Inspiring Motion
Since 1988

Release Notes for Drive .NET Library 1.0.0.8

Version 1.0.0.8

July 2015 (Ver. 1.000) www.elmomc.com

Elmo
Motion Control

Notice

This guide is delivered subject to the following conditions and restrictions:

- This guide contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for the purpose of assisting users of the Gold Line technology.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice.

Elmo Motion Control and the Elmo Motion Control logo are registered trademarks of Elmo Motion Control Ltd.

EtherCAT Conformance Tested. EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.

CANopen compliant. CANopen® is a registered trademark and patented technology, licensed by CAN in Automation (CiA) GmbH, Kontumazgarten 3, DE-90429 Nuremberg, Germany.

Revision History

Version	Date	Details
Ver. 1.000	July 2015	Initial version

Chapter 1: General

The following release notes highlight the features of the first Drive .NET library release (version 1.0.0.8).

This library provides a C# based API for connecting with the Elmo drives and activating those drives.

This document contains general information and it doesn't replace the library documentation and the example applications also available with the library or on the Elmo web site.

Please note: Version 1.0.0.8 includes spelling corrections to methods from version 1.0.0.7.

This release is for the use of all customers.

Using the Library

The library contains a few DLL (dynamic library) files.

In order to use the library, you should add the ElmoMotionControlComponents.Drive.EASComponents.dll to your project references.

The other DLL files will be loaded by the main library, as required:

- The ElmoMotionControlComponents.GMAS.MMCLibDotNET.dll is used when connecting to a CAN Gateway drive via a CAN G-MAS connection
- The canlibCLSNET32.dll and canlibCLSNET64.dll files are used for connecting to a drive using CANOpen protocol via a Kvaser CAN card

You do not need to include those other libraries in your project, the main library will automatically load them, as necessary.

Chapter 2: Communicating with the Drives

Means of Communication to the Drive

The drive .NET library allows you to communicate with the Elmo drives in all the communication methods supported by the drives, as detailed below:

- RS232 – communication through a serial RS232 port
- USB – communication through a USB port, using the Elmo USB driver
- UDP – communication using the UDP protocol over Ethernet connection
- Direct CANOpen – communication using the CANOpen protocol through a 3rd party CAN card connected to the PC
- CAN Gateway – communication using the CANOpen protocol through a GMAS CAN master

Communication Information

The object defines the communication method with the drive a set of classes implementing the interface `IdriveCommunicationInfo`.

In order to establish communication with the drive, first create an instance of a class implementing this interface using the `DriveCommunicationFactory`.

Communication Object

The class connecting to the drive is the `DriveCommunication`, implementing the interface `IDriveCommunication`.

In order to connect to a drive, call the `CreateCommunication` method of the `DriveCommunicationFactory` with the desired communication info object.

After that, you can call the `Connect` method of the communication object in order to connect to the drive and the `Disconnect` method in order to disconnect from the drive.

Sending Commands to the Drive

The drive responds to textual commands composed of two-letters with or without index, such as "AC", "MO", "UI[1]", "CA[41]" etc.

The communication object allows you to send commands to the drive and receive the drive response, or an error object in case of any error.

You can send a command to the drive by calling the communication object's `SendCommand` or `SendCommandAnalyzeError` methods.

The `SendCommand` method throws an exception if encountered any error, while the `SendCommandAnalyzeError` method returns an error object as an output parameter.
The error object (or the Exception thrown) contains both a drive error code and a library error code which are filled with non-zero values when relevant.

Using the Drive Personality

The drive personality is a part of the drive Firmware version which describes some of the current firmware's qualities and parameters.

The library can upload the personality data from the drive and save it into an XML file called the personality file.

The library uses the personality model for many purposes, including analyzing drive error codes, using the drive parameters list, obtaining the list of recording signals, etc.

It is recommended to run the method `CreatePersonalityModel` in the communication object, giving it the personality file path.

Chapter 3: Uploads and Downloads

Upload and Download Process Activation

The communication object implements the interface which contains the methods which activate all the possible uploads and downloads processes:

- Upload personality
- Upload parameters in binary format
- Upload parameters in textual format
- Upload user program
- Download parameters in binary format
- Download parameters in textual format
- Download user program
- Download firmware
- Download PAL

Each upload / download activation method returns an object implementing the interface `IUploadDownloadModel`.

In order to start the process, call the model's `Start` method.

You can cancel the upload / download process by calling the model's `Cancel` method. Please note only some of the upload / download methods can be canceled, and some of them will still affect the drive even if canceled in the middle.

Upload and Download Process Monitoring

During the upload / download process, the model will send the progress events which will enable the process monitoring:

- OnStart – indicates the process has started
- OnProgress – indicates progress in the process
- OnFinish – indicates the process has finished successfully
- OnFailed – indicates the process has failed
- OnCancel – sent when the process was canceled

The events do not contain any information, all information regarding the process status and progress is located in the model.

Chapter 4: Drive Recording

Recording Setup

Before you start recording you must configure the recording setup, which is contained in the class `RecordingSetup`.

The `RecordingSetup` class contains the following properties which are essential for the recording:

- TriggerSetup – defines the recording's trigger setup
- TimeResolution – defines the recording resolution or sampling rate in the drive, i.e. every how many cycles the drive should record a sample
- RecordingLength – defines the number of samples taken during the recording
- SignalData – a list of class RecordingSignalSetup instances, each defines a signal to be recorded

Recording Activation

The communication class contains a method called `GetRecordingObject`, which returns an instance of a class implementing the `IDriveRecording` interface.

This object contains a method called `ConfigureRecording`, which allows you to configure the recording with the `RecordingSetup` object you prepared.

After that you can start the recording by calling the recording object's `StartRecording` method.

The recording can be stopped at any phase by calling the recording object's `StopRecorder` method.

Recording Process Monitoring

In order to monitor the recording process you need to establish your own worker or thread which calls the recording object's `GetRecordingStatus` method in a loop.

This method returns an enumeration of type `RecordingStatus` with the following possible values:

- Roff – indicates recording has stopped in the middle due to error or cancelation
- Rwait – indicates the drive is waiting for trigger
- REnd – indicates recording was completed successfully
- RProgress – indicates recording is in progress

Uploading the Recording Data

When recording has reached the `REnd` status, you can upload the recording data.

This is done by calling the recording object's `UploadRecordingData` method, which returns an instance of class `RecordingData`.

The `RecordingData` class contains a `Dictionary` of the recorded signals data where the keys are the signals IDs and the values are a list of double variables containing each signal's recorded data.

Chapter 5: User Program Compilation

Compilation Settings

Setting up the compilation is done by creating an instance of the class `CompilerSettings`.

This class contains, among others, the following properties needed for the compilation:

- ProgramPath – the path to the program ("*.ehl") file to be compiled
- ProgramImagePath – the path to the compiled image created in the compilation
- CompilerFolder – the path to the folder where the compiler should be located
- PersonalityPath – the path to the personality file of the drive version according to which the compilation will be done

Compilation Process

Basically, the compiler takes the program source from the `ProgramPath` property of the settings object, compiles it and creates the image file in the location indicated by the `ProgramImagePath` property of the settings object.

The actual compilation is done by a compiler which sits in the `CompilerFolder` folder. This folder and its content is built according to a specific drive firmware version using the drive personality file.

The .NET library compiler can also build the specific drive firmware version drive compiler if needed.