ECOM Developer's API Reference Guide

Library Overview:

The functions provided by the ECOM library (ecommlib.dll) are exported using the `__stdcall` calling convention which is the convention used by the majority of the Microsoft Windows API. This means that any programming language and environment that can make calls to the Windows API can also use the ECOM library.

Tip: For programming languages or compilers that cannot automatically load the library functions, try searching for examples of using the Windows Functions `LoadLibrary` and `GetProcAddress` for your desired programming language.

Most of the library functions expect a `HANDLE`, which represents a single connected/opened ECOM device, as their first parameter. In order to obtain a `HANDLE` to be passed to the library functions, you must first call `CANOpen` for the respective ECOM device. Make sure to call `CloseDevice` once for each successful call to `CANOpen`, as each ECOM device can only be opened by one process at a time.

Most functions, besides the device open functions, return a single `BYTE` value that represents the function's return status code. For a complete listing of all possible return codes along with detailed descriptions, refer to the defined codes in the "ecommlib.h" file. Additionally, the function `GetFriendlyErrorMessage` can be called from within code to get a text based description of the error. In all functions, the `BYTE` data type represents an unsigned 8-bit value, and the `DWORD` and `ULONG` data types represent an unsigned 32-bit value.

This document groups all the library functions according to their intended purpose, as follows:

- ECOM Initialization
- Transmission and reception of CAN messages
- Transmission and reception of serial data
- Searching and enumerating devices
- Miscellaneous Functions

ECOM Initialization

Function CANOpen:

Declaration:

```
HANDLE CANOpen(ULONG SerialNumber, BYTE BaudRate, BYTE
*ErrorReturnCode)
```

Description:

This function is used to initiate an ECOM device in CAN mode. To open the first available device, pass 0 for the SerialNumber parameter, otherwise pass the desired device serial number. Each device has a unique serial number that is printed on the underside of the ECOM.

The function will initialize the ECOM device on the CAN bus with the respective baud rate. If an error occurs, a NULL value will be returned by the function. On successful connection, it will return a HANDLE to the ECOM device that can be used for all respective function calls. The HANDLE must be closed with the CloseDevice function once for every successful call to CANOpen.

The following are the allowed values to pass for BaudRate:

```
#define CAN_BAUD_250K         0
#define CAN_BAUD_500K         1
#define CAN_BAUD_1MB          2
#define CAN_BAUD_125K         3
```

ErrorReturnCode receives a reference to the return ErrorCode upon the functions return. This parameter can be passed a NULL reference if you are not interested in the return error code.

Function CANOpenFiltered

Declaration:

```
HANDLE CANOpenFiltered(ULONG SerialNumber, BYTE BaudRate,
DWORD AcceptanceCode, DWORD Acceptancemask, BYTE
*ErrorReturnCode)
```

Description:

This function behaves exactly like CANOpen except that it allows a hardware CAN receive filter to be applied in the AcceptanceCode and AcceptanceMask parameters. See the Philips SJA1000 CAN transceiver data sheet for information on how the acceptance filter is defined.

Function SerialOpen

Declaration:

```
HANDLE SerialOpen(USHORT SerialNumber, BYTE BaudRate, BYTE
*ErrorReturnCode);
```

Description:

```
This function behaves exactly like CANOpen except that
instead opens the ECOM as a serial device and initiates
communication using the serial port.  The serial
communication lines ARE NOT RS232, but rather 5V TTL lines;
therefore, they cannot be used to communicate directly with
an RS-232 port on a PC.  The serial feature is intended to
communicate directly with the standard hardware on most
microcontrollers without the additional cost of having to
add an RS-232 transceiver.

Each ECOM device can be connected as either CAN or Serial,
but not both!  Passing the HANDLE returned by SerialOpen to
a CAN based function will return an error code.

The following are the allowed values to pass for BaudRate:
#define SERIAL_BAUD_2400      0
#define SERIAL_BAUD_4800      1
#define SERIAL_BAUD_9600      2
#define SERIAL_BAUD_19200     3
#define SERIAL_BAUD_28800     4
#define SERIAL_BAUD_38400     5
#define SERIAL_BAUD_57600     6
```

Function CloseDevice

Declaration:

```
BYTE CloseDevice(HANDLE DeviceHandle)
```

Description:

```
This function closes an ECOM device HANDLE that was
obtained using CANOpen, CANOpenFiltered, or SerialOpen.
For devices opened as CAN, this function will also
disconnect the device from the CAN bus. CloseDevice must be
called once for every HANDLE that is obtained or else the
device may be left open and another program will not be
able to use it.  Failure to call CloseDevice for every
HANDLE obtained will also result in a memory leak until the
DLL library is unloaded.
```

Function CANSetupDevice

Declaration:

```
BYTE CANSetupDevice(HANDLE DeviceHandle, BYTE SetupCommand,
BYTE SetupProperty)
```

Description:

```
This function is used to alter the behavior of the
CANTransmitMessage and CANTransmitMessageEx functions.
SetupCommand should be set to zero (CAN_CMD_TRANSMIT) for
all calls to this function - currently the only supported
SetupCommand is CAN_CMD_TRANSMIT.

For the CAN_CMD_TRANSMIT SetupCommand, the following are
valid values for SetupProperty:

#define CAN_PROPERTY_ASYNC          0
#define CAN_PROPERTY_SYNC           1

CAN_PROPERTY_ASYNC: Tells the CANTransmit functions to
behave asynchronously, this means that calls to CANTransmit
can return before the message is actually transmitted on
the CAN bus.

CAN_PROPERTY_SYNC: This property tells the CANTransmit
functions to wait indefinitely for the ECOM device to send
the message before returning back to the caller.

It is recommended that users use the CAN_PROPERTY_ASYNC
method.  The default after calling CANOpen is synchronous
mode (CAN_PROPERTY_SYNC), so this function must be called
to switch operation to asynchronous mode.
```

Transmission and Reception of CAN Messages

Function CANTransmitMessage:

Declaration:

```
BYTE CANTransmitMessage(HANDLE cdev, SFFMessage *message)
```

Description:

This function will transmit a CAN message in Short Frame Format (SFF) 11-bit mode. Depending on the configuration, the message may be transfered in asynchronous or synchronous mode (See CANSetupDevice for explanation). In asynchronous mode the calling program will immediately be given back control but no error detection or indication of whether the message was sent successfully will be given. In synchronous mode, the function will wait until the ECOM device indicates that the message was successfully sent (or an error indicating otherwise will be returned). In both modes, the CAN hardware will be put in automatic retransmit mode, so that message will be retried until it is sent out on the bus.

Required Structure:

```
typedef struct
{
      BYTE  IDH;
      BYTE  IDL;
      BYTE  Data[8];
      BYTE  Options;    //BIT 6 = remote frame bit
                        //BIT 4 = self-reception bit
      BYTE  DataLength;
      DWORD TimeStamp;  //Timestamp with 64us resolution
} SFFMessage;
```

Fill this structure with the desired 11-bit CAN data. The upper 5 bits of IDH are not used and ignored since the message is only 11-bits. DataLength must be set to a valid CAN value between 0 and 8, and the corresponding data should be set as well. For transmit messages, the TimeStamp field has no meaning. For receive messages, the TimeStamp will be set to a value representing the number of 64us ticks that have passed since the call to CANOpen or CANOpenFiltered. The timestamp is a hardware timestamp captured during the CAN receive interrupt. To enable transmission of remote frame, set bit 6 (base 0) of the Options flag. To enable hardware based self-reception, set bit 4 of the Options flag. Self-reception allows for the reception of all self-transmitted messages and it is hardware based - a receive interrupt is generated and received message is processed just like other incoming packets.

Function CANTransmitMessageEx:

Declaration:

```
BYTE CANTransmitMessageEx(HANDLE cdev, EFFMessage *message)
```

Description:

This function will transmit a CAN message in Extended Frame Format (EFF) 29-bit mode. CANTransmitEx behaves otherwise in the exact manner as CANTransmit.

Required Structure:

```
typedef struct
{
      DWORD ID;  //29-bit ID, upper 3 bits are ignored
      BYTE  data[8];
      BYTE  options;  //BIT 6 = remote frame bit
                      // BIT 4 = self-reception
      BYTE  DataLength;
      DWORD TimeStamp;  //Timestamp with 64us resolution
} EFFMessage;
```

Fill this structure with the desired 29-bit CAN message data. Aside from the ID field being 29-bits, all other fields have the same meaning as in the SFFMessage structure.

Function CANReceiveMessageEx:

Declaration:

```
BYTE CANReceiveMessageEx(HANDLE cdev, EFFMessage *message)
```

Description:

This function will read one message from the current EFF (29-bit) receive buffer.

On success, the function will fill the message structure with the oldest CAN packet in the EFF buffer and return 0. It will return CAN_NO_RX_MESSAGES if there are no messages in the EFF buffer. See the explanation of CANTransmitMessageEx for a description of the EFFMessage structure.

To retrieve how many messages are currently in the EFF buffer, the function GetQueueSize can be called with CAN_GET_EFF_SIZE passed for the flag.

Function CANReceiveMessage:

Declaration:

```
BYTE CANReceiveMessage(HANDLE cdev, SFFMessage *message)
```

Description:

```
This function will read one message from the current SFF
(11-bit) receive buffer.

On success, the function will fill the message structure
with the oldest CAN packet in the SFF buffer and return 0.
It will return CAN_NO_RX_MESSAGES if there are no messages
in the SFF buffer.  See the explanation of
CANTransmitMessage for a description of the SFFMessage
structure.

To retrieve how many messages are currently in the SFF
buffer, the function GetQueueSize can be called with
CAN_GET_SFF_SIZE passed for the flag.
```

Transmission and Reception of Serial Data

Function SerialWrite:

Declaration:

```
BYTE SerialWrite(HANDLE DeviceHandle, BYTE *DataBuffer,
LONG *Length);
```

Description:

This function will write a buffer out the serial port of an ECOM device that has been opened using SerialOpen.

Pass the array of data to send in DataBuffer and pass the length of the data array in Length. On success, the function will return 0 and will set Length to the number of data bytes that were actually sent. Even if there is a non-zero (error) return code, Length will have been set and some bytes may have still been sent.

Function SerialRead:

Declaration:

```
BYTE SerialRead(HANDLE DeviceHandle, BYTE *DataBuffer, LONG
*BufferLength);
```

Description:

This function will read data that has been received by the serial port for an ECOM device that has been opened using SerialOpen.

Pass a BYTE array that will receive that data in DataBuffer and pass the length of the data array in BufferLength. On success, the function will return 0 and will set Length to the number of data bytes that were actually filled in the buffer.

Searching and Enumerating Devices

The functions StartDeviceSearch, FindNextDevice, and CloseDeviceSearch are used to list all ECOM devices that are connected to the respective computer. To perform a search, call StartDeviceSearch to obtain a DEV_SEARCH_HANDLE. Then call FindNextDevice using the handle until FindNextDevice reports that there are no more devices left to retrieve. Each successful call to FindNextDevice will fill a DeviceInfo structure which contains information about the respective device. When done listing devices, make sure to call CloseDeviceSearch with the respective handle to properly clean-up and free memory used by the search.

Below is an example that will list all ECOM devices that are attached to the computer.

```
#include "ecommlib.h"  //Include definitions for constants and functions used by the ECOM

int ListECOMDevices()
{
        //structure that will be used to retrieve information about each device
        DeviceInfo     deviceInfoStruct;

        //Obtain a search handle that can be used to retrieve ALL connected ECOM devices.
        DEV_SEARCH_HANDLE searchHandle = StartDeviceSearch(ECOM_FIND_ALL);

        //Check for errors
        if (searchHandle == NULL)
        {
                printf("Unexpected error allocating memory for device search\n");
                return -1;
        }

        //Now retrieve each attached device until there are no more left
        //When the searching is done, it will return ECI_NO_MORE_DEVICES
        int deviceCount = 0;
        while(FindNextDevice(searchHandle, &deviceInfoStruct) == ECI_NO_ERROR)
        {
                //Print the serial number of the found device
                printf("Device Found: %d\n", deviceInfoStruct.SerialNumber);
                deviceCount++;  //keep count of the number connected
        }
        printf("Found %d devices\n", deviceCount);

        //Make sure to close the search handle to free up memory used.
        CloseDeviceSearch(searchHandle);

        return deviceCount; //return the number of attached devices
}
```

Function StartDeviceSearch:

Declaration:

```
DEV_SEARCH_HANDLE StartDeviceSearch(BYTE Flag);
```

Description:

This function is used to start a device search that can be used to list every device that is attached to the current computer.

To start a search, create a handle by passing the type of search (FIND_ALL, FIND_OPEN, or FIND_UNOPEN) you wish to perform in the Flag parameter of StartDeviceSearch. Then call FindNextDevice repeatedly with the respective DEV_SEARCH_HANDLE and each call to FindNextDevice will return the next connected device until no more are left. When all have been listed, FindNextDevice will return ECI_NO_MORE_DEVICES.

Function CloseDeviceSearch:

Declaration:

```
BYTE CloseDeviceSearch(DEV_SEARCH_HANDLE SearchHandle);
```

Description:

This function is used to close a device search that has been started using the StartDeviceSearch function. This function must be called once for each DEV_SEARCH_HANDLE obtained using StartDeviceSearch. It is used to free up resources and memory used by the device search functions.

Function FindNextDevice:

Declaration:

```
BYTE FindNextDevice(DEV_SEARCH_HANDLE SearchHandle,
DeviceInfo *deviceInfo);
```

Description:

```
This function is used to retrieve information about each
ECOM device that is attached to the computer.

Call this function repeatedly for a given DEV_SEARCH_HANDLE
until the function returns ECI_NO_MORE_DEVICES.  Each call
that returns ECI_NO_ERROR will fill the DeviceInfo
structure with information about the respective device.
```

Required Structure:

```
typedef struct
{
      ULONG SerialNumber;  //Device serial number
      BYTE    CANOpen;     //is device opened as CAN
      BYTE  SEROpen;       //is device opened as Serial
      BYTE  _reserved;     //reserved for future use
      BYTE  SyncCANTx;     //always FALSE - Legacy support
      HANDLE DeviceHandle;  //always NULL when returned by
this function - Legacy support, it used to return an opened
device's handle, but this was not valid across multiple
processes, so it has been removed altogether to avoid
confusion. Each process must keep track of its open device
HANDLEs
      BYTE  reserved[10];  //reserved for future use
}     DeviceInfo;
```

Miscellaneous Functions

Function GetErrorMessage:

Declaration:

```
BYTE GetErrorMessage(HANDLE DeviceHandle, ErrorMessage
*ErrorMessage);
```

Description:

```
This function will read one error message from the current
error message buffer.

On success, the function will fill the message structure
with the oldest error frame in the buffer and return 0.  It
will return CAN_NO_ERROR_MESSAGES if there are no messages
in the error buffer.

To retrieve how many messages are currently in the error
buffer, the function GetQueueSize can be called with
CAN_GET_ERROR_SIZE passed for the flag.
```

Required Structure:

```
typedef struct
{
      unsigned int ErrorFIFOSize; //number of remaining
error messages
      BYTE ErrorCode;  //See "ErrorMessage Control Bytes"
in ecommlib.h for more info
      BYTE ErrorData;
      double Timestamp;  //Timestamp when error was
captured
      BYTE reserved[2]; //Reserved for future use
} ErrorMessage;
```

Function GetDeviceInfo:

Declaration:

```
BYTE GetDeviceInfo(HANDLE DeviceHandle, DeviceInfo
*deviceInfo);
```

Description:

This function will fill the DeviceInfo structure with information about the device that is referenced by DeviceHandle. See the FindNextDevice function for a description of the DeviceInfo structure.

Function SetCallbackFunction:

Declaration:

```
BYTE SetCallbackFunction(HANDLE DeviceHandle,
pMessageHandler *ReceiveCallback, void *UserData);
```

Description:

This function assigns a "callback function" that will be executed by the DLL everytime a new CAN, Serial, or Error message is received.

Callback Declaration:

```
typedef BYTE (_stdcall *pMessageHandler)(HANDLE
DeviceHandle, BYTE Flag, DWORD FlagInfo, void* UserData);
```

Callback Description:

The assigned callback function is executed everytime a new CAN, Serial, or Error message is received. The Flag parameter will be set to CAN_EFF_MESSAGES, CAN_SFF_MESSAGES, CAN_ERR_MESSAGES, or SER_BYTES_RECEIVED to indicate which type of message was just received and the FlagInfo parameter will indicate how many messages are in the respective message buffer. UserData is the same value that was set in the SetCallbackFunction call.

Warning: The ReceiveCallback function is called in the context of a unique and high-priority thread; therefore, you must ensure that there are no concurrency issues between data that is accessed within the callback and the rest of your application. Users who do not understand threads and critical sections should not use this function. Also, ensure that the callback function is executed quickly with no waiting or data overruns can occur.

Function GetQueueSize:

Declaration:

```
int GetQueueSize(HANDLE DeviceHandle, BYTE Flag);
```

Description:

```
Call this function with a valid DeviceHandle and with the
Flag parameter set to one of the following values in order
to retrieve information about the devices buffers/queues.
The respective value is returned or -1 if an error occurs.

FOR DEVICES OPENED AS CAN
CAN_GET_EFF_SIZE         //Get message count in EFF buffer
CAN_GET_MAX_EFF_SIZE     //Get max size of the EFF buffer
CAN_GET_SFF_SIZE         //Get message count in SFF buffer
CAN_GET_MAX_SFF_SIZE     //Get max size of the SFF buffer
CAN_GET_ERROR_SIZE       //Get message count in error buffer
CAN_GET_MAX_ERROR_SIZE   //Get max size of error buffer
CAN_GET_TX_SIZE          //Get number of messages waiting to
                         be transmitted (both SFF and EFF)
CAN_GET_MAX_TX_SIZE      //Get max size of transmit buffer

FOR DEVICES OPENED AS SERIAL
SER_GET_RX_SIZE          //Get byte count in RX buffer
SER_GET_MAX_RX_SIZE      //Get max byte count of RX buffer
SER_GET_TX_SIZE          //Get number of bytes waiting to be
                         transmitted
SER_GET_MAX_TX_SIZE      //Get max size of transmit buffer
```

Function GetFriendlyErrorMessage:

Declaration:

```
void GetFriendlyErrorMessage(BYTE ErrorCode, char
*ErrorString, int ErrorStringSize);
```

Description:

```
Use this function to retrieve a string based error
description for any error code that is returned by any of
the functions in this DLL.  Pass a character array for
ErrorString and the length of the array in ErrorStringSize.
The function will then fill a description into ErrorString
that will be NULL terminated to indicate the end of the
string.
```