CANopen Magic Pro Library User Manual

Manual Revision 2.42

EMBEDDED SYSTEMS ACADEMY

For support contact support@esacademy.com

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

www.esacademy.com

EMBEDDED
SYSTEMS
ACADEMY

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 standardized 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 proprietary 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 Library

The CANopen Magic Pro Library 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 in Windows and a shared object in Linux. This library can be called by any application that knows how to use it. All copies of applications built on this platform include the library. By building upon the library an application immediately gains access to the knowledge of CANopen that have been built up over several years of effort. The library 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 relevant 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 library
- The C header file for the library
- The necessary support library files for the library
- An example application
- This manual that describes how to use the library

Features

The following are some of features of the CANopen library:

- 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 transmit and receive of PDOs
  - Event time and inhibit time supported
  - SYNC supported
  - Application layer transmit approval
  - Change of state detection
- 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 tools using the same library are running
    - PCANopen Magic Pro can show a trace of the CAN bus during development of applications using the library
- Supports Windows Vista/7/8/10, Windows CE 5.0 (see limitations) and Linux 2.6 with SocketCAN

Limitations

Windows Vista/7/8/10 Limitations

If CANopenDLL_Startup has been called and the user then changes the CAN interface type in the control panel, the list of available hardware interfaces returned from the DLL will not change to reflect the newly selected CAN interface 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.

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

Linux Limitations

CAN interfaces are supported using the SocketCAN framework. All interfaces that can be operated with SocketCAN will work with the library, however, some functions such as baud rate switching during operation may not be available. Full support is currently available for systems with Peak CAN interfaces, the Peak 'netdev' driver installed and SocketCAN installed and configured to use the Peak driver.

1.4 Obtaining Compatible CAN Interfaces

As mentioned in the feature list, PEAK-System Technik CAN interfaces are supported (except for Dongle and ISA). Visit www.peak-system.com to locate the nearest distributor. Additional CAN interfaces supported are:

- Kvaser

- VSCOM NET-CAN 110

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 Windows Installation

Minimum Requirements

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

- Windows Vista/7/8/10
- 3Mb of disk space
- A C , C++ or .NET Development system, such as Microsoft Visual Studio 2010 or Borland C++ Builder 5/6
- A supported CAN interface 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. See the sections below for a description of these steps.

Install PEAK Support

The PEAK support package must be installed before the library may be used with PEAK CAN interfaces. Normally it will be installed automatically at the end of the CANopen Magic Pro DLL installation.

2.2 Linux Installation

Minimum Requirements

The following is a list of the recommended minimum requirements for installing and using the package:

- Linux with Kernel 2.6 or greater (Debian based system recommended)
- A PEAK CAN interface with driver installed
- SocketCAN installed (included in kernels after 2.6.25)
- 2Mb of disk space
- gcc compiler

Installation Procedure

The PEAK driver and SocketCAN must be installed and working before the CANopen library can be installed. Please contact PEAK for details on how to install and test a PEAK CAN interface.

Verify that the PEAK driver is installed by running the following at the command prompt:

```
cat /proc/pcan
```

This will output a table showing your CAN interface. Next to check that SocketCAN is installed and working enter the following:

```
ifconfig can0
```

If can0 is functional then information about bytes transmitted and received, errors, etc. will be shown.

The GCC compiler must be installed. This varies depending on the Linux distribution. In Debian based systems the following can be used (at the command prompt):

```
sudo apt-get update
sudo apt-get install build-essential
```

Unpack the installation using:

```
tar xzf canopendllpro-x.xx.tar.gz
```

where x.xx are the version numbers. This will create the directory “canopendllpro-x.xx”. Next:

```
cd canopendllpro-x.xx
sudo make install
```

This will install the shared library on your system in /usr/lib.

To build the example application use the following:

```
cd gccexample
make
```

This will create an executable called “canopentest”. Run it with:

```
./canopentest
```

If the network is scanned and no errors are reported then the installation is complete.

Note that the origins of this library are as a Windows DLL. Therefore if this manual refers to the library as a DLL it also means a shared library.

Chapter 3 – Using the Library

3.1 Overview

The library 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_BroadcastSDODownload	Writing to multiple nodes at once
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_SetBlockSegmentWriteDelay	Sets the delay after each SDO block is written to control write speed
CANopen_SetPostTransmitDelay	Sets a delay after transmission of each CAN message
Hardware_GetCurrentTime	Gets the current time in timestamp format
Hardware_AddHardware	Adds a new CAN interface
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_OneShot	Turns on or off one shot transmission mode
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
LSSFastScan_ScanAndConfig	Scans for and configures LSS Fast Scan slaves
LSSFastScan_ScanAndConfig2	Scans for and configures LSS Fast Scan slaves
CANopenDLL_EnumerateDrivers	Obtains a list of supported drivers
PDO_CreateTPDO	Creates a new transmit PDO
PDO_CreateRPDO	Creates a new receive PDO
PDO_DeletePDO	Deletes a PDO
PDO_Transmit	Transmits a PDO
PDO_EventTransmitRequest	Registers a PDO transmit request callback function
PDO_EventReceive	Registers a PDO receive callback function
PDO_SetSYNCObject	Defines the SYNC object used for PDOs
PDO_SetData	Sets the data for a transmit PDO
CANopenMonitor_SDO	Creates a new SDO transfer monitor
CANopenMonitor_PDO	Creates a new PDO monitor
CANopenMonitor_EventHaveData	Registers a callback function for a monitor
CANopenMonitor_Start	Start a monitor
CANopenMonitor_Stop	Stops a monitor

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 in Windows

Microsoft Visual Studio 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 ESACANopenPro.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.

Microsoft Visual Studio C#

To use the DLL in a project:

- Copy ESACANopenProCS.DLL and ESACANopenPro.DLL into the same folder as your application executable
- Right click on the project and choose “Add Resource”
- Click on the Browse tab
- Select ESACANopenProCS.DLL and click on “OK”

Files that use the library should have the following using statement:

```
using ESACANopenProCS;
```

You must distribute both the DLLs 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 ESACANopenPro.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 Adding to a Project in Linux

GCC Compiler

To use the shared library in a project:

- Copy ESACANopenPro.h into the same directory as your project
- Include the header file in any files that call library functions
- When calling gcc, add the parameter -lesacanopenpro

You must distribute and install the shared DLL with your application. Do not distribute this documentation or the .h files.

3.4 C# Notes

All enumerations and objects are in the ESACANopenProCS namespace.

Enumerations are in the top level of the namespace, for example:

```
ESACANopenProCS.enumerrorcodes code = ESACANopenProCS.enumerrorcodes.OK;
```

The library is broken into multiple classes, for example Event, CAN, CANopen. The function descriptions note which class defines the function.

3.5 Calling Functions

Return Values

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

C/C++:

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

C#:

```
struct RESULTS
{
  enumerrorcodes code;
  string details;
}
```

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:

C/C++:

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

C#:

```
struct ESACAN_TIMESTAMP
{
  UInt32 millis;
  UInt32 millis_overflow;
  UInt32 micros
}
```

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:

C/C++:

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

C#:

```
struct ESACAN_MSG
{
  ESACAN_TIMESTAMP timestamp;
  UInt32 id;
  byte dlc;
  enummsgflags flags;
  byte[] data;
}
```

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

C# note: logically OR the flags rather than adding them.

The ESACAN_DRIVER type describes a hardware driver. It is defined as:

C/C++:

```
typedef struct
{
  wchar_t name[ESACAN_MAXDRIVERNAMELEN];
  wchar_t drivername[ESACAN_MAXDRIVERNAMELEN];
  unsigned char canedithardware;
  unsigned char canlistenonly;
} ESACAN_DRIVER;
```

C#:

```
struct ESACAN_DRIVER
{
  string name;
  string drivername;
  bool canedithardware;
  bool canlistenonly
}
```

name holds a user-friendly name of the hardware interface. drivername holds a name that can be passed to CANopenDLL_Startup. If canedithardware is true (non-zero) then the functions Hardware_AddHardware and Hardware_DeleteHardware need to be used to manually add and delete the available CAN interfaces, as they cannot be automatically discovered. If canlistenonly is true (non-zero) then **some** CAN interfaces can be set to silent/listen-only mode.

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

C/C++:

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

C#:

```
struct ESACAN_HARDWARE
{
  string name;
  Int32 handle;
}
```

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:

C/C++:

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

C#:

```
struct ESACAN_NETWORK
{
  string name;
  Int32 handle;
  Int32 baudrate;
}
```

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:

C/C++:

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

C#:

```
struct ESACAN_NODEINFO
{
  enumnodeinfo status;
  UInt32 devicetype;
}
```

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 in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate void PROGRESS_CALLBACK(float percentage, IntPtr 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 in C/C++ with the following prototype:

C/C++:

```
void (__stdcall *FINISHED_CALLBACK)(RESULTS *results, void *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 FINISHEDBROADCAST_CALLBACK is a function pointer type in C/C++ with the following prototype:

C/C++:

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

called when an operation such as broadcast SDO download is complete. Passed is a list of nodes that were written to, 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 function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate void MESSAGE_CALLBACK(String msg, IntPtr 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 in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate void MAJORERROR_CALLBACK(enummajorerrors error, IntPtr callbackparam);
```

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

Also passed is a user defined parameter. This function is only called when the major error changes.

The RECEIVE_CALLBACK is a function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate void RECEIVE_CALLBACK(ref ESACAN_MSG msg, Int32 reply, IntPtr
callbackparam);
```

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. Passed is a user defined parameter.

The TRANSMIT_CALLBACK is a function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate void TRANSMIT_CALLBACK(ref ESACAN_MSG msg, IntPtr callbackparam);
```

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. Passed is a user defined parameter.

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 in C/C++ and a delegate in C# with the following prototype:

C/C++:

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

C#:

```
delegate Int32 SWITCHNETWORKS_CALLBACK(Int32 newnethandle, Int32 pause, IntPtr
callbackparam);
```

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. Also passed is a user defined parameter. This function should return immediately and not wait for pause milliseconds to pass before returning.

Note: __stdcall does not exist in Linux and can be removed from the function prototypes.

The MONITOR_CALLBACK is a function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

```
typedef void (__stdcall *MONITOR_CALLBACK)(ESACAN_HANDLE monitorhandle,
unsigned char *data, unsigned long datasize, void * callbackparam);
```

C#:

```
delegate void MONITOR_CALLBACK(IntPtr monitorhandle, byte[] data, UInt32 datasize,
IntPtr callbackparam);
```

This function is called when a monitor has new data, obtained from watching for an operation on the CAN bus. monitorhandle is the handle to the previously created monitor, data is a pointer to the buffer originally passed to the monitor creation function, datasize is the amount of data recorded and callbackparam is the arbitrary value passed to the DLL when registering the callback function.

Note: __stdcall does not exist in Linux and can be removed from the function prototypes.

The PDOTXREQ_CALLBACK is a function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

```
typedef int (__stdcall *PDOTXREQ_CALLBACK)(ESACAN_HANDLE pdohandle, void *
callbackparam);
```

C#:

```
delegate Int32 PDOTXREQ_CALLBACK(Int32 pdohandle, IntPtr callbackparam);
```

This function is called when a PDO is about to be transmitted due to either a change of state or the transmission type of the PDO is zero (synchronous, device profile defined). pdohandle is the handle of the previously created transmit PDO. Also passed is a user defined parameter. This function should return TRUE (non-zero) to allow the PDO to be transmitted or return FALSE (zero) to stop transmission of the PDO.

Note: __stdcall does not exist in Linux and can be removed from the function prototypes.

The PDORX_CALLBACK is a function pointer type in C/C++ and a delegate in C# with the following prototype:

C/C++:

```
typedef void (__stdcall *PDORX_CALLBACK)(ESACAN_HANDLE pdohandle, unsigned char
*data, int length, void * callbackparam);
```

C#:

```
delegate void PDORX_CALLBACK(Int32 pdohandle, byte[] data, Int32 length, IntPtr
callbackparam);
```

This function is called when an asynchronous PDO has been received or a SYNC message has been detected on a bus and a PDO is defined as being synchronous. pdohandle is the handle of the previously created transmit PDO. Also passed is a user defined parameter.

Note: __stdcall does not exist in Linux and can be removed from the function prototypes.

Typical Call Flow

The following is a typical sequence of function calls.

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

- CANopenDLL_EnumerateDrivers
- 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
- PDO_CreateTPDO / PDO_CreateRPDO
- 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

3.6 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.

All functions that have a finished callback have a non-synchronous or blocking counterpart. This blocking version can be used in the same way as the non-blocking except that the function does not return until the operation has completed or a timeout has occurred. It is intended that these functions are executed in a separate thread to the user interface.

Only one function may be called at any one time in a single instance of the library. The single exception is that CANopen_Cancel (CANopen.Cancel in C#) 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 library may be loaded at any one time, allowing multiple parallel operations to take place on the CAN bus. For example, when using the library 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 library. Therefore the usual limitations and precautions apply when using data in a callback function.

Functions that execute in a separate thread do not make copies of buffers. Therefore the data in a buffer must remain valid and allocated until the callback function indicates the operation has finished.

3.7 Description

Overview

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

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

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 library, the hardware must be closed and the library shut down. Failure to do so may cause memory leaks.

Start Up

The function CANopenDLL_EnumerateDrivers is called to obtain a list of supported drivers. Before using a driver make sure the corresponding DLL from the hardware vendor is installed on the system or in the same folder as the CANopen DLL.

The function CANopenDLL_Startup is called to start the library. No other functions in the library may be called until after this function has been called with the exception of CANopenDLL_EnumerateDrivers.

The library may be used to allow a single process to access a CAN interface. This is the standard arrangement. However, the library also supports multi-process access, where multiple processes each using a copy of the library 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 (Hardware.EnumerateHardware in C#) 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 use the CAN-Hardware Control Panel applet. Once changed, Hardware_EnumerateHardware will then return a different list based on the new type.

Note that some CAN interfaces cannot be automatically discovered, for example the VSCAN interfaces from Vision Systems. For those interfaces the function Hardware_AddHardware must first be called to add interfaces that are known to exist. Once added Hardware_EnumerateHardware can be called to obtain a complete list with handles.

Once the user has selected a hardware interface to use, the handle to the interface is passed to Hardware_EnumerateNetworks (Hardware.EnumerateNetworks in C#) 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 (Hardware.AddNetwork in C#). Also the application can delete networks using Hardware_DeleteNetwork (Hardware.DeleteNetwork in C#).
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 (Hardware.Initalize in C#).

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 (Event.xxxxx in C#). 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 library, therefore the use of messages, signals, mutexs etc. is required to pass data to the rest of your application.

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

All the callback functions receive a user defined parameter. This is the exact same value that is passed to the library 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 documentation on how to define functions to use this calling convention.

CAN Bus Operations

SDO uploads and downloads may be performed by calling CANopen_SDOUpload (CANopen.SDOUpload in C#) and CANopen_SDODownload (CANopen.SDODownload in C#). 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 (CANopen.ScanNetwork in C#). 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 (CANopen.Cancel in C#).

Plain CAN messages may be transmitted by calling CAN_Transmit (CAN.Transmit in C#), and network management messages may be transmitted by calling CANopen_NMT (CANopen.NMT in C#).

Node and network configuration may be performed by calling CANopenConfig_WriteDCF (CANopenConfig.WriteDCF in C#) and CANopenConfig_WriteNCF (CANopenConfig.WriteNCF in C#). The configuration operations may be cancelled by calling CANopen_Cancel (CANopwn.Cancel in C#).

Shut Down

Once the library is no longer needed, Hardware_Close (Hardware.Close in C#) must be called followed by CANopenDLL_Shutdown. Once CANopenDLL_Shutdown has been called the only library function that may be called is CANopenDLL_Startup. No other functions may be called.

3.8 Distribution

To distribute an application based on the library 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 the PEAK redistributable on the user's PC. This came with your copy of this package. To do this: copy all the PcanDevRedist.* and PcanDevUtils.* into a temporary folder on the user’s PC, run PcanDevRedist.exe, delete all the files after installation has completed.

Windows Vista/7/8/10 C/C++:

- Include ESACANopenPro.dll with your application executable, preferably in the same folder as your application executable

- If using PEAK: Create the registry key HKEY_LOCAL_MACHINE\SOFTWARE\PEAK-System\CANAPI2 (32-bit Windows) or HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PEAK-System\CANAPI2 (64-bit Windows) and set the key permissions to allow full access to the "everyone" group
- For any other CAN interface vendor copy the driver DLL into the same folder as your executable

Windows CE:

- Include the driver wrapper DLL with your application executable.

Windows Vista/7/8/10 C#:

- Include ESACANopenPro.dll and ESACANopenProCS.dll with your application executable, in the same folder as your application executable
- If using PEAK: Create the registry key HKEY_LOCAL_MACHINE\SOFTWARE\PEAK-System\CANAPI2 (32-bit Windows) or HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\PEAK-System\CANAPI2 (64-bit Windows) and set the key permissions to allow full access to the "everyone" group
- For any other CAN interface vendor copy the driver DLL into the same folder as your executable

Linux:

- Include and install esacanopenpro.so as a shared library

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 or ESACANopenProCS file.**
- **Any source code showing the use of the ESACANopenPro.dll or ESACANopenProCS.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.

3.9 Optimizations

Windows

The default configuration for the DLL is to insert a delay of 1ms after the transmission of every message. Most of the CAN interface drivers supported, including the PEAK driver, have adequate buffer handling, making this delay unnecessary. To eliminate the delay call the following in your application:

C/C++:

```
CANopen_SetPostTransmitDelay(0);
```

C#:

```
CANopen canopen = new CANopen();
canopen.SetPostTransmitDelay(0);
```

Linux

The default configuration for SocketCAN set up a small number of transmit buffers. This means that the ESACANopenPro library has to insert a 1ms delay after transmission of every message to ensure there will always be a free transmit buffer.

This delay can be eliminated if the number of transmit buffers is increased. To increase the transmit buffers:

```
ifconfig can0 txqueuelen 300
```

where 300 is the number of new transmit buffers. Note that this must be executed after every reboot, unless a startup script is created for the Linux distribution to set this up automatically.

Next eliminate the transmit delay by calling the following in your application:

```
CANopen_SetPostTransmitDelay(0);
```

Chapter 4 – Function Reference

4.1 CANopenDLL_Startup

Prototype:

C/C++:

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

C#:

```
void CANopenDLL_Startup(enumoperationmodes mode, String drivername);
```

Params:

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

"CanApi2.dll"	= PEAK CAN interfaces driver
“SOCKETCAN”	= Linux SocketCAN interface

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.

Note: CanAPI2.dll and SOCKETCAN only support ESACAN_SINGLEPROCESS, as multi-application support is provided by the PEAK system.

Returns:

Nothing

C# Class:

ESACANopenPro

4.2 CANopenDLL_Shutdown

Prototype:

C/C++:

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

C#:

```
        void CANopenDLL_Shutdown();
```

Params:

None

Desc:

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

Returns:

Nothing

C# Class:

ESACANopenPro