OpenBLT Host Library - Reference Manual

Generated by Doxygen 1.9.4

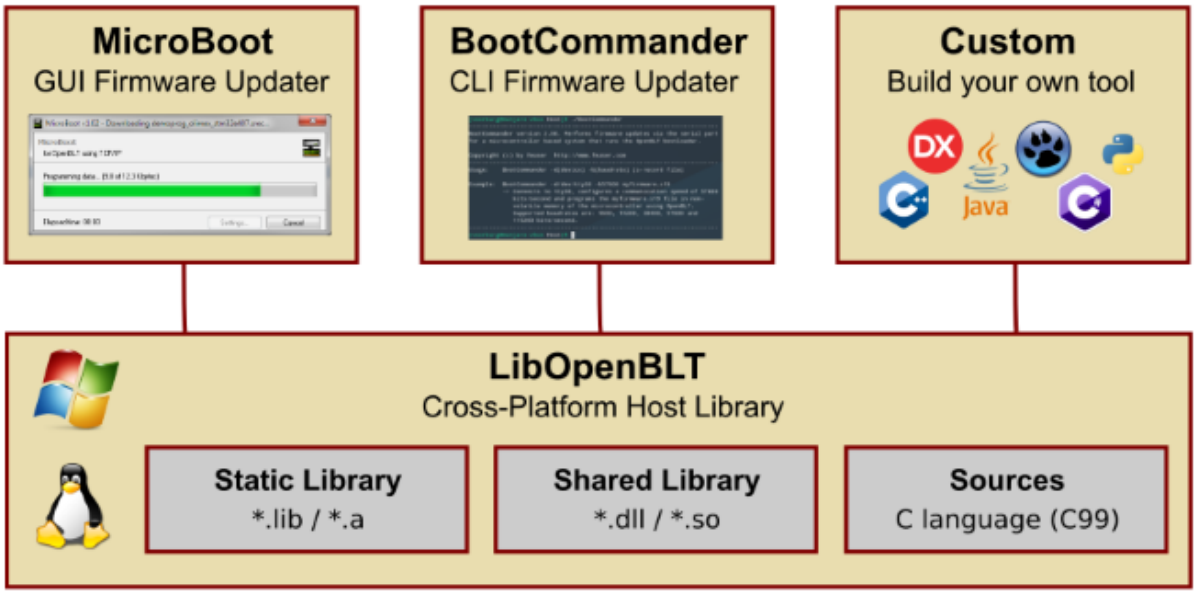
Chapter 1

OpenBLT Host Library (LibOpenBLT)

1.1 Introduction

LibOpenBLT is a host library for the OpenBLT bootloader. Its purpose is to allow quick and easy creation of programs that can connect to and perform firmware updates on a microcontroller that runs the OpenBLT bootloader.

LibOpenBLT is written in the C programming language (C99) and is cross-platform. It has been successfully tested on a Windows PC, Linux PC and even embedded Linux systems such as a Raspberry Pi and a Beagle Board.



Both the MicroBoot (GUI) and BootCommander (CLI) firmware updater tools, which are part of the OpenBLT bootloader package, make use of the OpenBLT Host Library. The source code of these two tools serve as an additional reference on how to use the OpenBLT Host Library when your are developing your own custom tool.

Refer to the OpenBLT website for additional information regarding the OpenBLT Host Library, including step-by-step instructions on how to build both that shared and static library from sources: https://www.feaser.com/openblt/doku.php?id=manual:libopenblt.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

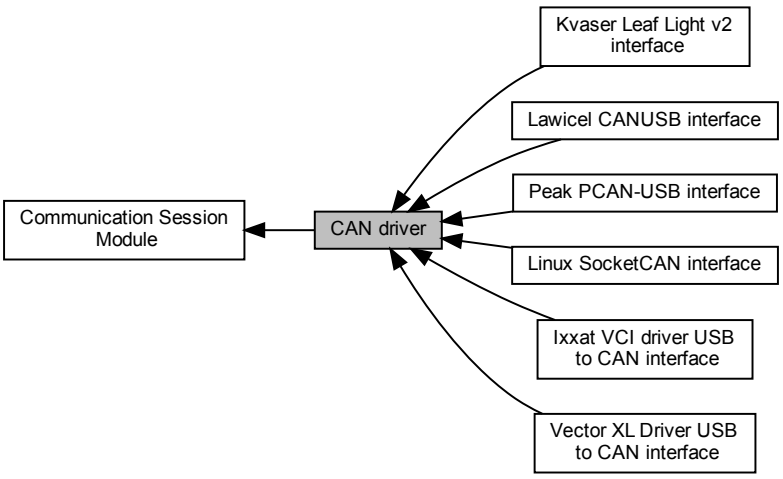
Chapter 5

Module Documentation

5.1 CAN driver

This module implements a generic CAN driver.

Collaboration diagram for CAN driver:



Modules

- Linux SocketCAN interface

  This module implements the CAN interface for Linux SocketCAN.
- Ixxat VCI driver USB to CAN interface

  This module implements the CAN interface for the Ixxat VCI driver.
- Kvaser Leaf Light v2 interface

  This module implements the CAN interface for the Kvaser Leaf Light v2.
- Lawicel CANUSB interface

  This module implements the CAN interface for the Lawicel CANUSB.
- Peak PCAN-USB interface

  This module implements the CAN interface for the Peak PCAN-USB.
- Vector XL Driver USB to CAN interface

  This module implements the CAN interface for the Vector XL Driver.

Files

- file candriver.c

  Generic CAN driver source file.
- file candriver.h

  Generic CAN driver header file.

5.1.1 Detailed Description

This module implements a generic CAN driver.

5.2 Firmware Data Module

Module with functionality to load, manipulate and store firmware data.

Collaboration diagram for Firmware Data Module:

Files

- file firmware.c

  Firmware data module source file.
- file firmware.h

  Firmware data module header file.
- file srecparser.c

  Motorola S-record file parser source file.
- file srecparser.h

  Motorola S-record file parser header file.

5.2.1 Detailed Description

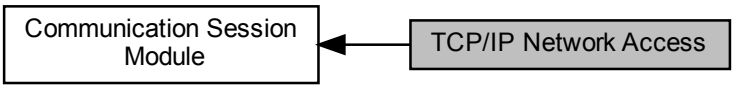
Module with functionality to load, manipulate and store firmware data.

The Firmwarwe Data module contains functionality to load, manipulate and store firmware data. It contains an interface for linking firmware file parsers that handle the loading and saving the firmware data from and to a file in the correct format. For example the Motorola S-record format.

5.3 TCP/IP Network Access

This module implements a generic TCP/IP network access client driver.

Collaboration diagram for TCP/IP Network Access:



Files

- file netaccess.h

  TCP/IP network access header file.
- file linux/netaccess.c

  TCP/IP network access source file.
- file windows/netaccess.c

  TCP/IP network access source file.

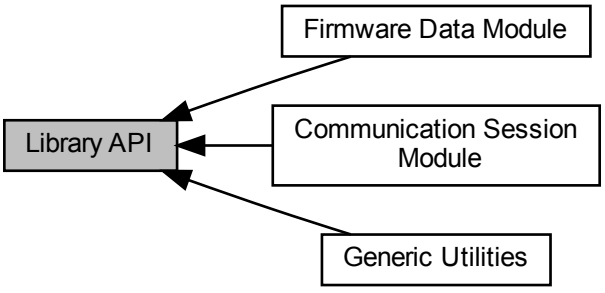
5.3.1 Detailed Description

This module implements a generic TCP/IP network access client driver.

5.4 Library API

OpenBLT Library API.

Collaboration diagram for Library API:



Modules

- Firmware Data Module

  Module with functionality to load, manipulate and store firmware data.
- Communication Session Module

  Module with functionality to communicate with the bootloader on the target system.
- Generic Utilities

  Generic utility functions and definitions.

Files

- file openblt.c

  OpenBLT host library source file.
- file openblt.h

  OpenBLT host library header file.

5.4.1 Detailed Description

OpenBLT Library API.

The Library API contains the application programming interface for the OpenBLT libary. it defines the functions and definitions that an external program uses to access the library's functionality.

5.5 Linux SocketCAN interface

This module implements the CAN interface for Linux SocketCAN.

Collaboration diagram for Linux SocketCAN interface:

Files

- file socketcan.c

  Linux SocketCAN interface source file.
- file socketcan.h

  Linux SocketCAN interface header file.

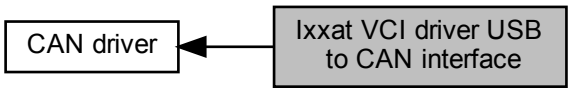
5.5.1 Detailed Description

This module implements the CAN interface for Linux SocketCAN.

5.6 Ixxat VCI driver USB to CAN interface

This module implements the CAN interface for the Ixxat VCI driver.

Collaboration diagram for Ixxat VCI driver USB to CAN interface:



Files

- file vcidriver.c

  Ixxat VCI driver interface source file.
- file vcidriver.h

  Ixxat VCI driver interface header file.

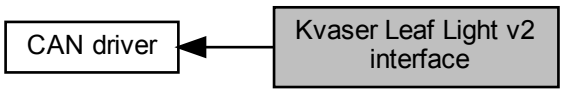
5.6.1 Detailed Description

This module implements the CAN interface for the Ixxat VCI driver.

5.7 Kvaser Leaf Light v2 interface

This module implements the CAN interface for the Kvaser Leaf Light v2.

Collaboration diagram for Kvaser Leaf Light v2 interface:



Files

- file leaflight.c

  Kvaser Leaf Light v2 interface source file.
- file leaflight.h

  Kvaser Leaf Light v2 interface header file.

5.7.1 Detailed Description

This module implements the CAN interface for the Kvaser Leaf Light v2.

5.8 Lawicel CANUSB interface

This module implements the CAN interface for the Lawicel CANUSB.

Collaboration diagram for Lawicel CANUSB interface:

Files

- file canusb.c

  Lawicel CANUSB interface source file.
- file canusb.h

  Lawicel CANUSB interface header file.

5.8.1 Detailed Description

This module implements the CAN interface for the Lawicel CANUSB.

When using the Lawicel CANUSB interface, the 32-bit driver for the CANUSB DLL API should be installed: http://www.can232.com/download/canusb_setup_win32_v_2_2.zip

5.9 Peak PCAN-USB interface

This module implements the CAN interface for the Peak PCAN-USB.

Collaboration diagram for Peak PCAN-USB interface:

Files

- file pcanusb.c

  Peak PCAN-USB interface source file.
- file pcanusb.h

  Peak PCAN-USB interface header file.

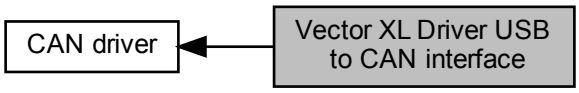
5.9.1 Detailed Description

This module implements the CAN interface for the Peak PCAN-USB.

5.10 Vector XL Driver USB to CAN interface

This module implements the CAN interface for the Vector XL Driver.

Collaboration diagram for Vector XL Driver USB to CAN interface:



Files

- file xldriver.c

  Vector XL driver interface source file.
- file xldriver.h

  Vector XL driver interface header file.

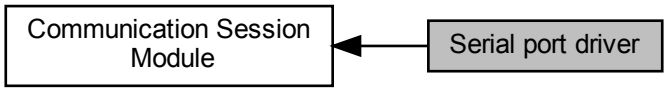
5.10.1 Detailed Description

This module implements the CAN interface for the Vector XL Driver.

5.11 Serial port driver

This module implements a generic serial port driver.

Collaboration diagram for Serial port driver:



Files

- file linux/serialport.c

  Serial port source file.
- file windows/serialport.c

  Serial port source file.
- file serialport.h

  Serial port header file.

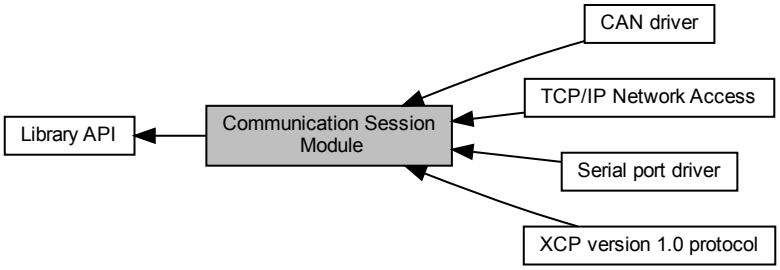
5.11.1 Detailed Description

This module implements a generic serial port driver.

5.12 Communication Session Module

Module with functionality to communicate with the bootloader on the target system.

Collaboration diagram for Communication Session Module:



Modules

- CAN driver

  This module implements a generic CAN driver.
- TCP/IP Network Access

  This module implements a generic TCP/IP network access client driver.
- Serial port driver

  This module implements a generic serial port driver.
- XCP version 1.0 protocol

  This module implements the XCP communication protocol that can be linked to the Session module.

Files

- file session.c

  Communication session module source file.
- file session.h

  Communication session module header file.

5.12.1 Detailed Description

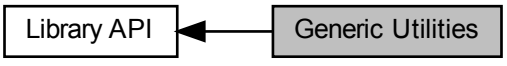
Module with functionality to communicate with the bootloader on the target system.

The Communication Session module handles the communication with the bootloader during firmware updates on the target system. It contains an interface to link the desired communication protocol that should be used for the communication. For example the XCP protocol.

5.13 Generic Utilities

Generic utility functions and definitions.

Collaboration diagram for Generic Utilities:



Files

- file linux/critutil.c

  Critical section utility source file.
- file linux/timeutil.c

  Time utility source file.
- file windows/critutil.c

  Critical section utility source file.
- file windows/timeutil.c

  Time utility source file.
- file util.c

  Utility module source file.
- file util.h

  Utility module header file.

5.13.1 Detailed Description

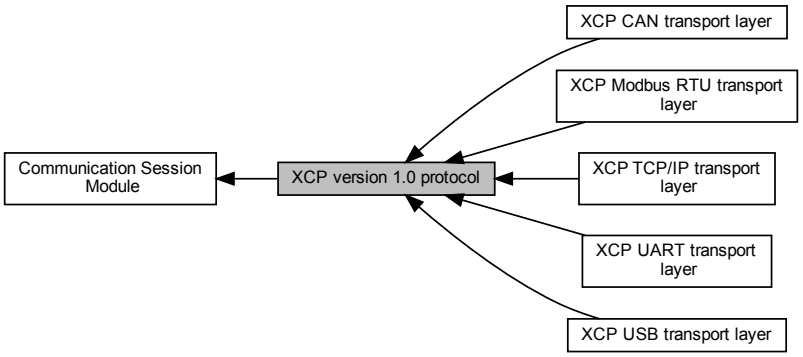
Generic utility functions and definitions.

The Utility module contains generic functions and definitions that can be handy for use internally in the library and also externally by another application that makes use of the library.

5.14 XCP version 1.0 protocol

This module implements the XCP communication protocol that can be linked to the Session module.

Collaboration diagram for XCP version 1.0 protocol:



Modules

- XCP CAN transport layer

  This module implements the XCP transport layer for CAN.
- XCP Modbus RTU transport layer

  This module implements the XCP transport layer for Modbus RTU.
- XCP TCP/IP transport layer

  This module implements the XCP transport layer for TCP/IP.
- XCP UART transport layer

  This module implements the XCP transport layer for UART.
- XCP USB transport layer

  This module implements the XCP transport layer for USB.

Files

- file linux/xcpprotect.c

  XCP Protection module source file.
- file windows/xcpprotect.c

  XCP Protection module source file.
- file xcploader.c

  XCP Loader module source file.
- file xcploader.h

  XCP Loader module header file.
- file xcpprotect.h

  XCP Protection module header file.

5.14.1 Detailed Description

This module implements the XCP communication protocol that can be linked to the Session module.

This XCP Loader module contains functionality according to the standardized XCP protocol version 1.0. XCP is a universal measurement and calibration communication protocol. Note that only those parts of the XCP master functionality are implemented that are applicable to performing a firmware update on the slave. This means functionality for reading, programming, and erasing (non-volatile) memory.

5.15 XCP CAN transport layer

This module implements the XCP transport layer for CAN.

Collaboration diagram for XCP CAN transport layer:

Files

- file xcptpcan.c

  XCP CAN transport layer source file.
- file xcptpcan.h

  XCP CAN transport layer header file.

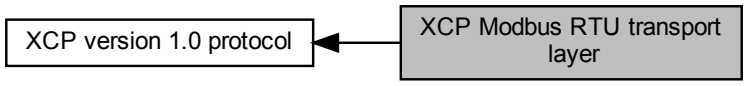
5.15.1 Detailed Description

This module implements the XCP transport layer for CAN.

5.16 XCP Modbus RTU transport layer

This module implements the XCP transport layer for Modbus RTU.

Collaboration diagram for XCP Modbus RTU transport layer:



Files

- file xcptpmbrtu.c

  XCP Modbus RTU transport layer source file.
- file xcptpmbrtu.h

  XCP Modbus RTU transport layer header file.

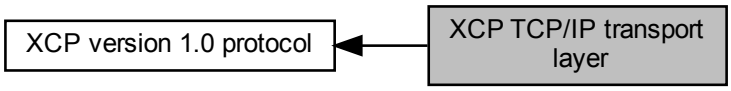
5.16.1 Detailed Description

This module implements the XCP transport layer for Modbus RTU.

5.17 XCP TCP/IP transport layer

This module implements the XCP transport layer for TCP/IP.

Collaboration diagram for XCP TCP/IP transport layer:



Files

- file xcptpnet.c

  XCP TCP/IP transport layer source file.
- file xcptpnet.h

  XCP TCP/IP transport layer header file.

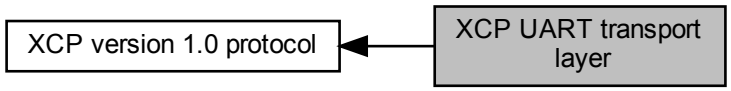
5.17.1 Detailed Description

This module implements the XCP transport layer for TCP/IP.

5.18 XCP UART transport layer

This module implements the XCP transport layer for UART.

Collaboration diagram for XCP UART transport layer:



Files

- file xcptpuart.c

  XCP UART transport layer source file.
- file xcptpuart.h

  XCP UART transport layer header file.

5.18.1 Detailed Description

This module implements the XCP transport layer for UART.

5.19 XCP USB transport layer

This module implements the XCP transport layer for USB.

Collaboration diagram for XCP USB transport layer:

Files

- file linux/usbbulk.c

  USB bulk driver source file.
- file windows/usbbulk.c

  USB bulk driver source file.
- file usbbulk.h

  USB bulk driver header file.
- file xcptpusb.c

  XCP USB transport layer source file.
- file xcptpusb.h

  XCP USB transport layer header file.

5.19.1 Detailed Description

This module implements the XCP transport layer for USB.

Chapter 6

Data Structure Documentation

6.1 tBltSessionSettingsXcpV10 Struct Reference

Structure layout of the XCP version 1.0 session settings.

`#include <openblt.h>`

Data Fields

- uint16_t timeoutT1
- uint16_t timeoutT3
- uint16_t timeoutT4
- uint16_t timeoutT5
- uint16_t timeoutT6
- uint16_t timeoutT7
- char const * seedKeyFile
- uint8_t connectMode

6.1.1 Detailed Description

Structure layout of the XCP version 1.0 session settings.

6.1.2 Field Documentation

6.1.2.1 connectMode

`uint8_t connectMode`

Connection mode parameter in XCP connect command.

6.1.2.2 seedKeyFile

```
char const* seedKeyFile
```

Seed/key algorithm library filename.

6.1.2.3 timeoutT1

```
uint16_t timeoutT1
```

Command response timeout in milliseconds.

6.1.2.4 timeoutT3

```
uint16_t timeoutT3
```

Start programming timeout in milliseconds.

6.1.2.5 timeoutT4

```
uint16_t timeoutT4
```

Erase memory timeout in milliseconds.

6.1.2.6 timeoutT5

```
uint16_t timeoutT5
```

Program memory and reset timeout in milliseconds.

6.1.2.7 timeoutT6

```
uint16_t timeoutT6
```

Connect response timeout in milliseconds.

6.1.2.8 timeoutT7

```
uint16_t timeoutT7
```

Busy wait timer timeout in milliseonds.

The documentation for this struct was generated from the following file:

- openblt.h

6.2 tBltTransportSettingsXcpV10Can Struct Reference

Structure layout of the XCP version 1.0 CAN transport layer settings. The deviceName field is platform dependent. On Linux based systems this should be the socketCAN interface name such as "can0". The terminal command "ip addr" can be issued to view a list of interfaces that are up and available. Under Linux it is assumed that the socket↩ CAN interface is already configured on the system, before using the OpenBLT library. When baudrate is configured when bringing up the system, so the baudrate field in this structure is don't care when using the library on a Linux was system. On Windows based systems, the device name is a name that is pre-defined by this library for the supported CAN adapters. The device name should be one of the following: "peak_pcanusb", "kvaser_leaflight", or "lawicel_canusb". Field use extended is a boolean field. When set to 0, the specified transmitId and receiveId are assumed to be 11-bit standard CAN identifier. It the field is 1, these identifiers are assumed to be 29-bit extended CAN identifiers.

`#include <openblt.h>`

Data Fields

- char const * deviceName
- uint32_t deviceChannel
- uint32_t baudrate
- uint32_t transmitId
- uint32_t receiveId
- uint32_t useExtended

6.2.1 Detailed Description

Structure layout of the XCP version 1.0 CAN transport layer settings. The deviceName field is platform dependent. On Linux based systems this should be the socketCAN interface name such as "can0". The terminal command "ip addr" can be issued to view a list of interfaces that are up and available. Under Linux it is assumed that the socket↩ CAN interface is already configured on the system, before using the OpenBLT library. When baudrate is configured when bringing up the system, so the baudrate field in this structure is don't care when using the library on a Linux was system. On Windows based systems, the device name is a name that is pre-defined by this library for the supported CAN adapters. The device name should be one of the following: "peak_pcanusb", "kvaser_leaflight", or "lawicel_canusb". Field use extended is a boolean field. When set to 0, the specified transmitId and receiveId are assumed to be 11-bit standard CAN identifier. It the field is 1, these identifiers are assumed to be 29-bit extended CAN identifiers.

6.2.2 Field Documentation

6.2.2.1 baudrate

`uint32_t baudrate`

Communication speed in bits/sec.

6.2.2.2 deviceChannel

```
uint32_t deviceChannel
```

Channel on the device to use.

6.2.2.3 deviceName

```
char const* deviceName
```

Device name such as can0, peak_pcanusb etc.

6.2.2.4 receiveId

```
uint32_t receiveId
```

Receive CAN identifier.

6.2.2.5 transmitId

```
uint32_t transmitId
```

Transmit CAN identifier.

6.2.2.6 useExtended

```
uint32_t useExtended
```

Boolean to configure 29-bit CAN identifiers.

The documentation for this struct was generated from the following file:

- openblt.h

6.3 tBltTransportSettingsXcpV10MbRtu Struct Reference

Structure layout of the XCP version 1.0 Modbus RTU transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".

```
#include <openblt.h>
```

Data Fields

- char const * portName
- uint32_t baudrate
- uint8_t parity
- uint8_t stopbits
- uint8_t destinationAddr

6.3.1 Detailed Description

Structure layout of the XCP version 1.0 Modbus RTU transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".

6.3.2 Field Documentation

6.3.2.1 baudrate

```
uint32_t baudrate
```

Communication speed in bits/sec.

6.3.2.2 destinationAddr

```
uint8_t destinationAddr
```

Destination address (receiver node ID).

6.3.2.3 parity

```
uint8_t parity
```

Parity (0 for none, 1 for odd, 2 for even).

6.3.2.4 portName

```
char const* portName
```

Communication port name such as /dev/tty0.

6.3.2.5 stopbits

```
uint8_t stopbits
```

Stopbits (1 for one, 2 for two stopbits).

The documentation for this struct was generated from the following file:

- openblt.h

6.4 tBltTransportSettingsXcpV10Net Struct Reference

Structure layout of the XCP version 1.0 NET transport layer settings. The address field can be set to either the IP address or the hostname, such as "192.168.178.23" or "mymicro.mydomain.com". The port should be set to the TCP port number that the bootloader target listens on.

```
#include <openblt.h>
```

Data Fields

- char const * address
- uint16_t port

6.4.1 Detailed Description

Structure layout of the XCP version 1.0 NET transport layer settings. The address field can be set to either the IP address or the hostname, such as "192.168.178.23" or "mymicro.mydomain.com". The port should be set to the TCP port number that the bootloader target listens on.

6.4.2 Field Documentation

6.4.2.1 address

```
char const* address
```

Target IP-address or hostname on the network.

6.4.2.2 port

```
uint16_t port
```

TCP port to use.

The documentation for this struct was generated from the following file:

- openblt.h

6.5 tBltTransportSettingsXcpV10Rs232 Struct Reference

Structure layout of the XCP version 1.0 RS232 transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".

```
#include <openblt.h>
```

Data Fields

- char const * portName
- uint32_t baudrate
- uint8_t csType

6.5.1 Detailed Description

Structure layout of the XCP version 1.0 RS232 transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".

6.5.2 Field Documentation

6.5.2.1 baudrate

```
uint32_t baudrate
```

Communication speed in bits/sec.

6.5.2.2 csType

```
uint8_t csType
```

Checksum type, 0=none, 1=byte.

6.5.2.3 portName

```
char const* portName
```

Communication port name such as /dev/tty0.

The documentation for this struct was generated from the following file:

- openblt.h

6.6 tBulkUsbDev Struct Reference

Type for grouping together all USB bulk device related data.

6.6.1 Detailed Description

Type for grouping together all USB bulk device related data.

The documentation for this struct was generated from the following file:

- windows/usbbulk.c

6.7 tCanEvents Struct Reference

Structure with CAN event callback functions.

```
#include <candriver.h>
```

Data Fields

- void(* **MsgTxed**)(tCanMsg const *msg)

  Event function that should be called when a message was transmitted.
- void(* **MsgRxed**)(tCanMsg const *msg)

  Event function that should be called when a message was received.

6.7.1 Detailed Description

Structure with CAN event callback functions.

The documentation for this struct was generated from the following file:

- candriver.h

6.8 tCanInterface Struct Reference

CAN interface type.

`#include <candriver.h>`

Data Fields

- void(∗ **Init**)(tCanSettings const ∗settings)
  Initialization of the CAN interface.
- void(∗ **Terminate**)(void)
  Terminates the CAN interface.
- bool(∗ **Connect**)(void)
  Connects the CAN interface to the CAN bus.
- void(∗ **Disconnect**)(void)
  Disconnects the CAN interface from the CAN bus.
- bool(∗ **Transmit**)(tCanMsg const ∗msg)
  Submits a CAN message for transmission.
- bool(∗ **IsBusError**)(void)
  Check if a bus off and/or bus heavy situation occurred.
- void(∗ **RegisterEvents**)(tCanEvents const ∗events)
  Registers the event callback functions.

6.8.1 Detailed Description

CAN interface type.

The documentation for this struct was generated from the following file:

- candriver.h

6.9 tCanMsg Struct Reference

Layout of a CAN message. Note that CAN_MSG_EXT_ID_MASK can be used to configure the CAN message identifier as 29-bit extended.

`#include <candriver.h>`

Data Fields

- uint32_t id
- uint8_t dlc
- uint8_t data [CAN_MSG_MAX_LEN]

6.9.1 Detailed Description

Layout of a CAN message. Note that CAN_MSG_EXT_ID_MASK can be used to configure the CAN message identifier as 29-bit extended.

6.9.2 Field Documentation

6.9.2.1 data

```
uint8_t data[CAN_MSG_MAX_LEN]
```

Array with CAN message data.

Referenced by CanUsbLibReceiveCallback(), CanUsbTransmit(), IxxatVciReceptionThread(), IxxatVciTransmit(), LeafLightReceptionThread(), LeafLightTransmit(), PCanUsbReceptionThread(), PCanUsbTransmit(), SocketCanTransmit(), VectorXlReceptionThread(), VectorXlTransmit(), XcpTpCanEventMessageReceived(), and XcpTpCanSendPacket().

6.9.2.2 dlc

```
uint8_t dlc
```

CAN message data length code.

Referenced by CanUsbLibReceiveCallback(), CanUsbTransmit(), IxxatVciReceptionThread(), IxxatVciTransmit(), LeafLightReceptionThread(), LeafLightTransmit(), PCanUsbReceptionThread(), PCanUsbTransmit(), SocketCanTransmit(), VectorXlReceptionThread(), VectorXlTransmit(), XcpTpCanEventMessageReceived(), and XcpTpCanSendPacket().

6.9.2.3 id

```
uint32_t id
```

CAN message identifier.

Referenced by CanUsbLibReceiveCallback(), CanUsbTransmit(), IxxatVciReceptionThread(), IxxatVciTransmit(), LeafLightReceptionThread(), LeafLightTransmit(), PCanUsbReceptionThread(), PCanUsbTransmit(), SocketCanTransmit(), VectorXlReceptionThread(), VectorXlTransmit(), XcpTpCanEventMessageReceived(), and XcpTpCanSendPacket().

The documentation for this struct was generated from the following file:

- candriver.h

6.10 tCanSettings Struct Reference

Type to group of CAN interface related settings. The device name specifies the name of the CAN interface device. For some CAN interfaces this is don't care, but for other absolutely necessar, for example Linux SocketCAN. The channel specifies the channel on the CAN interface, in case it has multiple CAN channels. The baudrate specifies the communication speed on the CAN network. The code and mask values configure the message reception acceptance filter. A mask bit value of 0 means don't care. The code part of the filter determines what bit values to match in the received message identifier. Example 1: Receive all CAN identifiers .code = 0x00000000 .mask = 0x00000000 Example 2: Receive only CAN identifier 0x124 (11-bit or 29-bit) .code = 0x00000124 .mask = 0x1fffffff Example 3: Receive only CAN identifier 0x124 (11-bit) .code = 0x00000124 .mask = 0x9fffffff Example 4: Receive only CAN identifier 0x124 (29-bit) .code = 0x80000124 .mask = 0x9fffffff.

`#include <candriver.h>`

Data Fields

- char const * devicename
- uint32_t channel
- tCanBaudrate baudrate
- uint32_t code
- uint32_t mask

6.10.1 Detailed Description

Type to group of CAN interface related settings. The device name specifies the name of the CAN interface device. For some CAN interfaces this is don't care, but for other absolutely necessar, for example Linux SocketCAN. The channel specifies the channel on the CAN interface, in case it has multiple CAN channels. The baudrate specifies the communication speed on the CAN network. The code and mask values configure the message reception acceptance filter. A mask bit value of 0 means don't care. The code part of the filter determines what bit values to match in the received message identifier. Example 1: Receive all CAN identifiers .code = 0x00000000 .mask = 0x00000000 Example 2: Receive only CAN identifier 0x124 (11-bit or 29-bit) .code = 0x00000124 .mask = 0x1fffffff Example 3: Receive only CAN identifier 0x124 (11-bit) .code = 0x00000124 .mask = 0x9fffffff Example 4: Receive only CAN identifier 0x124 (29-bit) .code = 0x80000124 .mask = 0x9fffffff.

6.10.2 Field Documentation

6.10.2.1 baudrate

`tCanBaudrate baudrate`

Communication speed.

Referenced by CanUsbInit(), CanUsbOpenChannel(), CanUsbTerminate(), IxxatVciConvertBaudrate(), IxxatVciInit(), IxxatVciTerminate(), LeafLightConnect(), LeafLightInit(), LeafLightTerminate(), PCanUsbConnect(), PCanUsbInit(), PCanUsbTerminate(), SocketCanInit(), SocketCanTerminate(), VectorXlConnect(), VectorXlInit(), VectorXlTerminate(), and XcpTpCanInit().

6.10.2.2 channel

```
uint32_t channel
```

Zero based CAN channel index.

Referenced by CanUsbInit(), CanUsbTerminate(), IxxatVciConnect(), IxxatVciInit(), IxxatVciTerminate(), LeafLightInit(), LeafLightTerminate(), PCanUsbConnect(), PCanUsbDisconnect(), PCanUsbInit(), PCanUsbIsBusError(), PCanUsbReceptionThread(), PCanUsbTerminate(), PCanUsbTransmit(), SocketCanInit(), SocketCanTerminate(), VectorXlConnect(), VectorXlInit(), VectorXlTerminate(), and XcpTpCanInit().

6.10.2.3 code

```
uint32_t code
```

Code of the reception acceptance filter.

Referenced by CanUsbInit(), CanUsbOpenChannel(), CanUsbTerminate(), IxxatVciConnect(), IxxatVciInit(), IxxatVciTerminate(), LeafLightConnect(), LeafLightInit(), LeafLightTerminate(), PCanUsbConnect(), PCanUsbInit(), PCanUsbTerminate(), SocketCanConnect(), SocketCanInit(), SocketCanTerminate(), VectorXlConnect(), VectorXlInit(), VectorXlTerminate(), and XcpTpCanInit().

6.10.2.4 devicename

```
char const* devicename
```

CAN interface device name (pcanusb, vcan0).

Referenced by CanInit(), CanUsbInit(), CanUsbTerminate(), IxxatVciInit(), IxxatVciTerminate(), LeafLightInit(), LeafLightTerminate(), PCanUsbInit(), PCanUsbTerminate(), SocketCanConnect(), SocketCanInit(), SocketCanTerminate(), VectorXlInit(), VectorXlTerminate(), and XcpTpCanInit().

6.10.2.5 mask

```
uint32_t mask
```

Mask of the reception acceptance filter.

Referenced by CanUsbInit(), CanUsbOpenChannel(), CanUsbTerminate(), IxxatVciConnect(), IxxatVciInit(), IxxatVciTerminate(), LeafLightConnect(), LeafLightInit(), LeafLightTerminate(), PCanUsbConnect(), PCanUsbInit(), PCanUsbTerminate(), SocketCanConnect(), SocketCanInit(), SocketCanTerminate(), VectorXlConnect(), VectorXlInit(), VectorXlTerminate(), and XcpTpCanInit().

The documentation for this struct was generated from the following file:

- candriver.h

6.11 tFirmwareParser Struct Reference

Firmware file parser.

`#include <firmware.h>`

Data Fields

- bool(∗ **LoadFromFile**)(char const ∗firmwareFile, uint32_t addressOffset)
  Extract the firmware segments from the firmware file and add them as nodes to the linked list.
- bool(∗ **SaveToFile**)(char const ∗firmwareFile)
  Write all the firmware segments from the linked list to the specified firmware file.

6.11.1 Detailed Description

Firmware file parser.

The documentation for this struct was generated from the following file:

- firmware.h

6.12 tFirmwareSegment Struct Reference

Groups information together of a firmware segment, such that it can be used as a node in a linked list.

`#include <firmware.h>`

Data Fields

- uint32_t **base**
  Start memory address of the segment.
- uint32_t **length**
  Number of data bytes in the segment.
- uint8_t ∗ **data**
  Pointer to array with the segment's data bytes.
- struct t_firmware_segment ∗ **prev**
  Pointer to the previous node, or NULL if it is the first one.
- struct t_firmware_segment ∗ **next**
  Pointer to the next node, or NULL if it is the last one.

6.12.1 Detailed Description

Groups information together of a firmware segment, such that it can be used as a node in a linked list.

The documentation for this struct was generated from the following file:

- firmware.h

6.13 tSessionProtocol Struct Reference

Session communication protocol interface.

`#include <session.h>`

Data Fields

- void(∗ **Init**)(void const ∗settings)

  Initializes the protocol module.
- void(∗ **Terminate**)(void)

  Terminates the protocol module.
- bool(∗ **Start**)(void)

  Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.
- void(∗ **Stop**)(void)

  Stops the firmware update. This is where the bootloader starts the user program on the target if a valid one is present. After this the connection with the target is severed.
- bool(∗ **ClearMemory**)(uint32_t address, uint32_t len)

  Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.
- bool(∗ **WriteData**)(uint32_t address, uint32_t len, uint8_t const ∗data)

  Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.
- bool(∗ **ReadData**)(uint32_t address, uint32_t len, uint8_t ∗data)

  Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

6.13.1 Detailed Description

Session communication protocol interface.

The documentation for this struct was generated from the following file:

- session.h

6.14 tSocketCanThreadCtrl Struct Reference

Groups data for thread control.

Data Fields

- bool terminate
- bool terminated

6.14.1 Detailed Description

Groups data for thread control.

6.14.2 Field Documentation

6.14.2.1 terminate

```
bool terminate
```

flag to request thread termination.

Referenced by SocketCanEventThread(), SocketCanStartEventThread(), and SocketCanStopEventThread().

6.14.2.2 terminated

```
bool terminated
```

handshake flag.

Referenced by SocketCanEventThread(), SocketCanStartEventThread(), and SocketCanStopEventThread().

The documentation for this struct was generated from the following file:

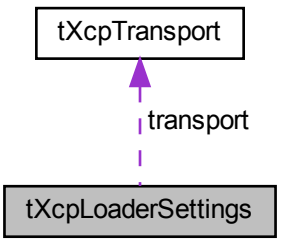
- socketcan.c

6.15 tXcpLoaderSettings Struct Reference

XCP protocol specific settings.

```
#include <xcploader.h>
```

Collaboration diagram for tXcpLoaderSettings:



Data Fields

- uint16_t **timeoutT1**

  Command response timeout in milliseconds.
- uint16_t **timeoutT3**

  Start programming timeout in milliseconds.
- uint16_t **timeoutT4**

  Erase memory timeout in milliseconds.
- uint16_t **timeoutT5**

  Program memory and reset timeout in milliseconds.
- uint16_t **timeoutT6**

  Connect response timeout in milliseconds.
- uint16_t **timeoutT7**

  Busy wait timer timeout in milliseconds.
- uint8_t **connectMode**

  Connection mode used in the XCP connect command.
- char const * **seedKeyFile**

  Seed/key algorithm library filename.
- tXcpTransport const * **transport**

  Pointer to the transport layer to use during protocol communications.
- void const * **transportSettings**

  Pointer to the settings for the transport layer.

6.15.1 Detailed Description

XCP protocol specific settings.

The documentation for this struct was generated from the following file:

- xcploader.h

6.16 tXcpTpCanSettings Struct Reference

Layout of structure with settings specific to the XCP transport layer module for CAN.

```
#include <xcptpcan.h>
```

Data Fields

- char const * device
- uint32_t channel
- uint32_t baudrate
- uint32_t transmitId
- uint32_t receiveId
- bool useExtended

6.16.1 Detailed Description

Layout of structure with settings specific to the XCP transport layer module for CAN.

6.16.2 Field Documentation

6.16.2.1 baudrate

```
uint32_t baudrate
```

Communication speed in bits/sec.

Referenced by XcpTpCanInit(), and XcpTpCanTerminate().

6.16.2.2 channel

```
uint32_t channel
```

Channel on the device to use.

Referenced by XcpTpCanInit(), and XcpTpCanTerminate().

6.16.2.3 device

```
char const* device
```

Device name such as can0, peak_pcanusb, etc.

Referenced by XcpTpCanInit(), and XcpTpCanTerminate().

6.16.2.4 receiveId

```
uint32_t receiveId
```

Receive CAN identifier.

Referenced by XcpTpCanEventMessageReceived(), XcpTpCanInit(), and XcpTpCanTerminate().

6.16.2.5 transmitId

```
uint32_t transmitId
```

Transmit CAN identifier.

Referenced by XcpTpCanInit(), XcpTpCanSendPacket(), and XcpTpCanTerminate().

6.16.2.6 useExtended

```
bool useExtended
```

Boolean to configure 29-bit CAN identifiers.

Referenced by XcpTpCanEventMessageReceived(), XcpTpCanInit(), XcpTpCanSendPacket(), and XcpTpCanTerminate().

The documentation for this struct was generated from the following file:

- xcptpcan.h

6.17 tXcpTpMbRtuSettings Struct Reference

Layout of structure with settings specific to the XCP transport layer module for Modbus RTU.

```
#include <xcptpmbrtu.h>
```

Data Fields

- char const * portname
- uint32_t baudrate
- uint8_t parity
- uint8_t stopbits
- uint8_t destinationAddr

6.17.1 Detailed Description

Layout of structure with settings specific to the XCP transport layer module for Modbus RTU.

6.17.2 Field Documentation

6.17.2.1 baudrate

```
uint32_t baudrate
```

Communication speed in bits/sec.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuInit(), and XcpTpMbRtuTerminate().

6.17.2.2 destinationAddr

```
uint8_t destinationAddr
```

Destination address (receiver node ID).

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuInit(), XcpTpMbRtuSendPacket(), and XcpTpMbRtuTerminate().

6.17.2.3 parity

```
uint8_t parity
```

Parity (0 for none, 1 for odd, 2 for even.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuInit(), and XcpTpMbRtuTerminate().

6.17.2.4 portname

```
char const* portname
```

Interface port name, i.e. /dev/ttyUSB0.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuInit(), and XcpTpMbRtuTerminate().

6.17.2.5 stopbits

```
uint8_t stopbits
```

Stopbits (1 for one, 2 for two stopbits.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuInit(), and XcpTpMbRtuTerminate().

The documentation for this struct was generated from the following file:

- xcptpmbrtu.h

6.18 tXcpTpNetSettings Struct Reference

Layout of structure with settings specific to the XCP transport layer module for TCP/IP.

`#include <xcptpnet.h>`

Data Fields

- char const * address
- uint16_t port

6.18.1 Detailed Description

Layout of structure with settings specific to the XCP transport layer module for TCP/IP.

6.18.2 Field Documentation

6.18.2.1 address

`char const* address`

Target IP-address or hostname on the network.

Referenced by XcpTpNetConnect(), XcpTpNetInit(), and XcpTpNetTerminate().

6.18.2.2 port

`uint16_t port`

TCP port to use.

Referenced by XcpTpNetConnect(), XcpTpNetInit(), and XcpTpNetTerminate().

The documentation for this struct was generated from the following file:

- xcptpnet.h

6.19 tXcpTpUartSettings Struct Reference

Layout of structure with settings specific to the XCP transport layer module for UART.

`#include <xcptpuart.h>`

Data Fields

- char const * portname
- uint32_t baudrate
- uint8_t cstype

6.19.1 Detailed Description

Layout of structure with settings specific to the XCP transport layer module for UART.

6.19.2 Field Documentation

6.19.2.1 baudrate

```
uint32_t baudrate
```

Communication speed in bits/sec.

Referenced by XcpTpUartConnect(), XcpTpUartInit(), and XcpTpUartTerminate().

6.19.2.2 cstype

```
uint8_t cstype
```

Checksum type, 0=none, 1=byte.

Referenced by XcpTpUartInit(), XcpTpUartSendPacket(), and XcpTpUartTerminate().

6.19.2.3 portname

```
char const* portname
```

Interface port name, i.e. /dev/ttyUSB0.

Referenced by XcpTpUartConnect(), XcpTpUartInit(), and XcpTpUartTerminate().

The documentation for this struct was generated from the following file:

- xcptpuart.h

6.20 tXcpTransport Struct Reference

XCP transport layer.

`#include <xcploader.h>`

Data Fields

- void(∗ **Init**)(void const ∗settings)

  Initialization of the XCP transport layer.
- void(∗ **Terminate**)(void)

  Termination the XCP transport layer.
- bool(∗ **Connect**)(void)

  Connects the XCP transport layer.
- void(∗ **Disconnect**)(void)

  Disconnects the XCP transport layer.
- bool(∗ **SendPacket**)(tXcpTransportPacket const ∗txPacket, tXcpTransportPacket ∗rxPacket, uint16_t timeout)

  Sends an XCP packet and waits for the response to come back.

6.20.1 Detailed Description

XCP transport layer.

The documentation for this struct was generated from the following file:

- xcploader.h

6.21 tXcpTransportPacket Struct Reference

XCP transport layer packet type.

`#include <xcploader.h>`

Data Fields

- uint8_t data [XCPLOADER_PACKET_SIZE_MAX]
- uint8_t len

6.21.1 Detailed Description

XCP transport layer packet type.

6.21.2 Field Documentation

6.21.2.1 data

```
uint8_t data[XCPLOADER_PACKET_SIZE_MAX]
```

Packet data.

Referenced by XcpLoaderSendCmdConnect(), XcpLoaderSendCmdGetSeed(), XcpLoaderSendCmdGetStatus(), XcpLoaderSendCmdProgram(), XcpLoaderSendCmdProgramClear(), XcpLoaderSendCmdProgramMax(), XcpLoaderSendCmdProgramReset(), XcpLoaderSendCmdProgramStart(), XcpLoaderSendCmdSetMta(), XcpLoaderSendCmdUnlock(), XcpLoaderSendCmdUpload(), XcpTpCanSendPacket(), XcpTpMbRtuSendPacket(), XcpTpNetSendPacket(), XcpTpUartSendPacket(), and XcpTpUsbSendPacket().

6.21.2.2 len

```
uint8_t len
```

Packet length.

Referenced by XcpLoaderSendCmdConnect(), XcpLoaderSendCmdGetSeed(), XcpLoaderSendCmdGetStatus(), XcpLoaderSendCmdProgram(), XcpLoaderSendCmdProgramClear(), XcpLoaderSendCmdProgramMax(), XcpLoaderSendCmdProgramReset(), XcpLoaderSendCmdProgramStart(), XcpLoaderSendCmdSetMta(), XcpLoaderSendCmdUnlock(), XcpLoaderSendCmdUpload(), XcpTpCanSendPacket(), XcpTpMbRtuSendPacket(), XcpTpNetSendPacket(), XcpTpUartSendPacket(), and XcpTpUsbSendPacket().

The documentation for this struct was generated from the following file:

- xcploader.h

Chapter 7

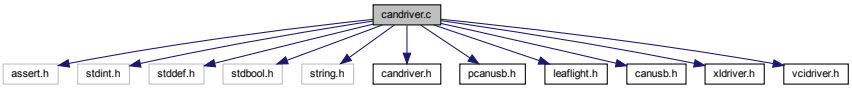
File Documentation

7.1 candriver.c File Reference

Generic CAN driver source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <string.h>
#include "candriver.h"
#include "pcanusb.h"
#include "leaflight.h"
#include "canusb.h"
#include "xldriver.h"
#include "vcidriver.h"
```

Include dependency graph for candriver.c:



Functions

- void CanInit (tCanSettings const *settings)

  Initializes the CAN module. Typically called once at program startup.
- void **CanTerminate** (void)

  Terminates the CAN module. Typically called once at program cleanup.
- bool CanConnect (void)

  Connects the CAN module.
- void **CanDisconnect** (void)

  Disconnects the CAN module.
- bool CanIsConnected (void)

  Obtains the connection state of the CAN module.

- bool CanTransmit (tCanMsg const *msg)
  Submits a message for transmission on the CAN bus.
- bool CanIsBusError (void)
  Checks if a bus off or bus heavy situation occurred.
- void CanRegisterEvents (tCanEvents const *events)
  Registers the event callback functions that should be called by the CAN module.

Variables

- static tCanInterface const * **canIfPtr**
  Pointer to the CAN interface that is linked.
- static bool **canConnected**
  Flag to store the connection status.

7.1.1 Detailed Description

Generic CAN driver source file.

7.1.2 Function Documentation

7.1.2.1 CanConnect()

```
bool CanConnect (
            void  )
```

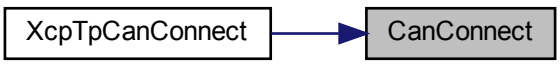
Connects the CAN module.

Returns

True if connected, false otherwise.

Referenced by XcpTpCanConnect().

Here is the caller graph for this function:



7.1.2.2 CanInit()

```
void CanInit (
            tCanSettings const * settings )
```

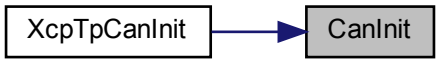
Initializes the CAN module. Typically called once at program startup.

Parameters

settings	Pointer to the CAN module settings.

Referenced by XcpTpCanInit().

Here is the caller graph for this function:



7.1.2.3 CanIsBusError()

```
bool CanIsBusError (
            void  )
```

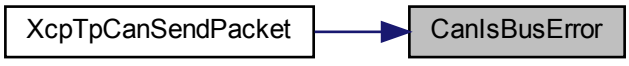
Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

Referenced by XcpTpCanSendPacket().

Here is the caller graph for this function:



7.1.2.4 CanIsConnected()

```
bool CanIsConnected (
            void  )
```

Obtains the connection state of the CAN module.

Returns

True if connected, false otherwise.

7.1.2.5 CanRegisterEvents()

```
void CanRegisterEvents (
            tCanEvents const * events )
```

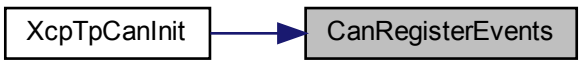
Registers the event callback functions that should be called by the CAN module.

Parameters

events	Pointer to structure with event callback function pointers.

Referenced by XcpTpCanInit().

Here is the caller graph for this function:



7.1.2.6 CanTransmit()

```
bool CanTransmit (
            tCanMsg const * msg )
```

Submits a message for transmission on the CAN bus.

Parameters

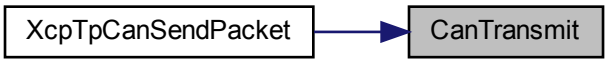
msg	Pointer to CAN message structure.

Returns

True if successful, false otherwise.

Referenced by XcpTpCanSendPacket().

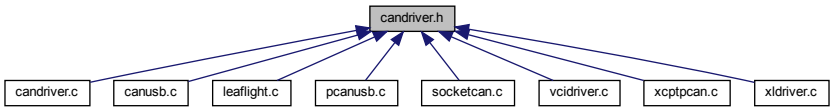
Here is the caller graph for this function:



7.2 candriver.h File Reference

Generic CAN driver header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tCanMsg

  Layout of a CAN message. Note that CAN_MSG_EXT_ID_MASK can be used to configure the CAN message identifier as 29-bit extended.

- struct tCanSettings

  Type to group of CAN interface related settings. The device name specifies the name of the CAN interface device. For some CAN interfaces this is don't care, but for other absolutely necessar, for example Linux SocketCAN. The channel specifies the channel on the CAN interface, in case it has multiple CAN channels. The baudrate specifies the communication speed on the CAN network. The code and mask values configure the message reception acceptance filter. A mask bit value of 0 means don't care. The code part of the filter determines what bit values to match in the received message identifier. Example 1: Receive all CAN identifiers .code = 0x00000000 .mask = 0x00000000 Example 2: Receive only CAN identifier 0x124 (11-bit or 29-bit) .code = 0x00000124 .mask = 0x1fffffff Example 3: Receive only CAN identifier 0x124 (11-bit) .code = 0x00000124 .mask = 0x9fffffff Example 4: Receive only CAN identifier 0x124 (29-bit) .code = 0x80000124 .mask = 0x9fffffff.

- struct tCanEvents

  Structure with CAN event callback functions.

- struct tCanInterface

  CAN interface type.

Macros

- #define **CAN_MSG_MAX_LEN** (8u)

  Maximum number of data bytes in a CAN message.
- #define CAN_MSG_EXT_ID_MASK (0x80000000u)

Enumerations

- enum tCanBaudrate {
  CAN_BR10K = 0 , CAN_BR20K = 1 , CAN_BR50K = 2 , CAN_BR100K = 3 ,
  CAN_BR125K = 4 , CAN_BR250K = 5 , CAN_BR500K = 6 , CAN_BR800K = 7 ,
  CAN_BR1M = 8 }

  Enumeration of the supported baudrates.

Functions

- void CanInit (tCanSettings const ∗settings)

  Initializes the CAN module. Typically called once at program startup.
- void **CanTerminate** (void)

  Terminates the CAN module. Typically called once at program cleanup.
- bool CanConnect (void)

  Connects the CAN module.
- void **CanDisconnect** (void)

  Disconnects the CAN module.
- bool CanIsConnected (void)

  Obtains the connection state of the CAN module.
- bool CanTransmit (tCanMsg const ∗msg)

  Submits a message for transmission on the CAN bus.
- bool CanIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- void CanRegisterEvents (tCanEvents const ∗events)

  Registers the event callback functions that should be called by the CAN module.

7.2.1 Detailed Description

Generic CAN driver header file.

7.2.2 Macro Definition Documentation

7.2.2.1 CAN_MSG_EXT_ID_MASK

```
#define CAN_MSG_EXT_ID_MASK (0x80000000u)
```

Bit mask that configures a CAN message identifier as 29-bit extended as opposed to 11-bit standard. Whenever this bit is set in the CAN identifier field of tCanMsg, then the CAN identifier is configured for 29-bit CAN extended.

7.2.3 Enumeration Type Documentation

7.2.3.1 tCanBaudrate

```
enum tCanBaudrate
```

Enumeration of the supported baudrates.

Enumerator

CAN_BR10K	10 kbits/sec
CAN_BR20K	20 kbits/sec
CAN_BR50K	50 kbits/sec
CAN_BR100K	100 kbits/sec
CAN_BR125K	125 kbits/sec
CAN_BR250K	250 kbits/sec
CAN_BR500K	500 kbits/sec
CAN_BR800K	800 kbits/sec
CAN_BR1M	1 Mbits/sec

7.2.4 Function Documentation

7.2.4.1 CanConnect()

```
bool CanConnect (
            void  )
```

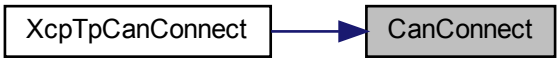
Connects the CAN module.

Returns

True if connected, false otherwise.

Referenced by XcpTpCanConnect().

Here is the caller graph for this function:



7.2.4.2 CanInit()

```
void CanInit (
            tCanSettings const * settings )
```

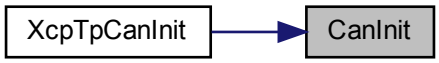
Initializes the CAN module. Typically called once at program startup.

Parameters

settings	Pointer to the CAN module settings.

Referenced by XcpTpCanInit().

Here is the caller graph for this function:



7.2.4.3 CanIsBusError()

```
bool CanIsBusError (
            void  )
```

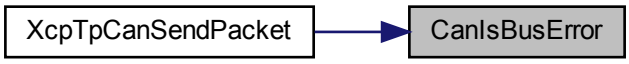
Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

Referenced by XcpTpCanSendPacket().

Here is the caller graph for this function:



7.2.4.4 CanIsConnected()

```
bool CanIsConnected (
            void  )
```

Obtains the connection state of the CAN module.

Returns

True if connected, false otherwise.

7.2.4.5 CanRegisterEvents()

```
void CanRegisterEvents (
            tCanEvents const * events )
```

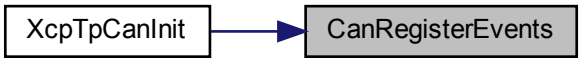
Registers the event callback functions that should be called by the CAN module.

Parameters

events	Pointer to structure with event callback function pointers.

Referenced by XcpTpCanInit().

Here is the caller graph for this function:



7.2.4.6 CanTransmit()

```
bool CanTransmit (
            tCanMsg const * msg )
```

Submits a message for transmission on the CAN bus.

Parameters

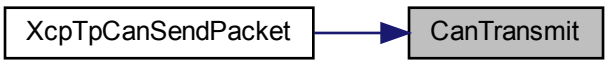
msg	Pointer to CAN message structure.

Returns

True if successful, false otherwise.

Referenced by XcpTpCanSendPacket().

Here is the caller graph for this function:



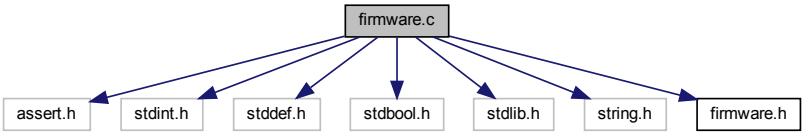
7.3 firmware.c File Reference

Firmware data module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
```

```
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "firmware.h"
```

Include dependency graph for firmware.c:



Functions

- static void FirmwareCreateSegment (uint32_t address, uint32_t len, uint8_t const ∗data)

  Creates and adds a new segment to the linked list. It allocates memory for the segment data and copies the data to it.
- static void FirmwareDeleteSegment (tFirmwareSegment const ∗segment)

  Deletes the specified segment from the linked list and handles the release of the segment's allocated memory.
- static void FirmwareTrimSegment (tFirmwareSegment const ∗segment, uint32_t address, uint32_t len)

  Removes the specified data range (address to address + len) from the segment. If if overlaps the entire segment, the segment will be deleted. Otherwise, the segment will be trimmed and, if needed, split into multiple segments.
- static void **FirmwareSortSegments** (void)

  Helper function to sort the segments in the linked list in order of ascending base address. It uses a bubble sort algorithm.
- static void **FirmwareMergeSegments** (void)

  Helper function to merge the segments in the linked list. When the firmware data in two adjacent segments also holds an adjacent range, then the firmware data from both segments are combined into one new one. Note that this function only works properly if the segments are already ordered. For this reasonse, the segments are explicitely sorted at the start.
- static uint32_t FirmwareGetFirstAddress (void)

  Helper function to obtain the first memory address of the firmware data that is present in the linked list with segments.
- static uint32_t FirmwareGetLastAddress (void)

  Helper function to obtain the last memory address of the firmware data that is present in the linked list with segments.
- void FirmwareInit (tFirmwareParser const ∗parser)

  Initializes the module.
- void **FirmwareTerminate** (void)

  Terminates the module.
- bool FirmwareLoadFromFile (char const ∗firmwareFile, uint32_t addressOffset)

  Uses the linked parser to load the firmware data from the specified file into the linked list of segments.
- bool FirmwareSaveToFile (char const ∗firmwareFile)

  Uses the linked parser to save the dat stored in the segments of the linked list to the specified file.
- uint32_t FirmwareGetSegmentCount (void)

  Obtains the total number of segments in the linked list with firmware data.
- tFirmwareSegment ∗ FirmwareGetSegment (uint32_t segmentIdx)

  Obtains the segment as the specified index from the linked list with firmware data.
- bool FirmwareAddData (uint32_t address, uint32_t len, uint8_t const ∗data)

  Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

- bool FirmwareRemoveData (uint32_t address, uint32_t len)

  Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary. Note that it is safe to assume in this function that the segments are already ordered in the linked list by ascending base memory address.
- void **FirmwareClearData** (void)

  Clears all data and segments that are currently present in the linked list.

Variables

- static tFirmwareParser const * **parserPtr**

  Pointer to the firmware parser that is linked.
- static tFirmwareSegment * **segmentList**

  Linked list with firmware segments.

7.3.1 Detailed Description

Firmware data module source file.

7.3.2 Function Documentation

7.3.2.1 FirmwareAddData()

```
bool FirmwareAddData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

Parameters

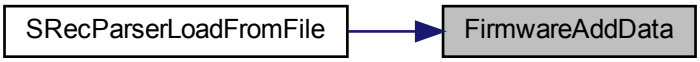
address	Base address of the firmware data.
len	Number of bytes to add.
data	Pointer to array with data bytes that should be added.

Returns

True if successful, false otherwise.

Referenced by SRecParserLoadFromFile().

Here is the caller graph for this function:



7.3.2.2 FirmwareCreateSegment()

```
static void FirmwareCreateSegment (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )  [static]
```

Creates and adds a new segment to the linked list. It allocates memory for the segment data and copies the data to it.

Parameters

address	Base address of the firmware data.
len	Number of bytes to add to the new segment.
data	Pointer to the byte array with data for the segment.

Referenced by FirmwareAddData(), and FirmwareTrimSegment().

Here is the caller graph for this function:

7.3.2.3 FirmwareDeleteSegment()

```
static void FirmwareDeleteSegment (
            tFirmwareSegment const * segment )  [static]
```

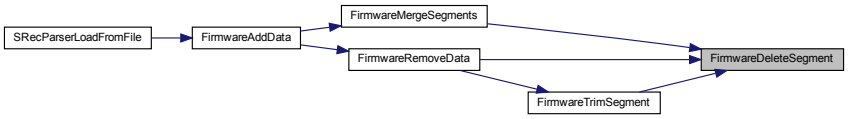
Deletes the specified segment from the linked list and handles the release of the segment's allocated memory.

Parameters

segment	Pointer to the segment.

Referenced by FirmwareMergeSegments(), FirmwareRemoveData(), and FirmwareTrimSegment().

Here is the caller graph for this function:



7.3.2.4 FirmwareGetFirstAddress()

```
static uint32_t FirmwareGetFirstAddress (
            void  )  [static]
```

Helper function to obtain the first memory address of the firmware data that is present in the linked list with segments.

Returns

The first memory address.

Referenced by FirmwareRemoveData().

Here is the caller graph for this function:

7.3.2.5 FirmwareGetLastAddress()

```
static uint32_t FirmwareGetLastAddress (
            void  )  [static]
```

Helper function to obtain the last memory address of the firmware data that is present in the linked list with segments.

Returns

The last memory address.

Referenced by FirmwareRemoveData().

Here is the caller graph for this function:

7.3.2.6 FirmwareGetSegment()

```
tFirmwareSegment * FirmwareGetSegment (
            uint32_t segmentIdx )
```

Obtains the segment as the specified index from the linked list with firmware data.

Parameters

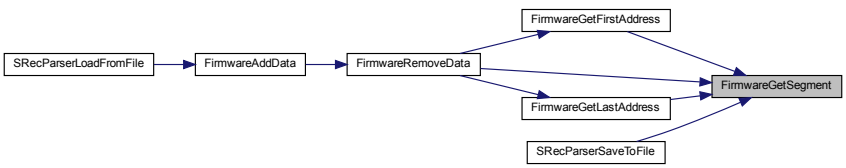
segmentIdx	The segment index. It should be a value greater or equal to zero and smaller than the value returned by FirmwareGetSegmentCount.

Returns

The segment if successful, NULL otherwise.

Referenced by FirmwareGetFirstAddress(), FirmwareGetLastAddress(), FirmwareRemoveData(), and SRecParserSaveToFile().

Here is the caller graph for this function:



7.3.2.7 FirmwareGetSegmentCount()

```
uint32_t FirmwareGetSegmentCount (
            void  )
```

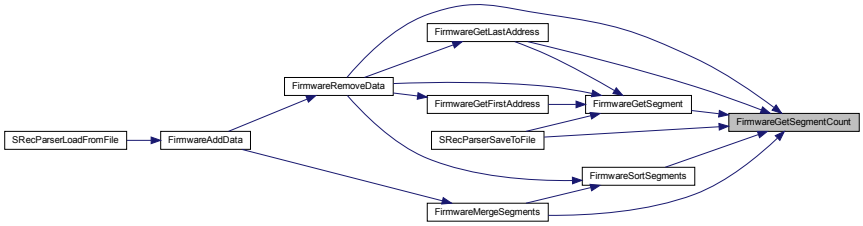
Obtains the total number of segments in the linked list with firmware data.

Returns

Total number of segments.

Referenced by FirmwareGetLastAddress(), FirmwareGetSegment(), FirmwareMergeSegments(), FirmwareRemoveData(), FirmwareSortSegments(), and SRecParserSaveToFile().

Here is the caller graph for this function:



7.3.2.8 FirmwareInit()

```
void FirmwareInit (
            tFirmwareParser const * parser )
```

Initializes the module.

Parameters

parser	The firmware file parser to link. It is okay to specify NULL if no file parser is needed.

7.3.2.9 FirmwareLoadFromFile()

```
bool FirmwareLoadFromFile (
            char const * firmwareFile,
            uint32_t addressOffset )
```

Uses the linked parser to load the firmware data from the specified file into the linked list of segments.

Parameters

firmwareFile	Filename of the firmware file to load.
addressOffset	Optional memory address offset to add when loading the firmware data from the file.

Returns

True if successful, false otherwise.

7.3.2.10 FirmwareRemoveData()

```
bool FirmwareRemoveData (
            uint32_t address,
            uint32_t len )
```

Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary. Note that it is safe to assume in this function that the segments are already ordered in the linked list by ascending base memory address.

Parameters

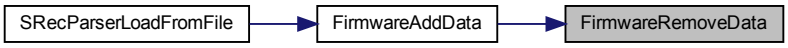
address	Base address of the firmware data.
len	Number of bytes to remove.

Returns

True if successful, false otherwise.

Referenced by FirmwareAddData().

Here is the caller graph for this function:



7.3.2.11 FirmwareSaveToFile()

```
bool FirmwareSaveToFile (
            char const * firmwareFile )
```

Uses the linked parser to save the dat stored in the segments of the linked list to the specified file.

Parameters

firmwareFile	Filename of the firmware file to write to.

Returns

True if successful, false otherwise.

7.3.2.12 FirmwareTrimSegment()

```
static void FirmwareTrimSegment (
            tFirmwareSegment const * segment,
            uint32_t address,
            uint32_t len )  [static]
```

Removes the specified data range (address to address + len) from the segment. If if overlaps the entire segment, the segment will be deleted. Otherwise, the segment will be trimmed and, if needed, split into multiple segments.

Parameters

segment	Pointer to the segment to trim.
address	Start address of the data that should be removed.
len	Total number of data bytes that should be removed.

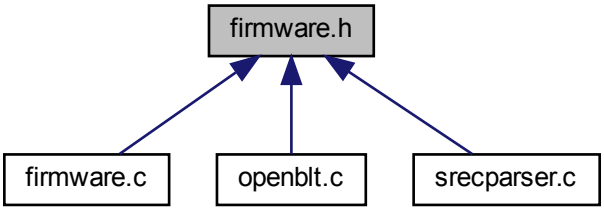
Referenced by FirmwareRemoveData().

Here is the caller graph for this function:

7.4 firmware.h File Reference

Firmware data module header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tFirmwareSegment

  Groups information together of a firmware segment, such that it can be used as a node in a linked list.
- struct tFirmwareParser

  Firmware file parser.

Functions

- void FirmwareInit (tFirmwareParser const ∗parser)

  Initializes the module.
- void **FirmwareTerminate** (void)

  Terminates the module.
- bool FirmwareLoadFromFile (char const ∗firmwareFile, uint32_t addressOffset)

  Uses the linked parser to load the firmware data from the specified file into the linked list of segments.
- bool FirmwareSaveToFile (char const ∗firmwareFile)

  Uses the linked parser to save the dat stored in the segments of the linked list to the specified file.
- uint32_t FirmwareGetSegmentCount (void)

  Obtains the total number of segments in the linked list with firmware data.
- tFirmwareSegment ∗ FirmwareGetSegment (uint32_t segmentIdx)

  Obtains the segment as the specified index from the linked list with firmware data.
- bool FirmwareAddData (uint32_t address, uint32_t len, uint8_t const ∗data)

  Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.
- bool FirmwareRemoveData (uint32_t address, uint32_t len)

  Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary. Note that it is safe to assume in this function that the segments are already ordered in the linked list by ascending base memory address.
- void **FirmwareClearData** (void)

  Clears all data and segments that are currently present in the linked list.

7.4.1 Detailed Description

Firmware data module header file.

7.4.2 Function Documentation

7.4.2.1 FirmwareAddData()

```
bool FirmwareAddData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

Parameters

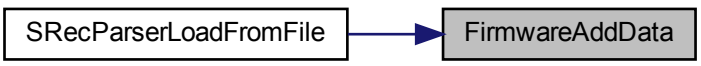
address	Base address of the firmware data.
len	Number of bytes to add.
data	Pointer to array with data bytes that should be added.

Returns

True if successful, false otherwise.

Referenced by SRecParserLoadFromFile().

Here is the caller graph for this function:



7.4.2.2 FirmwareGetSegment()

```
tFirmwareSegment * FirmwareGetSegment (
            uint32_t segmentIdx )
```

Obtains the segment as the specified index from the linked list with firmware data.

Parameters

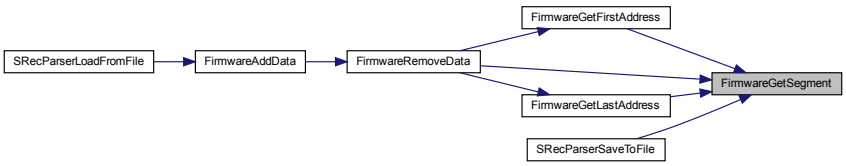
segmentIdx	The segment index. It should be a value greater or equal to zero and smaller than the value returned by FirmwareGetSegmentCount.

Returns

The segment if successful, NULL otherwise.

Referenced by FirmwareGetFirstAddress(), FirmwareGetLastAddress(), FirmwareRemoveData(), and SRecParserSaveToFile().

Here is the caller graph for this function:



7.4.2.3 FirmwareGetSegmentCount()

```
uint32_t FirmwareGetSegmentCount (
            void  )
```

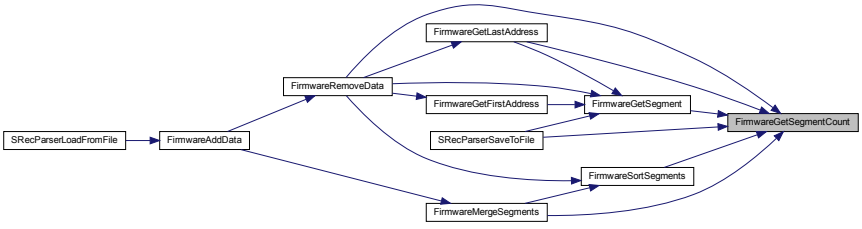
Obtains the total number of segments in the linked list with firmware data.

Returns

Total number of segments.

Referenced by FirmwareGetLastAddress(), FirmwareGetSegment(), FirmwareMergeSegments(), FirmwareRemoveData(), FirmwareSortSegments(), and SRecParserSaveToFile().

Here is the caller graph for this function:



7.4.2.4 FirmwareInit()

```
void FirmwareInit (
            tFirmwareParser const * parser )
```

Initializes the module.

Parameters

parser	The firmware file parser to link. It is okay to specify NULL if no file parser is needed.

7.4.2.5 FirmwareLoadFromFile()

```
bool FirmwareLoadFromFile (
            char const * firmwareFile,
            uint32_t addressOffset )
```

Uses the linked parser to load the firmware data from the specified file into the linked list of segments.

Parameters

firmwareFile	Filename of the firmware file to load.
addressOffset	Optional memory address offset to add when loading the firmware data from the file.

Returns

True if successful, false otherwise.

7.4.2.6 FirmwareRemoveData()

```
bool FirmwareRemoveData (
            uint32_t address,
            uint32_t len )
```

Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary. Note that it is safe to assume in this function that the segments are already ordered in the linked list by ascending base memory address.

Parameters

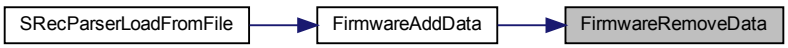
address	Base address of the firmware data.
len	Number of bytes to remove.

Returns

True if successful, false otherwise.

Referenced by FirmwareAddData().

Here is the caller graph for this function:



7.4.2.7 FirmwareSaveToFile()

```
bool FirmwareSaveToFile (
            char const * firmwareFile )
```

Uses the linked parser to save the dat stored in the segments of the linked list to the specified file.

Parameters

firmwareFile	Filename of the firmware file to write to.

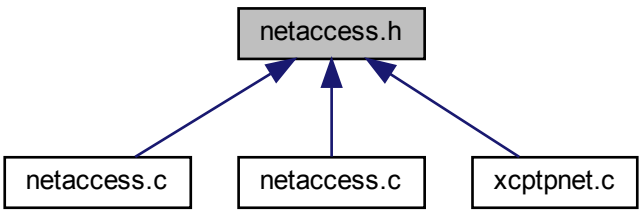
Returns

True if successful, false otherwise.

7.5 netaccess.h File Reference

TCP/IP network access header file.

This graph shows which files directly or indirectly include this file:



Functions

- void **NetAccessInit** (void)

  Initializes the network access module.
- void **NetAccessTerminate** (void)

  Terminates the network access module.
- bool NetAccessConnect (char const ∗address, uint16_t port)

  Connects to the TCP/IP server at the specified address and the given port.
- void **NetAccessDisconnect** (void)

  Disconnects from the TCP/IP server.
- bool NetAccessSend (uint8_t const ∗data, uint32_t length)

  Sends data to the TCP/IP server.
- bool NetAccessReceive (uint8_t ∗data, uint32_t ∗length, uint32_t timeout)

  Receives data from the TCP/IP server in a blocking manner.

7.5.1 Detailed Description

TCP/IP network access header file.

7.5.2 Function Documentation

7.5.2.1 NetAccessConnect()

```
bool NetAccessConnect (
            char const * address,
            uint16_t port )
```

Connects to the TCP/IP server at the specified address and the given port.

Parameters

address	The address of the server. This can be a hostname (such as mydomain.com) or an IP address (such as 127.0.0.1).
port	The port number on the server to connect to.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetConnect().

Here is the caller graph for this function:

7.5.2.2 NetAccessReceive()

```
bool NetAccessReceive (
            uint8_t * data,
            uint32_t * length,
            uint32_t timeout )
```

Receives data from the TCP/IP server in a blocking manner.

Parameters

data	Pointer to byte array to store the received data.
length	Holds the max number of bytes that can be stored into the byte array. This function also overwrites this value with the number of bytes that were actually received.
timeout	Timeout in milliseconds for the data reception.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetSendPacket().

Here is the caller graph for this function:

7.5.2.3 NetAccessSend()

```
bool NetAccessSend (
            uint8_t const * data,
            uint32_t length )
```

Sends data to the TCP/IP server.

Parameters

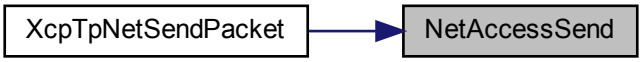
data	Pointer to byte array with data to send.
length	Number of bytes to send.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetSendPacket().

Here is the caller graph for this function:

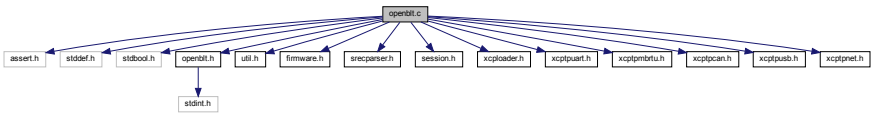


7.6 openblt.c File Reference

OpenBLT host library source file.

```
#include <assert.h>
#include <stddef.h>
#include <stdbool.h>
#include "openblt.h"
#include "util.h"
#include "firmware.h"
#include "srecparser.h"
#include "session.h"
#include "xcploader.h"
#include "xcptpuart.h"
#include "xcptpmbrtu.h"
#include "xcptpcan.h"
#include "xcptpusb.h"
#include "xcptpnet.h"
```

Include dependency graph for openblt.c:



Macros

- #define **BLT_VERSION_NUMBER** (10312u)

  The version number of the library as an integer. The number has two digits for major-, minor-, and patch-version. Version 1.05.12 would for example be 10512.
- #define **BLT_VERSION_STRING** "1.03.12"

  The version number of the library as a null-terminated string.

Functions

- LIBOPENBLT_EXPORT uint32_t BltVersionGetNumber (void)

  Obtains the version number of the library as an integer. The number has two digits for major-, minor-, and patch-version. Version 1.05.12 would for example return 10512.
- LIBOPENBLT_EXPORT char const * BltVersionGetString (void)

  Obtains the version number of the library as a null-terminated string. Version 1.05.12 would for example return "1.05.12".
- LIBOPENBLT_EXPORT void BltSessionInit (uint32_t sessionType, void const *sessionSettings, uint32_t transportType, void const *transportSettings)

  Initializes the firmware update session for a specific communication protocol and transport layer. This function is typically called once at the start of the firmware update.
- LIBOPENBLT_EXPORT void **BltSessionTerminate** (void)

  Terminates the firmware update session. This function is typically called once at the end of the firmware update.
- LIBOPENBLT_EXPORT uint32_t BltSessionStart (void)

  Starts the firmware update session. This is were the library attempts to activate and connect with the bootloader running on the target, through the transport layer that was specified during the session's initialization.
- LIBOPENBLT_EXPORT void **BltSessionStop** (void)

  Stops the firmware update session. This is there the library disconnects the transport layer as well.
- LIBOPENBLT_EXPORT uint32_t BltSessionClearMemory (uint32_t address, uint32_t len)

  Requests the target to erase the specified range of memory on the target. Note that the target automatically aligns this to the erasable memory block sizes. This typically results in more memory being erased than the range that was specified here. Refer to the target implementation for details.
- LIBOPENBLT_EXPORT uint32_t BltSessionWriteData (uint32_t address, uint32_t len, uint8_t const *data)

  Requests the target to program the specified data to memory. Note that it is the responsibility of the application to make sure the memory range was erased beforehand.
- LIBOPENBLT_EXPORT uint32_t BltSessionReadData (uint32_t address, uint32_t len, uint8_t *data)

  Requests the target to upload the specified range from memory and store its contents in the specified data buffer.
- LIBOPENBLT_EXPORT void BltFirmwareInit (uint32_t parserType)

  Initializes the firmware data module for a specified firmware file parser.
- LIBOPENBLT_EXPORT void **BltFirmwareTerminate** (void)

  Terminates the firmware data module. Typically called at the end of the program when the firmware data module is no longer needed.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareLoadFromFile (char const *firmwareFile, uint32_t addressOffset)

  Loads firmware data from the specified file using the firmware file parser that was specified during the initialization of this module.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareSaveToFile (char const *firmwareFile)

  Writes firmware data to the specified file using the firmware file parser that was specified during the initialization of this module.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareGetSegmentCount (void)

  Obtains the number of firmware data segments that are currently present in the firmware data module.
- LIBOPENBLT_EXPORT uint8_t * BltFirmwareGetSegment (uint32_t idx, uint32_t *address, uint32_t *len)

  Obtains the contents of the firmware data segment that was specified by the index parameter.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareAddData (uint32_t address, uint32_t len, uint8_t const *data)

Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

- LIBOPENBLT_EXPORT uint32_t BltFirmwareRemoveData (uint32_t address, uint32_t len)

  Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary.
- LIBOPENBLT_EXPORT void **BltFirmwareClearData** (void)

  Clears all data and segments that are currently present in the firmware data module.
- LIBOPENBLT_EXPORT uint16_t BltUtilCrc16Calculate (uint8_t const ∗data, uint32_t len)

  Calculates a 16-bit CRC value over the specified data.
- LIBOPENBLT_EXPORT uint32_t BltUtilCrc32Calculate (uint8_t const ∗data, uint32_t len)

  Calculates a 32-bit CRC value over the specified data.
- LIBOPENBLT_EXPORT uint32_t BltUtilTimeGetSystemTime (void)

  Get the system time in milliseconds.
- LIBOPENBLT_EXPORT void BltUtilTimeDelayMs (uint16_t delay)

  Performs a delay of the specified amount of milliseconds.
- LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Encrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.
- LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Decrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Variables

- char const **bltVersionString** [ ] = BLT_VERSION_STRING

  Constant null-terminated string with the version number of the library.

7.6.1 Detailed Description

OpenBLT host library source file.

7.6.2 Function Documentation

7.6.2.1 BltFirmwareAddData()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareAddData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

Parameters

address	Base address of the firmware data.
len	Number of bytes to add.
data	Pointer to array with data bytes that should be added.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.2 BltFirmwareGetSegment()

```
LIBOPENBLT_EXPORT uint8_t * BltFirmwareGetSegment (
            uint32_t idx,
            uint32_t * address,
            uint32_t * len )
```

Obtains the contents of the firmware data segment that was specified by the index parameter.

Parameters

idx	The segment index. It should be a value greater or equal to zero and smaller than the value returned by BltFirmwareGetSegmentCount.
address	Pointer to where the segment's base address will be written to.
len	Pointer to where the segment's length will be written to.

Returns

Pointer to the segment data if successful, NULL otherwise.

7.6.2.3 BltFirmwareGetSegmentCount()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareGetSegmentCount (
            void  )
```

Obtains the number of firmware data segments that are currently present in the firmware data module.

Returns

The total number of segments.

7.6.2.4 BltFirmwareInit()

```
LIBOPENBLT_EXPORT void BltFirmwareInit (
            uint32_t parserType )
```

Initializes the firmware data module for a specified firmware file parser.

Parameters

parserType	The firmware file parser to use in this module. It should be a BLT_FIRMWARE_PARSER_xxx value.

7.6.2.5 BltFirmwareLoadFromFile()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareLoadFromFile (
            char const * firmwareFile,
            uint32_t addressOffset )
```

Loads firmware data from the specified file using the firmware file parser that was specified during the initialization of this module.

Parameters

firmwareFile	Filename of the firmware file to load.
addressOffset	Optional memory address offset to add when loading the firmware data from the file. This is typically only useful when loading firmware data from a binary formatted firmware file.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.6 BltFirmwareRemoveData()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareRemoveData (
            uint32_t address,
            uint32_t len )
```

Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary.

Parameters

address	Base address of the firmware data.
len	Number of bytes to remove.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.7 BltFirmwareSaveToFile()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareSaveToFile (
            char const * firmwareFile )
```

Writes firmware data to the specified file using the firmware file parser that was specified during the initialization of this module.

Parameters

firmwareFile	Filename of the firmware file to write to.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.8 BltSessionClearMemory()

```
LIBOPENBLT_EXPORT uint32_t BltSessionClearMemory (
            uint32_t address,
            uint32_t len )
```

Requests the target to erase the specified range of memory on the target. Note that the target automatically aligns this to the erasable memory block sizes. This typically results in more memory being erased than the range that was specified here. Refer to the target implementation for details.

Parameters

address	The starting memory address for the erase operation.
len	The total number of bytes to erase from memory.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.9 BltSessionInit()

```
LIBOPENBLT_EXPORT void BltSessionInit (
            uint32_t sessionType,
            void const * sessionSettings,
            uint32_t transportType,
            void const * transportSettings )
```

Initializes the firmware update session for a specific communication protocol and transport layer. This function is typically called once at the start of the firmware update.

Parameters

sessionType	The communication protocol to use for this session. It should be a BLT_SESSION_xxx value.
sessionSettings	Pointer to a structure with communication protocol specific settings.
transportType	The transport layer to use for the specified communication protocol. It should be a BLT_TRANSPORT_xxx value.
transportSettings	Pointer to a structure with transport layer specific settings.

7.6.2.10 BltSessionReadData()

```
LIBOPENBLT_EXPORT uint32_t BltSessionReadData (
            uint32_t address,
            uint32_t len,
            uint8_t * data )
```

Requests the target to upload the specified range from memory and store its contents in the specified data buffer.

Parameters

address	The starting memory address for the read operation.
len	The number of bytes to upload from the target and store in the data buffer.
data	Pointer to the byte array where the uploaded data should be stored.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.11 BltSessionStart()

```
LIBOPENBLT_EXPORT uint32_t BltSessionStart (
            void  )
```

Starts the firmware update session. This is were the library attempts to activate and connect with the bootloader running on the target, through the transport layer that was specified during the session's initialization.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.12 BltSessionWriteData()

```
LIBOPENBLT_EXPORT uint32_t BltSessionWriteData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Requests the target to program the specified data to memory. Note that it is the responsibility of the application to make sure the memory range was erased beforehand.

Parameters

address	The starting memory address for the write operation.
len	The number of bytes in the data buffer that should be written.
data	Pointer to the byte array with data to write.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.13 BltUtilCrc16Calculate()

```
LIBOPENBLT_EXPORT uint16_t BltUtilCrc16Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 16-bit CRC value over the specified data.

Parameters

data	Array with bytes over which the CRC16 should be calculated.
len	Number of bytes in the data array.

Returns

The 16-bit CRC value.

7.6.2.14 BltUtilCrc32Calculate()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCrc32Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 32-bit CRC value over the specified data.

Parameters

data	Array with bytes over which the CRC32 should be calculated.
len	Number of bytes in the data array.

Returns

The 32-bit CRC value.

7.6.2.15 BltUtilCryptoAes256Decrypt()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Decrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to decrypt. The decrypted bytes are stored in the same array.
len	The number of bytes in the data-array to decrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit decryption key as a array of 32 bytes.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.16 BltUtilCryptoAes256Encrypt()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Encrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to encrypt. The encrypted bytes are stored in the same array.
len	The number of bytes in the data-array to encrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit encryption key as a array of 32 bytes.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.6.2.17 BltUtilTimeDelayMs()

```
LIBOPENBLT_EXPORT void BltUtilTimeDelayMs (
            uint16_t delay )
```

Performs a delay of the specified amount of milliseconds.

Parameters

delay	Delay time in milliseconds.

7.6.2.18 BltUtilTimeGetSystemTime()

```
LIBOPENBLT_EXPORT uint32_t BltUtilTimeGetSystemTime (
            void  )
```

Get the system time in milliseconds.

Returns

Time in milliseconds.

7.6.2.19 BltVersionGetNumber()

```
LIBOPENBLT_EXPORT uint32_t BltVersionGetNumber (
            void  )
```

Obtains the version number of the library as an integer. The number has two digits for major-, minor-, and patch-version. Version 1.05.12 would for example return 10512.

Returns

Library version number as an integer.

7.6.2.20 BltVersionGetString()

```
LIBOPENBLT_EXPORT char const * BltVersionGetString (
            void  )
```

Obtains the version number of the library as a null-terminated string. Version 1.05.12 would for example return "1.05.12".

Returns

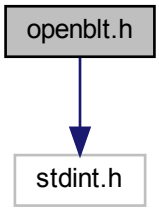
Library version number as a null-terminated string.

7.7 openblt.h File Reference

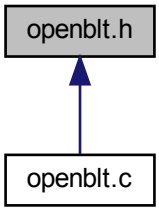
OpenBLT host library header file.

```
#include <stdint.h>
```

Include dependency graph for openblt.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct tBltSessionSettingsXcpV10

  Structure layout of the XCP version 1.0 session settings.
- struct tBltTransportSettingsXcpV10Rs232

  Structure layout of the XCP version 1.0 RS232 transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".
- struct tBltTransportSettingsXcpV10Can

  Structure layout of the XCP version 1.0 CAN transport layer settings. The deviceName field is platform dependent. On Linux based systems this should be the socketCAN interface name such as "can0". The terminal command "ip addr" can be issued to view a list of interfaces that are up and available. Under Linux it is assumed that the socketCAN interface is already configured on the system, before using the OpenBLT library. When baudrate is configured when bringing up the system, so the baudrate field in this structure is don't care when using the library on a Linux was system. On Windows based systems, the device name is a name that is pre-defined by this library for the supported CAN adapters. The device name should be one of the following: "peak_pcanusb", "kvaser_leaflight", or "lawicel_canusb". Field use extended is a boolean field. When set to 0, the specified transmitId and receiveId are assumed to be 11-bit standard CAN identifier. It the field is 1, these identifiers are assumed to be 29-bit extended CAN identifiers.

- struct tBltTransportSettingsXcpV10Net

  Structure layout of the XCP version 1.0 NET transport layer settings. The address field can be set to either the IP address or the hostname, such as "192.168.178.23" or "mymicro.mydomain.com". The port should be set to the TCP port number that the bootloader target listens on.
- struct tBltTransportSettingsXcpV10MbRtu

  Structure layout of the XCP version 1.0 Modbus RTU transport layer settings. The portName field is platform dependent. On Linux based systems this should be the filename of the tty-device, such as "/dev/tty0". On Windows based systems it should be the name of the COM-port, such as "COM1".

Macros

- #define **BLT_RESULT_OK** (0u)

  Function return value for when everything went okay.
- #define **BLT_RESULT_ERROR_GENERIC** (1u)

  Function return value for when a generic error occured.
- #define **BLT_SESSION_XCP_V10** ((uint32_t)0u)

  XCP protocol version 1.0. XCP is a universal measurement and calibration communication protocol. It contains functionality for reading, programming, and erasing (non-volatile) memory making it a good fit for bootloader purposes.
- #define **BLT_TRANSPORT_XCP_V10_RS232** ((uint32_t)0u)

  Transport layer for the XCP v1.0 protocol that uses RS-232 serial communication for data exchange.
- #define **BLT_TRANSPORT_XCP_V10_CAN** ((uint32_t)1u)

  Transport layer for the XCP v1.0 protocol that uses Controller Area Network (CAN) for data exchange.
- #define **BLT_TRANSPORT_XCP_V10_USB** ((uint32_t)2u)

  Transport layer for the XCP v1.0 protocol that uses USB Bulk for data exchange.
- #define **BLT_TRANSPORT_XCP_V10_NET** ((uint32_t)3u)

  Transport layer for the XCP v1.0 protocol that uses TCP/IP for data exchange.
- #define **BLT_TRANSPORT_XCP_V10_MBRTU** ((uint32_t)4u)

  Transport layer for the XCP v1.0 protocol that uses Modbus RTU communication for data exchange.
- #define **BLT_FIRMWARE_PARSER_SRECORD** ((uint32_t)0u)

  The S-record parser enables writing and reading firmware data to and from file formatted as Motorola S-record. This is a widely known file format and pretty much all microcontroller compiler toolchains included functionality to output or convert the firmware's data as an S-record.

Functions

- LIBOPENBLT_EXPORT uint32_t BltVersionGetNumber (void)

  Obtains the version number of the library as an integer. The number has two digits for major-, minor-, and patch-version. Version 1.05.12 would for example return 10512.
- LIBOPENBLT_EXPORT char const * BltVersionGetString (void)

  Obtains the version number of the library as a null-terminated string. Version 1.05.12 would for example return "1.05.12".
- LIBOPENBLT_EXPORT void BltSessionInit (uint32_t sessionType, void const *sessionSettings, uint32_t transportType, void const *transportSettings)

  Initializes the firmware update session for a specific communication protocol and transport layer. This function is typically called once at the start of the firmware update.
- LIBOPENBLT_EXPORT void **BltSessionTerminate** (void)

  Terminates the firmware update session. This function is typically called once at the end of the firmware update.
- LIBOPENBLT_EXPORT uint32_t BltSessionStart (void)

  Starts the firmware update session. This is were the library attempts to activate and connect with the bootloader running on the target, through the transport layer that was specified during the session's initialization.
- LIBOPENBLT_EXPORT void **BltSessionStop** (void)

  Stops the firmware update session. This is there the library disconnects the transport layer as well.

- LIBOPENBLT_EXPORT uint32_t BltSessionClearMemory (uint32_t address, uint32_t len)

  Requests the target to erase the specified range of memory on the target. Note that the target automatically aligns this to the erasable memory block sizes. This typically results in more memory being erased than the range that was specified here. Refer to the target implementation for details.
- LIBOPENBLT_EXPORT uint32_t BltSessionWriteData (uint32_t address, uint32_t len, uint8_t const *data)

  Requests the target to program the specified data to memory. Note that it is the responsibility of the application to make sure the memory range was erased beforehand.
- LIBOPENBLT_EXPORT uint32_t BltSessionReadData (uint32_t address, uint32_t len, uint8_t *data)

  Requests the target to upload the specified range from memory and store its contents in the specified data buffer.
- LIBOPENBLT_EXPORT void BltFirmwareInit (uint32_t parserType)

  Initializes the firmware data module for a specified firmware file parser.
- LIBOPENBLT_EXPORT void **BltFirmwareTerminate** (void)

  Terminates the firmware data module. Typically called at the end of the program when the firmware data module is no longer needed.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareLoadFromFile (char const *firmwareFile, uint32_t address↩ Offset)

  Loads firmware data from the specified file using the firmware file parser that was specified during the initialization of this module.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareSaveToFile (char const *firmwareFile)

  Writes firmware data to the specified file using the firmware file parser that was specified during the initialization of this module.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareGetSegmentCount (void)

  Obtains the number of firmware data segments that are currently present in the firmware data module.
- LIBOPENBLT_EXPORT uint8_t * BltFirmwareGetSegment (uint32_t idx, uint32_t *address, uint32_t *len)

  Obtains the contents of the firmware data segment that was specified by the index parameter.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareAddData (uint32_t address, uint32_t len, uint8_t const *data)

  Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.
- LIBOPENBLT_EXPORT uint32_t BltFirmwareRemoveData (uint32_t address, uint32_t len)

  Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary.
- LIBOPENBLT_EXPORT void **BltFirmwareClearData** (void)

  Clears all data and segments that are currently present in the firmware data module.
- LIBOPENBLT_EXPORT uint16_t BltUtilCrc16Calculate (uint8_t const *data, uint32_t len)

  Calculates a 16-bit CRC value over the specified data.
- LIBOPENBLT_EXPORT uint32_t BltUtilCrc32Calculate (uint8_t const *data, uint32_t len)

  Calculates a 32-bit CRC value over the specified data.
- LIBOPENBLT_EXPORT uint32_t BltUtilTimeGetSystemTime (void)

  Get the system time in milliseconds.
- LIBOPENBLT_EXPORT void BltUtilTimeDelayMs (uint16_t delay)

  Performs a delay of the specified amount of milliseconds.
- LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Encrypt (uint8_t *data, uint32_t len, uint8_t const *key)

  Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.
- LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Decrypt (uint8_t *data, uint32_t len, uint8_t const *key)

  Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

7.7.1 Detailed Description

OpenBLT host library header file.

7.7.2 Function Documentation

7.7.2.1 BltFirmwareAddData()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareAddData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Adds data to the segments that are currently present in the firmware data module. If the data overlaps with already existing data, the existing data gets overwritten. The size of a segment is automatically adjusted or a new segment gets created, if necessary.

Parameters

address	Base address of the firmware data.
len	Number of bytes to add.
data	Pointer to array with data bytes that should be added.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.2 BltFirmwareGetSegment()

```
LIBOPENBLT_EXPORT uint8_t * BltFirmwareGetSegment (
            uint32_t idx,
            uint32_t * address,
            uint32_t * len )
```

Obtains the contents of the firmware data segment that was specified by the index parameter.

Parameters

idx	The segment index. It should be a value greater or equal to zero and smaller than the value returned by BltFirmwareGetSegmentCount.
address	Pointer to where the segment's base address will be written to.
len	Pointer to where the segment's length will be written to.

Returns

Pointer to the segment data if successful, NULL otherwise.

7.7.2.3 BltFirmwareGetSegmentCount()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareGetSegmentCount (
            void  )
```

Obtains the number of firmware data segments that are currently present in the firmware data module.

Returns

The total number of segments.

7.7.2.4 BltFirmwareInit()

```
LIBOPENBLT_EXPORT void BltFirmwareInit (
            uint32_t parserType )
```

Initializes the firmware data module for a specified firmware file parser.

Parameters

parserType	The firmware file parser to use in this module. It should be a BLT_FIRMWARE_PARSER_xxx value.

7.7.2.5 BltFirmwareLoadFromFile()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareLoadFromFile (
            char const * firmwareFile,
            uint32_t addressOffset )
```

Loads firmware data from the specified file using the firmware file parser that was specified during the initialization of this module.

Parameters

firmwareFile	Filename of the firmware file to load.
addressOffset	Optional memory address offset to add when loading the firmware data from the file. This is typically only useful when loading firmware data from a binary formatted firmware file.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.6 BltFirmwareRemoveData()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareRemoveData (
            uint32_t address,
            uint32_t len )
```

Removes data from the segments that are currently present in the firmware data module. The size of a segment is automatically adjusted or removed, if necessary.

Parameters

address	Base address of the firmware data.
len	Number of bytes to remove.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.7 BltFirmwareSaveToFile()

```
LIBOPENBLT_EXPORT uint32_t BltFirmwareSaveToFile (
            char const * firmwareFile )
```

Writes firmware data to the specified file using the firmware file parser that was specified during the initialization of this module.

Parameters

firmwareFile	Filename of the firmware file to write to.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.8 BltSessionClearMemory()

```
LIBOPENBLT_EXPORT uint32_t BltSessionClearMemory (
            uint32_t address,
            uint32_t len )
```

Requests the target to erase the specified range of memory on the target. Note that the target automatically aligns this to the erasable memory block sizes. This typically results in more memory being erased than the range that was specified here. Refer to the target implementation for details.

Parameters

address	The starting memory address for the erase operation.
len	The total number of bytes to erase from memory.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.9 BltSessionInit()

```
LIBOPENBLT_EXPORT void BltSessionInit (
            uint32_t sessionType,
            void const * sessionSettings,
            uint32_t transportType,
            void const * transportSettings )
```

Initializes the firmware update session for a specific communication protocol and transport layer. This function is typically called once at the start of the firmware update.

Parameters

sessionType	The communication protocol to use for this session. It should be a BLT_SESSION_xxx value.
sessionSettings	Pointer to a structure with communication protocol specific settings.
transportType	The transport layer to use for the specified communication protocol. It should be a BLT_TRANSPORT_xxx value.
transportSettings	Pointer to a structure with transport layer specific settings.

7.7.2.10 BltSessionReadData()

```
LIBOPENBLT_EXPORT uint32_t BltSessionReadData (
            uint32_t address,
            uint32_t len,
            uint8_t * data )
```

Requests the target to upload the specified range from memory and store its contents in the specified data buffer.

Parameters

address	The starting memory address for the read operation.
len	The number of bytes to upload from the target and store in the data buffer.
data	Pointer to the byte array where the uploaded data should be stored.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.11 BltSessionStart()

```
LIBOPENBLT_EXPORT uint32_t BltSessionStart (
            void  )
```

Starts the firmware update session. This is were the library attempts to activate and connect with the bootloader running on the target, through the transport layer that was specified during the session's initialization.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.12 BltSessionWriteData()

```
LIBOPENBLT_EXPORT uint32_t BltSessionWriteData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Requests the target to program the specified data to memory. Note that it is the responsibility of the application to make sure the memory range was erased beforehand.

Parameters

address	The starting memory address for the write operation.
len	The number of bytes in the data buffer that should be written.
data	Pointer to the byte array with data to write.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.13 BltUtilCrc16Calculate()

```
LIBOPENBLT_EXPORT uint16_t BltUtilCrc16Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 16-bit CRC value over the specified data.

Parameters

data	Array with bytes over which the CRC16 should be calculated.
len	Number of bytes in the data array.

Returns

The 16-bit CRC value.

7.7.2.14 BltUtilCrc32Calculate()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCrc32Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 32-bit CRC value over the specified data.

Parameters

data	Array with bytes over which the CRC32 should be calculated.
len	Number of bytes in the data array.

Returns

The 32-bit CRC value.

7.7.2.15 BltUtilCryptoAes256Decrypt()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Decrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to decrypt. The decrypted bytes are stored in the same array.
len	The number of bytes in the data-array to decrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit decryption key as a array of 32 bytes.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.16 BltUtilCryptoAes256Encrypt()

```
LIBOPENBLT_EXPORT uint32_t BltUtilCryptoAes256Encrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to encrypt. The encrypted bytes are stored in the same array.
len	The number of bytes in the data-array to encrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit encryption key as a array of 32 bytes.

Returns

BLT_RESULT_OK if successful, BLT_RESULT_ERROR_xxx otherwise.

7.7.2.17 BltUtilTimeDelayMs()

```
LIBOPENBLT_EXPORT void BltUtilTimeDelayMs (
            uint16_t delay )
```

Performs a delay of the specified amount of milliseconds.

Parameters

delay	Delay time in milliseconds.

7.7.2.18 BltUtilTimeGetSystemTime()

```
LIBOPENBLT_EXPORT uint32_t BltUtilTimeGetSystemTime (
            void  )
```

Get the system time in milliseconds.

Returns

Time in milliseconds.

7.7.2.19 BltVersionGetNumber()

```
LIBOPENBLT_EXPORT uint32_t BltVersionGetNumber (
            void  )
```

Obtains the version number of the library as an integer. The number has two digits for major-, minor-, and patch-version. Version 1.05.12 would for example return 10512.

Returns

Library version number as an integer.

7.7.2.20 BltVersionGetString()

```
LIBOPENBLT_EXPORT char const * BltVersionGetString (
            void  )
```

Obtains the version number of the library as a null-terminated string. Version 1.05.12 would for example return "1.05.12".

Returns

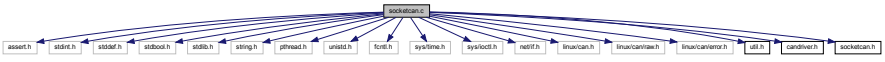
Library version number as a null-terminated string.

7.8 socketcan.c File Reference

Linux SocketCAN interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <linux/can/error.h>
#include "util.h"
#include "candriver.h"
#include "socketcan.h"
```

Include dependency graph for socketcan.c:



Data Structures

- struct tSocketCanThreadCtrl

  Groups data for thread control.

Functions

- static void SocketCanInit (tCanSettings const *settings)

  Initializes the CAN interface. Note that this module assumes that the CAN device was already properly configured and brought online on the Linux system. Terminal command "ip addr" can be used to verify this.
- static void **SocketCanTerminate** (void)

  Terminates the CAN interface.
- static bool SocketCanConnect (void)

  Connects the CAN interface. Note that the channel and baudrate settings are ignored for the SocketCAN, because these are expected to be configured when the CAN device was brought online on the Linux system.
- static void **SocketCanDisconnect** (void)

  Disconnects the CAN interface.
- static bool SocketCanTransmit (tCanMsg const *msg)

  Submits a message for transmission on the CAN bus.
- static bool SocketCanIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void SocketCanRegisterEvents (tCanEvents const *events)

  Registers the event callback functions that should be called by the CAN interface.
- static bool SocketCanStartEventThread (void)

  Starts the event thread.
- static void SocketCanStopEventThread (void)

  Stops the event thread. It sets the termination request and then waits for the termination handshake.
- static void * SocketCanEventThread (void *param)

  Event thread that handles the asynchronous reception of data from the CAN interface.
- tCanInterface const * SocketCanGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **socketCanInterface**

  CAN interface structure filled with SocketCAN specifics.
- static tCanSettings **socketCanSettings**

  The settings to use in this CAN interface.
- static volatile tCanEvents * **socketCanEventsList**

  List with callback functions that this driver should use.
- static volatile uint32_t **socketCanEventsEntries**

  Total number of event entries into the socketCanEventsList list.
- static volatile bool **socketCanErrorDetected**

  Flag to set in the event thread when either a bus off or bus heavy situation.
- static volatile tSocketCanThreadCtrl **eventThreadCtrl**

  Event thread control.
- static pthread_t **eventThreadId**

  The ID of the event thread.
- static volatile int32_t **canSocket**

  CAN raw socket.

7.8.1 Detailed Description

Linux SocketCAN interface source file.

7.8.2 Function Documentation

7.8.2.1 SocketCanConnect()

```
static bool SocketCanConnect (
            void  )  [static]
```

Connects the CAN interface. Note that the channel and baudrate settings are ignored for the SocketCAN, because these are expected to be configured when the CAN device was brought online on the Linux system.

Returns

True if connected, false otherwise.

7.8.2.2 SocketCanEventThread()

```
static void * SocketCanEventThread (
            void * param )  [static]
```

Event thread that handles the asynchronous reception of data from the CAN interface.

Parameters

arg	Pointer to thread parameters.

Returns

Thread return value. Not used in this case, so always set to NULL.

Referenced by SocketCanStartEventThread().

Here is the caller graph for this function:

7.8.2.3 SocketCanGetInterface()

```
tCanInterface const * SocketCanGetInterface (
            void  )
```

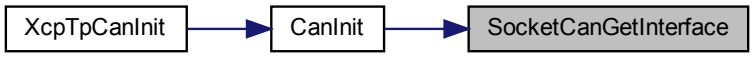
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.8.2.4 SocketCanInit()

```
static void SocketCanInit (
            tCanSettings const * settings )  [static]
```

Initializes the CAN interface. Note that this module assumes that the CAN device was already properly configured and brought online on the Linux system. Terminal command "ip addr" can be used to verify this.

Parameters

settings	Pointer to the CAN interface settings.

7.8.2.5 SocketCanIsBusError()

```
static bool SocketCanIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.8.2.6 SocketCanRegisterEvents()

```
static void SocketCanRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.8.2.7 SocketCanStartEventThread()

```
static bool SocketCanStartEventThread (
            void  )  [static]
```

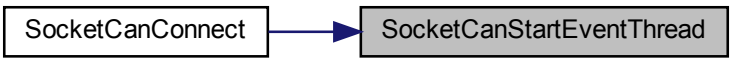
Starts the event thread.

Returns

True if the thread was successfully started, false otherwise.

Referenced by SocketCanConnect().

Here is the caller graph for this function:



7.8.2.8 SocketCanStopEventThread()

```
static void SocketCanStopEventThread (
            void  )  [static]
```

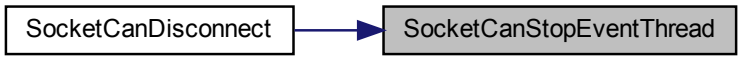
Stops the event thread. It sets the termination request and then waits for the termination handshake.

Returns

None.

Referenced by SocketCanDisconnect().

Here is the caller graph for this function:



7.8.2.9 SocketCanTransmit()

```
static bool SocketCanTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

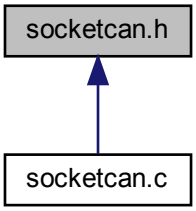
Returns

True if successful, false otherwise.

7.9 socketcan.h File Reference

Linux SocketCAN interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * SocketCanGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.9.1 Detailed Description

Linux SocketCAN interface header file.

7.9.2 Function Documentation

7.9.2.1 SocketCanGetInterface()

```
tCanInterface const * SocketCanGetInterface (
            void  )
```

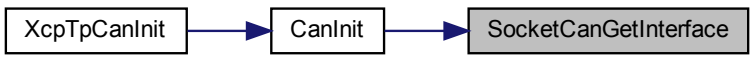
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:

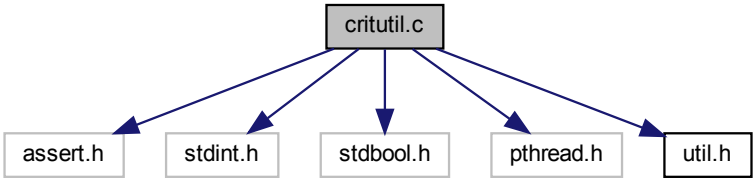


7.10 critutil.c File Reference

Critical section utility source file.

```
#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <pthread.h>
#include "util.h"
```

Include dependency graph for linux/critutil.c:



Functions

- void **UtilCriticalSectionInit** (void)

  Initializes the critical section module. Should be called before the Enter/Exit functions are used. It is okay to call this initialization multiple times from different modules.
- void **UtilCriticalSectionTerminate** (void)

  Terminates the critical section module. Should be called once critical sections are no longer needed. Typically called from another module's termination function that also initialized it. It is okay to call this termination multiple times from different modules.
- void **UtilCriticalSectionEnter** (void)

  Enters a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.
- void **UtilCriticalSectionExit** (void)

  Leaves a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.

Variables

- static volatile bool **criticalSectionInitialized** = false

  Flag to determine if the critical section object was already initialized.
- static volatile pthread_mutex_t **mtxCritSect**

  Critical section object.

7.10.1 Detailed Description

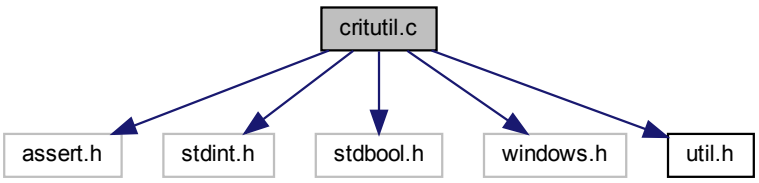
Critical section utility source file.

7.11 critutil.c File Reference

Critical section utility source file.

```
#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <windows.h>
#include "util.h"
```

Include dependency graph for windows/critutil.c:



Functions

- void **UtilCriticalSectionInit** (void)

  Initializes the critical section module. Should be called before the Enter/Exit functions are used. It is okay to call this initialization multiple times from different modules.
- void **UtilCriticalSectionTerminate** (void)

  Terminates the critical section module. Should be called once critical sections are no longer needed. Typically called from another module's termination function that also initialized it. It is okay to call this termination multiple times from different modules.
- void **UtilCriticalSectionEnter** (void)

  Enters a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.
- void **UtilCriticalSectionExit** (void)

  Leaves a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.

Variables

- static volatile bool **criticalSectionInitialized** = false

  Flag to determine if the critical section object was already initialized.
- static CRITICAL_SECTION **criticalSection**

  Critical section object.

7.11.1 Detailed Description

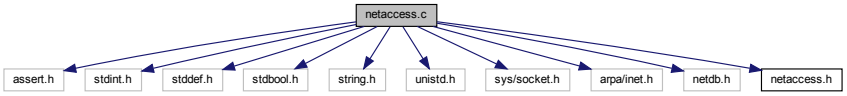
Critical section utility source file.

7.12 netaccess.c File Reference

TCP/IP network access source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
#include "netaccess.h"
```

Include dependency graph for linux/netaccess.c:



Macros

- #define **NETACCESS_INVALID_SOCKET** (-1)

  Constant value that indicates that the network socket is invalid.

Functions

- void **NetAccessInit** (void)

  Initializes the network access module.
- void **NetAccessTerminate** (void)

  Terminates the network access module.
- bool NetAccessConnect (char const ∗address, uint16_t port)

  Connects to the TCP/IP server at the specified address and the given port.
- void **NetAccessDisconnect** (void)

  Disconnects from the TCP/IP server.
- bool NetAccessSend (uint8_t const ∗data, uint32_t length)

  Sends data to the TCP/IP server.
- bool NetAccessReceive (uint8_t ∗data, uint32_t ∗length, uint32_t timeout)

  Receives data from the TCP/IP server in a blocking manner.

Variables

- static int **netAccessSocket**

  The socket that is used as an endpoint for the TCP/IP network communication.

7.12.1 Detailed Description

TCP/IP network access source file.

7.12.2 Function Documentation

7.12.2.1 NetAccessConnect()

```
bool NetAccessConnect (
            char const * address,
            uint16_t port )
```

Connects to the TCP/IP server at the specified address and the given port.

Parameters

address	The address of the server. This can be a hostname (such as mydomain.com) or an IP address (such as 127.0.0.1).
port	The port number on the server to connect to.

Returns

True if successful, false otherwise.

7.12.2.2 NetAccessReceive()

```
bool NetAccessReceive (
            uint8_t * data,
            uint32_t * length,
            uint32_t timeout )
```

Receives data from the TCP/IP server in a blocking manner.

Parameters

data	Pointer to byte array to store the received data.
length	Holds the max number of bytes that can be stored into the byte array. This function also overwrites this value with the number of bytes that were actually received.
timeout	Timeout in milliseconds for the data reception.

Returns

True if successful, false otherwise.

7.12.2.3 NetAccessSend()

```
bool NetAccessSend (
            uint8_t const * data,
            uint32_t length )
```

Sends data to the TCP/IP server.

Parameters

data	Pointer to byte array with data to send.
length	Number of bytes to send.

Returns

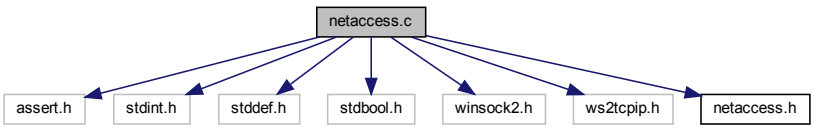
True if successful, false otherwise.

7.13 netaccess.c File Reference

TCP/IP network access source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include "netaccess.h"
```

Include dependency graph for windows/netaccess.c:



Functions

- void **NetAccessInit** (void)

  Initializes the network access module.
- void **NetAccessTerminate** (void)

Terminates the network access module.

- bool NetAccessConnect (char const ∗address, uint16_t port)

  Connects to the TCP/IP server at the specified address and the given port.
- void **NetAccessDisconnect** (void)

  Disconnects from the TCP/IP server.
- bool NetAccessSend (uint8_t const ∗data, uint32_t length)

  Sends data to the TCP/IP server.
- bool NetAccessReceive (uint8_t ∗data, uint32_t ∗length, uint32_t timeout)

  Receives data from the TCP/IP server in a blocking manner.

Variables

- static bool **winsockInitialized**

  Boolean flag to keep track if the Winsock library is initialized.
- static SOCKET **netAccessSocket**

  The socket that is used as an endpoint for the TCP/IP network communication.

7.13.1 Detailed Description

TCP/IP network access source file.

7.13.2 Function Documentation

7.13.2.1 NetAccessConnect()

```
bool NetAccessConnect (
            char const * address,
            uint16_t port )
```

Connects to the TCP/IP server at the specified address and the given port.

Parameters

address	The address of the server. This can be a hostname (such as mydomain.com) or an IP address (such as 127.0.0.1).
port	The port number on the server to connect to.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetConnect().

Here is the caller graph for this function:

7.13.2.2 NetAccessReceive()

```
bool NetAccessReceive (
            uint8_t * data,
            uint32_t * length,
            uint32_t timeout )
```

Receives data from the TCP/IP server in a blocking manner.

Parameters

data	Pointer to byte array to store the received data.
length	Holds the max number of bytes that can be stored into the byte array. This function also overwrites this value with the number of bytes that were actually received.
timeout	Timeout in milliseconds for the data reception.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetSendPacket().

Here is the caller graph for this function:

7.13.2.3 NetAccessSend()

```
bool NetAccessSend (
            uint8_t const * data,
            uint32_t length )
```

Sends data to the TCP/IP server.

Parameters

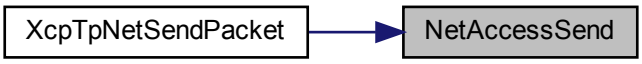
data	Pointer to byte array with data to send.
length	Number of bytes to send.

Returns

True if successful, false otherwise.

Referenced by XcpTpNetSendPacket().

Here is the caller graph for this function:

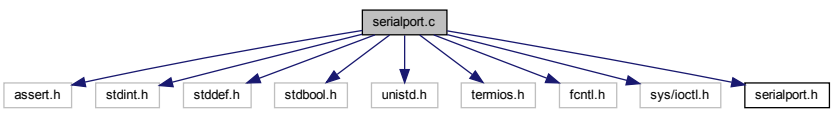


7.14 serialport.c File Reference

Serial port source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include "serialport.h"
```

Include dependency graph for linux/serialport.c:



Macros

- #define **SERIALPORT_INVALID_HANDLE** (-1)

  Invalid serial port device handle.

Functions

- void **SerialPortInit** (void)

  Initializes the serial port module.
- void **SerialPortTerminate** (void)

  Terminates the serial port module.
- bool SerialPortOpen (char const ∗portname, tSerialPortBaudrate baudrate, tSerialPortParity parity, tSerialPortStopbits stopbits)

  Opens the connection with the serial port.
- void **SerialPortClose** (void)

  Closes the connection with the serial port.
- bool SerialPortWrite (uint8_t const ∗data, uint32_t length)

  Writes data to the serial port.
- bool SerialPortRead (uint8_t ∗data, uint32_t length)

  Reads data from the serial port in a blocking manner.

Variables

- static int32_t **portHandle**

  Serial port handle.
- static const speed_t **baudrateLookup** [ ]

  Lookup table for converting this module's generic baudrate value to a value supported by the low level interface.

7.14.1 Detailed Description

Serial port source file.

7.14.2 Function Documentation

7.14.2.1 SerialPortOpen()

```
bool SerialPortOpen (
            char const * portname,
            tSerialPortBaudrate baudrate,
            tSerialPortParity parity,
            tSerialPortStopbits stopbits )
```

Opens the connection with the serial port.

Parameters

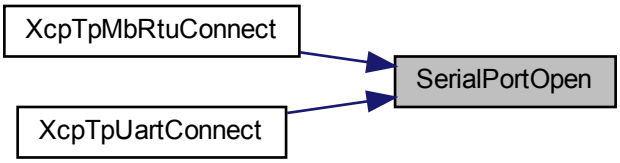
portname	The name of the serial port to open, i.e. /dev/ttyUSB0.
baudrate	The desired communication speed.
parity	The desired parity configuration.
stopbits	The desired stop bits configuration.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuConnect(), and XcpTpUartConnect().

Here is the caller graph for this function:



7.14.2.2 SerialPortRead()

```
bool SerialPortRead (
            uint8_t * data,
            uint32_t length )
```

Reads data from the serial port in a blocking manner.

Parameters

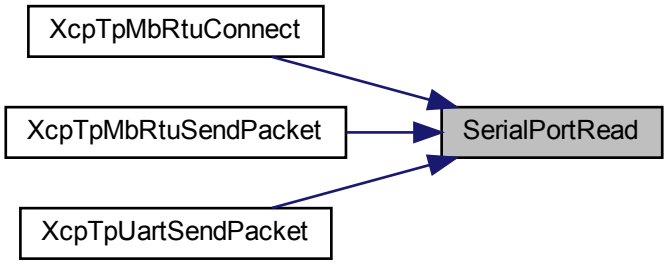
data	Pointer to byte array to store read data.
length	Number of bytes to read.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuSendPacket(), and XcpTpUartSendPacket().

Here is the caller graph for this function:



7.14.2.3 SerialPortWrite()

```
bool SerialPortWrite (
            uint8_t const * data,
            uint32_t length )
```

Writes data to the serial port.

Parameters

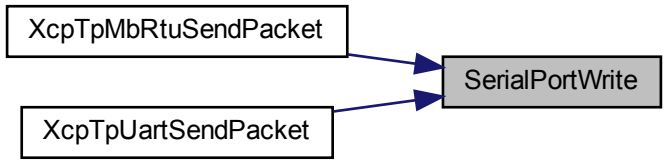
data	Pointer to byte array with data to write.
length	Number of bytes to write.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuSendPacket(), and XcpTpUartSendPacket().

Here is the caller graph for this function:

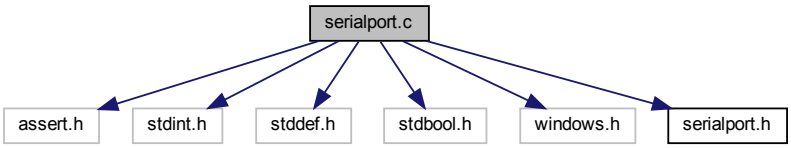


7.15 serialport.c File Reference

Serial port source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <windows.h>
#include "serialport.h"
```

Include dependency graph for windows/serialport.c:



Macros

- #define UART_TX_BUFFER_SIZE (1024u)
- #define UART_RX_BUFFER_SIZE (1024u)

Functions

- static uint32_t SerialConvertBaudrate (tSerialPortBaudrate baudrate)

  Opens the connection with the serial port configured as 8,N,1 and no flow control.
- void **SerialPortInit** (void)

  Initializes the serial port module.
- void **SerialPortTerminate** (void)

  Terminates the serial port module.
- bool SerialPortOpen (char const ∗portname, tSerialPortBaudrate baudrate, tSerialPortParity parity, tSerialPortStopbits stopbits)

  Opens the connection with the serial port.
- void **SerialPortClose** (void)

  Closes the connection with the serial port.
- bool SerialPortWrite (uint8_t const ∗data, uint32_t length)

  Writes data to the serial port.
- bool SerialPortRead (uint8_t ∗data, uint32_t length)

  Reads data from the serial port in a blocking manner.

Variables

- static HANDLE **hUart**

  Serial port handle.

7.15.1 Detailed Description

Serial port source file.

7.15.2 Macro Definition Documentation

7.15.2.1 UART_RX_BUFFER_SIZE

```
#define UART_RX_BUFFER_SIZE (1024u)
```

reception buffer size

7.15.2.2 UART_TX_BUFFER_SIZE

```
#define UART_TX_BUFFER_SIZE (1024u)
```

transmission buffer size

7.15.3 Function Documentation

7.15.3.1 SerialConvertBaudrate()

```
static uint32_t SerialConvertBaudrate (
            tSerialPortBaudrate baudrate )  [static]
```

Opens the connection with the serial port configured as 8,N,1 and no flow control.

Parameters

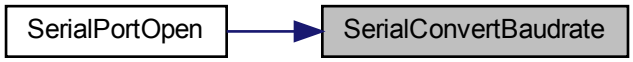
baudrate	The desired communication speed.

Returns

True if successful, false otherwise.

Referenced by SerialPortOpen().

Here is the caller graph for this function:



7.15.3.2 SerialPortOpen()

```
bool SerialPortOpen (
            char const * portname,
            tSerialPortBaudrate baudrate,
            tSerialPortParity parity,
            tSerialPortStopbits stopbits )
```

Opens the connection with the serial port.

Parameters

portname	The name of the serial port to open, i.e. COM4.
baudrate	The desired communication speed.
parity	The desired parity configuration.
stopbits	The desired stop bits configuration.

Returns

True if successful, false otherwise.

7.15.3.3 SerialPortRead()

```
bool SerialPortRead (
            uint8_t * data,
            uint32_t length )
```

Reads data from the serial port in a blocking manner.

Parameters

data	Pointer to byte array to store read data.
length	Number of bytes to read.

Returns

True if successful, false otherwise.

7.15.3.4 SerialPortWrite()

```
bool SerialPortWrite (
            uint8_t const * data,
            uint32_t length )
```

Writes data to the serial port.

Parameters

data	Pointer to byte array with data to write.
length	Number of bytes to write.

Returns

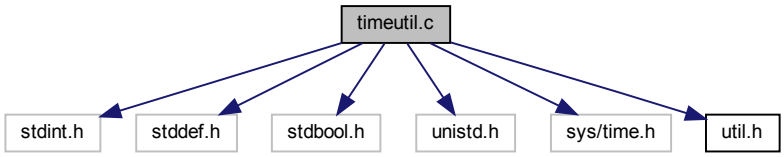
True if successful, false otherwise.

7.16 timeutil.c File Reference

Time utility source file.

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/time.h>
#include "util.h"
```

Include dependency graph for linux/timeutil.c:



Functions

- uint32_t UtilTimeGetSystemTimeMs (void)

  Get the system time in milliseconds.
- void UtilTimeDelayMs (uint16_t delay)

  Performs a delay of the specified amount of milliseconds.

7.16.1 Detailed Description

Time utility source file.

7.16.2 Function Documentation

7.16.2.1 UtilTimeDelayMs()

```
void UtilTimeDelayMs (
            uint16_t delay )
```

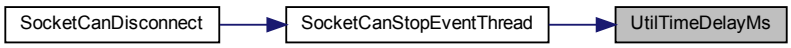
Performs a delay of the specified amount of milliseconds.

Parameters

delay	Delay time in milliseconds.

Referenced by SocketCanStopEventThread().

Here is the caller graph for this function:



7.16.2.2 UtilTimeGetSystemTimeMs()

```
uint32_t UtilTimeGetSystemTimeMs (
            void  )
```

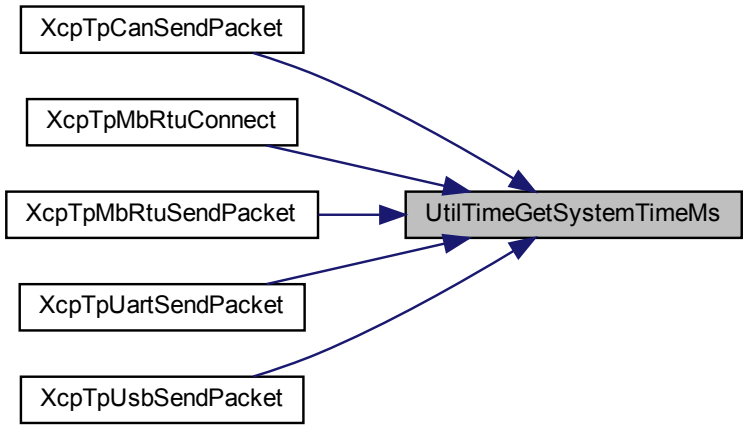
Get the system time in milliseconds.

Returns

Time in milliseconds.

Referenced by XcpTpCanSendPacket(), XcpTpMbRtuConnect(), XcpTpMbRtuSendPacket(), XcpTpUartSendPacket(), and XcpTpUsbSendPacket().

Here is the caller graph for this function:

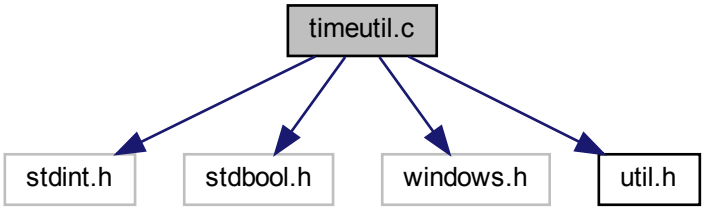


7.17 timeutil.c File Reference

Time utility source file.

```
#include <stdint.h>
#include <stdbool.h>
#include <windows.h>
#include "util.h"
```

Include dependency graph for windows/timeutil.c:



Functions

- uint32_t UtilTimeGetSystemTimeMs (void)

  Get the system time in milliseconds.
- void UtilTimeDelayMs (uint16_t delay)

  Performs a delay of the specified amount of milliseconds.

7.17.1 Detailed Description

Time utility source file.

7.17.2 Function Documentation

7.17.2.1 UtilTimeDelayMs()

```
void UtilTimeDelayMs (
            uint16_t delay )
```

Performs a delay of the specified amount of milliseconds.

Parameters

delay	Delay time in milliseconds.

Returns

none.

7.17.2.2 UtilTimeGetSystemTimeMs()

```
uint32_t UtilTimeGetSystemTimeMs (
            void  )
```

Get the system time in milliseconds.

Returns

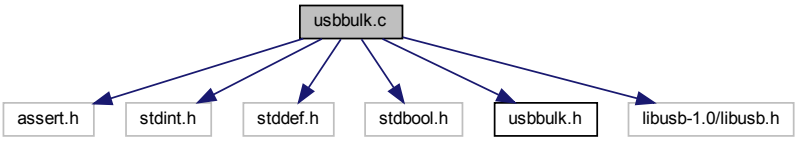
Time in milliseconds.

7.18 usbbulk.c File Reference

USB bulk driver source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "usbbulk.h"
#include <libusb-1.0/libusb.h>
```

Include dependency graph for linux/usbbulk.c:



Macros

- #define **USBBULK_READ_DATA_BUFFER_SIZE** (64u)

  Size of the internal endpoint read buffer. This should be the same as the size of the buffer size of the endpoint on the USB device itself.

Functions

- void **UsbBulkInit** (void)

  Initializes the USB bulk driver.
- void **UsbBulkTerminate** (void)

  Terminates the USB bulk driver.
- bool UsbBulkOpen (void)

  Opens the connection with the USB device.
- void **UsbBulkClose** (void)

  Closes the connection with the USB device.
- bool UsbBulkWrite (uint8_t const ∗data, uint16_t length)

  Writes data to the USB device.
- bool UsbBulkRead (uint8_t ∗data, uint16_t length, uint32_t timeout)

  Reads data from the USB device.

Variables

- static const uint16_t **openBltVendorId** = 0x1D50

  Vendor ID of the OpenBLT bootloader as assigned by the OpenMoko project.
- static const uint16_t **openBltProductId** = 0x60AC

  Product ID of the OpenBLT bootloader as assigned by the OpenMoko project.
- static libusb_context * **libUsbCtx**

  LibUsb context.
- static libusb_device_handle * **libUsbDevHandle**

  LibUsb device handle.
- static uint8_t **readDataBuffer** [USBBULK_READ_DATA_BUFFER_SIZE]

  Internal endpoint read buffer. With LibUsb endpoint read operations should always be attempted with the size of the endpoint buffer on the USB device itself.
- static uint8_t **readDataPending**

  Variable that holds the number of bytes that were read from the endpoint, but were not yet retrieved from this module via UsbBulkRead().
- static uint8_t **readDataCurrentReadIdx**

  Index into the endpoint read buffer (readDataBuffer[]) that point to the next byte value that should be read.

7.18.1 Detailed Description

USB bulk driver source file.

7.18.2 Function Documentation

7.18.2.1 UsbBulkOpen()

```
bool UsbBulkOpen (
            void  )
```

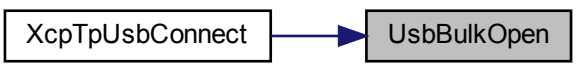
Opens the connection with the USB device.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbConnect().

Here is the caller graph for this function:



7.18.2.2 UsbBulkRead()

```
bool UsbBulkRead (
            uint8_t * data,
            uint16_t length,
            uint32_t timeout )
```

Reads data from the USB device.

Parameters

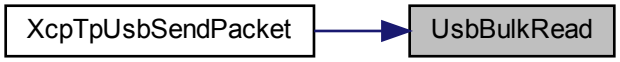
data	Pointer to byte array where received data should be stored.
length	Number of bytes to read from the USB device.
timeout	Timeout in milliseconds for the read operation.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbSendPacket().

Here is the caller graph for this function:



7.18.2.3 UsbBulkWrite()

```
bool UsbBulkWrite (
            uint8_t const * data,
            uint16_t length )
```

Writes data to the USB device.

Parameters

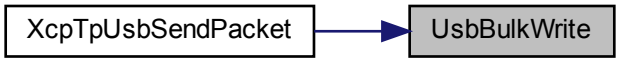
data	Pointer to byte array with data to write.
length	Number of bytes in the data array.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbSendPacket().

Here is the caller graph for this function:

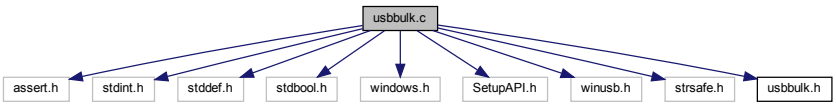


7.19 usbbulk.c File Reference

USB bulk driver source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <windows.h>
#include <SetupAPI.h>
#include <winusb.h>
#include <strsafe.h>
#include "usbbulk.h"
```

Include dependency graph for windows/usbbulk.c:



Data Structures

- struct tBulkUsbDev

  Type for grouping together all USB bulk device related data.

Macros

- #define **MAX_DEVPATH_LENGTH** (128)

  Max length of the device path.
- #define **INVALID_PIPE_ID** (255)

  Identifier value of an invalid USB PIPE.

Functions

- static uint8_t UblOpen (LPCGUID guid)

  Opens and configures the connection with the USB bulk device.
- static void **UblClose** (void)

  Closes the connection with the USB bulk device and frees all the related handles.
- static uint8_t UblTransmit (uint8_t ∗data, uint16_t len)

  Starts transmission of the data on the bulk OUT pipe. Because USB bulk transmissions are quick, this function does not use the overlapped functionality, which means the caller is blocked until the tranmission completed.
- static uint8_t UblReceive (uint8_t ∗data, uint16_t len, uint32_t timeout)

  Starts the asynchronous reception of the data from the bulk IN pipe. This function makes use of the overlapped functionality, which means the calling thread is placed into sleep mode until the reception is complete.
- void **UsbBulkInit** (void)

  Initializes the USB bulk driver.
- void **UsbBulkTerminate** (void)

  Terminates the USB bulk driver.
- bool UsbBulkOpen (void)

  Opens the connection with the USB device.
- void **UsbBulkClose** (void)

  Closes the connection with the USB device.
- bool UsbBulkWrite (uint8_t const ∗data, uint16_t length)

  Writes data to the USB device.
- bool UsbBulkRead (uint8_t ∗data, uint16_t length, uint32_t timeout)

  Reads data from the USB device.
- static HANDLE UblOpenDevice (LPCGUID InterfaceGuid)

  Opens the USB device and obtains its handle, which is needed to obtain a handle to the WinUSB device.
- static BOOL UblGetDevicePath (LPCGUID InterfaceGuid, PCHAR DevicePath, size_t BufLen)

  Attempts to obtain the path to the WinUSB device, based on its GUID.

7.19.1 Detailed Description

USB bulk driver source file.

7.19.2 Function Documentation

7.19.2.1 UblGetDevicePath()

```
static BOOL UblGetDevicePath (
            LPCGUID InterfaceGuid,
            PCHAR DevicePath,
            size_t BufLen )  [static]
```

Attempts to obtain the path to the WinUSB device, based on its GUID.

Parameters

InterfaceGuid	InterfaceGuid GUID of the device (not its class though).
DevicePath	Pointer to where the path should be stored.
BufLen	Maximum length of the path.

Returns

TRUE if the device path was obtained, FALSE otherwise.

Referenced by UblOpenDevice().

Here is the caller graph for this function:

7.19.2.2 UblOpen()

```
static uint8_t UblOpen (
            LPCGUID guid )  [static]
```

Opens and configures the connection with the USB bulk device.

Parameters

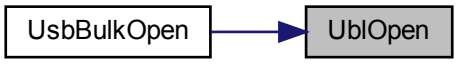
guid	Pointer to GUID of the USB bulk device as found in the driver's INF-file.

Returns

UBL_OKAY if successful, UBL_ERROR otherwise.

Referenced by UsbBulkOpen().

Here is the caller graph for this function:



7.19.2.3 UblOpenDevice()

```
static HANDLE UblOpenDevice (
            LPCGUID InterfaceGuid )  [static]
```

Opens the USB device and obtains its handle, which is needed to obtain a handle to the WinUSB device.

Parameters

InterfaceGuid	InterfaceGuid GUID of the device (not its class though).

Returns

The handle to the USB device or NULL.

7.19.2.4 UblReceive()

```
static uint8_t UblReceive (
            uint8_t * data,
            uint16_t len,
            uint32_t timeout )  [static]
```

Starts the asynchronous reception of the data from the bulk IN pipe. This function makes use of the overlapped functionality, which means the calling thread is placed into sleep mode until the reception is complete.

Parameters

data	Pointer to byte array where the data will be stored.
len	Number of bytes to receive.
timeout	Max time in milliseconds for the read to complete.

Returns

UBL_OKAY if all bytes were received, UBL_TIMEOUT if a timeout occured, UBL_ERROR otherwise.

Referenced by UsbBulkRead().

Here is the caller graph for this function:

7.19.2.5 UblTransmit()

```
static uint8_t UblTransmit (
              uint8_t * data,
              uint16_t len )  [static]
```

Starts transmission of the data on the bulk OUT pipe. Because USB bulk transmissions are quick, this function does not use the overlapped functionality, which means the caller is blocked until the tranmission completed.

Parameters

data	Pointer to byte array with transmit data.
len	Number of bytes to transmit.

Returns

UBL_OKAY if successful, UBL_ERROR otherwise.

Referenced by UsbBulkWrite().

Here is the caller graph for this function:

7.19.2.6 UsbBulkOpen()

```
bool UsbBulkOpen (
            void  )
```

Opens the connection with the USB device.

Returns

True if successful, false otherwise.

7.19.2.7 UsbBulkRead()

```
bool UsbBulkRead (
            uint8_t * data,
            uint16_t length,
            uint32_t timeout )
```

Reads data from the USB device.

Parameters

data	Pointer to byte array where received data should be stored.
length	Number of bytes to read from the USB device.
timeout	Timeout in milliseconds for the read operation.

Returns

True if successful, false otherwise.

7.19.2.8 UsbBulkWrite()

```
bool UsbBulkWrite (
            uint8_t const * data,
            uint16_t length )
```

Writes data to the USB device.

Parameters

data	Pointer to byte array with data to write.
length	Number of bytes in the data array.

Returns

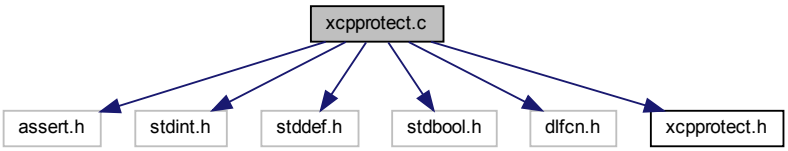
True if successful, false otherwise.

7.20 xcpprotect.c File Reference

XCP Protection module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <dlfcn.h>
#include "xcpprotect.h"
```

Include dependency graph for linux/xcpprotect.c:



Functions

- void XcpProtectInit (char const ∗seedKeyFile)

  Initializes the XCP protection module.
- void **XcpProtectTerminate** (void)

  Terminates the XCP protection module.
- bool XCPProtectComputeKeyFromSeed (uint8_t resource, uint8_t seedLen, uint8_t const ∗seedPtr, uint8_t ∗keyLenPtr, uint8_t ∗keyPtr)

  Computes the key for the requested resource.
- bool XcpProtectGetPrivileges (uint8_t ∗resourcePtr)

  Obtains a bitmask of the resources for which an key algorithm is available.

Variables

- static void ∗ **seedNKeyLibraryHandle**

  Handle to the dynamically loaded seed and key shared library. It can also be used as a flag to determine if the shared library was specified and success- fully loaded.
- static tXcpProtectLibComputeKey **xcpProtectLibComputeKey**

  Function pointer to the XCP_ComputeKeyFromSeed shared library function.
- static tXcpProtectLibGetPrivileges **xcpProtectLibGetPrivileges**

  Function pointer to the XCP_GetAvailablePrivileges shared library function.

7.20.1 Detailed Description

XCP Protection module source file.

7.20.2 Function Documentation

7.20.2.1 XCPProtectComputeKeyFromSeed()

```
bool XCPProtectComputeKeyFromSeed (
            uint8_t resource,
            uint8_t seedLen,
            uint8_t const * seedPtr,
            uint8_t * keyLenPtr,
            uint8_t * keyPtr )
```

Computes the key for the requested resource.

Parameters

resource	resource for which the unlock key is requested
seedLen	length of the seed
seedPtr	pointer to the seed data
keyLenPtr	pointer where to store the key length
keyPtr	pointer where to store the key data

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:

7.20.2.2 XcpProtectGetPrivileges()

```
bool XcpProtectGetPrivileges (
            uint8_t * resourcePtr )
```

Obtains a bitmask of the resources for which an key algorithm is available.

Parameters

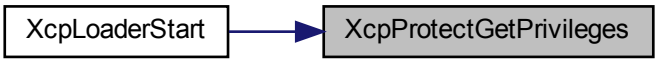
resourcePtr	pointer where to store the supported resources for the key computation.

Returns

XCP_RESULT_OK on success, otherwise XCP_RESULT_ERROR.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:



7.20.2.3 XcpProtectInit()

```
void XcpProtectInit (
            char const * seedKeyFile )
```

Initializes the XCP protection module.

Parameters

seedKeyFile	Filename of the seed and key shared library that contains the following functions: • XCP_ComputeKeyFromSeed() • XCP_GetAvailablePrivileges()

Referenced by XcpLoaderInit().

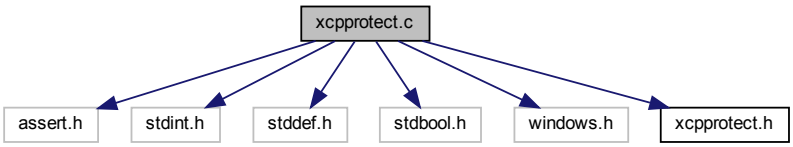
Here is the caller graph for this function:

7.21 xcpprotect.c File Reference

XCP Protection module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <windows.h>
#include "xcpprotect.h"
```

Include dependency graph for windows/xcpprotect.c:



Functions

- void XcpProtectInit (char const ∗seedKeyFile)

  Initializes the XCP protection module.
- void **XcpProtectTerminate** (void)

  Terminates the XCP protection module.
- bool XCPProtectComputeKeyFromSeed (uint8_t resource, uint8_t seedLen, uint8_t const ∗seedPtr, uint8_t ∗keyLenPtr, uint8_t ∗keyPtr)

  Computes the key for the requested resource.
- bool XcpProtectGetPrivileges (uint8_t ∗resourcePtr)

  Obtains a bitmask of the resources for which an key algorithm is available.

Variables

- static HINSTANCE **seedNKeyLibraryHandle**

  Handle to the dynamically loaded seed and key shared library. It can also be used as a flag to determine if the shared library was specified and success- fully loaded.
- static tXcpProtectLibComputeKey **xcpProtectLibComputeKey**

  Function pointer to the XCP_ComputeKeyFromSeed shared library function.
- static tXcpProtectLibGetPrivileges **xcpProtectLibGetPrivileges**

  Function pointer to the XCP_GetAvailablePrivileges shared library function.

7.21.1 Detailed Description

XCP Protection module source file.

7.21.2 Function Documentation

7.21.2.1 XCPProtectComputeKeyFromSeed()

```
bool XCPProtectComputeKeyFromSeed (
            uint8_t resource,
            uint8_t seedLen,
            uint8_t const * seedPtr,
            uint8_t * keyLenPtr,
            uint8_t * keyPtr )
```

Computes the key for the requested resource.

Parameters

resource	resource for which the unlock key is requested
seedLen	length of the seed
seedPtr	pointer to the seed data
keyLenPtr	pointer where to store the key length
keyPtr	pointer where to store the key data

Returns

True if successful, false otherwise.

7.21.2.2 XcpProtectGetPrivileges()

```
bool XcpProtectGetPrivileges (
            uint8_t * resourcePtr )
```

Obtains a bitmask of the resources for which an key algorithm is available.

Parameters

resourcePtr	pointer where to store the supported resources for the key computation.

Returns

XCP_RESULT_OK on success, otherwise XCP_RESULT_ERROR.

7.21.2.3 XcpProtectInit()

```
void XcpProtectInit (
            char const * seedKeyFile )
```

Initializes the XCP protection module.

Parameters

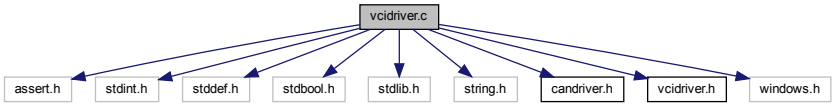
seedKeyFile	Filename of the seed and key shared library that contains the following functions: • XCP_ComputeKeyFromSeed() • XCP_GetAvailablePrivileges()

7.22 vcidriver.c File Reference

Ixxat VCI driver interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "candriver.h"
#include "vcidriver.h"
#include <windows.h>
#include "vcinpl.h"
```

Include dependency graph for vcidriver.c:



Functions

- static void IxxatVciInit (tCanSettings const *settings)

  Initializes the CAN interface.
- static void **IxxatVciTerminate** (void)

Terminates the CAN interface.

- static bool IxxatVciConnect (void)

  Connects the CAN interface.
- static void **IxxatVciDisconnect** (void)

  Disconnects the CAN interface.
- static bool IxxatVciTransmit (tCanMsg const *msg)

  Submits a message for transmission on the CAN bus.
- static bool IxxatVciIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void IxxatVciRegisterEvents (tCanEvents const *events)

  Registers the event callback functions that should be called by the CAN interface.
- static DWORD WINAPI IxxatVciReceptionThread (LPVOID pv)

  CAN message reception thread.
- static bool IxxatVciConvertBaudrate (UINT8 *bBtr0, UINT8 *bBtr1)

  Converts the baudrate enum value to the Ixxat API's BTR0 and BTR1 values.
- static void **IxxatVciLibLoadDll** (void)

  Loads the Ixxat VCI DLL and initializes the API function pointers.
- static void **IxxatVciLibUnloadDll** (void)

  Unloads the Ixxat VCI DLL and resets the API function pointers.
- static HRESULT IxxatVciLibFuncVciEnumDeviceOpen (PHANDLE hEnum)

  Opens the list of all fieldbus adapters registered with the VCI.
- static HRESULT IxxatVciLibFuncVciEnumDeviceClose (HANDLE hEnum)

  Closes the device list opened with the function vciEnumDeviceOpen.
- static HRESULT IxxatVciLibFuncVciEnumDeviceNext (HANDLE hEnum, PVCIDEVICEINFO pInfo)

  Determines the description of a fieldbus adapter of the device list and increases the internal list index so that a subsequent call of the function supplies the description to the next adapter.
- static HRESULT IxxatVciLibFuncVciDeviceOpen (REFVCIID rVciid, PHANDLE phDevice)

  Opens the fieldbus adapter with the specified device ID.
- static HRESULT IxxatVciLibFuncVciDeviceClose (HANDLE hDevice)

  Closes an opened fieldbus adapter.
- static HRESULT IxxatVciLibFuncCanControlOpen (HANDLE hDevice, UINT32 dwCanNo, PHANDLE phCanCtl)

  Opens the control unit of a CAN connection on a fieldbus adapter.
- static HRESULT IxxatVciLibFuncCanControlReset (HANDLE hCanCtl)

  Resets the controller hardware and resets the message filters of a CAN connection.
- static HRESULT IxxatVciLibFuncCanControlGetCaps (HANDLE hCanCtl, PCANCAPABILITIES pCanCaps)

  Determines the features of a CAN connection.
- static HRESULT IxxatVciLibFuncCanControlInitialize (HANDLE hCanCtl, UINT8 bMode, UINT8 bBtr0, UINT8 bBtr1)

  Sets the operating mode and bit rate of a CAN connection.
- static HRESULT IxxatVciLibFuncCanControlSetAccFilter (HANDLE hCanCtl, BOOL fExtend, UINT32 dwCode, UINT32 dwMask)

  Sets the 11- or 29-bit acceptance filter of a CAN connection.
- static HRESULT IxxatVciLibFuncCanControlStart (HANDLE hCanCtl, BOOL fStart)

  Starts or stops the controller of a CAN connection.
- static HRESULT IxxatVciLibFuncCanControlClose (HANDLE hCanCtl)

  Closes an opened CAN controller.
- static HRESULT IxxatVciLibFuncCanChannelOpen (HANDLE hDevice, UINT32 dwCanNo, BOOL fExclusive, PHANDLE phCanChn)

  Opens or creates a message channel for a CAN connection of a fieldbus adapter.
- static HRESULT IxxatVciLibFuncCanChannelGetStatus (HANDLE hCanChn, PCANCHANSTATUS pStatus)

Determines the current status of a message channel as well as the current settings and the current status of the controller that is connected to the channel.

- static HRESULT IxxatVciLibFuncCanChannelInitialize (HANDLE hCanChn, UINT16 wRxFifoSize, UINT16 wRxThreshold, UINT16 wTxFifoSize, UINT16 wTxThreshold)

  Initializes the receive and transmit buffers of a message channel.
- static HRESULT IxxatVciLibFuncCanChannelActivate (HANDLE hCanChn, BOOL fEnable)

  Activates or deactivates a message channel.
- static HRESULT IxxatVciLibFuncCanChannelPostMessage (HANDLE hCanChn, PCANMSG pCanMsg)

  Writes a CAN message in the transmit buffer of the specified message channel.
- static HRESULT IxxatVciLibFuncCanChannelReadMessage (HANDLE hCanChn, UINT32 dwTimeout, PCANMSG pCanMsg)

  Retrieves the next CAN message from the receive FIFO of the specified CAN channel. The function waits for a message to be received from the CAN bus.
- static HRESULT IxxatVciLibFuncCanChannelClose (HANDLE hCanChn)

  Closes an opened message channel.
- tCanInterface const * IxxatVciGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **ixxatVciInterface**

  CAN interface structure filled with Ixxat VCI driver specifics.
- static tCanSettings **ixxatVciSettings**

  The settings to use in this CAN interface.
- static tCanEvents * **ixxatVciEventsList**

  List with callback functions that this driver should use.
- static uint32_t **ixxatVciEventsEntries**

  Total number of event entries into the pCanUsbEventsList list.
- static HINSTANCE **ixxatVciDllHandle**

  Handle to the Ixxat VCI dynamic link library.
- static HANDLE **ixxatVciTerminateEvent**

  Handle for the event to terminate the reception thread.
- static HANDLE **ixxatVciRxThreadHandle**

  Handle for the CAN reception thread.
- static HANDLE **ixxatVciDeviceHandle**

  Handle for the Ixxat device.
- static HANDLE **ixxatCanControlHandle**

  Handle for the Ixxat control.
- static HANDLE **ixxatCanChannelHandle**

  Handle for the Ixxat channel.
- static PF_vciEnumDeviceOpen **ixxatVciLibFuncVciEnumDeviceOpenPtr**

  Function pointer to the Ixxat vciEnumDeviceOpen function.
- static PF_vciEnumDeviceClose **ixxatVciLibFuncVciEnumDeviceClosePtr**

  Function pointer to the Ixxat vciEnumDeviceClose function.
- static PF_vciEnumDeviceNext **ixxatVciLibFuncVciEnumDeviceNextPtr**

  Function pointer to the Ixxat vciEnumDeviceNext function.
- static PF_vciDeviceOpen **ixxatVciLibFuncVciDeviceOpenPtr**

  Function pointer to the Ixxat vciDeviceOpen function.
- static PF_vciDeviceClose **ixxatVciLibFuncVciDeviceClosePtr**

  Function pointer to the Ixxat vciDeviceClose function.
- static PF_canControlOpen **ixxatVciLibFuncCanControlOpenPtr**

Function pointer to the Ixxat canControlOpen function.

- static PF_canControlReset **ixxatVciLibFuncCanControlResetPtr**

  Function pointer to the Ixxat canControlReset function.
- static PF_canControlGetCaps **ixxatVciLibFuncCanControlGetCapsPtr**

  Function pointer to the Ixxat canControlGetCaps function.
- static PF_canControlInitialize **ixxatVciLibFuncCanControlInitializePtr**

  Function pointer to the Ixxat canControlInitialize function.
- static PF_canControlSetAccFilter **ixxatVciLibFuncCanControlSetAccFilterPtr**

  Function pointer to the Ixxat canControlSetAccFilter function.
- static PF_canControlStart **ixxatVciLibFuncCanControlStartPtr**

  Function pointer to the Ixxat canControlStart function.
- static PF_canControlClose **ixxatVciLibFuncCanControlClosePtr**

  Function pointer to the Ixxat canControlClose function.
- static PF_canChannelOpen **ixxatVciLibFuncCanChannelOpenPtr**

  Function pointer to the Ixxat canChannelOpen function.
- static PF_canChannelGetStatus **ixxatVciLibFuncCanChannelGetStatusPtr**

  Function pointer to the Ixxat canChannelGetStatus function.
- static PF_canChannelInitialize **ixxatVciLibFuncCanChannelInitializePtr**

  Function pointer to the Ixxat canChannelInitialize function.
- static PF_canChannelActivate **ixxatVciLibFuncCanChannelActivatePtr**

  Function pointer to the Ixxat canChannelActivate function.
- static PF_canChannelPostMessage **ixxatVciLibFuncCanChannelPostMessagePtr**

  Function pointer to the Ixxat canChannelPostMessage function.
- static PF_canChannelReadMessage **ixxatVciLibFuncCanChannelReadMessagePtr**

  Function pointer to the Ixxat canChannelReadMessage function.
- static PF_canChannelClose **ixxatVciLibFuncCanChannelClosePtr**

  Function pointer to the Ixxat canChannelClose function.

7.22.1 Detailed Description

Ixxat VCI driver interface source file.

7.22.2 Function Documentation

7.22.2.1 IxxatVciConnect()

```
static bool IxxatVciConnect (
            void  )  [static]
```

Connects the CAN interface.

Returns

True if connected, false otherwise.

7.22.2.2 IxxatVciConvertBaudrate()

```
static bool IxxatVciConvertBaudrate (
            UINT8 * bBtr0,
            UINT8 * bBtr1 )  [static]
```

Converts the baudrate enum value to the Ixxat API's BTR0 and BTR1 values.

Parameters

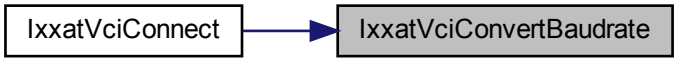
bBtr0	Storage for the BTR0 setting.
bBtr1	Storage for the BTR1 setting.

Returns

True if the baudrate could be converted, false otherwise.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.3 IxxatVciGetInterface()

```
tCanInterface const * IxxatVciGetInterface (
            void  )
```

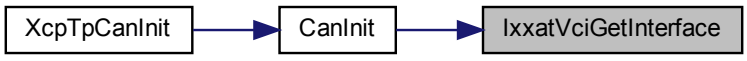
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.22.2.4 IxxatVciInit()

```
static void IxxatVciInit (
            tCanSettings const * settings )  [static]
```

Initializes the CAN interface.

Parameters

settings	Pointer to the CAN interface settings.

7.22.2.5 IxxatVciIsBusError()

```
static bool IxxatVciIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.22.2.6 IxxatVciLibFuncCanChannelActivate()

```
static HRESULT IxxatVciLibFuncCanChannelActivate (
            HANDLE hCanChn,
            BOOL fEnable )  [static]
```

Activates or deactivates a message channel.

Parameters

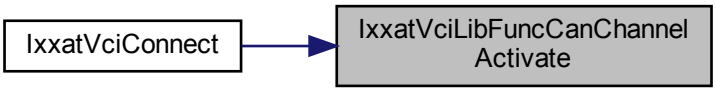
hCanChn	Handle of the opened message channel.
fEnable	With the value TRUE, the function activates the message flow between the CAN controller and the message channel, with the value FALSE the function deactivates the message flow.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.7 IxxatVciLibFuncCanChannelClose()

```
static HRESULT IxxatVciLibFuncCanChannelClose (
            HANDLE hCanChn )  [static]
```

Closes an opened message channel.

Parameters

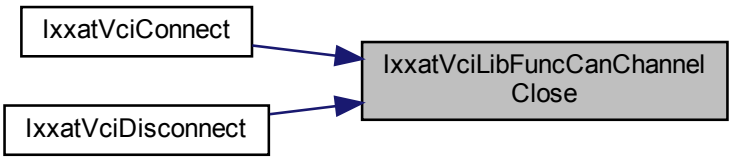
hCanChn	Handle of the message channel to be closed. The specified handle must come from a call of the function canChannelOpen.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect(), and IxxatVciDisconnect().

Here is the caller graph for this function:



7.22.2.8 IxxatVciLibFuncCanChannelGetStatus()

```
static HRESULT IxxatVciLibFuncCanChannelGetStatus (
            HANDLE hCanChn,
            PCANCHANSTATUS pStatus )  [static]
```

Determines the current status of a message channel as well as the current settings and the current status of the controller that is connected to the channel.

Parameters

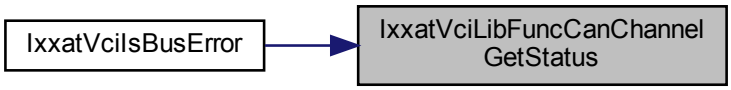
hCanChn	Handle of the opened message channel.
pStatus	Pointer to a structure of type CANCHANSTATUS. If run successfully, the function saves the current status of the channel and controller in the memory area specified here.

Returns

VCI_OK if successful.

Referenced by IxxatVciIsBusError().

Here is the caller graph for this function:



7.22.2.9 IxxatVciLibFuncCanChannelInitialize()

```
static HRESULT IxxatVciLibFuncCanChannelInitialize (
            HANDLE hCanChn,
            UINT16 wRxFifoSize,
            UINT16 wRxThreshold,
            UINT16 wTxFifoSize,
            UINT16 wTxThreshold )  [static]
```

Initializes the receive and transmit buffers of a message channel.

Parameters

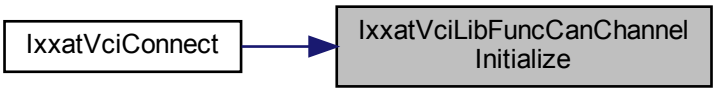
hCanChn	Handle of the opened message channel.
wRxFifoSize	Size of the receive buffer in number of CAN messages.
wRxThreshold	Threshold value for the receive event. The event is triggered when the number of messages in the receive buffer reaches or exceeds the number specified here.
wTxFifoSize	Size of the transmit buffer in number of CAN messages.
wTxThreshold	Threshold value for the transmit event. The event is triggered when the number of free entries in the transmit buffer reaches or exceeds the number specified here.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.10 IxxatVciLibFuncCanChannelOpen()

```
static HRESULT IxxatVciLibFuncCanChannelOpen (
            HANDLE hDevice,
            UINT32 dwCanNo,
            BOOL fExclusive,
            PHANDLE phCanChn )  [static]
```

Opens or creates a message channel for a CAN connection of a fieldbus adapter.

Parameters

hDevice	Handle of the fieldbus adapter.
dwCanNo	Number of the CAN connection for which a message channel is to be opened. The value 0 selects the first connection, the value 1 the second connection and so on.
fExclusive	Defines whether the connection is used exclusively for the channel to be opened. If the value TRUE is specified here, the CAN connection is used exclusively for the new message channel. With the value FALSE, more than one message channel can be opened for the CAN connection.
phCanChn	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN message channel in this variable. In the event of an of an error, the variable is set to ZERO.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:

7.22.2.11 IxxatVciLibFuncCanChannelPostMessage()

```
static HRESULT IxxatVciLibFuncCanChannelPostMessage (
            HANDLE hCanChn,
            PCANMSG pCanMsg )  [static]
```

Writes a CAN message in the transmit buffer of the specified message channel.

Parameters

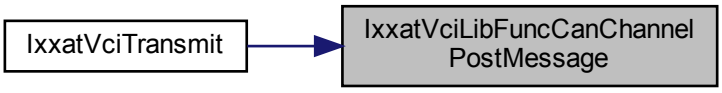
hCanChn	Handle of the opened message channel.
pCanMsg	Pointer to an initialized structure of type CANMSG with the CAN message to be transmitted.

Returns

VCI_OK if successful.

Referenced by IxxatVciTransmit().

Here is the caller graph for this function:



7.22.2.12 IxxatVciLibFuncCanChannelReadMessage()

```
static HRESULT IxxatVciLibFuncCanChannelReadMessage (
            HANDLE hCanChn,
            UINT32 dwTimeout,
            PCANMSG pCanMsg )  [static]
```

Retrieves the next CAN message from the receive FIFO of the specified CAN channel. The function waits for a message to be received from the CAN bus.

Parameters

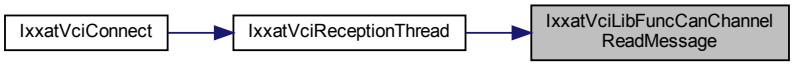
hCanChn	Handle of the opened message channel.
dwTimeout	Maximum waiting time in milliseconds. The function returns to the caller with the error code VCI_E_TIMEOUT if no message is read or received within the specified time. With the value INFINITE (0xFFFFFFFF), the function waits until a message has been read.
pCanMsg	Pointer to a CANMSG structure where the function stores the retrieved CAN message. If this parameter is set to NULL, the function simply removes the next CAN message from the FIFO.

Returns

VCI_OK if successful, VCI_E_RXQUEUE_EMPTY if there currently is no CAN message available, VCI_E↩ _TIMEOUT if the time-out interval elapsed without a new CAN message available. Another value indicates a generic error.

Referenced by IxxatVciReceptionThread().

Here is the caller graph for this function:



7.22.2.13 IxxatVciLibFuncCanControlClose()

```
static HRESULT IxxatVciLibFuncCanControlClose (
            HANDLE hCanCtl )  [static]
```

Closes an opened CAN controller.

Parameters

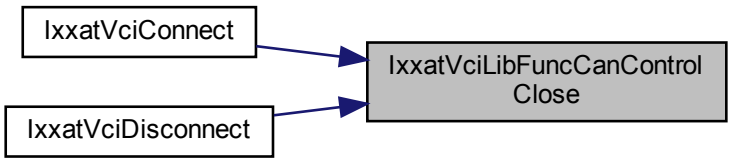
hCanCtl	Handle of the CAN controller to be closed. The specified handle must come from a call of the function canControlOpen.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect(), and IxxatVciDisconnect().

Here is the caller graph for this function:



7.22.2.14 IxxatVciLibFuncCanControlGetCaps()

```
static HRESULT IxxatVciLibFuncCanControlGetCaps (
            HANDLE hCanCtl,
            PCANCAPABILITIES pCanCaps )  [static]
```

Determines the features of a CAN connection.

Parameters

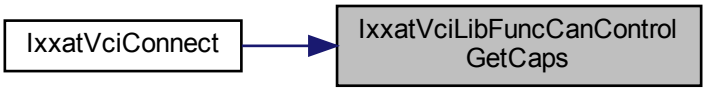
hCanCtl	Handle of the opened CAN controller.
pCanCaps	Pointer to a structure of type CANCAPABILITIES. If run successfully, the function saves the features of the CAN connection in the memory area specified here.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.15 IxxatVciLibFuncCanControlInitialize()

```
static HRESULT IxxatVciLibFuncCanControlInitialize (
            HANDLE hCanCtl,
            UINT8 bMode,
            UINT8 bBtr0,
            UINT8 bBtr1 )  [static]
```

Sets the operating mode and bit rate of a CAN connection.

Parameters

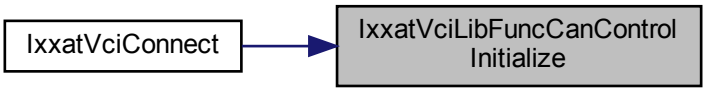
hCanCtl	Handle of the opened CAN controller.
bMode	Operating mode of the CAN controller.
bBtr0	Value for the bus timing register 0 of the CAN controller. The value corresponds to the BTR0 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.
bBtr1	Value for the bus timing register 1 of the CAN controller. The value corresponds to the BTR1 register of the Philips SJA 1000 CAN controller with a cycle frequency of 16 MHz.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.16 IxxatVciLibFuncCanControlOpen()

```
static HRESULT IxxatVciLibFuncCanControlOpen (
            HANDLE hDevice,
            UINT32 dwCanNo,
            PHANDLE phCanCtl )  [static]
```

Opens the control unit of a CAN connection on a fieldbus adapter.

Parameters

hDevice	Handle of the fieldbus adapter.
dwCanNo	Number of the CAN connection of the control unit to be opened. The value 0 selects the first connection, the value 1 the second connection and so on.
phCanCtl	Pointer to a variable of type HANDLE. If run successfully, the function returns the handle of the opened CAN controller in this variable. In the event of an error, the variable is set to ZERO.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:

7.22.2.17 IxxatVciLibFuncCanControlReset()

```
static HRESULT IxxatVciLibFuncCanControlReset (
            HANDLE hCanCtl )  [static]
```

Resets the controller hardware and resets the message filters of a CAN connection.

Parameters

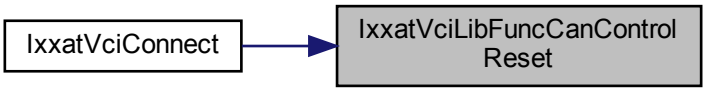
hCanCtl	Handle of the opened CAN controller.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.18 IxxatVciLibFuncCanControlSetAccFilter()

```
static HRESULT IxxatVciLibFuncCanControlSetAccFilter (
            HANDLE hCanCtl,
            BOOL fExtend,
            UINT32 dwCode,
            UINT32 dwMask )  [static]
```

Sets the 11- or 29-bit acceptance filter of a CAN connection.

Parameters

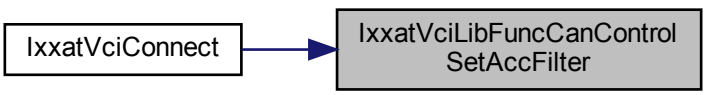
hCanCtl	Handle of the opened CAN controller.
fExtend	Selection of the acceptance filter. The 11-bit acceptance filter is selected with the value FALSE and the 29-bit acceptance filter with the value TRUE.
dwCode	Bit sample of the identifier(s) to be accepted including RTR-bit.
dwMask	Bit sample of the relevant bits in dwCode. If a bit has the value 0 in dwMask, the corresponding bit in dwCode is not used for the comparison. If a bit has the value 1, it is relevant for the comparison.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.19 IxxatVciLibFuncCanControlStart()

```
static HRESULT IxxatVciLibFuncCanControlStart (
            HANDLE hCanCtl,
            BOOL fStart )  [static]
```

Starts or stops the controller of a CAN connection.

Parameters

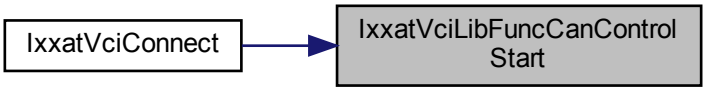
hCanCtl	Handle of the opened CAN controller.
fStart	The value TRUE starts and the value FALSE stops the CAN controller.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.20 IxxatVciLibFuncVciDeviceClose()

```
static HRESULT IxxatVciLibFuncVciDeviceClose (
            HANDLE hDevice )  [static]
```

Closes an opened fieldbus adapter.

Parameters

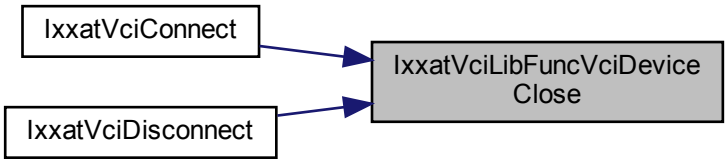
hDevice	Handle of the adapter to be closed. The specified handle must come from a call of one of the functions vciEnumDeviceOpen or vciDeviceOpenDlg.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect(), and IxxatVciDisconnect().

Here is the caller graph for this function:



7.22.2.21 IxxatVciLibFuncVciDeviceOpen()

```
static HRESULT IxxatVciLibFuncVciDeviceOpen (
            REFVCIID rVciid,
            PHANDLE phDevice )  [static]
```

Opens the fieldbus adapter with the specified device ID.

Parameters

rVciid	Device ID of the adapter to be opened.
phDevice	Address of a variable of type HANDLE. If run successfully, the function returns the handle of the opened adapter in this variable. In the event of an error, the variable is set to ZERO.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:

7.22.2.22 IxxatVciLibFuncVciEnumDeviceClose()

```
static HRESULT IxxatVciLibFuncVciEnumDeviceClose (
            HANDLE hEnum )  [static]
```

Closes the device list opened with the function vciEnumDeviceOpen.

Parameters

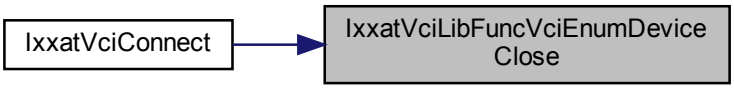
hEnum	Handle of the device list to be closed.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.23 IxxatVciLibFuncVciEnumDeviceNext()

```
static HRESULT IxxatVciLibFuncVciEnumDeviceNext (
            HANDLE hEnum,
            PVCIDEVICEINFO pInfo )  [static]
```

Determines the description of a fieldbus adapter of the device list and increases the internal list index so that a subsequent call of the function supplies the description to the next adapter.

Parameters

hEnum	Handle to the opened device list.
pInfo	Address of a data structure of type VCIDEVICEINFO. If run successfully, the function saves information on the adapter in the memory area specified here.

Returns

VCI_OK if successful, VCI_E_NO_MORE_ITEMS if the list does not contain any more entries, otherwise an error occurred.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:

7.22.2.24 IxxatVciLibFuncVciEnumDeviceOpen()

```
static HRESULT IxxatVciLibFuncVciEnumDeviceOpen (
            PHANDLE hEnum )  [static]
```

Opens the list of all fieldbus adapters registered with the VCI.

Parameters

hEnum	Address of a variable of type HANDLE. If run successfully, the function returns the handle of the opened device list in this variable. In the case of an error, the variable is set to ZERO.

Returns

VCI_OK if successful.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:

7.22.2.25 IxxatVciReceptionThread()

```
static DWORD WINAPI IxxatVciReceptionThread (
            LPVOID pv )  [static]
```

CAN message reception thread.

Parameters

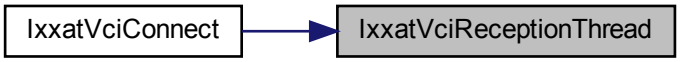
pv	Pointer to thread parameters.

Returns

Thread exit code.

Referenced by IxxatVciConnect().

Here is the caller graph for this function:



7.22.2.26 IxxatVciRegisterEvents()

```
static void IxxatVciRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.22.2.27 IxxatVciTransmit()

```
static bool IxxatVciTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

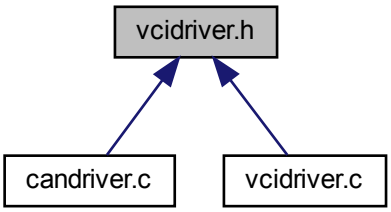
Returns

True if successful, false otherwise.

7.23 vcidriver.h File Reference

Ixxat VCI driver interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * IxxatVciGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.23.1 Detailed Description

Ixxat VCI driver interface header file.

7.23.2 Function Documentation

7.23.2.1 IxxatVciGetInterface()

```
tCanInterface const * IxxatVciGetInterface (
            void  )
```

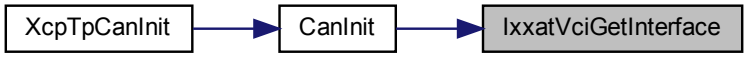
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:

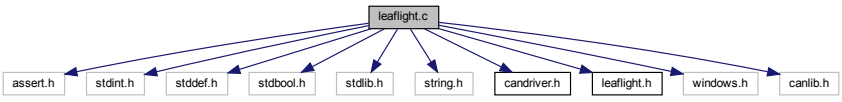


7.24 leaflight.c File Reference

Kvaser Leaf Light v2 interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "candriver.h"
#include "leaflight.h"
#include <windows.h>
#include "canlib.h"
```

Include dependency graph for leaflight.c:



Functions

- static void LeafLightInit (tCanSettings const *settings)

  Initializes the CAN interface.
- static void **LeafLightTerminate** (void)

  Terminates the CAN interface.
- static bool LeafLightConnect (void)

  Connects the CAN interface.
- static void **LeafLightDisconnect** (void)

  Disconnects the CAN interface.
- static bool LeafLightTransmit (tCanMsg const *msg)

  Submits a message for transmission on the CAN bus.
- static bool LeafLightIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void LeafLightRegisterEvents (tCanEvents const *events)

  Registers the event callback functions that should be called by the CAN interface.
- static DWORD WINAPI LeafLightReceptionThread (LPVOID pv)

  CAN message reception thread.
- static void **LeafLightLibLoadDll** (void)

  Loads the Kvaser CANLIB DLL and initializes the API function pointers.
- static void **LeafLightLibUnloadDll** (void)

  Unloads the Kvaser CANLIB DLL and resets the API function pointers.
- static void **LeafLightLibFuncInitializeLibrary** (void)

  This function must be called before any other functions is used. It will initialize the driver.
- static canStatus LeafLightLibFuncUnloadLibrary (void)

  Frees allocated memory, unload the DLLs canlib32.dll has loaded and de- initializes data structures.
- static CanHandle LeafLightLibFuncOpenChannel (int32_t channel, int32_t flags)

  Opens a CAN channel and returns a handle which is used in subsequent calls.
- static canStatus LeafLightLibFuncSetBusParams (const CanHandle hnd, int32_t freq, uint32_t tseg1, uint32_t tseg2, uint32_t sjw, uint32_t noSamp, uint32_t syncmode)

  This function sets the nominal bus timing parameters for the specified CAN controller. The library provides default values for tseg1, tseg2, sjw and noSamp when freq is specified to one of the pre-defined constants, canBITRATE_xxx for classic CAN and canFD_BITRATE_xxx for CAN FD.
- static canStatus LeafLightLibFuncSetBusOutputControl (const CanHandle hnd, const uint32_t drivertype)

  This function sets the driver type for a CAN controller. This corresponds loosely to the bus output control register in the CAN controller, hence the name of this function. CANLIB does not allow for direct manipulation of the bus output control register; instead, symbolic constants are used to select the desired driver type.
- static canStatus LeafLightLibFuncSetAcceptanceFilter (const CanHandle hnd, uint32_t code, uint32_t mask, int32_t is_extended)

  This routine sets the message acceptance filters on a CAN channel.
- static canStatus LeafLightLibFuncIoCtl (const CanHandle hnd, uint32_t func, void *buf, uint32_t buflen)

  This API call performs several different functions; these are described below. The functions are handle-specific unless otherwise noted; this means that they affect only the handle you pass to canIoCtl(), whereas other open handles will remain unaffected. The contents of buf after the call is dependent on the function code you specified.
- static canStatus LeafLightLibFuncBusOn (const CanHandle hnd)

  Takes the specified channel on-bus.
- static canStatus LeafLightLibFuncWrite (const CanHandle hnd, int32_t id, void *msg, uint32_t dlc, uint32_t flag)

  This function sends a CAN message. The call returns immediately after queuing the message to the driver.
- static canStatus LeafLightLibFuncRead (const CanHandle hnd, int32_t *id, void *msg, uint32_t *dlc, uint32_t *flag, uint32_t *time)

  Reads a message from the receive buffer. If no message is available, the function returns immediately with return code canERR_NOMSG.

- static canStatus LeafLightLibFuncReadStatus (const CanHandle hnd, uint32_t *const flags)

  Returns the status for the specified circuit. flags points to a longword which receives a combination of the canSTAT_xxx flags.
- static canStatus LeafLightLibFuncBusOff (const CanHandle hnd)

  Takes the specified channel off-bus.
- static canStatus LeafLightLibFuncClose (const CanHandle hnd)

  Closes the channel associated with the handle. If no other threads are using the CAN circuit, it is taken off bus. The handle can not be used for further references to the channel, so any variable containing it should be zeroed.
- tCanInterface const * LeafLightGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **leafLightInterface**

  CAN interface structure filled with Kvaser Leaf Light v2 specifics.
- static tCanSettings **leafLightSettings**

  The settings to use in this CAN interface.
- static tCanEvents * **leafLightEventsList**

  List with callback functions that this driver should use.
- static uint32_t **leafLightEventsEntries**

  Total number of event entries into the leafLightEventsList list.
- static HINSTANCE **leafLightDllHandle**

  Handle to the Kvaser CANLIB dynamic link library.
- static CanHandle **leafLightCanHandle**

  Handle to the CAN channel.
- static CanHandle **leafLightRxCanHandle**

  Handle to the CAN channel for usage in the CAN reception thread.
- static tLeafLightLibFuncInitializeLibrary **leafLightLibFuncInitializeLibraryPtr**

  Function pointer to the Kvaser CANLIB canInitializeLibrary function.
- static tLeafLightLibFuncUnloadLibrary **leafLightLibFuncUnloadLibraryPtr**

  Function pointer to the Kvaser CANLIB canUnloadLibrary function.
- static tLeafLightLibFuncOpenChannel **leafLightLibFuncOpenChannelPtr**

  Function pointer to the Kvaser CANLIB canOpenChannel function.
- static tLeafLightLibFuncSetBusParams **leafLightLibFuncSetBusParamsPtr**

  Function pointer to the Kvaser CANLIB canSetBusParams function.
- static tLeafLightLibFuncSetBusOutputControl **leafLightLibFuncSetBusOutputControlPtr**

  Function pointer to the Kvaser CANLIB canSetBusOutputControl function.
- static tLeafLightLibFuncSetAcceptanceFilter **leafLightLibFuncSetAcceptanceFilterPtr**

  Function pointer to the Kvaser CANLIB canSetAcceptanceFilter function.
- static tLeafLightLibFuncIoCtl **leafLightLibFuncIoCtlPtr**

  Function pointer to the Kvaser CANLIB canIoCtl function.
- static tLeafLightLibFuncBusOn **leafLightLibFuncBusOnPtr**

  Function pointer to the Kvaser CANLIB canBusOn function.
- static tLeafLightLibFuncWrite **leafLightLibFuncWritePtr**

  Function pointer to the Kvaser CANLIB canWrite function.
- static tLeafLightLibFuncRead **leafLightLibFuncReadPtr**

  Function pointer to the Kvaser CANLIB canRead function.
- static tLeafLightLibFuncReadStatus **leafLightLibFuncReadStatusPtr**

  Function pointer to the Kvaser CANLIB canReadStatus function.
- static tLeafLightLibFuncBusOff **leafLightLibFuncBusOffPtr**

Function pointer to the Kvaser CANLIB canBusOff function.

- static tLeafLightLibFuncClose **leafLightLibFuncClosePtr**

  Function pointer to the Kvaser CANLIB canClose function.

- static HANDLE **leafLightTerminateEvent**

  Handle for the event to terminate the reception thread.

- static HANDLE **leafLightCanEvent**

  Handle for a CAN related event.

- static HANDLE **leafLightRxThreadHandle**

  Handle for the CAN reception thread.

7.24.1 Detailed Description

Kvaser Leaf Light v2 interface source file.

7.24.2 Function Documentation

7.24.2.1 LeafLightConnect()

```
static bool LeafLightConnect (
            void ) [static]
```

Connects the CAN interface.

Returns

True if connected, false otherwise.

7.24.2.2 LeafLightGetInterface()

```
tCanInterface const * LeafLightGetInterface (
            void )
```

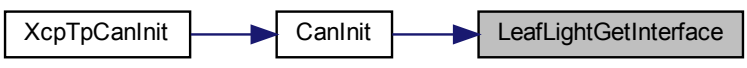
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.24.2.3 LeafLightInit()

```
static void LeafLightInit (
            tCanSettings const * settings )  [static]
```

Initializes the CAN interface.

Parameters

settings	Pointer to the CAN interface settings.

7.24.2.4 LeafLightIsBusError()

```
static bool LeafLightIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.24.2.5 LeafLightLibFuncBusOff()

```
static canStatus LeafLightLibFuncBusOff (
            const CanHandle hnd )  [static]
```

Takes the specified channel off-bus.

Parameters

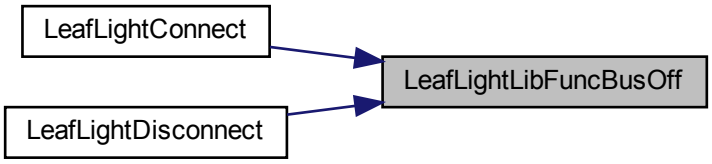
hnd	A handle to an open circuit.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect(), and LeafLightDisconnect().

Here is the caller graph for this function:



7.24.2.6 LeafLightLibFuncBusOn()

```
static canStatus LeafLightLibFuncBusOn (
            const CanHandle hnd )  [static]
```

Takes the specified channel on-bus.

Parameters

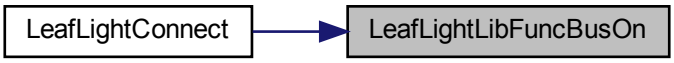
hnd	An open handle to a CAN channel.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:



7.24.2.7 LeafLightLibFuncClose()

```
static canStatus LeafLightLibFuncClose (
            const CanHandle hnd )  [static]
```

Closes the channel associated with the handle. If no other threads are using the CAN circuit, it is taken off bus. The handle can not be used for further references to the channel, so any variable containing it should be zeroed.

Parameters

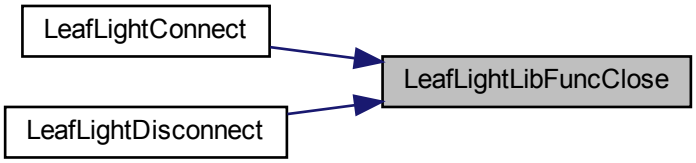
hnd	A handle to an open circuit.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect(), and LeafLightDisconnect().

Here is the caller graph for this function:



7.24.2.8 LeafLightLibFuncIoCtl()

```
static canStatus LeafLightLibFuncIoCtl (
            const CanHandle hnd,
            uint32_t func,
            void * buf,
            uint32_t buflen )  [static]
```

This API call performs several different functions; these are described below. The functions are handle-specific unless otherwise noted; this means that they affect only the handle you pass to canIoCtl(), whereas other open handles will remain unaffected. The contents of buf after the call is dependent on the function code you specified.

Parameters

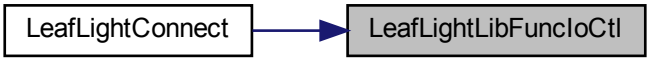
hnd	A handle to an open circuit.
func	A canIOCTL_xxx function code.
buf	Pointer to a buffer containing function-dependent data; or a NULL pointer for certain function codes. The buffer can be used for both input and output depending on the function code. See canIOCTL_xxx.
buflen	The length of the buffer.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:



7.24.2.9 LeafLightLibFuncOpenChannel()

```
static CanHandle LeafLightLibFuncOpenChannel (
            int32_t channel,
            int32_t flags )  [static]
```

Opens a CAN channel and returns a handle which is used in subsequent calls.

Parameters

channel	The number of the channel. Channel numbering is hardware dependent.
flags	A combination of canOPEN_xxx flags.

Returns

Handle (positive) to the channel if successful, canERR_xxx (negative) otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:

7.24.2.10 LeafLightLibFuncRead()

```
static canStatus LeafLightLibFuncRead (
             const CanHandle hnd,
             int32_t * id,
             void * msg,
             uint32_t * dlc,
             uint32_t * flag,
             uint32_t * time )  [static]
```

Reads a message from the receive buffer. If no message is available, the function returns immediately with return code canERR_NOMSG.

Parameters

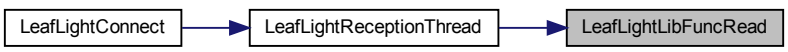
hnd	A handle to an open circuit.
id	Pointer to a buffer which receives the CAN identifier. This buffer will only get the identifier. To determine whether this identifier was standard (11-bit) or extended (29-bit), and/or whether it was remote or not, or if it was an error frame, examine the contents of the flag argument.
msg	Pointer to the buffer which receives the message data. This buffer must be large enough (i.e. 8 bytes.) Only the message data is copied; the rest of the buffer is left as-is.
dlc	Pointer to a buffer which receives the message length.
flag	Pointer to a buffer which receives the message flags, which is a combination of the canMSG_xxx and canMSGERR_xxx values.
time	Pointer to a buffer which receives the message time stamp.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightReceptionThread().

Here is the caller graph for this function:



7.24.2.11 LeafLightLibFuncReadStatus()

```
static canStatus LeafLightLibFuncReadStatus (
             const CanHandle hnd,
             uint32_t *const flags )  [static]
```

Returns the status for the specified circuit. flags points to a longword which receives a combination of the can↩ STAT_xxx flags.

Parameters

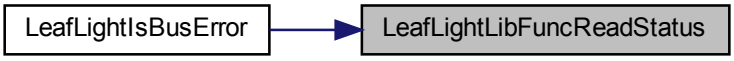
hnd	A handle to an open circuit.
flags	Pointer to a DWORD which receives the status flags; this is a combination of any of the canSTAT_xxx.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightIsBusError().

Here is the caller graph for this function:



7.24.2.12 LeafLightLibFuncSetAcceptanceFilter()

```
static canStatus LeafLightLibFuncSetAcceptanceFilter (
            const CanHandle hnd,
            uint32_t code,
            uint32_t mask,
            int32_t is_extended )  [static]
```

This routine sets the message acceptance filters on a CAN channel.

Parameters

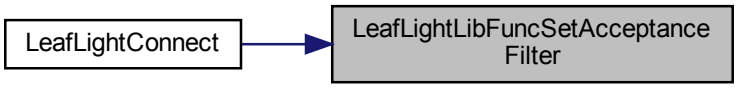
hnd	A handle to an open circuit.
code	The acceptance code to set.
mask	The acceptance mask to set
is_extended	Select 29-bit CAN identifiers.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:



7.24.2.13 LeafLightLibFuncSetBusOutputControl()

```
static canStatus LeafLightLibFuncSetBusOutputControl (
            const CanHandle hnd,
            const uint32_t drivertype )  [static]
```

This function sets the driver type for a CAN controller. This corresponds loosely to the bus output control register in the CAN controller, hence the name of this function. CANLIB does not allow for direct manipulation of the bus output control register; instead, symbolic constants are used to select the desired driver type.

Parameters

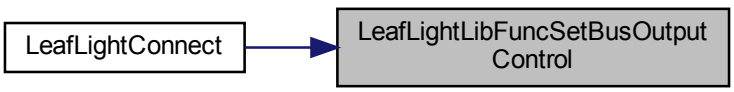
hnd	A handle to an open circuit.
drivertype	Can driver type (canDRIVER_xxx).

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:



7.24.2.14 LeafLightLibFuncSetBusParams()

```
static canStatus LeafLightLibFuncSetBusParams (
             const CanHandle hnd,
             int32_t freq,
             uint32_t tseg1,
             uint32_t tseg2,
             uint32_t sjw,
             uint32_t noSamp,
             uint32_t syncmode )  [static]
```

This function sets the nominal bus timing parameters for the specified CAN controller. The library provides default values for tseg1, tseg2, sjw and noSamp when freq is specified to one of the pre-defined constants, canBITRATE↩ _xxx for classic CAN and canFD_BITRATE_xxx for CAN FD.

Parameters

hnd	An open handle to a CAN controller.
freq	Bit rate (measured in bits per second); or one of the predefined constants (canBITRATE_xxx for classic CAN and canFD_BITRATE_xxx for CAN FD).
tseg1	Time segment 1, that is, the number of quanta from (but not including) the Sync Segment to the sampling point.
tseg2	Time segment 2, that is, the number of quanta from the sampling point to the end of the bit.
sjw	The Synchronization Jump Width.
noSamp	The number of sampling points; can be 1 or 3.
syncmode	Unsupported and ignored.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightConnect().

Here is the caller graph for this function:

7.24.2.15 LeafLightLibFuncUnloadLibrary()

```
static canStatus LeafLightLibFuncUnloadLibrary (
             void  )  [static]
```

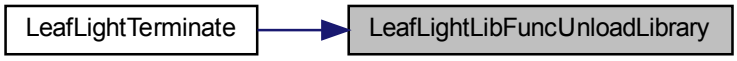
Frees allocated memory, unload the DLLs canlib32.dll has loaded and de- initializes data structures.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightTerminate().

Here is the caller graph for this function:



7.24.2.16 LeafLightLibFuncWrite()

```
static canStatus LeafLightLibFuncWrite (
            const CanHandle hnd,
            int32_t id,
            void * msg,
            uint32_t dlc,
            uint32_t flag )  [static]
```

This function sends a CAN message. The call returns immediately after queuing the message to the driver.

Parameters

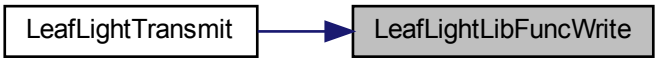
hnd	A handle to an open CAN circuit.
id	The identifier of the CAN message to send.
msg	A pointer to the message data, or NULL.
dlc	The length of the message in bytes.
flag	A combination of message flags, canMSG_xxx (including canFDMSG_xxx if the CAN FD protocol is enabled). Use this parameter to send extended (29-bit) frames and/or remote frames. Use canMSG_EXT and/or canMSG_RTR for this purpose.

Returns

canOK if successful, canERR_xxx otherwise.

Referenced by LeafLightTransmit().

Here is the caller graph for this function:



7.24.2.17 LeafLightReceptionThread()

```
static DWORD WINAPI LeafLightReceptionThread (
            LPVOID pv )  [static]
```

CAN message reception thread.

Parameters

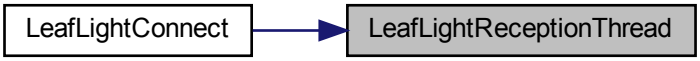
pv	Pointer to thread parameters.

Returns

Thread exit code.

Referenced by LeafLightConnect().

Here is the caller graph for this function:



7.24.2.18 LeafLightRegisterEvents()

```
static void LeafLightRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.24.2.19 LeafLightTransmit()

```
static bool LeafLightTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

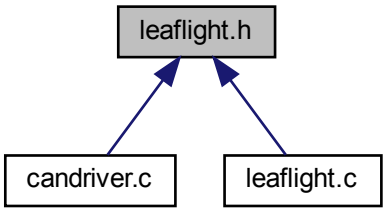
Returns

True if successful, false otherwise.

7.25 leaflight.h File Reference

Kvaser Leaf Light v2 interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * LeafLightGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.25.1 Detailed Description

Kvaser Leaf Light v2 interface header file.

7.25.2 Function Documentation

7.25.2.1 LeafLightGetInterface()

```
tCanInterface const * LeafLightGetInterface (
            void  )
```

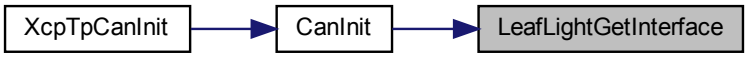
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:

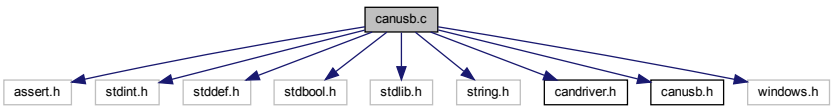


7.26 canusb.c File Reference

Lawicel CANUSB interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "candriver.h"
#include "canusb.h"
#include <windows.h>
#include "lawicel_can.h"
```

Include dependency graph for canusb.c:



Functions

- static void CanUsbInit (tCanSettings const *settings)

  Initializes the CAN interface.
- static void **CanUsbTerminate** (void)

  Terminates the CAN interface.
- static bool CanUsbConnect (void)

  Connects the CAN interface.
- static void **CanUsbDisconnect** (void)

  Disconnects the CAN interface.
- static bool CanUsbTransmit (tCanMsg const *msg)

  Submits a message for transmission on the CAN bus.
- static bool CanUsbIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void CanUsbRegisterEvents (tCanEvents const *events)

  Registers the event callback functions that should be called by the CAN interface.
- static bool CanUsbOpenChannel (void)

  Opens the CAN channel. Note that the opening of the CAN channel takes a long time in the Lawicel CANUSB API, therefore this is not done in CanUsbConnect() for this CAN interface.
- static bool CanUsbCloseChannel (void)

  Closes the CAN channel. Note that the closing of the CAN channel takes a long time in the Lawicel CANUSB API, therefore this is not done in CanUsbDisconnect() for this CAN interface.
- static void **CanUsbLibLoadDll** (void)

  Loads the Lawicel CANUSBDRV DLL and initializes the API function pointers.
- static void **CanUsbLibUnloadDll** (void)

  Unloads the Lawicel CANUSBDRV DLL and resets the API function pointers.
- static void __stdcall CanUsbLibReceiveCallback (CANMsg const *pMsg)

  Callback function that gets called by the Lawicel CANUSB API each time a CAN message was received.
- static CANHANDLE CanUsbLibFuncOpen (LPCSTR szID, LPCSTR szBitrate, uint32_t acceptance_code, uint32_t acceptance_mask, uint32_t flags)

  Open a channel to a physical CAN interface.
- static int32_t CanUsbLibFuncClose (CANHANDLE h)

  Close channel with handle h.
- static int32_t CanUsbLibFuncWrite (CANHANDLE h, CANMsg *msg)

  Write message to channel with handle h.
- static int32_t CanUsbLibFuncStatus (CANHANDLE h)

  Get Adapter status for channel with handle h.
- static int32_t CanUsbLibFuncSetReceiveCallBack (CANHANDLE h, LPFNDLL_RECEIVE_CALLBACK fn)

  With this method one can define a function that will receive all incoming messages.
- tCanInterface const * CanUsbGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **canUsbInterface**

  CAN interface structure filled with Lawicel CANUSB specifics.
- static tCanSettings **canUsbSettings**

  The settings to use in this CAN interface.
- static tCanEvents * **canUsbEventsList**

  List with callback functions that this driver should use.
- static uint32_t **canUsbEventsEntries**

Total number of event entries into the canUsbEventsList list.

- static HINSTANCE **canUsbDllHandle**

  Handle to the Lawicel CANUSB dynamic link library.
- static CANHANDLE **canUsbCanHandle**

  Handle to the CAN channel.
- static tCanUsbLibFuncOpen **canUsbLibFuncOpenPtr**

  Function pointer to the Lawicel CANUSB canusb_Open function.
- static tCanUsbLibFuncClose **canUsbLibFuncClosePtr**

  Function pointer to the Lawicel CANUSB canusb_Close function.
- static tCanUsbLibFuncWrite **canUsbLibFuncWritePtr**

  Function pointer to the Lawicel CANUSB canusb_Write function.
- static tCanUsbLibFuncStatus **canUsbLibFuncStatusPtr**

  Function pointer to the Lawicel CANUSB canusb_Status function.
- static tCanUsbLibFuncSetReceiveCallBack **canUsbLibFuncSetReceiveCallBackPtr**

  Function pointer to the Lawicel CANUSB canusb_setReceiveCallBack function.

7.26.1 Detailed Description

Lawicel CANUSB interface source file.

7.26.2 Function Documentation

7.26.2.1 CanUsbCloseChannel()

```
static bool CanUsbCloseChannel (
            void  )  [static]
```

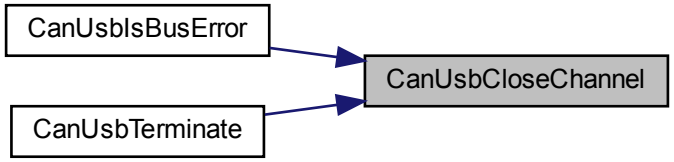
Closes the CAN channel. Note that the closing of the CAN channel takes a long time in the Lawicel CANUSB API, therefore this is not done in CanUsbDisconnect() for this CAN interface.

Returns

True if successful, false otherwise.

Referenced by CanUsbIsBusError(), and CanUsbTerminate().

Here is the caller graph for this function:



7.26.2.2 CanUsbConnect()

```
static bool CanUsbConnect (
            void ) [static]
```

Connects the CAN interface.

Returns

True if connected, false otherwise.

7.26.2.3 CanUsbGetInterface()

```
tCanInterface const * CanUsbGetInterface (
            void )
```

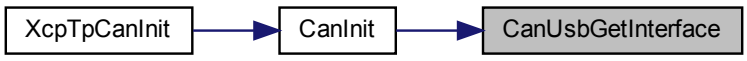
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.26.2.4 CanUsbInit()

```
static void CanUsbInit (
            tCanSettings const * settings ) [static]
```

Initializes the CAN interface.

Parameters

settings	Pointer to the CAN interface settings.

7.26.2.5 CanUsbIsBusError()

```
static bool CanUsbIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.26.2.6 CanUsbLibFuncClose()

```
static int32_t CanUsbLibFuncClose (
            CANHANDLE h )  [static]
```

Close channel with handle h.

Parameters

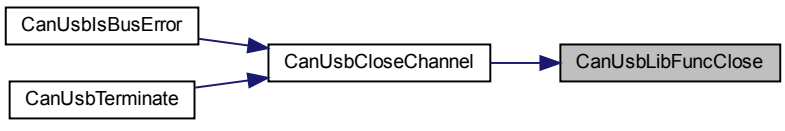
h	Handle to the opened device.

Returns

> 0 if successful, ERROR_CANUSB_xxx (<= 0) otherwise.

Referenced by CanUsbCloseChannel().

Here is the caller graph for this function:



7.26.2.7 CanUsbLibFuncOpen()

```
static CANHANDLE CanUsbLibFuncOpen (
            LPCSTR szID,
            LPCSTR szBitrate,
            uint32_t acceptance_code,
            uint32_t acceptance_mask,
            uint32_t flags )  [static]
```

Open a channel to a physical CAN interface.

Parameters

szID	Serial number for adapter or NULL to open the first found.
szBitrate	"10", "20", "50", "100", "250", "500", "800" or "1000" (kbps) or as a btr pair. btr0:btr1 pair ex. "0x03:0x1c" can be used to set a custom baudrate.
acceptance_code	Set to CANUSB_ACCEPTANCE_CODE_ALL to get all messages or another code to filter messages.
acceptance_mask	Set to CANUSB_ACCEPTANCE_MASK_ALL to get all messages or another code to filter messages.
flags	Optional flags CANUSB_FLAG_xxx.

Returns

Handle to device if open was successful or zero on failure.

Referenced by CanUsbOpenChannel().

Here is the caller graph for this function:

7.26.2.8 CanUsbLibFuncSetReceiveCallBack()

```
static int32_t CanUsbLibFuncSetReceiveCallBack (
            CANHANDLE h,
            LPFNDLL_RECEIVE_CALLBACK fn )  [static]
```

With this method one can define a function that will receive all incoming messages.

Parameters

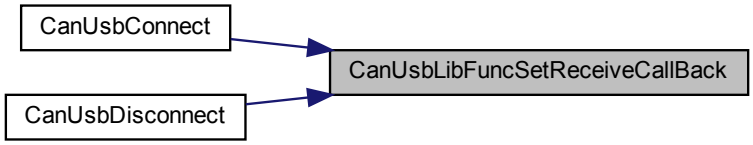
h	Handle to the opened device.
fn	Pointer to the callback function to set. NULL removes it again.

Returns

> 0 if successful, ERROR_CANUSB_xxx (<= 0) otherwise.

Referenced by CanUsbConnect(), and CanUsbDisconnect().

Here is the caller graph for this function:



7.26.2.9 CanUsbLibFuncStatus()

```
static int32_t CanUsbLibFuncStatus (
            CANHANDLE h )  [static]
```

Get Adapter status for channel with handle h.

Parameters

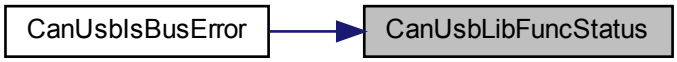
h	Handle to the opened device.

Returns

CANSTATUS_xxx if status info is set, 0 otherwise.

Referenced by CanUsbIsBusError().

Here is the caller graph for this function:



7.26.2.10 CanUsbLibFuncWrite()

```
static int32_t CanUsbLibFuncWrite (
            CANHANDLE h,
            CANMsg * msg )  [static]
```

Write message to channel with handle h.

Parameters

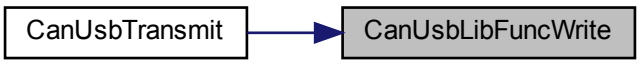
h	Handle to the opened device.
msg	CAN message to send.

Returns

> 0 if successful, ERROR_CANUSB_xxx (<= 0) otherwise.

Referenced by CanUsbTransmit().

Here is the caller graph for this function:



7.26.2.11 CanUsbLibReceiveCallback()

```
static void __stdcall CanUsbLibReceiveCallback (
            CANMsg const * pMsg )  [static]
```

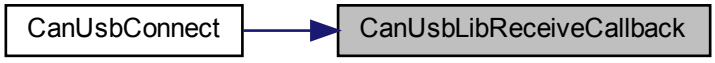
Callback function that gets called by the Lawicel CANUSB API each time a CAN message was received.

Parameters

pMsg	Pointer to the received CAN message.

Referenced by CanUsbConnect().

Here is the caller graph for this function:



7.26.2.12 CanUsbOpenChannel()

```
static bool CanUsbOpenChannel (
            void  )  [static]
```

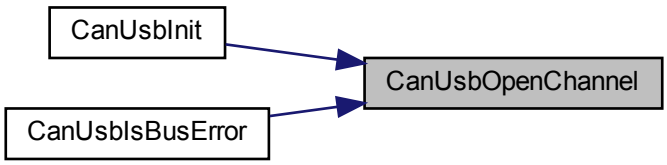
Opens the CAN channel. Note that the opening of the CAN channel takes a long time in the Lawicel CANUSB API, therefore this is not done in CanUsbConnect() for this CAN interface.

Returns

True if successful, false otherwise.

Referenced by CanUsbInit(), and CanUsbIsBusError().

Here is the caller graph for this function:



7.26.2.13 CanUsbRegisterEvents()

```
static void CanUsbRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.26.2.14 CanUsbTransmit()

```
static bool CanUsbTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

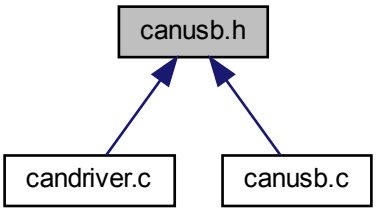
Returns

True if successful, false otherwise.

7.27 canusb.h File Reference

Lawicel CANUSB interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * CanUsbGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.27.1 Detailed Description

Lawicel CANUSB interface header file.

7.27.2 Function Documentation

7.27.2.1 CanUsbGetInterface()

```
tCanInterface const * CanUsbGetInterface (
            void  )
```

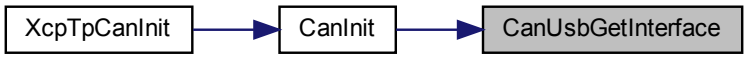
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:

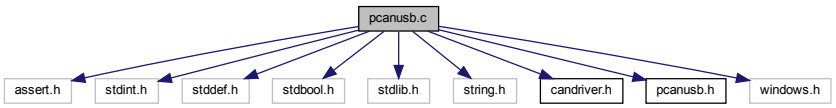


7.28 pcanusb.c File Reference

Peak PCAN-USB interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "candriver.h"
#include "pcanusb.h"
#include <windows.h>
#include "PCANBasic.h"
```

Include dependency graph for pcanusb.c:



Macros

- #define **PCANUSB_BUSOFF_AUTORECOVERY_ENABLE** (0u)

  Configurable to enabled/disable the automatic CAN bus off recovery feature. Testing shows that it is better to leave this disabled. If no connection with the target can be made, the PCAN-USB automatically re-initialized anyway.

Functions

- static void PCanUsbInit (tCanSettings const *settings)

  Initializes the CAN interface.
- static void **PCanUsbTerminate** (void)

  Terminates the CAN interface.
- static bool PCanUsbConnect (void)

  Connects the CAN interface.
- static void **PCanUsbDisconnect** (void)

  Disconnects the CAN interface.
- static bool PCanUsbTransmit (tCanMsg const *msg)

  Submits a message for transmission on the CAN bus.
- static bool PCanUsbIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void PCanUsbRegisterEvents (tCanEvents const *events)

  Registers the event callback functions that should be called by the CAN interface.
- static DWORD WINAPI PCanUsbReceptionThread (LPVOID pv)

  CAN message reception thread.
- static void **PCanUsbLibLoadDll** (void)

  Loads the PCAN-Basic DLL and initializes the API function pointers.
- static void **PCanUsbLibUnloadDll** (void)

  Unloads the PCAN-Basic DLL and resets the API function pointers.
- static TPCANStatus PCanUsbLibFuncInitialize (TPCANHandle Channel, TPCANBaudrate Btr0Btr1, TPCANType HwType, DWORD IOPort, WORD Interrupt)

  Initializes a PCAN Channel.
- static TPCANStatus PCanUsbLibFuncUninitialize (TPCANHandle Channel)

  Uninitializes a PCAN Channel.
- static TPCANStatus PCanUsbLibFuncGetStatus (TPCANHandle Channel)

  Gets the current BUS status of a PCAN Channel.
- static TPCANStatus PCanUsbLibFuncSetValue (TPCANHandle Channel, TPCANParameter Parameter, void *Buffer, DWORD BufferLength)

  Sets a configuration or information value within a PCAN Channel.
- static TPCANStatus PCanUsbLibFuncRead (TPCANHandle Channel, TPCANMsg *MessageBuffer, TPCANTimestamp *TimestampBuffer)

  Reads a CAN message from the receive queue of a PCAN Channel.
- static TPCANStatus PCanUsbLibFuncWrite (TPCANHandle Channel, TPCANMsg *MessageBuffer)

  Transmits a CAN message.
- static TPCANStatus PCanUsbLibFuncFilterMessages (TPCANHandle Channel, DWORD FromID, DWORD ToID, TPCANMode Mode)

  Configures the reception filter.
- tCanInterface const * PCanUsbGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **pCanUsbInterface**

  CAN interface structure filled with Peak PCAN-USB specifics.
- static const TPCANHandle **pCanUsbChannelLookup** [ ]

  PCAN-USB channel handle lookup table. The pCanUsbSettings.channel value can be used as the index.
- static tCanSettings **pCanUsbSettings**

  The settings to use in this CAN interface.
- static tCanEvents * **pCanUsbEventsList**

  List with callback functions that this driver should use.
- static uint32_t **pCanUsbEventsEntries**

  Total number of event entries into the pCanUsbEventsList list.
- static HINSTANCE **pCanUsbDllHandle**

  Handle to the PCAN-Basic dynamic link library.
- static tPCanUsbLibFuncInitialize **pCanUsbLibFuncInitializePtr**

  Function pointer to the PCAN-Basic Initialize function.
- static tPCanUsbLibFuncUninitialize **pCanUsbLibFuncUninitializePtr**

  Function pointer to the PCAN-Basic Uninitialize function.
- static tPCanUsbLibFuncGetStatus **pCanUsbLibFuncGetStatusPtr**

  Function pointer to the PCAN-Basic GetStatus function.
- static tPCanUsbLibFuncSetValue **pCanUsbLibFuncSetValuePtr**

  Function pointer to the PCAN-Basic SetValue function.
- static tPCanUsbLibFuncRead **pCanUsbLibFuncReadPtr**

  Function pointer to the PCAN-Basic Read function.
- static tPCanUsbLibFuncWrite **pCanUsbLibFuncWritePtr**

  Function pointer to the PCAN-Basic Write function.
- static tPCanUsbLibFuncFilterMessages **pCanUsbLibFuncFilterMessagesPtr**

  Function pointer to the PCAN-Basic FilterMessages function.
- static HANDLE **pCanUsbTerminateEvent**

  Handle for the event to terminate the reception thread.
- static HANDLE **pCanUsbCanEvent**

  Handle for a CAN related event.
- static HANDLE **pCanUsbRxThreadHandle**

  Handle for the CAN reception thread.

7.28.1 Detailed Description

Peak PCAN-USB interface source file.

7.28.2 Function Documentation

7.28.2.1 PCanUsbConnect()

```
static bool PCanUsbConnect (
            void ) [static]
```

Connects the CAN interface.

Returns

True if connected, false otherwise.

7.28.2.2 PCanUsbGetInterface()

```
tCanInterface const * PCanUsbGetInterface (
            void )
```

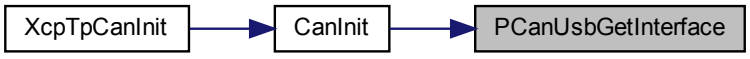
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.28.2.3 PCanUsbInit()

```
static void PCanUsbInit (
            tCanSettings const * settings ) [static]
```

Initializes the CAN interface.

Parameters

settings	Pointer to the CAN interface settings.

7.28.2.4 PCanUsbIsBusError()

```
static bool PCanUsbIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.28.2.5 PCanUsbLibFuncFilterMessages()

```
static TPCANStatus PCanUsbLibFuncFilterMessages (
            TPCANHandle Channel,
            DWORD FromID,
            DWORD ToID,
            TPCANMode Mode )  [static]
```

Configures the reception filter.

Parameters

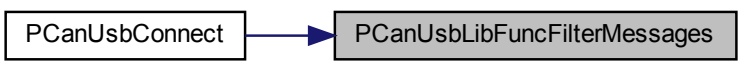
Channel	The handle of a PCAN Channel.
FromID	The lowest CAN ID wanted to be received.
ToID	The highest CAN ID wanted to be received.
Mode	The type of the filter being set.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbConnect().

Here is the caller graph for this function:



7.28.2.6 PCanUsbLibFuncGetStatus()

```
static TPCANStatus PCanUsbLibFuncGetStatus (
            TPCANHandle Channel )  [static]
```

Gets the current BUS status of a PCAN Channel.

Parameters

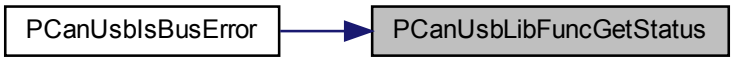
Channel	The handle of a PCAN Channel.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbIsBusError().

Here is the caller graph for this function:



7.28.2.7 PCanUsbLibFuncInitialize()

```
static TPCANStatus PCanUsbLibFuncInitialize (
            TPCANHandle Channel,
            TPCANBaudrate Btr0Btr1,
            TPCANType HwType,
            DWORD IOPort,
            WORD Interrupt )  [static]
```

Initializes a PCAN Channel.

Parameters

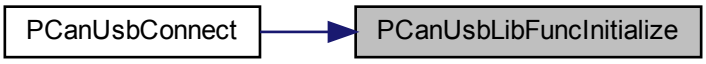
Channel	The handle of a PCAN Channel.
Btr0Btr1	The speed for the communication (BTR0BTR1 code).
HwType	The type of the Non-Plug-and-Play hardware and its operation mode.
IOPort	The I/O address for the parallel port of the Non-Plug-and-Play hardware.
Interrupt	The Interrupt number of the parallel port of the Non-Plug- and-Play hardware.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbConnect().

Here is the caller graph for this function:



7.28.2.8 PCanUsbLibFuncRead()

```
static TPCANStatus PCanUsbLibFuncRead (
            TPCANHandle Channel,
            TPCANMsg * MessageBuffer,
            TPCANTimestamp * TimestampBuffer )  [static]
```

Reads a CAN message from the receive queue of a PCAN Channel.

Parameters

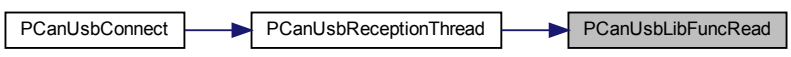
Channel	The handle of a PCAN Channel.
MessageBuffer	A TPCANMsg buffer to store the CAN message.
TimestampBuffer	A TPCANTimestamp buffer to get the reception time of the message.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbReceptionThread().

Here is the caller graph for this function:



7.28.2.9 PCanUsbLibFuncSetValue()

```
static TPCANStatus PCanUsbLibFuncSetValue (
            TPCANHandle Channel,
            TPCANParameter Parameter,
            void * Buffer,
            DWORD BufferLength )  [static]
```

Sets a configuration or information value within a PCAN Channel.

Parameters

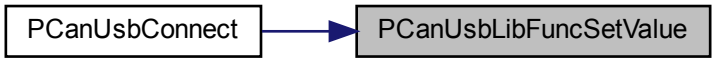
Channel	The handle of a PCAN Channel.
Parameter	The code of the value to be set .
Buffer	The buffer containing the value to be set.
BufferLength	The length in bytes of the given buffer.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbConnect().

Here is the caller graph for this function:



7.28.2.10 PCanUsbLibFuncUninitialize()

```
static TPCANStatus PCanUsbLibFuncUninitialize (
            TPCANHandle Channel )  [static]
```

Uninitializes a PCAN Channel.

Parameters

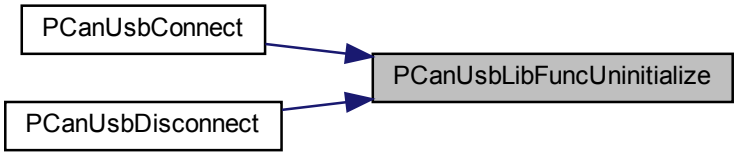
Channel	The handle of a PCAN Channel.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbConnect(), and PCanUsbDisconnect().

Here is the caller graph for this function:



7.28.2.11 PCanUsbLibFuncWrite()

```
static TPCANStatus PCanUsbLibFuncWrite (
            TPCANHandle Channel,
            TPCANMsg * MessageBuffer )  [static]
```

Transmits a CAN message.

Parameters

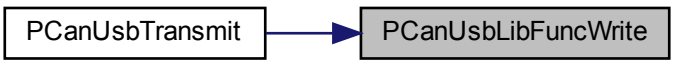
Channel	The handle of a PCAN Channel.
MessageBuffer	A TPCANMsg buffer containing the CAN message to be sent.

Returns

The return value is a TPCANStatus code. PCAN_ERROR_OK is returned on success.

Referenced by PCanUsbTransmit().

Here is the caller graph for this function:



7.28.2.12 PCanUsbReceptionThread()

```
static DWORD WINAPI PCanUsbReceptionThread (
            LPVOID pv )  [static]
```

CAN message reception thread.

Parameters

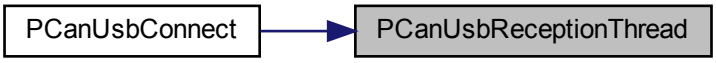
pv	Pointer to thread parameters.

Returns

Thread exit code.

Referenced by PCanUsbConnect().

Here is the caller graph for this function:



7.28.2.13 PCanUsbRegisterEvents()

```
static void PCanUsbRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.28.2.14 PCanUsbTransmit()

```
static bool PCanUsbTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

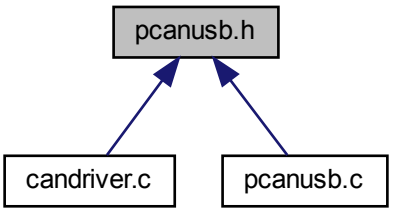
Returns

True if successful, false otherwise.

7.29 pcanusb.h File Reference

Peak PCAN-USB interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * PCanUsbGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.29.1 Detailed Description

Peak PCAN-USB interface header file.

7.29.2 Function Documentation

7.29.2.1 PCanUsbGetInterface()

```
tCanInterface const * PCanUsbGetInterface (
            void  )
```

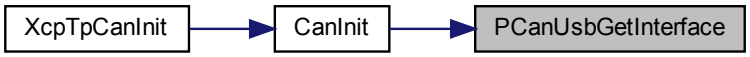
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:

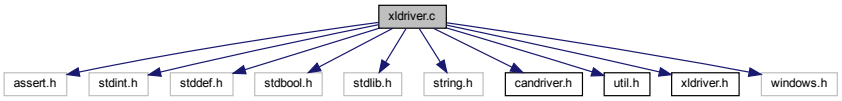


7.30 xldriver.c File Reference

Vector XL driver interface source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "candriver.h"
#include "util.h"
#include "xldriver.h"
#include <windows.h>
#include <vxlapi.h>
```

Include dependency graph for xldriver.c:



Macros

- #define **VECTOR_XL_RX_QUEUE_SIZE** (4096u)

  Internal driver queue size in CAN events.

Functions

- static void VectorXlInit (tCanSettings const ∗settings)

  Initializes the CAN interface.
- static void **VectorXlTerminate** (void)

  Terminates the CAN interface.
- static bool VectorXlConnect (void)

  Connects the CAN interface.
- static void **VectorXlDisconnect** (void)

  Disconnects the CAN interface.
- static bool VectorXlTransmit (tCanMsg const ∗msg)

  Submits a message for transmission on the CAN bus.
- static bool VectorXlIsBusError (void)

  Checks if a bus off or bus heavy situation occurred.
- static void VectorXlRegisterEvents (tCanEvents const ∗events)

  Registers the event callback functions that should be called by the CAN interface.
- static uint32_t VectorXlConvertToRawBitrate (tCanBaudrate baudrate)

  Converts the baudrate enumerated type value to a bitrate in bits/second.
- static DWORD WINAPI VectorXlReceptionThread (LPVOID pv)

  CAN event reception thread.
- tCanInterface const ∗ VectorXlGetInterface (void)

  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Variables

- static const tCanInterface **pVectorXlInterface**

  CAN interface structure filled with Vector XL driver specifics.
- static tCanSettings **vectorXlSettings**

  The settings to use in this CAN interface.
- static tCanEvents ∗ **vectorXlEventsList**

  List with callback functions that this driver should use.
- static uint32_t **vectorXlEventsEntries**

  Total number of event entries into the vectorXlEventsList list.
- static XLportHandle **vectorXlPortHandle**

  The handle to the CAN port needed for API functions.
- static XLaccess **vectorXLChannelMask**

  The mask for the configured CAN channel.
- static bool **vectorXlDriverOpened**

  Boolean flag to track if the driver was opened or not.
- static bool **vectorXlPortOpened**

  Boolean flag to track if the port was opened or not.
- static bool **vectorXlChannelActivated**

  Boolean flag to track if the channel was activated or not.
- static bool **vectorXlBusErrorDetected**

  Boolean flag to detect if a CAN bus error state was detected.
- static HANDLE **vectorXlTerminateEvent**

  Handle for the event to terminate the reception thread.
- static HANDLE **vectorXlCanEvent**

  Handle for a CAN related event.
- static HANDLE **vectorXlRxThreadHandle**

  Handle for the CAN reception thread.

7.30.1 Detailed Description

Vector XL driver interface source file.

7.30.2 Function Documentation

7.30.2.1 VectorXlConnect()

```
static bool VectorXlConnect (
            void  )  [static]
```

Connects the CAN interface.

Returns

True if connected, false otherwise.

7.30.2.2 VectorXlConvertToRawBitrate()

```
static uint32_t VectorXlConvertToRawBitrate (
            tCanBaudrate baudrate )  [static]
```

Converts the baudrate enumerated type value to a bitrate in bits/second.

Parameters

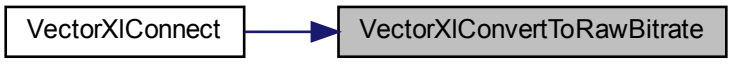
baudrate	Baudrate enumarated type.

Returns

Bitrate in bits/second.

Referenced by VectorXlConnect().

Here is the caller graph for this function:



7.30.2.3 VectorXlGetInterface()

```
tCanInterface const * VectorXlGetInterface (
            void  )
```

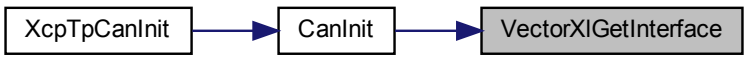
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

Here is the caller graph for this function:



7.30.2.4 VectorXlInit()

```
static void VectorXlInit (
            tCanSettings const * settings )  [static]
```

Initializes the CAN interface.

Parameters

settings	Pointer to the CAN interface settings.

7.30.2.5 VectorXlIsBusError()

```
static bool VectorXlIsBusError (
            void  )  [static]
```

Checks if a bus off or bus heavy situation occurred.

Returns

True if a bus error situation was detected, false otherwise.

7.30.2.6 VectorXlReceptionThread()

```
static DWORD WINAPI VectorXlReceptionThread (
            LPVOID pv )  [static]
```

CAN event reception thread.

Parameters

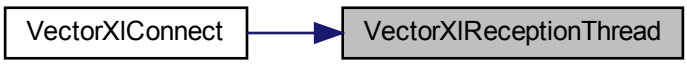
pv	Pointer to thread parameters.

Returns

Thread exit code.

Referenced by VectorXlConnect().

Here is the caller graph for this function:



7.30.2.7 VectorXlRegisterEvents()

```
static void VectorXlRegisterEvents (
            tCanEvents const * events )  [static]
```

Registers the event callback functions that should be called by the CAN interface.

Parameters

events	Pointer to structure with event callback function pointers.

7.30.2.8 VectorXlTransmit()

```
static bool VectorXlTransmit (
            tCanMsg const * msg )  [static]
```

Submits a message for transmission on the CAN bus.

Parameters

msg	Pointer to CAN message structure.

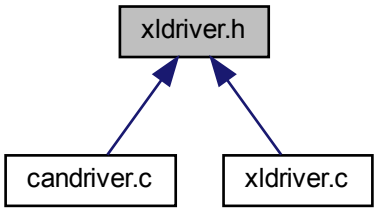
Returns

True if successful, false otherwise.

7.31 xldriver.h File Reference

Vector XL driver interface header file.

This graph shows which files directly or indirectly include this file:



Functions

- tCanInterface const * VectorXlGetInterface (void)
  Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

7.31.1 Detailed Description

Vector XL driver interface header file.

7.31.2 Function Documentation

7.31.2.1 VectorXlGetInterface()

```
tCanInterface const * VectorXlGetInterface (
            void  )
```

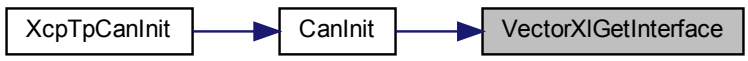
Obtains a pointer to the CAN interface structure, so that it can be linked to the generic CAN driver module.

Returns

Pointer to CAN interface structure.

Referenced by CanInit().

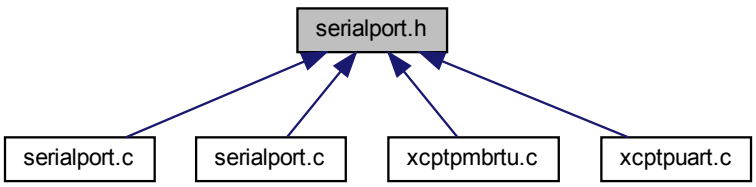
Here is the caller graph for this function:



7.32 serialport.h File Reference

Serial port header file.

This graph shows which files directly or indirectly include this file:



Enumerations

- enum tSerialPortBaudrate {
  SERIALPORT_BR9600 = 0 , SERIALPORT_BR19200 = 1 , SERIALPORT_BR38400 = 2 , SERIALPORT_BR57600 = 3 ,
  SERIALPORT_BR115200 = 4 }

  Enumeration of the supported baudrates.
- enum tSerialPortParity { SERIALPORT_PARITY_NONE = 0 , SERIALPORT_PARITY_ODD = 1 , SERIALPORT_PARITY_EVEN = 2 }

  Enumeration of the supported parities.
- enum tSerialPortStopbits { SERIALPORT_STOPBITS1 = 0 , SERIALPORT_STOPBITS2 = 1 }

  Enumeration of the supported stop bits.

Functions

- void **SerialPortInit** (void)

  Initializes the serial port module.
- void **SerialPortTerminate** (void)

  Terminates the serial port module.
- bool SerialPortOpen (char const *portname, tSerialPortBaudrate baudrate, tSerialPortParity parity, tSerialPortStopbits stopbits)

  Opens the connection with the serial port.
- void **SerialPortClose** (void)

  Closes the connection with the serial port.
- bool SerialPortWrite (uint8_t const *data, uint32_t length)

  Writes data to the serial port.
- bool SerialPortRead (uint8_t *data, uint32_t length)

  Reads data from the serial port in a blocking manner.

7.32.1 Detailed Description

Serial port header file.

7.32.2 Enumeration Type Documentation

7.32.2.1 tSerialPortBaudrate

```
enum tSerialPortBaudrate
```

Enumeration of the supported baudrates.

Enumerator

SERIALPORT_BR9600	9600 bits/sec
SERIALPORT_BR19200	19200 bits/sec
SERIALPORT_BR38400	38400 bits/sec
SERIALPORT_BR57600	57600 bits/sec
SERIALPORT_BR115200	115200 bits/sec

7.32.2.2 tSerialPortParity

`enum tSerialPortParity`

Enumeration of the supported parities.

Enumerator

SERIALPORT_PARITY_NONE	no parity bit
SERIALPORT_PARITY_ODD	odd parity bit
SERIALPORT_PARITY_EVEN	even parity bit

7.32.2.3 tSerialPortStopbits

`enum tSerialPortStopbits`

Enumeration of the supported stop bits.

Enumerator

SERIALPORT_STOPBITS1	1 stop bit
SERIALPORT_STOPBITS2	2 stop bits

7.32.3 Function Documentation

7.32.3.1 SerialPortOpen()

```
bool SerialPortOpen (
            char const * portname,
            tSerialPortBaudrate baudrate,
            tSerialPortParity parity,
            tSerialPortStopbits stopbits )
```

Opens the connection with the serial port.

Parameters

portname	The name of the serial port to open, i.e. /dev/ttyUSB0.
baudrate	The desired communication speed.
parity	The desired parity configuration.
stopbits	The desired stop bits configuration.

Returns

True if successful, false otherwise.

Parameters

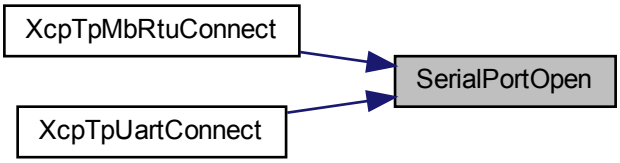
portname	The name of the serial port to open, i.e. COM4.
baudrate	The desired communication speed.
parity	The desired parity configuration.
stopbits	The desired stop bits configuration.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuConnect(), and XcpTpUartConnect().

Here is the caller graph for this function:



7.32.3.2 SerialPortRead()

```
bool SerialPortRead (
            uint8_t * data,
            uint32_t length )
```

Reads data from the serial port in a blocking manner.

Parameters

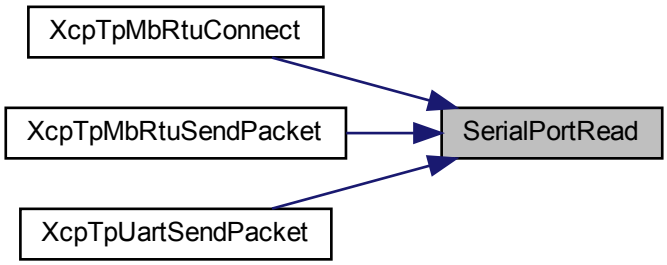
data	Pointer to byte array to store read data.
length	Number of bytes to read.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuConnect(), XcpTpMbRtuSendPacket(), and XcpTpUartSendPacket().

Here is the caller graph for this function:



7.32.3.3 SerialPortWrite()

```
bool SerialPortWrite (
            uint8_t const * data,
            uint32_t length )
```

Writes data to the serial port.

Parameters

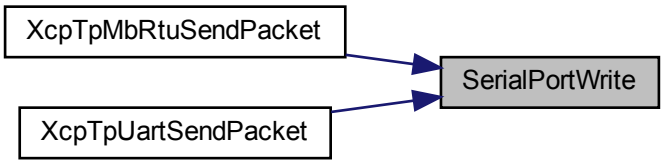
data	Pointer to byte array with data to write.
length	Number of bytes to write.

Returns

True if successful, false otherwise.

Referenced by XcpTpMbRtuSendPacket(), and XcpTpUartSendPacket().

Here is the caller graph for this function:

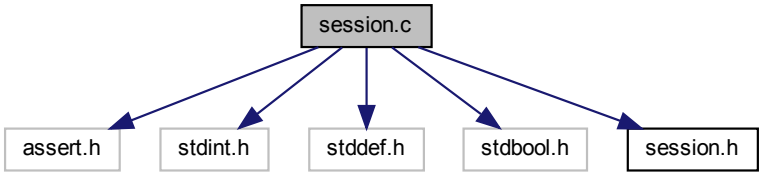


7.33 session.c File Reference

Communication session module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "session.h"
```

Include dependency graph for session.c:



Functions

- void SessionInit (tSessionProtocol const *protocol, void const *protocolSettings)

  Initializes the communication session module for the specified protocol.
- void **SessionTerminate** (void)

  Terminates the communication session module.
- bool SessionStart (void)

Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.

- void **SessionStop** (void)

  Stops the firmware update. This is where the bootloader starts the user program on the target if a valid one is present. After this the connection with the target is severed.

- bool SessionClearMemory (uint32_t address, uint32_t len)

  Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.

- bool SessionWriteData (uint32_t address, uint32_t len, uint8_t const ∗data)

  Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.

- bool SessionReadData (uint32_t address, uint32_t len, uint8_t ∗data)

  Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

Variables

- static tSessionProtocol const ∗ **protocolPtr**

  Pointer to the communication protocol that is linked.

7.33.1 Detailed Description

Communication session module source file.

7.33.2 Function Documentation

7.33.2.1 SessionClearMemory()

```
bool SessionClearMemory (
            uint32_t address,
            uint32_t len )
```

Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.

Parameters

address	The starting memory address for the erase operation.
len	The total number of bytes to erase from memory.

Returns

True if successful, false otherwise.

7.33.2.2 SessionInit()

```
void SessionInit (
            tSessionProtocol const * protocol,
            void const * protocolSettings )
```

Initializes the communication session module for the specified protocol.

Parameters

protocol	The session protocol module to link.
protocolSettings	Pointer to structure with protocol specific settings.

7.33.2.3 SessionReadData()

```
bool SessionReadData (
            uint32_t address,
            uint32_t len,
            uint8_t * data )
```

Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

Parameters

address	The starting memory address for the read operation.
len	The number of bytes to upload from the target and store in the data buffer.
data	Pointer to the byte array where the uploaded data should be stored.

Returns

True if successful, false otherwise.

7.33.2.4 SessionStart()

```
bool SessionStart (
            void  )
```

Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.

Returns

True if successful, false otherwise.

7.33.2.5 SessionWriteData()

```
bool SessionWriteData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.

Parameters

address	The starting memory address for the write operation.
len	The number of bytes in the data buffer that should be written.
data	Pointer to the byte array with data to write.

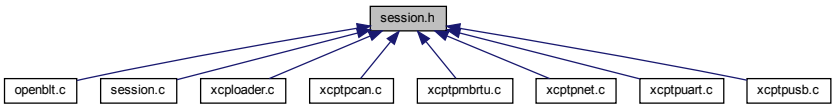
Returns

True if successful, false otherwise.

7.34 session.h File Reference

Communication session module header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tSessionProtocol

  Session communication protocol interface.

Functions

- void SessionInit (tSessionProtocol const ∗protocol, void const ∗protocolSettings)

  Initializes the communication session module for the specified protocol.
- void **SessionTerminate** (void)

  Terminates the communication session module.
- bool SessionStart (void)

  Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.
- void **SessionStop** (void)

Stops the firmware update. This is where the bootloader starts the user program on the target if a valid one is present. After this the connection with the target is severed.

- bool SessionClearMemory (uint32_t address, uint32_t len)

  Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.
- bool SessionWriteData (uint32_t address, uint32_t len, uint8_t const ∗data)

  Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.
- bool SessionReadData (uint32_t address, uint32_t len, uint8_t ∗data)

  Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

7.34.1 Detailed Description

Communication session module header file.

7.34.2 Function Documentation

7.34.2.1 SessionClearMemory()

```
bool SessionClearMemory (
            uint32_t address,
            uint32_t len )
```

Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.

Parameters

address	The starting memory address for the erase operation.
len	The total number of bytes to erase from memory.

Returns

True if successful, false otherwise.

7.34.2.2 SessionInit()

```
void SessionInit (
            tSessionProtocol const * protocol,
            void const * protocolSettings )
```

Initializes the communication session module for the specified protocol.

Parameters

protocol	The session protocol module to link.
protocolSettings	Pointer to structure with protocol specific settings.

7.34.2.3 SessionReadData()

```
bool SessionReadData (
            uint32_t address,
            uint32_t len,
            uint8_t * data )
```

Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

Parameters

address	The starting memory address for the read operation.
len	The number of bytes to upload from the target and store in the data buffer.
data	Pointer to the byte array where the uploaded data should be stored.

Returns

True if successful, false otherwise.

7.34.2.4 SessionStart()

```
bool SessionStart (
            void  )
```

Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.

Returns

True if successful, false otherwise.

7.34.2.5 SessionWriteData()

```
bool SessionWriteData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )
```

Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.

Parameters

address	The starting memory address for the write operation.
len	The number of bytes in the data buffer that should be written.
data	Pointer to the byte array with data to write.

Returns

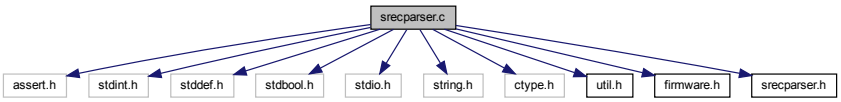
True if successful, false otherwise.

7.35 srecparser.c File Reference

Motorola S-record file parser source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "util.h"
#include "firmware.h"
#include "srecparser.h"
```

Include dependency graph for srecparser.c:



Enumerations

- enum tSRecParserLineType {
  SREC_PARSER_LINE_TYPE_S0 , SREC_PARSER_LINE_TYPE_S1 , SREC_PARSER_LINE_TYPE_S2 , SREC_PARSER_LINE_TYPE_S3 ,
  SREC_PARSER_LINE_TYPE_S7 , SREC_PARSER_LINE_TYPE_S8 , SREC_PARSER_LINE_TYPE_S9 , SREC_PARSER_LINE_TYPE_UNSUPPORTED }

  Enumeration for the different supported S-record line types.

Functions

- static bool SRecParserLoadFromFile (char const ∗firmwareFile, uint32_t addressOffset)

  Parses the specified firmware file to extract firmware data and adds this data to the firmware data that is currently managed by the firmware data module.
- static bool SRecParserVerifyFile (char const ∗firmwareFile)

  Parses the specified firmware file to verify that the file is a valid S-record file.
- static bool SRecParserSaveToFile (char const ∗firmwareFile)

Writes firmware data to the specified file in the correct file format.

- static bool SRecParserExtractLineData (char const ∗line, uint32_t ∗address, uint32_t ∗len, uint8_t ∗data)

  Checks if the specified S-record line is of the type that contains program data. If it does, then the program data and base address are extracted and stored at the function parameter pointers.
- static tSRecParserLineType SRecParserGetLineType (char const ∗line)

  Inspects a line from a Motorola S-Record file to determine its type.
- static bool SRecParserVerifyChecksum (char const ∗line)

  Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is corrrect.
- static bool SRecParserConstructLine (char ∗line, tSRecParserLineType lineType, uint32_t address, uint8_t const ∗data, uint8_t dataLen)

  Creates a NUL terminated S-record line, given the specified line type, address and data bytes. The checksum at the end of the line is also calculated and added.
- static uint8_t SRecParserHexStringToByte (char const ∗hexstring)

  Helper function to convert a sequence of 2 characters that represent a hexadecimal value to the actual byte value. Example: SRecParserHexStringToByte("2f") --> returns 47.
- tFirmwareParser const ∗ SRecParserGetParser (void)

  Obtains a pointer to the parser structure, so that it can be linked to the firmware data module.

Variables

- static const tFirmwareParser **srecParser**

  File parser structure filled with Motorola S-record parsing specifics.

7.35.1 Detailed Description

Motorola S-record file parser source file.

7.35.2 Enumeration Type Documentation

7.35.2.1 tSRecParserLineType

`enum tSRecParserLineType`

Enumeration for the different supported S-record line types.

Enumerator

SREC_PARSER_LINE_TYPE_S0	Header record.
SREC_PARSER_LINE_TYPE_S1	16-bit address data record.
SREC_PARSER_LINE_TYPE_S2	24-bit address data record.
SREC_PARSER_LINE_TYPE_S3	32-bit address data record.

Enumerator

SREC_PARSER_LINE_TYPE_S7	32-bit address termination.
SREC_PARSER_LINE_TYPE_S8	24-bit address termination.
SREC_PARSER_LINE_TYPE_S9	16-bit address termination.
SREC_PARSER_LINE_TYPE_UNSUPPORTED	Unsupported line.

7.35.3 Function Documentation

7.35.3.1 SRecParserConstructLine()

```
static bool SRecParserConstructLine (
            char * line,
            tSRecParserLineType lineType,
            uint32_t address,
            uint8_t const * data,
            uint8_t dataLen )  [static]
```

Creates a NUL terminated S-record line, given the specified line type, address and data bytes. The checksum at the end of the line is also calculated and added.

Parameters

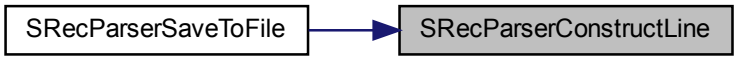
line	Pointer to character array where the string will be stored.
lineType	The type of S-record line to construct.
address	The address to embed into the line after the byte count.
data	Point to byte array with data bytes to add to the line.
dataLen	The number of data bytes present in the data-array.

Returns

True if successful, false otherwise.

Referenced by SRecParserSaveToFile().

Here is the caller graph for this function:



7.35.3.2 SRecParserExtractLineData()

```
static bool SRecParserExtractLineData (
            char const * line,
            uint32_t * address,
            uint32_t * len,
            uint8_t * data )  [static]
```

Checks if the specified S-record line is of the type that contains program data. If it does, then the program data and base address are extracted and stored at the function parameter pointers.

Parameters

line	Pointer to the line from an S-record file.
address	Pointer where the start address of the program data is stored.
len	Pointer for storing the number of extracted program data bytes.
data	Pointer to byte array where the extracted program data bytes are stored.

Returns

True if successful, false otherwise.

Referenced by SRecParserLoadFromFile().

Here is the caller graph for this function:

7.35.3.3 SRecParserGetLineType()

```
static tSRecParserLineType SRecParserGetLineType (
            char const * line )  [static]
```

Inspects a line from a Motorola S-Record file to determine its type.

Parameters

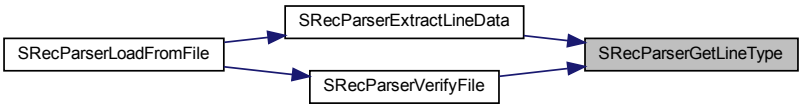
line	A line from the S-Record.

Returns

The S-Record line type.

Referenced by SRecParserExtractLineData(), and SRecParserVerifyFile().

Here is the caller graph for this function:



7.35.3.4 SRecParserGetParser()

```
tFirmwareParser const * SRecParserGetParser (
            void  )
```

Obtains a pointer to the parser structure, so that it can be linked to the firmware data module.

Returns

Pointer to firmware parser structure.

7.35.3.5 SRecParserHexStringToByte()

```
static uint8_t SRecParserHexStringToByte (
            char const * hexstring )  [static]
```

Helper function to convert a sequence of 2 characters that represent a hexadecimal value to the actual byte value. Example: SRecParserHexStringToByte("2f") --> returns 47.

Parameters

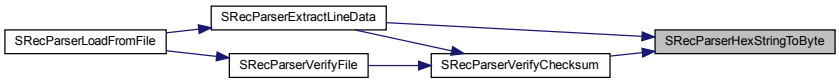
hexstring	String beginning with 2 characters that represent a hexa- decimal value.

Returns

The resulting byte value.

Referenced by SRecParserExtractLineData(), and SRecParserVerifyChecksum().

Here is the caller graph for this function:



7.35.3.6 SRecParserLoadFromFile()

```
static bool SRecParserLoadFromFile (
            char const * firmwareFile,
            uint32_t addressOffset )  [static]
```

Parses the specified firmware file to extract firmware data and adds this data to the firmware data that is currently managed by the firmware data module.

Parameters

firmwareFile	Filename of the firmware file to load.
addressOffset	Optional memory address offset to add when loading the firmware data from the file.

Returns

True if successful, false otherwise.

7.35.3.7 SRecParserSaveToFile()

```
static bool SRecParserSaveToFile (
            char const * firmwareFile )  [static]
```

Writes firmware data to the specified file in the correct file format.

Parameters

firmwareFile	Filename of the firmware file to write to.

Returns

True if successful, false otherwise.

7.35.3.8 SRecParserVerifyChecksum()

```
static bool SRecParserVerifyChecksum (
            char const * line )  [static]
```

Inspects an S1, S2 or S3 line from a Motorola S-Record file to determine if the checksum at the end is corrrect.

Parameters

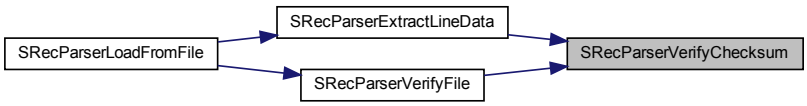
line	An S1, S2 or S3 line from the S-Record.

Returns

True if the checksum is correct, false otherwise.

Referenced by SRecParserExtractLineData(), and SRecParserVerifyFile().

Here is the caller graph for this function:



7.35.3.9 SRecParserVerifyFile()

```
static bool SRecParserVerifyFile (
            char const * firmwareFile )  [static]
```

Parses the specified firmware file to verify that the file is a valid S-record file.

Parameters

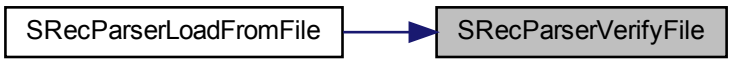
firmwareFile	Filename of the firmware file to verify.

Returns

True if successful, false otherwise.

Referenced by SRecParserLoadFromFile().

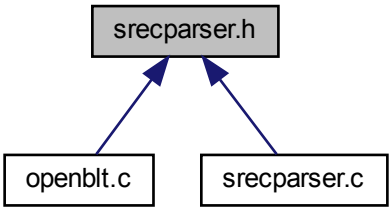
Here is the caller graph for this function:



7.36 srecparser.h File Reference

Motorola S-record file parser header file.

This graph shows which files directly or indirectly include this file:



Functions

- tFirmwareParser const * SRecParserGetParser (void)

  Obtains a pointer to the parser structure, so that it can be linked to the firmware data module.

7.36.1 Detailed Description

Motorola S-record file parser header file.

7.36.2 Function Documentation

7.36.2.1 SRecParserGetParser()

```
tFirmwareParser const * SRecParserGetParser (
            void  )
```

Obtains a pointer to the parser structure, so that it can be linked to the firmware data module.

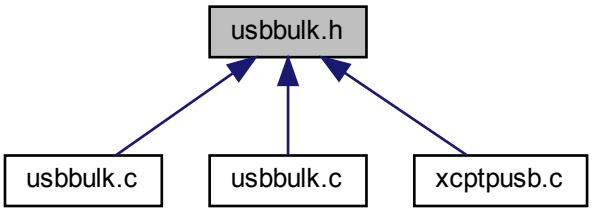
Returns

Pointer to firmware parser structure.

7.37 usbbulk.h File Reference

USB bulk driver header file.

This graph shows which files directly or indirectly include this file:



Functions

- void **UsbBulkInit** (void)
  Initializes the USB bulk driver.
- void **UsbBulkTerminate** (void)
  Terminates the USB bulk driver.
- bool UsbBulkOpen (void)
  Opens the connection with the USB device.
- void **UsbBulkClose** (void)
  Closes the connection with the USB device.
- bool UsbBulkWrite (uint8_t const ∗data, uint16_t length)
  Writes data to the USB device.
- bool UsbBulkRead (uint8_t ∗data, uint16_t length, uint32_t timeout)
  Reads data from the USB device.

7.37.1 Detailed Description

USB bulk driver header file.

7.37.2 Function Documentation

7.37.2.1 UsbBulkOpen()

```
bool UsbBulkOpen (
            void  )
```

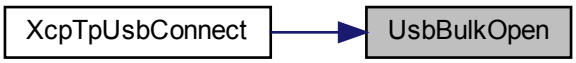
Opens the connection with the USB device.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbConnect().

Here is the caller graph for this function:



7.37.2.2 UsbBulkRead()

```
bool UsbBulkRead (
            uint8_t * data,
            uint16_t length,
            uint32_t timeout )
```

Reads data from the USB device.

Parameters

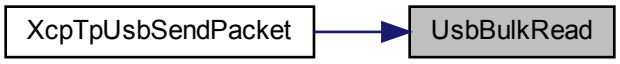
data	Pointer to byte array where received data should be stored.
length	Number of bytes to read from the USB device.
timeout	Timeout in milliseconds for the read operation.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbSendPacket().

Here is the caller graph for this function:



7.37.2.3 UsbBulkWrite()

```
bool UsbBulkWrite (
            uint8_t const * data,
            uint16_t length )
```

Writes data to the USB device.

Parameters

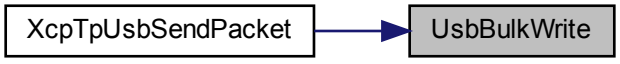
data	Pointer to byte array with data to write.
length	Number of bytes in the data array.

Returns

True if successful, false otherwise.

Referenced by XcpTpUsbSendPacket().

Here is the caller graph for this function:

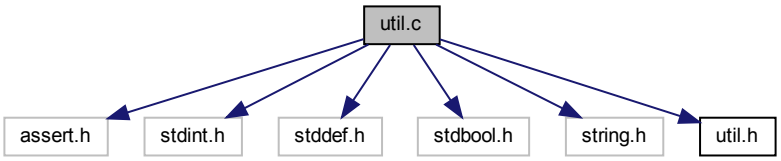


7.38 util.c File Reference

Utility module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <string.h>
#include "util.h"
#include "aes256.h"
```

Include dependency graph for util.c:



Functions

- uint16_t UtilChecksumCrc16Calculate (uint8_t const ∗data, uint32_t len)

  Calculates a 16-bit CRC value over the specified data using byte wise computation with a table.
- uint32_t UtilChecksumCrc32Calculate (uint8_t const ∗data, uint32_t len)

  Calculates a 32-bit CRC value over the specified data using byte wise computation with a table.
- bool UtilFileExtractFilename (char const ∗fullFilename, char ∗filenameBuffer)

  Extracts the filename including extention from the specified full filename, which could possible include a path. The function can handle both the backslash and forward slash path delimiter, to make it crossplatform.
- bool UtilCryptoAes256Encrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.
- bool UtilCryptoAes256Decrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Variables

- static const uint16_t **utilChecksumCrc16Table** [256]

  Lookup table for calculating a 16-bit CRC checksum. It was generated using an initial value of 0 and a polynomial of 0x8005.
- static const uint32_t **utilChecksumCrc32Table** [256]

  Lookup table for calculating a 32-bit CRC checksum. It was generated using an initial value of 0 and a polynomial of 0x04C11DB7.

7.38.1 Detailed Description

Utility module source file.

7.38.2 Function Documentation

7.38.2.1 UtilChecksumCrc16Calculate()

```
uint16_t UtilChecksumCrc16Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 16-bit CRC value over the specified data using byte wise computation with a table.

Parameters

data	Array with bytes over which the CRC16 should be calculated.
len	Number of bytes in the data array.

Returns

The 16-bit CRC value.

7.38.2.2 UtilChecksumCrc32Calculate()

```
uint32_t UtilChecksumCrc32Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 32-bit CRC value over the specified data using byte wise computation with a table.

Parameters

data	Array with bytes over which the CRC32 should be calculated.
len	Number of bytes in the data array.

Returns

The 32-bit CRC value.

7.38.2.3 UtilCryptoAes256Decrypt()

```
bool UtilCryptoAes256Decrypt (
            uint8_t * data,
```

```
            uint32_t len,
            uint8_t const * key )
```

Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to decrypt. The decrypted bytes are stored in the same array.
len	The number of bytes in the data-array to decrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit decryption key as a array of 32 bytes.

Returns

True if successful, false otherwise.

7.38.2.4 UtilCryptoAes256Encrypt()

```
bool UtilCryptoAes256Encrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to encrypt. The encrypted bytes are stored in the same array.
len	The number of bytes in the data-array to encrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit encryption key as a array of 32 bytes.

Returns

True if successful, false otherwise.

7.38.2.5 UtilFileExtractFilename()

```
bool UtilFileExtractFilename (
            char const * fullFilename,
            char * filenameBuffer )
```

Extracts the filename including extention from the specified full filename, which could possible include a path. The function can handle both the backslash and forward slash path delimiter, to make it crossplatform.

Parameters

fullFilename	The filename with path possible included.
filenameBuffer	Pointer to the character array where the resulting filename should be stored.

Returns

True if successful, false otherwise.

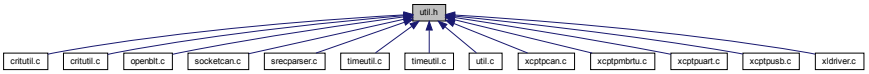
Referenced by SRecParserSaveToFile().

Here is the caller graph for this function:

7.39 util.h File Reference

Utility module header file.

This graph shows which files directly or indirectly include this file:



Functions

- uint16_t UtilChecksumCrc16Calculate (uint8_t const *data, uint32_t len)

  Calculates a 16-bit CRC value over the specified data using byte wise computation with a table.
- uint32_t UtilChecksumCrc32Calculate (uint8_t const *data, uint32_t len)

  Calculates a 32-bit CRC value over the specified data using byte wise computation with a table.
- bool UtilFileExtractFilename (char const *fullFilename, char *filenameBuffer)

  Extracts the filename including extention from the specified full filename, which could possible include a path. The function can handle both the backslash and forward slash path delimiter, to make it crossplatform.
- uint32_t UtilTimeGetSystemTimeMs (void)

  Get the system time in milliseconds.
- void UtilTimeDelayMs (uint16_t delay)

  Performs a delay of the specified amount of milliseconds.
- void **UtilCriticalSectionInit** (void)

Initializes the critical section module. Should be called before the Enter/Exit functions are used. It is okay to call this initialization multiple times from different modules.

- void **UtilCriticalSectionTerminate** (void)

  Terminates the critical section module. Should be called once critical sections are no longer needed. Typically called from another module's termination function that also initialized it. It is okay to call this termination multiple times from different modules.

- void **UtilCriticalSectionEnter** (void)

  Enters a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.

- void **UtilCriticalSectionExit** (void)

  Leaves a critical section. The functions UtilCriticalSectionEnter and UtilCriticalSectionExit should always be used in a pair.

- bool UtilCryptoAes256Encrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.

- bool UtilCryptoAes256Decrypt (uint8_t ∗data, uint32_t len, uint8_t const ∗key)

  Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

7.39.1 Detailed Description

Utility module header file.

7.39.2 Function Documentation

7.39.2.1 UtilChecksumCrc16Calculate()

```
uint16_t UtilChecksumCrc16Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 16-bit CRC value over the specified data using byte wise computation with a table.

Parameters

data	Array with bytes over which the CRC16 should be calculated.
len	Number of bytes in the data array.

Returns

The 16-bit CRC value.

7.39.2.2 UtilChecksumCrc32Calculate()

```
uint32_t UtilChecksumCrc32Calculate (
            uint8_t const * data,
            uint32_t len )
```

Calculates a 32-bit CRC value over the specified data using byte wise computation with a table.

Parameters

data	Array with bytes over which the CRC32 should be calculated.
len	Number of bytes in the data array.

Returns

The 32-bit CRC value.

7.39.2.3 UtilCryptoAes256Decrypt()

```
bool UtilCryptoAes256Decrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Decrypts the len-bytes in the specified data-array, using the specified 256- bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to decrypt. The decrypted bytes are stored in the same array.
len	The number of bytes in the data-array to decrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit decryption key as a array of 32 bytes.

Returns

True if successful, false otherwise.

7.39.2.4 UtilCryptoAes256Encrypt()

```
bool UtilCryptoAes256Encrypt (
            uint8_t * data,
            uint32_t len,
            uint8_t const * key )
```

Encrypts the len-bytes in the specified data-array, using the specified 256-bit (32 bytes) key. The results are written back into the same array.

Parameters

data	Pointer to the byte array with data to encrypt. The encrypted bytes are stored in the same array.
len	The number of bytes in the data-array to encrypt. It must be a multiple of 16, as this is the AES256 minimal block size.
key	The 256-bit encryption key as a array of 32 bytes.

Returns

True if successful, false otherwise.

7.39.2.5 UtilFileExtractFilename()

```
bool UtilFileExtractFilename (
            char const * fullFilename,
            char * filenameBuffer )
```

Extracts the filename including extention from the specified full filename, which could possible include a path. The function can handle both the backslash and forward slash path delimiter, to make it crossplatform.

Parameters

fullFilename	The filename with path possible included.
filenameBuffer	Pointer to the character array where the resulting filename should be stored.

Returns

True if successful, false otherwise.

Referenced by SRecParserSaveToFile().

Here is the caller graph for this function:

7.39.2.6 UtilTimeDelayMs()

```
void UtilTimeDelayMs (
            uint16_t delay )
```

Performs a delay of the specified amount of milliseconds.

Parameters

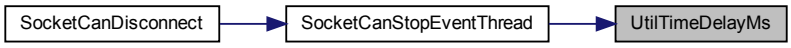
delay	Delay time in milliseconds.
delay	Delay time in milliseconds.

Returns

none.

Referenced by SocketCanStopEventThread().

Here is the caller graph for this function:



7.39.2.7 UtilTimeGetSystemTimeMs()

```
uint32_t UtilTimeGetSystemTimeMs (
            void  )
```

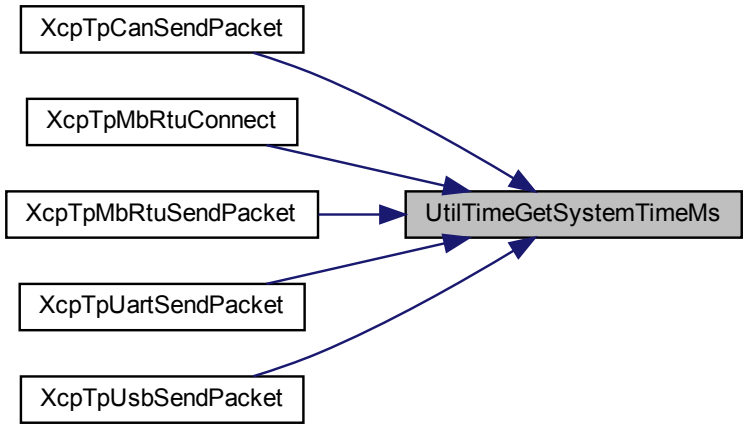
Get the system time in milliseconds.

Returns

Time in milliseconds.

Referenced by XcpTpCanSendPacket(), XcpTpMbRtuConnect(), XcpTpMbRtuSendPacket(), XcpTpUartSendPacket(), and XcpTpUsbSendPacket().

Here is the caller graph for this function:

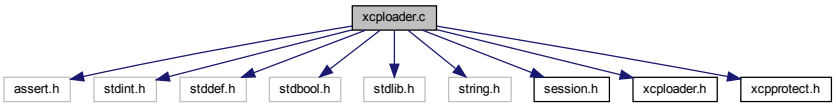


7.40 xcploader.c File Reference

XCP Loader module source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcpprotect.h"
```

Include dependency graph for xcploader.c:



Macros

- #define XCPLOADER_CMD_CONNECT (0xFFu)
- #define XCPLOADER_CMD_GET_STATUS (0xFDu)
- #define XCPLOADER_CMD_GET_SEED (0xF8u)
- #define XCPLOADER_CMD_UNLOCK (0xF7u)
- #define XCPLOADER_CMD_SET_MTA (0xF6u)
- #define XCPLOADER_CMD_UPLOAD (0xF5u)
- #define XCPLOADER_CMD_PROGRAM_START (0xD2u)
- #define XCPLOADER_CMD_PROGRAM_CLEAR (0xD1u)
- #define XCPLOADER_CMD_PROGRAM (0xD0u)
- #define XCPLOADER_CMD_PROGRAM_RESET (0xCFu)
- #define XCPLOADER_CMD_PROGRAM_MAX (0xC9u)
- #define XCPLOADER_CMD_PID_RES (0xFFu)
- #define **XCPLOADER_CONNECT_RETRIES** (5u)

  Number of retries to connect to the XCP slave.

Functions

- static void XcpLoaderInit (void const *settings)

  Initializes the protocol module.
- static void **XcpLoaderTerminate** (void)

  Terminates the protocol module.
- static bool XcpLoaderStart (void)

  Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.
- static void **XcpLoaderStop** (void)

  Stops the firmware update. This is where the bootloader starts the user program on the target if a valid one is present. After this the connection with the target is severed.

- static bool XcpLoaderClearMemory (uint32_t address, uint32_t len)

  Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.
- static bool XcpLoaderWriteData (uint32_t address, uint32_t len, uint8_t const *data)

  Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.
- static bool XcpLoaderReadData (uint32_t address, uint32_t len, uint8_t *data)

  Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.
- static void XcpLoaderSetOrderedLong (uint32_t value, uint8_t *data)

  Stores a 32-bit value into a byte buffer taking into account Intel or Motorola byte ordering.
- static uint16_t XcpLoaderGetOrderedWord (uint8_t const *data)

  Obtains a 16-bit value from a byte buffer taking into account Intel or Motorola byte ordering.
- static bool XcpLoaderSendCmdConnect (void)

  Sends the XCP Connect command.
- static bool XcpLoaderSendCmdGetStatus (uint8_t *session, uint8_t *protectedResources, uint16_t *config↩ Id)

  Sends the XCP Get Status command. Note that it is okay to specify a NULL value for the parameters if you are not interested in a particular one.
- static bool XcpLoaderSendCmdGetSeed (uint8_t resource, uint8_t mode, uint8_t *seed, uint8_t *seedLen)

  Sends the XCP Get Seed command.
- static bool XcpLoaderSendCmdUnlock (uint8_t const *key, uint8_t keyLen, uint8_t *protectedResources)

  Sends the XCP Unlock command.
- static bool XcpLoaderSendCmdSetMta (uint32_t address)

  Sends the XCP Set MTA command.
- static bool XcpLoaderSendCmdUpload (uint8_t *data, uint8_t length)

  Sends the XCP UPLOAD command.
- static bool XcpLoaderSendCmdProgramStart (void)

  Sends the XCP PROGRAM START command.
- static bool XcpLoaderSendCmdProgramReset (void)

  Sends the XCP PROGRAM RESET command. Note that this command is a bit different as in it does not require a response.
- static bool XcpLoaderSendCmdProgram (uint8_t length, uint8_t const *data)

  Sends the XCP PROGRAM command.
- static bool XcpLoaderSendCmdProgramMax (uint8_t const *data)

  Sends the XCP PROGRAM MAX command.
- static bool XcpLoaderSendCmdProgramClear (uint32_t length)

  Sends the XCP PROGRAM CLEAR command.
- tSessionProtocol const * XcpLoaderGetProtocol (void)

  Obtains a pointer to the protocol structure, so that it can be linked to the communication session module.

Variables

- static const tSessionProtocol **xcpLoader**

  Protocol structure filled with XCP loader specifics.
- static tXcpLoaderSettings **xcpSettings**

  The settings that should be used by the XCP loader.
- static bool **xcpConnected**

  Flag to keep track of the connection status.
- static bool **xcpSlaveIsIntel**

  Store the byte ordering of the XCP slave.

- static uint8_t **xcpMaxCto**

  The max number of bytes in the command transmit object (master->slave).
- static uint8_t **xcpMaxProgCto**

  The max number of bytes in the command transmit object (master->slave) during a programming session.
- static uint16_t **xcpMaxDto**

  The max number of bytes in the data transmit object (slave->master).

7.40.1 Detailed Description

XCP Loader module source file.

7.40.2 Macro Definition Documentation

7.40.2.1 XCPLOADER_CMD_CONNECT

```
#define XCPLOADER_CMD_CONNECT (0xFFu)
```

XCP connect command code.

7.40.2.2 XCPLOADER_CMD_GET_SEED

```
#define XCPLOADER_CMD_GET_SEED (0xF8u)
```

XCP get seed command code.

7.40.2.3 XCPLOADER_CMD_GET_STATUS

```
#define XCPLOADER_CMD_GET_STATUS (0xFDu)
```

XCP get status command code.

7.40.2.4 XCPLOADER_CMD_PID_RES

```
#define XCPLOADER_CMD_PID_RES (0xFFu)
```

positive response

7.40.2.5 XCPLOADER_CMD_PROGRAM

```
#define XCPLOADER_CMD_PROGRAM (0xD0u)
```

XCP program command code.

7.40.2.6 XCPLOADER_CMD_PROGRAM_CLEAR

```
#define XCPLOADER_CMD_PROGRAM_CLEAR (0xD1u)
```

XCP program clear command code.

7.40.2.7 XCPLOADER_CMD_PROGRAM_MAX

```
#define XCPLOADER_CMD_PROGRAM_MAX (0xC9u)
```

XCP program max command code.

7.40.2.8 XCPLOADER_CMD_PROGRAM_RESET

```
#define XCPLOADER_CMD_PROGRAM_RESET (0xCFu)
```

XCP program reset command code.

7.40.2.9 XCPLOADER_CMD_PROGRAM_START

```
#define XCPLOADER_CMD_PROGRAM_START (0xD2u)
```

XCP program start command code.

7.40.2.10 XCPLOADER_CMD_SET_MTA

```
#define XCPLOADER_CMD_SET_MTA (0xF6u)
```

XCP set mta command code.

7.40.2.11 XCPLOADER_CMD_UNLOCK

```
#define XCPLOADER_CMD_UNLOCK (0xF7u)
```

XCP unlock command code.

7.40.2.12 XCPLOADER_CMD_UPLOAD

```
#define XCPLOADER_CMD_UPLOAD (0xF5u)
```

XCP upload command code.

7.40.3 Function Documentation

7.40.3.1 XcpLoaderClearMemory()

```
static bool XcpLoaderClearMemory (
            uint32_t address,
            uint32_t len )  [static]
```

Requests the bootloader to erase the specified range of memory on the target. The bootloader aligns this range to hardware specified erase blocks.

Parameters

address	The starting memory address for the erase operation.
len	The total number of bytes to erase from memory.

Returns

True if successful, false otherwise.

7.40.3.2 XcpLoaderGetOrderedWord()

```
static uint16_t XcpLoaderGetOrderedWord (
            uint8_t const * data )  [static]
```

Obtains a 16-bit value from a byte buffer taking into account Intel or Motorola byte ordering.

Parameters

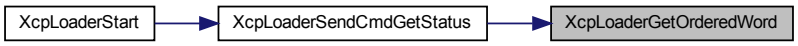
data	Array to the buffer with the word value stored as bytes.

Returns

The 16-bit value.

Referenced by XcpLoaderSendCmdGetStatus().

Here is the caller graph for this function:



7.40.3.3 XcpLoaderGetProtocol()

```
tSessionProtocol const * XcpLoaderGetProtocol (
            void  )
```

Obtains a pointer to the protocol structure, so that it can be linked to the communication session module.

Returns

Pointer to protocol structure.

7.40.3.4 XcpLoaderInit()

```
static void XcpLoaderInit (
            void const * settings )  [static]
```

Initializes the protocol module.

Parameters

settings	Pointer to the structure with protocol settings.

7.40.3.5 XcpLoaderReadData()

```
static bool XcpLoaderReadData (
            uint32_t address,
            uint32_t len,
            uint8_t * data )  [static]
```

Request the bootloader to upload the specified range of memory. The data is stored in the data byte array to which the pointer was specified.

Parameters

address	The starting memory address for the read operation.
len	The number of bytes to upload from the target and store in the data buffer.
data	Pointer to the byte array where the uploaded data should be stored.

Returns

True if successful, false otherwise.

7.40.3.6 XcpLoaderSendCmdConnect()

```
static bool XcpLoaderSendCmdConnect (
            void  )  [static]
```

Sends the XCP Connect command.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:

7.40.3.7 XcpLoaderSendCmdGetSeed()

```
static bool XcpLoaderSendCmdGetSeed (
            uint8_t resource,
            uint8_t mode,
            uint8_t * seed,
            uint8_t * seedLen )  [static]
```

Sends the XCP Get Seed command.

Parameters

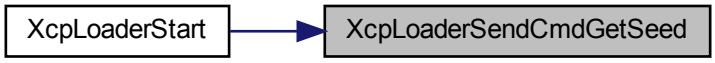
resource	The resource to unlock (XCPPROTECT_RESOURCE_xxx).
mode	0 for the first part of the seed, 1 for the remaining part.
seed	Pointer to byte array where the received seed is stored.
seedLen	Length of the seed in bytes.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:



7.40.3.8 XcpLoaderSendCmdGetStatus()

```
static bool XcpLoaderSendCmdGetStatus (
            uint8_t * session,
            uint8_t * protectedResources,
            uint16_t * configId )  [static]
```

Sends the XCP Get Status command. Note that it is okay to specify a NULL value for the parameters if you are not interested in a particular one.

Parameters

session	Current session status.
protectedResources	Current resource protection status.
configId	Session configuration identifier.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:

7.40.3.9 XcpLoaderSendCmdProgram()

```
static bool XcpLoaderSendCmdProgram (
            uint8_t length,
            uint8_t const * data )  [static]
```

Sends the XCP PROGRAM command.

Parameters

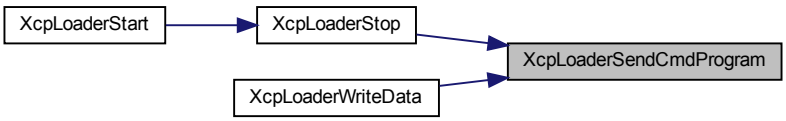
length	Number of bytes in the data array to program.
data	Array with data bytes to program.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStop(), and XcpLoaderWriteData().

Here is the caller graph for this function:



7.40.3.10 XcpLoaderSendCmdProgramClear()

```
static bool XcpLoaderSendCmdProgramClear (
            uint32_t length )  [static]
```

Sends the XCP PROGRAM CLEAR command.

Parameters

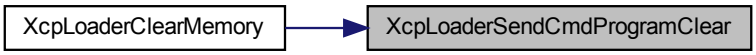
length	Number of elements to clear starting at the MTA address.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderClearMemory().

Here is the caller graph for this function:



7.40.3.11 XcpLoaderSendCmdProgramMax()

```
static bool XcpLoaderSendCmdProgramMax (
            uint8_t const * data ) [static]
```

Sends the XCP PROGRAM MAX command.

Parameters

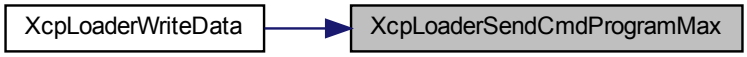
data	Array with data bytes to program.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderWriteData().

Here is the caller graph for this function:



7.40.3.12 XcpLoaderSendCmdProgramReset()

```
static bool XcpLoaderSendCmdProgramReset (
            void  )  [static]
```

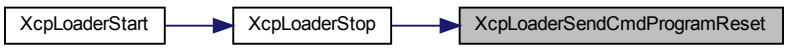
Sends the XCP PROGRAM RESET command. Note that this command is a bit different as in it does not require a response.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStop().

Here is the caller graph for this function:



7.40.3.13 XcpLoaderSendCmdProgramStart()

```
static bool XcpLoaderSendCmdProgramStart (
            void  )  [static]
```

Sends the XCP PROGRAM START command.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:

7.40.3.14 XcpLoaderSendCmdSetMta()

```
static bool XcpLoaderSendCmdSetMta (
            uint32_t address )  [static]
```

Sends the XCP Set MTA command.

Parameters

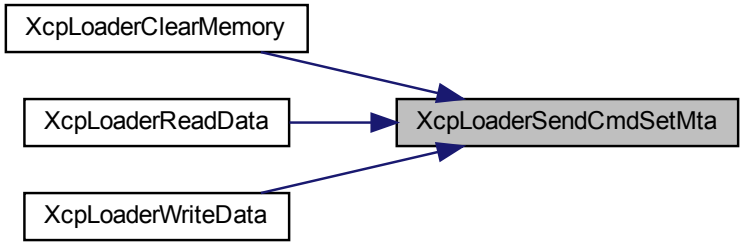
address	New MTA address for the slave.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderClearMemory(), XcpLoaderReadData(), and XcpLoaderWriteData().

Here is the caller graph for this function:



7.40.3.15 XcpLoaderSendCmdUnlock()

```
static bool XcpLoaderSendCmdUnlock (
            uint8_t const * key,
            uint8_t keyLen,
            uint8_t * protectedResources )  [static]
```

Sends the XCP Unlock command.

Parameters

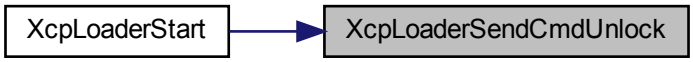
key	Pointer to a byte array containing the key.
keyLen	The length of the key in bytes.
protectedResources	Current resource protection status.

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:



7.40.3.16 XcpLoaderSendCmdUpload()

```
static bool XcpLoaderSendCmdUpload (
            uint8_t * data,
            uint8_t length )  [static]
```

Sends the XCP UPLOAD command.

Parameters

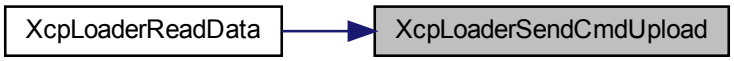
data	Destination data buffer.
length	Number of bytes to upload.

Returns

SB_TRUE is successfull, SB_FALSE otherwise.

Referenced by XcpLoaderReadData().

Here is the caller graph for this function:



7.40.3.17 XcpLoaderSetOrderedLong()

```
static void XcpLoaderSetOrderedLong (
            uint32_t value,
            uint8_t * data )  [static]
```

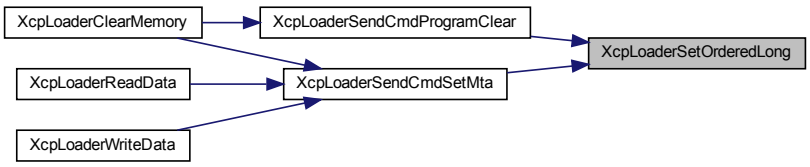
Stores a 32-bit value into a byte buffer taking into account Intel or Motorola byte ordering.

Parameters

value	The 32-bit value to store in the buffer.
data	Array to the buffer for storage.

Referenced by XcpLoaderSendCmdProgramClear(), and XcpLoaderSendCmdSetMta().

Here is the caller graph for this function:



7.40.3.18 XcpLoaderStart()

```
static bool XcpLoaderStart (
            void  )  [static]
```

Starts the firmware update session. This is where the connection with the target is made and the bootloader on the target is activated.

Returns

True if successful, false otherwise.

7.40.3.19 XcpLoaderWriteData()

```
static bool XcpLoaderWriteData (
            uint32_t address,
            uint32_t len,
            uint8_t const * data )  [static]
```

Requests the bootloader to program the specified data to memory. In case of non-volatile memory, the application needs to make sure the memory range was erased beforehand.

Parameters

address	The starting memory address for the write operation.
len	The number of bytes in the data buffer that should be written.
data	Pointer to the byte array with data to write.

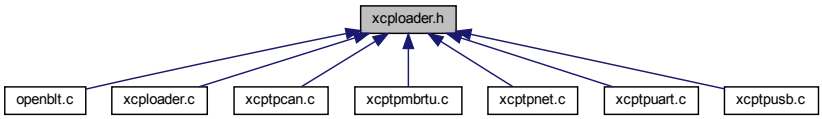
Returns

True if successful, false otherwise.

7.41 xcploader.h File Reference

XCP Loader module header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tXcpTransportPacket

  XCP transport layer packet type.
- struct tXcpTransport

  XCP transport layer.
- struct tXcpLoaderSettings

  XCP protocol specific settings.

Macros

- #define **XCPLOADER_PACKET_SIZE_MAX** (255u)

  Total number of bytes in a master<->slave data packet. It should be at least equal or larger than that configured on the slave.

Functions

- tSessionProtocol const * XcpLoaderGetProtocol (void)

  Obtains a pointer to the protocol structure, so that it can be linked to the communication session module.

7.41.1 Detailed Description

XCP Loader module header file.

7.41.2 Function Documentation

7.41.2.1 XcpLoaderGetProtocol()

```
tSessionProtocol const * XcpLoaderGetProtocol (
            void  )
```

Obtains a pointer to the protocol structure, so that it can be linked to the communication session module.

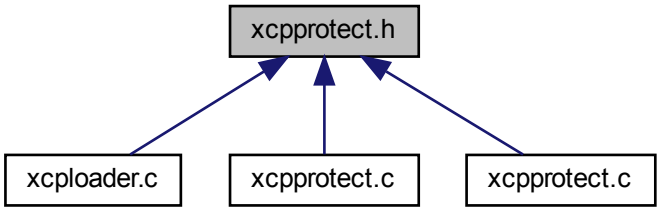
Returns

Pointer to protocol structure.

7.42 xcpprotect.h File Reference

XCP Protection module header file.

This graph shows which files directly or indirectly include this file:



Macros

- #define XCPPROTECT_RESOURCE_PGM (0x10u)
- #define XCPPROTECT_RESOURCE_STIM (0x08u)
- #define XCPPROTECT_RESOURCE_DAQ (0x04u)
- #define XCPPROTECT_RESOURCE_CALPAG (0x01u)

Functions

- void XcpProtectInit (char const ∗seedKeyFile)

  Initializes the XCP protection module.
- void **XcpProtectTerminate** (void)

  Terminates the XCP protection module.
- bool XCPProtectComputeKeyFromSeed (uint8_t resource, uint8_t seedLen, uint8_t const ∗seedPtr, uint8_t ∗keyLenPtr, uint8_t ∗keyPtr)

  Computes the key for the requested resource.
- bool XcpProtectGetPrivileges (uint8_t ∗resourcePtr)

  Obtains a bitmask of the resources for which an key algorithm is available.

7.42.1 Detailed Description

XCP Protection module header file.

7.42.2 Macro Definition Documentation

7.42.2.1 XCPPROTECT_RESOURCE_CALPAG

```
#define XCPPROTECT_RESOURCE_CALPAG (0x01u)
```

CALibration and PAGing resource.

7.42.2.2 XCPPROTECT_RESOURCE_DAQ

```
#define XCPPROTECT_RESOURCE_DAQ (0x04u)
```

Data AcQuisition resource.

7.42.2.3 XCPPROTECT_RESOURCE_PGM

```
#define XCPPROTECT_RESOURCE_PGM (0x10u)
```

ProGraMing resource.

7.42.2.4 XCPPROTECT_RESOURCE_STIM

```
#define XCPPROTECT_RESOURCE_STIM (0x08u)
```

data STIMulation resource.

7.42.3 Function Documentation

7.42.3.1 XCPProtectComputeKeyFromSeed()

```
bool XCPProtectComputeKeyFromSeed (
            uint8_t resource,
            uint8_t seedLen,
            uint8_t const * seedPtr,
            uint8_t * keyLenPtr,
            uint8_t * keyPtr )
```

Computes the key for the requested resource.

Parameters

resource	resource for which the unlock key is requested
seedLen	length of the seed
seedPtr	pointer to the seed data
keyLenPtr	pointer where to store the key length
keyPtr	pointer where to store the key data

Returns

True if successful, false otherwise.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:

7.42.3.2 XcpProtectGetPrivileges()

```
bool XcpProtectGetPrivileges (
            uint8_t * resourcePtr )
```

Obtains a bitmask of the resources for which an key algorithm is available.

Parameters

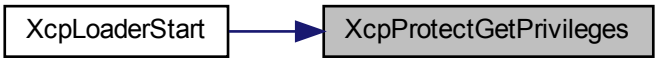
resourcePtr	pointer where to store the supported resources for the key computation.

Returns

XCP_RESULT_OK on success, otherwise XCP_RESULT_ERROR.

Referenced by XcpLoaderStart().

Here is the caller graph for this function:



7.42.3.3 XcpProtectInit()

```
void XcpProtectInit (
            char const * seedKeyFile )
```

Initializes the XCP protection module.

Parameters

seedKeyFile	Filename of the seed and key shared library that contains the following functions: • XCP_ComputeKeyFromSeed() • XCP_GetAvailablePrivileges()

Referenced by XcpLoaderInit().

Here is the caller graph for this function:

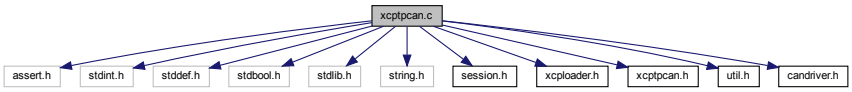
7.43 xcptpcan.c File Reference

XCP CAN transport layer source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
```

```
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcptpcan.h"
#include "util.h"
#include "candriver.h"
```

Include dependency graph for xcptpcan.c:



Functions

- static void XcpTpCanInit (void const *settings)

  Initializes the transport layer.
- static void **XcpTpCanTerminate** (void)

  Terminates the transport layer.
- static bool XcpTpCanConnect (void)

  Connects to the transport layer.
- static void **XcpTpCanDisconnect** (void)

  Disconnects from the transport layer.
- static bool XcpTpCanSendPacket (tXcpTransportPacket const *txPacket, tXcpTransportPacket *rxPacket, uint16_t timeout)

  Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.
- static void XcpTpCanEventMessageTransmitted (tCanMsg const *msg)

  CAN driver event callback function that gets called each time a CAN message was transmitted.
- static void XcpTpCanEventMessageReceived (tCanMsg const *msg)

  CAN driver event callback function that gets called each time a CAN message was received.
- tXcpTransport const * XcpTpCanGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Variables

- static const tXcpTransport **canTransport**

  XCP transport layer structure filled with CAN specifics.
- static const tCanEvents **canEvents**

  CAN driver event functions.
- static tXcpTpCanSettings **tpCanSettings**

  The settings to use in this transport layer.
- static volatile bool **tpCanResponseMessageReceived**

  Flag to indicate that a response packet was received via CAN. Made volatile because it is shared with an event callback function that could be called from a different thread.
- static volatile tCanMsg **tpCanResponseMessage**

  Buffer for storing the CAN message with response packet data. Made volatile because it is shared with an event callback function that could be called from a different thread.

7.43.1 Detailed Description

XCP CAN transport layer source file.

7.43.2 Function Documentation

7.43.2.1 XcpTpCanConnect()

```
static bool XcpTpCanConnect (
            void  )  [static]
```

Connects to the transport layer.

Returns

True is connected, false otherwise.

7.43.2.2 XcpTpCanEventMessageReceived()

```
static void XcpTpCanEventMessageReceived (
            tCanMsg const * msg )  [static]
```

CAN driver event callback function that gets called each time a CAN message was received.

Parameters

msg	Pointer to the received CAN message.

7.43.2.3 XcpTpCanEventMessageTransmitted()

```
static void XcpTpCanEventMessageTransmitted (
            tCanMsg const * msg )  [static]
```

CAN driver event callback function that gets called each time a CAN message was transmitted.

Parameters

msg	Pointer to the transmitted CAN message.

7.43.2.4 XcpTpCanGetTransport()

```
tXcpTransport const * XcpTpCanGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

7.43.2.5 XcpTpCanInit()

```
static void XcpTpCanInit (
            void const * settings )  [static]
```

Initializes the transport layer.

Parameters

settings	Pointer to settings structure.

Returns

None.

7.43.2.6 XcpTpCanSendPacket()

```
static bool XcpTpCanSendPacket (
            tXcpTransportPacket const * txPacket,
            tXcpTransportPacket * rxPacket,
            uint16_t timeout )  [static]
```

Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

Parameters

txPacket	Pointer to the packet to transmit.
rxPacket	Pointer where the received packet info is stored.
timeout	Maximum time in milliseconds to wait for the reception of the response packet.

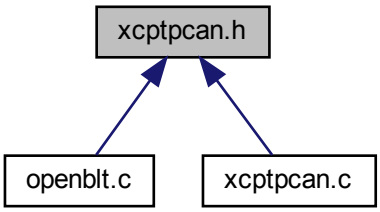
Returns

True is successful and a response packet was received, false otherwise.

7.44 xcptpcan.h File Reference

XCP CAN transport layer header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tXcpTpCanSettings

  Layout of structure with settings specific to the XCP transport layer module for CAN.

Functions

- tXcpTransport const * XcpTpCanGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

7.44.1 Detailed Description

XCP CAN transport layer header file.

7.44.2 Function Documentation

7.44.2.1 XcpTpCanGetTransport()

```
tXcpTransport const * XcpTpCanGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

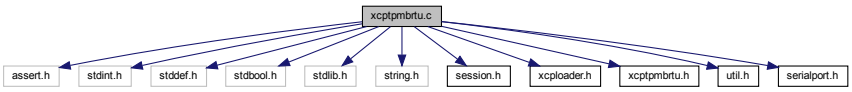
Pointer to transport layer structure.

7.45 xcptpmbrtu.c File Reference

XCP Modbus RTU transport layer source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcptpmbrtu.h"
#include "util.h"
#include "serialport.h"
```

Include dependency graph for xcptpmbrtu.c:



Macros

- #define **XCP_TP_MBRTU_FCT_CODE_USER_XCP** (109u)

  This module embeds the bootloader's XCP communication packets inside Modbus RTU communication packets. As this is a non-standard functionality of Modbus, a user-defined function code is used to extend the Modbus functionality for the purpose of embedded XCP packets. User-defined function codes are allowed, as long as they are in the range 65..72 or 100..110. By default, the user- defined function code 109 is selected. Note that some other Modbus device on the network might also already use this. In this case set this macro to another unique user-defined function code.

Functions

- static void XcpTpMbRtuInit (void const ∗settings)

  Initializes the transport layer.
- static void **XcpTpMbRtuTerminate** (void)

  Terminates the transport layer.
- static bool XcpTpMbRtuConnect (void)

Connects to the transport layer.

- static void **XcpTpMbRtuDisconnect** (void)

  Disconnects from the transport layer.

- static bool XcpTpMbRtuSendPacket (tXcpTransportPacket const ∗txPacket, tXcpTransportPacket ∗rxPacket, uint16_t timeout)

  Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

- static uint16_t XcpTpMbRtuCrcCalculate (uint8_t const ∗data, uint16_t len)

  Calculates the 16-bit CRC according to the Modbus RTU specification.

- tXcpTransport const ∗ XcpTpMbRtuGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Variables

- static const tXcpTransport **mbRtuTransport**

  XCP transport layer structure filled with Modbus RTU specifics.

- static tXcpTpMbRtuSettings **tpMbRtuSettings**

  The settings to use in this transport layer.

- static uint32_t **tpMbRtuT3_5Ms** = 2 + 1

  Modbus RTU 3.5 character time in milliseconds. In the baudrate is > 19200 bits/sec the T3_5 can be fixed at 1750 us. This is 2 ms when rounded to milliseconds. Use this as the initial value and adjust later on in case of a baudrate <= 19200 bps. Make sure to add 1 ms to adjust for millisecond resolution inaccuracy.

- static uint8_t **CRCHi** [ ]

  Table of CRC values for high-order byte.

- static uint8_t **CRCLo** [ ]

  Table of CRC values for low-order byte.

7.45.1 Detailed Description

XCP Modbus RTU transport layer source file.

7.45.2 Function Documentation

7.45.2.1 XcpTpMbRtuConnect()

```
static bool XcpTpMbRtuConnect (
            void  )  [static]
```

Connects to the transport layer.

Returns

True is connected, false otherwise.

7.45.2.2 XcpTpMbRtuCrcCalculate()

```
static uint16_t XcpTpMbRtuCrcCalculate (
            uint8_t const * data,
            uint16_t len )  [static]
```

Calculates the 16-bit CRC according to the Modbus RTU specification.

Parameters

data	Pointer to a byte array with values of which to compute the CRC.
len	Number of bytes in the data array.

Returns

the 16-bit CRC value.

Referenced by XcpTpMbRtuSendPacket().

Here is the caller graph for this function:

7.45.2.3 XcpTpMbRtuGetTransport()

```
tXcpTransport const * XcpTpMbRtuGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

7.45.2.4 XcpTpMbRtuInit()

```
static void XcpTpMbRtuInit (
            void const * settings )  [static]
```

Initializes the transport layer.

Parameters

settings	Pointer to settings structure.

Returns

None.

7.45.2.5 XcpTpMbRtuSendPacket()

```
static bool XcpTpMbRtuSendPacket (
            tXcpTransportPacket const * txPacket,
            tXcpTransportPacket * rxPacket,
            uint16_t timeout )  [static]
```

Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

Parameters

txPacket	Pointer to the packet to transmit.
rxPacket	Pointer where the received packet info is stored.
timeout	Maximum time in milliseconds to wait for the reception of the response packet.

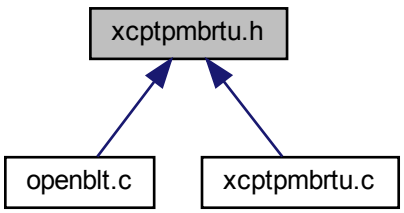
Returns

True is successful and a response packet was received, false otherwise.

7.46 xcptpmbrtu.h File Reference

XCP Modbus RTU transport layer header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tXcpTpMbRtuSettings

  Layout of structure with settings specific to the XCP transport layer module for Modbus RTU.

Functions

- tXcpTransport const * XcpTpMbRtuGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

7.46.1 Detailed Description

XCP Modbus RTU transport layer header file.

7.46.2 Function Documentation

7.46.2.1 XcpTpMbRtuGetTransport()

```
tXcpTransport const * XcpTpMbRtuGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

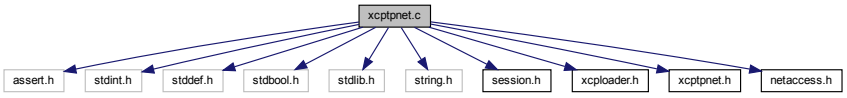
Pointer to transport layer structure.

7.47 xcptpnet.c File Reference

XCP TCP/IP transport layer source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcptpnet.h"
#include "netaccess.h"
```

Include dependency graph for xcptpnet.c:



Functions

- static void XcpTpNetInit (void const ∗settings)

  Initializes the transport layer.
- static void **XcpTpNetTerminate** (void)

  Terminates the transport layer.
- static bool XcpTpNetConnect (void)

  Connects to the transport layer.
- static void **XcpTpNetDisconnect** (void)

  Disconnects from the transport layer.
- static bool XcpTpNetSendPacket (tXcpTransportPacket const ∗txPacket, tXcpTransportPacket ∗rxPacket, uint16_t timeout)

  Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.
- tXcpTransport const ∗ XcpTpNetGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Variables

- static const tXcpTransport **netTransport**

  XCP transport layer structure filled with TCP/IP specifics.
- static tXcpTpNetSettings **tpNetSettings**

  The settings to use in this transport layer.
- static uint32_t **tpNetCroCounter**

  Command receive object (CRO) counter. This counter starts at 1 with each new connection and is sent with each command packet. The counter gets incremented for each command packet, allowing the server to determine the correct order for the received commands.

7.47.1 Detailed Description

XCP TCP/IP transport layer source file.

7.47.2 Function Documentation

7.47.2.1 XcpTpNetConnect()

```
static bool XcpTpNetConnect (
            void  )  [static]
```

Connects to the transport layer.

Returns

True is connected, false otherwise.

7.47.2.2 XcpTpNetGetTransport()

```
tXcpTransport const * XcpTpNetGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

7.47.2.3 XcpTpNetInit()

```
static void XcpTpNetInit (
            void const * settings )  [static]
```

Initializes the transport layer.

Parameters

settings	Pointer to settings structure.

Returns

None.

7.47.2.4 XcpTpNetSendPacket()

```
static bool XcpTpNetSendPacket (
            tXcpTransportPacket const * txPacket,
            tXcpTransportPacket * rxPacket,
            uint16_t timeout )  [static]
```

Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

Parameters

txPacket	Pointer to the packet to transmit.
rxPacket	Pointer where the received packet info is stored.
timeout	Maximum time in milliseconds to wait for the reception of the response packet.

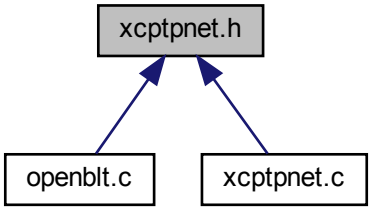
Returns

True is successful and a response packet was received, false otherwise.

7.48 xcptpnet.h File Reference

XCP TCP/IP transport layer header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tXcpTpNetSettings

  Layout of structure with settings specific to the XCP transport layer module for TCP/IP.

Functions

- tXcpTransport const * XcpTpNetGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

7.48.1 Detailed Description

XCP TCP/IP transport layer header file.

7.48.2 Function Documentation

7.48.2.1 XcpTpNetGetTransport()

```
tXcpTransport const * XcpTpNetGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

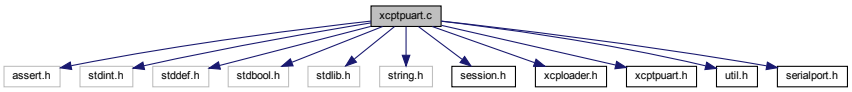
Pointer to transport layer structure.

7.49 xcptpuart.c File Reference

XCP UART transport layer source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcptpuart.h"
#include "util.h"
#include "serialport.h"
```

Include dependency graph for xcptpuart.c:



Functions

- static void XcpTpUartInit (void const *settings)

  Initializes the transport layer.
- static void **XcpTpUartTerminate** (void)

  Terminates the transport layer.
- static bool XcpTpUartConnect (void)

  Connects to the transport layer.
- static void **XcpTpUartDisconnect** (void)

  Disconnects from the transport layer.
- static bool XcpTpUartSendPacket (tXcpTransportPacket const *txPacket, tXcpTransportPacket *rxPacket, uint16_t timeout)

  Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.
- tXcpTransport const * XcpTpUartGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Variables

- static const tXcpTransport **uartTransport**

  XCP transport layer structure filled with UART specifics.
- static tXcpTpUartSettings **tpUartSettings**

  The settings to use in this transport layer.

7.49.1 Detailed Description

XCP UART transport layer source file.

7.49.2 Function Documentation

7.49.2.1 XcpTpUartConnect()

```
static bool XcpTpUartConnect (
            void  )  [static]
```

Connects to the transport layer.

Returns

True is connected, false otherwise.

7.49.2.2 XcpTpUartGetTransport()

```
tXcpTransport const * XcpTpUartGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

7.49.2.3 XcpTpUartInit()

```
static void XcpTpUartInit (
            void const * settings )  [static]
```

Initializes the transport layer.

Parameters

settings	Pointer to settings structure.

Returns

None.

7.49.2.4 XcpTpUartSendPacket()

```
static bool XcpTpUartSendPacket (
            tXcpTransportPacket const * txPacket,
            tXcpTransportPacket * rxPacket,
            uint16_t timeout )  [static]
```

Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

Parameters

txPacket	Pointer to the packet to transmit.
rxPacket	Pointer where the received packet info is stored.
timeout	Maximum time in milliseconds to wait for the reception of the response packet.

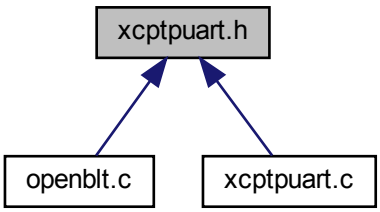
Returns

True is successful and a response packet was received, false otherwise.

7.50 xcptpuart.h File Reference

XCP UART transport layer header file.

This graph shows which files directly or indirectly include this file:



Data Structures

- struct tXcpTpUartSettings

  Layout of structure with settings specific to the XCP transport layer module for UART.

Functions

- tXcpTransport const * XcpTpUartGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

7.50.1 Detailed Description

XCP UART transport layer header file.

7.50.2 Function Documentation

7.50.2.1 XcpTpUartGetTransport()

```
tXcpTransport const * XcpTpUartGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

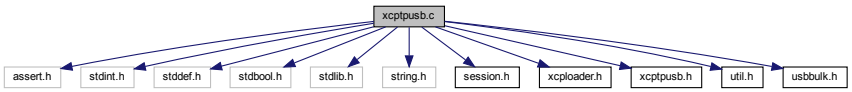
Pointer to transport layer structure.

7.51 xcptpusb.c File Reference

XCP USB transport layer source file.

```
#include <assert.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "session.h"
#include "xcploader.h"
#include "xcptpusb.h"
#include "util.h"
#include "usbbulk.h"
```

Include dependency graph for xcptpusb.c:



Functions

- static void XcpTpUsbInit (void const *settings)

  Initializes the transport layer.
- static void **XcpTpUsbTerminate** (void)

  Terminates the transport layer.
- static bool XcpTpUsbConnect (void)

  Connects to the transport layer.
- static void **XcpTpUsbDisconnect** (void)

  Disconnects from the transport layer.
- static bool XcpTpUsbSendPacket (tXcpTransportPacket const *txPacket, tXcpTransportPacket *rxPacket, uint16_t timeout)

  Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.
- tXcpTransport const * XcpTpUsbGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Variables

- static const tXcpTransport **usbTransport**

  XCP transport layer structure filled with USB specifics.

7.51.1 Detailed Description

XCP USB transport layer source file.

7.51.2 Function Documentation

7.51.2.1 XcpTpUsbConnect()

```
static bool XcpTpUsbConnect (
            void  )  [static]
```

Connects to the transport layer.

Returns

True is connected, false otherwise.

7.51.2.2 XcpTpUsbGetTransport()

```
tXcpTransport const * XcpTpUsbGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

7.51.2.3 XcpTpUsbInit()

```
static void XcpTpUsbInit (
            void const * settings )  [static]
```

Initializes the transport layer.

Parameters

settings	Pointer to settings structure.

Returns

None.

7.51.2.4 XcpTpUsbSendPacket()

```
static bool XcpTpUsbSendPacket (
            tXcpTransportPacket const * txPacket,
            tXcpTransportPacket * rxPacket,
            uint16_t timeout )  [static]
```

Transmits an XCP packet on the transport layer and attempts to receive the response packet within the specified timeout.

Parameters

txPacket	Pointer to the packet to transmit.
rxPacket	Pointer where the received packet info is stored.
timeout	Maximum time in milliseconds to wait for the reception of the response packet.

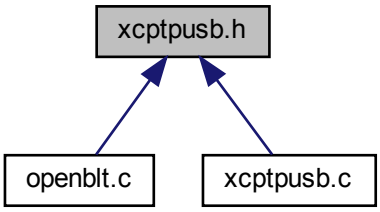
Returns

True is successful and a response packet was received, false otherwise.

7.52 xcptpusb.h File Reference

XCP USB transport layer header file.

This graph shows which files directly or indirectly include this file:



Functions

- tXcpTransport const * XcpTpUsbGetTransport (void)

  Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

7.52.1 Detailed Description

XCP USB transport layer header file.

7.52.2 Function Documentation

7.52.2.1 XcpTpUsbGetTransport()

```
tXcpTransport const * XcpTpUsbGetTransport (
            void  )
```

Obtains a pointer to the transport layer structure, so that it can be linked to the XCP protocol module.

Returns

Pointer to transport layer structure.

Index