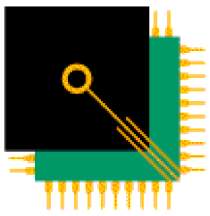
CANopen Magic Pro DLL
User Manual

Manual Revision 1.20

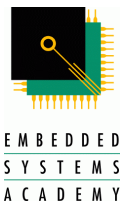


EMBEDDED
SYSTEMS
ACADEMY

For support contact support@esacademy.com

For the latest news on CANopen Magic Pro DLL visit us on the web at

www.esacademy.com



Embedded Systems Academy provides training and consulting services, specializing in CAN, CANopen and Embedded Internetworking. For more information visit

www.esacademy.com

Contents

About This Manual

This manual follows some set conventions with the aim of making it easier to read. The following conventions are used:

0x	Hexadecimal (base 16) values are prefixed with "0x".
italictext	Replace the text with the item it represents
[ ]	Items inside [and] are optional
a | b	a OR b may be used
…	One or more items may go here.

This manual frequently uses CANopen terminology as defined by the CANopen standard DS301 (see www.can-cia.org for more info). Readers that are not yet familiar with all the CANopen terms may want to consider reading a book like www.canopenbook.com or the official standard to update their knowledge on CANopen technology and terminology.

Chapter 1 – Introduction

1.1 About CANopen

CANopen is a higher layer protocol that runs on a CAN network. The CAN specification defines only the physical and data link layers in the ISO/OSI 7-layer Reference Model. This means that only the physical bus and the CAN message format is defined, but not how the CAN messages should be used. CANopen provides an open and standarized but customizable description of how to transfer data of different types between different CAN nodes. This allows off the shelf CANopen compliant nodes to be purchased and plugged into a network with the minimum of effort. It also can be used in place of an in-house proprietory higher layer protocol development.

The development of CANopen is supervised by the CAN in Automation User's Group and is being turned into an international standard. Use of CANopen does not require the payment of any royalties and the specification may be expanded or altered to suit if closed networks are being developed.

Typical applications for CANopen include:

- Commercial Vehicles
- Medical Equipment
- Maritime Electronics
- Building Automation
- Light Rail Systems

1.2 About the CANopen Magic Pro DLL

The CANopen Magic Pro DLL provides the necessary information and files to allow custom applications to be built that use CANopen functionality.

The functionality of the package is provided by a DLL. This DLL is rather like a library and can be called by any application that knows how to use it. All copies of applications built on this platform include this DLL. By building upon this DLL an application immediately gains access to the knowledge of CANopen that have been built up over several years of effort. The DLL is a tried and tested platform that is currently used by thousands of users worldwide.

This manual assumes familiarity with the features of CANopen. A description of the features will not be reproduced here. Instead, please refer to the relevent CAN in Automation specifications or the Embedded Networking with CAN and CANopen (www.canopenbook.com) book.

Familiarity with a C or C++ development system is also assumed. This manual does not describe any features that relate to development systems. Instead please refer to the manual or help that came with your development system.

It is recommended to read this manual completely before starting on any development work.

1.3 Package Overview

Contents

The package contains the following:

- The CANopen DLL
- The C header file for the DLL
- The necessary library files for the DLL
- An example application
- This manual that describes how to use the DLL

Features

The following are features of the CANopen DLL:

- Send Network Management messages to single nodes or all nodes
- Perform an SDO Download to the Object Dictionary of a node
  - Expedited and segmented transfers supported
- Perform an SDO Upload from the Object Dictionary of a node
  - Expedited and segmented transfers supported
- Progress callback during SDO transfers giving progress of transfer
- Option to cancel an SDO transfer in progress
- High speed network scan
  - Finds all CANopen nodes on the network in less than 0.5 seconds
- High speed mass expedited writing
- Configures the CAN interface for any standard CANopen baud rate
- All received messages have a high precision timestamp
- Transmit and receive callback functions
- Major error callback function
- Change baud rate on the fly
- LSS support
- Supports block transfers
- Able to receive error frames
- Supports CiA 447 Car Add-on Devices
- Supports multi process access to a single CAN interface
- Write Device Configuration Files to nodes
  - Allows configuration of nodes
- Write Network Configuration Files to the network
  - Allows configuration of all nodes at once
- Transmit and receive plain CAN messages
  - CAN 2.0B and RTRs supported

- Can be used to send and receive messages at the same time as other ESACANopenPro.DLL tools are running
  - PCANopen Magic Pro can show a trace of the CAN bus during development of applications using the DLL
- Supports Windows 2000/XP/Vista and Windows CE 5.0 (see limitations)

Limitations

Windows 2000/XP/Vista Limitations

When using a PEAK CAN interface, once in a while the timestamp for a received message or the current time might be incorrect by as much as 3 seconds.

If CANopenDLL_Startup has been called and the user then changes the CAN interface type in the control panel (2000/XP/Vista), the list of available hardware interfaces returned from the DLL will not change to reflect the newly selected CAN inteface type. In this situation, ensure any connections to networks are closed and call:

```
CANopenDLL_Shutdown();
CANopenDLL_Startup();
```

The list of hardware interfaces returned by the DLL will now use the new interface type.

Windows CE 5.0 Limitations

The DLL is compiled for ARMV4I processors only.
A DLL with a suitable API is required for the CAN driver being used.
The multi-process option is not supported.

This manual contains all the information needed to use the DLL. If you have questions, please contact us at **support@esacademy.com**

1.4 Obtaining Compatible CAN Interfaces

As mentioned in the feature list, all PEAK-System Technik CAN interfaces are supported. Visit www.peak-system.com to locate the nearest distributor. Also supported are Emtrion PCI CAN interfaces.

On Windows CE 5.0 any CAN interface is supported providing a DLL is written to access the CAN interface and the DLL has a suitable API. Details of the API are provided in this manual.

Chapter 2 – Installation

2.1 Installation

Minimum Requirements

The following is a list of the recommended minimum requirements for installing and use the package.

- Windows 2000/XP/Vista
- 3Mb of disk space
- A C or C++ Development system, such as Microsoft Visual Studio 2005 or Borland C++ Builder 5/6
- A PEAK CAN interface (Windows 2000/XP/Vista) or a Windows CE 5.0 development system with a CAN driver

Installation Procedure

Installation is very simple. Simply run the installation executable and follow the prompts. Once installed, access to this manual, and the folders for the files and example are available from the Start Menu. You will also need to install hardware drivers by following the instructions from the hardware vendor, and the PEAK CAN Driver. See the sections below for a description of these steps.

Install PEAK CAN Driver

The PEAK CAN Driver must be installed before the package may be used. Normally it will be installed automatically at the end of the CANopen Magic Pro DLL installation. If the installation file for the driver is not found, then you will be prompted for it's location.

Additional Step For PEAK PCAN Dongle Users

Before using the PEAK PCAN Dongle interface with the CANopen Magic Pro DLL, it must be removed from the setup window of the PCANView Dongle software. To do this complete the following steps:

- Start PCANView Dongle from the Start Menu
- Select the PCAN Dongle in the Available CAN Hardware section
- Click on "Delete"
- Click on "OK"
- Close PCANView Dongle

Chapter 3 – Using the DLL

3.1 Overview

The DLL implements a set of functions which together provide CANopen functionality. The following table lists the functions and what they do.

Function	Description
CANopen_NMT	Sends a Network Management message
CANopen_SDODownload	Starts an SDO download
CANopen_SDOUpload	Starts an SDO upload
CANopen_Cancel	Cancels an SDO download or upload
CANopen_ScanNetwork	Scans the network for CANopen nodes
CANopen_MassExpeditedWrite	Performs high speed expedited write to all nodes at once.
CANopen_SetSDOTimeout	Sets the timeout to use for SDO operations.
CANopen_SetScanMassOperationDelay	Sets a delay used during the network scan and mass expedited writes to slow them down.
CANopen_FindLSSSlave	Finds an LSS slave on the network
CANopen_SetLSSSlaveConfig	Sets the configuration of an LSS slave
CANopen_SetLSSSlaveBitTiming	Sets the bit timing of an LSS slave
CANopen_UseLSSSlaveBitTiming	Instructs all LSS slaves to use bit timings
CANopen_SetLSSTimings	Sets the timing information for the LSS protocol
CANopen_SDOChannels	Enables/disables SDO channel requesting
CANopen_SDOChannelsTimeout	Sets the timeout for SDO channel requesting
CANopen_SetSDOConfig	Sets the SDO transfer configuration
CANopen_SetMode	Sets the CANopen operating mode
CANopen_SetBlockSegmentWriteDelay	Sets the delay after each SDO block is written to control write speed
Hardware_GetCurrentTime	Gets the current time in timestamp format
Hardware_EnumerateHardware	Lists available CAN interfaces
Hardware_EnumerateNetworks	Lists available CAN networks
Hardware_AddNetwork	Adds a new network
Hardware_DeleteNetwork	Deletes a network
Hardware_GetBaudrate	Gets the current baudrate of a network
Hardware_Initialize	Initializes a CAN interface
Hardware_Close	Finishes with a CAN interface
Hardware_SwitchNetworks	Changes the baud rate on the fly
Hardware_Reset	Resets the CAN interface
Hardware_ErrorFrames	Turns on or off error frame reception
Hardware_SelfReceive	Turns on and off self receive
Hardware_IsNetworkFunctional	Checks if the current network is functional
Event_Transmit	Registers a transmit callback function
Event_Receive	Registers a receive callback function
Event_MajorError	Registers a major error callback function
CAN_Transmit	Transmits a plain CAN message
CANopenDLL_Startup	Initializes the DLL

CANopenDLL_Shutdown	Finishes with the DLL
CANopenConfig_WriteDCF	Writes a DCF to a node
CANopenConfig_WriteNCF	Writes a NCF to the network
CANopenServer_Startup	Starts a minimal CANopen server
CANopenServer_Shutdown	Stops the minimal CANopen server
MicroLSS_ScanAndConfig	Scans for and configures MicroLSS slaves

The rest of this chapter describes how the functions are used. The function reference chapter lists each function in detail.

3.2 Adding to a Project

Microsoft Visual C++

To use the DLL in a project:

- Make copies of the .lib file and .h file for your project
- Add the copied .lib file to the project
- Include the header file in any files that will call CANopen functions
- Copy the .dll file into the same folder as the executable

Ensure the correct .lib file is used. You must use the one from the MSVisualStudio2005 folder. Using the Borland .lib file will not work.

You must distribute the DLL with your application. Do not distribute this documentation, the .lib or .h files.

Borland C++ Builder

To use the DLL in a project:

- Make copies of the .lib file and .h file for your project
- Add the copied .lib file to the project
- Include the header file in any files that will call DLL functions
- Copy the .dll file into the same folder as the executable

Ensure the correct .lib file is used. You must use the one from the BorlandC++Builder5 folder. Using the Microsoft .lib file will not work.

You must distribute the DLL with your application. Do not distribute this documentation, the .lib or .h files.

3.3 Calling Functions

Return Values

The RESULTS type is used for return values from API functions. It is defined as:

```
typedef struct
{
  int code;
  wchar_t details[ESACAN_MAXDETAILSLEN];
} RESULTS;
```

The code indicates either success or a specific error. Depending on the error details may contain a string describing the error.

The code may be one of:

- OK
- ERR_USERCANCELLED
- ERR_INVALIDPARAM
- ERR_PROTCOL
- ERR_HWINIT
- ERR_BUS
- ERR_TIMEOUT
- ERR_UNSUPPORTED

Other Types

The ESACAN_TIMESTAMP type is used to hold a timestamp. Timestamps are used for such things as the current time or the time a message was received. It is defined as:

```
typedef struct
{
  unsigned long millis;
  unsigned int  millis_overflow;
  unsigned int  micros;
} ESACAN_TIMESTAMP;
```

The time is given in milliseconds with fractional microseconds.

The ESACAN_MSG type is used to hold a description of a single CAN message. It is defined as:

```
typedef struct
{
  ESACAN_TIMESTAMP timestamp;
  unsigned int id;
  unsigned char dlc;
  unsigned char flags;
  unsigned char data[8];
} ESACAN_MSG;
```

flags contains a combination of one or more of the following flags:

ESACAN_MSG_EXT - 29-bit identifer

ESACAN_MSG_RTR - RTR flag was set
ESACAN_MSG_ERRFRAME - message is an error frame

The ESACAN_HARDWARE type describes a hardware interface. It is defined as:

```
typedef struct
{
  wchar_t name[ESACAN_MAXDRIVERNAMELEN];
  int handle;
} ESACAN_HARDWARE;
```

name holds the name of the hardware interface. Handle holds a unique handle to the interface.

The ESACAN_NETWORK type describes a network, which is associated with a specific hardware interface. It is defined as:

```
typedef struct
{
  wchar_t name[ESACAN_MAXNETWORKNAMELEN];
  int handle;
  int baudrate;
} ESACAN_NETWORK;
```

The name holds the name of the network. The handle holds a unique handle to the network. The baudrate holds the speed of the network in kbps.

The ESACAN_NODEINFO type describes basic information about a node. It is defined as:

```
typedef struct
{
  unsigned char status;
  unsigned long devicetype;
} ESACAN_NODEINFO;
```

The status indicates if the node has been found on the bus or not or written to or not. The device type holds the value read from Index 1000H, subindex 00H. The status may have one of the following values:

ESACAN_NOTFOUND - node was not found. Ignore devicetype.
ESACAN_FOUND - node was found. Read devicetype.
ESACAN_NOTWRITTEN - node was not written to.
ESACAN_WRITTEN - node was written to.

Callback Functions

The PROGRESS_CALLBACK is a function pointer type with the following prototype:

```
void (__stdcall *PROGRESS_CALLBACK)(float percentage, unsigned long callbackparam);
```

called during operations such as SDO download, passed is the percentage of the operation that is completed and a user defined parameter. Used for providing feedback to the user.

The FINISHED_CALLBACK is a function pointer type with the following prototype:

```
void (__stdcall *FINISHED_CALLBACK)(RESULTS *results, unsigned long callbackparam);
```

called when an operation such as SDO download is complete. Passed is a RESULTS type containing the result of the operation and a user defined parameter. The results code is one of:

- OK
- ERR_PROTOCOL
- ERR_USERCANCELLED

The MESSAGE_CALLBACK is a fucntion pointer type with the following prototype:

```
void (__stdcall *MESSAGE_CALLBACK)(wchar_t *msg, unsigned long callbackparam);
```

called when an operation needs to provide status messages. For example when writing a DCF to a node this callback function will be called to indicate specific problems encountered. Also passed is a user defined parameter.

The MAJORERROR_CALLBACK is a function pointer type with the following prototype:

```
void (__stdcall *MAJORERROR_CALLBACK)(int error);
```

called when a major error has occurred. Passed is an error code. One of:

- MERR_NOERROR no error
- MERR_BUSOFF bus off
- MERR_OVERRUN controller rx buffer overrun

This function is only called when the major error changes.

The RECEIVE_CALLBACK is a function pointer type with the following prototype:

```
typedef void (__stdcall *RECEIVE_CALLBACK)(ESACAN_MSG *msg, int reply);
```

This function is called whenever a message is received. Note that the DLL receives it's own messages, so all transmitted messages will cause this function to be called.
Reply is zero unless the message is an SDO response from a node, in which case reply is one.

The TRANSMIT_CALLBACK is a function pointer type with the following prototype:

```
typedef void (__stdcall *TRANSMIT_CALLBACK)(ESACAN_MSG *msg);
```

This function is called whenever a message is transmitted by the DLL. The timestamp is not used in the copy of the message that is returned.

It is recommended that callback functions execute as quickly as possible to avoid causing performance problems for the DLL.

The SWITCHNETWORKS_CALLBACK is a function pointer type with the following prototype:

```
typedef int (__stdcall *SWITCHNETWORKS_CALLBACK)(int newnethandle, long pause);
```

This function is called when the LSS slaves on the bus are switching to a new bit timing. It is called when the application itself needs to switch baud rates. newnethandle is the handle of the network the application should use, and pause is the delay in milliseconds after switching networks before the application can transmit more messages. This function should return immediately and not wait for pause milliseconds to pass before returning.

Typical Call Flow

The following is a typical sequence of function calls.

Normally an applicaton will call the functions in the following order:

- CANopenDLL_Startup
  - start the DLL
- Event_Receive
- Event_Transmit
- Event_MajorError
  - initialize callback functions
- Hardware_EnumerateHardware
  - allow the user to choose from the available list of hardware
- Hardware_EnumerateNetworks
  - allow the user to choose frrom the available list of networks for the selected hardware.
- Hardware_AddNetwork
- Hardware_DeleteNetwork
  - allow the user to create new networks and delete old networks
- Hardware_Initialize
  - connect the application to the selected hardware and network
- Hardware_GetBaudrate
  - get baudrate being used by the application
- Hardware_GetCurrentTime
- CANopen_SetSDOConfig
  - Configure the SDO transfers
- If using LSS, call the LSS functions in the order described below
- If requesting SDO channels, use the functions in the order described below
- CAN_Transmit
- CANopen_NMT
- CANopen_Cancel

- CANopen_SDODownload
- CANopen_SDOUpload
- CANopen_ScanNetwork
    - perform operations on the CAN bus
- Hardware_Close
    - disconnect the application from the network
- CANopenDLL_Shutdown
    - finish using the DLL

The following is a typical sequence of function calls when using LSS:

- CANopen_SetLSSTimings
- CANopen_FindLSSSlave
    - To discover a single LSS slave
- CANopen_SetLSSSlaveBitTiming
    - Call for each slave on the network
- CANopen_UseLSSSlaveBitTiming
    - In the SwitchNetworksFunc callback function call Hardware_SwitchNetworks
- CANopen_SetLSSSlaveConfig
    - Call for each slave on the network

The following is a typical sequence of function calls when requesting SDO channels:

- CANopenServer_Start
- CANopen_SDOChannels
    - To enable requesting of SDO channels
- CANopen_SDOChannelsTimeout

3.4 Threads

Functions which do not have progress and finished callback functions passed as parameters execute in the same thread as the function caller.

Functions which do have progress and finished callback functions as parameters are executed in a separate thread. These are usually SDO operations which may take some time to complete. By executing in a separate thread, the user interface can remain responsive rather than freezing up.

Only one function may be called at any one time in a single instance of the DLL. The single exception is that CANopen_Cancel may be called while an SDO upload, download or network scan is in progress. Normally CANopen_Cancel is called from a progress callback function.

Multiple copies of the DLL may be loaded at any one time, allowing multiple parallel operations to take place on the CAN bus. For example, when using the DLL and running PCANopen Magic Pro at the same time this situation is taking place.

All callback functions are executed in a separate thread that is internal to the DLL. Therefore the usual limitations and precautions apply when using data in a callback function.

3.5 Description

Overview

In general terms, each application using the DLL goes through the following steps:

- Start up DLL
- Configure hardware
- Register callback functions
- Use CAN bus
- Shut down DLL

Because a PC may have multiple CAN interfaces connected at once, and each interface may have the option of connecting to a range of CAN networks with different speeds, the hardware configuration may seem confusing at first.

Once an application has finished with the DLL, the hardware must be closed and the DLL shut down. Failure to do so may cause memory leaks.

Start Up

The function CANopenDLL_Startup is called to start the DLL. No other functions in the DLL may be called until after this function has been called.

The DLL may be used to allow a single process to access a CAN interface. This is the standard arrangement. However, the DLL also supports multi-process access, where multiple processes each using a copy of the DLL can talk to each other via a simulated CAN bus internal to the PC, and also optionally a CAN interface. See the description of CANopenDLL_Startup for more information.

Hardware Configuration

First the CAN interface must be selected. To present the user with a list, Hardware_EnumerateHardware is called. This will return a list of hardware currently found on the PC. For PEAK interfaces the list will be limited to a type of CAN interface, for example plug and play. To change the type, using the CAN-Hardware Control Panel applet. Once changed, Hardware_EnumerateHardware will then return a different list based on the new type.

Once the user has selected a hardware interface to use, the handle to the interface is passed to Hardware_EnumerateNetworks to obtain a list of currently defined networks for that interface. The user then selects the network they wish to use. If a network at the speed desired is not present, then one can be added using Hardware_AddNetwork. Also the application can delete networks using Hardware_DeleteNetwork.
Alternatively, the PEAK tool PCAN Netconfig (available on the Start menu after installing the PEAK CAN Driver) can be used to add and delete networks if using PEAK interfaces.

The next step is to select the specific hardware interface and network to use. This is achieved by calling Hardware_Initialize.

Callback Configuration

If you wish your application to be informed of certain events, then the callback functions must be implemented and registered. Registration is performed by calling the Event_xxxxx functions. Callback functions may be registered or unregistered at any time.

Callback functions must execute as quickly as possible. They execute in a thread internal to the DLL, therefore the use of messages, signals, mutexs etc. is required to pass data to the rest of your application.

All messages transmitted by the DLL are also received by the DLL. Therefore by registering a receive callback function, it is possible to obtain the timestamp of when a message was transmitted by the DLL.

Some callback functions receive a user defined parameter. This is the exact same value that is passed to the DLL when the operation involving the callbacks is started. For example if an SDO download is started and the callback parameter is set to the value 5, then when the progress and finished callback functions are called, the parameter will have the value 5. This is useful for passing class instances to ensure that the callback function knows which class instance is performing the operation.

Callback functions use the __stdcall calling convention. Check your compiler documention on how to define functions to use this calling convention.

CAN Bus Operations

SDO uploads and downloads may be performed by calling CANopen_SDOUpload and CANopen_SDODownload. Only one operation may be performed at any one time. Callback functions notify your application of the progress and when the operation has finished. CANopen_Cancel may be called to cancel the SDO transfer.

A high speed network scan may be performed by calling CANopen_ScanNetwork. In order for a node to be detected by the scan, it must implement Object Dictionary entry [1000,00], which is mandatory for all CANopen nodes. The scan may be cancelled by calling CANopen_Cancel.

Plain CAN messages may be transmitted by calling CAN_Transmit, and network management messages may be transmitted by calling CANopen_NMT.

Node and network configuration may be performed by calling CANopenConfig_WriteDCF and CANopenConfig_WriteNCF. The configuration operations may be cancelled by calling CANopen_Cancel.

Shut Down

Once the DLL is no longer needed, Hardware_Close must be called followed by CANopenDLL_Shutdown. Once CANopenDLL_Shutdown has been called the only DLL function that may be called is CANopenDLL_Startup. No other functions may be called.

3.6 Distribution

To distribute an application based on the DLL in the package you need to do the following:

- Tell users to install the driver for their CAN interface. This comes on a CD or floppy disk or can be downloaded from www.peak-system.com.
- If using PEAK: install your copy of the PEAK CAN Driver on the user's PC. This came with your copy of this package and usually has the name PCANDrv.exe. Note that this driver is linked to you by a serial number issued by PEAK.
- If using Emtrion: include HiCANPCI.dll with your application executable, preferably in the same folder as your application executable.
- Include ESACANopenPro.dll with your application executable, preferably in the same folder as your application executable.
- If using Windows CE: include the driver wrapper DLL with your application executable.

You must not under any circumstances distribute the following:

- **Any .h, .lib, .def or .exp files provided with this package**
- **This manual or the contents of this manual, in any format**
- **Any applications that allow other custom CANopen software to be developed using the ESACANopenPro.dll file.**
- **Any source code showing the use of the ESACANopenPro.dll file.**

Doing so will render your license to use this product invalid.

This product includes a copy if the CANAPI.DLL by PEAK System Technik. This DLL may only be distributed freely with products generated with this package if not used directly by the application program. Developers that wish to use the CANAPI.DLL directly must purchase PCAN-Developer or PCAN-Evaluation from PEAK System Technik.

Chapter 4 – Function Reference

4.1 CANopenDLL_Startup

Prototype:

```
void CANopenDLL_Startup(int mode, wchar_t *drivername);
```

Params:

mode = ESACAN_SINGLEPROCESS or ESACAN_MULTIPROCESS
drivername = name of driver to use. Examples:

"CanApi2.dll" = PEAK CAN interfaces driver
"HICANPCI.dll" = Emtrion PCI CAN interfaces driver

Desc:

Starts up the DLL. Must be called before any other function in the DLL. There are three configurations possible:

One process using CAN interface:
Pass ESACAN_SINGLEPROCESS and the driver name.
Multiple processes using same CAN interface:
Pass ESACAN_MULTIPROCESS and the driver name. The driver name must be the same for all processes.
Multiple processes, no CAN interface (simulation only):
Pass ESACAN_MULTIPROCESS, "" for the driver name. The driver name must be the same for all processes.

In order for the multi-process system to work, ESACANServer.exe must be first copied to the same folder as ESACANopenPro.DLL.

Returns:

Nothing

4.2 CANopenDLL_Shutdown

Prototype:

```
void CANopenDLL_Shutdown(void);
```

Params:

None

Desc:

Shuts down the DLL when the application has finished using it. Must be the last function called.

Returns:

Nothing

4.3 Event_Transmit

Prototype:

```
void Event_Transmit(TRANSMIT_CALLBACK TransmitFunc);
```

Params:

A transmit callback function

Desc:

Registers a callback function to be called when the DLL transmits a single CAN message. Passing a null pointer unregisters the callback function. Once registered, the callback function will be called for each message transmitted by the DLL.

Returns:

Nothing

4.4 Event_Receive

Prototype:

```
void Event_Receive(RECEIVE_CALLBACK ReceiveFunc);
```

Params:

Callback function

Desc:

Registers a callback function to be called when the DLL receives a single CAN message. Passing a null pointer unregisters the callback function. Once registered, the callback function will be called for each message received by the DLL, including messages transmitted by the DLL.

Returns:

Nothing

4.5 Event_MajorError

Prototype:

```
void Event_MajorError(MAJORERROR_CALLBACK MajorErrorFunc);
```

Params:

Callback function

Desc:

Registers a callback function to be called when the DLL detects a major error. Passing a null pointer unregisters the callback function.

Returns:

Nothing

4.6 Hardware_GetCurrentTime

Prototype:

```
void Hardware_GetCurrentTime(ESACAN_TIMESTAMP *timestamp);
```

Params:

timestamp = pointer to place to receive timestamp for the current moment in time.

Desc:

Timestamps the current moment in time and returns the timestamp.

Returns:

Nothing

4.7 Hardware_EnumerateHardware

Prototype:

```
int Hardware_EnumerateHardware(ESACAN_HARDWARE *hwlist, int hwlistsize);
```

Params:

hwlist = pointer to array to receive hardware descriptions
hwlistsize = number of hardware descriptions that can fit into the hwlist array

Desc:

Returns a list of current CAN hardware interfaces. Note that the list only contains the interfaces in the currently selected category (Plug-n-play, USB, etc.). The currently selected category may be changed via the Control Panel. Also returns a hardware interface with the name "None", which can be used to create internal only networks (i.e. no CAN interface).

Returns:

Number of CAN interfaces found and stored in hwlist.

4.8 Hardware_EnumerateNetworks

Prototype:

```
int Hardware_EnumerateNetworks(ESACAN_NETWORK *netlist, int netlistsize, int hwhandle);
```

Params:

netlist = pointer to array to receive network descriptions
netlistsize = number of network descriptions that can fit into the netlist array.
hwhandle = handle of the CAN hardware interface whose networks should be listed.

Desc:

Returns the currently defined networks for a specific CAN interface. Pass the CAN interface the user has selected and the available networks including their baudrates will be returned.

Returns:

Number of networks found and stored in netlist.

4.9 Hardware_AddNetwork

Prototype:

```
int Hardware_AddNetwork(wchar_t *name, int baudrate, int hwhandle);
```

Params:

name = name of the network to add
baudrate = bus speed of the network in kbps
hwhandle = handle to the CAN interface the network is connected to

Desc:

Creates a new network and associates it with a specific CAN interface. The network details will be stored in the registry so the network can be automatically added to the current list the next time the DLL is used.

Returns:

Handle to the new network.

4.10 Hardware_DeleteNetwork

Prototype:

```
void Hardware_DeleteNetwork(int nethandle);
```

Params:

Nethandle = handle to the network to delete

Desc:

Deletes a network. Also removes the network details from the registry so the network will not be available next time the DLL is used.

Returns:

Nothing

4.11 Hardware_GetBaudrate

Prototype:

```
int Hardware_GetBaudrate(int nethandle);
```

Params:

Nethandle = handle of the network whose baudrate is needed

Desc:

Returns the baudrate of a network in kbps

Returns:

The baudrate of the network in kbps

4.12 Hardware_Initialize

Prototype:

```
void Hardware_Initialize(int nethandle, wchar_t *appname, HWND
hWnd, RESULTS *preturnresult);
```

Params:

nethandle = handle to the network to use.
appname = the name of the application
hWnd = handle to the application's main window
preturnresult = pointer to the results of the initialization

Desc:

Connects the application to a specific network. Once connected the application can then send and receive, which will automatically use the network passed to this function.

Returns:

Nothing

4.13 Hardware_Close

Prototype:

```
void Hardware_Close(void)
```

Params:

none

Desc:

Disconnects the application from the network.

Returns:

Nothing

4.14 Hardware_SwitchNetworks

Prototype:

```
void Hardware_SwitchNetworks(int operation, int nethandle,
RESULTS *preturnresult)
```

Params:

operation = the operation to perform (ESACAN_SN_xxx).
nethandle = handle to the network to switch to. The network must be associated with the current CAN interface in use.
preturnresult = pointer to the results of the switch

Desc:

Switches to a new network for the same CAN interface that is currently in use. The switch is performed on the fly so it is not necessary to close or reinitialize the hardware. The switch is performed in two steps. First the function is called with operation set to ESACAN_SN_DISCONNECT. This will

disconnect the node from the current network. Next the function is called with operation set to ESACAN_SN_CONNECT, which will connect the node to the new network.

The reason why this feature is performed in two steps is because it is not possible to switch from one network to another using the same CAN interface if there are any nodes still connected to the old network while the current node is trying to connect to the new network. If there are multiple nodes on the network then they must all call this function to disconnect first before any of them call this function to connect.

Returns:

Nothing

4.15 Hardware_Reset

Prototype:

```
void Hardware_Reset(void);
```

Params:

none

Desc:

Resets the CAN interface.

Returns:

Nothing

4.16 Hardware_ErrorFrames

Prototype:

```
void Hardware_ErrorFrames(int mode);
```

Params:

mode can be either ESACAN_IGNOREERRORFRAMES or ESACAN_RECEIVEERRORFRAMES.

Desc:

Toggles whether error frames should be received or not. The default setting when starting the DLL is that they are not received.

4.17 Hardware_SelfReceive

Prototype:

```
void Hardware_SelfReceive(int mode, RESULTS *preturnresult);
```

Params:

mode = self receive mode to use. Can be either ESACAN_NOSELFRECEIVE or ESACAN_SELFRECEIVE
preturnresult = pointer to the results of the initialization

Desc:

Toggles whether self receive mode is turned on or not in the CAN interface. The default settings when the DLL is started is that self receive is turned on. The results code will be ERR_UNSUPPORTED if self receive on or off is not supported in the CAN interface in use.

Returns:

Nothing

4.18 Hardware_IsNetworkFunctional

Prototype:

```
int Hardware_IsNetworkFunctional(void);
```

Params:

none

Desc:

Checks if the current network in use is functional, i.e. can send and receive CAN messages, whether is it via a simulated CAN bus or a CAN interface.

Returns:

1 if the network is functional, 0 if the network is not functional.

4.19 CANopen_SDODownload

Prototype:

```
void CANopen_SDODownload(unsigned char nodeid, unsigned int index, unsigned char subindex, unsigned char *buffer, unsigned long size, PROGRESS_CALLBACK ProgressFunc, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *presult);
```

Params:

nodeid = ID of node to send data to.
index = index of OD entry to send data to
subindex = subindex of OD entry to send data to
buffer = pointer to buffer that holds the data
size = number of bytes in buffer to send to node
ProgressFunc = function to call to indicate progress of operation. Null pointer if not required.
FinishedFunc = function to call when operation is finished. Null pointer if not required.
callbackparam = value passed to callback functions

Desc:

Starts an SDO download. Returns immediately. When the operation is completed or fails the FinishedFunc function will be called. The ProgressFunc function will be periodically called to indicate progress of the operation. This function will start a thread in which the download will be performed. Operation may be cancelled by calling the CANopen_Cancel function.

Returns in results code:

OK

4.20 CANopen_SDOUpload

Prototype:

```
void CANopen_SDOUpload(unsigned char nodeid, unsigned int index, unsigned char subindex, unsigned char *buffer, unsigned long *size, PROGRESS_CALLBACK ProgressFunc, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *presult);
```

Params:

nodeid = ID of node to receive data to.
index = index of OD entry to receive data to
subindex = subindex of OD entry to receive data to
buffer = pointer to buffer to receive the data
size = pointer to value that contains the size of the buffer in bytes
ProgressFunc = function to call to indicate progress of operation. Null pointer if not required.
FinishedFunc = function to call when operation is finished. Null pointer if not required.

callbackparam = value passed to callback functions

Desc:

Starts an SDO upload. Returns immediately. When the operation is completed or fails the FinishedFunc function will be called. The ProgressFunc function will be periodically called to indicate progress of the operation. This function will start a thread in which the upload will be performed. Once complete size will hold the number of bytes read from the node.
Operation may be cancelled by calling the CANopen_Cancel function.

Returns in results code:

OK

4.21 CANopen_Cancel

Prototype:

```
void CANopen_Cancel(void);
```

Params:

None

Desc:

Call to indicate the current operation (upload, etc.) should be cancelled at the first opportunity. The finished function will return ERR_USERCANCELLED in the results code.

Returns:

Nothing

4.22 CANopen_ScanNetwork

Prototype:

```
void CANopen_ScanNetwork(ESACAN_NODEINFO *pnodeinfo,
FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS
*presult);
```

Params:

pnodeinfo = pointer to an array of 127 ESACAN_NODEINFO structures. Once the network scan is complete, the array will hold the results of the scan.
FinishedFunc = pointer to the callback function that will be called when the network scan is complete.
callbackparam = value passed to callback function

Desc:

Scans the network and determines which nodes are present and their device type. Entry 0 in the array contains the details for node 1, entry 126 in the array contains the details for node 127.

Returns:

Nothing

4.23 CANopen_MassExpeditedWrite

Prototype:

```
void CANopen_MassExpeditedWrite(unsigned int index, unsigned char subindex, unsigned char *buffer, unsigned long size, ESACAN_NODEINFO *pnodeinfo, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *preturnresult);
```

Params:

index = index of OD entry to send data to
subindex = subindex of OD entry to send data to
buffer = pointer to buffer that holds the data
size = number of bytes in buffer to send to each node
pnodeinfo = pointer to an array of 127 ESACAN_NODEINFO structures. Once the write is complete, the array will hold the results of the write (written or not written).
FinishedFunc = pointer to the callback function that will be called when the write is complete.
callbackparam = value passed to callback function

Desc:

Attempts to perform the same expedited write to every node as fast as possible (typically the SDO timeout period). The data size must be four bytes or less. Results of which nodes were written to is placed in the pnodeinfo array.

Returns:

Nothing

4.24 CANopen_SetSDOTimeout

Prototype:

```
void CANopen_SetSDOTimeout(long timeout);
```

Params:

timeout = SDO protocol timeout in milliseconds

Desc:

Sets a new SDO timeout to use for SDO reads and writes.

Returns:

Nothing

4.25 CANopen_SetScanMassOperationDelay

Prototype:

```
void CANopen_SetScanMassOperationDelay(long delay);
```

Params:

delay = delay to use in milliseconds

Desc:

Used to slow down the network scan and mass expedited writes by inserting a delay after every read or write access.

Returns:

Nothing

4.26 CANopen_FindLSSSlave

Prototype:

```
void CANopen_FindLSSSlave(ESACAN_LSSSLAVE *plssslave, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *preturnresult);
```

Params:

plssslave = pointer to buffer to receive a description of the LSS slave found.
FinishedFunc = pointer to a callback function that is called when the search is complete.
preturnresult = pointer to buffer to receive results of calling the function.
callbackparam = value passed to callback function

Desc:

Finds a single LSS slave on the network and returns information about the slave. When calling this function there must be only one LSS slave on the network. The ESACAN_LSSSLAVE type is filled with the information about the found LSS slave, if one was found.

Returns:

Nothing

4.27 CANopen_SetLSSSlaveConfig

Prototype:

```
void CANopen_SetLSSSlaveConfig(ESACAN_LSSSLAVE *plssslave, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *preturnresult);
```

Params:

plssslave = description of LSS slave whose configuration is to be set along with the configuration to set
FinishedFunc = pointer to a callback function that is called when the operation has completed.
preturnresult = pointer to buffer to receive results of calling function.
callbackparam = value passed to callback function

Desc:

Configures the node ID of a specific LSS slave on the network and optionally instructs the slave to store the configuration. A description of which LSS slave to configure is passed along with the ID to use. To send the store command set storeconfguration to 1, otherwise set it to 0. The necessary commands will be sent to configure the node ID and store the configuration. If successful the

node will boot up into Pre-operational mode using the node ID. Can be used when there are other LSS slaves on the network.

Returns:

Nothing

4.28 CANopen_SetLSSSlaveBitTiming

Prototype:

```
void CANopen_SetLSSSlaveBitTiming(ESACAN_LSSSLAVE *plssslave, unsigned char tableselector, unsigned char tableindex, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *preturnresult);
```

Params:

plssslave = description of lss slave whose bit timing is to be set.
tableselector = the bit timing table to use. See the LSS specification for details.
tableindex = the bit timing index to use. See the LSS specification for details.
FinishedFunc = pointer to callback function that is called when the operation has completed.
returnresult = pointer to buffer to receive results of calling the function.
callbackparam = value passed to callback function

Desc:

Configures the bit timing of a specific LSS slave on the network. Note that this function does not tell the slave to use the bit timing, only what the new bit timing will be. Can be used when there are other LSS slaves on the network. To activate the bit timing call CANopen_UseLSSSlaveBitTiming.

Returns:

Nothing

4.29 CANopen_UseLSSSlaveBitTiming

Prototype:

```
void __cdecl CANopen_UseLSSSlaveBitTiming(int nethandle, SWITCHNETWORKS_CALLBACK SwitchNetworksFunc, FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS *preturnresult);
```

Params:

nethandle = the handle of the network that is being switched to.
SwitchNetworksFunc = pointer to callback function that instructs the application to switch networks.
FinishedFunc = pointer to callback function that is called when the operation has completed.
preturnresult = pointer to buffer to receive results of calling function.
callbackparam = value passed to callback function

Desc:

Instructs all the LSS slaves to switch to the new bit timing that was previously programmed by calling CANopen_SetLSSSlaveBitTiming. At the appropriate

point the SwitchNetworksFunc will be called to tell the application to switch to the new network, passing nethandle as a parameter.

Returns:

Nothing

4.30 CANopen_SetLSSTimings

Prototype:

```
void CANopen_SetLSSTimings(long responsetimeout, long cmdtime, long switchdelay);
```

Params:

responsetimeout = time to wait for responses from LSS slaves in milliseconds.
cmdtime = time to wait for an LSS slave to process an unconfirmed command before sending the next command, in milliseconds.
switchdelay = time to wait before an LSS slave should switch to new bit timing, and the time to wait after switching before transmitting any messages, in milliseconds.

Desc:

Sets the timing parameters for the LSS protocol. If this function is not called then default settings are used which are: 300ms for responsetimeout, 30ms for cmdtime and 300ms for switchdelay.

Returns:

Nothing

4.31 CANopen_SDOChannels

Prototype:

```
void CANopen_SDOChannels(int mode);
```

Params:

mode = SDO channel mode to use. Can be either ESACAN_DONTREQUESTCHANNELS or ESACAN_REQUESTCHANNELS

Desc:

Specifies whether all default SDO channels on the network should be requested from an SDO manager before attempting to send an SDO. Only used if the DLL is running a minimal CANopen server (started by calling CANopenServer_Startup).

Returns:

Nothing

4.32 CANopen_SDOChannelsTimeout

Prototype:

```
void CANopen_SDOChannelsTimeout(long timeout);
```

Params:
timeout = timeout to use

Desc:
Specifies the time to wait for a response from the SDO manager when attempting to request all default SDO channels on the network. If the SDO manager does not response within this time period, then it will be assumed that there is no SDO manager.

Returns:
Nothing

4.33 CANopen_SetSDOConfig

Prototype:

```
void CANopen_SetSDOConfig(int enableblocktransfer, unsigned char segfallbackthreshold, int useblockcrcs, unsigned char blocksize);
```

Params:
enableblocktransfer = set to 1 to enable block transfers, 0 to disable
segfallbackthreshold = set to a value which at or below that size of data in byte, segmented transfer will always be used, even if block transfer is enabled
useblockcrcs = set to 1 to enable CRC generation and checking for block transfers
blocksize = number of segments in a block, when using block transfers

Desc:
Configures the SDO protocol. Enables or disabled block transfer and when block transfer is used.

Returns:
Nothing

4.34 CANopen_SetBlockSegmentWriteDelay

Prototype:

```
void CANopen_SetBlockSegmentWriteDelay(long delay);
```

Params:
delay = delay to use in milliseconds

Desc:
Used to slow down the write of SDO blocks by inserting a delay after each segment.

Returns:
Nothing

4.35 CANopen_SetMode

Prototype:

```
void CANopen_SetMode(unsigned long mode);
```

Params:

mode = CANopen operating mode

Desc:

Specifies the CANopen operating mode to use. The default is CiA 301, which is the CANopen specification. The bits have the following meanings:

bit 0	When set use CiA 447 Car Add-on Devices mode

The CiA447 mode changes the SDO COB-IDs used to access servers on the network. The DLL uses the SDO channels assigned to node ID 16.

Returns:

Nothing

4.36 CANopen_NMT

Prototype:

```
void CANopen_NMT(unsigned char nodeid, int operation, RESULTS
*presult);
```

Params:

nodeid = ID of node to send NMT message to, or zero for all nodes
operation = NMT operation to perform. One of:

CANOPEN_START
CANOPEN_STOP
CANOPEN_PREOP
CANOPEN_RESET
CANOPEN_RESETAPP
CANOPEN_RESETCOMM

Desc:

Specifies the CANopen operating mode to use. The default is CiA 301, which is the CANopen specification. The bits have the following meanings:

bit 0	When set use CiA 447 Car Add-on Devices mode

The CiA447 mode changes the SDO COB-IDs used to access servers on the network. The DLL uses the SDO channels assigned to node ID 16.

Returns:

Nothing

4.37 CAN_Transmit

Prototype:

```
void CAN_Transmit(ESACAN_MSG *msg, RESULTS *presult);
```

Params:

Msg = pointer to details of message to be transmitted

Desc:

Transmits a CAN message. The timestamp is ignored.

Returns in results code:

OK
ERR_PROTCOL (data contains description of error)

4.38 CANopenConfig_WriteDCF

Prototype:

```
void CANopenConfig_WriteDCF(unsigned char nodeid, wchar_t
*dcffile, PROGRESS_CALLBACK ProgressFunc, MESSAGE_CALLBACK
MessageFunc, FINISHED_CALLBACK FinishedFunc, unsigned long
callbackparam, RESULTS *presult);
```

Params:

nodeid = ID of node to write DCF to
dcffile = path and name of DCF to write
ProgressFunc = function to call to indicate progress of operation. Null pointer if not required.
MessageFunc = function to call to provide status messages during DCF write. Null pointer if not required.
FinishedFunc = function to call when operation is finished. Null pointer if not required.
callbackparam = value passed to callback functions
presult = pointer to buffer to receive results of calling function.

Desc:

Writes a Device Configuration File to a specific node. If the finished callback function returns the error code ERR_WARNING, then some parts of the DCF write may not have been successful. Check the messages that were returned for detailed information. The reason the operation does not stop and return an error is that it may be acceptible for some writes to fail. OK is only returned if there were no problems at all. Call CANopen_Cancel to cancel the operation.

Returns in Results Code:

OK

4.39 CANopenConfig_WriteNCF

Prototype:

```
        void CANopenConfig_WriteNCF(wchar_t *ncffile,
PROGRESS_CALLBACK ProgressFunc, MESSAGE_CALLBACK MessageFunc,
FINISHED_CALLBACK FinishedFunc, unsigned long callbackparam, RESULTS
*presult);
```

Params:

ncffile = path and name of NCF to write
ProgressFunc = function to call to indicate progress of operation. Null pointer if not required.
MessageFunc = function to call to provide status messages during DCF write. Null pointer if not required.
FinishedFunc = function to call when operation is finished. Null pointer if not required.
callbackparam = value passed to callback functions
presult = pointer to buffer to receive results of calling function.

Desc:

Writes a Network Configuration File to the network. A Network Configuration File is a proprietory file format that contains the DCFs for multiple nodes. Each DCF is extracted and written to the corresponding node in turn, allowing an entire network to be configured. If the finished callback function returns the error code ERR_WARNING, then some parts of a DCF write may not have been successful. Check the messages that were returned for detailed information. The reason the operation does not stop and return an error is that it may be acceptible for some writes to fail. OK is only returned if there were no problems at all. Call CANopen_Cancel to cancel the operation. Ensure the NCF is stored on writeable media before calling this function, as a temporary file must be created in the same folder during execution.

The NCF has the following file format:

[DCF <*nodeid*>]
<*dcf*>
[DCF <*nodeid*>]
<*dcf*>

Where *nodeid* is the ID of the node in hexadecimal prefixed with "0x", and *dcf* is the DCF of that node.

CANopen Magic Pro and CANopen Magic ProDS can generate NCFs.

Returns in Results Code:

OK

4.40 CANopenServer_Startup

Prototype:

```
void CANopenServer_Startup(unsigned char nodeid, unsigned long devicetype, unsigned long vendorid, unsigned long productcode, unsigned long revisionnumber, unsigned long serialnumber, RESULTS *presult);
```

Params:

nodeid = ID to use for server
devicetype = device type to use for server
vendorid = vendor ID to use for server
productcode = product code to use for server
revisionnumber = revision number to use for server
serialnumber = serial number to use for server
presult = pointer to buffer to receive results of calling function

Desc:

Starts a minimal CANopen server running in the DLL. The server should be used when a feature in the DLL is enabled that requires a server to be running for data and/or control purposes. For example, requesting SDO channels from an SDO manager.

The minimal server implements Object Dictionary entries [1000], [1001] and [1018], plus any entries necessary to support DLL features that require the server to be running.

Returns:

Nothing

4.41 CANopenServer_Shutdown

Prototype:

```
void CANopenServer_Shutdown(void);
```

Params:

None

Desc:

Stops the minimal CANopen server.

Returns:

Nothing

4.42 MicroLSS_ScanAndConfig

Prototype:

```
void MicroLSS_ScanAndConfig(int eightybit, int byteoptimize,
ESACAN_LSSSLAVE plssslave[], int lssarraysize, unsigned char startnodeid,
unsigned long timeout, PROGRESS_CALLBACK ProgressFunc,
FINISHED_CALLBACK FinishedFunc, unsigned long callbkparam, RESULTS
*preturnresult);
```

Params:

eightybit = set to 1 to use 80-bit scan instead of 128-bit scan.
byteoptimize = set to 1 to use byte optimization where possible during scan.
plssslave = array to receive description of LSS slaves found on network. The valid field will be set to 1 for entries that describe LSS slaves.
lssarraysize = the number of entries in the plssslave array.
startnodeid = LSS slaves found are configured using a consecutive range of node IDs. This parameter specifies the first node ID.
timeout = the length of time to wait for a response in milliseconds.
ProgressFunc = progress callback function or NULL for none.
FinishedFunc = scan finished callback function.
callbkparam = value to pass to callback functions.
preturnresult = result of attempting to start the scan.

Desc:

Performs a scan for MicroLSS nodes on the network and configures them for operation. Each MicroLSS node that is found is configured with a node ID. The node IDs used are consecutive, starting with the node ID that is passed to the function.

The scan type may be 80 bits or 128 bits. In addition byte optimization may be used. Combining a 80 bit scan with byte optimization will typically produce the fastests scan times.

Note that MicroLSS is different from regular LSS. Please refer to the MicroLSS specification for details.

Returns:

Nothing

Chapter 5 – Windows CE Driver DLL API

5.1 Introduction

At runtime the appropriate driver to use for access to a CAN controller is selected by calling CANopenDLL_Startup. This package is provided with DLLs to access various PC based CAN interfaces, such as those from PEAK. However when using Windows CE there are many different kinds of integrated CAN controller/interface available.

In order to accommodate the various designs and implementations, the CANopen Magic Pro DLL supports the use of a generic API DLL, which can be used with the CANopen DLL. The workflow looks like the following:

Obtain the documentation for the CAN driver or existing driver DLL.
Write a Windows CE wrapper DLL with the API described in this chapter to access the CAN driver directly, or the existing driver DLL.
Make sure that the wrapper DLL has "WinCE" in it's name.
Call CANopenDLL_Startup with the name of the new wrapper DLL.

As long as the DLL has "WinCE" somewhere in it's name (for example WinCECANWrapper.DLL), then the CANopen Magic Pro DLL will know that it is a DLL for Windows CE and load and access it in the appropriate way.

Some CAN drivers or WinCE development systems may come with a wrapper DLL already written for use with this package. Please contact the vendor for details.

The API is very easy to implement.

5.2 API

The following code is the header file that should be used to generate the DLL. It contains a description of the functions.

The API does not need to be thread safe.

```
#ifndef _WINDOWSCECANH_
#define _WINDOWSCECANH_

#include <windows.h>

#ifdef __cplusplus
  #define C "C"
#else
```

```
  #define C
#endif

// The following ifdef block is the standard way of creating macros which make exporting
// from a DLL simpler. All files within this DLL are compiled with the
// WINDOWSCECAN_EXPORTS
// symbol defined on the command line. this symbol should not be defined on any project
// that uses this DLL. This way any other project whose source files include this file see
// WINDOWSCECAN_API functions as being imported from a DLL, whereas this DLL sees
// symbols
// defined with this macro as being exported.
#ifdef WINDOWSCECAN_EXPORTS
#define WINDOWSCECAN_API C __declspec(dllexport)
#else
#define WINDOWSCECAN_API C __declspec(dllimport)
#endif

// opens the CAN controller
// returns INVALID_HANDLE_VALUE for error or a handle to controller
WINDOWSCECAN_API HANDLE __stdcall WinCE_Open
  (
  wchar_t *name,                  // name of CAN interface "CAN1:", etc.
  int baudrate                    // baudrate in kbps
  );

// closes a CAN controller
WINDOWSCECAN_API void __stdcall WinCE_Close
  (
  HANDLE controller               // handle to previously opened can controller
  );

// sets the receive event for a CAN controller
// returns 1 for success, 0 for error
WINDOWSCECAN_API int __stdcall WinCE_SetReceiveEvent
  (
  HANDLE controller,              // handle to can controller to set event for
  wchar_t *eventname              // name of the event to use (must be previously
                                  // created)
  );

// starts a CAN controller
// returns 1 for success, 0 for error
WINDOWSCECAN_API int __stdcall WinCE_Start
  (
  HANDLE controller               // handle to can controller to start
  );

// stops a CAN controller
// returns 1 for success, 0 for error
```

```
WINDOWSCECAN_API int __stdcall WinCE_Stop
  (
  HANDLE controller               // handle to can controller to stop
  );

// transmits a CAN message
// returns 1 for success, 0 for error
WINDOWSCECAN_API int __stdcall WinCE_Transmit
  (
  HANDLE controller,              // handle to CAN controller to transmit on
  unsigned int id,                // message identifier
  unsigned char rtr,              // remote transmit request flag
  unsigned char ext,              // 29-bit id flag
  unsigned char err,              // error frame flag
  unsigned char dlc,              // number of data bytes
  unsigned char *pdata            // data bytes - unsigned char data[8]
  );

// recieves a CAN message from the controller
// returns 1 for message received, 0 for error
WINDOWSCECAN_API int __stdcall WinCE_Receive
  (
  HANDLE controller,              // handle to CAN controller to transmit on
  unsigned int *pid,              // filled with message identifier
  unsigned char *prtr,            // filled with remote transmit request flag
  unsigned char *pext,            // filled with 29-bit id flag
  unsigned char *perr,            // filled with error frame flag
  unsigned char *pdlc,            // filled with number of data bytes
  unsigned char *pdata            // filled with data bytes - unsigned char data[8]
  );

// returns the number of CAN interfaces supported
WINDOWSCECAN_API int __stdcall WinCE_NumberOfInterfaces
  (
  void
  );

// configures whether error frames should be received or not
// optional – feature can be unsupported but function must be implemented
WINDOWSCECAN_API void __stdcall WinCE_ConfigureErrorFrames
  (
  int mode                        // set to 1 to receive error frames, otherwise 0
  );

// configures whether transmitted messages should be received
// optional – feature can be unsupported but function must be implemented
WINDOWSCECAN_API void __stdcall WinCE_ConfigureSelfReceive
  (
  int mode                        // set to 1 to self receive, otherwise 0
```

```
  );

// gets status information from the CAN controller
// returns 1 for success, 0 for error
WINDOWSCECAN_API int __stdcall WinCE_Status
  (
  HANDLE controller,              // handle to CAN controller to transmit on
  int *pbusoff,                   // set to 1 if bus is off, otherwise 0
  int *perrorpassive,             // set to 1 if error passive, otherwise 0
  int *prxoverrun                 // set to 1 if receive buffer overrun, otherwise 0
  );

#endif
```