X2CAN API
(version 2.x)

Ing. David Španěl
Info@canlab.cz
www.canlab.cz

Contents:

1. INTRODUCTION 2

2. GENERAL FUNCTIONS, BASIC DATA STRUCTURES 2

3. PP2CAN API FUNCTIONS 7

4. USB2CAN API FUNCTIONS 11

5. V2CAN API FUNCTIONS 17

6. X2CAN API FUNCTION 18

7. EXAMPLE 01 20

8. DETAILED DESCRIPTION OF USB2CAN API FUNCTIONS 23

9. CHANGES IN API VERSIONS 32

1. Introduction

Together with development of USB2CAN interface, new API was created, allowing to use both PP2CAN converter and new modern USB2CAN converter together. This API structure allows to use eventual further variants of CAN interface cards, including third-party cards. The API covers basic functions of PP2CAN, USB2CAN and V2CAN interfaces to unify calling of basic functions to open, communication and close interface port. This way, source code of application is not dependent on used adapter. However, there is still enough space for using of special routines for particular interface.

2. General functions, basic data structures

The X2CAN API allows to unify structure definition carrying the CAN message. This structure is defined as follows:

```
typedef struct
{
       unsigned short Hour;
       unsigned short Minute;
       unsigned short Second;
       unsigned short Milliseconds;
} CANMessageTime;

typedef struct
{
       unsigned __int16 Id1;
       unsigned __int32 Id2;
       unsigned __int32 Id;
       unsigned char length;
       bool rtr;
       bool st_ext;
       unsigned char data[8];
       CANMessageTime time;
} CAN_MESSAGE;
```

Id1	standard identifier part (11-bit)
Id2	extended identifier part (18-bit)
Id	29-bit identifier format
length	number of message data bytes
rtr	RTR message flag (Remote Transfer Request)
st_ext	discern between message with standard or extended identifier
data	based on data bytes
time	time stamp structure

This structure is defined in header file canbus.h. The time element includes time stamp and the CANMessageTime structure is defined in the same file.

The CAN_MESSAGE structure includes two identifier syntax types: 11+18-bit identifier and 29-bit identifier. Use following two functions to synchronize identifier values:

```
void CANMsgUpdateFrom11_18(CAN_MESSAGE *message);
void CANMsgUpdateFrom29(CAN_MESSAGE *message);
```

The first one updates 29-bit identifier from the 11+18 format, the second one works backwards. Internal API identifier format is 11+18. When 29-bit format is used, ID accuracy must be verified before the message is sent to CAN using the CANMsgUpdateFrom29. On Receipt the API provides conversion automatically.

Based on CAN message structure definition, this file provides a storage for auxiliary

function headers used to convert identifier from CAN_MESSAGE structure format to registry-saved format used often for SJA1000, MCP2510, MCP2515, I82527, CC750 and CC770 circuits. There are two parameters for these functions. The first is the CAN_MESSAGE structure. To do the conversion, fill the ID1, Id2, length, rtr and st_ext items. The second is a pointer to array of unsigned chars with length of 5. For each circuit there are 2 variants TX and RX for transmit (TX) and receive (RX) buffer format.

```
void CANMsg2MCP251x_TX(CAN_MESSAGE message, unsigned char *data);
void CANMsg2MCP251x_RX(CAN_MESSAGE message, unsigned char *data);
void CANMsg2SJA1000_TX(CAN_MESSAGE message, unsigned char *data);
void CANMsg2SJA1000_RX(CAN_MESSAGE message, unsigned char *data);
void CANMsg2I82527_TX (CAN_MESSAGE message, unsigned char *data);
void CANMsg2I82527_RX (CAN_MESSAGE message, unsigned char *data);
void CANMsg2CC7x0_TX (CAN_MESSAGE message, unsigned char *data);
void CANMsg2CC7x0_RX (CAN_MESSAGE message, unsigned char *data);
```

There are several functions for creation of identifier, enabling to work with particular high-level protocols.

```
void CANMsgSAE(CAN_MESSAGE *message
                    ,unsigned char Priority
                    ,bool DataPage
                    ,unsigned char PDUFormat
                    ,unsigned char DestinationAddress
                    ,unsigned char SourceAddress);

void CANMsgDeviceNet(CAN_MESSAGE *Message
                    ,unsigned char Group
                    ,unsigned char Data1
                    ,unsigned char Data2);

void CANMsgSDSShort(CAN_MESSAGE *message
                    ,bool Dir_Pri
                    ,unsigned char LogicalAddress
                    ,unsigned char ServiceType);

void CANMsgSDSLong(CAN_MESSAGE *Message
                    ,bool Dir_Pri
                    ,unsigned char LogicalAddress
                    ,unsigned char ServiceType
                    ,unsigned char Dlc
                    ,unsigned char ServiceSpecifiers
                    ,unsigned char EmbeddedObjectID
                    ,unsigned char ServiceParameters);

void CANMsgSDSFragmentedLong(CAN_MESSAGE *Message
                    ,bool Dir_Pri
                    ,unsigned char LogicalAddress
                    ,unsigned char ServiceType
                    ,unsigned char Dlc
                    ,unsigned char ServiceSpecifiers
                    ,unsigned char EmbeddedObjectID
                    ,unsigned char ServiceParameters
                    ,unsigned char FragmentNumber
                    ,unsigned char TotalFragmentBytes);
```

This file defines an enumeration type CAN_SPEED too. As obvious, this type is designed to enter/save communication speed. Here is the definition:

```
enum CAN_SPEED
{
      SPEED_10k = 10,
      SPEED_20k = 20,
```

```
        SPEED_33_3k = 33,
        SPEED_50k = 50,
        SPEED_62_5k = 62,
        SPEED_83_3k = 83,
        SPEED_100k = 100,
        SPEED_125k = 125,
        SPEED_250k = 250,
        SPEED_500k = 500,
        SPEED_1M = 1000,
        SPEED_USR = 0,
};
```

Use PP2CAN_timing and USB2CAN_timing structures to setup the registry for setting of communication speed and sampling point. Here is the definition:

```
typedef struct
{
        unsigned char speed_10k[3];
        unsigned char speed_20k[3];
        unsigned char speed_33k[3];
        unsigned char speed_50k[3];
        unsigned char speed_62k[3];
        unsigned char speed_83k[3];
        unsigned char speed_100k[3];
        unsigned char speed_125k[3];
        unsigned char speed_250k[3];
        unsigned char speed_500k[3];
        unsigned char speed_1000k[3];
}PP2CAN_timing;

typedef struct
{
        unsigned char speed_10k[2];
        unsigned char speed_20k[2];
        unsigned char speed_33k[2];
        unsigned char speed_50k[2];
        unsigned char speed_62k[2];
        unsigned char speed_83k[2];
        unsigned char speed_100k[2];
        unsigned char speed_125k[2];
        unsigned char speed_250k[2];
        unsigned char speed_500k[2];
        unsigned char speed_800k[2];
        unsigned char speed_1000k[2];
}USB2CAN_timing;
```

Two global variables PP2CAN_default_timing and USB2CAN_default_timing are defined, which are derived from aforesaid structures. Their values are preset to default and these variables are being used to initialize the CAN interface. Values of these global variables can be modified by user. Here is the definition of default values:

```
#define MCP_10KB_CNF1_Osc20 0x27
#define MCP_10KB_CNF2_Osc20 0xBF
#define MCP_10KB_CNF3_Osc20 0x07

#define MCP_20KB_CNF1_Osc20 0x31
#define MCP_20KB_CNF2_Osc20 0xA0
#define MCP_20KB_CNF3_Osc20 0x02

#define MCP_33KB_CNF1_Osc20 0x1D
#define MCP_33KB_CNF2_Osc20 0xA0
#define MCP_33KB_CNF3_Osc20 0x02

#define MCP_50KB_CNF1_Osc20 0x13
#define MCP_50KB_CNF2_Osc20 0xA0
#define MCP_50KB_CNF3_Osc20 0x02

#define SJA_10KB_BTR0_Osc16 0x31
#define SJA_10KB_BTR1_Osc16 0xBA

#define SJA_20KB_BTR0_Osc16 0x18
#define SJA_20KB_BTR1_Osc16 0xBA

#define SJA_33KB_BTR0_Osc16 0x13
#define SJA_33KB_BTR1_Osc16 0xA7

#define SJA_50KB_BTR0_Osc16 0x09
#define SJA_50KB_BTR1_Osc16 0x1C

#define SJA_62KB_BTR0_Osc16 0x07
#define SJA_62KB_BTR1_Osc16 0xBA
```

```
#define MCP_62KB_CNF1_Osc20 0x0F
#define MCP_62KB_CNF2_Osc20 0xA0
#define MCP_62KB_CNF3_Osc20 0x02

#define MCP_83KB_CNF1_Osc20 0x0B
#define MCP_83KB_CNF2_Osc20 0xA0
#define MCP_83KB_CNF3_Osc20 0x02

#define MCP_100KB_CNF1_Osc20 0x09
#define MCP_100KB_CNF2_Osc20 0xA0
#define MCP_100KB_CNF3_Osc20 0x02

#define MCP_125KB_CNF1_Osc20 0x07
#define MCP_125KB_CNF2_Osc20 0xA0
#define MCP_125KB_CNF3_Osc20 0x02

#define MCP_250KB_CNF1_Osc20 0x02
#define MCP_250KB_CNF2_Osc20 0xA0
#define MCP_250KB_CNF3_Osc20 0x02

#define MCP_500KB_CNF1_Osc20 0x01
#define MCP_500KB_CNF2_Osc20 0xA0
#define MCP_500KB_CNF3_Osc20 0x02

#define MCP_1000KB_CNF1_Osc20 0x00
#define MCP_1000KB_CNF2_Osc20 0xA0
#define MCP_1000KB_CNF3_Osc20 0x02
```

```
#define SJA_83KB_BTR0_Osc16 0x07
#define SJA_83KB_BTR1_Osc16 0xA7

#define SJA_100KB_BTR0_Osc16 0x04
#define SJA_100KB_BTR1_Osc16 0x1C

#define SJA_125KB_BTR0_Osc16 0x03
#define SJA_125KB_BTR1_Osc16 0x1C

#define SJA_250KB_BTR0_Osc16 0x01
#define SJA_250KB_BTR1_Osc16 0x1C

#define SJA_500KB_BTR0_Osc16 0x00
#define SJA_500KB_BTR1_Osc16 0x1C

#define SJA_800KB_BTR0_Osc16 0x00
#define SJA_800KB_BTR1_Osc16 0x16

#define SJA_1000KB_BTR0_Osc16 0x00
#define SJA_1000KB_BTR1_Osc16 0x14
```

You can finish the canport.h file content description with several auxiliary functions. The first 6 functions are used to convert communication speeds in CAN_SPEED, Kbaud and Int. format.

```
CAN_SPEED KBaud2CANSpeed(int speed);
CAN_SPEED Int2CANSpeed(int speed);

int CANSpeed2Int(CAN_SPEED speed);
int CANSpeed2KBaud(CAN_SPEED speed);

int KBaud2Int(int speed);
int Int2KBaud(int speed);
```

The functions do the conversion based on following table:

CAN_SPEED	Int	KBaud
SPEED_10k	0	10
SPEED_20k	1	20
SPEED_33_3k	2	33
SPEED_50k	3	50
SPEED_62.5k	4	62
SPEED_83_3k	5	83
SPEED_100k	6	100
SPEED_125k	7	125
SPEED_250k	8	250
SPEED_500k	9	500
SPEED_1000k	10	1000

```
void Byte2BinString(unsigned char data, char *text);
```

This last function converts an unsigned char number to binary string of length 8. E.g. 3 decimal to string 00000011.

The bitbase.h file contains following macros for testing bit status:

```
#define hw_bitset(var,bitno) ((var)|=1<<(bitno))
#define hw_bitclr(var,bitno) ((var)&=~(1<<(bitno)))
#define hw_bitest(var,bitno) (((var)>>(bitno))&0x01)
#define hw_bittest(var,bitno) (hw_bitest(var,bitno))
```

The MCP2510 and SJA1000 files contain registry address and bit mask definitions for SJA1000, MCP2510 and MCP2515 CAN bus controllers.

Let's step now from auxiliary functions and definitions to description of the X2CAN API. We will start first with function description for PP2CAN adapter, then we will go through function description for USB2CAN and virtual port V2CAN and finally we will finish with description of arch-over interface X2CAN. The PP2CAN and USB2CAN adapter functions can be used separately without the X2CAN, however, X2CAN allows to simply substitute both adapters so it is recommended to use it. The knowledge of PP2CAN and USB2CAN API, though, will allow you to use some "specialties" of these converters and to better understand their function principle.

3. PP2CAN API functions

There are four types of this adapter (High speed rev. 0 and 1, Low speed rev. 0 and Single wire rev. 0). This is why following constants necessary to initialize the adapter are defined in the PP2CAN.h file:

```
#define PP2CAN_HW_HIGH_SPEED_0 0
#define PP2CAN_HW_HIGH_SPEED_1 1
#define PP2CAN_HW_LOW_SPEED_0 2
#define PP2CAN_HW_SINGLE_WIRE_0 3
```

Call the PP2CAN_Open function to initialize the adapter. Only one PP2CAN adapter can be operated on one PC at the time. However, any number of USB2CAN adapters can be simultaneously used.

```
bool PP2CAN_Open(WORD Address
      , CAN_SPEED Speed
      , void (*error_function)(int err_code, const char * error_string)
      , void (*msg_receiver)(MCP2510Msg *msg)
      , int HW_Version
      , bool OneShotMode
      , bool PassiveMode
      , int ThreadPriority );
```

There are two parameters for the function:

Address	Parallel port address. Go to System –> Device Manager –> Ports (COM and LPT) –> Printer port (LPTx) –> Resources – I/O range. You will find here usually one of the following values: 0x378, 0x278, 0x3BC.
Speed	Communication speed.
error_function	Pointer to function called when error occurs. The function gets the error code and text description.
msg_receiver	Pointer to function called when message is received. As a parameter, this function has a message structure. If this parameter is NULL no function is called and user application accesses to received message buffer. This second way is recommended.
HW_Version	Version of PP2CAN hardware (e.g. PP2CAN_HW_HIGH_SPEED_1 in most cases).
OneShotMode	Message is sent only once, if receipt is not confirmed, it is not repeated.
PassiveMode	The parameter sets the adapter to status where message cannot be sent, only received. It protects from unwanted sending of messages and increases the number of captured messages.
ThreadPriority	Thread priority for communication with adapter. THREAD_PRIORITY_TIME_CRITICAL=0, THREAD_PRIORITY_HIGHEST=1, THREAD_PRIORITY_NORMAL=2.

The PP2CAN_Open function initializes the adapter with default filter settings (all messages are received) and switches the adapter to ready mode. If initialization was OK, it returns true. Any errors are signalized by calling error function (error_function) and then it returns false.

Use the PP2CAN_Close function to close the port (close communication). There are no parameters for this function.

There are 4 modes for the device and are defined in PP2CAN_MODE enumeration type. Here is the definition:

```
enum PP2CAN_MODE
{
	PP2CAN_MODE_CONFIG = 0,
	PP2CAN_MODE_NORMAL,
	PP2CAN_MODE_LOOPBACK,
	PP2CAN_MODE_LISTEN_ONLY,
};
```

Use following functions for setting and detecting of current mode:

```
void PP2CAN_SetMode(PP2CAN_MODE mode);
void PP2CAN_SetConfigMode(void);
void PP2CAN_SetNormalMode(void);
void PP2CAN_SetLoopbackMode(void);
void PP2CAN_SetListenOnlyMode(void);
PP2CAN_MODE PP2CAN_GetMode(void);
```

In PP2CAN_MODE_CONFIG mode you can change communication speed, set message filters, change the OneShotMode mode etc. In PP2CAN_MODE_NORMAL mode the adapter can send and receive messages. In PP2CAN_MODE_LOOPBACK mode sent messages are received back using HW loopback. The PP2CAN_MODE_LISTEN_ONLY allows only to receive messages, even corrupted ones.

If the PP2CAN_Open call succeeded, messages can be received and sent to bus. Use following functions to send messages to CAN bus:

```
bool PP2CAN_SendMessage(MCP2510Msg &data);
bool PP2CAN_SendCANMessage(CAN_MESSAGE message);
bool PP2CAN_SendRegisterMessage(unsigned char Data[13]);
bool PP2CAN_SendMCPMessage(MCP2510Msg *Data);
bool PP2CAN_SendStandardMessage(unsigned __int16 StandardId
	, bool RTR
	, unsigned char Length
	, unsigned char *Data);
bool PP2CAN_SendExtendedMessage(unsigned __int16 StandardId
	, unsigned __int32 ExtendedId
	, bool RTR
	, unsigned char Length
	, unsigned char *Data);
bool PP2CAN_SendExtendedMessage29(unsigned __int32 ExtendedId
	, bool RTR
	, unsigned char Length
	, unsigned char *Data);
```

The CAN_MESSAGE structure description was already discussed. The MCP2510Msg structure functions are implemented because of backward compatibility with PP2CAN API v1.x. This structure is defined as follows:

```
typedef union
{
	unsigned char bytes[13];
	struct
	{
		unsigned char ID[4];
		unsigned char DLC;
		unsigned char data[8];
	} item;
} MCP2510MsgData;

typedef struct
{
	MCP2510MsgData msg;
```

```
      CANMessageTime time;
} MCP2510Msg;
```

Use several functions for getting and setting of identifier and message type (standard/extended identifier, data/rtr frame) to simplify working with the MCP2510MsgData structure. As for data parameter, the first byte address of the structure is committed.

```
bool PP2CAN_GetId1(unsigned char *data
             ,unsigned __int16 &Id1
             ,unsigned __int32 &Id2);

bool PP2CAN_GetId2(unsigned char *data
             ,unsigned __int16 *Id1
             ,unsigned __int32 *Id2);

void PP2CAN_SetExtendedId(unsigned char *data
             ,unsigned __int16 Id1
             ,unsigned __int32 Id2);

void PP2CAN_SetStandardId(unsigned char *data, unsigned __int16 Id1);
void PP2CAN_SetExtendedId29(unsigned char *data, unsigned __int32 Id);

bool PP2CAN_SetDLC(unsigned char *data, bool RTR, unsigned char length);
void PP2CAN_GetDLC1(unsigned char *data, bool &RTR, unsigned char &length);
void PP2CAN_GetDLC2(unsigned char *data, bool *RTR, unsigned char *length);
```

You can use these functions to read received messages from input buffer:

```
bool PP2CAN_GetMessage(MCP2510Msg **data);
bool PP2CAN_GetMCPMessage(MCP2510Msg *data);
bool PP2CAN_GetCANMessage(CAN_MESSAGE *message);
bool PP2CAN_GetMessage11_18(bool *StExt
             , unsigned __int16 *StandardId
             , unsigned __int32 *ExtendedId
             , bool *RTR
             , unsigned char *Length
             , unsigned char *Data);
bool PP2CAN_GetMessage29(bool *StExt
             , unsigned __int32 *Id
             , bool *RTR
             , unsigned char *Length
             , unsigned char *Data);
void PP2CAN_DeleteMessage(MCP2510Msg *data);
```

If the message has been read out using the PP2CAN_GetMessage function, you must delete it from memory after the message is processed by calling the PP2CAN_DeleteMessage function. If the buffer is empty the functions return false.

Use this function to get the number of messages in send buffer:

```
int PP2CAN_GetTXBufferLength(void);
```

Use this function to get the number of messages waiting for processing:

```
int PP2CAN_GetRXBufferLength(void);
```

You can empty the buffers by calling:

```
void PP2CAN_ClearBuffers(void);
```

Use this function to wait on incoming message in thread:

```
bool USB2CAN_WaitForRxMessage(unsigned int Timeout);
```

This function returns true if the message was received. If no message was received in an interval specified in timeout parameter, the result is false. Put in the INFINITE value to set infinite waiting on message receipt.

From user point of view, another useful functions are functions for registry value monitoring of TEC (Transmit Error Counter), REC (Receive Error Counter) and RST counter (Reset Counter). See the CAN bus specification for explanation of the meaning of TEC and REC registries. The RST counter counts the number of restarts of the adapter when switched to Bus-off status. For simple PP2CAN adapter, reading of TEC and REC registries is influenced by number of sent and received messages. If huge amount of messages is transferred, it is advisable to disable reading of these registries. Use following 2 functions to enable/disable reading. Further three functions are used for reading of discussed registries (counters).

```
void PP2CAN_EnableReadTEC(bool enable);
void PP2CAN_EnableReadREC(bool enable);

unsigned char PP2CAN_GetREC(void);
unsigned char PP2CAN_GetTEC(void);
unsigned int PP2CAN_GetRST(void);
```

The PP2CAN by itself contains a lot of further functions – e.g. you can set aside the adapter communication thread and replace it with your own one using API functions for writing, reading and modifying the MCP251x circuit registries, you can set message filters etc. See the pp2can.h header file for description of these functions.

4. USB2CAN API functions

Compared to PP2CAN adapter which allows to have only one active adapter on one PC at a time, you can use virtually any number of USB2CAN adapters on one PC simultaneously. For selection and working with particular adapters are defined these functions:

```
void* USB2CAN_PrepareAdapter(void);
void* USB2CAN_PrepareAdapterEx(EUSB2CANDevice selector, char *name);
void USB2CAN_SelectActualAdapter(void* adapter);
void USB2CAN_DestroyAdapter(void* adapter);
```

The first USB2CAN_PrepareAdapter function returns void type pointer to main data structure. This functions is necessary for the adapter being opened to work. The USB2CAN_PrepareAdapter function opens the first adapter found. If you want to specify particular adapter use the USB2CAN_PrepareAdapterEx function. It has two parameters and the first one specifies adapter selection method. You can select the adapter by name (same for each adapter by default; use the FTDI utility, serial number and number of adapter connection to change it). Particular variants then look like this:

```
enum EUSB2CANDevice
{
	OPEN_BY_DEVICE_NUMBER = 0,
	OPEN_BY_SERIAL_NUMBER = FT_OPEN_BY_SERIAL_NUMBER, // 1
	OPEN_BY_DESCRIPTION = FT_OPEN_BY_DESCRIPTION, // 2
};

void *my_can_adapter;

// Prvni pripojene zarizeni
my_can_adapter = USB2CAN_PrepareAdapterEx(OPEN_BY_DEVICE_NUMBER, "0");

// Druhé pripojene zarizeni
my_can_adapter = USB2CAN_PrepareAdapterEx(OPEN_BY_DEVICE_NUMBER, "1");

// Seriove cislo
my_can_adapter = USB2CAN_PrepareAdapterEx(FT_OPEN_BY_SERIAL_NUMBER,
                                                              "000112");

// Adapter USB2CAN s konfiguracni pameti
my_can_adapter = USB2CAN_PrepareAdapterEx(FT_OPEN_BY_DESCRIPTION, "USB2CAN");

// Adapter USB2CAN bez konfiguracni pameti
my_can_adapter = USB2CAN_PrepareAdapterEx(FT_OPEN_BY_DESCRIPTION
                                            , "USB <-> Serial");
```

Use the USB2CAN_SelectActualAdapter to select actual adapter. After usage of this adapter is finished, call the USB2CAN_DestroyAdapter function to delete data from memory.

Use the following function to open and close the CAN port of the adapter:

```
bool USB2CAN_Open(CAN_SPEED speed
	, bool CreateCommThreads
	, void (*error_function)(int err_code, const char * error_string)
	, bool LowSpeed);
bool USB2CAN_Close(void);
```

The first parameter of USB2CAN_Open function is communication speed. The second parameter is set to true; only set this parameter to false when using custom mechanisms for communication with device using USB. The third parameter is pointer to error function; this pointer and associated function is the same as in case of PP2CAN

adapter. The last parameter distinguishes between high-speed and low-speed variant; it is true for low-speed. If the adapter was successfully initialized and CAN port is opened, this function returns true. Use the USB2CAN_Close function to close the port.

After the port was successfully opened, the usual set of functions is available for sending and receiving messages using the CAN bus:

```
bool USB2CAN_SendMessage(SJA1000MsgData &data);

bool USB2CAN_SendCANMessage(CAN_MESSAGE message);

bool USB2CAN_SendRegisterMessage(unsigned char Data[13]);

bool USB2CAN_SendSJAMessage(SJA1000MsgData *Data);

bool USB2CAN_SendStandardMessage(unsigned __int16 StandardId
                         ,bool RTR
                         ,unsigned char Length
                         ,unsigned char *Data);

bool USB2CAN_SendExtendedMessage(unsigned __int16 StandardId¨
                         ,unsigned __int32 ExtendedId
                         ,bool RTR
                         ,unsigned char Length
                         ,unsigned char *Data);

bool USB2CAN_SendExtendedMessage29(unsigned __int32 ExtendedId
                         ,bool RTR
                         ,unsigned char Length
                         ,unsigned char *Data);

bool USB2CAN_GetMessage(SJA1000MsgData **data); - ZRUŠENO

bool USB2CAN_GetSJAMessage(SJA1000MsgData *data);

bool USB2CAN_GetCANMessage(CAN_MESSAGE *message);

bool USB2CAN_GetMessage11_18(bool *StExt
                         ,unsigned __int16 *StandardId
                         ,unsigned __int32 *ExtendedId
                         ,bool *RTR
                         ,unsigned char *Length
                         ,unsigned char *Data);

bool USB2CAN_GetMessage11_18(bool* StExt
                         ,unsigned __int16* StandardId
                         ,unsigned __int32* ExtendedId
                         ,bool* RTR
                         ,unsigned char* Length
                         ,unsigned char* Data
                         ,CANMessageTime* time);

bool USB2CAN_GetMessage29(bool *StExt
                         ,unsigned __int32 *Id
                         ,bool *RTR
                         ,unsigned char *Length
                         ,unsigned char *Data);
```

The USB2CAN_GetSJAMessage and USB2CAN_GetCANMessage functions expect a valid pointer. This is where message data are written then.

```
void USB2CAN_DeleteMessage(SJA1000MsgData *data); - ZRUŠENO
```

Do not use delete to delete a message received using USB2CAN_GetMessage; use this function instead.

```
bool USB2CAN_WaitForRxMessage(unsigned int timeout);
```

Waiting to receive CAN message.

```
int USB2CAN_GetTXBufferLength();
```

Returns buffer size of received messages.

```
int USB2CAN_GetRXBufferLength();
```

Returns buffer size of messages waiting to be sent.

```
void USB2CAN_SetTimeStampMode (bool mode);
```

Enables precise measurement of the time message was received.

```
void USB2CAN_ClearBuffers (void);
```

Empties buffers for sending and receiving the CAN messages.

```
unsigned char USB2CAN_GetREC (void);
```

Returns the Receive Error Counter value.

```
unsigned char USB2CAN_GetTEC (void);
```

Returns the Transmit Error Counter value.

```
unsigned int USB2CAN_GetRST (void);
```

Returns the Reset Counter value (number of SJA1000 resets).

```
void USB2CAN_EnableReadTEC (bool enable);
```

Enabling of automatic TEC reading.

```
void USB2CAN_EnableReadREC (bool enable);
```

Enabling of automatic REC reading.

These functions are analogous to PP2CAN adapter functions, let's mention here only a SJA1000MsgData structure. Here it is:

```
typedef union
{
	unsigned char bytes[13];
	struct {
		unsigned char DLC;
		unsigned char ID[2];
		unsigned char data[8];
	} standard;
	struct {
		unsigned char DLC;
		unsigned char ID[4];
		unsigned char data[8];
	} extended;
} SJA1000MsgData;
```

For low-level control (for owners of development documentation) there is a set of following functions:

```
void USB2CAN_SetupBasic(int BaudRate);
```

Default initialization of SJA1000 and USB2CAN, the same is done for USB2CAN_Open.

```
void USB2CAN_SetTimeout(int ms);
```

Timeout setting function for adapter reply (confirmation) to messages sent from PC.

```
bool USB2CAN_Loopback();
```

USB_LOOPBACK message transmission; if the adapter reply is received back, it returns true.

```
bool USB2CAN_SetMode(int Mode);
```

Operation mode setting (BOOT, CONFIG, NORMAL, LOOPBACK).

```
bool USB2CAN_GetMode(int *Mode);
```

Detection function of current operation mode; if the adapter doesn't send current mode before timeout occurs, returns false.

```
bool USB2CAN_GetFirmwareVersion(char *Version);
```

Detection function for current firmware version; if the adapter doesn't reply before timeout expires, returns false.

```
bool USB2CAN_Command0(unsigned char Command);

bool USB2CAN_Command1(unsigned char Command
                     ,unsigned char Param1);

bool USB2CAN_Command2(unsigned char Command
                     ,unsigned char Param1
                     ,unsigned char Param2);

bool USB2CAN_Command3(unsigned char Command
                     ,unsigned char Param1
                     ,unsigned char Param2
                     ,unsigned char Param3);
```

These functions are designed to send special COMMAND group commands (see development documentation).

```
bool USB2CAN_ReadReg(unsigned char Address, unsigned char *Data);
```

The function reads the specified registry value of the SJA1000 circuit in the USB2CAN adapter.

```
bool USB2CAN_WriteReg(unsigned char Address, unsigned char Data);
```

The function writes value to specified registry of the SJA1000 circuit in the USB2CAN adapter.

```
bool USB2CAN_WriteReadReg(unsigned char Address´
              ,unsigned char Data
              ,unsigned char *DataOut);
```

The function reads and re-reads the specified registry value of the SJA1000 circuit in the USB2CAN adapter.

```
bool USB2CAN_BitModReg(unsigned char Address
            ,unsigned char Mask
            ,unsigned char Data);
```

Bit modification using the value and registry mask of the SJA1000 circuit.

```
bool USB2CAN_BitModReadReg(unsigned char Address
            ,unsigned char Mask
            ,unsigned char Data
            ,unsigned char *DataOut);
```

Bit modification using the value and mask and re-read of registry value of the SJA1000 circuit.

```
bool USB2CAN_WriteInstruction(unsigned __int16 Address
,unsigned __int16 Instruction);
```

Writes instruction into program memory (firmware change) of the control processor.

```
bool USB2CAN_ReadTEC(unsigned char *TEC);
```

The command sends a query to TEC registry status (Transmit Error Counter) and value of this registry is returned back. The value is stored on address specified in TEC. If no reply comes before timeout occurs, it returns false. However, it is recommended to enable automatic reading using USB2CAN_EnableReadTEC and reading of value using USB2CAN_GetTEC.

```
bool USB2CAN_ReadREC(unsigned char *REC);
```

The command queries the REC registry status (Receive Error Counter) and returns value of this registry. If no reply comes before timeout occurs, it returns false.

```
bool USB2CAN_ReadRST(unsigned int *RST);
```

The command queries the RST counter (Reset Error Counter) and returns value of this registry. If no reply comes before timeout occurs, it returns false. This counter represents number of switches to Bus-Off status (and restarts of SJA1000).

```
void USB2CAN_CmdPeliCAN(void);
```

Setting of the PeliCAN mode for SJA1000 circuit.

```
void USB2CAN_CmdResetMode(void);
```

Setting of configuration mode for SJA1000 circuit.

```
void USB2CAN_CmdOperatingMode(void);
```

Setting of basic operation mode for SJA1000 circuit.

```
void USB2CAN_CmdBaudRate(unsigned char t0, unsigned char t1);
```

Writing values into timing registries of SJA1000 circuit. Only possible in configuration mode.

```
void USB2CAN_CmdEnableReadTEC(bool Enable);
```

Enabling of reading the TEC registry of SJA1000 circuit.

```
void USB2CAN_CmdEnableReadREC(bool Enable);
```

Enabling of reading the REC registry of SJA1000 circuit.

5. V2CAN API functions

Implementing of virtual CAN bus port allows for application based on X2CAN API to work without connected CAN bus adapter. A CAN message sent using virtual port is received back. This allows e.g. offline analysis of logged data and above all simplifies testing of developed SW built on X2CAN API. A usual set of functions is available, however with a V2CAN prefix.

```
bool V2CAN_Open();
bool V2CAN_Close();

bool V2CAN_SendRegisterMessage(unsigned char Data[13]);

bool V2CAN_SendStandardMessage(unsigned __int16 StandardId
             ,bool RTR
             ,unsigned char Length
             ,unsigned char *Data);

bool V2CAN_SendExtendedMessage(unsigned __int16 StandardId
             ,unsigned __int32 ExtendedId
             ,bool RTR, unsigned char Length
             ,unsigned char *Data);

bool V2CAN_SendExtendedMessage29(unsigned __int32 ExtendedId
             ,bool RTR
             ,unsigned char Length
             ,unsigned char *Data);

bool V2CAN_SendCANMessage(CAN_MESSAGE message);

bool V2CAN_GetMessage11_18(bool *StExt
             ,unsigned __int16 *StandardId
             ,unsigned __int32 *ExtendedId
             ,bool *RTR
             ,unsigned char *Length
             ,unsigned char *Data);

bool V2CAN_GetMessage29(bool *StExt
             ,unsigned __int32 *Id
             ,bool *RTR, unsigned char *Length
             ,unsigned char *Data);

bool V2CAN_GetCANMessage(CAN_MESSAGE *message);

void V2CAN_SetTimeStampMode(bool mode);
int V2CAN_GetRXBufferLength(void);
int V2CAN_GetTXBufferLength(void);
void V2CAN_ClearBuffers(void);
bool V2CAN_WaitForRxMessage(unsigned int timeout);

unsigned char V2CAN_GetREC(void);
unsigned char V2CAN_GetTEC(void);
unsigned int V2CAN_GetRST(void);
void V2CAN_EnableReadTEC(bool enable);
void V2CAN_EnableReadREC(bool enable);
```

6. X2CAN API functions

For applications to be independent on real-used CAN adapter, X2CAN API contains a group of functions that allows this. However, you must specify used adapter when initializing (opening) the CAN port. Other functions for sending and receiving of messages are same as for PP2CAN, USB2CAN and V2CAN. Current version supports only simultaneous work of more USB2CAN adapters. You cannot use X2CAN to work simultaneously with one PP2CAN adapter and one or more USB2CAN adapter(s). When this combination is needed, use X2CAN API or USB2CAN API to work with USB2CAN adapter, and PP2CAN API to work with PP2CAN adapter.

```
void X2CAN_Prepare(void);
```

Initialization of data structures.

```
bool X2CAN_Open(CAN_INTERFACE interface_type
      ,CAN_SPEED speed
      ,void (*error_function)(int err_code, const char * error_string) );
```

Opening of CAN port; CAN adapter, communication speed and error function is specified here.

```
bool X2CAN_Open_PP2CAN(WORD address
      ,CAN_SPEED speed
      ,void (*error_function)(int err_code, const char * error_string)
      ,int HW_Version
      ,bool OneShotMode
      ,bool PassiveMode
      ,int ThreadPriority );
```

Opening of PP2CAN CAN port. CAN adapter, communication speed and error function is specified here. Allows to set OneShotMode or Passive mode and thread priority.

```
bool X2CAN_Open_V2CAN(void);
```

Opening of V2CAN CAN port.

```
bool X2CAN_Open_USB2CAN(CAN_SPEED speed
      ,void (*error_function)(int err_code, const char * error_string)
      ,bool low_speed);
```

Opening of USB2CAN port. Call USB2CAN_PrepareAdapter and SelectActualAdapter prior to usage.

```
bool X2CAN_Close();
```

Closing of port.

```
bool X2CAN_IsInitialized();
```

Returns true if the port is initialized.

```
bool X2CAN_IsPP2CAN();
```

Returns true if currently used adapter is PP2CAN.

```
bool X2CAN_IsUSB2CAN();
```

Returns true if currently used adapter is USB2CAN.

```
bool X2CAN_IsV2CAN();
```

Returns true if currently used adapter is V2CAN.

```
CAN_INTERFACE X2CAN_GetInterfaceType();
```

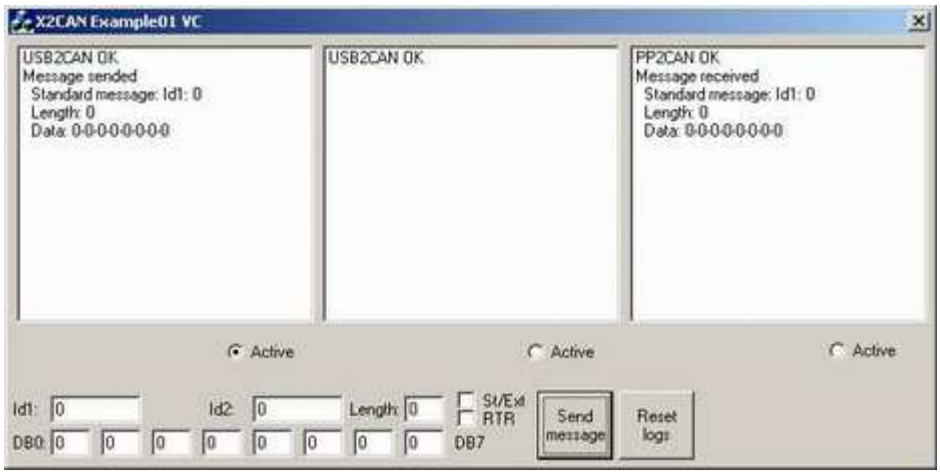
Returns the type of currently used CAN adapter.

Then, there is a usual set of functions available for sending/receiving of messages, reading buffer sizes and all kinds of flags. This group of functions is actually implemented using pointers to functions of particular adapters. Yet, you don't have to limit yourself to use these functions. You can call them combined with particular adapter API-native specialized functions. However, it is necessary only in special cases; most of the times we will get along with this set of functions allowing to send and read messages from CAN bus.

```
bool X2CAN_SendStandardMessage(unsigned __int16 StandardId
             ,bool RTR
             ,unsigned char Length
             ,unsigned char *Data);

bool X2CAN_SendExtendedMessage(unsigned __int16 StandardId
             ,unsigned __int32 ExtendedId
             ,bool RTR
             ,unsigned char Length
             ,unsigned char *Data);

bool X2CAN_SendExtendedMessage29(unsigned __int32 ExtendedId
             ,bool RTR
             ,unsigned char Length
             ,unsigned char *Data);

bool X2CAN_SendCANMessage(CAN_MESSAGE message);

bool X2CAN_GetMessage11_18(bool *StExt
             ,unsigned __int16 *StandardId
             ,unsigned __int32 *ExtendedId
             ,bool *RTR
             ,unsigned char *Length
             ,unsigned char *Data);

bool X2CAN_GetMessage29(bool *StExt
             ,unsigned __int32 *Id
             ,bool *RTR
             ,unsigned char *Length
             ,unsigned char *Data);

bool X2CAN_GetCANMessage(CAN_MESSAGE *message);

void X2CAN_SetTimeStampMode(bool mode);
int X2CAN_GetRXBufferLength(void);
int X2CAN_GetTXBufferLength(void);
void X2CAN_ClearBuffers(void);

bool X2CAN_WaitForRxMessage(unsigned int timeout);

unsigned char X2CAN_GetREC(void);
unsigned char X2CAN_GetTEC(void);
unsigned int X2CAN_GetRST(void);
void X2CAN_EnableReadTEC(bool enable);
void X2CAN_EnableReadREC(bool enable);
```

7. Example 01

This example is designed for Microsoft Visual Studio C++. It was created in version 6, however, it can be imported to NET version also. It demonstrates elementary work with two USB2CAN adapters and one PP2CAN adapter simultaneously. To run this example, you don't have to have connected all of them. The following picture shows the application window. It provides a box to fill in the message being sent and three log lists each for particular CAN bus adapter. Communication speed is forced directly in code. All of the adapters are of High speed type. Click here to do download complete source code. You must include the X2CAN API library to the project.



It is a Dialog-based MFC application Specific PP2CAN and USB2CAN APIs are used for adapters. Main dialog class is called CExampe01_VCDlg. The OnInitDialog method, as a part of this class, initializes the adapters. The following part of code does the initialization:

```
// Vytvorime datove struktury pro adaptery USB2CAN
// Otevirame adaptery parametru device number
// Prvni adapter
USB2CAN_adapter_1 = USB2CAN_PrepareAdapterEx(OPEN_BY_DEVICE_NUMBER, "0");
// Druhy adapter
USB2CAN_adapter_2 = USB2CAN_PrepareAdapterEx(OPEN_BY_DEVICE_NUMBER, "1");

// Nastaveni se kterym adapterem budu pracovat
USB2CAN_SelectActualAdapter(USB2CAN_adapter_1);
// Otevreni adapteru
bool back = USB2CAN_Open(SPEED_125k, TRUE ,ErrorUSB2CAN_1, false);
if(!back) m_log_usb2can_1.AddString("USB2CAN Error");
else m_log_usb2can_1.AddString("USB2CAN OK");
// Nastaveni se kterym adapterem budu pracovat
USB2CAN_SelectActualAdapter(USB2CAN_adapter_2);
// Otevreni adapteru
back = USB2CAN_Open(SPEED_125k, TRUE ,ErrorUSB2CAN_2, false);
if(!back) m_log_usb2can_2.AddString("USB2CAN Error");
else m_log_usb2can_2.AddString("USB2CAN OK");

// Otevreni adapteru PP2CAN, adresa paralelniho portu je 888 dekadicky (378h)
back = PP2CAN_Open(888,SPEED_125k,
ErrorPP2CAN,NULL,PP2CAN_HW_HIGH_SPEED_1,false,false,1);
if(!back) m_log_pp2can.AddString("PP2CAN Error");
else m_log_pp2can.AddString("PP2CAN OK");
```

On application close the ports are closed too:

```
// Ukonceni prace s adaptery
```

```
// Vyber aktualniho adapteru USB2CAN
USB2CAN_SelectActualAdapter(USB2CAN_adapter_1);
// Uzavreni CAN portu
USB2CAN_Close();
USB2CAN_DestroyAdapter(USB2CAN_adapter_1);

// Vyber aktualniho adapteru USB2CAN
USB2CAN_SelectActualAdapter(USB2CAN_adapter_2);
// Uzavreni CAN portu
USB2CAN_Close();
USB2CAN_DestroyAdapter(USB2CAN_adapter_2);
// Uzavreni adapteru PP2CAN
PP2CAN_Close();
```

Sending of message when button is pressed (OnSend function):

```
// CAN zprava kterou chceme odeslat
CAN_MESSAGE message;
UpdateData(TRUE);
// Vyplneni datovych polozek zpravy
message.Id1 = m_id1;
message.Id2 = m_id2;
(m_st_ext) ? message.st_ext = true : message.st_ext = false;
(m_rtr) ? message.rtr = true : message.rtr = false;
message.length = m_length;

message.data[0] = m_db0;
message.data[1] = m_db1;
message.data[2] = m_db2;
message.data[3] = m_db3;
message.data[4] = m_db4;
message.data[5] = m_db5;
message.data[6] = m_db6;
message.data[7] = m_db7;

// Odeslani zpravy podle vybraneho (aktivniho) adapteru
switch(m_active)
{
	case 0:
		USB2CAN_SelectActualAdapter(USB2CAN_adapter_1);
		USB2CAN_SendCANMessage(message);
		m_log_usb2can_1.AddString("Message sended");
		PrintMessageInfo(&m_log_usb2can_1, &message);
	break;
	case 1:
		USB2CAN_SelectActualAdapter(USB2CAN_adapter_2);
		USB2CAN_SendCANMessage(message);
		m_log_usb2can_2.AddString("Message sended");
		PrintMessageInfo(&m_log_usb2can_2, &message);
	break;
	case 2:
		PP2CAN_SendCANMessage(message);
		m_log_pp2can.AddString("Message sended");
		PrintMessageInfo(&m_log_pp2can, &message);
	break;
}
```

Here's how incoming messages are being checked in OnTimer function:

```
// Prijem a zpracovani zprav
CAN_MESSAGE message;
// Vyber aktualniho USB2CAN adapteru
USB2CAN_SelectActualAdapter(USB2CAN_adapter_1);
// Jsou prijaty nejake zpravy prvnim USB2CAN adapterem
if(USB2CAN_GetRXBufferLength()>0)
{
	// Jestlize ano, prectu jednu z nich
```

```
        if(USB2CAN_GetCANMessage(&message))
        {
                m_log_usb2can_1.AddString("Message received");
                PrintMessageInfo(&m_log_usb2can_1, &message);
        }
}
// Jsou prijaty nejake zpravy druhym USB2CAN adapterem
USB2CAN_SelectActualAdapter(USB2CAN_adapter_2);
if(USB2CAN_GetRXBufferLength()>0)
{
        if(USB2CAN_GetCANMessage(&message))
        {
                m_log_usb2can_2.AddString("Message received");
                PrintMessageInfo(&m_log_usb2can_2, &message);
        }
}

// Jsou prijaty nejake zpravy PP2CAN adapterem
if(PP2CAN_GetRXBufferLength()>0)
{
        if(PP2CAN_GetCANMessage(&message))
        {
                m_log_pp2can.AddString("Message received");
                PrintMessageInfo(&m_log_pp2can, &message);
        }
}
```

However, more messages can come during the time interval of OnTimer calling; this is why it's more advisable to use this way of reading of CAN messages:

```
while(PP2CAN_GetCANMessage(&message))
{
        m_log_pp2can.AddString("Message received");
        PrintMessageInfo(&m_log_pp2can, &message);
}
```

If a separate thread would read the messages, it is advisable to use PP2CAN_WaitForMessage and USB2CAN_WaitForMessage functions. If more USB2CAN adapters are used in more threads, you must close calling of functions (or group of functions) between USB2CAN_SelectActualAdapterAccess and USB2CAN_UnselectActualAdapterAccess.

8. Detailed description of USB2CAN API functions

The most frequently used variant of USB2CAN API is X2CAN. That's why we will discuss in detail the behavior and usage of this API's functions.

```
X2CAN_DLLMAPPING void* USB2CAN_PrepareAdapter(void);
```

This function allocates memory and initializes several variables of the data structure the API functions work with. This function is basic and the simplest variant of how to create this structure. You must initialize this structure prior to opening the adapter. It is designed for adapter without EEPROM configuration memory. By default, the "OPEN_BY_DESCRIPTION" device is selected and "USB <-> Serial" is set as identification string. This is standard identification string for FTDI 245 circuit without configuration EPROM. This function also initializes a critical section for handling of exclusive access to the device in case of multi-thread application using "USB2CAN_SelectActualAdapterAccess".

```
X2CAN_DLLMAPPING void* USB2CAN_PrepareAdapterEEPROM(void);
```

This function is designed for adapters with EEPROM. The only difference is in initializing of identification string to "USB2CAN".

```
X2CAN_DLLMAPPING void* USB2CAN_PrepareAdapterEx(EUSB2CANDevice selector, char
*name)
```

The last variant of data structure initialization allows to specify adapter selection using the Device_selector parameter. You can use following variants:

- OPEN_BY_DEVICE_NUMBER
- OPEN_BY_SERIAL_NUMBER
- OPEN_BY_DESCRIPTION

The identification string must be contained in the name parameter. Use OPEN_BY_DEVICE_NUMBER to select adapter by connection number, than a string "0" can be in the name parameter. Use OPEN_BY_DESCRIPTION for "USB <-> Serial" or "USB2CAN". Use OPEN_BY_SERIAL_NUMBER to select adapter by serial number set in configuration EEPROM.

```
X2CAN_DLLMAPPING void USB2CAN_SelectActualAdapter(void* adapter);
```

Use this function only in special cases when application is not working with adapter on multi-thread basis. It sets adapter to work with, however, it doesn't set exclusive access to adapter using critical section; it only sets current adapter for user to work with. You must call this function first before you call functions for opening the adapter, sending the messages etc.

```
X2CAN_DLLMAPPING void USB2CAN_SelectActualAdapterAccess(void* adapter);
```

Sets the adapter to work with and locks critical section for exclusive access. Consequently, you can call function(s) for work with adapter. Call the "USB2CAN_UnselectActualAdapterAccess" after calling these functions. The "USB2CAN_UnselectActualAdapterAccess" is designed primarily for usage in multi-thread applications and when working with more adapters simultaneously, to change current adapter.

```
X2CAN_DLLMAPPING void USB2CAN_UnselectActualAdapterAccess(void);
```

Opens critical section locked in "USB2CAN_SelectActualAdapterAccess".

```
X2CAN_DLLMAPPING void USB2CAN_DestroyAdapter(void* adapter);
```

This function frees up memory allocated for adapter after finishing work with adapter e.g. when closing user application.

```
X2CAN_DLLMAPPING bool USB2CAN_Open(CAN_SPEED speed, bool CreateCommThreads , void
(*error_function)(int err_code, const char * error_string), bool low_speed);
```

This function opens and initializes the USB2CAN adapter. Use the CAN_SPEED parameter to set communication speed. Always set CreateCommThreads on true; otherwise low-level communication threads necessary for adapter to work are not created. Normally they are not created only in special cases, e.g. during firmware debugging. The error_function parameter is a pointer to an API-called function in case of error to carry a text description of error. This pointer can be set to NULL. Set the low_speed parameter on true when using a low_speed adapter.

```
X2CAN_DLLMAPPING USB2CAN_CloseErrorChannel(void);
```

This function sets the pointer to error function, passed on when calling "USB2CAN_Open", on NULL. This function is designed for cases when e.g. because of the BusOff status the error function is called frequently, so the user application cannot be closed.

```
X2CAN_DLLMAPPING bool USB2CAN_Close(void);
```

Standard closing of CAN port of the USB2CAN device.

```
X2CAN_DLLMAPPING bool USB2CAN_CloseExt(void);
```

This variant of CAN port closing of the USB2CAN device performs in addition a software simulation of disconnecting and connecting of the device. It is designed for cases when a serious error of USB driver occurred.

```
X2CAN_DLLMAPPING void USB2CAN_CheckUSBStatus(char *after_action);
```

The function is primarily designed for internal USB2CAN API usage. It checks the status of USB device and calls error function in case of error. The "after_action" text is used in error message passed on the error function to identify where exactly the error occurred.

```
X2CAN_DLLMAPPING bool USB2CAN_CheckUSB(void);
```

The function is primarily designed for internal USB2CAN API usage. It returns false if an incorrect USB status is signalized.

```
X2CAN_DLLMAPPING void USB2CAN_SimulateUSBError(unsigned int status);
```

The function is primarily used for development of USB2CAN API. It simulates error on USB.

```
X2CAN_DLLMAPPING bool USB2CAN_CyclePort(void);
```

The function is primarily designed for internal USB2CAN API usage. It performs SW simulation of disconnecting and connecting of the USB2CAN device. It returns true if simulation of disconnecting and connecting passed OK.

```
X2CAN_DLLMAPPING bool USB2CAN_WaitForConnection(unsigned int hundreds_of_ms);
```

The function waits on device to be connected, that was specified in previous calling of some of the USB2CAN_PrepareAdapter-type function.

```
X2CAN_DLLMAPPING void USB2CAN_SetTimeout(int ms);
```

The function is primarily designed for internal USB2CAN API usage. It is not recommended for user to use this function. It sets timeout intervals for waiting on data from USB2CAN adapter.

```
X2CAN_DLLMAPPING void USB2CAN_SetUSBBlockLimit(int Limit);
```

The function is primarily designed for internal USB2CAN API usage. It sets limit for USB2CAN to block accepting of requirements for sending of CAN messages because of processor internal buffer overflow. Allowed values are 1-18. It modifies this parameter, however it doesn't make settings in USB2CAN adapter.

```
X2CAN_DLLMAPPING void USB2CAN_SetupBasic(int BaudRate);
```

The function is primarily designed for internal USB2CAN API usage. It initializes the SJA1000 controller and sets communication speed. You don't have to use this function during standard API usage because it is called together with USB2CAN_Open.

```
X2CAN_DLLMAPPING void USB2CAN_ListenOnly(int BaudRate, bool listen_only);
```

This function allows to switch the Listen only function on and off. In this mode, USB2CAN cannot send messages; it only passively captures messages and doesn't intervene into CAN bus communication. See the documentation of the SJA1000 circuit for detailed information about this mode.

```
X2CAN_DLLMAPPING bool USB2CAN_Loopback();
```

The function is primarily designed for internal USB2CAN API usage. It sends the Loopback message to USB2CAN device and the device sends it back to PC. The function waits until a message is received back or timeout occurs.

```
X2CAN_DLLMAPPING bool USB2CAN_SetMode(int Mode);
```

The function is primarily designed for internal USB2CAN API usage. It sets one of four operation modes of USB2CAN device: BOOT_MODE, CONFIG_MODE, NORMAL_MODE and LOOPBACK_MODE.

```
X2CAN_DLLMAPPING bool USB2CAN_GetMode(int *Mode);
```

Returns currently set mode. The Mode parameter must point to a valid variable.

```
X2CAN_DLLMAPPING bool USB2CAN_GetFirmwareVersion(char *Version);
```

Returns current firmware version of the USB2CAN device. The Version parameter must point to an array of minimum length of 13 chars.

```
X2CAN_DLLMAPPING bool USB2CAN_Command0(unsigned char Command);
```

The function is primarily designed for internal USB2CAN API usage. It allows to send Command-type commands to a USB2CAN device. See the USB2CAN: Structure of USB communication document for more information about these commands.

```
X2CAN_DLLMAPPING bool USB2CAN_Command1(unsigned char Command, unsigned char
Param1);
```

The function is primarily designed for internal USB2CAN API usage. It allows to send Command-type commands to a USB2CAN device. See the USB2CAN: Structure of USB communication document for more information about these commands.

```
X2CAN_DLLMAPPING bool USB2CAN_Command2(unsigned char Command, unsigned char Param1,
unsigned char Param2);
```

The function is primarily designed for internal USB2CAN API usage. It allows to send Command-type commands to a USB2CAN device. See the USB2CAN: Structure of USB communication document for more information about these commands.

```
X2CAN_DLLMAPPING bool USB2CAN_Command3(unsigned char Command, unsigned char Param1,
unsigned char Param2, unsigned char Param3);
```

The function is primarily designed for internal USB2CAN API usage. It allows to send Command-type commands to a USB2CAN device. See the USB2CAN: Structure of USB communication document for more information about these commands.

```
X2CAN_DLLMAPPING bool USB2CAN_ReadReg(unsigned char Address, unsigned char *Data);
```

The function is primarily designed for internal USB2CAN API usage. It reads specified registry of the SJA1000 CAN controller.

```
X2CAN_DLLMAPPING bool USB2CAN_WriteReg(unsigned char Address, unsigned char Data);
```

The function is primarily designed for internal USB2CAN API usage. It writes into specified registry of the SJA1000 CAN controller.

```
X2CAN_DLLMAPPING bool USB2CAN_WriteReadReg(unsigned char Address, unsigned char
Data, unsigned char *DataOut);
```

The function is primarily designed for internal USB2CAN API usage. It writes into specified registry of the SJA1000 CAN controller. After the data is written it reads it back.

```
X2CAN_DLLMAPPING bool USB2CAN_BitModReg(unsigned char Address, unsigned char Mask,
unsigned char Data);
```

The function is primarily designed for internal USB2CAN API usage. It modifies the registry bits by selected mask. Bits with a value of 1 in the mask are modified.

```
X2CAN_DLLMAPPING bool USB2CAN_BitModReadReg(unsigned char Address, unsigned char
Mask, unsigned char Data, unsigned char *DataOut);
```

The function is primarily designed for internal USB2CAN API usage. It modifies the registry bits by selected mask. Bits with a value of 1 in the mask are modified. After the modification it reads beack the registry value.

```
X2CAN_DLLMAPPING bool USB2CAN_WriteInstruction(unsigned __int16 Address, unsigned
__int16 Instruction);
```

The function is primarily designed for internal USB2CAN API usage. It modifies firmware of the USB2CAN adapter. However, the function doesn't nothing in user versions of API.

```
X2CAN_DLLMAPPING bool USB2CAN_ReadInstruction(unsigned __int16 Address, unsigned
__int16 *Instruction);
```

The function is primarily designed for internal USB2CAN API usage. It modifies firmware of the USB2CAN adapter. However, the function doesn't nothing in user versions of API.

```
X2CAN_DLLMAPPING bool USB2CAN_ReadTEC(unsigned char *TEC);
```

The function reads the TEC registry value (Transmit Error Counter). Internally, it sends request for reading of this registry and waits until it is finished. This way, other USB transactions with this adapter are blocked. That's why it is recommended to read this registry

using a combination of calling “USB2CAN_EnableReadTEC” and “USB2CAN_GetTEC” which ensures theat no transaction blocking occurs. Reading is provided by inner API layer.

```
X2CAN_DLLMAPPING bool USB2CAN_ReadREC(unsigned char *REC);
```

The function reads the TEC registry value (Transmit Error Counter). Internally, it sends request for reading of this registry and waits until it is finished. This way, other USB transactions with this adapter are blocked. That’s why it is recommended to read this registry using a combination of calling “USB2CAN_EnableReadTEC” and “USB2CAN_GetTEC” which ensures theat no transaction blocking occurs. Reading is provided by inner API layer.

```
X2CAN_DLLMAPPING bool USB2CAN_ReadRST(unsigned int *RST);
```

The function reads the value of Reset counter. This counter counts the number of resets of the SJA1000 circuit when turning to BusOff status.

```
X2CAN_DLLMAPPING void USB2CAN_SetPassiveMode(bool mode);
```

Sets passive communication mode. It blocks sending of messages to CAN in API functions to prevent unwanted sending of CAN message when working on unknown CAN system.

```
X2CAN_DLLMAPPING void USB2CAN_SetOneShotMode(bool mode);
```

Sets a “One Shot” mode. If CAN controller cannot send a message in this mode, no further attempt to send will be done.

```
X2CAN_DLLMAPPING void USB2CAN_CmdBasicCAN(void);
```

The function is primarily designed for internal USB2CAN API usage. It sets the BasicCAN mode of the SJA1000 controller. It is not recommended for users to use this function.

```
X2CAN_DLLMAPPING void USB2CAN_CmdResetMode(void);
```

The function is primarily designed for internal USB2CAN API usage. It sets the Reset mode of the SJA1000 controller. It is not recommended for users to use this function.

```
X2CAN_DLLMAPPING void USB2CAN_CmdPeliCAN(void);
```

The function is primarily designed for internal USB2CAN API usage. It sets the PeliCAN mode of the SJA1000 controller. It is not recommended for users to use this function.

```
X2CAN_DLLMAPPING void USB2CAN_CmdOperatingMode(void);
```

The function is primarily designed for internal USB2CAN API usage. It sets operation mode of the SJA1000 controller. It is not recommended for users to use this function.

```
X2CAN_DLLMAPPING void USB2CAN_CmdBaudRate(unsigned char t0, unsigned char t1);
```

The function is primarily designed for internal USB2CAN API usage. It sets registries of the SJA1000 controller.

```
X2CAN_DLLMAPPING void USB2CAN_CmdCriticalTransmitLimit(unsigned char Limit);
```

The function is primarily designed for internal USB2CAN API usage. It sets limit for USB2CAN to start blocking accepting of requirements for sending of CAN messages because of processor internal buffer overflow. Allowed values are 1-18. It modifies this parameter in the USB2CAN adapter.

```
X2CAN_DLLMAPPING void USB2CAN_CmdReadyTransmitLimit(unsigned char Limit);
```

The function is primarily designed for internal USB2CAN API usage. It sets limit for USB2CAN to stop blocking accepting of requirements for sending of CAN messages because of processor internal buffer overflow. Allowed values are 1-18. It modifies this parameter in the USB2CAN adapter.

```
void USB2CAN_HandBasicCAN(void);
```
The function is primarily designed for internal USB2CAN API usage. Unlike similar aforementioned function that sends requirement for setting the mode into USB2CAN device, this function sets the BasicCAN mode by setting particular registries from PC.

```
void USB2CAN_HandPeliCAN(void);
```
The function is primarily designed for internal USB2CAN API usage. Unlike similar aforementioned function that sends requirement for setting the mode into USB2CAN device, this function sets the PeliCAN mode by setting particular registries from PC.

```
void USB2CAN_HandResetMode(void);
```
The function is primarily designed for internal USB2CAN API usage. Unlike similar aforementioned function that sends requirement for setting the mode into USB2CAN device, this function sets the Reset mode by setting particular registries from PC.

```
void USB2CAN_HandOperatingMode(void);
```
The function is primarily designed for internal USB2CAN API usage. Unlike similar aforementioned function that sends requirement for setting the mode into USB2CAN device, this function sets the Operating mode by setting particular registries from PC.

```
void USB2CAN_HandListenOnlyMode(void);
```
The function is primarily designed for internal USB2CAN API usage. Unlike similar aforementioned function that sends requirement for setting the mode into USB2CAN device, this function sets the ListenOnly mode by setting particular registries from PC.

```
X2CAN_DLLMAPPING void USB2CAN_BaudRate_Osc16(int BaudRate);
```
The function is primarily designed for internal USB2CAN API usage. It sets the Timing registry values in data structure of the adapter. However, it doesn't make any settings to USB2CAN device.

```
X2CAN_DLLMAPPING void USB2CAN_BaudRate_Osc20(int BaudRate);
```
The function is primarily designed for internal USB2CAN API usage. It sets the Timing registry values in data structure of the adapter. However, it doesn't make any settings to USB2CAN device. This function is designed only for special type of adapter with 20MHz crystal.

```
bool USB2CAN_SendUSBLoopback();
```
The function is primarily designed for internal USB2CAN API usage. It sends a Loopback message to USB2CAN device. Unlike the USB2CAN_Loopback function, this function doesn't wait until the message is received back. The function is available only when a static library is used.

```
bool USB2CAN_SendUSB(unsigned char *data, int length);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
bool USB2CAN_SendUSBReadReg(unsigned char Address);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
bool USB2CAN_SendUSBWriteReg(unsigned char Address, unsigned char data);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
X2CAN_DLLMAPPING void USB2CAN_GetStatistics(USB2CAN_Statistics *Statistics);
```
The function is designed only for testing and development. Returns statistics about communication with USB2CAN adapter and the CAN bus to address of the Statistics structure.

```
double USB2CAN_PerformanceTest(int NumberOfMessages);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
double USB2CAN_PerformanceReadTest(int NumberOfMessages);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
double USB2CAN_PerformanceWriteTest(int NumberOfMessages);
```
The function is designed only for testing and development. It is available only when a static library is used.

```
X2CAN_DLLMAPPING void USB2CAN_SWSetTransmitCritical(bool state);
```
The function is designed only for testing and development. It simulates overflow of internal buffers of the USB2CAN device.

```
X2CAN_DLLMAPPING int USB2CAN_GetTransmitErrors();
```
The function is designed only for testing and development.

```
X2CAN_DLLMAPPING void USB2CAN_ClearTransmitErrors();
```
The function is designed only for testing and development.

```
X2CAN_DLLMAPPING int USB2CAN_GetTXBufferLength();
```
Returns size of message software buffer in PC waiting for sending to CAN.

```
X2CAN_DLLMAPPING int USB2CAN_GetRXBufferLength();
```
Returns size of message software buffer in PC; the messages were received from CAN and are waiting for processing.

```
X2CAN_DLLMAPPING void USB2CAN_SetTimeStampMode (bool mode);
```
Allows to measure time of receiving the message. However, CPU usage increases.

```
X2CAN_DLLMAPPING void USB2CAN_ClearBuffers (void);
```
Deletes message buffers both waiting for send to CAN and receive from CAN and waiting for processing.

```
X2CAN_DLLMAPPING unsigned char USB2CAN_GetREC (void);
```
The function is designed to read from the REC (Receive Error Counter) from the SJA1000 CAN controller. Use the USB2CAN_EnableReadREC function to enable reading.

```
X2CAN_DLLMAPPING unsigned char USB2CAN_GetTEC (void);
```
The function is designed to read from the TEC (Teceive Error Counter) from the SJA1000 CAN controller. Use the USB2CAN_EnableRead TEC function to enable reading.

```
X2CAN_DLLMAPPING unsigned int USB2CAN_GetRST (void);
```
The function is designed to read from the RST counter (Reset Counter).

```
X2CAN_DLLMAPPING void USB2CAN_EnableReadTEC (bool enable);
```
Allows to read the TEC registry. API takes care about periodical reading. Use the “USB2CAN_GetTEC” to read current value.

```
X2CAN_DLLMAPPING void USB2CAN_EnableReadREC (bool enable);
```
Allows to read the REC registry. API takes care about periodical reading. Use the "USB2CAN_GetREC" to read current value.

```
X2CAN_DLLMAPPING void USB2CAN_AutoRecconectEnable(bool enable);
```
The function is primarily designed for internal USB2CAN API usage.

```
X2CAN_DLLMAPPING void USB2CAN_HighBusLoad(void);
```
Increases USB data read thread priority.

```
X2CAN_DLLMAPPING void USB2CAN_LowBusLoad(void);
```
Decreases USB data read thread priority. For cases when decrease of PC load is needed and there is low traffic on CAN.

```
X2CAN_DLLMAPPING void USB2CAN_LowTransmit(void);
```
Decreases CAN messages sending thread priority in case we prefer receiving and sending level is low or none at all.

```
X2CAN_DLLMAPPING void USB2CAN_ImproveEMCImmunity(unsigned int count);
```
Increases EMC USB interference resistance. It calls FTDI function of the FT_SetResetPipeRetryCount driver internally.

```
X2CAN_DLLMAPPING bool USB2CAN_SendCANMessage(CAN_MESSAGE message);
```
The function is designed to send CAN message. The message is passed on using the CAN_MESSAGE structure. The API is oriented mainly to work with messages in this format. Before this, you must set the Id1 (standard 11-bit identifier part) and Id2 (extended 18-bit identifier part) items. If you work with 29-bit Id identifier use the "CANMsgUpdateFrom29" function to update Id1 and Id2 (canbus.h).

```
X2CAN_DLLMAPPING bool USB2CAN_SendRegisterMessage(unsigned char Data[13]);
```
The function is designed to send CAN message. The message is entered in SJA1000 circuit registry format.

```
X2CAN_DLLMAPPING bool USB2CAN_SendSJAMessage(SJA1000MsgData *Data);
```
The function is designed to send CAN message. The message is entered in SJA1000 circuit registry format. However, the "SJA1000MsgData" structure is used instead of array of bytes because it allows better orientation in particular bytes.

```
X2CAN_DLLMAPPING bool USB2CAN_SendStandardMessage(unsigned __int16 StandardId, bool
RTR, unsigned char Length, unsigned char *Data);
```
The function is designed to send standard 11-bit CAN message. Particular items of the message are entered separately, so you don't have to fill in the CAN_MESSAGE structure.

```
X2CAN_DLLMAPPING bool USB2CAN_SendExtendedMessage(unsigned __int16 StandardId,
unsigned __int32 ExtendedId, bool RTR, unsigned char Length, unsigned char *Data);
```
The function is designed to send extended 29-bit CAN message. Particular items of the message are entered separately, so you don't have to fill in the CAN_MESSAGE structure. Both standard and extended parts are entered separately.

```
X2CAN_DLLMAPPING bool USB2CAN_SendExtendedMessage29(unsigned __int32 ExtendedId,
bool RTR, unsigned char Length, unsigned char *Data);
```
The function is designed to send extended 29-bit CAN message. Particular items of the message are entered separately, so you don't have to fill in the CAN_MESSAGE structure. The identifier is entered in 29-bit format.

```
X2CAN_DLLMAPPING bool USB2CAN_SimulateCANMessage(CAN_MESSAGE message);
```
The function allows to simulate receiving of message using PC.

```
X2CAN_DLLMAPPING bool USB2CAN_GetSJAMessage(SJA1000MsgData *data);
```

Returns true if the message is read; if received message buffer is empty, it returns false. The message is stored to an address specified with the “data” parameter in the SJA1000 circuit registry format. The “SJA1000MsgData” structure address must be thus valid.

```
X2CAN_DLLMAPPING bool USB2CAN_GetCANMessage(CAN_MESSAGE *message);
```

Returns true if the message is read; if received message buffer is empty, it returns false. The message is stored to an address specified with the “message” parameter in the CAN_MESSAGE registry format. The “CAN_MESSAGE” structure address must be thus valid.

```
X2CAN_DLLMAPPING bool USB2CAN_GetMessage11_18(bool *StExt, unsigned __int16
*StandardId, unsigned __int32 *ExtendedId, bool *RTR, unsigned char *Length,
unsigned char *Data);
```

Returns true if the message is read; if received message buffer is empty, it returns false. The message is stored to an address specified with parameters, all the pointers must be thus valid. The message is read in 11+(18)-bit format.

```
X2CAN_DLLMAPPING bool USB2CAN_GetMessage11_18_t(bool* StExt, unsigned __int16*
StandardId, unsigned __int32* ExtendedId, bool* RTR, unsigned char* Length,
unsigned char* Data, CANMessageTime* time);
```

Returns true if the message is read; if received message buffer is empty, it returns false. The message is stored to an address specified with parameters, all the pointers must be thus valid. The message is read in 11+(18)-bit format. Unlike the previous function, it sets message receipt time to the address specified with the CANMessageTime parameter.

```
X2CAN_DLLMAPPING bool USB2CAN_GetMessage29(bool *StExt, unsigned __int32 *Id, bool
*RTR, unsigned char *Length, unsigned char *Data);
```

Returns true if the message is read; if received message buffer is empty, it returns false. The message is stored to an address specified with parameters, all the pointers must be thus valid. The message is read in 29-bit format.

```
X2CAN_DLLMAPPING bool USB2CAN_WaitForRxMessage(unsigned int timeout);
```

The function waits for message to be received, until specified amount of milliseconds (timeout) passes. If message was received, it returns true.

```
X2CAN_DLLMAPPING bool USB2CAN_WaitForRxMessage2(unsigned int timeout,void
*adapter);
```

The function waits for message to be received, until specified amount of milliseconds (timeout) passes. If message was received, it returns true. This variant is suitable when working with more adapters. The pointer to adapter data structure is passed into the function, it doesn’t require setting of current adapter using “USB2CAN_SelectActualAdapterAccess”.

```
X2CAN_DLLMAPPING char* USB2CAN_GetUSBDeviceInfo(void);
```

The function returns pointer to text string with adapter serial number.

```
X2CAN_DLLMAPPING bool USB2CAN_ErrorCaptureDecoder(unsigned char ECC_register, char*
TextDescription);
```

The function is designed to decode the ECC error registry. Text description of error is saved to an address specified with the TextDescription parameter.

9. Changes in API versions

2.025	– Added the interface for both network adapter operation and Remote CAN server. – Added feature for registering functions for operation with third-party adapters.
2.020	– The functions are declared with Extern “C”. This adjustment was made to ensure better compatibility between compilers. This required renaming of some functions that were originally overloaded (e.g. the USB2CAN_GetMessage11_18 function). – The USB2CAN_PrepareAdapter is designed for adapter without configuration EEPROM; if configuration EEPROM is present, use the USB2CAN_PrepareAdapterEEPROM function. – The USB2CAN_GetUSBDeviceInfo returns serial number of the adapter. – The USB2CAN_HighBusLoad, USB2CAN_LowBusLoad and USB2CAN_LowTransmit functions allow to decrease CPU load in some applications.
2.015	– Functions to ensure exclusive access to adapters when operating multiple adapters simultaneously: USB2CAN_SelectActualAdapterAccess and USB2CAN_SelectActualAdapterAccess. The USB2CAN_ErrorCaptureDecoder function for decoding CAN transfer errors. – CANMsgUpdateFrom11_18 and CANMsgUpdateFrom29 functions for synchronizing the identifier syntax. – Added functions for creation of identifiers for some high-level protocols (e.g. CANMsgSAE).