution used by a CAN node. The four time segments are:
- the Synchronization Segment
- the Propagation Time Segment
- the Phase Segment 1
- and the Phase Segment 2

The sample point is the point of time at which the bus level is read and interpreted as the value (recessive or dominant) of the respective bit. Its location is at the end of Phase Segment 1 (between the two Phase Segments).

Programming of the sample point allows "tuning" of the characteristics to suit the bus. Early sampling allows more Time Quanta in the Phase Segment 2, so that the Synchronization Jump Width can be programmed to its maximum. This maximum capacity to shorten or lengthen the bit time decreases the sensitivity to node oscillator tolerances, so that lower cost oscillators such as ceramic resonators may be used. Late sampling allows more Time Quanta in the Propagation Time Segment which allows a poorer bus topology and maximum bus length.

In order to allow interoperability between CAN nodes of different vendors it is essential that both - the absolute bit length (e.g. 1µs) **and** the sample point - are within certain limits. The following table gives an overview of recommended bit-timing setups.

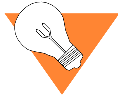
Bitrate	*Bittime*	*Valid range for sample point location*	*Recommended sample point location*
1 MBit/s	1 µs	75% .. 90%	87,5%
800 kBit/s	1,25 µs	75% .. 90%	87,5%
500 kBit/s	2 µs	85% .. 90%	87,5%
250 kBit/s	4 µs	85% .. 90%	87,5%
125 kBit/s	8 µs	85% .. 90%	87,5%
50 kBit/s	20 µs	85% .. 90%	87,5%
20 kBit/s	50 µs	85% .. 90%	87,5%
10 kBit/s	100 µs	85% .. 90%	87,5%

Table 7: Recommended bit timing setup

The default bit-rates defined in table 7 can be setup via the core function `CpCoreBitrate()`. The supplied parameter for bit-rate selection are taken from the enumeration `CpBitrate_e` (refer to header file `canpie.h`).

Bitrate	*Definition in CpBitrate_e*
10 kBit/s	eCP_BITRATE_10K
20 kBit/s	eCP_BITRATE_20K
50 kBit/s	eCP_BITRATE_50K
100 kBit/s	eCP_BITRATE_100K
125 kBit/s	eCP_BITRATE_125K
250 kBit/s	eCP_BITRATE_250K
500 kBit/s	eCP_BITRATE_500K
800 kBit/s	eCP_BITRATE_800K
1 MBit/s	eCP_BITRATE_1M

Table 8: Definition for bit-rate values



If the pre-defined bit-rates do not meet the requirements, it is possible to setup the CAN bit-timing individually via the `CpCoreBittiming()` function.

3.5 CAN Statistic Information

Statistic information about a physical CAN interface can be gathered via the function `CpCoreStatistic()`. All counters are set to 0 upon initialisation of the CAN interface (`CpCoreDriverInit()`).

```
typedef struct CpStatistic_s{

     // Total number of received data & remote frames
     uint32_t     ulRcvMsgCount;

     // Total number of transmitted data & remote
     // frames
     uint32_t     ulTrmMsgCount;

     // Total number of state change / error events
     uint32_t     ulErrMsgCount;
}; CpStatistic_ts
```

3

3.6 Error Codes

All functions that may cause an error condition will return an error code. The CANpie error codes are within the value range from 0 to 127. The designer of the core functions might extend the error code table with hardware specific error codes, which must be in the range from 128 to 255.

Error Code	*Description*
eCP_ERR_NONE	No error occured
eCP_ERR_GENERIC	Reason is not specified
eCP_ERR_HARDWARE	Hardware failure
eCP_ERR_INIT_FAIL	CAN channel or buffer initialisation failed
eCP_ERR_INIT_READY	CAN channel or buffer already initialized
eCP_ERR_INIT_MISSING	CAN channel or buffer not initialized
eCP_ERR_RCV_EMPTY	Receive buffer empty
eCP_ERR_RCV_OVERRUN	Receive buffer overrun
eCP_ERR_TRM_FULL	Transmit buffer is full
eCP_ERR_CAN_MESSAGE	CAN message format is not valid
eCP_ERR_CAN_ID	identifier is not valid
eCP_ERR_CAN_DLC	data length code is not valid
eCP_ERR_FIFO_EMPTY	FIFO is empty (read operation)
eCP_ERR_FIFO_FULL	FIFO is full (write operation)
eCP_ERR_FIFO_SIZE	not enough memory for FIFO
eCP_ERR_FIFO_PARAM	Parameter of FIFO function mismatch
eCP_ERR_BUS_PASSIVE	CAN controller is in bus passive state
eCP_ERR_BUS_OFF	CAN controller is in bus off state
eCP_ERR_BUS_WARNING	CAN controller is in warning state
eCP_ERR_CHANNEL	channel number is out of range
eCP_ERR_REGISTER	register address out of range
eCP_ERR_BITRATE	bitrate is out of range / not supported
eCP_ERR_BUFFER	buffer index is out of range
eCP_ERR_PARAM	Parameter out of range
eCP_ERR_NOT_SUPPORTED	the function is not supported

Table 9: CANpie error codes

The error codes are defined in the header file `canpie.h` by the enumeration `CpErr_e`.

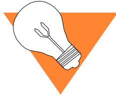
4. Core Functions

The core functions provide the direct interface to the CAN controller (hardware). Please note that due to hardware limitations certain functions may not be implemented. A call to an unsupported function will always return the error code `eCP_ERR_NOT_SUPPORTED`.

Function	*Description*
CpCoreBitrate()	Set the bit-rate of the CAN controller
CpCoreBufferConfig()	Initialize message buffer
CpCoreBufferGetData()	Get message data from buffer
CpCoreBufferGetDlc()	Get data length code from buffer
CpCoreBufferRelease()	Release messge buffer
CpCoreBufferSend()	Send message out of specified buffer
CpCoreBufferSetData()	Set message data
CpCoreBufferSetDlc()	Set data length code
CpCoreCanMode	Set the mode of CAN controller
CpCoreCanState()	Retrieve the mode of CAN controller
CpCoreDriverInit()	Initialize the CAN driver
CpCoreDriverRelease()	Stop the CAN driver
CpCoreFifoConfig()	Assign FIFO to message buffer
CpCoreFifoRead()	Read a CAN message from FIFO
CpCoreFifoRelease()	Release FIFO from message buffer
CpCoreFifoWrite()	Write a CAN message to FIFO
CpCoreHDI()	Read the Hardware Description Information (HDI structure)
CpCoreIntFunctions()	Install callback functions for different CAN controller interrupts
CpCoreStatistic()	Get statistical information

Table 10: CANpie core functions

4



Because the core functions are highly dependent on the hardware environment and the used operating system, the CANpie source package can only supply function bodies for these functions.

4.1 Deprecated Functions

The following functions are deprecated (CANpie version 2.00) and shall not be used for new implementations.

Function	*Description*
CpCoreAutobaud()	Start automatic bit-rate detection
CpCoreBaudrate()	Set the bit-rate of the CAN controller via predefined values
CpCoreBufferInit()	Initialize message buffer
CpCoreMsgRead()	Read CAN message
CpCoreMsgWrite()	Write CAN message

Table 11: Deprecated core functions

4.2 CpCoreBitrate

Syntax

```
CpStatus_tv CpCoreBitrate(
     CpPort_ts *      ptsPortV
     int32_t          slNomBitRateV,
     int32_t          slDatBitRateV)
```

Function

Set bit-rate of CAN controller

This function initializes the bit timing registers of a CAN controller to pre-defined values. The values are defined in the header file `canpie.h` (enumeration `CpBitrate_e`). Please refer to "Bittiming" on page 20 for a detailed description of common bit-timing values. For a classical CAN controller (or if bit-rate switching is not required) the parameter `slDatBitRateV` is set to `eCP_BITRATE_NONE`.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`slNomBitRateV`	Nominal bit-timing value
`slDatBitRateV`	Data phase bit-timing value

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoCanInit(void)
{
   CpPort_ts  tsCanPortT;   // logical CAN port


   memset(&tsCanPortT, 0, sizeof(CpPort_ts));
   CpCoreDriverInit(eCP_CHANNEL_1, &tsCanPortT, 0);

   //----------------------------------------------------
   // setup 500 kBit/s
   //
   CpCoreBitrate(&tsCanPortT,
                 eCP_BITRATE_500K,
                 eCP_BITRATE_NONE);


}
```

Example 3: Setup of bit-rate

4.3 CpCoreBufferConfig

Syntax

```
CpStatus_tv CpCoreBufferConfig(
    CpPort_ts *      ptsPortV
    uint8_t          ubBufferIdxV,
    uint32_t         ulIdentifierV,
    uint32_t         ulAcceptMaskV,
    uint8_t          ubControlV,
    uint8_t          ubDirectionV)
```

Function

Initialize a message buffer (mailbox)

This function allocates a message buffer in a CAN controller. The number of the message buffer inside the CAN controller is denoted via the parameter `ubBufferIdxV`. The first buffer starts at position `eCP_BUFFER_1`. The message buffer is allocated to the identifier value `ulIdentifierV`. If the buffer is used for reception (parameter `ubDirectionV` is `eCP_BUFFER_DIR_RCV`), the parameter `ulAcceptMaskV` is used for acceptance filtering. A message buffer can be released via the function `CpCoreBufferRelease()`. An allocated transmit buffer can be sent via the function `CpCoreBufferSend()`.

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`ulIdentifierV`	Identifier value
`ulAcceptMaskV`	Acceptance mask value
`ubFormatV`	Message format
`ubDirectionV`	Direction of message (receive or transmit) `eCP_BUFFER_DIR_RCV`: receive `eCP_BUFFER_DIR_TRM`: transmit

The parameter `ubFormatV` can have the following values:

Parameter 'ubFormatV'	Description
CP_MSG_FORMAT_CBFF	Classical CAN frame, Standard Identifier
CP_MSG_FORMAT_CEFF	Classical CAN frame, Extended Identifier
CP_MSG_FORMAT_FBFF	ISO CAN FD frame, Standard Identifier
CP_MSG_FORMAT_FEFF	ISO CAN FD frame, Extended Identifier

Table 12: Configuration of CAN message format

Return Value

Error code is defined by the CpErr_e enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value eCP_ERR_NONE.

Example

```
void DemoTransmitBufferConfig(CpPort_ts * ptsCanPortV)
{

   //------------------------------------------------------
   // set message buffer 1 as transmit buffer for classic
   // CAN frame with Standard Identifier 123h, DLC = 4
   //
   CpCoreBufferConfig(ptsCanPortV, eCP_BUFFER_1,
                      (uint32_t) 0x123,
                      CP_MASK_STD_FRAME,
                      CP_MSG_FORMAT_CBFF,
                      eCP_BUFFER_DIR_TRM);

   CpCoreBufferSetDlc(ptsCanPortV, eCP_BUFFER_1, 4);
}
```

Example 4: Allocation of a message buffer

4.4 CpCoreBufferGetData

Syntax

```
CpStatus_tv CpCoreBufferGetData(
    CpPort_ts *     ptsPortV
    uint8_t         ubBufferIdxV,
    uint8_t *       pubDestDataV,
    uint8_t         ubStartPosV,
    uint8_t         ubSizeV)
```

Function

Get data from message buffer

The functions copies `ubSizeV` data bytes from the CAN message buffer defined by `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The parameter `ubStartPosV` denotes the start position, the first data byte is at position 0. The destination buffer (pointer `pubDestDataV`) must have sufficient space for the data. The buffer has to be configured by `CpCoreBufferConfig()` in advance.

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`pubDestDataV`	Pointer to destination buffer
`ubStartPosV`	Start position
`ubSizeV`	Number of bytes to read

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoBufferGetData(CpPort_ts * ptsCanPortV)
{
   uint8_t  aubBufferT[8];   // temporary buffer

   //--------------------------------------------------
   // read 3 byte from message buffer 1,
   // start position is byte 0

   CpCoreBufferGetData(ptsCanPortV, eCP_BUFFER_1,
                       &aubBufferT[0],  // destination
                       0,               // start position
                       3);              // size
   .....

}
```

Example 5: Read CAN data of a message buffer

4.5 CpCoreBufferGetDlc

Syntax

```
CpStatus_tv CpCoreBufferGetDlc(
    CpPort_ts *     ptsPortV,
    uint8_t         ubBufferIdxV,
    uint8_t *       pubDlcV)
```

Function

Get DLC of specified buffer

This function retrieves the Data Length Code (DLC) of the specified buffer `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The parameter `pubDlcV` is a pointer to a memory location where the function will store the DLC value on success. The buffer has to be configured by `CpCoreBufferConfig()` in advance.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`pubDlcV`	Pointer to destination buffer for DLC

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoBufferGetDlc(CpPort_ts * ptsCanPortV)
{
   uint8_t  ubDlcT;   // temporary buffer

   //-------------------------------------------------
   // read DLC from message buffer 1,
   //
   CpCoreBufferGetDlc(ptsCanPortV, eCP_BUFFER_1,
                      &ubDlcT);
   .....

}
```

Example 6: Read DLC value of a message buffer

4.6 CpCoreBufferRelease

Syntax

```
CpStatus_tv CpCoreBufferRelease(
    CpPort_ts *     ptsPortV
    uint8_t         ubBufferIdxV)
```

Function

Release message buffer

The function releases the allocated message buffer specified by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. Both - reception and transmission - will be disabled on the specified message buffer afterwards.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoReleaseAllBuffers(CpPort_ts * ptsCanPortV)
{
   uint8_t  ubBufferIdxT;


   //------------------------------------------------------
   // release all message buffers
   //
   for (ubBufferIdxT = eCP_BUFFER_1;
        ubBufferIdxT < CP_BUFFER_MAX; ubBufferIdxT++ )
   {
      CpCoreBufferRelease(ptsCanPortV, ubBufferIdxT);
   }


}
```

Example 7: Release of all message buffers

4.7 CpCoreBufferSend

Syntax

```
CpStatus_tv CpCoreBufferSend(
     CpPort_ts *      ptsPortV
     uint8_t          ubBufferIdxV)
```

Function

Send message from message buffer

This function transmits a message from the specified message buffer `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The message buffer has to be configured as transmit buffer (`eCP_BUFFER_DIR_TRM`) by a call to `CpCoreBufferConfig()` in advance. A transmission request on a receive buffer will fail with the return code `eCP_ERR_INIT_FAIL`.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoBufferSend(CpPort_ts * ptsCanPortV)
{
   CpStatus_tv  tvResultT;

   //------------------------------------------------------
   // try to send message
   //
   tvResultT = CpCoreBufferSend(ptsCanPortV, eCP_BUFFER_1);
   switch (tvResultT)
   {
      case eCP_ERR_NONE:
         // message was send
         break;

      case eCP_ERR_INIT_MISSING:
         // message was not send: buffer not initialised
         break;

      case eCP_ERR_TRM_FULL:
         // message was not send, transmit buffer busy
         break;

      default:
         // other error
         break;
   }
}
```

Example 8: Transmission of message buffer

4.8 CpCoreBufferSetData

Syntax

```
CpStatus_tv CpCoreBufferSetData(
     CpPort_ts *      ptsPortV
     uint8_t          ubBufferIdxV,
     uint8_t *        pubSrcDataV,
     uint8_t          ubStartPosV,
     uint8_t          ubSizeV)
```

Function

Set data in message buffer

This function copies `ubSizeV` data bytes into the message buffer defined by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The parameter `ubStartPosV` denotes the start position, the first data byte is at position 0. The message buffer has to be configured by `CpCoreBufferConfig()` in advance.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`pubSrcDataV`	Pointer to source buffer
`ubStartPosV`	Start position
`ubSizeV`	Number of bytes to write

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

```
uint8_t aubDataT[8];      // buffer for 8 bytes

aubDataT[0] = 0x11;    // byte 0: set to 11hex
aubDataT[1] = 0x22;    // byte 1: set to 22hex

//--- copy the data to message buffer 1 ----------
CpCoreBufferSetData(ptsCanPortV, eCP_BUFFER_1,
                    &aubDataT[0], 0, 2);

//--- send this message -------------------------
CpCoreBufferSend(ptsCanPortV, eCP_BUFFER_1);
```

Example 9: Manipulation of data in message buffer

4.9 CpCoreBufferSetDlc

Syntax

```
CpStatus_tv CpCoreBufferSetDlc(
     CpPort_ts *      ptsPortV
     uint8_t          ubBufferIdxV,
     uint8_t          ubDlcV)
```

Function

Set Data Length Code (DLC) of specified message buffer

This function sets the Data Length Code (DLC) of the specified message buffer `ubBufferIdxV`. The DLC value `ubDlcV` must be in the range from 0 to 8 for Classical CAN frames and 0 to 15 for ISO CAN FD frames.

An invalid DLC value is rejected with the return value `eCP_ERR_CAN_DLC`. The message buffer has to be configured by a call to `CpCoreBufferConfig()` in advance.

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`ubDlcV`	DLC value

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.10 CpCoreCanMode

Syntax

```
CpStatus_tv CpCoreCanMode(
     CpPort_ts *     ptsPortV
     uint8_t         ubModeV)
```

Function

Set operating mode of CAN controller

This function changes the operating mode of the CAN controller. Possible values for mode are defined in the `CpMode_e` enumeration. At least the modes `eCP_MODE_INIT` and `eCP_MODE_OPERATION` shall be supported. Other modes depend on the capabilities of the CAN controller.

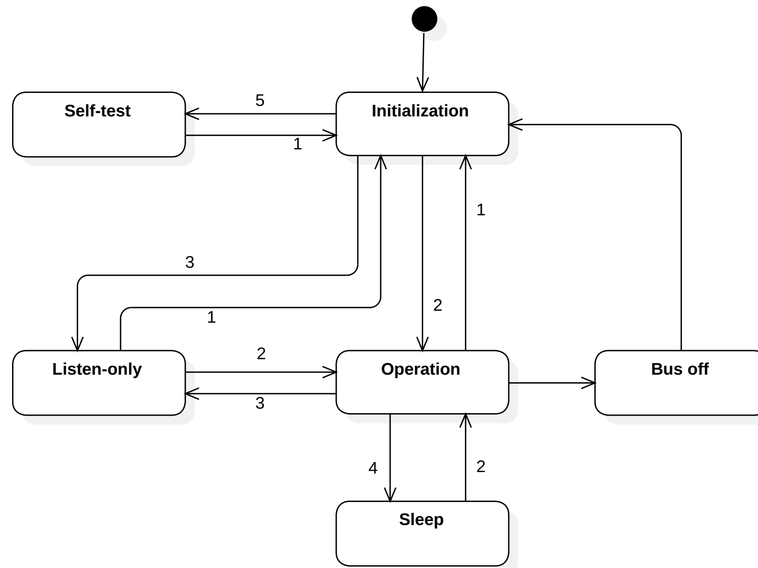


Figure 3: CAN controller FSA

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubModeV`	New CAN controller mode

Transition	*Parameter "ubModeV"*	*Description*
1	`eCP_MODE_INIT`	set controller into 'Initialization' mode
2	`eCP_MODE_OPERATION`	set controller into 'Operation' mode
3	`eCP_MODE_LISTEN_ONLY`	set controller into 'Listen-only' mode
4	`eCP_MODE_SLEEP`	set controller into 'Sleep' (power-down) mode
5	`eCP_MODE_SELF_TEST`	set controller into 'Self-test' mode

Table 13: Value definition for parameter `ubModeV`

Return Value

Error code is defined by the CpErr_e enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value eCP_ERR_NONE.

Example

```
void DemoCanSelfTest(void)
{
   CpPort_ts  tsCanPortT;   // logical CAN port


   //--------------------------------------------------
   // setup the CAN controller / open a physical CAN
   // port
   //
   memset(&tsCanPortT, 0, sizeof(CpPort_ts));

   CpCoreDriverInit(eCP_CHANNEL_1, &tsCanPortT, 0);

   //--------------------------------------------------
   // setup 500 kBit/s
   //
   CpCoreBitrate(&tsCanPortT,
                 eCP_BITRATE_500K,
                 eCP_BITRATE_NONE);

   //--------------------------------------------------
   // start CAN self-test
   //
   if (CpCoreCanMode(&tsCanPortT,
                     eCP_MODE_SELF_TEST) == eCP_ERR_NONE)
   {
      //.. run self-test


   }
}
```

Example 10: Setting the mode of the CAN FSA

4.11 CpCoreCanState

Syntax

```
CpStatus_tv CpCoreCanState(
    CpPort_ts *      ptsPortV
    CpState_ts *     ptsStateV)
```

Function

Retrieve state of CAN controller

This function retrieves the present state of the CAN controller. The parameter `ptsStateV` is a pointer to a memory location where the function will store the state. The value of the structure element `CpState_ts::ubCanErrState` is defined by the `CpState_e` enumeration. The value of the structure element `CpState_ts::ubCanErrType` is defined by the `CpErrType_e` enumeration.

4

Parameters

`ptsPortV` Pointer to CAN port structure

`ptsStateV` Pointer to CAN state structure

Possible state values	*Description*
`eCP_STATE_INIT`	CAN controller is in Initialization state
`eCP_STATE_SLEEPING`	CAN controller is in Sleep mode
`eCP_STATE_BUS_ACTIVE`	CAN controller is active, no errors
`eCP_STATE_BUS_WARN`	Warning level is reached
`eCP_STATE_BUS_PASSIVE`	CAN controller is error passive
`eCP_STATE_BUS_OFF`	CAN controller went into Bus-Off
`eCP_STATE_PHY_FAULT`	General failure of physical layer detected
`eCP_STATE_PHY_H`	Fault on CAN-H (Low Speed CAN)
`eCP_STATE_PHY_L`	Fault on CAN-L (Low Speed CAN)

Table 14: Possible states of CAN controller

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.12 CpCoreDriverInit

Syntax

```
CpStatus_tv CpCoreDriverInit(
     uint8_t          ubPhyIfV,
     CpPort_ts *      ptsPortV,
     uint8_t          ubConfigV)
```

Function

Initialize the CAN driver

The functions opens the physical CAN interface defined by the parameter `ubPhyIfV`. The value for `ubPhyIfV` is taken from the enumeration `CpChannel_e`. The function sets up the field members of the CAN port structure `CpPort_ts`. The parameter `ptsPortV` is a pointer to a memory location where structure `CpPort_ts` is stored. An opened CAN port must be closed via the `CpCoreDriverRelease()` function.

Parameters

`ubPhyIfV`	CAN channel of the hardware
`ptsPortV`	Pointer to CAN port structure
`ubConfigV`	Reserved for future enhancement

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoCanInit(void)
{
   CpPort_ts  tsCanPortT;   // logical CAN port


   //----------------------------------------------------
   // setup the CAN controller / open a physical CAN
   // port
   //
   memset(&tsCanPortT, 0, sizeof(CpPort_ts));

   CpCoreDriverInit(eCP_CHANNEL_1, &tsCanPortT, 0);

   //----------------------------------------------------
   // setup 500 kBit/s
   //
   CpCoreBitrate(&tsCanPortT,
                 eCP_BITRATE_500K,
                 eCP_BITRATE_NONE);

   //----------------------------------------------------
   // start CAN operation
   //
   CpCoreCanMode(&tsCanPortT, eCP_MODE_OPERATION);


   //.. now we are operational

}
```

Example 11: Initialization process of the CAN interface

4.13 CpCoreDriverRelease

Syntax

```
CpStatus_tv CpCoreDriverRelease(
    CpPort_ts *     ptsPortV)
```

Function

Release the CAN driver

The function closes a CAN port. The parameter `ptsPortV` is a pointer to a memory location where structure `CpPort_ts` is stored. The implementation of this function is dependent on the operating system. Typical tasks might be:

- clear the interrupt vector
- close all open paths to the hardware

4

Parameters

`ptsPortV`	Pointer to CAN port structure

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.14 CpCoreFifoConfig

Syntax

```
CpStatus_tv CpCoreFifoConfig(
    CpPort_ts *      ptsPortV
    uint8_t          ubBufferIdxV,
    CpFifo_ts *      ptsFifoV)
```

Function

Assign FIFO to a message buffer

This function assigns a FIFO to a message buffer defined by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The buffer has to be configured by `CpCoreBufferConfig()` in advance. The parameter `ptsFifoV` is a pointer to a memory location where a FIFO has been initialized using the `CpFifoInit()` function.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`ptsFifoV`	Pointer to FIFO

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
#define  FIFO_RCV_SIZE     32

static CpFifo_ts        tsFifoRcvS;
static CpCanMsg_ts      atsCanMsgRcvS[FIFO_RCV_SIZE];


void DemoFifoConfig(CpPort_ts * ptsCanPortV)
{
   //-------------------------------------------------------
   // set message buffer 2 as receive buffer for classic
   // CAN frame with identifier 180h .. 18Fh
   //
   CpCoreBufferConfig(ptsCanPortV, eCP_BUFFER_2,
                      (uint32_t) 0x180,
                      (uint32_t) 0x7F0,   // mask
                      CP_MSG_FORMAT_CBFF,
                      eCP_BUFFER_DIR_RCV);

   CpFifoInit(&tsFifoRcvS, &atsCanMsgRcvS[0], FIFO_RCV_SIZE);
   CpCoreFifoConfig(&ptsCanPortV, eCP_BUFFER_2, &tsFifoRcvS);

}
```

Example 12: Configuration of a FIFO

4.15 CpCoreFifoRead

Syntax

```
CpStatus_tv CpCoreFifoRead(
    CpPort_ts *      ptsPortV,
    uint8_t          ubBufferIdxV,
    CpCanMsg_ts *    ptsCanMsgV,
    uint32_t *       pulMsgCntV);
```

Function

Read CAN message from FIFO

This function reads CAN messages from a receive FIFO defined by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The FIFO has to be configured by `CpCoreFifoConfig()` in advance. The parameter `ptsCanMsgV` is a pointer to the application buffer as array of `CpCanMsg_ts` objects to store the received CAN messages. The parameter `pulMsgCntV` is a pointer to a memory location which has to be initialized before the call to the size of the buffer referenced by `ptsCanMsgV` as multiple of `CpCanMsg_ts` objects. Upon return, the driver has stored the number of messages copied into the application buffer into this parameter.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`ptsCanMsgV`	Pointer to a CAN message structure
`pulMsgCntV`	Pointer to message count variable

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoFifoRead(CpPort_ts * ptsCanPortV)
{
   CpCanMsg_ts  tsCanMsgReadT;
   uint32_t     ulMsgCntT;

   //----------------------------------------------------
   // try to read one CAN message
   //
   ulMsgCntT = 1;
   CpCoreFifoRead(ptsCanPortV, eCP_BUFFER_2,
                  &tsCanMsgReadT,
                  &ulMsgCntT);


}
```

Example 13: Read message from FIFO

4.16 CpCoreFifoRelease

Syntax

```
CpStatus_tv CpCoreFifoRelease(
     CpPort_ts *      ptsPortV
     uint8_t          ubBufferIdxV)
```

Function

Release FIFO from message buffer

This function releases an assigned FIFO from a message buffer defined by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The FIFO has to be configured by `CpCoreFifoConfig()` in advance.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.17 CpCoreFifoWrite

Syntax

```
CpStatus_tv CpCoreFifoWrite(
    CpPort_ts *      ptsPortV
    uint8_t          ubBufferIdxV,
    CpCanMsg_ts *    ptsCanMsgV,
    uint32_t *       pulMsgCntV)
```

Function

Transmit a CAN message

This function writes CAN messages to a transmit FIFO defined by the parameter `ubBufferIdxV`. The first message buffer starts at the index `eCP_BUFFER_1`. The FIFO has to be configured by `CpCoreFifoConfig()` in advance. The parameter `ptsCanMsgV` is a pointer to the application buffer as array of `CpCanMsg_ts` objects which contain the CAN messages that should be transmitted.

The parameter `pulMsgCntV` is a pointer to a memory location which has to be initialized before the call to the size of the buffer referenced by `ptsCanMsgV` as multiple of `CpCanMsg_ts` objects. Upon return, the driver has stored the number of messages transmitted successfully into this parameter.

Parameters

`ptsPortV`	Pointer to CAN port structure
`ubBufferIdxV`	Index of message buffer
`ptsCanMsgV`	Pointer to a CAN message structure
`pulMsgCntV`	Pointer to message count variable

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.18 CpCoreHDI

Syntax

```
CpStatus_tv CpCoreHDI(
    CpPort_ts *      ptsPortV
    CpHdi_ts *       ptsHdiV)
```

Function

Get Hardware Description Information

This function retrieves information about the CAN interface. The parameter `ptsHdiV` is a pointer to a memory location where the function will store the information. Please refer to “Hardware Description Interface” on page 16 for details on the structure `CpHdi_ts`.

4

Parameters

`ptsPortV`	Pointer to CAN port structure
`ptsHdiV`	Pointer to the `CpHdi_ts` structure

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value `eCP_ERR_NONE`.

4.19 CpCoreIntFunctions

Syntax

```
CpStatus_tv CpCoreIntFunctions(
   CpPort_ts *       ptsPortV,
   CpRcvHandler_Fn   pfnRcvHandlerV,
   CpTrmHandler_Fn   pfnTrmHandlerV,
   CpErrHandler_Fn   pfnErrHandlerV)
```

Function

Install callback functions

This function will install three different callback routines in the interrupt handler. If you do not want to install a callback for a certain interrupt condition the parameter must be set to NULL.

4

The callback functions for receive and transmit interrupt have the following syntax:

```
    uint8_t Handler(CpCanMsg_ts * ptsCanMsgV,
                    uint8_t ubBufferIdxV)
```

The callback function for the CAN status / error interrupt has the following syntax:

```
    uint8_t Handler(CpState_ts * ubStateV)
```

Parameters

`ptsPortV`	Pointer to CAN port structure
`pfnRcvHandlerV`	Pointer to callback function for receive interrupt
`pfnTrmHandlerV`	Pointer to callback function for transmit interrupt
`pfnErrHandlerV`	Pointer to callback function for error interrupt

Return Value

Error code is defined by the CpErr_e enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value eCP_ERR_NONE.

Example

```
uint8_t MyCanReceive(CpCanMsg_ts * ptsCanMsgV,
                     uint8_t ubBufferIdxV)
{
   switch( CpMsgGetIdentifier(ptsCanMsgV) )
   {
      case 0x022:
         // do something with ID 0x022
         break;
   }
}

main()
{
   //....
   CpCoreIntFunctions(&tsCanPortT, MyReceiveFunc, 0L, 0L);
   //...
}
```

Example 14: Install a callback for receive interrupt

4

4.20 CpCoreStatistic

Syntax

```
CpStatus_tv CpCoreStatistic(
    CpPort_ts *     ptsPortV,
    CpStatistic_ts *ptsStatsV )
```

Function

Get statistic information from CAN controller

This function copies CAN statistic information to the structure pointed by ptsStatsV.

Parameters

ptsPortV	Pointer to CAN port structure
ptsStatsV	Pointer to CAN statistic structure

4

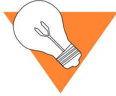
Return Value

Error code is defined by the CpErr_e enumeration (refer to table 9 on page 22). If no error occurred, the function will return the value eCP_ERR_NONE.

4

5. CAN Message Functions

Access to the members of the CAN message structure ***CpCanMsg_s*** shall be performed via macros or functions calls. This ensures - upon change of the CAN message structure - that the application does not have to be adapted.



The CAN message functions are implemented as conventionell functions as well as macros. The symbol **`CP_CAN_MSG_MACRO`** defines which implementation is used.

Function	***Description***
CpMsgGetData()	Read CAN message payload
CpMsgGetDlc()	Read CAN message DLC
CpMsgGetIdentifier()	Read CAN message identifier
CpMsgGetTime()	Read CAN message time-stamp
CpMsgInit()	Initialize CAN message structure
CpMsgSetData()	Write CAN message payload
CpMsgSetDlc()	Write CAN message DLC
CpMsgSetIdentifier()	Write CAN message identifier
CpMsgSetTime()	Write CAN message time-stamp

Table 15: Functions for CAN message manipulation

5.1 CpMsgGetData

Syntax

```
uint8_t CpMsgGetData(
     CpCanMsg_ts *    ptsCanMsgV,
     uint8_t          ubPosV)
```

Function

Read data bytes from CAN message

This function retrieves a single data byte of a CAN message. The parameter **ubPosV** must be within the range from 0 to 7 for Classical CAN frames and from 0 to 64 for ISO CAN FD frames.

Parameters

ptsCanMsgV Pointer to CAN message structure

ubPosV Zero based index of byte position

5

Return Value

Data value at specified position.

```
void MyDataRead(CpCanMsg_ts * ptsCanMsgV)
{
   uint8_t ubByte0T;
   ....

   //--------------------------------------------------
   // read first data byte from CAN message, check
   // the data length code (DLC) first
   //
   if( CpMsgGetDlc(ptsCanMsgV) > 0 )
   {
      ubByte0T = CpMsgGetData(ptsCanMsgV, 0);

      ....
   }

   ....

}
```

Example 15: Get data byte from CAN message structure

5.2 CpMsgGetDlc

Syntax

```
uint8_t CpMsgGetDlc(
     CpCanMsg_ts *   ptsCanMsgV)
```

Function

Get DLC value from CAN message

This function returns the data length code (DLC) of a CAN message. Refer to table 6 for conversion between DLC value and payload size.

Parameters

`ptsCanMsgV` Pointer to CAN message structure

Return value

DLC value of CAN message

```
void MyDataRead(CpCanMsg_ts * ptsCanMsgV)
{
   uint8_t ubByte0T;
   ....

   //-------------------------------------------------
   // read first data byte from CAN message, check
   // the data length code (DLC) first
   //
   if( CpMsgGetDlc(ptsCanMsgV) == 8 )
   {
      ubByte0T = CpMsgGetData(ptsCanMsgV, 0);

      ....
   }

   ....

}
```

Example 16: Check data length code from CAN message structure

5.3 CpMsgGetIdentifier

Syntax

```
uint32_t CpMsgGetIdentifier(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Get identifier value

This function retrieves the value for the identifier of a CAN frame. The frame format of the CAN message can be tested with the `CpMsgIsExtended()` function.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

Identifier value

5

```
void MyMessageRead(CpCanMsg_ts * ptsCanMsgV)
{
   uint32_t ubExtIdT;
   ....

   //-------------------------------------------------
   // read identifier from CAN message
   //
   if( CpMsgIsExtended(ptsCanMsgV) == true )
   {
      ubExtIdT = CpMsgGetIdentifier(ptsCanMsgV);

      ....
   }

   ....

}
```

Example 17: Get identifier value

5.4 CpMsgInit

Syntax

```
void CpMsgInit(
    CpCanMsg_ts *    ptsCanMsgV,
    uint8_t          ubFormatV)
```

Function

Initialise message structure

This function sets the identifier field and the DLC field of a CAN message structure to 0. The parameter **ubFormatV** defines the frame format. Possible value are defined by table 12, “Configuration of CAN message format,” on page 26.

The contents of the data field and all other optional fields (time-stamp, user, message marker) are not altered.

Parameters

ptsCanMsgV Pointer to CAN message structure

ubFormatV Frame format

5

Return value

None

```
void MyMessageInit(CpCanMsg_ts * ptsCanMsgV)
{
   uint32_t ulExtIdT = 0x01FFEE01;


   //--------------------------------------------------
   // setup ISO CAN FD frame with extended identifier
   //
   CpMsgInit(ptsCanMsgV, CP_MSG_FORMAT_FEFF);
   CpMsgSetIdentifier(ptsCanMsgV, ulExtIdT);

   ...

}
```

Example 18: Initialise CAN message

5.5 CpMsgIsExtended

Syntax

```
bool_t CpMsgIsExtended(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Test for extended frame format

This function checks the frame format. If the message is a base frame format (11 bit identifier) the value **`false`** is returned. If the message is an extended frame format the value **`true`** is returned.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

`true` on extended frame format, **`false`** on standard frame format

5

```
void MyMessageRead(CpCanMsg_ts * ptsCanMsgV)
{
   uint32_t ubExtIdT;
   ....

   //------------------------------------------------
   // read identifier from CAN message
   //
   if( CpMsgIsExtended(ptsCanMsgV) == true )
   {
      ubExtIdT = CpMsgGetIdentifier(ptsCanMsgV);

      ....
   }

   ....

}
```

Example 19: Test frame format

5.6 CpMsgIsRemote

Syntax

```
uint8_t CpMsgIsRemote(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Test for remote frame

This function checks if the Remote Transmission bit (RTR) is set. If the RTR bit is set, the function will return **`true`**, otherwise **`false`**.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

`true` on remote frame, **`false`** on data frame.

5.7 CpMsgSetData

Syntax

```
void CpMsgSetData(
    CpCanMsg_ts *   ptsCanMsgV,
    uint8_t         ubPosV,
    uint8_t         ubValueV)
```

Function

Set data bytes to CAN message

This function sets the data of a CAN message. The parameter **ubPosV** must be within the range 0 .. 7 for Classical CAN frames. For ISO CAN FD frames the valid range is 0 .. 63.

Parameters

ptsCanMsgV	Pointer to CAN message structure
ubPosV	Zero based index of byte position
ubValueV	Data value for CAN message

5

Return value

None

5.8 CpMsgSetDlc

Syntax

```
void CpMsgSetDlc(
     CpCanMsg_ts *   ptsCanMsgV,
     uint8_t         ubDlcV)
```

Function

Set DLC value of CAN message

This function converts the number of bytes that are valid inside the data field to a data length code (DLC) value. For CAN frames the DLC value is equal to the number of bytes in the data field. For ISO CAN FD frames the DLC value is converted according to table 6.

Parameters

ptsCanMsgV Pointer to CAN message structure

ubSizeV DLC value of CAN payload

Return value

None

5

```
CpCanMsg_ts  tsMyCanMsgT;  // temporary CAN message struct.

//---------------------------------------------------
// initialize message and setup CAN-ID = 100h and DLC = 4
CpMsgInit(&tsMyCanMsgT, CP_MSG_FORMAT_CBFF);
CpMsgSetStdId(&tsMyCanMsgT, 0x0100);    // set ID = 0x0100
CpMsgSetDlc(&tsMyCanMsgT, 4);           // set DLC = 4
```

Example 20: Setup the data length code

5.9 CpMsgSetIdentifier

Syntax

```
void CpMsgSetIdentifier(
    CpCanMsg_ts *    ptsCanMsgV,
    uint32_t         ulIdentifierV)
```

Function

Set identifier value

This function sets the identifier value for a CAN frame. The parameter **`ulIdentifierV`** is truncated to a 11-bit value (AND operation with **`CP_MASK_STD_FRAME`**) when the message uses base frame format. The parameter **`ulIdentifierV`** is truncated to a 29-bit value (AND operation with **`CP_MASK_EXT_FRAME`**) when the message uses extended frame format.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure
`ulIdentifierV`	Identifier value

5

Return value

None

```
void MyMessageInit(CpCanMsg_ts * ptsCanMsgV)
{
   uint32_t ulExtIdT = 0x01FFEE01;


   //--------------------------------------------------
   // setup ISO CAN FD frame with extended identifier
   //
   CpMsgInit(ptsCanMsgV, CP_MSG_FORMAT_FEFF);
   CpMsgSetIdentifier(ptsCanMsgV, ulExtIdT);

   ...

}
```

Example 21: Set CAN message identifier

5.10 CpMsgSetRemote

Syntax

```
void CpMsgSetRemote(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Set RTR bit

This function sets the remote transmission bit (RTR) in the CAN message structure.

Parameters

`ptsCanMsgV` Pointer to CAN message structure

Return value

None

5

5

A LGPL LICENSE

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

   0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

   1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

B Index

B

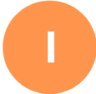
bit-rate
  function call **25**

C

CpCoreBitrate **25**
CpCoreBufferConfig **26**
CpCoreBufferGetData **28**
CpCoreBufferGetDlc **29**
CpCoreBufferRelease **30**
CpCoreBufferSend **31**
CpCoreBufferSetData **32**
CpCoreBufferSetDlc **33**
CpCoreCanMode **34**
CpCoreCanState **36**
CpCoreDriverInit **37**
CpCoreDriverRelease **39**
CpCoreFifoConfig **40**
CpCoreFifoRead **41**
CpCoreFifoRelease **42**
CpCoreFifoWrite **43**
CpCoreHDI **44**
CpCoreRegWrite **47**

S

Structure
  CpCanMsg_s **18**
  CpHdi_ts **16**


I

MicroControl reserves the right to modify this manual and/or product described herein without further notice. Nothing in this manual, nor in any of the data sheets and other supporting documentation, shall be interpreted as conveying an express or implied warranty, representation, or guarantee regarding the suitability of the products for any particular purpose. MicroControl does not assume any liability or obligation for damages, actual or otherwise of any kind arising out of the application, use of the products or manuals.

The products described in this manual are not designed, intended, or authorized for use as components in systems intended to support or sustain life, or any other application in which failure of the product could create a situation where personal injury or death may occur.

© 2017 MicroControl GmbH & Co. KG, Troisdorf

MicroControl GmbH & Co. KG
Junkersring 23
D-53844 Troisdorf
Fon: +49 / 2241 / 25 65 9 - 0
Fax: +49 / 2241 / 25 65 9 - 11
http://www.microcontrol.net