FAULHABER

Library Manual

Communications API

MomanLib

Imprint

Version:
2nd edition, 8.10.2022

Software status:
V2.0, according to "FAULHABER Motion Manager 6.9"

This document has been prepared with care.
Dr. Fritz Faulhaber GmbH & Co. KG cannot accept any liability for any errors in this document or for the consequences of such errors. Equally, no liability can be accepted for direct or consequential damages resulting from improper use of the equipment.

The relevant regulations regarding safety engineering and interference suppression as well as the requirements specified in this document are to be noted and followed when using the software.

Subject to change without notice.

The respective current version of this technical manual is available on FAULHABER's internet site:
www.faulhaber.com

Content

Content

1 About this document

1.1 Validity of this document

This document describes the **MomanLib** application programming interface for programming control software for FAULHABER Motion Controllers under Microsoft Windows.

This document is intended for trained programmers and computer scientists.

1.2 Associated documents

For the use of the API, additional information from the following manuals is useful:

Manual	Description
Motion Manager 6	Operating instructions for FAULHABER Motion Manager PC software
Communications manual	Description of communication with the drive
Drive functions	Description of the operating modes and functions of the drive

These manuals can be downloaded in pdf format from the web page www.faulhaber.com/manuals

.

1.3 Symbols and designations

Instructions for understanding or optimizing the operational procedures

✓ Pre-requirement for a requested action

1. First step for a requested action
   ↳ Result of a step
2. Second step of a requested action

↳ Result of an action

▸ Request for a single-step action

1.4 List of abbreviations

Abbreviation	Meaning
API	Application Programming Interface
CAN	Controller Area Network
CiA	CAN in Automation e.V.
COM	Serial RS232 interface
DLL	Dynamic Link Library
MC	Motion Controller
NMT	CANopen network management
OD	Object dictionary
PC	Personal Computer
USB	Universal Serial Bus

1.5 Legal notices

Industrial property rights
In publishing the Motion Manager Library Dr. Fritz Faulhaber & Co. KG does not expressly or implicitly grant any rights in industrial property rights on which the applications and functions of the Motion Manager Library described are directly or indirectly based, nor does it transfer rights of use in such industrial property rights.

Not a constituent part of contract; non-binding character of the Motion Manager Library
Unless otherwise stated the Motion Manager Library is not a constituent part of contracts concluded by Dr. Fritz Faulhaber & Co. KG. The Motion Manager Library is a non-binding description of a possible application example. In particular Dr. Fritz Faulhaber & Co. KG does not guarantee and makes no representation that the processes and functions illustrated in the Motion Manager Library can always be executed and implemented as described and that they can be used in other contexts and environments with the same result without additional tests or modifications.

No liability
Owing to the non-binding character of the Motion Manager Library Dr. Fritz Faulhaber & Co. KG will not accept any liability for losses arising in connection with it.

Changes in the Motion Manager Library
The Motion Manager Library is subject to changes. The current version of this Motion Manager Library may be obtained from Dr. Fritz Faulhaber & Co. KG by calling +49 7031 638 688 or sending an e-mail to mcsupport@faulhaber.de.

2 Description

The **MomanLib** API makes a uniform function interface available for communicating with FAULHABER Motion Controllers via different interfaces (USB, RS232, CAN).

The API can be used to program 32-bit control software under Microsoft Windows with a programming language selected by the developer (e.g. C++, C#, Delphi, LabVIEW).

The API functions are written in C++. To use with other programming languages, the API functions must be integrated using appropriate wrapper functions. Examples are provided for common programming languages.

2.1 Architecture

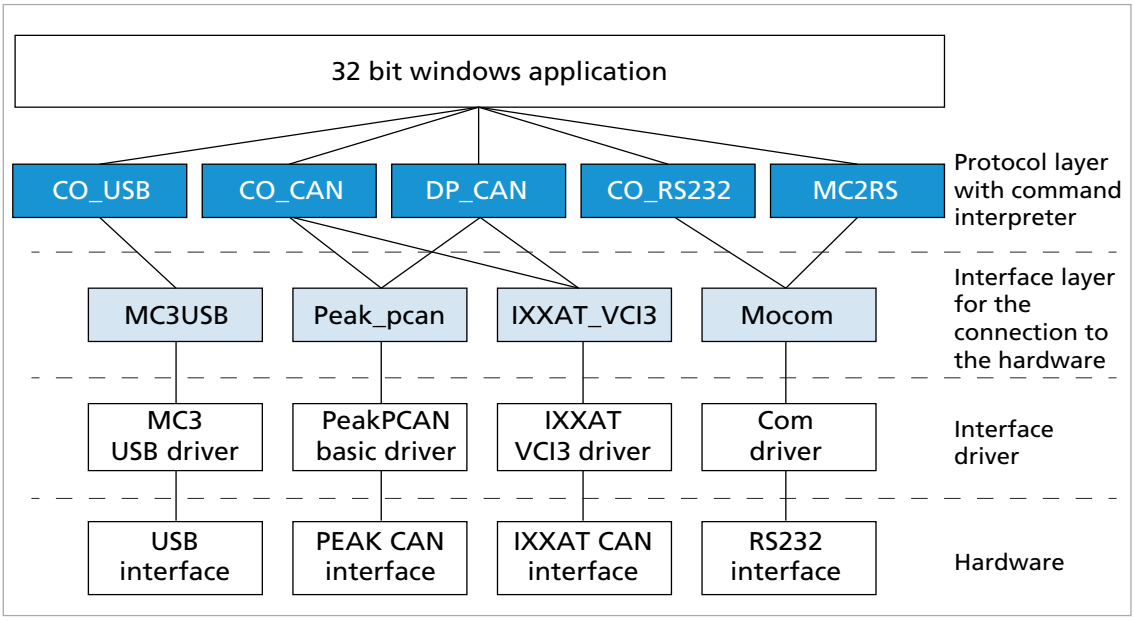


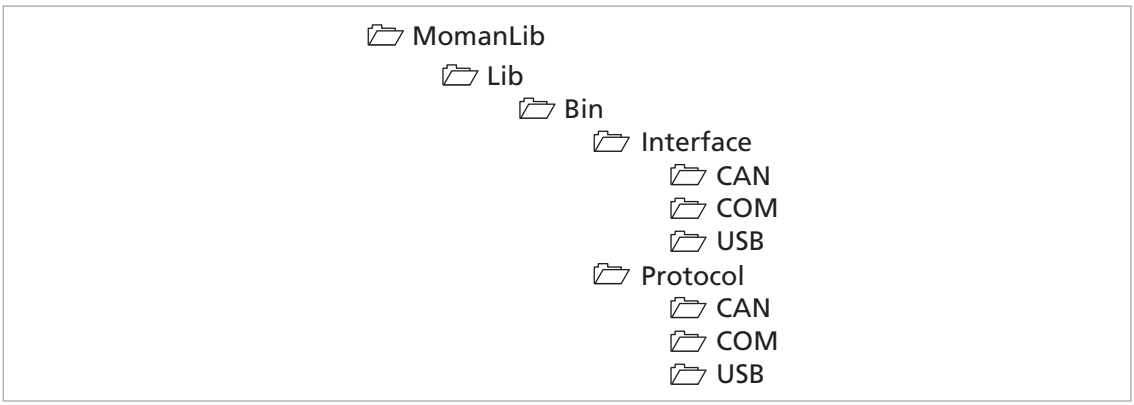
Fig. 1: Architecture of the API

The two-layer plug-in architecture permits the support of different communication protocols and interface cards for the RS232, USB and CAN interface.

Each layer is represented by a DLL file, a protocol DLL and an interface DLL. Communication between the layers takes place via a defined function interface.

2.2 Files

The files needed for using the API are contained in this packet in subdirectory \Moman-Lib\Lib\Bin. The Bin directory contains the \Protocol and \Interface subdirectories, each of which contains the respective subdirectories for the supported interfaces with the corresponding DLL files:



To use the API in your own programs, it is recommended that the required files be copied to a separate project directory and used from there.

Two DLL files are always needed:

- Protocol DLL
- Interface DLL

Depending on the type of protocol and interface, different DLL files are available:

Tab. 1: Protocol DLL

Interface	Protocol DLL	Meaning
RS232	MC2RS	Serial protocol for Motion Controllers of the family MC V2.x
	CO_RS232	CO protocol via RS232 for Motion Controllers of the family MC V3.x
CAN	CO_CAN	Standard CANopen-Protocol via CAN
USB	CO_USB	CO protocol via USB for Motion Controllers of the family MC V3.x

Tab. 2: Interface DLL

Interface	Interface DLL	Function
RS232	Mocom.dll	Connection to standard serial COM port
CAN	Ixxat_vci3.dll	Connection to HMS-IXXAT VCI3- / VCI4 driver
	Peak_pcan.dll	Connection to PEAK PCAN driver
	Ems_cpc.dll	Connection to EMS CPC driver
	Esd_ntcan.dll	Connection to ESD NTCAN driver
	...	Other CAN interface DLLs if necessary
USB	MC3Usb.dll	Connection to FAULHABER MC V3.x USB driver

USB
The driver for accessing the FAULHABER Motion Controllers of the family MC V3.x is installed with Motion Manager 6.

RS232
When using a serial interface installed in a PC, no other drivers are needed. If a USB to serial adapter is used, it is generally necessary to install the supplied driver. The adapter then appears as a virtual COM port, the port number of which can be found in the Windows Device Manager (e.g., COM5).

CAN
The driver of the used CAN interface card appropriate for the selected interface DLL must be installed separately. Currently supported are interface cards from the manufacturers listed in Tab. 2. CAN interface cards from other manufacturers can be used if an interface DLL exists for the given card. Inquire with FAULHABER for further information.

For some interfaces an additional DLL is needed in the application directory, e.g. for PEAK the PCANBasic.dll. You can find the additional DLL files in the folder \Examples\Win32\Common.

3 Integration in the application

The protocol DLL can be integrated statically or dynamically in Windows applications that were developed with Win32 development tools (e.g. C/C++, Delphi, Visual Basic, LabVIEW).

The desired DLL files (protocol DLL and interface DLL) must be copied to the project directory or a suitable subdirectory.

In addition, the functions that are to be used, which are defined in C header file **Momanprot.h**, must be made known to the application. To use these functions, the definitions from file **Momancmd.h** are needed, which must likewise be made known to the application. The C header files are located in directory \MomanLib\Lib\Include.

The desired interface DLL must first be initialized in the program via the **`mmProtInitInterface()`** function. Afterwards, the interface can be opened, commands sent and answers read. Lastly, the interfaces must be closed again.

3.1 Typical call sequence

3.1.1 Synchronous communication

1. Initialisation

   Connection to a Motion Controller of the family MC V3.x via USB:

   ```
   mmProtInitInterface("MC3Usb.dll", NULL, NULL);
   mmProtOpenCom(1, 0, 0);
   ```

2. Data exchange

   Read "Device Name" parameter of node 1:

   ```
   const char* answData = NULL;
   mmProtGetStrObj(1, 0x1008, 0x00, &answData);
   ```

3. Close interfaces:

   ```
   mmProtCloseCom();
   mmProtCloseInterface();
   ```

3.1.2 Asynchronous communication

1. Initialisation

   Connection to a Motion Controller of the family MC V3.x via an HMS-IXXAT-CAN card with 250 kBit/s:

```
mmProtInitInterface("Ixxat_vci3.dll", &CBDataReceived, NULL);
mmProtOpenCom(1, 0, 250000);
hReceiveEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

2. Data exchange

   Switch node 1 to the "Operation Enabled" state and receive data asynchronously:

```
mmProtSendCommand(1, 0x0000, eMomancmd_start, 0, 0);
mmProtSendCommand(1, 0x0000, eMomancmd_shutdown, 0, 0);
mmProtSendCommand(1, 0x0000, eMomancmd_switchon, 0, 0);
mmProtSendCommand(1, 0x0000, eMomancmd_EnOp, 0, 0);
// Callback function for the signalling of data reception:
void CBDataReceived(void)
{
   SetEvent(hReceiveEvent);
}
```

3. Read data (here in a separate thread with event signalling):

```
if (WaitForSingleObject(hReceiveEvent, INFINITE) == WAIT_OBJECT_0)
{
   const char* answData = NULL;
   const char* cmdString = NULL;
   const char* receiveTelegram = NULL;
   int nodeNr;
   mmProtReadAnswer(&answData, nodeNr, &cmdString, &receiveTelegram);
}
```

4. Close interfaces:

```
mmProtCloseCom();
mmProtCloseInterface();
CloseHandle(hReceiveEvent);
```

3.2 C/C++

For the static integration, a suitable import library of the protocol DLL (*.lib) must be added to the project. Because the import libraries differ in format depending on the used development environment, this method is not used in the examples. For some development environments, there is a tool for creating an import library from a DLL (see documentation of the used development environment).

For dynamic integration, the Win32 function **`LoadLibrary()`** must first be used to load the desired protocol DLL and then each function that is to be used given an equivalent function name via **`GetProcAddress()`**. At the end of use, the DLL must again be removed from memory using **`FreeLibrary()`**.

To access the DLL functions, the header files **Momanprot.h** and **Momancmd.h** must be copied to the project directory and added to the source text file with `#include`.

Example (\MomanLib\Examples\Source\C++)

- \Common\MomanLibSample

  Shows the dynamic integration of the library in a standard C++ program and the use of individual functions

- \C++Builder\DemoVCL

  C++ Builder project for Embarcadero RAD Studio 10 with VCL user interface for the use of MomanLibSample

- \Visual C++\DemoVCPP

  Visual C++ project for Microsoft Visual Studio 2012 with "Windows Forms" user interface for the use of MomanLibSample

3.3 Delphi

For the static integration, the DLL functions to be used must be declared as **external** in the source text file with the `stdcall` calling convention. This preferred method is also used in the example.

The library can also be loaded dynamically via Win32 function **`LoadLibrary()`**. The functions are then integrated as with C++ via **`GetProcAddress()`**.

To use the DLL functions, some definitions are needed from the **Momancmd.h** file and must be specified in the source text file as `type`. When using callback functions (asynchronous communication), note that they must be declared with the calling convention CDECL.

Example (\MomanLib\Examples\Source\Delphi)

- \MomanLibSample

  Shows the static integration and use of individual functions of the library in a Delphi program

- \Demo

  Delphi project for Embarcadero RAD Studio 10 with VCL user interface for the use of MomanLibSample

3.4 C#

For the connection, a wrapper class must be created with the `DllImport` references to the required functions of the specified DLL. `CallingConvention=CallingConvention.StdCall` must be specified as the calling convention.

To use the DLL functions, some definitions are needed from the **Momancmd.h** file and must be specified in the source text file. When using callback functions (asynchronous communication), note that they must be declared with the calling convention CDECL.

Example (\MomanLib\Examples\Source\C#)

- \MomanLibSample

  Shows the integration and use of individual functions of the library in a C# program

- \DemoCSharp

  C# project for Microsoft Visual Studio 2012 with "Windows Forms" user interface for the use of MomanLibSample

3.5 LabVIEW

For integration, the path of the DLL file must be specified in "Call Library Function Node" and the DLL function that is to be used selected and configured. `stdcall` must be specified as the calling convention.

To use these functions, "Specify path on diagram" must be selected. In addition, a number of definitions from file **Momancmd.h** must be specified. When using callback functions (asynchronous communication), note that they must be declared with the calling convention CDECL.

When using the automatic import function, it must be ensured that the import declarations are imported correctly. An option is to reliably capture the corresponding return values.

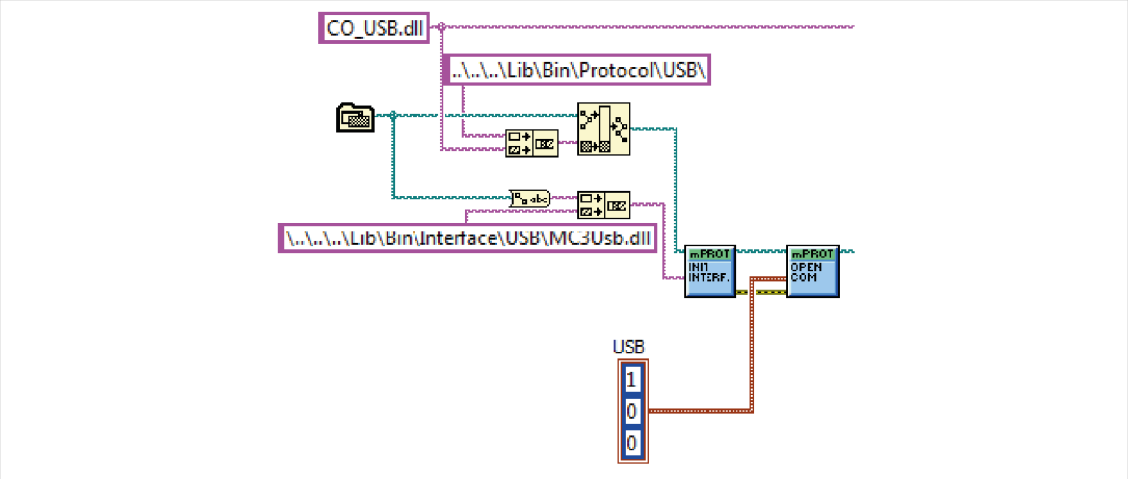


Fig. 2: Initialisation with wrapper component

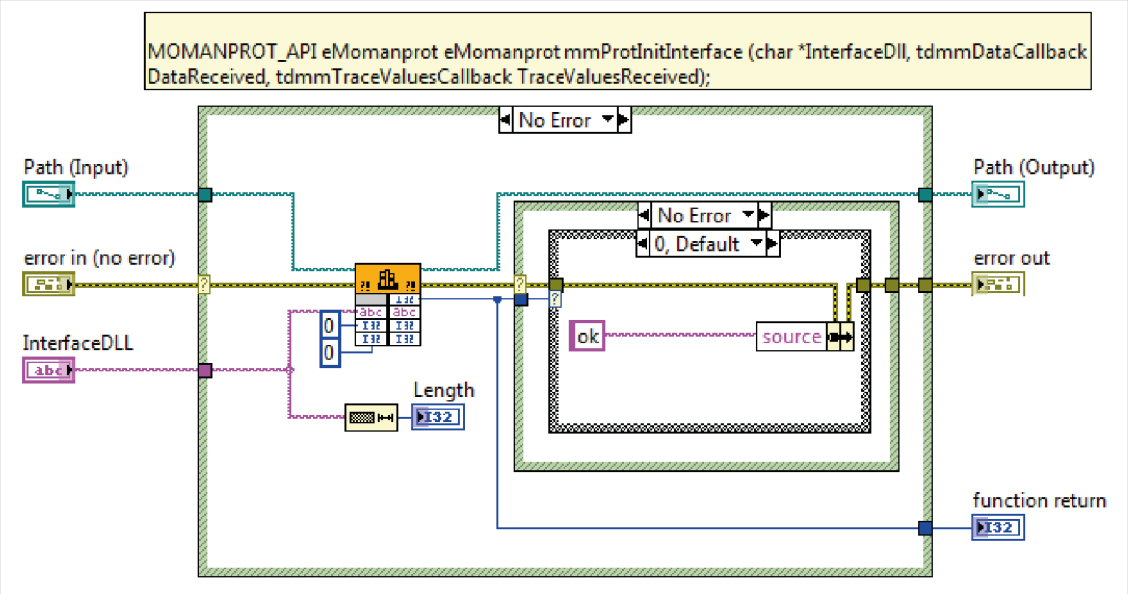


Fig. 3: Implementation of a wrapper component

In this example, asynchronous communication (see chap. 3.1.2, p. 11) is only prepared. For use with LabView, an extension of the API interface is necessary.

Example (\MomanLib\Examples\Source\Labview)

- Project for LabView 2015 with user interface and main loop
- \Typedefinition

  Definitions of used API parameters
- \SubVIs
  - \Object-convert.vi, \Command_format_WRITE.vi

    Preformatting of the API parameters
  - \mmProt___.vi

    Wrappers for each "Call Library Function Node" function block
  - \Async___.vi, \.NET_Init.vi

    Preparations for the use of the asynchronous function

4 Examples

Simple examples are available for the programming languages listed in chap. 3, p. 10.

The source code for the individual programming languages can be found in subfolder \Examples\Source. The underlying C-header files can be found in \Lib\Include. For some programming languages, there is a file called *MomanLibSample*, which loads the communication DLLs and encapsulates access to the API functions in appropriate wrapper functions. In addition, the required definitions are implemented here from the **Momancmd.h** file for programming languages other than C/C++.

Located in subfolder \Examples\Win32 are the executable EXE files produced with the respective programming languages; these EXE files use the communication DLLs in folder \Lib\Bin.

The examples are structured so as to be easy to understand. To facilitate better readability of the code, the examples do not include adequate error handling and other modularisation. The examples should be adapted to your software structure prior to use.

The examples illustrate the following aspects:

- Synchronous access to objects in the object dictionary
- Operation of the device-control state machine with asynchronous data reception
- Execution of relative positioning

All examples use USB as communication interface to a Motion Controller of the family V3.x. To execute the examples, a USB connection must be established to a Motion Controller of the family V3.x. If the connection is to be established via a different interface, the necessary protocol and interface DLLs must be assigned the respective constants in the source code.

Before executing an example program, the Motion Controller must be adapted to the connected motor via the Motion Manager.

5 API documentation

The protocol DLL contains the interface to the application. This is also used to initialise the interface DLL.

5.1 General

In the **Momancmd.h** file, several enumerators and structures are defined that are used as return or transfer parameters of a number of functions.

The function description specifies which enumerators and structures the respective functions use. This file can be integrated directly in C/C++ programs with `#include`. For other programming languages, the used enumerators and structures must be made known accordingly.

The API definition for the C/C++ programming language is stored in the **Momanprot.h** file.

Some functions expect a double pointer `const char**` as transfer parameter. This pointer allows strings whose memory is managed in the protocol layer to be read out. For the further processing of these strings, the referenced memory area must be copied to a suitable string variable immediately after calling the function.

Examples

C++: Function parameter defined as (`const char** data`)

Function call:

```
const char* data = NULL;
DoSomething(&data);
if (data != NULL) {
    std::string s = data;
}
```

C#: Function parameter defined as (`out IntPtr data`)

Function call:

```
IntPtr data = IntPtr.Zero;
DoSomething(out data);
if (data != IntPtr.Zero) {
    string s = Marshal.PtrToStringAnsi(data)
}
```

Delphi: Function parameter defined as (*data:* PAnsiChar)

Function call:

```
s: String;
data: PAnsiChar;
data := nil;
DoSomething(@data);
if data <> nil then begin
     s = String(data)
end;
```

LabVIEW: Function parameter defined as (uintptr_t *data)

Function call ("Call Library Function Node"):

GetValueByPointer VI

```
Type: Numeric
Datatype: unsigned Pointer-sized Integer
Pass: Pointer to Value
```

The Motion Manager API generally uses 8-bit ANSI strings. To use Unicode strings, appropriate conversions must be performed.

5.2 Initialisation

The initialisation functions are necessary for establishing a connection between PC software and the Motion Controller.

5.2.1 InitInterface

```
eMomanprot mmProtInitInterface ( char*                   InterfaceDll,
                                 tdmmDataCallback        DataReceived,
                                 tdmmTraceValuesCallback TraceValuesReceived
                               )
```

Initialisation of the interface connection.

Parameter

[in] *InterfaceDll* — File name of the interface DLL with path

[in] *DataReceived* — Callback function that is called when data are received. The received data must then be read out via the **ReadAnswer** function.

ZERO, if asynchronous data reception is not used.

[in] *TraceValuesReceived* — Callback function that is called following analysis of received trace data and which passes the trace data.

ZERO, if the trace function is not used.

Return data

eMomanprot_ok — Interface DLL successfully initialised

eMomanprot_error — Error when loading the interface DLL

To use the callback functions, the respective function pointers must be declared:

```
typedef void (*tdmmProtDataCallback)(void);
typedef void (*tdmmProtTraceValuesCallback)(int nodeNr,
             unsigned int value[], int timecode);
```

These callback functions are called in the receive thread. Thus, processing operations that are relatively long may not be performed here.

The callback functions serve only to signal data reception. The further processing of the received data, which is cached in the protocol layer, should occur in another thread (e.g., the main thread of the application).

Example

```
tdmmProtDataCallback DataReceived;
tdmmProtTraceValuesCallback TraceValuesReceived;
std::string InterfaceDll = "Ixxat_vci3.dll";
eMomanprot ret = mmProtInitInterface((char*)InterfaceDll.c_str(),
     DataReceived, TraceValuesReceived);
```

5.2.2 CloseInterface

```
void mmProtCloseInterface (void)
```

Close interface connection and release memory.

5.2.3 OpenCom

```
eMomanprot mmProtOpenCom ( int   port,
                           int   channel,
                           int   baud
                         )
```

Open interfaces.

Parameter

[in] *port* Port number

COM: 0

USB: Sequential beginning with 1 for each USB device on the selected interface

CAN: Sequential beginning with 1 for each CAN card on the selected interface

[in] *channel* Channel number

COM: COM number, e.g. 8 for COM8

USB: 0

CAN: 0 for port with just one channel, otherwise sequential beginning with 1

[in] *baud* Baud rate in bit/s

COM: e.g. 9600

USB: 0

CAN: e.g. 250000

Return data

eMomanprot_ok Interface successfully opened

eMomanprot_error Error when opening the interface

See chap. 6.1, p. 53 for determining the available ports and channels.

See chap. 5.6.8, p. 52 for determining the supported baud rates.

Example

Open first MC V3.x USB port:

```
eMomanprot ret = mmProtOpenCom(1, 0, 0);
```

5.2.4 CloseCom

```
void mmProtCloseCom (void)
```

Close interface.

5.2.5 ScanNode

```
int mmProtScanNode ( int          nodeNr,
                     const char** deviceName,
                     int&         deviceMode
                   )
```

Find network nodes.

Parameter

[in] *nodeNr* Node number
–1: Broadcast

[out] *deviceName* Read device name

[out] *deviceMode* eDeviceMode device property

eDeviceMode_Faulhaber	FAULHABER drive
eDeviceMode_Bootloader	FAULHABER drive in bootloader mode
eDeviceMode_AnyDrive	Non-FAULHABER drive
eDeviceMode_AnyDevice	Other device, not a drive

Return data

≥0 Node number, if node was found

–1 No node found under the specified node number

A check is performed to determine whether a node exists with the specified node number. If only one node with an unknown node number is connected, the node number can also be determined via a broadcast call.

Example

```
std::string deviceName;
int deviceMode;
const char* name = NULL;
int foundNodeNr = mmProtScanNode (-1, &name, deviceMode);
if (name != NULL) deviceName = name;
```

5.2.6 LoadCommandSet

`int` **`mmProtLoadCommandSet (`**`int` *`cmdType`*`)`

- Load command set of a CAN device type, provided it is not an MC V3.x.
- Read currently loaded command set (*`eCmdType_LoadedCommandSet`*).

Parameter

[in] *`cmdType`* Number of the device type command set (eCmdType):

`eCmdType_LoadedCommandSet`	Read loaded command set
`eCmdType_Default`	Load command set MC V3.x
`eCmdType_MC3`	Load command set MC V3.x
`eCmdType_CO`	Load command set MC V2.x CO
`eCmdType_CF`	Load command set MC V2.x CF

Return data

eCommandSet_invalidCmdType	Invalid transfer parameter
eCommandSet_Unknown	Unknown command set
eCommandSet_MC3RS	MC V3.x RS232 command set
eCommandSet_MC3USB	MC V3.x USB command set
eCommandSet_MC3CAN	MC V3.x CAN command set
eCommandSet_MC2RS	MC V2.x RS command set
eCommandSet_MC2CAN	MC V2.x CAN command set

5.2.7 SetDataCallback

`void` **`mmProtSetDataCallback (`**`tdmmProtDataCallback` *`DataReceived`***`)`**

Set callback function for signalling data received asynchronously; alternative to specifying for **`InitInterface`**. A callback function can hereby be set or deleted at any point in time independent of the initialisation of the interface.

Parameter

[in] *`DataReceived`* Callback function that is called when data are received. The received data must then be read out via the **`ReadAnswer`** function.

ZERO, if asynchronous data reception is to be switched off.

To use the callback function, a corresponding function pointer must be declared:

```
typedef void (*tdmmProtDataCallback)(void);
```

This callback function is called in the receive thread. Thus, processing operations that are relatively long may not be performed here.

The callback function serves only to signal data reception. The further processing of the subsequently read data should occur in another thread (e.g., the main thread of the application).

5.2.8 SetTraceValuesCallback

`void` **`mmProtSetTraceValuesCallback (`**`tdmmProtTraceValuesCallback` *`TraceValues-Received`***`)`**

Set callback function for signalling trace data received asynchronously; alternative to specifying for **`InitInterface`**. A callback function can hereby be set or deleted at any point in time independent of the initialisation of the interface.

Parameter

[in] *`TraceValuesReceived`* Callback function that is called following analysis of received trace data and which passes the trace data.

ZERO if the trace function is to be switched off.

To use the callback function, a corresponding function pointer must be declared:

```
typedef void (*tdmmProtTraceValuesCallback)(int nodeNr,
             unsigned int value[], int timecode);
```

This callback function is called in the receive thread. Thus, processing operations that are relatively long may not be performed here.

The further processing of the received data should occur in another thread (e.g., the main thread of the application).

5.3 Accessing the object dictionary

The functions for accessing the object dictionary offer simple methods for synchronously reading and writing objects. These methods are not supported by Motion Controllers of the family MC V2.x RS.

The values to be passed for *index*, *subindex* and *len* can be determined in the following ways:

- From the Communications Manual or the Functional Manual of the respective Motion Controller
- Via the Motion Manager:

  All objects supported by the device are listed in the object browser of the Motion Manager. The value range and the data type for the *len* parameter of the selected object can be read from the corresponding status bar.

  The command sequence, which must be sent for specific actions, can be found in the recording in the terminal log of the Motion Cockpit.

5.3.1 GetObj

```
int mmProtGetObj ( int  nodeNr,
                   int  index,
                   int  subIndex,
                   int& value
                 )
```

Read integer object entry from object dictionary.

Parameter

[in] *nodeNr* Node number

[in] *index* Index of the object

[in] *subIndex* Subindex of the object

[in] *value* Read integer value

Return data

eMomanprot_ok	Answer successfully received
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device → CiA error code is returned via *value*
eMomanprot_error	Other error

Example
Query actual position of node 1:

```
int value;
eMomanprot ret = mmProtGetObj (1, 0x6064, 0x00, value);
```

5.3.2 GetInt64Obj

```
int mmProtGetInt64Obj ( int      nodeNr,
                        int      index,
                        int      subIndex,
                        _int64&  value
                      )
```

Objects that are linked to a Momancmd are read out from the object dictionary as 64-bit integers. It is thereby possible to distinguish between `unsigned int32` and `signed int32`. All other objects are returned as `signed int32`, analogous to **`GetObj`**.

Parameter

[in]	*nodeNr*	Node number
[in]	*index*	Index of the object
[in]	*subIndex*	Subindex of the object
[out]	*value*	Read value as Int64

Return data

eMomanprot_ok	Answer successfully received
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device → CiA error code is returned via *value*
eMomanprot_error	Other error

5.3.3 GetStrObj

```
int mmProtGetStrObj  ( int           nodeNr,
                       int           index,
                       int           subIndex,
                       const char**  value
                     )
```

Read string object entry from object dictionary.

Parameter

[in] *nodeNr* Node number

[in] *index* Index of the object

[in] *subIndex* Subindex of the object

[out] *value* Read string value

Return data

eMomanprot_ok	Answer successfully received
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device → CiA error code returned via *value* as hexadecimal value
eMomanprot_error	Other error

Example

Query software version of node 1:

```
std::string version;
const char* ver = NULL;
eMomanprot ret = mmProtGetStrObj (1, 0x100A, 0x00, &ver);
if (ver != NULL) version = ver;
```

5.3.4 SetObj

```
int mmProtSetObj ( int           nodeNr,
                   int           index,
                   int           subIndex,
                   int           value,
                   int           len,
                   unsigned int& abortCode
                 )
```

Write integer object entry in the object dictionary.

Parameter

[in]	*nodeNr*	Node number
[in]	*index*	Index of the object
[in]	*subIndex*	Subindex of the object
[in]	*value*	New value
[in]	*len*	Data length of the object entry in bytes
[out]	*abortCode*	CiA error code for *eMomanprot_error_cmd*

Return data

eMomanprot_ok	Value successfully transferred
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device
eMomanprot_error	Other error

Example

Target position = preset 1000 for node 1:

```
unsigned int abortCode;
eMomanprot ret = mmProtSetObj (1, 0x607A, 0x00, 1000, 4, abortCode);
```

5.3.5 SetStrObj

```
int mmProtSetStrObj ( int           nodeNr,
                      int           index,
                      int           subIndex,
                      char*         value,
                      unsigned int& abortCode
                    )
```

Write string object entry in object dictionary.

Parameter

[in] *nodeNr* Node number

[in] *index* Index of the object

[in] *subIndex* Subindex of the object

[in] *value* New value

[out] *abortCode* CiA error code for *eMomanprot_error_cmd*

Return data

eMomanprot_ok	Value successfully transferred
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device
eMomanprot_error	Other error

5.3.6 SetObjTimeout

```
void mmProtSetObjTimeout (int timeout)
```

For object queries: set timeout for commands that require a longer execution time (e.g., save, restore or reset). The timeout value is set to 500 ms by default.

Parameter

[in] *timeout* ≥500: timeout value in ms
<500: default value (500 ms)

5.3.7 GetAbortMessage

```
const char* mmProtGetAbortMessage (unsigned int abortCode)
```

Read out error message text of the passed abort code.

Parameter

[in] *abortCode* Error code

Return data

Text of the abort message

5.4 Advanced communication

The advanced communication functions are available for commands without object dictionary and for asynchronous data reception. In addition, functions exist for the analysis of communication data, for evaluating the communication history and for setting a CAN message filter.

5.4.1 SendMotionCommand

```
bool mmProtSendMotionCommand  ( char* cmd,
                                int   nodeNr
                              )
```

Send ASCII motion command.

Parameter

[in] *cmd* ASCII command

[in] *nodeNr* Node number, if not contained in the command

Return data

true Command sent successfully

false Error when sending

The commands listed in the Motion Manager command reference can be used as ASCII commands.

For Motion Controllers of the family MC V2.x with an RS interface, all commands listed in the Functional Manual of the respective control can be used; these are sent directly to the drive control. These commands can also be used for Motion Controllers of the family MC V2.x with a CF interface, in which case they are sent via a CAN telegram.

The program does not wait for an answer. The **WaitAnswer** function is available for this purpose. If a DataReceived callback function is specified for **InitInterface** and an answer is received asynchronously, the DataReceived callback function is called. The answer can then be read using **ReadAnswer**.

5.4.2 SendCommand

```
bool mmProtSendCommand ( int   nodeNr,
                         int   index,
                         int   subIndex,
                         int   dataLen,
                         int   data
                       )
```

Send object command.

Parameter

[in] *nodeNr* Node number

[in] *index* Index of the data object

[in] *subIndex* Subindex of the data object

[in] *dataLen* Data length and command type:

- 0: Query command for standard objects (SDO Upload) or Motion Manager command without data (e.g., NMT, Device Control)
- –1: Query command for long objects (segmented SDO Upload)
- 1…4: Send command for standard objects (SDO Download)
- >4: Send command for long objects (segmented SDO Download)

[in] *data* Data value to be sent or pointer to data block

0, when *dataLen* = 0

Return data

true Command sent successfully

false Error when sending

This function can be used to read or write parameters in the object dictionary without waiting for the answer. The answer can be read by subsequently calling **WaitAnswer** (e.g., reading a data buffer). If a DataReceived callback function is specified for **InitInterface** and an answer is received asynchronously, the DataReceived callback function is called. The answer can then be read using **ReadAnswer**.

The functions in chap. 5.3, p. 23 are available for synchronously reading and writing objects. To send data buffer objects, the **SendBuffer** function should be used.

In addition to the objects defined in the device, special Motion Manager commands are available at index 0x0000 that can be used for standard tasks, NMT and Device Control (see `enum eMomancmd` in **Momancmd.h**).

5.4.3 SendBuffer

```
bool mmProtSendBuffer ( int   nodeNr,
                        int   index,
                        int   subIndex,
                        int   dataLen,
                        char* dataBuffer
                      )
```

Send buffer content or string via object command.

Parameter

[in] *nodeNr* Node number

[in] *index* Index of the data object

[in] *subIndex* Subindex of the data object

[in] *dataLen* Data length in bytes

[in] *dataBuffer* Pointer to the data block that is to be sent

Return data

true Command sent successfully

false Error when sending

This function can be used to write string and data buffer objects in the object dictionary.

If the function call fails, error code and error text can be read out via **WaitAnswer** if necessary.

5.4.4 SendTelegram

```
bool mmProtSendTelegram ( int   id,
                          char* data,
                          int   len
                        )
```

Send any telegram.

The specified data are sent directly to the specified COB-ID (CAN) or node number (RS232, USB). The telegram structure is described in the Communications Manual.

Is not supported by Motion Controllers of the family MC V2.x RS.

Parameter

[in] *id* COB-ID or node number

[in] *data* Data field

[in] *len* Length of the data field

Return data

true Telegram sent successfully

false Error when sending

5.4.5 WaitAnswer

```
eMomanprot mmProtWaitAnswer ( int          timeout,
                              int          answ,
                              const char** answData
                            )
```

Wait for answer from device.

Parameter

[in]	*timeout*	Timeout time in ms
[in]	*answ*	Answer mode (eWaitMode)
[out]	*answData*	Answer to previously sent command If *eMomanprot_error_cmd* is returned, *answData* contains an error text.

Return data

eMomanprot_ok	Answer successfully received
eMomanprot_error_timeout	No response
eMomanprot_error_cmd	Error message from device
eMomanprot_error_param	Invalid transfer parameter
eMomanprot_error	Other error
>4	Length of the received binary data packet

This function is available for synchronous communication with the **SendMotionCommand**, **SendCommand** and **SendBuffer** functions.

Transfer parameter *answ* can contain one of the following values:

eWaitMode_Int	Waiting for an integer answer
eWaitMode_String	Waiting for a string answer
eWaitMode_BinData	Waiting for a binary data buffer > 4 bytes
eWaitMode_noAsync_noAck	Waiting for MC V2.x RS answer without async. and ack.
eWaitMode_noAck	Waiting for MC V2.x RS answer without acknowledge
eWaitMode_noAsync	Waiting for MC V2.x RS answer without async.
eWaitMode_someAsync	Waiting for MC V2.x RS answer with specific async.
eWaitMode_Bin	Waiting for MC V2.x RS binary answer When waiting for trace data, 9 bytes are always returned: the first 4 bytes for trace parameter 1, the next 4 bytes for trace parameter 2, followed by 1 byte for the timecode.
>eWaitMode_Id	Waiting for answer with specific identifier (COB-ID)

Example
Read actual position of an MC V2.x:

```
std::string position;
if (mmProtSendMotionCommand ("POS", 0) == true) {
      const char* data = NULL;
      eMomanprot ret = mmProtWaitAnswer (1000, eWaitMode_String, data);
      if (data != NULL) position = data;
}
```

5.4.6 ReadAnswer

```
eMomanprot mmProtReadAnswer ( const char**  answData,
                              int&          nodeNr,
                              const char**  cmdString,
                              const char**  receiveTelegram
                            )
```

Read received messages.

Following a receipt notification (DataReceived callback), this can be used to read out the messages cached in the ring buffer.

Parameter

[out] *answData*	Answer data interpreted as string
[out] *nodeNr*	Node number of the received message
[out] *cmdString*	Corresponding command string
[out] *receiveTelegram*	Receive telegram

Return data

eMomanprot_ok	Answer successfully received
eMomanprot_error_cmd	Error message from device
eMomanprot_error_emcy	Emergency error from device
eMomanprot_error	Other error
eMomanprot_noData	No data available for reading

ReadAnswer should not be called directly within the DataReceived callback function as this would slow the receive thread.

The signalling of data reception and reading of the data must be decoupled from one another. To decouple, **ReadAnswer** can be called, e.g., in a separate thread (event signalling), in a Windows message handler (signalling via PostMessage) or within a timer event.

The received data can be further examined via the **DecodeAnswStr** and **DecodeCmdStr** auxiliary functions.

Example

```
std::string data, cmd;
int statusword;
const char* answData = NULL;
const char* cmdString = NULL;
const char* receiveTelegram = NULL;
int nodeNr;

eMomanprot ret = mmProtReadAnswer (&answData, nodeNr, &cmdString, &receiveTele-
gram);

if (ret != eMomanprot_noData) {
      if (answData != NULL) data = answData;
      if (cmdString != NULL) cmd = cmdString;
      _int64 value;
      if (mmProtDecodeAnswStr (answData, value) == eDecoded_Statusword) {
            statusword = (int)value;
      }
}
```

5.4.7 DecodeAnswStr

```
eDecoded mmProtDecodeAnswStr ( const char** answStr,
                                _int64&      value
                              )
```

Decode the passed **ReadAnswer** answer string according to specific properties.

Parameter

[in] *answStr* Answer string from **ReadAnswer**

[out] *value* Value contained in the answer string

Return data

eDecoded_none	*answStr* is not decoded
eDecoded_Bootup	*answStr* is a boot-up message
eDecoded_NMT	*answStr* is an NMT message
eDecoded_NMTRequest	*answStr* is an NMT request answer
eDecoded_Heartbeat	*answStr* is a heartbeat message
eDecoded_Statusword	*answStr* is a status word message

5.4.8 DecodeCmdStr

```
eDecoded mmProtDecodeCmdStr ( const char*    cmdStr,
                              int&           nodeNr,
                              int&           index,
                              int&           subIndex,
                              const char**   valueStr
                            )
```

Decode the passed **ReadAnswer** command string. If it is an object command, node number, index, subindex and SOBJ value are returned.

Parameter

[in]	*cmdStr*	Command string
[out]	*nodeNr*	Node number
[out]	*index*	Object index
[out]	*subIndex*	Object subindex
[out]	*valueStr*	Value to be written for SOBJ command

Return data

eDecoded_none	*cmdStr* is not decoded
eDecoded_SOBJ	*cmdStr* is a SOBJ command
eDecoded_GOBJ	*cmdStr* is a GOBJ command

5.4.9 CheckMotionCommand

`int` **`mmProtCheckMotionCommand`** `(char*` *`cmd`*`)`

Check whether the passed ASCII command is linked to a specific action.

Parameter

[in] *`cmd`* ASCII command

Return data

eCheckCommand_noAction	No special action
eCheckCommand_State	Command can change the NMT, device or Opmode state
eCheckCommand_SwitchOn	Command switches the motor on
eCheckCommand_Save	Command is a SAVE command
eCheckCommand_Restore	Command is a RESTORE command
eCheckCommand_Reset	Command is a RESET command

5.4.10 CheckCommand

```
eCheckCommand mmProtCheckCommand ( int  index,
                                   int  subIndex,
                                   int  dataLen,
                                   int  data
                                 )
```

Check whether a specific action is linked to the passed object command.

Parameter

[in] *index* Index of the data object

[in] *subIndex* Subindex of the data object

[in] *dataLen* Data length in bytes

[in] *data* Data value to be sent

Return data

eCheckCommand_noAction No special action

eCheckCommand_State Command can change the NMT, device or Opmode state

eCheckCommand_SwitchOn Command switches the motor on

eCheckCommand_Save Command is a SAVE command

eCheckCommand_Restore Command is a RESTORE command

eCheckCommand_Reset Command is a RESET command

5.4.11 GetCommunicationHistory

```
bool mmProtGetCommunicationHistory ( const char**   timestamp,
                                     eHistoryState& state,
                                     const char**   data,
                                     const char**   telegram,
                                     eHistoryError& error
                                   )
```

Read out the communication history that is cached in the protocol layer.

Can be read out in a loop as long as *true* is returned.

Parameter

[out] *timestamp* Send or receive time stamps

[out] *state* Status of the telegram:

eHistoryState_SendData	Sent data
eHistoryState_ReceiveWaitData	Received expected data
eHistoryState_ReceiveData	Asynchronously received data
eHistoryState_Message	Informative message

[out] *data* Command / interpreted received data

[out] *telegram* Send / receive telegram

[out] *error* Error code:

eHistoryError_Ok	No error
eHistoryError_SendError	Error sending
eHistoryError_ReceiveError	Receive error
eHistoryError_ReceiveTimeout	Timeout
eHistoryError_ReceiveEmcy	Emergency error received

Return data

true Read out buffer content

false Buffer empty

5.4.12 GetSendTelegram

```
void mmProtGetSendTelegram ( const char** sendTelegram,
                             const char** cmdString
                           )
```

Read last transferred send telegram.

Parameter

[out] *sendTelegram* Send telegram

[out] *cmdString* Send command

5.4.13 SetupMessageFilter

```
void mmProtSetupMessageFilter ( int nodeNr,
                                int activated,
                                int cobId,
                                int cobIdCount
                              )
```

Activate a CAN message filter.

The CAN message filter is used to filter asynchronous CAN messages. If the filter is active, COB-ID exceptions can be specified that are still to be allowed through.

Parameter

[in] *nodeNr* Node numbers whose telegrams should always be displayed

[in] *activated* 0: Message filter deactivated
1: Filter out external messages apart from exceptions
2: Filter out internal TxPDOs apart from exceptions
3: Filter out external messages and internal TxPDOs

[in] *cobId* Array with COB-ID exceptions

[in] *cobIdCount* Number of COB-ID exceptions

5.5 Data recording

With the functions for data recording, internal device parameters can be recorded constantly (logger) or buffered (recorder) and read out.

To use the trace function, a callback function must be specified during initialization by means of which the recorded data can be passed to the application asynchronously.

5.5.1 SetupTrace

```
eMomanprot mmProtSetupTrace  ( STraceSetup&    TraceSetup,
                               STraceTrigger&  TraceTrigger
                             )
```

Configure trace.

Parameter

[in] *TraceSetup*	Setting the trace channels
[in] *TraceTrigger*	Trigger settings

Return data

eMomanprot_ok	Successfully configured
eMomanprot_error	Configuration error

Structures

The `STraceSetup` and `STraceTrigger` structures are defined for the trace configuration:

```
typedef struct {
  bool run;           /*!< run or stop tracing */
  int  nodeNr;        /*!< node number to be traced */
  int  chan;          /*!< 0: default channel, 1: logger uses CAN-PDOs */
  int  mode;          /*!< 0: logger(single requests), 1: recorder(buffered) */
  int  source[4];     /*!< trace parameter: OD Index/Subindex */
  int  sourceLen[4]; /*!< data length in byte; 0: source not used */
  int  sourceType[4];/*!< 0: default; CAN logger: cobId of trace pdo;
                        -1: deactivated */
  int  buffer;        /*!< buffer size (number of samples to be recorded) */
  int  sampletime     /*!< sampletime in recorder mode */
} STraceSetup;
```

```
typedef struct {
  int  mode;         /*!< 0: no trigger, 1: singleshot, 2: retriggered */
  int  source;       /*!< trigger parameter: OD Index/Subindex */
  int  sourceType;   /*!< TrigType: 0 = OD */
  int  threshold;    /*!< trigger if source value > or < threshold value */
  int  edge          /*!< 0: rising (>), 1: falling (<) */
  int  delay         /*!< triggerdelay */
} STraceTrigger;
```

The listed trace and trigger settings are not available for all devices (see Functional Manual for the respective device).

Example: Logger MC V2.x RS

```
STraceTrigger traceTrigger;          //not used
STraceSetup traceSetup;
traceSetup.run = true;
traceSetup.source[0] = 200;          //Trace parameter 1: actual position
traceSetup.source[1] = 255;          //Trace parameter 2: not used
traceSetup.sampletime = 0;
eMomanprot ret = mmProtSetupTrace(traceSetup, traceTrigger);
```

Example: Logger MC V3.x RS/USB

```
STraceTrigger traceTrigger;          //not used
STraceSetup traceSetup;
traceSetup.run = true;
traceSetup.nodeNr = 1;
traceSetup.chan = 0;                 //Default channel
traceSetup.mode = 0;                 //Logger
traceSetup.source[0] = 0x606400;     //Trace parameter 1: actual position
traceSetup.source[1] = 0x606C00;     //Trace parameter 2: actual speed
traceSetup.source[2] = 0x607800;     //Trace parameter 3: actual current
                                     consumption
traceSetup.source[3] = 0;            //Trace parameter 4: not used
traceSetup.sourceLen[0] = 4;
traceSetup.sourceLen[1] = 4;
traceSetup.sourceLen[2] = 2;
traceSetup.sourceLen[3] = 0;
traceSetup.sourceType[0] = 0;
traceSetup.sourceType[1] = 0;
traceSetup.sourceType[2] = 0;
traceSetup.sourceType[3] = -1;
eMomanprot ret = mmProtSetupTrace(traceSetup, traceTrigger);
```

Example: Logger MC V3.x CAN

```
STraceTrigger traceTrigger;          //not used
STraceSetup traceSetup;
traceSetup.run = true;
traceSetup.nodeNr = 1;
traceSetup.chan = 1;                 //Logger uses CAN-PDOs
traceSetup.mode = 0;                 //Logger
traceSetup.source[0] = 0x606400;     //Trace parameter 1: actual position
traceSetup.source[1] = 0x606C00;     //Trace parameter 2: actual speed
traceSetup.source[2] = 0;            //Trace parameter 3: not used
traceSetup.source[3] = 0;            //Trace parameter 4: not used
traceSetup.sourceLen[0] = 4;
traceSetup.sourceLen[1] = 4;
traceSetup.sourceLen[2] = 0;
traceSetup.sourceLen[3] = 0;
traceSetup.sourceType[0] = 0x481;    //CobId of trace pdo for Source0
traceSetup.sourceType[1] = 0x481;    //CobId of trace pdo for Source1
traceSetup.sourceType[2] = -1;       //Deactivated
traceSetup.sourceType[3] = -1;       //Deactivated

eMomanprot ret = mmProtSetupTrace(traceSetup, traceTrigger);
```

Examples: Recorder MC V3.x

```
STraceTrigger traceTrigger;
traceTrigger.mode = 0;                  //No trigger
STraceSetup traceSetup;
traceSetup.run = true;
traceSetup.nodeNr = 1;
traceSetup.chan = 0;                    //Default channel
traceSetup.mode = 1;                    //Recorder
traceSetup.source[0] = 0x606400;        //Trace parameter 1: actual position
traceSetup.source[1] = 0x606C00;        //Trace parameter 2: actual speed
traceSetup.source[2] = 0;               //Trace parameter 3: not used
traceSetup.source[3] = 0;               //Trace parameter 4: not used
traceSetup.sourceLen[0] = 4;
traceSetup.sourceLen[1] = 4;
traceSetup.sourceLen[2] = 0;
traceSetup.sourceLen[3] = 0;
traceSetup.sourceType[0] = 0;
traceSetup.sourceType[1] = 0;
traceSetup.sourceType[2] = -1;
traceSetup.sourceType[3] = -1;
traceSetup.buffer = 512;
traceSetup.sampletime = 1;

eMomanprot ret = mmProtSetupTrace(traceSetup, traceTrigger);
```

5.5.2 RequestTrace

`eTraceRequest` **`mmProtRequestTrace (`**`int` *`mode`*`)`

Send trace request.

Parameter

[in] *mode* Request mode
0: Standard (read recorded data immediately)
1: Only status check in recorder mode
2: Read recorded data in recorder mode

Return data

eTraceRequest_inactive	Trace not activated
eTraceRequest_sent	Trace request sent
eTraceRequest_stateWaitResponse	Recorder expecting answer to trace request
eTraceRequest_stateWaitForTrigger	Recorder waiting for fulfilled trigger condition
eTraceRequest_stateRecordingInProgress	Recorder recording in progress
eTraceRequest_stateRecordingFinished	Recorder recording finished
eTraceRequest_errorReadBuffer	Error when reading the recorder buffer

Logger

Executing **`RequestTrace(`***`0`***`)`** sends a single data request to the device. In response, this device immediately transfers the configured data at the current point in time. The data are then passed on to the specified TraceValuesReceived callback function.

The following values are returned with the TraceValuesReceived callback function:

- Node number of the addressed device
- Array with 4 values (unsigned int) of the configured sources at the reading time
- Time code as time difference to the last reading time in milliseconds

After processing the received data, **`RequestTrace(`***`0`***`)`** can be called again.

Recorder (MC V3.x only)

Executing **`RequestTrace(`***`0`***`)`** or **`RequestTrace(`***`1`***`)`** sends a recorder request to the device. As soon as the set trigger condition is met, the device records the configured data.

- With *`mode`* = `0`, the recorded data are read as soon as they become available. *eTraceRequest_stateRecordingFinished* is returned at the same time.
- With *`mode`* = `1`, only *eTraceRequest_stateRecordingFinished* is returned. Execute **`RequestTrace(`***`2`***`)`** to read the data.

`RequestTrace(`*`0`***`)`** or **`RequestTrace(`***`1`***`)`** must be called in regular intervals to obtain information about the current trace request status and to request the recorded data as soon as they become available.

After being read in, the data are passed on in succession to the specified TraceValuesReceived callback function.

The following values are returned with the TraceValuesReceived callback function:

- Node number of the addressed device
- Array with 4 values (unsigned int) of the configured sources at the reading time
- Timecode with value of the set sample time

The number of calls of the callback function corresponds to the set size of the trace buffer.

After transferring the trace buffer, recording in the device is immediately reactivated (trigger enable). The availability of new data can be checked by making further calls of **`RequestTrace(`**`0`**`)`** or **`RequestTrace(`**`1`**`)`**.

5.6 Connection settings

The functions for the connection setting can be used to read and change the communication settings of connected devices. In the case of an unknown communication setting, a connection to connected devices can be searched for.

5.6.1 NetworkService

```
eMomanprot mmProtNetworkService ( int   mode,
                                  int   vendorId,
                                  int   productCode,
                                  int   revisionNumber,
                                  int   serialNumber
                                )
```

Identify and address network nodes, even if their node number is not known or not yet set.

With CANopen, the LSS protocol according to CiA 305 is used for this purpose.

Parameter

[in]	*mode*	Mode of the network service: 0: LSS switch normal mode 1: LSS switch config mode global 2: LSS switch config mode selective 3: LSS identify remote slave 4: select nodeNr (in param serialNumber)
[in]	*vendorId*	CAN-ID of the manufacturer (0x1018.01)
[in]	*productCode*	Identification number of the product (0x1018.02)
[out]	*revisionNumber*	Version number of the product (0x1018.03)
[out]	*serialNumber*	Serial number (0x1018.04) / node number

Return data

>0	Number of found nodes
eMomanprot_ok	Function successfully executed
eMomanprot_error	Error when executing the function

For the *vendorId*, *productCode* and *revisionNumber* values, see the Functional or Communications Manual of the respective device. The *serialNumber* is generally printed on the product.

5.6.2 GetCommunicationSettings

```
bool mmProtGetCommunicationSettings (int& nodeNr,
                                     int& baudrate
                                    )
```

Read communication settings of the node addressed via **NetworkService**.

Parameter

[out]	*nodeNr*	Currently set node number
[out]	*baudrate*	Currently set baud rate

Return data

true	Communications settings successfully read out
false	Error when reading out the communication settings

5.6.3 ChangeNodeNr

```
bool mmProtChangeNodeNr (int nodeNr)
```

Change the node number of node addressed via **NetworkService**.

Parameter

[in]	*nodeNr*	New node number

Return data

true	Node number successfully changed
false	Not possible to change the node number

5.6.4 ChangeBaudrate

```
bool mmProtChangeBaudrate ( int  baudrate,
                            int  activate
                          )
```

Change the baud rate of the node addressed via **NetworkService**.

Parameter

[in] *baudrate* New baud rate that is to be set in bit/s
0: Autobaud

[in] *activate* 0: Activate baud rate later (call with *activate* = 2)
1: Activate baud rate immediately
2: Activate previously set baud rate (parameter *baudrate* not used here)

Return data

true Baud rate successfully changed

false Not possible to change the baud rate

5.6.5 Connect

```
int mmProtConnect  ( int&  baudrate,
                     bool  tryBaudrates
                   )
```

Check or establish connection to a node.

If no connection has yet been established, a broadcast method is used to determine the node number of a connected device.

Parameter

[in, out] *baudrate* In: Current baud rate
Out: Determined baud rate in bit/s if *tryBaudrates* = true

[in] *tryBaudrates* True: Try baud rates if no connection is possible

Return data

≥0 First connected node number (255: unconfigured)

–1 No connection found

5.6.6 FindConnection

```
int mmProtFindConnection (int   scanMin,
                          int   scanMax,
                          int&  baudrate
                         )
```

Find connection to an RS232 node.

If no RS232 connection could be established with **Connect** via broadcast, it may be a RS232 network with which various node numbers can be tried with different baud rates.

Parameter

[in]	*scanMin*	Smallest node number
[in]	*scanMax*	Higher node number
[out]	*baudrate*	Determined baud rate in bit/s

Return data

≥0	First connected node number (255: unconfigured)
–1	No connection found

5.6.7 UnconfiguredSlavesCount

```
int mmProtUnconfiguredSlavesCount (void)
```

Read number of unconfigured nodes in the network.

With CANopen, the LSS protocol according to CiA 305 is used for this purpose. With USB and RS232, 0 is always returned.

Return data
Number of unconfigured nodes.

5.6.8 SupportedBaudratesList

```
const unsigned int* mmProtSupportedBaudratesList (int& count)
```

Read list of baud rates supported by this protocol.

Parameter

[out] *count* Number of supported baud rates

Return data

Pointer to the constant baud rate list.

Example

```
int count = 0;
const unsigned int* baudList = NULL;
baudList = mmProtSupportedBaudratesList (count);
if (baudList != Null) {
      unsigned int baudrate;
      for (int i = 0; i < count; i++) baudrate = baudList[i];
}
```

6 Ports and channels

All interface DLLs provide two general functions for accessing the interface information:

- **`mmIntfEnumPorts()`**

  Returns a list of available ports and channels, including the port information of the loaded interface.

- **`mmIntfIsPortAvailable()`**

  Checks whether the specified port/channel is available or already in use.

Both functions are defined in the **Momanintf.h** file.

6.1 EnumPorts

```
int mmIntfEnumPorts  ( const char**  portList,
                       const char**  chanList,
                       const char**  deviceInfoList
                     )
```

Read out available ports and channels of the loaded interface.

A sequential port ID is assigned for each interface (USB device, CAN card) registered via an interface DLL. When using multiple USB devices or CAN cards of this interface type, this port ID can be used to again access the device connected there. For COM, the port ID is always 0. For COM, the channel number designates the communication channel (e.g. 5 for COM5); for CAN cards with multiple bus connections, this is the number of the respective channel. For CAN cards with just one bus connection and with USB, the channel number is 0.

Port ID and channel number are required as parameters of protocol function **`OpenCom`**.

Parameter

[out] *portList* — List of the found port IDs (separated by commas).

If a port has multiple channels, the port ID for each channel is specified.

Example: "1,2,2" (port 1 with one channel, port 2 with two channels)

[out] *chanList* — List of the channel numbers for each port (separated by commas).

0: Only one channel on this port (no channel number)

>0: Channel number of this port.

Example: "0,1,2" (port 1 with one channel, port 2 with channel 1 and 2)

[out] *deviceInfoList* — List of the device information (interface name, serial number) for each port (separated by commas).

Return data

Number of found channels.

6.2 IsPortAvailable

```
eMomanIntf mmIntfIsPortAvailable ( int port,
                                   int chan
                                 )
```

Checks whether the specified port/channel is available or already in use.

Parameter

[in] *port* Port ID

[in] *chan* Channel number

Return data

eMomanintf_not_avail Port or channel not present

eMomanintf_avail Port present and available

eMomanintf_avail_inuse Port present but already in use

7 Alternative programming possibilities

In addition to the possibility to program the FAULHABER Motion Controllers via the MomanLib API described here, standard methods of the respective programming language can be used for communication. When using standard methods, there are no restrictions with respect to functionality and performance. In addition, the dependence on other external components no longer applies.

In the terminal window, the Motion Manager displays the communication history with the content of each sent and received telegram. For each command, this can be used to read the sent telegram and transfer it to a separate program.

7.1 RS232

Support for the RS232 interface is provided standard in most programming languages. The communication protocol to be used is described in the Communications Manual of the respective controller.

- Motion Controllers of the family MC V2.x are addressed via simple ASCII commands with subsequent carriage return. The commands can be sent directly character for character to the RS232 interface. Answers are read in the same way.
- Motion Controllers of the family MC V3.x are addressed via binary telegrams. Note that, besides the data that are to be transferred, the telegram length and a checksum that is to be calculated are included in the telegram. The algorithm for calculating the checksum is described in the communications manual of the respective controller.

7.1.1 C/C++

In C/C++, functions from the Windows API (`CreateFile()`, `WriteFile()`, `ReadFile()`) can be used. Alternatively, a number of commercial and free libraries are available for serial communication.

7.1.2 Delphi

In Delphi, functions from the Windows API (`CreateFile()`, `WriteFile()`, `ReadFile()`) can be used. Alternatively, a number of commercial and free libraries are available for serial communication.

7.1.3 C#

In C#, a `System.IO.Ports.SerialPort` can be created that can then be used either via the `System.IO.Stream` interface or the `Read` and `Write` class functions. By using the `DataReceived` event, constant polling does not need to be used; this event is recommended in cases of strict performance requirements.

7.1.4 LabVIEW

On the VISA palette, LabVIEW makes "serial" VIs available for communication via the serial interface.

7.2 USB

When using a USB-to-serial adapter, all controllers with RS232 interface can be programmed as described in chap. 7.1, p. 55.

7.2.1 Virtual COM port under Windows 10

Under Windows 10, a FAULHABER Motion Controller of the family MC V3.x can also be addressed via a virtual COM port as an alternative to the direct USB connection. To do this, the driver connection must be changed via the Windows Device Manager:

1. Update driver software.
2. Search for driver software on the computer.
3. Select from a list of device drivers on the computer.
4. Change from "Faulhaber MC3 WinUSB" to "Serial USB device".

Displayed in the Device Manager is which COM port number was assigned. It can be changed if necessary.

After changing the driver, the Motion Controller can be operated via USB as an RS232 device with the services described in the Communications Manual.

After changing the driver, the Motion Controller can – also with the Motion Manager – only be addressed via the virtual COM port instead of via USB.

7.2.2 Using the Moman USB-DLL

The MC3Usb.dll Motion Manager interface DLL can also be integrated in an application. The interface functions are defined in header file **Momanusb.h**:

```
void mmUsbInitLib(tdmmUsbDataCallback DataReceived);

void mmUsbDeinitLib(void);

int mmUsbOpen(int port, int channel);

void mmUsbClose(void);

void mmUsbSendData(char* data, int len);

int mmUsbReadData(char* data);

int mmUsbGetBufCount(int direction);
```

The callback function passed to **mmUsbInitLib()** is called on incoming messages. The received data must then be read via **mmUsbReadData()** and interpreted according to the documentation in the Communications Manual.

USB telegrams are sent with the **mmUsbSendData()** function.

7.3 CAN

The manufacturers of CAN interface cards (e.g., HMS-IXXAT, Peak) offer their own API libraries for use with various programming languages for their drivers.

8 Function overview

MomanLib makes the following API functions available:

mmIntfEnumPorts

mmIntfIsPortAvailable

mmProtChangeBaudrate

mmProtChangeNodeNr

mmProtCheckCommand

mmProtCheckMotionCommand

mmProtCloseCom

mmProtCloseInterface

mmProtConnect

mmProtDecodeAnswStr

mmProtDecodeCmdStr

mmProtFindConnection

mmProtGetAbortMessage

mmProtGetCommunicationHistory

mmProtGetCommunicationSettings

mmProtGetInt64Obj

mmProtGetObj

mmProtGetSendTelegram

mmProtGetStrObj

mmProtInitInterface

mmProtLoadCommandSet

mmProtNetworkService

mmProtOpenCom

mmProtReadAnswer

mmProtRequestTrace

mmProtScanNode

mmProtSendBuffer

mmProtSendCommand

mmProtSendMotionCommand

mmProtSendTelegram

mmProtSetDataCallback

mmProtSetObj

mmProtSetObjTimeout

mmProtSetStrObj

Function overview

mmProtSetTraceValuesCallback

mmProtSetupMessageFilter

mmProtSetupTrace

mmProtSupportedBaudratesList

mmProtUnconfiguredSlavesCount

mmProtWaitAnswer

9 FAULHABER licence agreement

End User Licence Agreement for Software of
Dr. Fritz Faulhaber GmbH & Co. KG

between

(1) **Dr. Fritz Faulhaber GmbH & Co. KG**, Faulhaberstraße 1, 71101 Schönaich
- hereinafter "FAULHABER" -

and

(2) You as the user
- hereinafter "Licensee" -

The parties (1) and (2) hereinafter referred to individually as "**Party**" and together as "**Parties**".

PRELIMINARY REMARKS

(A) Faulhaber designs drive systems and produces them. In addition, FAULHABER has developed various software products. For example, the "FAULHABER Motion Manager" (hereinafter "**Motion Manager**") supports the commissioning and configuration of FAULHABER drive systems. Details - if available - can be found in the respective software manual. Unless otherwise expressly agreed, the software product is made available to the Licensee free of charge as a supplement to other hardware and software products offered by FAULHABER.

(B) The Licensee intends to use one or more software product(s) in its business. FAULHABER is willing to grant the Licensee rights of use to the software product(s) of Dr. Fritz Faulhaber GmbH & Co. KG under the terms of this End User License Agreement (hereinafter "**Contract**"). The details of this result from **§ 2**.

This having been explained, the Parties agree the following:

§ 1
Subject of the Contract

(1) The subject of this Contract is the transfer of one or more of the software products listed in subsection (2) (hereinafter "**License Subject**") and the granting of the rights of use described in **§ 2** by FAULHABER to the Licensee.

(2) The provisions of this Agreement apply to the following categories of License Subjects, including related manuals, if available:

a) Motion Manager with associated user documentation;

b) b) Programming Libraries;

c) Firmware;

d) Sequential Programs.

(3) The following services in particular are not the subject of this Contract:

a) Installation or other setup of the License Subject at the Licensee;

b) any individual settings of variable parameters of the License Subject according to the requirements of the Licensee (customizing);

c) individual program extensions for the Licensee (individual modifications);

d) Adaptations of interfaces to the needs of the Licensee;

e) Instruction and training of the Licensee's program users;

f) Maintenance of the License Subject, in particular delivery of new, future versions.

§ 2
Content and Scope of the Rights of Use

(1) FAULHABER or its licensors are exclusively entitled to all rights to the License Subject. The Licensee is only entitled to the rights to the License Subject as agreed in this Contract.

(2) The Parties agree that the License Subject and the associated documents, including future versions, are protected by copyright and constitute confidential information and trade secrets of FAULHABER in accordance with **§ 6**.

(3) Unless otherwise provided in subsection (4), FAULHABER grants the Licensee the following rights of use to the License Subject:

a) FAULHABER grants the Licensee the non-exclusive right, unlimited in time and territory, to use the License Subject for its own purposes and for the purposes of Licensee's customer in accordance with the following subsections.

b) This right includes the installation of the License Subject as well as the loading, displaying and running of the installed License Subject as well as the saving of the License Subject in the memory of the hardware on which the License Subject is installed. In particular, the Licensee is not entitled to edit or otherwise modify the License Subject, unless this is expressly permitted in this Contract.

c) Duplications of the License Subject shall only be permitted to the extent necessary for the contractual use. The Licensee may make backup copies of the License Subject in accordance with the rules of technology to the extent necessary and in unchanged form, and in particular also as part of its normal backup of the system environment.

d) The Licensee shall be entitled to transfer the License Subject if (i) the Licensee transfers the License Subject together with original hardware components from FAULHABER, (ii) the transfer of the License Subject is free of charge for the third party, (iii) the Licensee ensures that the third party is not granted any further rights to the License Subject than the Licensee is entitled to under this Contract and (iv) the third party is imposed at least the obligations of this Contract with regard to the License Subject. Third parties shall also include companies which are affiliated with the Licensee within the meaning of § 15 of the German Stock Corporation Act (AktG).

e) FAULHABER is entitled to update the License Subject without prior notice, e.g. in order to correct errors or to improve or extend functions. If the updated version replaces the previous License Subject, it shall also be subject to the provisions of this Contract.

f) The Licensee may only use the License Subject within the scope of the intended use and only for productive operation if the License Subject is qualified for the specific application. "**Productive operation**" means the control of the respective drive system manufactured by FAULHABER by the License Subject in the ongoing operation of the application in the specific area of use of the Licensee, alone or in combination with other components of an overall system. Qualification for the specific application presupposes in particular that appropriate tests in the productive environment have been carried out successfully and that existing legal requirements and requirements for the specific application are met in full by the Licensee during use (e.g. international standards and norms). This applies in particular to use for medical and military purposes and in safety-critical areas (e.g. in the aerospace sector and for controlling nuclear facilities).

g) The Licensee shall have no claim against FAULHABER for release of the source code or the source code documentation. Notwithstanding the foregoing, the source code of the License Subject shall be part of the grant of use, provided that this is expressly stipulated in this Contract (in particular in subsection (4) below).

h) Insofar as the License Subject provided to the Licensee by FAULHABER contains open source software or software for which FAULHABER only has a derived right of use (hereinafter "**Third Party Software**"), the usage regulations to which this Third Party Software is subject shall apply additionally and with priority. The Third Party Software used within a License Subject, the license condition(s) applicable to the Third Party Software as well as any existing copyright notices are each named in the associated manual or are made available to the Licensee for downloading in a separate file with the delivery of the License Subject. The Licensee is obliged to comply with the respective license conditions. In the event of a breach of these license conditions by the Licensee, the licensors as well as FAULHABER shall be entitled to assert the resulting claims and rights in their own name.

(4) The rights of use pursuant to subsection (3) shall be supplemented or modified as follows for the License Subjects listed below:

a) Motion Manager

aa) The intended use of the Motion Manager results - if available - from the respective current version of the associated manual, which is made available on the FAULHABER website.

bb) The Motion Manager may only be used if the Licensee ensures that no injury or damage to health and no risk of material damage to property (e.g. equipment) is possible when it is used.

cc) The Licensee may not use the Motion Manager in productive operation. By way of clarification, the Parties state that such use does not constitute an intended use of the Motion Manager. The same applies to the use for controlling drive systems which were not manufactured by FAULHABER, as well as the use for controlling drive systems which were manufactured by FAULHABER but which are not listed in the program description. Deviating from this, Sequential Programs contained in the Motion Manager may be adapted for and used in productive operation, provided that they have been qualified in the application in accordance with subsection (3) f).

dd) Decompiling and other types of reverse engineering are generally not permitted. This does not apply to the Licensee's right to observe, examine or test the functioning of the Motion Manager in order to determine the ideas and principles underlying a program element if this is done by actions to load, display, run, transmit or store the program, which the Licensee is entitled to do in accordance with this Contract (§ 69d (3) of the German Copyright Act (UrhG)). In addition, notwithstanding sentence 1, the Licensee shall be entitled to decompile for the purpose of producing an interoperable program exclusively under the conditions of § 69e (1) and within the limits of § 69e (2) UrhG. The foregoing rights shall only exist if the Licensee has requested the information it requires from FAULHABER prior to any such action and has not received the required information within a reasonable period of time. As part of its request, the Licensee shall provide FAULHABER with all information necessary to evaluate the request.

ee) Any further use of the Motion Manager, in particular the granting of sublicenses, requires the prior express written consent of FAULHABER. This shall not apply in the event of a sale of the drive systems, insofar as their proper use requires the use of the Motion Manager.

ff) The Motion Manager may only be used in connection with original hardware components from FAULHABER. Use for third-party hardware is prohibited.

b) Programming Libraries

aa) The Licensee is granted the right to edit source code files of the Programming Libraries and to transfer these edits to third parties. However, the editing of object code files of the Programming Libraries is prohibited.

bb) The use of the Programming Libraries is only permitted in connection with original hardware components from FAULHABER. Use for third-party hardware is prohibited.

cc) FAULHABER shall make manuals for the Programming Libraries available to the Licensee for downloading on FAULHABER's website as required and at its own discretion. The Licensee shall have no claim to the provision of a manual. Insofar as an intended use of the Programming Libraries is specified, this shall result from the respective current version of the associated manual - if available.

dd) The Programming Libraries may only be used if the Licensee ensures that no injury or damage to health and no risk of material damage to property (e.g. equipment) is possible when they are used.

c) Firmware

aa) The intended use of the Firmware results from the respective current version of the associated manual, which is made available on FAULHABER's website.

bb) The Firmware may only be used if the Licensee ensures that no injury or damage to health and no risk of material damage to property (e.g. equipment) is possible when it is used.

cc) The use of the Firmware is only permitted in connection with original hardware components from FAULHABER. The right to use the Firmware for hardware of a third party shall only exist after prior written consent by FAULHABER.

dd) Decompiling and other types of reverse engineering are generally not permitted. This does not apply to the Licensee's right to observe, examine or test the functioning of the Firmware in order to determine the ideas and principles underlying a program element if this is done by actions to load, display, run, transmit or store the program, which the Licensee is entitled to do in accordance with this Contract (§ 69d (3) of the German Copyright Act (UrhG)). In addition, notwithstanding sentence 1, the Licensee shall be entitled to decompile for the purpose of producing an interoperable program exclusively under the conditions of § 69e (1) and within the limits of § 69e (2) UrhG. The foregoing rights shall only exist if the Licensee has requested the information it requires from FAULHABER prior to any such action and has not received the required information within a reasonable period of time. As part of its request, the Licensee shall provide FAULHABER with all information necessary to evaluate the request.

d) Sequential Programs

aa) Sequential Programs are programs that can be executed on specific FAULHABER controller hardware.

bb) The Licensee is granted the right to edit Sequential Programs and to transfer these edits to third parties, provided that they have been delivered in source code form.

cc) The use of Sequential Programs is only permitted in connection with original hardware components from FAULHABER. The right to use Sequential Programs for hardware of a third party shall only exist after prior written consent by FAULHABER.

§ 3
Delivery

(1) The License Subject shall be delivered in the form existing at the time of delivery ("as is").

(2) The delivery of the License Subject shall be made in digital form by providing it for download on FAULHABER's website or individually by e-mail. FAULHABER is not obliged to provide the License Subject on physical data carriers.

(3) Prior to provision, FAULHABER shall check the License Subject for any malware using a virus scanner that is up-to-date at the time of the respective provision. FAULHABER shall have no further obligations with regard to freedom from malware.

§ 4
Obligations of the Licensee

(1) The Licensee is obliged to ensure a sufficient technical operating and system environment and the proper operation of the License Subject. The establishment of the operating and system environment for the License Subject shall be the sole responsibility of the Licensee.

(2) If productive operation of the License Subject is permitted under this Agreement, the Licensee shall ensure that the requirements under A.I. § 2 (3) f) are fully met prior to productive operation of the License Subject. § 377 of the German Commercial Code (HGB) remains unaffected.

(3) The Licensee shall be solely responsible for the installation and implementation of the License Subject on Licensee's systems.

(4) The Licensee shall take all necessary and reasonable measures to prevent or limit damage caused by the License Subject. In particular, the Licensee shall use the current protective mechanisms to defend against malware.

(5) The Licensee is aware that, under certain circumstances, separate drivers from the adapter manufacturers may be required for any required communication interfaces (in particular for the Motion Manager and Programming Libraries), which are not provided by FAULHABER. Communication interfaces are interfaces for data exchange between PC and controller, e.g. via CAN, RS232, USB or EtherCAT. The Licensee is obliged to independently procure, license and install any drivers required to use these communication interfaces.

(6) The Licensee is prohibited from removing or modifying any copyright information from the License Subject.

§ 5
Liability of FAULHABER

(1) Insofar as the License Subject is provided free of charge, the following liability regulations shall apply:

a) FAULHABER shall only be liable for material and legal defects if FAULHABER fraudulently conceals such a defect.

b) FAULHABER shall be liable in accordance with the statutory provisions in the event of intent, gross negligence and claims under the Product Liability Act, fraudulent concealment of a defect, guarantee claims and in the event of injury to life, limb or health. Otherwise, FAULHABER's liability for claims for damages and reimbursement of expenses - irrespective of the reason - shall be excluded.

c) Insofar as a loss or destruction of data at the Licensee was caused by grossly negligent or intentional breach of contractual or statutory obligations, FAULHABER shall only be liable up to the amount of the typical recovery expense that arises despite regular, state-of-the-art data backup.

(2) In the event that Firmware is provided for installation on hardware components of FAULHABER, the liability regulations applicable to the respective hardware component shall apply in deviation from subsection (1).

§ 6
Confidentiality

(1) The Parties undertake to treat as confidential for an unlimited period of time all information of the other Party obtained in the course of the initiation and execution of the Contract which is marked as confidential or is confidential by its nature ("**Confidential Information**") for an unlimited period of time and to use it only for the purpose of executing this Contract. FAULHABER's Confidential Information shall also include the License Subject. Notwithstanding the foregoing, unless otherwise agreed, License Subjects made publicly available for download by FAULHABER on FAULHABER's website shall not be deemed Confidential Information.

(2) The Licensee shall only make the License Subject accessible to employees and other third parties insofar as this is necessary to exercise the granted rights of use. The Licensee shall inform all persons to whom it grants access to the License Subject of FAULHABER's existing rights thereto and of the confidentiality obligations and shall oblige these persons in writing to maintain confidentiality to the same extent as in this § 6, insofar as the persons concerned are not obliged to maintain secrecy at least to the aforementioned extent for other legal reasons.

(3) The obligations of confidentiality under the preceding subsections shall not apply to Confidential Information that (i) was already in the public domain or known to the other Party at the time it was transmitted by the Party; (ii) became apparent after being transmitted by the Party through no fault of the other Party; (iii) have been made available to the other Party by a third party after their transmission by the Party in a non-illegal manner and without restriction as to confidentiality or exploitation; and/or (iv) have been developed by one Party independently without using the Confidential Information or trade secrets of the other Party. Furthermore, the obligations shall not apply if the Confidential Information has to be published according to law, in particular due to an administrative order or a court decision; in this respect, the publishing Party shall inform the other Party thereof without undue delay and support it in defending such orders or decisions.

§ 7
Final Provisions

(1) Changes or additions to this Contract shall be made in writing. If they do not satisfy this requirement, they are invalid. The same also applies to changes to this written form clause.

(2) This Contract is subject to and shall be interpreted according to the laws of the Federal Republic of Germany. The application of the UN sales law (CISG United Nations Convention on Contracts for International Sale of Goods of April 11, 1980) is excluded.

(3) The sole place of jurisdiction is Stuttgart, if the Licensee is a businessperson in the meaning of the Commercial Code, a legal person under public law or a special fund under public law or on bringing the action the Licensee does not have any registered offices or usual place of residence (permanent address) in the Federal Republic of Germany.

(4) Should a provision of this Contract be or become invalid, all other provisions shall remain unaffected. Statutory law shall take the place of provisions that are not included or are invalid (§ 306 (2) of the German Civil Code (BGB)). Otherwise, the Parties shall replace the void or invalid provision with a valid provision that comes as close as possible to it in economic terms, unless a supplementary interpretation of the contract takes precedence or is possible.

DR. FRITZ FAULHABER GMBH & CO. KG
Antriebssysteme

Faulhaberstraße 1
71101 Schönaich • Germany
Tel. +49(0)7031/638-0
Fax +49(0)7031/638-100
info@faulhaber.de
www.faulhaber.com

7000.05064, 2nd edition, 8.10.2022