

LAWICEL AB CANUSB API

Version 0.1.8

Copyright © 2005-2008 LAWICEL AB Sweden

All defines and prototypes are in lawicel_can.usb.h

Link your application with canusbdrv.lib. Two versions are supplied with the driver. One for Microsoft environments and one for Borland environments.

The canusbdrv.dll should be copied to the system folder if not already done so by the installation program.

From version 0.1.7 the canusbdrv also supports the CANAL interface. This interface is described here http://www.vscp.org/wiki/doku.php?id=canal_specification and there is many open and free applications and programming tools available for free at this <http://www.vscp.org> site. How to configure the driver for use with CANAL/VSCP tools is described here http://www.vscp.org/wiki/doku.php?id=canal_driver_for_lawicel_canusb

```
CANHANDLE canusb_Open( LPCSTR szID,  
                        LPCSTR szBtrate,  
                        unsigned long acceptance_code,  
                        unsigned long acceptance_mask,  
                        unsigned long flags );
```

Open a channel to a physical CAN interface.

You can open several channels to the same physical interface by calling this method several times. A virtual channel is opened for each call. When doing this it is important to remember to close the interface the same number of times as it was opened. The last close terminates the physical connection regardless of its handle. There is therefore no need to close the virtual channels in a specific order.

Very Important

To open virtual channels the interface must be named in szID. A blank id is only valid for the first open. SzBtrate, acceptance_code, acceptance_mask and flags does not have any meaning except for the first open.

Returns: handle to device if open was successful or zero on failure.

szID

Serial number for adapter or NULL to open the first found.

/szBtrate

"10" for 10kbps
"20" for 20kbps
"50" for 50kbps
"100" for 100kbps
"250" for 250kbps
"500" for 500kbps
"800" for 800kbps
"1000" for 1Mbps

or as a btr pair

btr0:btr1 pair ex. "0x03:0x1c" or 3:28 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..

The following codes should be used to get all messages

```
#define CANUSB_ACCEPTANCE_CODE_ALL      0x00000000
#define CANUSB_ACCEPTANCE_MASK_ALL     0xFFFFFFFF
```

flags

CANUSB_FLAG_TIMESTAMP - Timestamp will be set by adapter. If not set timestamp will be set by the driver.

CANUSB_FLAG_QUEUE_REPLACE Normally when the input queue is full new messages received are disregarded by setting this flag the first message in the queue is removed to make room for the new message. This flag is useful when using the ReadFirst method,

CANUSB_FLAG_BLOCK Can be set to make Read and Writes blocking. Default is an infinite block but the timeout can be changed with **canusb_SetTimeouts**. Note that **ReadFirst** and **ReadFirstEx** never block.

CANUSB_FLAG_SLOW This flag can be used at slower transmission speeds where the data stream still can be high. Every effort is made to transfer the frame even if the adapter reports that the internal buffer is full. Normally a frame is trashed if the hardware buffer becomes full to promote speed.

CANUSB_FLAG_NO_LOCAL_SEND Normally when several channels have been opened on the same physical adapter a send of a frame on one channel will be seen on the other channels. By setting this flag this is prevented.

int canusb_Close(CANHANDLE h);

Close channel with handle h.

If several channels are open only a virtual channel is closed. The last close terminates the actual physical interface.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.

Note:

If there are data in the transmission when the channel is close it will be lost. Code that open the channel, transmit and then close should bit a bit before closing to allow for the frame to be transmitted.

int canusb_getFirstAdapter(char *szAdapter, int size);

Initiate a search of available CANUSB adapters on a system. szAdapter is a pointer to a buffer that will receive the serial number of the first found adapter if one is found. Size is the size of the buffer to receive the serial number. 32 bytes is a good choice.

If more adapters are available the method will return a number greater than one and canusb_getNextAdapter should be used to get the serial numbers for the other available adapters.

Returns: <= 0 on failure. Number of available adapters on success.

Note:

Due to the internal functionality of windows USB functionality this call may fail to detect a device that has been added or removed after the driver dll has been loaded. This detection works most of the time but if absolute security is needed it is best if the CAN control and handling code is on one thread and the adapter detection on another thread/process.

int canusb_getNextAdapter(char *szAdapter, int size);

Get available CANUSB adapters on a system. szAdapter is a pointer to a buffer that will receive the serial number of a found adapter if one is found. Size is the size of the buffer to receive the serial number. 32 bytes is a good choice.

Before this call is done a call to canusb_getFirstAdapter must have been made.

If called repeatedly this call will return found adapters in order for ever.

Returns: <= 0 on failure. Positive integer on success.

int canusb_Read(CANHANDLE h, CANMsg *msg);

Read message from channel with handle h.

Note: If a callback function is defined this call will not work and returns ERROR_CANUSB_GENERAL.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.
ERROR_CANUSB_NO_MESSAGE is returned if there is no message to read.

```
// Message flags
#define CANMSG_EXTENDED    0x80 // Extended CAN id
#define CANMSG_RTR         0x40 // Remote frame
```

```
// CAN Frame
typedef struct {
    _u32 id;      // Message id
    _u32 timestamp; // timestamp in milliseconds
    _u8 flags;    // [extended_id:1][RTR:1][reserver:6]
    _u8 len;      // Frame size (0..8)
    _u8 data[ 8 ]; // Databytes 0..7
} CANMsg;
```

int canusb_ReadEx(CANHANDLE h, CANMsgEx *msg , CANDATA *pdata);

Read message from channel with handle h. Alternative version without data character array in message stucture

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.
ERROR_CANUSB_NO_MESSAGE is returned if there is no message to read.

```
// Message flags
#define CANMSG_EXTENDED    0x80 // Extended CAN id
#define CANMSG_RTR         0x40 // Remote frame

// CAN Frame
typedef struct {
    _u32 id;      // Message id
    _u32 timestamp; // timestamp in milliseconds
    _u8 flags;    // [extended_id|1][RTR:1][reserver:6]
    _u8 len;      // Frame size (0.8)
} CANMsgEx;
```

int canusb_ReadFirst(CANHANDLE h, _u32 selectid, _u8 selectflags, CANMsg *msg);

Read the first message from channel with handle h that has an id equal to selectid and flags equal to selectflags .

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.
ERROR_CANUSB_NO_MESSAGE is returned if there is no message to read.

```
// Message flags
#define CANMSG_EXTENDED    0x80 // Extended CAN id
#define CANMSG_RTR         0x40 // Remote frame
```

```
// CAN Frame
typedef struct {
    _u32 id;      // Message id
    _u32 timestamp; // timestamp in milliseconds
    _u8 flags;    // [extended_id|1][RTR:1][reserver:6]
    _u8 len;      // Frame size (0..8)
    _u8 data[ 8 ]; // Databytes 0..7
} CANMsg;
```

int canusb_ReadFirstEx(CANHANDLE h, _u32 selectid, _u8 selectflags, CANMsgEx *msg , CANDATA *pdata);

Read the first message from channel with handle h that has an id equal to selectid and flags equal to selectflags . Alternative version without data character array in message structure.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.
ERROR_CANUSB_NO_MESSAGE is returned if there is no message to read.

```
// Message flags
#define CANMSG_EXTENDED    0x80 // Extended CAN id
#define CANMSG_RTR         0x40 // Remote frame
```

```
// CAN Frame
typedef struct {
    _u32 id;      // Message id
    _u32 timestamp; // timestamp in milliseconds
    _u8 flags;    // [extended_id:1][RTR:1][reserver:6]
    _u8 len;      // Frame size (0.8)
} CANMsgEx;
```

int canusb_Write(CANHANDLE h, CANMsg *msg);

Write message to channel with handle h.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.

ERROR_CANUSB_TX_FIFO_FULL is returned if there is no room for the message..

// Message flags

#define CANMSG_EXTENDED 0x80 // Extended CAN id

#define CANMSG_RTR 0x40 // Remote frame

// CAN Frame

typedef struct {

 _u32 id; // Message id

 _u32 timestamp; // timestamp in milliseconds

 _u8 flags; // [extended_id:1][RTR:1][reserver:6]

 _u8 len; // Frame size (0..8)

 _u8 data[8]; // Databytes 0..7

} CANMsg;

int canusb_WriteEx(CANHANDLE h, CANMsgEx *msg, CANDATA *pdata);

Write message to channel with handle h.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.

ERROR_CANUSB_TX_FIFO_FULL is returned if there is no room for the message..

// Message flags

#define CANMSG_EXTENDED 0x80 // Extended CAN id

#define CANMSG_RTR 0x40 // Remote frame

// CAN Frame

typedef struct {

 _u32 id; // Message id

 _u32 timestamp; // timestamp in milliseconds

 _u8 flags; // [extended_id|1][RTR:1][reserver:6]

 _u8 len; // Frame size (0.8)

} CANMsgEx;

int canusb_Status(CANHANDLE h);

Get Adapter status for channel with handle h.

Returns: Status value with bit values as below

```
// Status bits
#define CANSTATUS_RECEIVE_FIFO_FULL      0x01
#define CANSTATUS_TRANSMIT_FIFO_FULL    0x02
#define CANSTATUS_ERROR_WARNING         0x04
#define CANSTATUS_DATA_OVERRUN          0x08
#define CANSTATUS_ERROR_PASSIVE          0x20
#define CANSTATUS_ARBITRATION_LOST      0x40
#define CANSTATUS_BUS_ERROR              0x80
```

or

ERROR_CANUSB_TIMEOUT if unable to get status in time. Call canusb_status again if this happens.

If this method is called very often performance will degrade. It is recommended that it is called at most once every ten seconds.

int canusb_VersionInfo(CANHANDLE h, LPSTR verinfo);

Get hardware/firmware and driver version for channel with handle h.

Returns: <= 0 on failure. >0 on ERROR_CANUSB_OK.

Format: "VHhFf - Nxxxx - n.n.n - CCCCCCCCCC"

V, N = Constant
H = Hardware_Major
h = Hardware_Minor
F = Firmware_Major
f = Firmware_Minor
x = CANUSB serial #
n = Driver version
C = Custom String, Default "LAWICEL AB"

int canusb_Flush(CANHANDLE h, _u8 flushflags);

Flush output buffer for channel with handle h.

If flushflags is set to FLUSH_DONTWAIT the queue is just emptied and there will be no wait for any frames in it to be sent. FLUSH_EMPTY_INQUEUE can be used to empty the inqueue.

Returns: <= 0 on failure. >0 on success.

```
// Flush bits
#define FLUSH_WAIT           0x00
#define FLUSH_DONTWAIT      0x01
#define FLUSH_EMPTY_INQUEUE 0x02
```

```
int WINAPI canusb_SetTimeouts( CANHANDLE h,  
                                _u32 receiveTimeout,  
                                _u32 transmitTimeout );
```

When an channel has been opened with the CANUSB_FLAG_BLOCK read and write operations will block infinitely. This can be changed by setting the timeout in milliseconds with this call.

Returns: <= 0 on failure. >0 on success.

```
int setReceiveCallBack( CANHANDLE h,
                        LPFNDLL_RECEIVE_CALLBACK cbfn )
```

With this method one can define a function that will receive all incoming messages. Note that **canusb_Read** will not work after this call. You can make it work again by calling this method and set **cbfn** equal to NULL.

The callback function should be defined as

```
void fn( CANMsg *pMsg );
```

Note: that the channel has to be open to be able to set a callback function.

Returns: <= 0 on failure. >0 on success.

Sample code

```
////////////////////////////////////
// Set and enable callback
//

void CTestDriverDlg::OnCheckTestCallBack()
{
    int rv;

    UpdateData( true );    // Get dialog data

    if ( m_bOpen ) {
        if ( m_bTestCallback ) {
            // Set Callback
            *gDisplayBuf = 0;
            gReceiveCount = 0;
            gbUpdate = TRUE;
            rv = canusb_setReceiveCallBack( m_drvHandle,
            (LPFNDLL_RECEIVE_CALLBACK)ReceiveCallback );
        }
        else {
            // Disable callback
            rv = canusb_setReceiveCallBack( m_drvHandle, NULL );
        }
    }
}
```

```

////////////////////////////////////
//      Receive Callback function
//
// This callback can be a member of the class if it is defined static. If so
// also all methods it calls must be static.
//

void __stdcall ReceiveCallback( CANMsg *pmsg )
{
    int i;
    char buf[MAX_PATH], wrkbuf[ 10 ];
    char cExtended;
    char cRTR;

    // A new message received
    gReceiveCount++;

    cExtended = 'S';
    if ( pmsg->flags & CANMSG_EXTENDED ) {
        cExtended = 'E';
    }

    cRTR = ' ';
    if ( pmsg->flags & CANMSG_RTR ) {
        cRTR = 'R';
    }

    sprintf( buf,
        "Callback <- %c%c id = %08X timestamp = %08X len = %d data =",
        cExtended,
        cRTR,
        pmsg->id,
        pmsg->timestamp,
        pmsg->len );

    for ( i=0; i<pmsg->len; i++ ) {
        sprintf( wrkbuf, "%02x ", pmsg->data[ i ] );
        strcat( buf, wrkbuf );
    }

    strcat( buf, "\r\n" );

    if ( sizeof( gDisplayBuf ) < ( strlen(buf) + strlen(gDisplayBuf) ) ) {
        *gDisplayBuf = 0;
        strcat( gDisplayBuf, "Buffer cleared because of overrun" );
        strcat( gDisplayBuf, "\r\n" );
    }

    strcat( gDisplayBuf, buf );
    gbUpdate = TRUE;

    //theApp.m_pdlg->msgToStatusList( pmsg, true );
}

```

Error return codes

#define ERROR_CANUSB_OK	1
#define ERROR_CANUSB_GENERAL	-1
#define ERROR_CANUSB_OPEN_SUBSYSTEM	-2
#define ERROR_CANUSB_COMMAND_SUBSYSTEM	-3
#define ERROR_CANUSB_NOT_OPEN	-4
#define ERROR_CANUSB_TX_FIFO_FULL	-5
#define ERROR_CANUSB_INVALID_PARAM	-6
#define ERROR_CANUSB_NO_MESSAGE	-7
#define ERROR_CANUSB_MEMORY_ERROR	-8
#define ERROR_CANUSB_NO_DEVICE	-9
#define ERROR_CANUSB_TIMEOUT	-10
#define ERROR_CANUSB_INVALID_HARDWARE	-11