CANpie FD

User manual

Version 3.08

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

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

1.2 Abbreviations

BRS	Bit rate switch
CAN	Controller Area Network
CAN FD	CAN with flexible data rate
CAN-ID	CAN identifier
CBFF	Classical base frame format
CEFF	Classical extended frame format
CRC	Cyclic redundancy check
DLC	Data length code
ESI	Error state indicator
FBFF	FD base frame format
FEFF	FD extended frame format
FDF	FD format indicator
FSA	Finite state automaton
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 licensed under the **Apache License 2.0**, the complete license text can be found as appendix to this manual.

1.5 Source Code Repository

The source code of the CANpie FD project is available at:
https://github.com/canpie/CANpie

The HTML documentation of the CANpie FD project is available here:
https://canpie.github.io

1.6 Document History

Version	*Date*	*Description*
3.00 WD	01.12.2016	Work draft
3.00	13.04.2017	Release version
3.02	18.12.2017	- Change license conditions - Update CAN message structure
3.04	24.09.2018	- Add source code repository - Add example code in chapter 2.6 - Add chapter 2.7 (FIFO) - Add chapter 3.7 (CAN controller state) - Update function description of chapter 4.3 - Update function description of chapter 4.19
3.06	10.05.2019	- Extend structure CpHdi_s in chapter 3.2 - Add functions for DLC conversion
3.08	24.06.2020	- Add RPC in CAN message structure - Update function description in chapter 5

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.

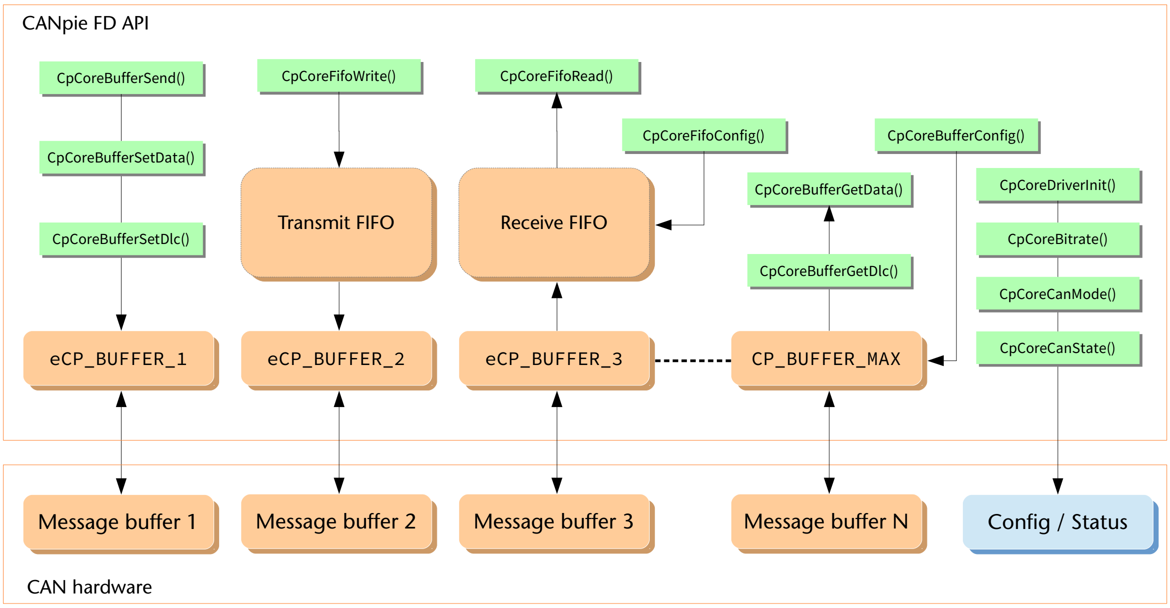

Figure 1: CANpie Structure

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 20).

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).

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 File Structure

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

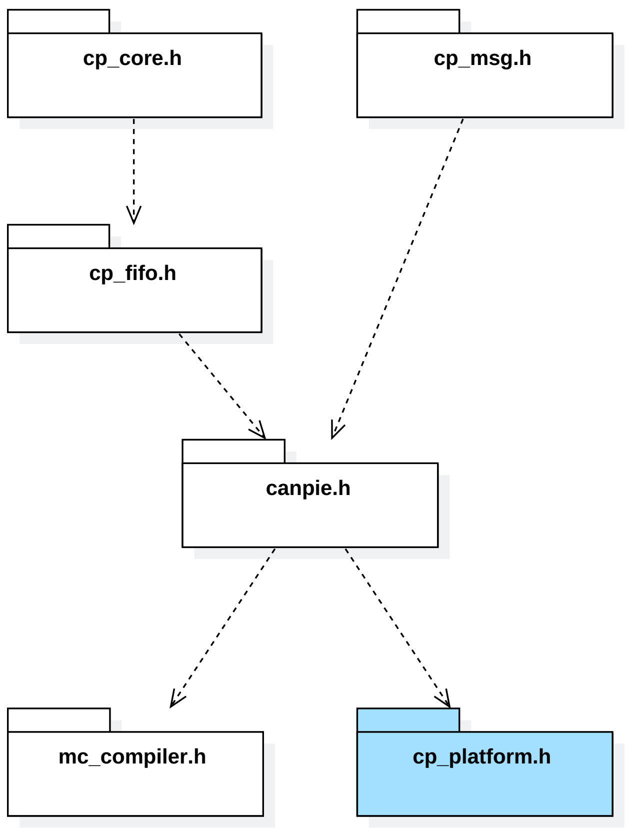

Figure 2: Include dependency graph

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
mc_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 2: CANpie files

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>
Enumerations	eCP_<name>
Error Codes	eCP_ERR_<name>

Table 3: Naming conventions

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

2.4 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 `mc_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 4: Data Type definitions

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

2.6 Initialisation 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()

```
CpPort_ts  tsCanPortG;   // logical CAN port


void MyCanInit(void)
{
   //--------------------------------------------------
   // clear physical CAN port structure
   //
   memset(&tsCanPortG, 0, sizeof(CpPort_ts));


   //--------------------------------------------------
   // Initialise physical CAN port
   //
   CpCoreDriverInit(eCP_CHANNEL_1, &tsCanPortG, 0);

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

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


   //--------------------------------------------------
   // CAN controller is error active now, initialisation
   // of message buffers is still missing

}
```

Example 1: Initialisation 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.

2.6.1 Configure buffer for message reception

In order to receive CAN data frames one or more CAN message buffers need to be configured for reception. The following example shows how to receive CAN data frames using the identifier values 211_h and $18EEFF00_h$ (both classical frame format).


```
void ReceiveBufferSetup(CpPort_ts * ptsCanPortV)
{
   //-----------------------------------------------------
   // set message buffer 2 as receive buffer for classical
   // CAN frame with Standard Identifier 211h
   //
   CpCoreBufferConfig(ptsCanPortV,
                      eCP_BUFFER_2,
                      (uint32_t) 0x211,
                      CP_MASK_STD_FRAME,
                      CP_MSG_FORMAT_CBFF,
                      eCP_BUFFER_DIR_RCV);

   //-----------------------------------------------------
   // set message buffer 3 as receive buffer for classical
   // CAN frame with Extended Identifier 18EEFF00h
   //
   CpCoreBufferConfig(ptsCanPortV,
                      eCP_BUFFER_3,
                      (uint32_t) 0x18EEFF00,
                      CP_MASK_EXT_FRAME,
                      CP_MSG_FORMAT_CEFF,
                      eCP_BUFFER_DIR_RCV);

}

uint8_t MyCanReceive(CpCanMsg_ts * ptsCanMsgV,
                     uint8_t ubBufferIdxV)
{
   switch(ubBufferIdxV)
   {
      case eCP_BUFFER_2:
         // do something with standard frame, ID 0x211
         break;
      case eCP_BUFFER_3:
         // do something with extended frame, ID 0x18EEFF00
         break;
    }
}

void MyCanInit(void)
{
   //....
   ReceiveBufferSetup(&tsCanPortG);
   CpCoreIntFunctions(&tsCanPortG,
                      MyCanReceive,
                      (CpTrmHandler_Fn) 0L,
                      (CpTrmHandler_Fn) 0L);
   //...
}
```

Example 2: Reception of CAN messages

2.6.2 Configure buffer for message transmission

In order to transmit CAN data frames one or more CAN message buffers need to be configured for transmission. The following example shows how to transmit a CAN data frames using the identifier value 123_h (classical frame format).

```
void TransmitBufferSetup(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);
}

void MyCanInit(void)
{
   //....
   TransmitBufferSetup(&tsCanPortG);

   //-----------------------------------------------------
   // Transmit message buffer 1
   //
   CpCoreBufferSend(ptsCanPortV, eCP_BUFFER_1);

   //...
}
```

Example 3: Transmission of CAN message

The function `CpCoreBufferConfig()` initialises the DLC value of a message buffer to 0, a subsequent call of `CpCoreBufferSetDlc()` is necessary to change the default value.

Once an identifier value has been assigned to a message buffer for transmission it can not be altered afterwards. Only the payload of the message buffer can be modified using `CpCoreBufferSetDlc()` and `CpCoreBufferSetData()`.

2.7 Working with a FIFO

A FIFO of arbitrary length can be assigned to any message buffer. The direction of the FIFO (either reception or transmission) is defined by the configuration of the message buffer.

Using a FIFO for a specific message buffer will disable callback functions (refer to “CpCoreIntFunctions” on page 49) for that message buffer.

2.7.1 Configure a FIFO for message reception

The following example code shows how to receive CAN data frames using the identifier range 180_h to $18F_h$. The identifier range is configured by setting an acceptance mask value of $7F0_h$. The receive FIFO is initialised with a maximum size of 32 CAN message objects.

```
#define  FIFO_RCV_SIZE     32

static CpFifo_ts         tsFifoRcvS;
static CpCanMsg_ts       atsCanMsgRcvS[FIFO_RCV_SIZE];


void ReceiveFifoConfig(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 4: Configuration of a FIFO for reception

Once the receive FIFO is configured messages can be read calling the function `CpCoreFifoRead()`.

2

2.7.2 Configure a FIFO for message transmission

The following example code shows how to transmit CAN data frames using a FIFO. Please note that both parameters - identifier value and acceptance mask value - of the function CpCoreBufferConfig() are ignored by subsequent calls of CpCoreFifoWrite().

```
#define  FIFO_TRM_SIZE     16

static CpFifo_ts        tsFifoTrmS;
static CpCanMsg_ts      atsCanMsgTrmS[FIFO_TRM_SIZE];


void TransmitFifoConfig(CpPort_ts * ptsCanPortV)
{
   //------------------------------------------------------
   // set message buffer 6 as transmit buffer for classic
   // CAN frames
   //
   CpCoreBufferConfig(ptsCanPortV, eCP_BUFFER_6,
                      (uint32_t) 0x000,   // ignored by FIFO
                      (uint32_t) 0x7FF,   // ignored by FIFO
                      CP_MSG_FORMAT_CBFF,
                      eCP_BUFFER_DIR_TRM);

   CpFifoInit(&tsFifoTrmS, &atsCanMsgTrm[0], FIFO_TRM_SIZE);
   CpCoreFifoConfig(&ptsCanPortV, eCP_BUFFER_6, &tsFifoTrmS);

}
```

Example 5: Configuration of a FIFO for transmission

Once the transmit FIFO is configured messages can be written by calling the function CpCoreFifoWrite().

```
void DemoFifoWrite(CpPort_ts * ptsCanPortV)
{
   CpCanMsg_ts  tsCanMsgT;
   uint32_t     ulMsgCntT;

   //------------------------------------------------------
   // setup classical CAN frame with standard identifier
   //
   CpMsgInit(&tsCanMsgT, CP_MSG_FORMAT_CBFF);
   CpMsgSetIdentifier(&tsCanMsgT, 0x123);
   CpMsgSetDlc(&tsCanMsgT, 0);          // set DLC = 0

   //------------------------------------------------------
   // put CAN message to FIFO
   //
   ulMsgCntT = 1;
   CpCoreFifoWrite(ptsCanPortV, eCP_BUFFER_6,
                   &tsCanMsgT,
                   &ulMsgCntT);
}
```

Example 6: Write message to FIFO

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 {

   /*!   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 7: 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    ubDriverMajor;
      uint8_t    ubDriverMinor;
      uint8_t    ubReserved[1];
      uint32_t   ulTimeStampRes;
      uint32_t   ulCanClock;
      uint32_t   ulBitRateMin;
      uint32_t   ulBitRateMax;
      uint32_t   ulNomBitRate;
      uint32_t   ulDatBitRate;
} CpHdi_ts;
```

Example 8: Structure for hardware description

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

Version Major

The element `ubVersionMajor` defines the major version number of the CANpie FD API release. The current number of this release is 3.

Version Minor

The element `ubVersionMinor` defines the minor version number of the CANpie FD API release. The current number of this release is 8.

CAN Features

The element `ubCanFeatures` defines the capabilities of the CAN controller. Reserved bits (res.) are read as 0.

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

Driver Features

The element `ubDriverFeatures` defines the capabilities of the software driver. Reserved bits (res.) are read as 0.

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

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

Driver Major The element `ubDriverMajor` defines the major version number of the CANpie FD driver.

Driver Minor The element `ubVersionMinor` defines the minor version number of the CANpie FD driver.

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:

CpCanMsg_ts

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

      // data field: 8 bytes (CAN) or 64 bytes (CAN FD)
      union {
            uint8_t   aubByte[CP_DATA_SIZE];
            uint16_t  auwWord[CP_DATA_SIZE / 2];
            uint32_t  aulLong[CP_DATA_SIZE / 4];
            uint64_t  auqQuad[CP_DATA_SIZE / 8];
      } tuMsgData;

      // 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;
```

Example 9: Structure of a CAN message

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).

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

Data Field

The data field union (`tuMsgData`) 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

3

Message Control

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

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 *FD Format indicator* if set to 1 (i.e. CAN FD 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 RPC bit defines a *Remote Procedure Call* if set. The operation of RPC frames is explained in chapter 5 of this document.

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

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

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

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.

User Data

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

3.4 Bittiming

To ensure correct sampling up to the last bit, a CAN node needs to resynchronize 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;
```

Example 10: Structure for statistic information

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`.

3.7 State of CAN Controller

The actual state of the CAN controller can be gathered by calling the function `CpCoreCanState()`. The information is copied into a structure of type `CpState_ts`. The structure is defined in the header file `canpie.h` and has the following data fields:

CpState_ts

```
typedef struct CpState_s
{

   // CAN error state
   uint8_t     ubCanErrState;

   // Last error type occurred
   uint8_t     ubCanErrType;

   // receive error counter
   uint8_t     ubCanRcvErrCnt;

   // transmit error counter
   uint8_t     ubCanTrmErrCnt;

} CpState_ts;;
```

Example 11: Structure for CAN controller state

3

3

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

The functions are defined inside the `cp_core.h` file.

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 pre-defined 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 24 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`.

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 26). 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 12: 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          ubFormatV,
    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()`.

4

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` may 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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

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.

4

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 26). 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 13: 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 26). 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 14: 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.

In case a FIFO is assigned to the message buffer the function will call `CpCoreFifoRelease()` automatically.

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 26). 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 15: 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 26). 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 16: 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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
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 17: 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.

4

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 26). 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.

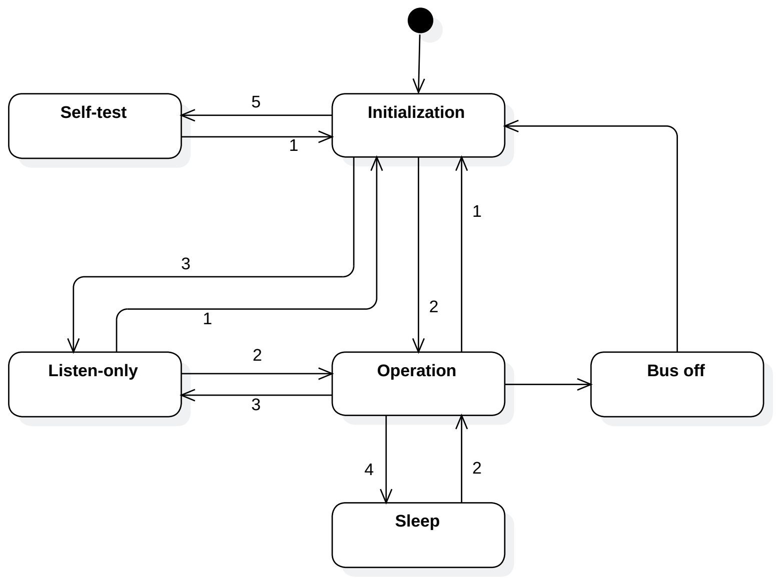

Figure 3: CAN controller FSA

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 26). 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 18: 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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

```
void DemoGetStateOfCAN(CpPort_ts * ptsCanPortV)
{
   CpState_ts  tsStateT;

   CpCoreCanState(ptsCanPortV, &tsStateT);
   if (tsStateT.ubCanErrState == eCP_STATE_BUS_OFF)
   {
      // No communication - Rien ne va plus!
   }
}
```

Example 19: Retrieve present state of CAN controller

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.

4

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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

Please refer to “Initialisation Process” on page 14 for a code example.

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

Parameters

`ptsPortV`	Pointer to CAN port structure

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 26). 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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

Please refer to “Working with a FIFO” on page 17 for a code example.

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.

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 26). 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 20: Read message from FIFO

4

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 26). 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 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

Please refer to “Configure a FIFO for message transmission” on page 18 for a code example.

4

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 20 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 26). 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.

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

The callback functions for receive and transmit interrupt provide a pointer to the `CpCanMsg_ts` structure. The following elements of the structure are guaranteed to be updated by the CAN interrupt handler:

- identifier field (`ulIdentifier`)
- time-stamp field, if supported (`tsMsgTime`)
- message marker field, if supported (`ulMsgMarker`)

The value of all other members is not defined, i.e. they are not updated by the CAN interrupt handler.

Return Value

Error code is defined by the `CpErr_e` enumeration (refer to table 9 on page 26). If no error occurred, the function will return the value `eCP_ERR_NONE`.

Example

Please refer to “Configure buffer for message reception” on page 15 for a code example.

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 26). If no error occurred, the function will return the value eCP_ERR_NONE.

5. CAN Message Functions

Access to the members of the CAN message structure `CpCanMsg_ts` 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 functions as well as macros. The symbol **`CP_CAN_MSG_MACRO`** defines which implementation is used.

Function	*Description*
CpMsgDlcToSize()	Convert DLC to payload size
CpMsgGetData()	Read CAN message payload
CpMsgGetDlc()	Read CAN message DLC
CpMsgGetIdentifier()	Read CAN message identifier
CpMsgInit()	Initialise message structure
CpMsgIsBitrateSwitchSet()	Test for bit-rate switch
CpMsgIsExtended()	Test for Extended frame format
CpMsgIsFdFrame()	Test for FD frame format
CpMsgIsRpc()	Test for Remote Procedure Call
CpMsgSetBitrateSwitch()	Set BRS flag in CAN frame
CpMsgSetData()	Write CAN message payload
CpMsgSetDlc()	Write CAN message DLC
CpMsgSetIdentifier()	Write CAN message identifier
CpMsgSizeToDlc()	Convert payload size to DLC value

Table 15: Functions for CAN message manipulation

The functions are defined inside the `cp_msg.h` file.

5.1 CpMsgDlcToSize

Syntax

```
uint8_t CpMsgDlcToSize(
     const uint8_t   ubDlcV)
```

Function

Convert DLC to payload size

This helper function performs a conversion between a DLC value and the payload size in bytes according to table 6.

Parameters

ubDlcV DLC value

Return Value

Number of bytes in CAN message payload.

5

5.2 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

Return Value

Data value at specified position.

Example

```
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 21: Get data byte from CAN message structure

5.3 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

Example

```
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 22: Check data length code from CAN message structure

5

5.4 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 using the following functions:

`CpMsgIsFdFrame()`
`CpMsgIsExtended()`

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

Identifier value

5

Example

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

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

      ....
   }

   ....

}
```

Example 23: Get identifier value

5.5 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 32.

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

Return value

None

Example

```
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 24: Initialise CAN message

5

5.6 CpMsgIsBitrateSwitchSet

Syntax

```
bool_t CpMsgIsBitrateSwitchSet(
     CpCanMsg_ts *   ptsCanMsgV)
```

Function

Test for BRS value

This function checks the BRS value inside a CAN FD frame. If the message is a CAN FD frame and the BRS bit is set the value **`true`** is returned. In all other cases the value **`false`** is returned.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

`true` on BRS bit set, **`false`** otherwise

5

Example

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

   //-------------------------------------------------
   // test for FD frame first
   //
   if( CpMsgIsBitrateSwitchSet(ptsCanMsgV) == true )
   {
       // CAN FD frame with BRS active
     ....
   }

   ....

}
```

Example 25: Test for BRS bit

5.7 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

Example

5

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

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

      ....
   }

   ....

}
```

Example 26: Test frame format

5.8 CpMsgIsFdFrame

Syntax

```
bool_t CpMsgIsFdFrame(
     CpCanMsg_ts *   ptsCanMsgV)
```

Function

Test for FD frame format

This function checks the frame format. If the message is a Classical CAN frame the value **`false`** is returned. If the message is a CAN FD frame format the value **`true`** is returned.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

`true` on CAN FD frame, **`false`** on Classical CAN frame

5

Example

```
void MyMessageRead(CpCanMsg_ts * ptsCanMsgV)
{

   //------------------------------------------------
   // read identifier from CAN message
   //
   if( CpMsgIsFdFrame(ptsCanMsgV) == true )
   {
      // this is a CAN FD frame

      ....
   }

   ....

}
```

Example 27: Test frame format

5.9 CpMsgIsRpc

Syntax

```
bool_t CpMsgIsRpc(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Test for Remote Procedure Call

This function checks if the RPC flag inside CAN message structure is set.

Parameters

`ptsCanMsgV` Pointer to CAN message structure

Return value

`true` on RPC frame, `false` on CAN frame

5.10 CpMsgSetBitrateSwitch

Syntax

```
void CpMsgSetBitrateSwitch(
    CpCanMsg_ts *   ptsCanMsgV)
```

Function

Set BRS bit in CAN message

This function checks the frame type. If the frame is a CAN FD frame the bit-rate switch (BRS) bit is set, otherwise the bit value in the message control field is not altered.

Parameters

`ptsCanMsgV`	Pointer to CAN message structure

Return value

None

5

Example

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


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

   // CAN FD frame with BRS
     ....


   ....

}
```

Example 28: Set BRS bit

5.11 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

Return value

None

Example

```
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
CpMsgSetData(&tsMyCanMsgT, 0, 0x11);   // byte 0 = 0x11
CpMsgSetData(&tsMyCanMsgT, 1, 0x22);   // byte 1 = 0x22
CpMsgSetData(&tsMyCanMsgT, 2, 0x33);   // byte 2 = 0x33
CpMsgSetData(&tsMyCanMsgT, 3, 0x44);   // byte 3 = 0x44
```

Example 29: Modify data of CAN message

5.12 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

5

Return value

None

Example

```
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 30: Setup the data length code

5.13 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

Return value

None

Example

```
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 31: Set CAN message identifier

5

5.14 CpMsgSizeToDlc

Syntax

```
uint8_t CpMsgSizeToDlc(
    const uint8_t   ubSizeV)
```

Function

Convert payload size to DLC

This helper function performs a conversion between the payload size in bytes and the DLC value according to table 6.

Parameters

ubSizeV	CAN message payload size

Return Value

DLC value

5

A Apache License

Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.:

5. Submission of Contributions

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

B Index

B

C

F

S

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.

MicroControl
Systemhaus für Automatisierung

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