



Programming Guide: PLCHandler

Document Version 6.0

CONTENT

1	INTRODUCTION	6
2	CONTENT OF THE PLCHANDLER SDK	9
3	CPLCHANDLER CLASS	10
3.1	Constructors and Destructor of the PLCHandler	10
3.1.1	Constructors	10
3.1.2	Destructor	12
3.2	Configuration and use of the logger	12
3.2.1	SetLogging	12
3.2.2	GetLogging	13
3.2.3	SetLogFileCapacity	13
3.2.4	SetLogFile	13
3.2.5	GetLogFile	14
3.2.6	AddLogEntry	14
3.3	Get PLCHandler's version	14
3.3.1	GetVersion	14
3.4	Scan the PLC network	15
3.4.1	ScanNetwork	15
3.5	Late configuration of the PLCHandler after instantiation	15
3.5.1	SetConfig	15
3.5.2	SetConfigInteractive	16
3.5.3	GetConfig	16
3.5.4	SaveConfigInFile	17
3.5.5	SetTimeout	17
3.5.6	GetTimeout	17
3.5.7	GetName	17
3.5.8	GetId	17
3.6	Connection handling	17
3.6.1	Connect	18
3.6.2	Disconnect	19
3.6.3	GetState	19
3.6.4	GetLastError	20
3.6.5	SetStateChangedCallback	20
3.6.6	GetStateChangedCallback	20
3.6.7	GetReconnectCycles	20
3.6.8	StartKeepalive	20
3.6.9	StopKeepalive	20
3.7	Retrieve variable information from the PLC	21
3.7.1	LoadSymbolsFromPlc	21

3.7.2	EnterItemAccess	21
3.7.3	LeaveItemAccess	21
3.7.4	GetAllItems	21
3.7.5	GetItem	22
3.7.6	GetAddressOfMappedItem	23
3.8	Cyclic Update of Variables	24
3.8.1	CycDefineVarList	24
3.8.2	CycUpdateVarList	25
3.8.3	CycDeleteVarList	25
3.8.4	CyclValidList	25
3.8.5	CycEnterVarAccess	25
3.8.6	CycLeaveVarAccess	26
3.8.7	CycReadVars	26
3.8.8	CycReadChangedVars	27
3.8.9	CycGetOperatingRate	28
3.8.10	CycSetUpdateRate	28
3.8.11	CycGetUpdateRate	28
3.8.12	CycGetSymbolList	28
3.8.13	CycGetNumOfLists	29
3.8.14	CycGetVarListByIndex	29
3.8.15	CycGetVarListIndex	29
3.8.16	CycAddSymbolsToVarlist	29
3.8.17	CycRemoveSymbolsFromVarlist	30
3.9	Synchronous variable access	30
3.9.1	SyncReadVarsFromPlc	31
3.9.2	SyncReadVarsFromPlcReleaseValues	31
3.9.3	SyncWriteVarsToPlc	32
3.10	File and directory functions	33
3.10.1	UploadFile	33
3.10.2	DownloadFile	33
3.10.3	RenameFile	33
3.10.4	DeleteFile	34
3.10.5	ReadDirectory	34
3.10.6	CreateDirectory	35
3.10.7	RenameDirectory	35
3.10.8	DeleteDirectory	36
3.11	Retrieve project and application information from the PLC	36
3.11.1	GetApplicationList	37
3.11.2	GetProjectInfo	37
3.11.3	GetApplicationInfo	38

3.12	Get, set and reset the status of the application/PLC	38
3.12.1	GetPlcStatus	39
3.12.2	SetPlcStatus	39
3.12.3	ResetPlc	39
3.12.4	GetApplicationStatus	39
3.12.5	SetApplicationStatus	40
3.12.6	ResetApplication	40
3.13	Miscellaneous	41
3.13.1	ReloadBootproject	41
3.13.2	RegisterBootApplication	41
3.13.3	ReloadBootApplication	42
3.13.4	CheckTarget	42
3.13.5	GetDeviceInfo	42
3.13.6	EnterOnlineAccess	43
3.13.7	LeaveOnlineAccess	44
3.13.8	GetPlcComObject	44
3.13.9	Member variable ulCstData	44
3.14	Protected methods for sending any service to the PLC	44
3.14.1	SyncSendService	44
3.14.2	AsyncSendService	45
3.14.3	AsyncGetServiceReply	46
3.14.4	GetDeviceSessionId	46
4	SIMPLIFIED CONFIGURATION WITH THE CEASYPLCHANDLER CLASS	47
4.1	Connect...() methods	47
4.1.1	ConnectTcpipViaGateway	47
4.1.2	ConnectRs232ViaGateway	47
4.1.3	ConnectRs232ViaGatewayEx	48
4.1.4	ConnectTcpipViaArti	48
4.1.5	ConnectRs232ViaArti	48
4.1.6	ConnectToSimulation	49
4.1.7	ConnectViaGateway3	49
4.1.8	ConnectViaGateway3Ex	49
4.1.9	ConnectViaArti3	49
4.1.10	ConnectToSimulation3	49
5	CONFIGURATION PARAMETERS	50
5.1	Structure and parameters of the Ini-file	50
5.1.1	Parameters to describe the connection to the PLC	53
5.1.1.1	Parameters for a V2 PLC connection	53
5.1.1.2	Parameters for a V3 PLC connection	54
5.1.2	Special options for the PLCHandler	54

5.2	How to get a typical V2 PLC configuration	58
5.3	How to get a typical V3 PLC configuration	59
	APPENDIX A: ERROR CODES	61
	APPENDIX B: C-INTERFACE OF THE PLCHANDLER CLASS	63
	APPENDIX C: ACTIVEX-CONTROL OF THE PLCHANDLER (WINDOWS ONLY)	64
	CHANGE HISTORY	66

1 Introduction

The PLCHandler is a C++-class which provides at a comfortable level services for the communication between a client (e. g. visualization) and a 3S Automation-Alliance compliant PLC (controller). The PLCHandler encapsulates the complete low layer protocols and provides an API. This API provides functional access to all available features and services.

The following features and services are available:

- Establishing and terminating the communication with the PLC
- Reading the list of all variables on the PLC
- Cyclic reading of variables' values from the PLC
- Synchronous reading of variables' values from the PLC
- Synchronous writing of variables' values to the PLC
- Possibility of instancing for the purpose of a simultaneous communication with several PLCs
- Automatic reconnecting with the PLC after an break of the connection
- Automatic restart after a program download from CoDeSys to the PLC
- Callbacks to signal events (DataChange, StateChange, ...) to the client
- Getting/setting of the Application/PLC state
- Access to the PLC's file system
- Transfer of (custom specific) services to the PLC
- ...

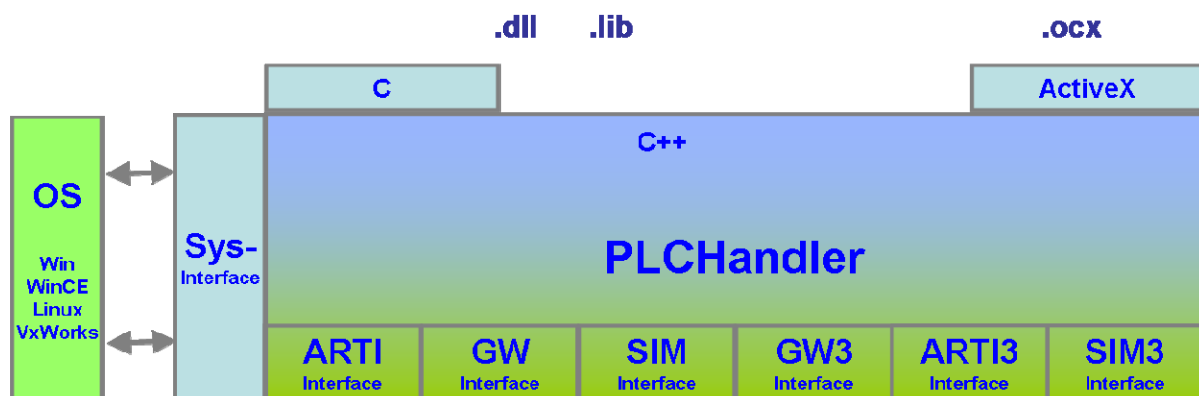
Thus the PLCHandler can be used as a basic component for visualizations and custom service tools. Also the CoDeSys OPC-Server V3 is based on the PLCHandler.

The PLCHandler supports CoDeSys V2.3 and CoDeSys V3 programmable PLCs. All specific differences are hidden as far as possible from the client.

The PLCHandler is delivered as SDK, i.e. all C++ header files, the PLCHandler libraries, example configuration files and demo source code are part of the package. This includes also a Visual Studio C++ V6.0 workspace for compiling the sample code. Furthermore this document and also the release information are contained.

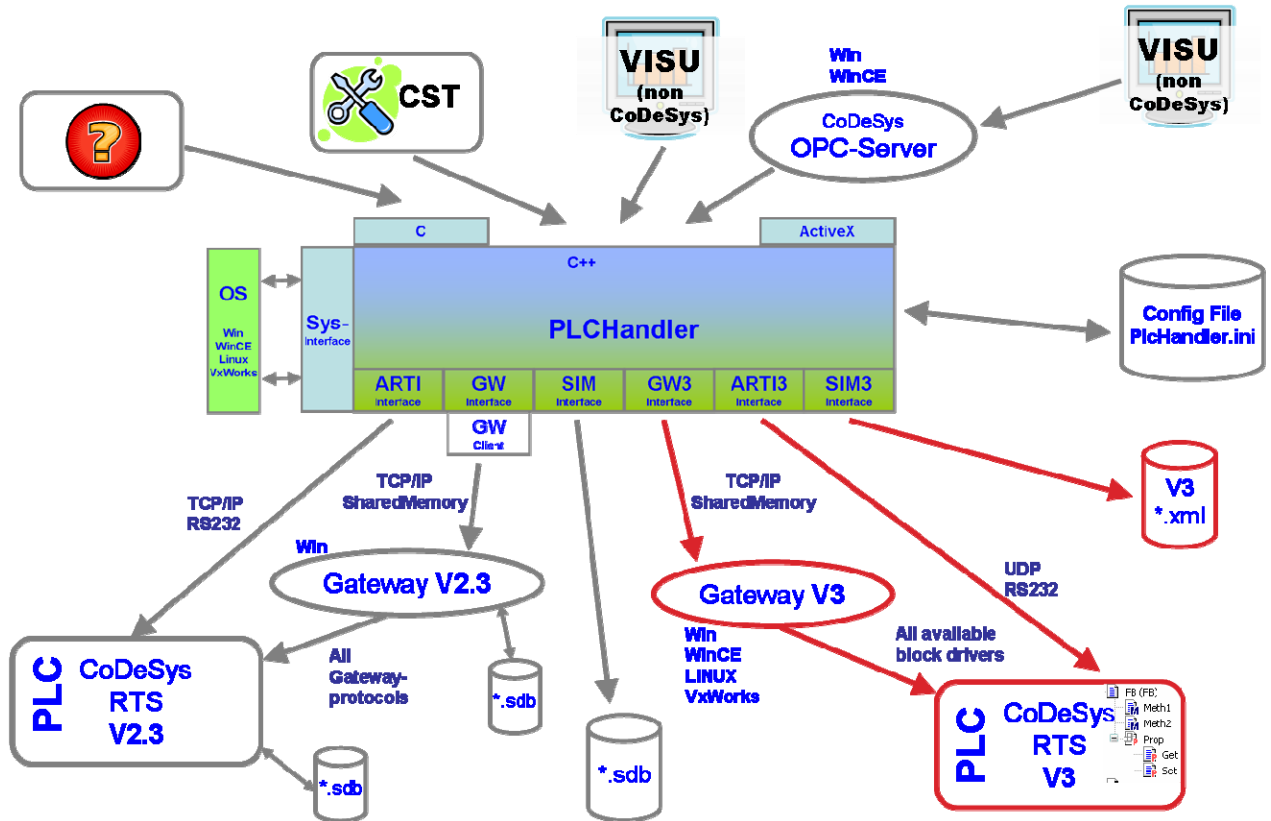
Typically the PLCHandler library is provided as static link library (e.g. Win32: PLCHandlerLink.lib or PLCHandlerLinkMFC.lib), which contains the PLCHandler's C++-class and additional an C-Interface. For the Windows Platform the PLCHandler additional is available as a dynamic link library (PLCHandlerDll.lib/dll) and as ActiveX-Control (PLCHandlerX.ocx).

The following figure illustrates the internal structure of the PLCHandler and the API-Interface.



To make the PLCHandler work on different operating systems, the PLCHandler uses the system components of the CoDeSys Runtime System V3. These components represent the hardware and operating system abstraction layer and hide processor and operating system specific stuff from all other components of the PLCHandler. Every system component has its specific functions, e. g. file access, access to heap memory, access to a serial RS232 interface, operating system tasks, etc.

The next picture shows an overview of the system architecture using the PLCHandler. Several different applications take advantage of the common PLCHandler API for all the different communication channels (named "interfaces" in the following).



The communication to the PLC can be done via the following interfaces:

1. **Gateway** (V2.3 only): Because of the fact, that the Gateway itself is restricted to Windows 95/98/NT/2000/XP/VISTA/7 (Win32), this interface can only be used, if also the PLCHandler resides on Win32. In this combination the PLCHandler can parameterize and utilize all drivers, which are known by the Gateway. So this interface offers drivers for various communication media and protocols (TCP/IP, RS232, Shared-Memory, CANopen, ...). Furthermore the Gateway keeps a copy of the SDB file¹ from the last project download to the PLC, which can be used by the PLCHandler, if the PLC has no capability to store the SDB file. The Gateway (V2.3) must be installed separately on this system. For this 3S provides a stand alone setup for the CoDeSys Gateway V2.3.
All in all this is the preferred interface to connect to CoDeSys V2.3 programmable PLCs, if the PLCHandler runs on a Win32 platform.
2. **ARTI** (V2.3 only): The ARTI-Interface is completely included into the PLCHandler and can therefore be used on all platforms for which the PLCHandler already has been ported to (Win32, WinCE, Linux, VxWorks) and serves as a communication layer concerning the PLC. The ARTI only

¹ SDB = Symbolic Data Base; will be created by CoDeSys V2.3 during project compilation if in Project / Options / Symbol configuration the export of variables is activated. This is a precondition for the symbolic variable access.

supports communication via TCPIP and serial interface. On non-Win32 platforms, this is the only way to communicate to a CoDeSys V2.3 programmable PLC.

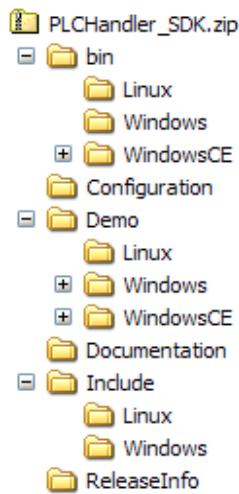
3. **Simulation** (V2.3 only): The symbolic information of the PLC is read directly from SDB file. Provided this way all variables can be written to and read from the PLC like it is possible in a connection. Overall this simulates the PLC connection. The typical use case is the design phase of a visualization, in which the set of exported variables is defined, but not the PLC (program) is available. Furthermore this interface can be applied to parse the SDB file and retrieve the information by calling GetAllItems(). Please find an example of a SDB file in the folder configuration of the SDK.
4. **Gateway3** (V3 only): One big improvement in the V3 communication is, that the CoDeSys Gateway V3 is also programmed in a platform independent way. Furthermore for an easy use the V3 gateway client is totally integrated in the PLCHandler. So to use this interface it's enough, if there is somewhere in the PLC network a Gateway is running, to which can the PLCHandler connect to. This can also be integrated on a PLC. The Gateway3 interface is nearly always used to establish a connection to a V3 PLC.
5. **ARTI3** (V3 only): The ARTI3 interface provides analogue to the ARTI interface a direct connection without having a Gateway between the PLCHandler and the PLC. For this the PLCHandler includes some V3 communications components (CmpChannelClient, CmpChannelMgr, CmpRouter, some block drivers, etc.) to talk directly to the runtime system. Depending on the system architecture, there is sometimes a special configuration for these components necessary. Therefore this interface should only be used in the very rare case, in which the Gateway3 interface can not be used. Such situations are:
 - local connection to a PLC, if there is no Gateway on the system or part of the PLC
 - no gateway in the PLC network available
6. **Simulation3** (V3 only): In V3 the symbolic variable information is part of the generated PLC code and hence there is for the online access no extra SDB file necessary. But for human reference and especially for the interface Simulation3 of the PLCHandler, a XML file is generated, which contains the symbolic information of the variables exported by symbol configuration of CoDeSys V3. Beside the data source, all other features of the interface Simulation3 match exactly to the features of the interface Simulation (see above). Please find an example of an XML file in the folder configuration of the SDK.

Important note:

Please take into account, that some of the described methods are not supported for all interfaces. This is usually noted in the description of the API function. Furthermore even if a feature is supported by a specific interface, it may not be supported by all CoDeSys programmable PLCs. For example if a PLC does not have a file system, then all file transfer methods will fail.

2 Content of the PLCHandler SDK

In the PLCHandler SDK you will find the following folders. The operating system specific subfolders are only available for the platforms you have ordered.



In the **bin** folder you will find the libraries for your platform. Typically there are several builds available with different linkage (static, dynamic) or for different processor architectures (ARM, PPC, X86) or for different operating system versions (WinCE 4.2, WinCE 5.0) or a combination of them.

It is strongly recommended to build your application only with PLCHandler header files, which are delivered with the same SDK version as the libs. This is also true for the dynamic linked libraries, at least if you are using an derived class of the C++ PLCHandler class. Furthermore you have to retest your application after each update of the PLCHandler to make sure, that it still behaves in the intended way.

Some example PLCHandler.ini files are located in the folder **Configuration**. These examples are good entry points for own configuration files.

The **Demo** folder contains example code “DemoPLCHandler.cpp”, “DemoEasyPLCHandler.cpp”, workspaces, projects or makefiles for common development environments and binaries for a quick test of the PLCHandler. Furthermore you will find several Visual Basic test clients as source code and executable for using the ActiveX interface of the PLCHandler (only Win32).

In the **Include** folder you find all C/C++ header files, which are necessary for using the libraries. The platform dependent header files are located in subfolders. Please note, that there is no difference between the Windows (Win32) und WinCE files, therefore there is no specific WinCE folder.

This document is also part of the Delivery and resides in the folder **Documentation**. Please find the release information in the folder **ReleaseInfo**. This includes the list of improvements, fixes and changes for the current version and hints regarding known restrictions.

3 CPLCHandler Class

3.1 Constructors and Destructor of the PLCHandler

A PLCHandler object of the CPLCHandler class always is assigned to a single PLC. Thus at creation of the object it can already be configured, with which PLC resp. which communication interface the PLCHandler should work. The configuration is specified in chapter 5.

3.1.1 Constructors

```
CPLCHandler(/*[In]*/ unsigned long ulId,
/*[In]*/ char *pszIniFile,
/*[In]*/ RTS_HANDLE hLogFile = RTS_INVALID_HANDLE)
```

ulId: The parameter *ulId* designates the index of the PLCHandler assumed that more instances are used. This corresponds to the PLC number in the server section of the ini-file.

pszIniFile: An ini-file for reading the configuration will be handed over to this constructor. The string can optionally contain a path. In the SDK there are several sample files as examples included. The structure of the ini-file is described in detail in the chapter 5.1 of this guide.

hLogFile: Handle to a logger, which was previously created with the runtime system API function *CAL_LogCreate()* or retrieved from another PLCHandler instance by *GetLogFile()*. This parameter is typically unused (value *RTS_INVALID_HANDLE*) for the first instance of the PLCHandler.

```
CPLCHandler(/*[In]*/ char *pszPlcName,
/*[In]*/ char *pszIniFile,
/*[In]*/ RTS_HANDLE hLogFile = RTS_INVALID_HANDLE)
```

pszPlcName: Name of the PLC in the ini-file. This can be used as an alternative to the PLC number.

All other parameters: see above.

```
CPLCHandler(/*[In]*/ PlcConfig *pPlcConfig,
/*[In]*/ PlcDeviceDesc *pDeviceDesc,
/*[In]*/ RTS_HANDLE hLogFile = RTS_INVALID_HANDLE)
```

This constructor allows configuring the PLCHandler instance without a configuration file. The configurable data are identical to the one, which can be set with the ini-file. So this is an alternative to the constructors above. The PLCHandler copies this structures internally, thus this structures can/must be deleted by the application after creating the PLCHandler instance.

pPlcConfig: Pointer to the general configuration structure of the PLCHandler object.

pDeviceDesc: Pointer to the device description structure, which provides the specific communication settings and additional options for the PLC connection of the PLCHandler object

hLogFile: see above.

The *PlcConfig* structure contains the configuration data for the PLCHandler. For a detailed description of the configuration parameters see also the description of the ini-file in the chapter 5.

```
struct PlcConfig
```

```
{
    PlcConfig(void);
    ~PlcConfig(void);
    unsigned long    ulId;                unique ID for every PLC
    char             *pszName;            name of the PLC
    ItfType          it;                  interface type (IT_ARTI, IT_GATEWAY, ...)
```

char	<i>bActive;</i>	flag if the PLC is active or not
char	<i>bMotorola;</i>	flag if data has to be swapped for Motorola byte order (only used for IT_ARTI, IT_GATEWAY)
char	<i>bLogin;</i>	flag if the application can do a login
char	<i>bLogToFile;</i>	flag if the application should log to a file
char	<i>bPreCheckIdentity;</i>	flag if the project identity will be checked before every read / write of variables (only used for IT_ARTI, IT_GATEWAY)
unsigned long	<i>ulTimeout;</i>	communication timeout in ms
unsigned long	<i>ulNumTries;</i>	number of tries before throwing COMM_FATAL
unsigned long	<i>ulWaitTime;</i>	max. time in ms to until Connect() returns
unsigned long	<i>ulReconnectTime;</i>	time interval in ms to try for a reconnection
char	<i>*pszHwType;</i>	HW type of the PLC (type: PLCC_HW_STANDARD, value for some ELAU targets: PLCC_HW_MAX4)
unsigned long	<i>ulHwVersion;</i>	HW version (typical value: 0)
unsigned long	<i>ulBufferSize;</i>	communication buffer size of the PLC; 0 = default size
char	<i>*pszProjectName;</i>	name of the CoDeSys project running on the PLC, typically NULL, only of interest for Simulation V2 or V3 and for Gateway V2, if the symbol file is not on the

PLC

char	<i>*pszDIIDirectory;</i>	typically NULL, fallback for very special environments
GatewayConnection	<i>*gwc;</i>	pointer to the gateway connection(only GW and GW3)
unsigned long	<i>ulLogFilter;</i>	filter for logging actions

};

For gateway connections (V2 or V3), the structure *GatewayConnection* describes the communication path between the PLCHandler and the CoDeSys Gateway:

```
struct GatewayConnection
```

{

	<i>GatewayConnection(void);</i>	
	<i>~GatewayConnection(void);</i>	
char*	<i>pszDeviceName;</i>	name of the gateway protocol ("Tcp/Ip", "Local", ...)
char*	<i>pszAddress;</i>	TCP/IP address of the gateway
unsigned long	<i>ulPort;</i>	TCP/IP port of the gateway
char*	<i>pszPassword;</i>	optional gateway password (Gateway V2 only)

};

The *PlcDeviceDesc* structure contains the data to connect to a PLC and all further optional parameters, which are not part of one of the configuration structures. For this the *PlcDeviceDesc* structure keeps the parameters in a generic format, which allows variable numbers and types of parameters :

```
struct PlcDeviceDesc
```

{

	<i>PlcDeviceDesc(void);</i>	
	<i>~PlcDeviceDesc(void);</i>	
char*	<i>pszName;</i>	name of the device / protocol (V2 only)
char*	<i>pszInstance;</i>	optional instance name of the gateway device (V2)
char*	<i>pszProject;</i>	obsolete, should always be NULL.
unsigned long	<i>ulNumParams;</i>	number of the following parameters
PlcParameterDesc*	<i>ppd;</i>	pointer to an array of parameter descriptions

};

Each parameter is described by the structure *PlcParameterDesc*:

```
struct PlcParameterDesc
```

{

	<i>PlcParameterDesc(void);</i>	
	<i>~PlcParameterDesc(void);</i>	
unsigned long	<i>ulId;</i>	id for the parameter, currently not evaluated
char*	<i>pszName;</i>	name of the parameter, identifies the parameter

```

    PlcParameter*    pParameter;    type and value of the parameter
};

```

To get the definition of the *PlcParameter* structure and the list of all possible parameters, please have a look at the PLCHandler header *PlcConfig.h*.

```
CPLCHandler(/*[In]*/ RTS_HANDLE hLogFile = RTS_INVALID_HANDLE)
```

This constructor is used to create a PLCHandler object with default configuration. In each case before connection establishment the communication device must be configured (with *SetConfig()*) or directly by opening the configuration with *GetConfig()* and setting the appropriate values).

hLogFile: see above.

3.1.2 Destructor

```
~CPLCHandler()
```

As soon as the PLCHandler object has been deleted, the connection to the PLC gets terminated (if there was one), all resources are deallocated and all internal threads are terminated.

3.2 Configuration and use of the logger

The PLCHandler contains a logging functionality, which is an important feature to analyze communication problems. Therefore all PLCHandler based applications should activate the logger and allow the user to enable/disable it and set the log filters. That is the reason why the enable flag and the log filter are part of the PLC configuration structure and are also read from the PLCHandler's configuration file. Moreover this is not a global setting, but can be individual configured for each instance of the PLCHandler. Ideal is to configure the logger directly after instantiate the PLCHandler object:

```

CPLCHandler *pPlcHandler = new CPLCHandler(...);
if (pPlcHandler != NULL)
{
    pPlcHandler->SetLogging (TRUE, LOG_STD); // only needed, if not
                                           // already enabled in the
                                           // PLCHandler's constructor
    pPlcHandler->SetLogFileCapacity (1000000, 5);
    pPlcHandler->SetLogFile("PLCHandlerDemo");
}

```

The PLCHandler class provides the following methods to configure the logger:

3.2.1 SetLogging

```

int ::SetLogging(/*[In]*/ int bEnable,
/*[In]*/ unsigned long ulLogFilter = LOG_STD)

```

SetLogging() enables or disables the logging at all, or sets a new log filter. If the logger is disabled, a call of *SetLogFile()* will not create a log file to keep the file system clean. Hence the logger must be first enabled by instantiation of the PLCHandler object (configuration file or configuration structure) or by calling *SetLogging()*.

bEnable: TRUE / FALSE, enables or disables the logger.

ulLogFilter: The log filter of the PLCHandler is a bit mask to select more or less details for the log file. The typical values are:
 16#0000000F: log infos, warnings, errors and exceptions
 16#000000FF: above + low level communication and some debug information
 16#FFFFFFFF: log all messages

Return value: TRUE / FALSE: New state of the logger.

3.2.2 GetLogging

```
int ::GetLogging(void)
```

GetLogging() retrieves the state of the logger.

Return value: TRUE / FALSE: State of the logger.

3.2.3 SetLogFileCapacity

```
long ::SetLogFileCapacity(/*[In]*/ int iMaxFileSize,
/*[In]*/ int iMaxFiles)
```

SetLogFileCapacity() configures the number of files and the size of each file used for the logger, to adapt the logger to the limits given by the present file system.

iMaxFileSize: If the current log file exceeds this value, then the next log messages will be written into a new file. Default value: 1.000.000 Bytes.

iMaxFiles: If this value is 0, then the PLCHandler will rename the log file, which has reached this maximum file size to <logfilename>_YYYY-MM-DD_HH:MM:SS.<extension>. So the complete history of the logger is available and the files can easily be sorted by timestamp, because it is part of the filename. **Attention:** In this case the PLCHandler can fill up the file system with log files, because there is no limit. Therefore this setting should be used carefully.

If this value is different from 0, then the log files will be used in the manner of a ring buffer. In this case the old log files will be renamed to a <logfilename>_#.<extension>, where “#” is a counter in the range 0..(*iMaxFiles* - 1).
Default value: 5 files.

Return value: This function always returns RESULT_OK.

3.2.4 SetLogFile

```
long ::SetLogFile(/*[In]*/ char *pszLogFile)
```

This method sets the name of the log files. If the logging is enabled (see *SetLogging()*), then the PLCHandler will check if already a logger exists, which is assigned to the passed file name. Depending on this the logger will use the existing instance or create a new logger with this file name. So the application can decide which PLCHandler instances have to share a log file and which have to create own log files.

pszLogFile: Name of the log file. The string can include a path or an extension. If there is no extension specified, “.log” is added.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_DISABLED: Logging is disabled (see *SetLogging()*).

RESULT_FAILED: Logger was not able to open or create the logger object.

```
long ::SetLogFile(/*[In]*/ RTS_HANDLE hLogFile)
```

Method to set the log file by handle. This method is typically used to set the log file of the second and further PLCHandler instances to the log file of the first instance. The same can already be done by using the *hLogFile* parameter in the constructor of the PLCHandler.

hLogFile: Handle to a logger, which was previously created with the runtime system API function *CAL_LogCreate()* or retrieved from another PLCHandler instance by *GetLogFile()*.

Return value: This function always returns RESULT_OK.

3.2.5 GetLogFile

```
RTS_HANDLE ::GetLogFile(void)
```

GetLogFile() retrieves the handle of the logger. Usually this is used to get the handle and pass it to another instance of the PLCHandler.

Return value: Handle of the logger.

3.2.6 AddLogEntry

```
long ::AddLogEntry(/*[In]*/ unsigned long CmpId,  
/*[In]*/ int iClassID, /*[In]*/ int iErrorID,  
/*[In]*/ char *pszInfo, ...)
```

AddLogEntry() allows a PLCHandler based application to add entries to the internal logger of a PLCHandler instance.

CmpId: Currently not evaluated. Should be set to 0.

iClassID: Error class of the entry. This is evaluated by log filter. A log message is only stored in the log file, if the corresponding bit is set in the log filter. The following error classes are defined:

```
#define LOG_NONE                0x00000000  
#define LOG_INFO                0x00000001  
#define LOG_WARNING             0x00000002  
#define LOG_ERROR               0x00000004  
#define LOG_EXCEPTION           0x00000008  
#define LOG_DEBUG               0x00000010  
#define LOG_PRINTF              0x00000020  
#define LOG_COM                 0x00000040  
#define LOG_INFO_TIMESTAMP_RELATIVE 0x00000080  
#define LOG_CRIT_SEC            0x00000100
```

iErrorID: Result (return value) of the operation, which leads to the log entry. Currently not stored in the log file.

pszInfo: String to add to the log file. In combination with the optional arguments all *printf()* format placeholders can be used. **Attention:** The resulting string must not exceed 255 characters.

Return value: If no error occurs, the function will return `RESULT_OK`. Otherwise, it returns ...

`RESULT_DISABLED`: Logging is disabled (see *SetLogging()*).

`RESULT_FAILED`: The entry could not be added to the logger.

3.3 Get PLCHandler's version

3.3.1 GetVersion

```
unsigned long ::GetVersion(/*[InOut]*/ char *pszVersion = NULL,  
/*[In]*/ long lLen = 0)
```

Retrieve the version of the PLCHandler as version number or as version String.

pszVersion: Pointer to a string, which is allocated by the application. If Null, no string will be returned. The PLCHandler copies the version string to this address.

lLen: Length of the memory area, to which *pszVersion* points to. If the version string is longer than *lLen*, the string will be truncated.

Return value: Version as unsigned long value. Example: 0x03040100 means V3.4.1.0.

3.4 Scan the PLC network

As from version 3 the CoDeSys runtime systems support network scan functionality. This is mainly used for the CoDeSys IDE to select the PLC to work with. The PLCHandler does also support this feature for the interfaces Gateway3 and ARTI3 to give an application the chance to provide also a network scan to the user. After the network scan the PLCHandler can be configured with the selected PLC connection and connect to this.

3.4.1 ScanNetwork

```
long ::ScanNetwork(/*[In]*/ GatewayConnection *pGatewayConnection,
/*[In]*/ CPLHandlerCallback *pPlcFoundCallback)
```

Only interfaces ARTI3 and Gateway3.

Method to scan the PLC network. This functionality is not connection based and can be used directly after instantiation of the PLCHandler or also while a connection exists. The network scan will be executed directly from the PLCHandler's base network layers, which are used for the interface ARTI3, to get all PLCs, which are directly reachable. Or a gateway can be specified, from which the scan request should be issued. The configuration of the network scan's entry point is independent from all other communication settings. It is recommended to setup the logging (log file and log filter) before, so that the scan has the possibility to add log messages, if needed. The function returns immediately after the internal start of the scan. The next scan can only be started, if the current one has finished and the last callback was called.

pGatewayConnection: Pointer to a filled structure with the gateway connection parameters. If NULL, then the local ARTI3 interface sends the scan request.

pStateChangedCallback: Pointer to an instance of a derivative class of *CPLHandlerCallback*. For each found PLC the *Notify* method of this class is called. The end of the network scan is signalled by an additional callback. After this last callback, the next scan can be started.

Attention: The callback may be called before this function returns.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_INVALID_PARAMETER`: *pStateChangedCallback* is NULL.

`RESULT_NOT_SUPPORTED`: The configured interface does not support the network scan.

`RESULT_COMPONENTS_NOT_LOADED`: The PLCHandler was not able to load all needed components during the instantiation.

`RESULT_BUSY`: The last scan is still running.

3.5 Late configuration of the PLCHandler after instantiation

If the configuration for the PLCHandler is not set by instantiation using the ini-file or configuration structure, then this can be done afterwards with the following methods. Furthermore is it possible to change parameters. The functions, which change the configuration must be only called before *Connect()* or after *Disconnect()* to be sure, that the PLCHandler does not access the parameters at the same time.

3.5.1 SetConfig

```
long ::SetConfig(/*[In]*/ char *pszIniFile)
```

Method to set the configuration of the PLCHandler out of a configuration file. Uses the PLC number (PLC Id) set in the current PLCHandler instance. The default value for the PLC Id is 0.

pszIniFile: Name of the ini-file; can optionally contain a path.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED` if the ini-file or the PLC section for the given Id does not exist.

```
long ::SetConfig(/*[In]*/ PlcConfig *pPlcConfig,  
/*[In]*/ PlcDeviceDesc *pDeviceDescFile)
```

Method to set the configuration of the PLCHandler by configuration structures. The PLCHandler copies this structures internally, thus the application is responsible to free the structures after calling this function. Please find the detailed description of the structures in the chapter 3.1.

pPlcConfig: Pointer to the general configuration structure of the PLCHandler object.

pDeviceDesc: Pointer to the device description structure, which provides the specific communication settings and additional options for the PLC connection of the PLCHandler object.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NO_CONFIGURATION` if the passed configuration is invalid.

`RESULT_INVALID_PARAMETER`: Invalid function parameters (e. g. `NULL`).

```
long ::SetConfig /*[In]*/ char *pszConfig,  
/*[In]*/ unsigned long ulConfigLen, /*[In]*/ char *pszLineEnd)
```

Method to set the configuration of the PLCHandler by a string, which was typically previous read from a file. This is useful, if the PLCHandler is part of an application, that provides own mechanisms to load and store the configuration. So the PLCHandler configuration can be handled as binary blob. The given configuration is stored temporarily as file, so does is internally handled by *SetConfig(char *pszIniFile)*.

pszConfig: String with the configuration data. Is only accessed during the function call.

ulConfigLen: Length of the string.

pszLineEnd: Currently not evaluated, must be `NULL`;

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED` if the ini-file or the PLC section for the given Id does not exist.

3.5.2 SetConfigInteractive

```
long ::SetConfigInteractive(void)
```

Only interface Gateway (V2.3).

SetConfigInteractive() is used to configure the communication settings interactive by the user. For this the PLCHandler displays the communication settings dialog of the CoDeSys Gateway V2.3. After the user closes the dialog, the settings are taken over by the PLCHandler and the function returns.

Please note, that for all full configuration you have to set first most of the configuration parameters (used interface, timeouts, logging, etc.) by another *SetConfig()* method or directly within the constructor, because *SetConfigInteractive()* modifies only the gateway and the PLC connection parameters.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NO_CONFIGURATION` if the interface type is invalid.

`RESULT_FAILED` in all other cases.

3.5.3 GetConfig

```
long ::GetConfig /*[Out]*/ PlcConfig **ppPlcConfig,  
/*[Out]*/ PlcDeviceDesc **ppDeviceDesc)
```

Method to get the complete configuration of the PLCHandler.

Attention: This method can be used in two ways:

To get a pointer to the internal *PlcConfig* or *PlcDeviceDesc* object, the function has to be called with **ppPlcConfig == NULL* and **ppDeviceDesc == NULL*. This allows modifying the internal structure of the PLCHandler instance directly. Be careful to change the parameters only before calling *Connect()*

or after calling *Disconnect()* to be sure, that the PLCHandler does not access the parameters at the same time.

Secondly you get a copy of the configuration structures, if **ppPlcConfig != NULL* and **ppDeviceDesc != NULL*. In this case the application is responsible for allocating the memory for the structures before calling *GetConfig()* and to free it after use.

ppPlcConfig: Pointer to the address of the *PLCConfig* structure.

ppDeviceDesc: Pointer to the address of the *PlcDeviceDesc* structure.

Return value: This function always returns `RESULT_OK`.

3.5.4 SaveConfigInFile

```
long ::SaveConfigInFile(/*[In]*/ char *pszIniFile)
```

Method is not implemented yet.

Return value: This function always returns `RESULT_FAILED`.

3.5.5 SetTimeout

```
long ::SetTimeout(/*[In]*/ unsigned long ulTimeout)
```

Method to change the communication timeout (*PlcConfig.ulTimeout*). Typically used for the ARTI or Gateway interface (V2.3) to extend the timeout for a specific online service.

ulTimeout: Communication timeout in ms.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED` if the timeout could not be set.

3.5.6 GetTimeout

```
unsigned long ::GetTimeout(void)
```

Method to read the communication timeout (*PlcConfig.ulTimeout*).

Return value: Communication timeout in ms.

3.5.7 GetName

```
char* ::GetName(void)
```

Method to get the name of the PLCHandler instance (*PlcConfig.pszName*).

Return value: Pointer to the internal PLCHandler configuration object. **Attention:** Name must not be deleted by the application.

3.5.8 GetId

```
char* ::GetId(void)
```

Method to get the Id of the PLCHandler instance (*PlcConfig.ulId*).

Return value: Id of the PLCHandler instance. This corresponds to the PLC number in the server section of the ini-file.

3.6 Connection handling

The PLCHandler works connection based, that means it is only possible to send a service to the PLC, if there is an active connection. Internally this is managed by a state machine, which can be triggered from outside by calling *Connect()* or *Disconnect()*. This state machine is realized using an own background thread named *ReconnectThread*, which is created and started by the call of *Connect()* or

if the PLCHandler leaves the state `STATE_RUNNING` because of a communication error, new project download to the PLC or an online change. The *ReconnectThread* is stopped, if the PLCHandler enters the state `STATE_RUNNING` after the connection is established successfully (again) or at the call of *Disconnect()*. At entering the state `STATE_RUNNING`, the PLCHandler starts a thread, which reads every 4 seconds the status of the PLC. This is done to prevent the PLC to close the channel again, because of inactivity of the client.

The next table shows all states of the state machine:

State	Value	Description
<code>STATE_TERMINATE</code>	-1	PLCHandler currently is terminating (Destructor)
<code>STATE_PLC_NOT_CONNECTED</code>	0	PLC is not connected (init state)
<code>STATE_PLC_CONNECTED</code>	1	PLC is connected
<code>STATE_NO_SYMBOLS</code>	2	No symbols are available or symbol mismatch because of changed project.
<code>STATE_SYMBOLS_LOADED</code>	3	PLC is connected and symbols are loaded
<code>STATE_RUNNING</code>	4	PLC is connected, symbols are loaded and all of them are verified. This state is also reached without available symbols, if <i>Connect()</i> was called with <i>bLoadSymbols = 0</i>. Now you can work correctly with the PLCHandler object.
<code>STATE_DISCONNECT</code>	5	Connection is just getting terminated
<code>STATE_NO_CONFIGURATION</code>	6	Configuration of the PLCHandler is invalid
<code>STATE_PLC_NOT_CONNECTED_SYMBOLS_LOADED</code>	7	PLC is currently not available, but the symbol information could be loaded offline. The PLCHandler tries further to connect to the PLC.

Besides analyzing a connection problem, for most applications it is enough to react on the state change to `STATE_RUNNING` and the leaving of this state. In the first case the application can start to work with the PLC, e. g. (re-)define and read cyclic variable lists. In the second case the PLCHandler lost the connection to the PLC and deletes internally all connection based data, including the cyclic variable lists. Thus also the application must handle this new state.

3.6.1 Connect

```
long ::Connect(/*[In]*/ unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT,
/*[In]*/ CPLCHandlerCallback *pStateChangedCallback = NULL,
/*[In]*/ int bLoadSymbols = 1)
```

After the configuration has been done, via this method a connection to the PLC can be established. The method will return as soon as the connection has been built up successfully or the timeout condition is fulfilled. In case of an error the current state of the PLCHandler resp. the last occurred error can be get by *GetStatus()* and *GetLastError()*.

If a connection could not be established, the method will return an error and internally a thread will be started, trying to reconnect to the PLC in the defined reconnect interval.

ulTimeout: This value specifies how long the connection establishment maximally can take, before the function returns.
If this value is set to `PLCHANDLER_USE_DEFAULT (=0)`, the value set actually in the configuration will be used (see: *WaitTime*).

If *ulTimeout* is set to `PLCHANDLER_TIMEOUT_INFINITE (-1)` the method will do one connect try and return independently of the result. In this case the PLCHandler does not start the internal reconnect thread for the initial connect, even, if the connect was not successful.

To disable the reconnect thread not only for the initial connect, but also for all further connection loss, the reconnect time in the configuration have to be set to `PLCHANDLER_TIMEOUT_INFINITE (-1)`.

pStateChangedCallback: Pointer to an instance of a derivative class of *CPLCHandlerCallback*. On each transition to a new PLCHandler state the *Notify* method of this class is called. This is the preferred way for applications to get always the latest state changes.

bLoadSymbols: This parameter can be used to specify whether at a connection establishment the symbols of the PLC should be loaded or not. If this parameter is set to `FALSE`, the connection establishment will also succeed, if there are no symbols available. In this case only services of the PLCHandler can be used, which do not access variables or the symbolic variable information. This configuration is typically set, if the PLCHandler is used for a service application, which supports for example file transfer and start/stop features.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED`: common error in the underlying interfaces. A specific error code can be retrieved by *GetStatus()* resp. *GetLastError()*.

`RESULT_COMPONENTS_NOT_LOADED`: The PLCHandler was not able to load all needed components during the instantiation.

`RESULT_NO_CONFIGURATION`: No configuration for this PLCHandler instance (*Id* unknown).

`RESULT_PLCHANDLER_INACTIVE`: PLCHandler instance isn't set active (see ini-file).

`RESULT_RECONNECTTHREAD_STILL_ACTIVE`: Reconnect thread is still active, *Connect()* was already called before.

`RESULT_ITF_NOT_SUPPORTED`: The configured interface isn't supported.

`RESULT_ITF_FAILED`: The interface can't start successfully (missing interface dependent DLL's).

`RESULT_PLC_NOT_CONNECTED`: Cannot open connection to the PLC.

`RESULT_PLC_NOT_CONNECTED_SYMBOLS_LOADED`: Cannot open connection to the PLC but could load the symbol file offline.

`RESULT_LOADING_SYMBOLS_FAILED`: No symbols are available or symbol mismatch because of changed project.

`RESULT_COMM_FATAL`: Communication error occurred during login or symbol load.

`RESULT_PLC_LOGIN_FAILED`: Login service is not supported by this PLC or has failed.

3.6.2 Disconnect

```
long ::Disconnect(void)
```

At a call of this method the connection to the PLC will be terminated and all connection based data will be released, including all cyclic variable lists.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_COMM_FATAL`: A still active reconnect thread could not be terminated gracefully. Please check the configured timeout values. If this happens, the PLCHandler instance have usually to be deleted, else the next connect may result in a deadlock.

`RESULT_FAILED`: no corresponding interface, which can be closed .

3.6.3 GetState

```
PLCHANDLER_STATE ::GetState(void)
```

This method can be called to get the current state of the PLCHandler's internal state machine. The same information is provided by the *StateChangedCallback* of the *Connect()* function.

Return value: Current state of the PLCHandler, see above for a list of all states.

3.6.4 GetLastError

```
long :: GetLastError(void)
```

GetLastError() returns the last error code, which is set during reconnect. See Appendix A: for a full list of all possible return values.

Return value: Last occurred error.

3.6.5 SetStateChangedCallback

```
long :: SetStateChangedCallback(CPLCHandlerCallback *pCallback)
```

Method to register a callback class for the *StateChangedCallback*, if this was not already done with *Connect()*. Furthermore this function can be used to disable the callback again.

pCallback: Pointer to an instance of a derivative class of *CPLCHandlerCallback*. On each transition to a new PLCHandler state the *Notify* method of this class is called. The callback can be deactivated by setting this parameter to NULL.

Return value: This function always returns RESULT_OK.

3.6.6 GetStateChangedCallback

```
CPLCHandlerCallback * :: GetStateChangedCallback(void)
```

This function returns a pointer to the currently registered instance of the *StateChangeCallback*.

Return value: Pointer to the registered instance of the *CPLCHandlerCallback* class or NULL, if no callback is registered.

3.6.7 GetReconnectCycles

```
long :: GetReconnectCycles(void)
```

This function returns the internal reconnect count, which counts the connect retries since the last start of the *ReconnectThread*.

Return value: Number of the reconnect tries.

3.6.8 StartKeepalive

```
long :: StartKeepalive(void)
```

This method can be used to start the internal KeepAliveThread manually. It is not recommended to call this function, because this thread is part of the connection management of the PLCHandler and is handled automatically.

Return value: This function always returns RESULT_OK.

3.6.9 StopKeepalive

```
long :: StopKeepalive(void)
```

This method can be used to stop the internal KeepAliveThread manually. It is not recommended to call this function, because this thread is part of the connection management of the PLCHandler and is handled automatically.

Return value: This function always returns RESULT_OK.

3.7 Retrieve variable information from the PLC

The PLCHandler provides a full list with all available variables in the PLC. This list contains a description for each variable and is the internal basis for all symbolic variable access by the application.

3.7.1 LoadSymbolsFromPlc

```
long :: LoadSymbolsFromPlc(void)
```

If the *Connect()* was called without loading the symbols (*bLoadSymbols=0*), the symbols can be loaded by *LoadSymbolsFromPlc()*. So there is no need to close and re-establish the connection. The favoured way is to call *Connect()* with *bLoadSymbols=1* instead of calling this method afterwards.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_COMM_FATAL`: Communication error occurred during symbol load.

`RESULT_LOADING_SYMBOLS_FAILED`: No symbols are available or symbol mismatch because of changed project.

`RESULT_FAILED`: Failed due to internal error.

3.7.2 EnterItemAccess

```
long :: EnterItemAccess(void)
```

Protects the access to the symbol list, so it can not be changed by another thread. (see *GetAllItems()* for details).

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_PLC_NOT_CONNECTED`: Internal Semaphore is not entered. PLCHandler is not in state `STATE_RUNNING` or `STATE_PLC_NOT_CONNECTED_SYMBOLS_LOADED`.

`RESULT_FAILED`: EnterItemAccess failed due to internal error.

3.7.3 LeaveItemAccess

```
void :: LeaveItemAccess(void)
```

Releases the access to the symbol list (see *GetAllItems()* for details). Must only be called, if *EnterItemAccess()* has returned `RESULT_OK`.

Return value: -

3.7.4 GetAllItems

```
long :: GetAllItems(/*[Out]*/ PlcSymbolDesc **ppSymbolList,  
/*[Out]*/ unsigned long *pulNumOfSymbols)
```

This method can be used to read the list of all variables which are specified in the PLC.

Attention: Before calling *GetAllItems()* the method *EnterItemAccess()* must be called first to prevent the internal symbol list to be deleted or modified by another thread. After accessing *ppSymbolList*, *LeaveItemAccess()* must be called. Make sure, that after this release of the symbol list *ppSymbolList* is not referenced anymore.

Intended use of this function:

```

PlcSymbolDesc *pSymbolsTemp = NULL;
unsigned long ulSymbols;

if pPlcHandler->EnterItemAccess() == RESULT_OK)
{
    if (pPlcHandler->GetAllItems(&pSymbolsTemp, &ulSymbols) != RESULT_OK)
    {
        // Error occurred
    }
    else
    {
        // Copy symbols from PLCHandler to application
    }
    pSymbolsTemp = NULL; // Do not access pSymbolsTemp anymore
    pPlcHandler->LeaveItemAccess();
}

```

ppSymbolList: Pointer to pointer to the *PlcSymbolDesc* structure. *ppSymbolist will be filled with a pointer to the symbol list of the PLCHandler, whereas each symbol is described by the *PlcSymbolDesc* structure. **Attention:** The object, to which the pointer points to, must not be deleted by the application.

pulNumOfSymbols: Pointer to a variable of type unsigned long, in which the number of variables will be returned.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_NOT_CONNECTED`: Symbol information cannot be retrieved, because PLCHandler is not in state `STATE_RUNNING` or `STATE_PLC_NOT_CONNECTED_SYMBOLS_LOADED`.

`RESULT_INVALID_PARAMETER`: Invalid function parameters (for example NULL).

`RESULT_DISABLED`: Symbol load was disabled by the setting `DontLoadSymbolsFromPlc=1` (only interfaces ARTI3 and Gateway3)

The *PlcSymbolDesc* structure contains the symbolic description of each variable:

```

struct PlcSymbolDesc
{
    char*          pszName;      symbol name of the variable
    unsigned long  ulTypeId;     reference into the symbol type table (PLCComBase.h)
    char*          pszType;      type name of the variable
    unsigned short usRefId;      variable's type (input, output, global, ...)
    unsigned long  ulOffset;     offset in the memory image
    unsigned long  ulSize;       size of the variable
    char           szAccess[2];  access type of the variable (no, read, write access, both)
    unsigned char  bySwapSize;   basic swap size of the variable
};

```

The size of a bit symbol is per definition 0 in the CoDeSys runtime system. So you have to check the symbol's size for 0 and do a special bit coding (e. g. *memset()* or *memcpy()* with parameter *size=1* instead with the original symbol's size)

The elements *usRefId* and *ulOffset* are only informative and must not be used as references, because the values are not online change safe. Therefore this information is no longer provided for all V3 interfaces.

3.7.5 GetItem

```

long ::GetItem(/*[In]*/ char *pszSymbol,
/*[Out]*/ PlcSymbolDesc *pSymbol)

```

GetItem() is used to retrieve the description of a single variable given by name.

Attention: Before calling *GetItem()* the method *EnterItemAccess()* must be called first to prevent the internal symbol list to be deleted or modified by another thread. After accessing *pSymbol*,

LeaveItemAccess() must be called. Make sure, that after this release of the symbol list *pSymbol* is not referenced anymore.

pszSymbol: Specifies the variable, for which the description should be read.

pSymbol: Pointer to a *PlcSymbolDesc* structure allocated by the application. The method copies the description of the variable from the internal symbol list to this structure.

Attention: The pointer to strings inside the structure (*pszName* and *pszType*) must not be deleted by the application, because these point to the internal symbol list.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_NOT_CONNECTED`: Symbol information cannot be retrieved, because PLCHandler is not in state `STATE_RUNNING` or `STATE_PLC_NOT_CONNECTED_SYMBOLS_LOADED`.

`RESULT_INVALID_PARAMETER`: Invalid function parameters (e. g. `NULL`).

`RESULT_NO_OBJECT`: Requested symbol does not exist in the symbol list.

`RESULT_DISABLED`: Symbol load was disabled by the setting `DontLoadSymbolsFromPlc=1` (only interfaces ARTI3 and Gateway3)

3.7.6 GetAddressOfMappedItem

```
long ::GetAddressOfMappedItem(/*[In]*/ char *pszSymbol,  
/*[InOut]*/ char *pszMappedAddr, /*[In]*/ long lLen)
```

GetAddressOfMappedItem() retrieves the address string (e. g. %IB5) for the specified symbol. If the requested variable is not mapped to a direct address in the %M, %I or %Q area, the method returns an empty string.

Restrictions:

V2.3 PLCs: The address strings are "calculated" by the PLCHandler. Because of the fact, that the PLCHandler has no chance to retrieve the addressing mode from the PLC (ByteAddressing/WordAddressing and special BitCountOptions in the PLC's IO-Mapping), the calculated address may differ from the address in the CoDeSys-Project. Please check this carefully, before using this method.
V3 PLCs: Feature is only supported for runtime system versions \geq V3.4. SP4.

Attention: Before calling *GetAddressOfMappedItem()*, the method *EnterItemAccess()* must be called first to prevent the internal symbol list to be deleted or modified by another thread. Afterwards *LeaveItemAccess()* must be called.

pszSymbol: Specifies the variable, for which the address string should be read.

pszMappedAddr: Pointer to a buffer, in which the PLCHandler should copy the address string. This buffer must be allocated by the caller. Make sure that the buffer is suitable for the address string. A buffer of 20 bytes should be enough for all currently existing PLCs.

lLen: Size of the buffer in bytes. If the buffer is too small, the address string is truncated by the PLCHandler.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_NOT_CONNECTED`: Symbol information cannot be retrieved, because PLCHandler is not in state `STATE_RUNNING` or `STATE_PLC_NOT_CONNECTED_SYMBOLS_LOADED`.

`RESULT_INVALID_PARAMETER`: Invalid function parameters (e. g. `NULL`).

`RESULT_NO_OBJECT`: Requested symbol does not exist in the symbol list.

`RESULT_DISABLED`: Symbol load was disabled by the setting `DontLoadSymbolsFromPlc=1` (only interfaces ARTI3 and Gateway3)

3.8 Cyclic Update of Variables

Most of the client applications, which have to read variables from PLCs, read the same set of variables very often. For example in a visualization there are typically background variables, that are monitored all the time (machine state, values for alarm limits to be checked), and the variables needed for displaying the current page or frame inside the visualization. To minimize the communication overhead, the PLCHandler serves the feature of cyclic lists. So for each page and also the background variables there can be defined an own cyclic list. For each defined list, the PLCHandler creates an *UpdateThread*, which reads the values of the variables in the configured interval and copies it to an internal cache. All read accesses from the application will then get the values from the cache. Furthermore the PLCHandler can optionally call a user defined method, if a new update has finished or if a value has changed.

3.8.1 CycDefineVarList

```
HCYCLIST ::CycDefineVarList(/*[In]*/ char **ppszSymbols,
/*[In]*/ unsigned long ulNumOfSymbols,
/*[In]*/ unsigned long ulUpdateRate,
/*[In]*/ CPLCHandlerCallback *pUpdateReadyCallback = NULL,
/*[In]*/ CPLCHandlerCallback *pDataChangeCallback = NULL,
/*[In]*/ RTS_HANDLE hUpdateEvent = 0)
```

This method is used to create a list of variables, whose values should be updated cyclically by the PLCHandler. The method returns a handle for the list. This handle must be passed with all further method calls.

ppszSymbols: List of pointers to variable names. These variables will be read cyclically from the PLC.

ulNumOfSymbols: This parameter specifies the number of variables in the list.

ulUpdateRate: Specifies the update rate for the cyclic update of the variable values in ms.

pUpdateReadyCallback: Pointer to an instance of a derivative class of *CPLCHandlerCallback*. Each time the PLCHandler has updated the internal cache, the *Notify* method of this class is called. The callback can be deactivated by setting this parameter to NULL.

Attention: The callback is direct called from the update thread of the cyclic list. So the execution time of the callback may have an impact to the effective update rate, also called operating rate.

pDataChangeCallback: Pointer to an instance of a derivative class of *CPLCHandlerCallback*. After the PLCHandler has updated the internal cache, the values are compared with the last values in the cache. If there is a difference, the *Notify* method of this class is called. This callback is also called after the first read of the values from the PLC. The callback can be deactivated by setting this parameter to NULL.

Attention: The callback is direct called from the update thread of the cyclic list. So the execution time of the callback may have an impact to the effective update rate, also called operating rate.

hUpdateEvent: This event allows triggering the update from outside. The passed event must be created first with the runtime api function *CAL_SysEventCreate*. This feature is usually not be used, hence 0 should be passed here.

Return value: If no error occurs, the function returns a handle for the cyclic list. Otherwise, it returns ...

NULL: List could not be created. Typical reasons are:

- The PLCHandler is not in the state *STATE_RUNNING*.
- The list with variable names is empty or it contains at least one variable name, which is not available in the internal symbol list of the PLCHandler and also not in the PLC.

3.8.2 CycUpdateVarList

```
HCYCLIST ::CycUpdateVarList(/*[In]*/ HCYCLIST hCycVarList,
/*[In]*/ char **ppszSymbols,
/*[In]*/ unsigned long ulNumOfSymbols,
/*[In]*/ unsigned long ulUpdateRate,
/*[In]*/ CPLHandlerCallback *pUpdateReadyCallback = NULL,
/*[In]*/ CPLHandlerCallback *pDataChangeCallback = NULL,
/*[In]*/ RTS_HANDLE hUpdateEvent = 0)
```

This method can be used to modify an existing variable list. For the PLCHandler this is the same as the combination of *CycDeleteVarList()* and *CycDefineVarList()*, with the only difference, that the cyclic list does not change the handle. This function should only be used, if this is a necessary condition for the client application.

hCycVarList: Handle of a previous defined variable list. If the function succeeds, this will be returned.

All other parameters: see above.

Return value: If no error occurs, the function returns a handle for the cyclic list. Otherwise, it returns ...

NULL: List could not be created. Typical reasons are:

- The passed variable list handle was not a valid handle.
- The PLCHandler is not in the state `STATE_RUNNING`.
- The list with variable names is empty or it contains at least one variable name, which is not available in the internal symbol list of the PLCHandler and also not in the PLC.

3.8.3 CycDeleteVarList

```
long ::CycDeleteVarList(/*[In]*/ HCYCLIST hCycVarList,
/*[In]*/ int bKeepalive = 1)
```

CycDeleteVarList() is used to delete a list of variables which had been set up for cyclic update. As part of this also the update thread for this list is terminated.

hCycVarList: Specifies the handle which has been returned by *CycDefineVarList()*.

bKeepalive: Here you define, whether the PLCHandler should start a so-called "Keepalive" if the current list is the last cyclic list which gets deleted. In this case typically a thread will be started in the PLCHandler, sending cyclically a service to the PLC, in order that the PLC will not log out if no further services are sent. Should always be set to 1, to let the PLCHandler manage this thread automatically.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: no list to delete.

3.8.4 CycIsValidList

```
long ::CycIsValidList(/*[In]*/ HCYCLIST hCycVarList)
```

This method can be used to let the PLCHandler check, if a handle is a valid variable list handle or not.

hCycVarList: Specifies the handle to be checked.

Return value: TRUE if the handle is valid, otherwise FALSE.

3.8.5 CycEnterVarAccess

```
long ::CycEnterVarAccess(/*[In]*/ HCYCLIST hCycVarList)
```

Protects the access to the variable list, so it can not be changed or removed by the update thread of the variable list. To be used only in combination with *CycReadVars()* (see there for details).

hCycVarList: Handle of the variable list, which should be prepared for read.

Return value: TRUE if the handle is valid and the semaphore could be entered, otherwise FALSE.

3.8.6 CycLeaveVarAccess

```
Void ::CycLeaveVarAccess(/*[In]*/ HCYCLIST hCycVarList)
```

Releases the access to the variable list. To be used only in combination with *CycReadVars()* (see there for details).

hCycVarList: Handle of the variable list, which should be released.

Return value: -

3.8.7 CycReadVars

```
long ::CycReadVars(/*[In]*/ HCYCLIST hCycVarList,
/*[Out]*/ PlcVarValue ***ppValues,
/*[Out]*/ unsigned long *pulNumOfValues)
```

The method *CycReadVars()* accesses the values of a cyclic list. The values are read from the PLCHandler's cache, which is updated by the update thread of the PLCHandler in the configured interval.

Attention: Before calling *CycReadVars()* the method *CycEnterVarAccess()* must be called first to prevent the cache for this variable list to be modified or deleted by the update thread. After accessing *ppValues*, *CycLeaveVarAccess()* must be called. Make sure, that after the release of the variable list *ppValues* is not referenced anymore. Because of the fact, that these semaphores block the internal update thread of the cyclic variable list, the access time should be as short as possible, to have an minimal effect to the operating rate of the list.

This method can be directly called from the *Notify* method of the *UpdateReadyCallback* or *DataChangeCallback*, to get the values synchronized with the updates of the cache. If called directly from the callback, then it is not allowed to enclose it with calls of *CycEnterVarAccess()* and *CycLeaveVarAccess()*.

Intended use of this function, assuming there is a valid list defined:

```
HCYCLIST hCycVarList = NULL;
PlcVarValue **ppValues = NULL;
unsigned long ulNumOfValues = 0;

// Preconditions:
// - PLCHandler is in state STATE_RUNNING
// - hCycVarList is a valid handle to a cyclic variable list:
//   hCycVarList = pPlcHandler->CycDefineVarList(...)

if (pPlcHandler->CycEnterVarAccess(hCycVarList))
{
    lResult = pPlcHandler->CycReadVars(hCycVarList, &ppValues,
&ulNumOfValues);
    if (lResult != RESULT_NO_UPDATE && lResult != RESULT_OK)
    {
        // (communication) error occurred, ppValues is invalid
        pPlcHandler->CycLeaveVarAccess(hCycVarList);
        pPlcHandler->CycDeleteVarList(hCycVarList);
        hCycVarList = NULL;
    }
    else
    {
        if (ppValues != NULL)
        {
            //
            // ppValues can be accessed here to read the values
            //
            ppValues = NULL; // Do not access ppValues anymore
            pPlcHandler->CycLeaveVarAccess(hCycVarList);
        }
    }
}
}
```

hCycVarList: Handle of the variable list, which should be read.

pppValues: Pointer to Pointer to Pointer to *PlcVarValue*. After the call **pppValues* contains a direct reference to the internal cache of the PLCHandler for this cyclic variable list.
Attention: This object must not be deleted by the application.

pulNumOfValues: Pointer to a variable of type unsigned long, to return the number of variables.

Return value: If no error occurs, the function *CycReadVars()* returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_INVALID_PARAMETER`: Invalid function parameters (e. g. `NULL`).

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: Invalid list specified.

`RESULT_NO_UPDATE`: Cyclist list was defined and `UpdateThread` is running, but the first read of the variable list has not yet been finished. Try again later to read the variables.

The *PlcVarValue* structure contains the value for each variable.

```
struct PlcVarValue
{
    unsigned long    ulTimeStamp;    UTC-Timestamp of the value, is read from the PLC.
                                    For the interfaces Simulation and Simulation3 ulTimeStamp
                                    is taken from the local machine.
                                    Attention: The UTC-timestamp is not supported by all PLCs.
    unsigned char    bQuality;       Quality of the value:
                                    TRUE: Value was read from the PLC.
                                    FALSE: Value was not read from the PLC.
                                    For the interfaces Simulation and Simulation3 bQuality
                                    is always set to TRUE.
    unsigned char    byData[1];      Array with the value. The size of this array depends on the
                                    data type (byData[1] is only a placeholder with the minimum
                                    size). Should only be accessed, if bQuality is TRUE.
};
```

3.8.8 CycReadChangedVars

```
long ::CycReadChangedVars( /*[In]*/HCYCLIST hCycVarList,
 /*[Out]*/PlcVarValue ***pppChangedValues,
 /*[Out]*/unsigned long **ppChangedValuesIndex,
 /*[Out]*/unsigned long *pulNumOfChangedValues)
```

The method *CycReadChangedVars()* must be called only from the *Notify* method of the *DataChangeCallback*. Therefore it is not allowed to enclose it with calls of *CycEnterVarAccess()* and *CycLeaveVarAccess()*. In opposite of *CycReadVars()* this function return only variables, which have been changed by the last update.

hCycVarList: Handle of the variable list, whose changed variables should be read.

pppChangedValues: Pointer to Pointer to Pointer to *PlcVarValue*. After the call **pppChangedValues* contains a direct reference to the internal cache of the PLCHandler for this cyclic variable list. This list contains only a subset of the variable list with the changed values.
Attention: This object must not be deleted by the application.

ppChangedValuesIndex: Pointer of pointer to a variable of type unsigned long, to return the number of changed variables. In **ppChangedValuesIndex* the index list of the changed variables is returned. Example: If there was a variable list defined with the variables "a", "b", "c", "d" and "e" and the values of the variables "a", "d" and "e" have been changed, then this list will contain 3 entries: 0, 3, 4. So also **pppChangedValues* will point to a list with 3 elements, these are exactly the values for the variables "a", "d" and "e".
Attention: This object must not be deleted by the application.

pulNumOfChangedValues: Pointer to a variable of type unsigned long, to return the number of changed variables. This is the actual size of the list with the changed values

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameters (e. g. `NULL`).

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: Invalid list specified.

`RESULT_NO_UPDATE`: Cyclist list was defined and `UpdateThread` is running, but the first read of the variable list has not yet been finished (only occurs, if the method is misused).

3.8.9 CycGetOperatingRate

```
unsigned long ::CycGetOperatingRate(/*[In]*/HCYCLIST hCycVarList)
```

This method returns the ratio between the last needed time to update the variable list and the configured update rate in percent: $\text{CycGetOperatingRate} = (\text{needed time to update} / \text{configured update rate}) * 100$. The needed time includes also the execution time of both, the *UpdateReadyCallback* and the *DataChangeCallback*.

hCycVarList: Handle of the variable list, whose operating rate should be returned.

Return value: If no error occurs, the function returns the last measured operating rate. Otherwise, it returns ...

`0xFFFFFFFF`: The passed handle is an invalid variable list handle.

3.8.10 CycSetUpdateRate

```
long ::CycSetUpdateRate(/*[In]*/HCYCLIST hCycVarList,  
/*[In]*/ unsigned long ulUpdateRate)
```

This method reconfigures the update rate of a cyclic list. Typical use case: At startup the visualization client defines for each page an own variable list. The update rate of the visualization's variable list for the start startup page is set to a proper value and all others are set to a very high value. On each page switch the client sets the update rate of the variable list for the old page to the high value and for the new page to a low (useful) value. This results in a very fast page switch, because there is no need of deleting or defining variable lists.

hCycVarList: Handle of the variable list, whose update rate should be set.

ulUpdateRate: New update rate of the cyclic list in ms. The next update is triggered, if the time since the last update has elapsed the *ulUpdateRate*.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (`ulUpdateRate = 0`).

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: Invalid list or no list variables to read.

3.8.11 CycGetUpdateRate

```
unsigned long ::CycGetUpdaterate(/*[In]*/HCYCLIST hCycVarList)
```

This method reads the configured update rate of a cyclic list.

hCycVarList: Handle of the variable list, whose operating rate should be returned.

Return value: If no error occurs, the function returns the configured update rate in ms. Otherwise, it returns

`0xFFFFFFFF`: The passed handle is an invalid variable list handle.

3.8.12 CycGetSymbolList

```
long ::CycGetSymbolList(/*[In]*/HCYCLIST hCycVarList,  
/*[Out]*/ char ***pppCycSymbolList)
```

This method retrieves the list of symbols of the specified cyclic variable list.

hCycVarList: Handle of the variable list, whose symbol list should be read.

pppCycSymbolList: Pointer to pointer to pointer to char. In **pppCycSymbolList* a direct reference to the names of the variable list is returned.

Attention: This object must not be deleted by the application.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: Handle of the variable list is invalid.

`RESULT_FAILED`: Variable list does not contain variables.

3.8.13 CycGetNumOfLists

```
unsigned long ::CycGetNumOfLists(void)
```

CycGetNumOfLists() returns the number of the currently defined cyclic variable lists.

Return value: Number of the currently defined cyclic variable lists.

3.8.14 CycGetVarListByIndex

```
HCYCLIST ::CycGetVarListByIndex(/*[In]*/ unsigned long ulIndex)
```

CycGetVarListByIndex() returns the handle of the cyclic variable list, which is referenced by the index in the PLCHandler's internal variable list management.

ulIndex: Index of the variable list.

Return value: Handle of the requested cyclic list, or `NULL`, if the variable list does not exist.

3.8.15 CycGetVarListIndex

```
long ::CycGetVarListIndex(/*[In]*/HCYCLIST hCycVarList)
```

CycGetVarListIndex() returns the index of a cyclic variable list in the PLCHandler's internal variable list management.

Attention: The index of a cyclic variable list may change, if another cyclic variable list is deleted.

hCycVarList: Handle of the variable list, whose index should be retrieved.

Return value: Index of the requested cyclic list, or `-1`, if the variable list does not exist.

3.8.16 CycAddSymbolsToVarlist

```
HCYCLIST ::CycAddSymbolsToVarList (/*[In]*/ HCYCLIST hCycVarList,  
/*[In]*/ char **ppszSymbols, /*[In]*/ unsigned long ulNumOfSymbolsToAdd)
```

Only supported for V3 interfaces and runtime system versions \geq V3.5.

This method adds variables to an already existing cyclic variable list. Variables are always added at the end of the existing list. The application has to prepare all its management structures before calling this function, because this modification is applied immediately and may for example generate a `DataChangeCallback` before *CycAddSymbolsToVarList()* returns.

hCycVarList: Handle of the variable list, to which variables should be added.

ppszSymbols: List of pointers to variable names. These variables will be added to the cyclic list.

ulNumOfSymbolsToAdd: This parameter specifies the number of variables to be added.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NO_CYCLIC_LIST_DEFINED`: Handle of the variable list is invalid.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this feature.

RESULT_FAILED: Common error in the underlying interfaces.

3.8.17 CycRemoveSymbolsFromVarlist

```
HCYCLIST ::CycRemoveSymbolsFromVarList (/*[In]*/ HCYCLIST hCycVarList,  
/*[In]*/ char **ppszSymbols,  
/*[In]*/ unsigned long ulNumOfSymbolsToRemove)
```

Only supported for V3 interfaces and runtime system versions >= V3.5.

This method removes variables from an already existing cyclic variable list. The variables to be removed are specified by name. The modification is applied immediately.

hCycVarList: Handle of the variable list, from which the variables should be removed.

ppszSymbols: List of pointers to variable names. These variables will be removed from the cyclic list.

ulNumOfSymbolsToRemove: This parameter specifies the number of variables to be removed.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL) or at least one of the symbols to be removed is not a member of the cyclic list.

RESULT_PLC_NO_CYCLIC_LIST_DEFINED: Handle of the variable list is invalid.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_NOT_SUPPORTED: The underlying interface does not support this feature.

RESULT_FAILED: Common error in the underlying interfaces.

The following method allows to specify the variables by the position (index) inside the list instead of the name. This avoids the string comparison to find the variables to be removed. The index list is only accepted by the PLCHandler, if the list's content is strictly monotonic increasing. That means, there are no duplicate allowed and the indices must be sorted in ascending order.

```
HCYCLIST ::CycRemoveSymbolsFromVarList (/*[In]*/ HCYCLIST hCycVarList,  
/*[In]*/ unsigned long *pulRemoveIndexList,  
/*[In]*/ unsigned long ulNumOfSymbolsToRemove)
```

Only supported for V3 interfaces and runtime system versions >= V3.5.

This method removes variables from an already existing cyclic variable list. . The variables to be removed are specified by the position (index). The modification is applied immediately.

hCycVarList: Handle of the variable list, from which the variables should be removed.

pulRemoveIndexList: List of indices of the variables to remove.

ulNumOfSymbolsToRemove: This parameter specifies the number of variables to be removed.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL) or the *pulRemoveIndexList* contains invalid members or is not correctly sorted.

RESULT_PLC_NO_CYCLIC_LIST_DEFINED: Handle of the variable list is invalid.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_NOT_SUPPORTED: The underlying interface does not support this feature.

RESULT_FAILED: Common error in the underlying interfaces.

3.9 Synchronous variable access

In opposite to the feature of the cyclic variable lists, the PLCHandler provides also synchronous variable access to read or write variables only one time and not cyclically. Typically use cases are

input variables in a visualization, which must be written to the PLC, or variables, which should be read only at user request.

All these methods send messages to the PLC and wait for the answer. The functions return after getting the answer from the PLC or if an error occurs.

3.9.1 SyncReadVarsFromPlc

```
HVARLIST ::SyncReadVarsFromPlc(/*[In]*/ char **ppszSymbols,
/*[In]*/ unsigned long ulNumOfSymbols,
/*[Out]*/ PlcVarValue ***pppValues,
/*[Out]*/ unsigned long *pulNumOfValues)
```

The method *SyncReadVarsFromPlc()* can be used to read the values of a list of variables from the PLC.

Attention: *SyncReadVarsFromPlcReleaseValues()* must be called to deallocate the list of read variables, after the read values and the list is not be used anymore.

ppszSymbols: List of pointers to variable names. These variables will be read one time from the PLC.

ulNumOfSymbols: Number of variables in the list.

pppValues: Pointer to Pointer to Pointer to *PlcVarValue* (see method *CycReadVars()* for a detailed description of this structure). After the call **pppValues* contains a direct reference to the values of the PLCHandler for this variable list.

Attention: This object must not be deleted by the application.

pulNumOfValues: Pointer to a variable of type unsigned long, to return the number of variables.

Return value: If no error occurs, the function returns a handle for the list. Otherwise, it returns ...

NULL: Function failed.

```
Long ::SyncReadVarsFromPlc(/*[In]*/ HVARLIST hVarList,
/*[Out]*/ PlcVarValue ***pppValues,
/*[Out]*/ unsigned long *pulNumOfValues)
```

This method allows reading the passed variable list again. In opposite to the previous method the variable list is referenced by the variable list handle and not by the list of variable names.

Attention: *SyncReadVarsFromPlcReleaseValues()* must be called to deallocate the list of read variables, after the read values and the list is not be used anymore.

hVarList: Handle of the variable list, which was returned by the first call of *SyncReadVarsFromPlc()*, which has defined this list.

All other parameters: see above.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER:` Invalid function parameter (e. g. NULL).

`RESULT_PLC_NOT_CONNECTED:` PLCHandler is not in state `STATE_RUNNING`.

`RESULT_EXCEPTION:` Handled exception occurred in the underlying interfaces. Typically the variable list contains invalid elements.

`RESULT_FAILED:` Common error in the underlying interfaces.

3.9.2 SyncReadVarsFromPlcReleaseValues

```
long ::SyncReadVarsFromPlcReleaseValues(/*[In]*/ HVARLIST hSyncRead)
```

SyncReadVarsFromPlcReleaseValues() must be called to deallocate the list of read variables, after the read values and the list is not be used anymore.

Intended use of this function:

```

HCYCLIST hSyncReadList = NULL;
char **ppReadItems = NULL;
PlcVarValue **ppValues = NULL;
unsigned long ulNumOfValues = 0;
long lResult = RESULT_FAILED;

// Precondition: PLCHandler is in state STATE_RUNNING
// Handling of return values is not part of the example and must be added

ppReadItems = new char*[2];
ppReadItems[0] = new char[strlen("MySymbol1")+1];
strcpy(ppReadItems[0], "MySymbol1");
ppReadItems[1] = new char[strlen("MySymbol2")+1];
strcpy(ppReadItems[1], "MySymbol2");

hSyncReadList = pPlcHandler->SyncReadVarsFromPlc(ppReadItems, 2,
&ppValues, &ulNumOfValues);

//
// ppValues can be accessed here to get the values
//

// (repeated) optional call of SyncReadVarsFromPlc()
lResult = pPlcHandler->SyncReadVarsFromPlc(hSyncReadList, &ppValues,
&ulNumOfValues);

//
// ppValues can be accessed here to get the values
//

lResult = pPlcHandler->SyncReadVarsFromPlcReleaseValues(hSyncReadList);

// do not access ppValues or hSyncReadList anymore
hSyncReadList = NULL;

delete [] ppReadItems[0];
delete [] ppReadItems[1];
delete [] ppReadItems;
ppValues = NULL;

```

hVarList: Handle of the variable list, which was returned by the first call of *SyncReadVarsFromPlc()*, which has defined this list.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER:` Invalid function parameter (e. g. `NULL`).

`RESULT_FAILED:` Common error in the underlying interfaces or `PLCHandler` is not in state `STATE_RUNNING`.

3.9.3 SyncWriteVarsToPlc

```

long ::SyncWriteVarsToPlc(/*[In]*/ char **ppszSymbols,
/*[In]*/ unsigned long ulNumOfSymbols,
/*[In]*/ unsigned char **ppbyValues)

```

This method can be used to transfer a list of variable values to the PLC. The values are written directly to the PLC.

ppszSymbols: List of pointers to variable names. These variables will be written one time to the PLC.

ulNumOfSymbols: Number of variables in the list

ppbyValues: List of pointers to the variables values which should be written to the PLC.
Attention: Must match to the list of the specified variables!

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_EXCEPTION: Handled exception occurred in the underlying interfaces. Typically the variable list contains invalid elements or the value list does not match to the variable list.

RESULT_FAILED: Common error in the underlying interfaces.

3.10 File and directory functions

The following functions allow interacting with the PLC's file system.

All these methods send messages to the PLC and wait for the answer. The functions return after getting the answer from the PLC or if an error occurs.

3.10.1 UploadFile

```
long ::UploadFile(/*[In]*/ char *pszPlc, /*[In]*/ char *pszHost = NULL)
```

The method *UploadFile()* can be used to transfer files from the PLC to the client.

pszPlc: Specifies the name of the file on the PLC.

pszHost: Specifies the name of the file on the client. If NULL, then *pszPlc* is also used for the host's file name.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

RESULT_PLC_FAILED: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the requested file does not exist on the PLC.

3.10.2 DownloadFile

```
long ::DownloadFile /*[In]*/ char *pszHost, /*[In]*/ char *pszPlc = NULL)
```

The method *DownloadFile()* can be used to transfer files from the client to the PLC.

pszHost: Specifies the name of the file on the client.

pszPlc: Specifies the name of the file on the PLC. If NULL, then *pszHost* is also used for the PLC's file name.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

RESULT_PLC_FAILED: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the file could not be written to the PLC's file system.

3.10.3 RenameFile

```
long ::RenameFile(/*[In]*/ char *pszOldFile, /*[In]*/ char *pszNewFile)
```

The method *RenameFile()* can be used to rename a file in the PLC's file system.

pszOldFile: Specifies the name of the existing file.

pszNewFile: Specifies the new name of the file.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the file does not exist in the PLC's file system or that the file is write protected.

3.10.4 DeleteFile

```
long ::DeleteFile(/*[In]*/ char *pszFileName)
```

The method *RenameFile()* can be used to delete a file in the PLC's file system.

pszFileName: Specifies the name of the file to delete.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the file does not exist in the PLC's file system or that the file is write protected.

3.10.5 ReadDirectory

```
long ::ReadDirectory(/*[Out]*/ CdirInfo **ppdi,  
/*[In]*/ char *pszBaseDir = NULL)
```

This method can be used to read the directory structure on the PLC.

Example for the use of this function:

```
long lResult;  
CdirInfo *pdi = NULL;  
  
lResult = pPLCHandler->ReadDirectory(&pdi, "");  
if (lResult == RESULT_OK)  
{  
    // Dump the directory contents  
    for (int i=0; i<pdi->GetNumOfEntries(); i++)  
    {  
        char *psz;  
        int bDir;  
  
        pdi->GetEntry(&psz, &bDir, i);  
        if (bDir)  
            printf("<DIR>");  
        printf("\t\t%s\n", psz);  
    }  
    delete pdi;  
    pdi = NULL;  
}
```

ppdi: Pointer to pointer to *CdirInfo*. In **ppdi* a pointer to an instance of the *CdirInfo* class is returned. This object contains the complete directory structure (including files and sub-directories).

Attention: The returned object must be deleted by the application.

pszBaseDir: Specifies an optional basic folder. Can be `NULL` or an empty string.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the directory does not exist in the PLC's file system.

The function returns in **ppdi* a pointer to an instance of the *CdirInfo* class. This class is defined as:

```
class PLCH_DLL_DECL CdirInfo
{
public:
    CdirInfo();
    ~CdirInfo();

    long GetNumOfEntries(void);
    long GetEntry(char **ppszEntry, int *pbDirectory, long lEntry);
    long AddEntry(char *szEntry, int bDirectory);

private:
    _DirInfo    **m_ppdi;
    long m_lEntries;
};
```

To get the directory content item by item, this class provides to access methods: *GetNumOfEntries()* and *GetEntry()*. See above for an example of the typical usage.

3.10.6 CreateDirectory

```
long ::CreateDirectory(/*[In]*/ char *pszDirectoryName)
```

Only supported for V3 PLCs.

The method *CreateDirectory()* can be used to create a new directory in the PLC's file system.

pszDirectoryName: Specifies the name of the directory to create.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the directory could not be created.

3.10.7 RenameDirectory

```
long ::RenameDirectory(char *pszOldDirectory,
/*[In]*/ char *pszNewDirectory)
```

Only supported for V3 PLCs.

The method *RenameDirectory()* can be used to rename a directory in the PLC's file system.

pszOldDirectory: Specifies the name of the existing directory.

pszNewDirectory: Specifies the new name of the directory.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

RESULT_PLC_FAILED: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the directory does not exist in the PLC's file system.

3.10.8 DeleteDirectory

```
long ::DeleteDirectory(/*[In]*/ char *pszDirectoryName)
```

Only supported for V3 PLCs.

The method *DeleteDirectory()* can be used to delete an existing directory in the PLC's file system.

pszDirectoryName: Specifies the name of the directory to delete.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

RESULT_PLC_FAILED: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is that the directory does not exist in the PLC's file system.

3.11 Retrieve project and application information from the PLC

These functions can be used to retrieve some information about the actual PLC project and the applications. For V2 PLCs, only the *ProjectInfo* is available, for V3 both the *ProjectInfo* and *ApplicationInfo* can be retrieved.

For V3 PLCs *GetProjectInfo()* and *GetApplicationInfo()* is only supported by runtime system versions >= V3.4.1.0. Furthermore both information are optional and linked to an application. This is also true for the project information, because it is possible to download several applications to one PLC, which can be part of different CoDeSys projects.

The structure *ProjectInfo* is defined as:

```
struct ProjectInfo
{
    unsigned long    ulProjectId;           V2 PLC: ProjectId of the currently running project
                                                V3 PLC: not supported, always 0.
    unsigned long    ulTimestampUtc;       V2 PLC: UTC build time in seconds since 1.1.1970
                                                V3 PLC: not supported, always 0.
    char*            pszProject;           Name of the project file
    char*            pszTitle;             Title of the project
    char*            pszVersion;           Version of the project
    char*            pszAuthor;            Author of the project
    char*            pszDescription;        Description of the project
};
```

The structure *ApplicationInfo* is defined as:

```
struct ProjectInfo
{
```

```

        unsigned long  ullLastChanges;          UTC build time in seconds since 1.1.1970
        char*          pszProject;             Name of the project file
        char*          pszVersion;            Version of the application
        char*          pszAuthor;             Author of the application
        char*          pszDescription;        Description of the application
        char*          pszProfile;           Profile name of the used CoDeSys IDE
};

```

All off the following methods send messages to the PLC and wait for the answer. The functions return after getting the answer from the PLC or if an error occurs.

3.11.1 GetApplicationList

```

long ::GetApplicationList(/*[Out]*/ char ***pppszApplications,
/*[Out]*/ unsigned long *pulNumOfApplications)

```

Only supported for V3 PLCs.

GetApplicationList() retrieves the list with the names of all applications on the PLC. The names are needed for example to get or set the status of a single application.

pppszApplications: Pointer to pointer to pointer to char. In **pppszApplications* a direct reference to the internal list of application names is returned. If *pppszApplications* = NULL, only the number of applications on the PLC can be read.

Attention: The returned object must not be deleted by the application.

pulNumOfApplications: Pointer to a variable of type PLC_STATUS to which the number of variables should be written to.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

3.11.2 GetProjectInfo

```

long ::GetProjectInfo(/*[Out]*/ ProjectInfo **ppPrjInfo)

```

Method to retrieve the project info structure of the currently running PLC project. The content is stored as part of the project and can be edited within the CoDeSys tool. This function is intended for the use with V2 PLCs, but supports a compatibility mode for V3 PLCs. This means, the function works for V3 PLCs as expected, if there is exactly one application. If there are more applications on the PLC, this function will ask each application for the project info and return the first found one.

ppPrjInfo: Pointer to Pointer to *ProjectInfo*. After the successful call **ppPrjInfo* contains a direct reference to the internal project info structure of the PLCHandler. **ppPrjInfo* is set to NULL, if no project information exists on the PLC. Furthermore each pointer of the structure can be set to NULL or can point to an empty string, if a partial information is not defined on the PLC.

Attention: This object must not be deleted by the application.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V3 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

3.11.3 GetApplicationInfo

```
long ::GetApplicationInfo(/*[In]*/ char *pszApplication,
/*[Out]*/ ProjectInfo **ppPrjInfo, /*[Out]*/ ApplicationInfo **ppAppInfo)
```

Only supported for V3 PLCs.

Method to retrieve both the project info structure and the application info structure of the specified application. The content is stored as part of the project and can be edited within the CoDeSys tool.

pszApplication: Name of the application, whose information should be read.

ppPrjInfo: Pointer to Pointer to *ProjectInfo*. After the successful call **ppPrjInfo* contains a direct reference to the internal project info structure of the PLCHandler. **ppPrjInfo* is set to NULL, if the application does not exist on the PLC, or if the application provides no project information. Furthermore each pointer of the structure can be set to NULL or can point to an empty string, if a partial information is not defined on the PLC.
Attention: This object must not be deleted by the application.

ppAppInfo: Pointer to Pointer to *ApplicationInfo*. After the successful call **ppAppInfo* contains a direct reference to the internal application info structure of the PLCHandler. **ppAppInfo* is set to NULL, if the application does not exist on the PLC or if the application provides no application information. Furthermore each pointer of the structure can be set to NULL or can point to an empty string, if a partial information is not defined on the PLC.
Attention: This object must not be deleted by the application.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V3 PLC.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

3.12 Get, set and reset the status of the application/PLC

The following functions can be used to get and set the status (RUN/STOP) of the PLC. Additional there are different reset commands supported. For V3 PLCs the status can be retrieved and modified for each application separately.

All these methods send messages to the PLC and wait for the answer. The functions return after getting the answer from the PLC or if an error occurs.

The status is stored in a variable of type `PLC_STATUS`, which is defined as:

```
typedef enum
{
    PLC_STATE_RUNNINING    = 0,          RUN
    PLC_STATE_STOP         = 1,          STOP
    PLC_STATE_STOP_ON_BP   = 2,          Stopped on a breakpoint
    PLC_STATE_UNKNOWN      = 255        PLCHandler could not get the state of the
PLC
} PLC_STATUS;
```

The PLCHandler allows the following reset commands, which have to be passed in a variable of the type `RESET_OPTION`:

```
typedef enum
{
    PLC_RESET_WARM    = 0,          Stops the project/application and initialize variables
    PLC_RESET_COLD    = 1,          Same as 0, but initialize also retain variables
    PLC_RESET_ORIGIN  = 2,          Stops the project/application and deletes it from PLC
} RESET_OPTION;
```

3.12.1 GetPlcStatus

```
long ::GetPlcStatus( /*[Out]*/ PLC_STATUS *pPlcStatus)
```

Method to get the current status (RUN/STOP) of the PLC. This function is intended for the use with V2 PLCs, but supports a compatibility mode for V3 PLCs. This means, the function works for V3 PLCs as expected, if there is exactly one application or if all applications have the same status. If there is no application or if the status of the applications does not match to each other, the function will return the status `PLC_STATE_UNKNOWN`. Furthermore this value is returned, if one or more applications are in an unstable status during debugging.

For V2 PLCs the returned status may have a value, which is not defined in the enum above. Such states are transitional states during debugging. In this case the function should be called again to get the next stable status.

pPlcStatus: Pointer to a variable of type `PLC_STATUS` to which the status should be written to.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

3.12.2 SetPlcStatus

```
long ::SetPlcStatus( /*[In]*/ PLC_STATUS PlcStatus)
```

Method to set the current status (RUN/STOP) of the PLC. This function is intended for the use with V2 PLCs, but supports a compatibility mode for V3 PLCs. For V3 PLCs the status of all applications is set to the new value.

PlcStatus: Variable of type `PLC_STATUS` with the new state. Only the values `PLC_STATE_RUNNING` and `PLC_STATE_STOPPED` are valid.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

3.12.3 ResetPlc

```
long ::ResetPlc ( /*[In]*/ RESET_OPTION ResetCommand)
```

Method to reset the PLC. This function is intended for the use with V2 PLCs, but supports a compatibility mode for V3 PLCs. On V3 PLCs all applications are reset.

ResetCommand: Variable of type `RESET_OPTION` to specify the reset type (see above).

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid reset option.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

3.12.4 GetApplicationStatus

```
long ::GetApplicationStatus( /*[Out]*/ PLC_STATUS *pAppStatus,  
 /*[In]*/ char *pszApplication = NULL)
```

Only supported for V3 PLCs.

Method to get the current status (RUN/STOP) of an application on the PLC. If the application is in an unstable status during debugging, the function will return the status `PLC_STATE_UNKNOWN`. In this case the function should be called again to get the next stable status.

pAppStatus: Pointer to a variable of type PLC_STATUS to which the status should be written to.

pszApplication: Name of the application, whose status should be read. If NULL, then the state of all applications is read. If there is no or no matching application or if the status of the applications does not match to each other, the function will return the status PLC_STATE_UNKNOWN.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid function parameter (e. g. NULL).

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

3.12.5 SetApplicationStatus

```
long ::SetApplicationStatus(/*[In]*/ PLC_STATUS AppStatus,  
/*[In]*/ char *pszApplication = NULL)
```

Only supported for V3 PLCs.

Method to set the current status (RUN/STOP) of an application on the PLC.

AppStatus: Variable of type PLC_STATUS with the new state. Only the values PLC_STATE_RUNNNG and PLC_STATE_RUNNNG are valid.

pszApplication: Name of the application, whose status should be set. If NULL, then the state of all applications is set to the new status.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

3.12.6 ResetApplication

```
long ::ResetApplication (/*[In]*/ RESET_OPTION ResetCommand,  
/*[In]*/ char *pszApplication = NULL)
```

Only supported for V3 PLCs.

Method to reset an application on the PLC.

ResetCommand: Variable of type RESET_OPTION to specify the reset type (see above).

pszApplication: Name of the application, which should be reset. If NULL, all applications are reset.

Return value: If no error occurs, the function returns RESULT_OK. Otherwise, it returns ...

RESULT_INVALID_PARAMETER: Invalid reset option.

RESULT_NOT_SUPPORTED: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

RESULT_PLC_NOT_CONNECTED: PLCHandler is not in state STATE_RUNNING.

RESULT_FAILED: Common error in the underlying interfaces.

3.13 Miscellaneous

3.13.1 ReloadBootproject

```
long ::ReloadBootproject(void)
```

In service tools there may be the need to exchange the boot project of a PLC. For doing this, first the new boot project files have to be downloaded to the PLC's file system. After that the PLC have to be triggered to load the new boot project.

This function is intended for the use with V2 PLCs, but supports a compatibility mode for V3 PLCs: In this case the boot project of all applications is (re-)loaded. Make sure you have registered the V3 applications before by calling *RegisterBootApplication()* for each application. For V2 PLCs there is no registration needed.

ReloadBootproject() stops the currently running project on the PLC and initiates the reload of the boot project. This allows to update the PLC project without the CoDeSys editor and without reboot of the PLC.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for the interface `Simulation3`.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the boot project does not exist.

3.13.2 RegisterBootApplication

```
long ::RegisterBootApplication(/*[In]*/ char *pszApplication)
```

Only supported for V3 PLCs.

Method to register a previously downloaded boot application at the runtime system, which will update its list of boot applications, which should be loaded after the next power cycle. Furthermore it will prepare the initialization of retain variables, which is done during the first load of this application. A typical sequence for updating a boot application looks like this:

```
long lResult;  
lResult = pPLCHandler->ResetApplication(PLC_RESET_ORIGIN);  
lResult = pPLCHandler->DownloadFile("Application.app");  
lResult = pPLCHandler->DownloadFile("Application.crc");  
lResult = pPLCHandler->RegisterBootApplication("Application");  
lResult = pPLCHandler->ReloadBootApplication("Application");  
lResult = pPLCHandler->SetApplicationStatus(PLC_STATE_RUNNNGING,  
"Application");
```

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the boot application does not exist.

3.13.3 ReloadBootApplication

```
long :: RegisterBootApplication(/*[In]*/ char *pszApplication = NULL)
```

Only supported for V3 PLCs.

Method to (re-)load a previously downloaded and registered boot application. This is automatically done during the startup of the PLC. So this method can be used instead of restarting the PLC.

Attention: Some PLCs need a power cycle after exchanging the boot application. Please check carefully, if the PLC and the application work in the intended way after calling *ReloadBootApplication()*.

pszApplication: Name of the application, which should be (re-)loaded. If NULL, then all registered applications are reloaded.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the boot application does not exist.

3.13.4 CheckTarget

```
long :: CheckTarget(/*[In]*/ unsigned long ulTargetId,  
/*[In]*/ unsigned long ulHookId = 0,  
/*[In]*/ unsigned long ulMagic = 0)
```

Only supported for V2 PLCs.

This method can be used to verify the type of the connected target. This is usually not needed by the PLCHandler, but is a useful feature for service tools to verify the target type before loading the boot project.

ulTargetId: Unique id of the target, provided by the PLC vendor.

ulHookId: Id of the Hook-DLL, provided by the PLC vendor, typically 0.

ulMagic: Sub number of the target id, provided by the PLC vendor, typically 0.

Return value: If the target matches, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_TARGETID_MISMATCH`: Target does not match to the passed parameters.

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V3 PLC.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_PLC_FAILED`: Service was successfully sent to the PLC, but the PLC was not able to execute this service and returns an error. One possible reason is, that the boot project does not exist.

3.13.5 GetDeviceInfo

```
long :: GetDeviceInfo(/*[Out]*/ DeviceInfo **ppDeviceInfo)
```

Only supported for V3 PLCs.

Method to retrieve the device info structure of the PLC to which the PLCHandler is connected to.

For sending own services (e. g. with *SyncSendService()*) to V3 runtime systems, in general the session id, the effective buffer size and the byte order of the runtime system is needed. After the

PLCHandler is logged in successfully to a PLC, this method allows to read out this and some further information.

The structure *DeviceInfo* is defined as:

```
struct DeviceInfo
{
    char*          pszNodeAddress;          V3 CoDeSys address
    RTS_WCHAR*    pwszNodeName;           Name of the device
    RTS_WCHAR*    pwszTargetName;         Name of the target type
    RTS_WCHAR*    pwszTargetVendorName;   Name of the vendor
    char*         pszTargetVersion;       Version of the device
    unsigned long ulTargetId;             Target id of the device
    unsigned long ulTargetType;           Target type of the device
    unsigned long ulBufferSize;           communication buffer size of the device
                                           No service must exceed this size!
    unsigned long DeviceSessionId;        Session id of the current connection
    unsigned long bMotorola;              Byte order of the device
};
```

ppDeviceInfo: Pointer to Pointer to *DeviceInfo*. After the successful call ****ppDeviceInfo** contains a direct reference to the internal device info structure of the PLCHandler. ****ppDeviceInfo** is set to NULL, if no project information exists on the PLC. Furthermore each pointer of the structure can be set to NULL or can point to an empty string, if a partial information is not available for this PLC.
Attention: This object must not be deleted by the application.

Return value: If the device info is available, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. NULL).

`RESULT_NOT_SUPPORTED`: The underlying interface does not support this service, e. g. the function is called for a connection to a V2 PLC or for a simulation interface.

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

3.13.6 EnterOnlineAccess

```
long :: EnterOnlineAccess(/*[In]*/ unsigned long ulTimeout =
    PLCHANDLER_TIMEOUT_INFINITE)
```

Protects the access to the online interface. Therefore all internal threads (e. g. all update threads of the cyclic lists) of the PLCHandler have to enter this semaphore, before using the online interface. Both the `UpdateReadyCallback` and the `DataChangeCallback` are called while keeping this semaphore, to make sure, that no other thread changes internal states, while the callbacks are executed. Thus this semaphore can also be used to synchronize the access to application data structures between the application and the callbacks.

Intended use of this function (pseudo code only):

Thread inside the application:

```
EnterOnlineAccess()
CycDeleteVarlist(...)
/* -> CycDeleteVarlist calls internally also EnterOnlineAccess()
   -> CycDeleteVarlist deletes the varlist internally
   -> CycDeleteVarlist calls internally LeaveOnlineAccess() */
/* access the application data structures here (e. g. update maps) */
LeaveOnlineAccess()
```

Internal UpdateThread of the PLCHandler:

```
EnterOnlineAccess()  
ReadValuesOfCycListFromPLC()  
Callback-->Notify()  
    /* Notify implementation handles the received variable information  
       and accesses the application data structures */  
LeaveOnlineAccess()
```

ulTimeout: Time in ms after the method should latest return, even if the semaphore could not be entered. The default `PLCHANDLER_TIMEOUT_INFINITE` let wait *EnterOnlineAccess()* for ever.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_TIMEOUT:` Method was called with a timeout value \neq `PLCHANDLER_TIMEOUT_INFINITE`, and the semaphore could not entered within the specified timeout. Check if a reasonable value was for *ulTimeout* was used.

`RESULT_FAILED:` *EnterOnlineAccess()* failed due to internal error.

3.13.7 LeaveOnlineAccess

```
long :: LeaveOnlineAccess(void)
```

Releases the access to the online interface. Must only be called, if *EnterOnlineAccess()* has returned `RESULT_OK`.

Return value: The function returns always `RESULT_OK`.

3.13.8 GetPlcComObject

```
CPLCComBase* :: GetPlcComObject(void)
```

Method to get the communication object (only for internal use and test purposes).

Return value: Pointer to the internal `CPLCCom` object or `NULL`.

3.13.9 Member variable ulCstData

```
unsigned long ulCstData
```

The `PLCHandler` class contains a public member variable of type `unsigned long` to store application (client) data, which is related to an instance of the `PLCHandler`. This data is guaranteed to be untouched by the `PLCHandler`.

3.14 Protected methods for sending any service to the PLC

All of the following methods can only be used, if you derive your class from the `CPLCHandler` class. These methods should typically only be used from the PLC vendor itself, to send his specific services to the PLC.

Attention: The structure of the service must be known in detail! Keep in mind that the services of V3 PLCs differ completely from the V2 PLCs. An erroneous service might cause a crash of the PLC! Due to this reason this methods are only accessible in derivative classes and has not been published straightly.

3.14.1 SyncSendService

```
long :: SyncSendService(/*[In]*/ unsigned char *pbySend,  
/*[In]*/ unsigned long ulSendSize,  
/*[Out]*/ unsigned char **ppbyRecv,  
/*[Out]*/ unsigned long *pulRecvSize)
```

This method can be used to transfer any runtime system service to the PLC. The function returns after getting the answer from the PLC or if an error occurs.

pbySend: Pointer to the byte stream, which should be sent to the PLC. The content depends on the service and the PLC software version to talk with.

ulSendSize: Length of the byte stream to send.

ppbyRecv: Pointer to pointer to unsigned char. After the call **ppbyRecv* contains a pointer to the receive buffer with the PLC's response for the sent service.
If the function is called with *ppbyRecv == 0*, then the service will be sent, but no answer will be returned to the caller.
If the function is called with **ppbyRecv == 0*, then the receive buffer will be allocated by the PLCHandler and the caller is responsible for deleting it.
If the function is called with a receive buffer, which is allocated by the application (**ppbyRecv = <address of the receive buffer>*), then the PLCHandler returns the service reply in this buffer. Make sure, that **pulRecvSize* is set to the buffer size before calling the function.

pulRecvSize: Pointer to unsigned long. After the call **pulRecvSize* will contain the length of the PLC's service replay. Must contain the size of the receive buffer, if this is allocated by the application (**ppbyRecv != 0*).

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_PLC_NOT_CONNECTED`: PLCHandler is not in state `STATE_RUNNING`.

`RESULT_FAILED`: Common error in the underlying interfaces.

`RESULT_EXCEPTION`: An exception occurred in the underlying interface.

3.14.2 AsyncSendService

```
long ::AsyncSendService(/*[Out]*/ int *piInvokeId,  
/*[In]*/ unsigned char *pbySend,  
/*[In]*/ unsigned long ulSendSize,  
/*[In]*/ CPLHandlerCallback *pAsyncServiceCallback = NULL)
```

This method can be used to transfer any runtime system service to the PLC. The function returns immediately after putting the send request in the PLCHandler's send service queue, later on the PLC response must be retrieved by calling *AsyncGetServiceReply()*.

piInvokeId: Pointer to a variable of type `int`, in which the PLCHandler returns the *InvokeId*. The *InvokeId* identifies the request is also returned by *AsyncGetServiceReply()*.

pbySend: see *SyncSendService()*.

ulSendSize: see *SyncSendService()*.

pAsyncServiceCallback: Pointer to an instance of a derivative class of *CPLHandlerCallback*. If the PLCHandler has received the PLC's response to the service, the *Notify* method of this class is called. Usually the *Notify* method calls *AsyncGetServiceReply()* to retrieve the response. The callback can be deactivated by setting this parameter to `NULL`. In this case *AsyncGetServiceReply()* must be called cyclically (for example every 100 ms) to check, if a service answer is available.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. `NULL`).

`RESULT_FAILED`: Common error occurred.

`RESULT_NO_OBJECT`: Internal send service queue is full. Call *AsyncGetServiceReply()* to check for completed services and remove it from the queue.

3.14.3 AsyncGetServiceReply

```
long ::AsyncGetServiceReply (/*[In]*/ int *piInvokeId,  
/*[Out]*/ unsigned char **ppbyRecv,  
/*[Out]*/ unsigned long *pulRecvSize,  
/*[Out]*/ long *plServiceResult)
```

This method checks, if a PLC response for a sent request is available. If this is the case, the function returns the answer.

piInvokeId: Pointer to a variable of type int, in which the PLCHandler returns the *InvokeId*. So the client can assign the answer to the corresponding request.

ppbyRecv: see *SyncSendService()*.

pulRecvSize: see *SyncSendService()*.

plServiceResult: Pointer to a variable of type long, in which the PLCHandler returns the result code of the sent service. See list of return values of *SyncSendService()* for all possible values.

Return value: If no error occurs, the function returns `RESULT_OK`. Otherwise, it returns ...

`RESULT_INVALID_PARAMETER`: Invalid function parameter (e. g. NULL).

`RESULT_NO_OBJECT`: No element in the service send queue. Call *AsyncSendService()* first to send a service.

`RESULT_NO_UPDATE`: There is no finished service in the queue. Call *AsyncGetServiceReply()* later again.

3.14.4 GetDeviceSessionId

```
unsigned long ::GetDeviceSessionId(void)
```

Only for connections to V3 PLCs.

For building the V3 runtime services, the *SessionId* of the current online session is required. This Id is returned from the PLC during the login. This function allows the application to get this id from the PLCHandler.

Obsolete function, use *GetDeviceInfo()* instead, because typically you need also the communication buffer size and the byte order of the device. All three information can be retrieved by *GetDeviceInfo()*.

Return value: SessionId or 0, if not available.

4 Simplified configuration with the CEasyPLCHandler class

The PLCHandler contains a derived class *CEasyPLCHandler*. This class encapsulates the most important and popular communication channels and applies the appropriate correct settings.

4.1 **Connect...()** methods

The following methods can be called instead of the *Connect()* method directly after instantiating the PLCHandler class without the need of an additional configuration. Internally this method call *SetConfig()* and *Connect()*.

4.1.1 **ConnectTcpipViaGateway**

```
long ::ConnectTcpipViaGateway(char *pszGatewayIP, char *pszPlcIP,
char *pszProtocol = PLCC_DN_TCPIP_L2ROUTE, int bMotorola = 0,
int bLoadSymbols = 1, unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT,
unsigned long ulPort = 1200)
```

This method can be used to set up a connection to a PLC via Tcp/Ip and the Gateway on a Windows system. For this purpose only the IP address of the gateway as well as the IP address and the port of the PLC must be passed on.

pszGatewayIP: String with the IP address of the Gateway.

pszPlcIP: String with the IP address of the PLC.

pszProtocol: String with the name of the protocol (e. g. *Tcp/Ip (Level 2 Route)*).

bMotorola: Byte order of the PLC (FALSE=Intel, TRUE=Motorola).

bLoadSymbols: This parameter can be used to specify whether at a connection establishment the symbols of the PLC should be loaded or not. If this parameter is set to FALSE, the connection establishment will also succeed, if there are no symbols available. In this case only services of the PLCHandler can be used, which do not access variables or the symbolic variable information. This configuration is typically set, if the PLCHandler is used for a service application, which supports for example file transfer and start/stop features.

ulTimeout: This value specifies how long the connection establishment maximally can take, before the function returns.
If this value is set to PLCHANDLER_USE_DEFAULT (=0), the value set actually in the (default) configuration will be used.
If *ulTimeout* is set to PLCHANDLER_TIMEOUT_INFINITE (-1) the method will do one connect try and return independently of the result. In this case the PLCHandler does not start the internal reconnect thread, even, if the connect was not successful.

ulPort: Tcp/Ip port number of the PLC.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.2 **ConnectRs232ViaGateway**

```
long ::ConnectRs232ViaGateway(char *pszGatewayIP, short sPort,
unsigned long ulBaudrate, int bMotorola = 0, int bLoadSymbols = 1,
unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to the PLC via the Gateway on a Windows system and the serial interface.

pszGatewayIP: String with the IP address of the Gateway.

sPort: Serial port to be used (sPort = 1 corresponds to COM1, etc.).

ulBaudrate: Baud rate of the serial port (e. g. 115200).

bMotorola: Byte order of the PLC (FALSE=Intel, TRUE=Motorola).

bLoadSymbols: see above.

ulTimeout: see above.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.3 ConnectRs232ViaGatewayEx

```
long ::ConnectRs232ViaGatewayEx(char *pszGatewayIP, short sPort,  
    unsigned long ulBaudrate, int bMotorola = 0, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT,  
    EXT_RS232_PARAMStyp *pExtParams = NULL)
```

This method can be used to set up a connection to the PLC via the Gateway on a Windows system and the serial interface. In opposite to *ConnectRs232ViaGateway()* this method allows additionally to set some specific parameters of the serial port.

pszGatewayIP: String with the IP address of the Gateway.

sPort: Serial port to be used (*sPort* = 1 corresponds to COM1, etc.).

ulBaudrate: Baud rate of the serial port (e. g. 115200).

bMotorola: Byte order of the PLC (FALSE=Intel, TRUE=Motorola).

bLoadSymbols: see above.

ulTimeout: see above.

pExtParams: Pointer to an *EXT_RS232_PARAMStyp* structure, which contains some additional parameters for the serial port.

Return value: See return values of *SetConfig()* and *Connect()*.

The *EXT_RS232_PARAMStyp* structure is defined as:

```
typedef struct tagEXT_RS232_PARAMS  
{  
    unsigned long    ulSize;           Size of this structure = sizeof(EXT_RS232_PARAMStyp)  
    char             *pszParity;       Parity: "No" or "Yes"  
    int              nStopBits;       Number of Stop bits  
}  
EXT_RS232_PARAMStyp;
```

4.1.4 ConnectTcpiViaArti

```
long ::ConnectTcpiViaArti(char *pszPlcIP,  
    char *pszProtocol = PLCC_DN_TCPIP_L2ROUTE, int bMotorola = 0,  
    int bLoadSymbols = 1, unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT,  
    unsigned long ulPort = 1200)
```

This method can be used to set up a connection to a PLC via TCPIP and ARTI. For this purpose only the IP address and port number of the PLC must be passed on.

All parameters: See *ConnectTcpiViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.5 ConnectRs232ViaArti

```
long ::ConnectRs232ViaGateway(short sPort, unsigned long ulBaudrate,  
    int bMotorola = 0, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to a PLC via ARTI and the serial interface.

All parameters: See *ConnectRs232ViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.6 ConnectToSimulation

```
long ::ConnectToSimulation(char *pszSdbFile, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to the simulation with a SDB file.

pszSdbFile: String with the name of the SDB file; also a complete path can be specified.

All other parameters: See *ConnectTcpipViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.7 ConnectViaGateway3

```
long ::ConnectViaGateway3(char *pszGatewayIP, char *pszAddress,  
    int bLoadSymbols = 1, unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to a PLC via GATEWAY3. For this purpose only the IP address of the gateway as well as the logical address or name of the PLC must be passed on.

pszAddress: String with the logical CoDeSys address (e. g. 015B, 0001.0023) or name of the PLC.

All other parameters: See *ConnectTcpipViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.8 ConnectViaGateway3Ex

```
long ::ConnectViaGateway3Ex(char *pszGatewayIP, unsigned long ulPort,  
    char *pszAddress, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to a PLC via GATEWAY3. In extension to *ConnectViaGateway3()*, the TcpIp port number can be specified.

ulPort: TcpIp port number of the Gateway, which should be used for the connection.

All other parameters: See *ConnectViaGateway3 ()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.9 ConnectViaArti3

```
long ::ConnectViaArti3(char *pszAddress, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to a PLC via ARTI3.

pszAddress: String with the logical CoDeSys address (e. g. 015B, 0001.0023) or name of the PLC.

All other parameters: See *ConnectTcpipViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

4.1.10 ConnectToSimulation3

```
long ::ConnectToSimulation3(char *pszSdb3XmlFile, int bLoadSymbols = 1,  
    unsigned long ulTimeout = PLCHANDLER_USE_DEFAULT)
```

This method can be used to set up a connection to the simulation with a CoDeSys V3 symbol XML file.

pszSdb3XmlFile: String with the name of the CoDeSys V3 symbol XML file; also a complete path can be specified.

All other parameters: See *ConnectTcpipViaGateway()*.

Return value: See return values of *SetConfig()* and *Connect()*.

5 Configuration parameters

This chapter describes the possible settings of the PLCHandler and the structure of the ini-file. All of the described parameters can also be set by the PLCHandler's configuration structures. These structures are documented in brief in chapter 3.1. With this reference you will find for each possible setting below the corresponding element of the configuration structures.

5.1 Structure and parameters of the Ini-file

The PLCHandler SDK contains several configuration file examples for the different interfaces. Furthermore the CoDeSys OPC Configurator can be used to generate ini-files, which are applicable for the CoDeSys OPC Server and also for the PLCHandler.

Next table describes the entries, which are relevant for the PLCHandler. All other settings are not evaluated by the PLCHandler.

[Server]	PLCHandler-Section
PLCs=2	Number of configured PLCHandler's (for several instances/connections)
PLC0=My_V3_PLC	Name of the first PLC; here e. g. My_V3_PLC
PLC1=My_V2_PLC	Name of the second PLC; here e. g. My_V2_PLC
...	
[PLC: My_V3_PLC]	This section is assigned to a particular PLCHandler instance. The assignment is done by specifying the index in the constructor of the PLCHandler object. This index corresponds to the index in the above defined list of instances (PLC<n>=...).
Interfacetype=GATEWAY3	Interface type: ARTI, GATEWAY, SIMULATION, ARTI3, GATEWAY3, SIMULATION3
active=1	Flag, defines whether the PLC connection is active or not. Must be 1, if the PLCHandler should establish a connection.
Logevents=1	Activating/Deactivating the output of events into the log file.
Logfilter=16#000000FF	Setting the log filter of the PLCHandler. Does only apply, if the logging is configured and logevents=1. The typical values are: 16#0000000F: log infos, warnings, errors and exceptions 16#000000FF: above + low level communication and some debug information 16#FFFFFFFF: log all messages
isualiz=0	Only for V2 PLCs: Flag, defines the byte order of the PLC. Depending on that flag, services to/from the PLC are swapped or not. V3 PLCs: Not evaluated, byte order is read from the target.
Nologin=0	Only for V2 PLCs: Flag, defines whether the PLCHandler should not log in on the PLC; some runtime systems only allow a login of one application! V3 PLCs: Not evaluated, the PLCHandler needs always a login. <i>Special: V2: ELAU, ABB -> nologin = 1</i>

[Server]	PLCHandler-Section
precheckidentity=0	<p>Flag, defines whether the symbol file should be checked for up-to-dateness by a separate runtime system service before each reading/writing of variables. This is a fallback to support also some older runtime system versions (< V2.3).</p> <p>Only for V2 PLCs: On runtime systems \geq V2.3 at each read or write service it will be checked automatically, whether the symbols are still up to date. For this reason this entry is should be 0.</p> <p>V3 PLCs: Not evaluated, the PLCHandler checks always the integrity of the symbols automatically.</p>
Max4version=0	<p>Only for V2 PLCs: Must be used only together with a ELAU-Max4 controller and in this case set to the corresponding hardware version (1100, 1200). For all other controllers this entry is not applicable!</p> <p>V3 PLCs: Not evaluated at all.</p>
Buffersize=0	<p>Only for V2 PLCs: Size of the communication buffer; must match with the used runtime system. 0 = Size is retrieved form the runtime system or if not possible default value for the currently used communication protocol.</p> <p>V3 PLCs: Max. buffer size, which is supported by the PLCHandler, 0 means the value is set to the default of 512 KB. The effective used buffer size is negotiated with the runtime system (smaller value wins).</p> <p><i>Special: V2: ELAU -> buffersize = 1500</i></p>
project=ProjectName	<p>Interfaces SIMULATION and SIMULATION3: Used to specify the name and location of the symbol file (V2: *.sdb, V3: *.xml).</p> <p>Interface GATEWAY: If all of the following conditions are true, then the name of the project must be set:</p> <ul style="list-style-type: none"> - Symbol file can not be stored on the PLC - PLC does not know the name of the currently running project - Symbol file is manually copied to the Gateway and not by a download or online change <p>Interface ARTI: This entry can be used to load the SDB file locally by the underlying ARTI, if the SDB file cannot be stored on the PLC.</p> <p>All other interfaces: Not evaluated.</p>
Timeout=10000	<p>Time in ms, defines how long the answer on a data package from the PLC might take, before an error will be dumped.</p> <p>Interfaces ARTI3 and GATEWAY3: Timeout handling is done by lower layers. Should be set to 10000 to adapt the overall timeout to the internal behaviour.</p>
Tries=3	<p>Number of communication trials in case the receiving of data fails.</p> <p>Interfaces ARTI3 and GATEWAY3: Timeout handling is done by lower layers. Should be set to 3 to adapt the overall timeout to the internal behaviour.</p>

[Server]	PLCHandler-Section
Waittime=20	Total time in seconds in <code>::Connect()</code> for the connection establishment. As soon as the time is exceeded, the <code>::Connect()</code> returns with an error, but the <code>Reconnect-Thread()</code> still tries to connect.
Reconnecttime=20	Time slice in seconds according to which a reconnect is tried by the <code>Reconnect-Thread()</code> . If set to <code>PLCHANDLER_TIMEOUT_INFINITE (-1)</code> the internal <code>Reconnect-Thread()</code> is disabled at all and the client is responsible to handle the initial connect and also the reconnect attempts after download, online change or lost connection.
Gateway=Tcp/Ip	Only interfaces GATEWAY and GATEWAY3: Used protocol/connection between the PLCHandler and the Gateway. Possible values: GATEWAY: "Tcp/Ip" and "Local" GATEWAY3: "Tcp/Ip" and "SharedMemory" Typically "Tcp/Ip" is used, because this support both, local and remote gateway connections. All other interfaces: Not evaluated.
Gatewayaddress=localhost	Only interfaces GATEWAY and GATEWAY3, if "Tcp/Ip" is used as Gateway protocol: IP-Address or Hostname of the Gateway. All other configurations: Not evaluated.
Gatewayport=1217	Only interfaces GATEWAY and GATEWAY3, if "Tcp/Ip" is used as Gateway protocol: Port number of the Gateway. Typical values: GATEWAY: 1210 GATEWAY3: 1217 All other configurations: Not evaluated.
Gatewaypassword=MyPassword	Only interface GATEWAY: If the Gateway, which is used for the connection, protected by a Gateway password, than this can be entered here, to allow the PLCHandler to use this Gateway. It is recommended to use the configuration structures in such cases, to avoid storing of the plain password in a configuration file. All other interfaces: Not evaluated.
Instance=PLCWinNT_TCPIP	Only interface GATEWAY: Here you can assign an optional name to the package of configuration settings, typically not used. All other interfaces: Not evaluated.
Device=Tcp/Ip (Level 2 Route)	Only interfaces ARTI and GATEWAY: Communication protocol to the PLC; the most important settings are: Tcp/Ip (Level 4): TCP/IP Level 4 Protocol Tcp/Ip (Level 2): TCP/IP Level 2 Protocol Tcp/Ip (Level 2 Route): TCP/IP Level 2 Route Serial (RS232): Serial connection All other interfaces: Not evaluated. <i>Special: V2: ELAU -> device = Tcp/IP (Level 4)</i>

[Server]	PLCHandler-Section
parameters=2	Number of parameters in the generic format. All further parameters are described in a generic format. This allows a flexible use and extension of the parameters. For each parameter there are two lines in the configuration file to set the name and the value of the parameter: parameter<#>=<name of the parameter> value<#>=<value of the parameter>
parameter0=Address	Name of the first parameter. Example "Address": Logical address or node name of the PLC (V3)
value0=0001.01AB	Value of the first parameter. Example "0001.01AB": Logical Address of the PLC (V3)
parameter1=WriteThroughReadCache	Name of the second parameter. Example "WriteThroughReadCache": Specifies, if <i>SyncWriteVarsToPlc()</i> , should additional update the cache of cyclic read lists or not.
Value1=1	Value of the second parameter. Example "1": Enabled
...	
[PLC: My_V2_PLC]	This is the second instance to be configured
interfacetype=ARTI	Interface type: ARTI, GATEWAY, SIMULATION, ARTI3, GATEWAY3, SIMULATION3
...	...

The PLCHandler's generic parameter format

parameter<#>=<name of the parameter>
value<#>=<value of the parameter>

allows a flexible use and extension of the parameters. Currently this is used for two types of parameters. Both types can be freely combined as you can see in the table above.

5.1.1 Parameters to describe the connection to the PLC

5.1.1.1 Parameters for a V2 PLC connection

Especially in V2 environments, the type and number of the parameters depend on the kind of the used protocol. Therefore all PLC connection parameters are passed in the generic format. The following two tables show the parameters for the *Tcp/Ip (Level 2 Route)* protocol and the *Serial(RS232)* protocol as example for V2 protocols:

Tcp/Ip (Level 2 Route):

Parameter name	Description
Address	Tcp/Ip address of the PLC
Port	Tcp/Ip port number of the PLC
TargetId	For Tcp/Ip (Level 2 Route): ID of the runtime system, to which the data should be routed; typically set to 0.
Motorola byteorder	Byte order of the PLC, possible values: No, Yes.

Serial(RS232):

Parameter name	Description
Port	Name of the Comport, e. g. COM1.
Baudrate	Baud rate of the serial port, e. g. 115200
Parity	Parity of the serial port, possible values: No, Even, Odd.
Stop bits	Number of stop bits of the serial port, possible values: 1, 2.
Motorola Byteorder	Byte order of the PLC, possible values: No, Yes.
Flow Control	Flow Control of the serial line, possible values: Off, On.

5.1.1.2 Parameters for a V3 PLC connection

To address V3 PLCs, there is almost only one parameter needed, the logical PLC address or the PLC name.

Parameter name	Description
Address	Logical CoDeSys address (e. g. 015B, 0001.0023) or node name of the PLC. Depending on the format the PLCHandler recognizes automatically, if this is an address or a node name.

Starting from version V3.4 SP2 (PLCHandler and runtime system) there is also the possibility to use the CmpBlkDrvTcp to communicate to the PLC. To configure the PLC connection using this driver, the following parameters are used instead of the logical address or name.

Parameter name	Description
IpAddress	Tcp/ip address of the PLC
Port	Tcp/ip port number of the PLC. If not specified, the default value 11740 is used.

5.1.2 Special options for the PLCHandler

Some special options for the PLCHandler or options, which can only be used with one communication interface have also to be set in the PLCHandler's generic configuration format. The following table shows a list of the available options:

Parameter name	Description
Ping	Only interface ARTI: Enable/disable the ICMP ping to target before connect (1=enabled [default], 0=disabled). This should only be disabled, if the network stack of the client or the PLC does not support the ICMP protocol. All other interfaces: Not evaluated.
SymbolFile	Only interface ARTI: Name of the symbol file if it's not "Download.sdb" or on ELAU targets "Default.sdb". Only to be used in very rare cases. All other interfaces: Not evaluated.
SymbolFilePath	Only interface ARTI: Path for the local symbol file cache. Typically the symbol file is uploaded on each (re-)connect from the PLC. If the PLC is connected over a line with bad quality and/or bandwidth, this can consume a considerable time for big symbol files. To optimize this, the interface ARTI

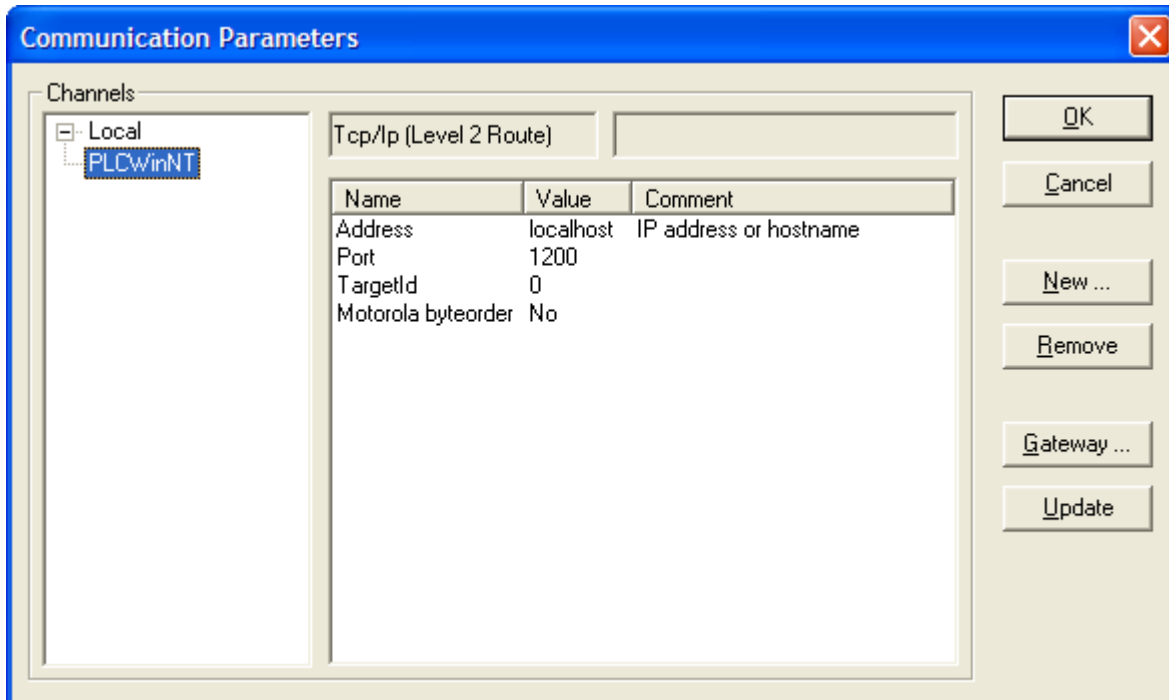
Parameter name	Description
	<p>contains a caching mechanism, which is activated by setting this path.</p> <p>All other interfaces: Not evaluated.</p>
DirectAddressesOnly	<p>Only interface ARTI: If a PLC does not support the feature of symbolic access (e. g. can not keep the symbol file) and the application, which is using the PLCHandler access only direct addresses (e. g. %MB0), then this feature can be activated, to activate this limited access (0=disabled [default], 1=enabled).</p> <p>All other interfaces: Not evaluated.</p>
User	<p>Only interfaces ARTI3 and GATEWAY3: If the user management is activated on the PLC, then the user name for the PLCHandler login can be specified here.</p> <p>If this setting is used, it is recommended to set up the PLCHandler by the configuration structures, to avoid storing of the user name and the password in a configuration file.</p> <p>All other interfaces: Not evaluated.</p>
Password	<p>Only interfaces ARTI3 and GATEWAY3: If the user management is activated on the PLC, then the password for the PLCHandler login can be specified here.</p> <p>If this setting is used, it is recommended to set up the PLCHandler by the configuration structures, to avoid storing of the user name and the password in a configuration file.</p> <p>All other interfaces: Not evaluated.</p>
CheckDataChangeOnPlc	<p>Only interfaces ARTI3 and GATEWAY3: If this option is activated (0=disabled [default], 1=enabled), then the change of values is evaluated on the PLC and the PLC transmits only the changed values to the PLCHandler. Please take in account, that this feature consumes some extra resources on the PLC. Furthermore this can even lead to more communication load, if most of the read values changes in every read cycle. So this feature is typically be used in closed systems with well known application data.</p> <p>Only supported by CoDeSys runtime versions >= V3.4 SP1.</p> <p>The combination of the options CheckDataChangeOnPlc and WriteThroughReadCache is only supported by CoDeSys runtime versions >= V3.5.</p> <p>All other interfaces: Not evaluated.</p>
ClientAddressResolution	<p>Only interfaces ARTI3 and GATEWAY3: By default in V3 during the definition of a variable list, the name of a variable is sent to the PLC and the PLC returns a handle for this variable (list). If a PLC is connected over a line with very limited bandwidth or if a low-end PLC consumes a lot of CPU time for resolving the symbolic name into a address, then the PLCHandler can take over this part (0=disabled [default], 1=enabled).</p> <p>This should only be used in the two described scenarios above, because by activating this feature the PLCHandler consumes some percent more CPU time and memory. But more important is that the PLCHandler will not swap the read or written values, if the PLC has a different byte order as the system on which the PLCHandler runs. And last, there is no</p>

Parameter name	Description
	<p>support for all features of complex variable access (property access, referencing pointers, references inside structures, etc.).</p> <p>All other interfaces: Not evaluated.</p>
DontExpandSimpleTypeArrays	<p>Only for V3 interfaces: For this interfaces the PLCHandler loads all type descriptions for all exported variables during connect. After that the PLCHandler expands all complex variables (arrays and structures) to present all elements to the application (<i>GetAllItems()</i>). For doing this the PLCHandler consumes some time and memory. For intelligent client applications, which provide itself the feature of expanding simple type variables, it is not necessary to do dies additional in the PLCHandler. Therefore the expansion can be suppressed by this setting (0=expand simple type arrays [default], 1=don't expand simple type arrays).</p> <p>All V2 interfaces: Not evaluated.</p>
DontExpandComplexTypeArrays	<p>Only for V3 interfaces: This setting can be used to suppress the expansion of arrays of complex base types (structures). The option DontExpandComplexTypeArrays can typically only be applied in closed systems, because if a complete structure or an array of structures is accessed by a client, he has to know the exact alignment of the structure members. Closed system in this context means, that the IEC-Application, the PLC and the PLCHandler client is provided from the same vendor. This rare clients may set this option to 1 (0=expand complex type arrays [default], 1=don't expand complex type arrays).</p> <p>If both options DontExpandComplexTypeArrays and DontExpandSimpleTypeArrays are used, <i>GetAllItems()</i> returns only the top level nodes of GVLs or POUs.</p> <p>All V2 interfaces: Not evaluated.</p>
WriteThroughReadCache	<p>All interfaces: By default all calls of <i>SyncWriteVarsToPlc()</i> do not affect the values of cyclic read lists. If a value is written from the visualization, which uses the PLCHandler, and also from the PLC, this can result in the following effects: Depending on the client there can be after a write event a short flicker with an old value from the PLC in the visualization or seemingly missing data change callbacks, if the PLCHandler client have an extra data cache with another policy.</p> <p>In principle this should be avoided by writing a variable only from one side (PLC or PLCHandler). If this can not be guaranteed, then the PLCHandler can be adapted to the application's policy. By enabling this feature (0=disabled [default], 1=enabled), the PLCHandler checks on each write call, if the variables to write to are also part of defined cyclic variable lists. If this is the case, the cache value for the affected variables inside all lists is set to the written value and the data change callback is called for each list which contains now a modified value.</p> <p>Please note, that if a callback is registered, this will be called more often and not only from the context of the update thread of the corresponding cyclic variable list. Furthermore this feature consumes during each call of <i>SyncWriteVarsToPlc()</i> some extra CPU time, which depends strongly on the</p>

Parameter name	Description
	number of variables to write to, the number of variables per cyclic variable list and the number of cyclic variable lists.
DontLoadSymbolsFromPlc	<p>Only interfaces ARTI3 and GATEWAY3: Clients which have not the need of browsing the symbols, can suppress reading the symbol information from the PLC to speed up the connect time. If the client knows all the symbol names and types, this option can be set to 1 (0=load symbol information from PLC [default], 1= don't load symbol information from PLC).</p> <p>All other interfaces: Not evaluated.</p>

5.2 How to get a typical V2 PLC configuration

In CoDeSys V2.3 the communication parameters for the PLC are set up with the *Communication Parameters* dialog (see *Online* menu), for which an example is shown below.



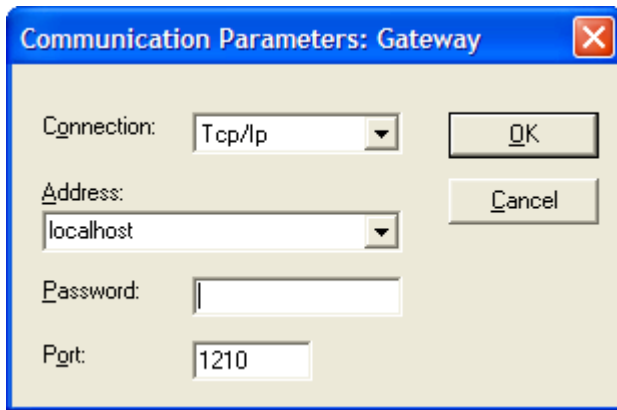
The first step is to configure the communication parameters in CoDeSys and login successfully into the PLC. Now you can be sure, that you have a good template to set up the PLC configuration parameters for the PLCHandler. For the interfaces ARTI and GATEWAY all of the following settings can be read out from this dialog:

```
device=Tcp/Ip (Level 2 Route)
parameters=4
parameter0=Address
value0=localhost
parameter1=Port
value1=1200
parameter2=TargetId
value2=0
parameter3=Motorola byteorder
value3=No
```

You have to make sure, that all parameter names are exactly spelled in the same way as in the CoDeSys *Communication Settings* dialog, because all parameters are referenced by the name.

Please note, that the interface ARTI supports only a subset of the protocols (all 3S Tcp/Ip protocols and the 3S RS232 protocol).

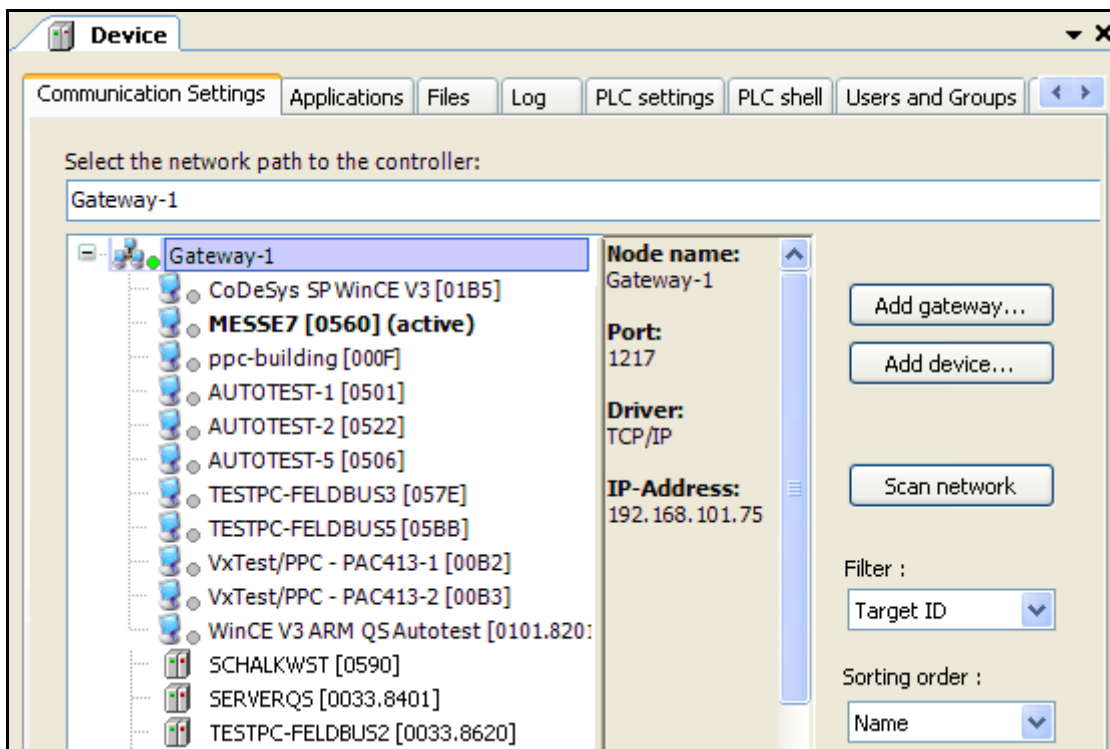
Furthermore in case of the interface GATEWAY, you can read also the needed parameters for configuring the communication path to the Gateway from the *Communication Parameters: Gateway* sub dialog, if you press the *Gateway...* button.



gateway=Tcp/Ip
 gatewayaddress=localhost
 gatewayport=1210
 gatewaypassword= (only if set)

5.3 How to get a typical V3 PLC configuration

In CoDeSys V3 the communication path to the PLC is set in the *Communication Settings* dialog of the Device. Usually first a network scan is triggered and after that the target device is selected by pressing the button *Set active path*.



The logical CoDeSys addresses (e.g. 0560) are unique in the reachable network and contain all needed information to address the node. Furthermore each node has a node name (e.g. MESSE7). This node name can also be used to specify the PLC to which the PLCHandler should connect to, if it is non-ambiguous in the network. The connection buildup will fail, if there exist two or more PLCs with the specified name.

If the node names are unique in the PLC network, then typically the PLC connection should be parameterized by the PLC name, in order to ignore changes of the logical address, which can result from modifications of the PLC network.

Depending on this the screenshot above can lead to two possible configurations of the PLC communication settings for the interfaces ARTI3 and GATEWAY3:

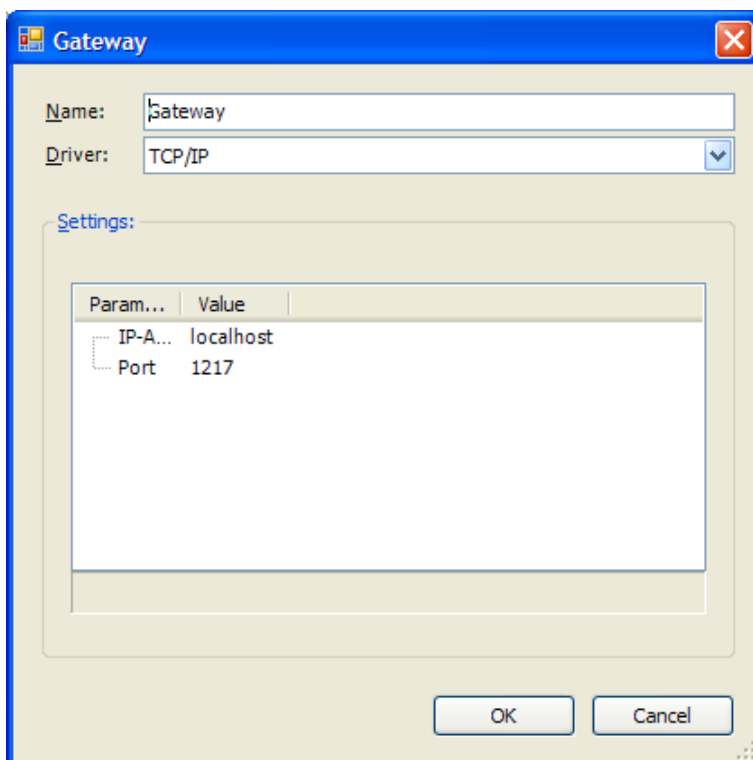
1. Using the logical address of the PLC

```
parameters=1  
parameter0=Address  
value0=0560
```

2. Using the PLC's node name

```
parameters=1  
parameter0=Address  
value0=MESSE7
```

Analogue to CoDeSys V2.3 also in CoDeSys V3 the communication path to the Gateway can be read out from CoDeSys by using the *Edit...* command, after the focus has been set to the Gateway node at top of the network scan result:



With the help of this screenshot the gateway connection of the interface GATEWAY3 can be configured as follows:

```
gateway=Tcp/Ip  
gatewayaddress=localhost  
gatewayport=1217
```

Appendix A: Error codes

Define	Value	Description
RESULT_FAILED	-1	Action erroneous.
RESULT_OK	0	Action successful
RESULT_PLC_NOT_CONNECTED	1	PLC not connected
RESULT_PLC_LOGIN_FAILED	2	Login to PLC has failed
RESULT_PLC_NO_CYCLIC_LIST_DEFINED	3	No cyclic list has been found
RESULT_PLCHANDLER_INACTIVE	4	PLCHandler instance was set inactive
RESULT_LOADING_SYMBOLS_FAILED	5	Loading of the symbols has failed
RESULT_ITF_NOT_SUPPORTED	6	The defined communication interface is not valid or not supported
RESULT_COMM_FATAL	7	Communication error occurred during action
RESULT_NO_CONFIGURATION	8	Wrong or erroneous configuration of the PLCHandler
RESULT_INVALID_PARAMETER	9	At least one parameter is invalid
RESULT_ITF_FAILED	10	Communication interface could not be initialized correctly (e. g. Gateway DLLs not available)
RESULT_NOT_SUPPORTED	11	Method not yet supported resp. implemented for this configuration (e. g. not supported for this interface)
RESULT_EXCEPTION	12	Handled exception in a low layer occurred during action
RESULT_TIMEOUT	13	Timeout exceeded
RESULT_STILL_CONNECTED	14	PLC already connected (at a further ::Connect() call)
RESULT_RECONNECTTHREAD_STILL_ACTIVE	15	Reconnect Thread already active (started at a further ::Connect() call)
RESULT_PLC_NOT_CONNECTED_SYMBOLS_LOADED	16	No connection to the PLC, but symbols available offline
RESULT_NO_UPDATE	17	Asynchronous operation (e. g. cyclic read of variables) has not yet finished

Define	Value	Description
RESULT_OCX_CONVERSION_FAILED	18	Error during conversion of values inside the PLCHandler's ActiveX interface occurred
RESULT_TARGETID_MISMATCH	19	PLC does not match to the passed target id etc.
RESULT_NO_OBJECT	20	No object found for the required action (e. g. tried to get an element beyond the end of the list)
RESULT_COMPONENTS_NOT_LOADED	21	PLCHandler instantiation has failed, because of missing components
RESULT_BUSY	22	Last action still in progress, cannot start the required one
RESULT_DISABLED	23	Feature is disabled by the configuration (e. g. Logging)
RESULT_PLC_FAILED	24	Communication to the PLC was successful, but the PLC has returned a bad result

Appendix B: C-Interface of the PLCHandler class

The PLCHandler library provides also a C-Interface for the PLCHandler. The C-Interface is declared in the file PLCHandlerIrf.h (see include folder of the SDK).

All functions of the C-Interface have the prefix *PLCHandler*. For most methods of the PLCHandler class there is an corresponding function with the same name as part of the C-Interface of the PLCHandler. Example: The method `::GetVersion()` of the PLCHandler class is encapsulated by *PLCHandlerGetVersion()* in the C-Interface.

Instead of the constructors of the PLCHandler class, the C-Interface contains several init functions (e. g. *PLCHandlerInitByFile()*), which allow to instantiate an internal PLCHandler object. All this functions return an unique number, which represents the instance. This instance number must be passed as first parameter *ulPLCHandler* to all other C-Interface functions to identify the instance to work with.

Also the destructor is replaced by a function *PLCHandlerExit()*.

Appendix C: ActiveX-Control of the PLCHandler (Windows only)

The PLCHandler is also provided as ActiveX Control.

Before first use the ActiveX-Control must be registered with the following command:

```
regsvr32 plchandlerx.ocx
```

Then the PLCHandlerX class can be used in a Visual-Basic, Delphi or C++ project.

The following methods deviate from the methods in the PLCHandler:

Connection establishment:

Method: long ::ConnectByString(long lld, LPCTSTR pszConfigString, long lConfigLen, long StateChangedEvent, LPCTSTR pszLogFile);

Parameter ulld specifies the index of the PLCHandler, if several instances are used.

Parameter pszConfigString points to a string containing the content of the ini-file of the configuration.

lConfigLen specifies the length of the configuration string.

StateChangedEvent: Here optionally the handle of an event can be processed, via which the client can be informed at each state change.

pszLogFile is the name of a log file, where the PLCHandler optionally can record all important actions.

Method: long ::ConnectViaGateway(LPCTSTR pszGatewayIP, LPCTSTR pszPlcIP, LPCTSTR pszProtocol, LPCTSTR pszLogFile);

This method can be used to establish a connection via TCPIP and the Gateway to a PLC under Windows. For this purpose only the IP-address of the Gateway as well as the IP-address of the PLC must be passed. Default setting is the Tcp/Ip L2 Route Protocol („Tcp/Ip (Level 2 Route)“).

Method: long ::ConnectViaArti(LPCTSTR pszPlcIP, LPCTSTR pszProtocol, LPCTSTR pszLogFile);

This method can be used to establish a connection via TCPIP and ARTI to a PLC. For this purpose only the IP-address of the PLC must be passed. Default setting is Tcp/Ip L2 Route Protocol.

Method: long ::ConnectViaSimulation(LPCTSTR pszSdbFile, LPCTSTR pszLogFile);

Via this method the connection to the simulation mode and to a SDB-file is established. Just the name of the SDB-file must be specified. Thereby also a complete path can be defined.

Browsing of variables:

Method: long ::GetNumberOfSymbols();

Returns the number of available variables in the PLC.

Method: long ::GetSymbol(long lIndex, LPCTSTR pszSymbol, long lMaxLen);

GetNumberOfSymbols() can be used to get the number of available variables.

GetSymbol() can be used to get the name of the variable which is specified by the index.

Synchronous Reading of variables:

Method: long ::SyncReadVarFromPlc(LPCTSTR pszSymbol, long lAddressData, long lSize);

lAddressData specifies the address, where the value of the variables is stored.

pszSymbol specifies the name of one variable.

Method: long :: SyncReadVarsFromPlc(LPCTSTR pszSymbols, long lAddressDataList, long lAddressSizeList, long lNumOfVars);

lAddressData specifies the address, where the value of the variables is stored.

pszSymbol specifies the names of several variables. *lNumOfVars* defines the number of variables.

Synchronous Writing of variable values:

Method: long SyncWriteVarToPlc(LPCTSTR pszSymbol, long IAddressData, long ISize);

IAddressData specifies the address where the value of the variable to be written is stored.

pszSymbol specifies the name of one variable.

Method: long SyncWriteVarsToPlc(LPCTSTR pszSymbols, long IAddressDataList, long IAddressSizeList, long INumOfVars);

IAddressData specifies the address, where the value of the variables to be written is stored.

pszSymbol specifies the names of several variables. *LnumOfVars* specifies the number of variables.

Change History

Version	Description	Editor	Date
1.0	Release Version		27.02.2004
1.1	VersionTcp/Ip (Level 4), Elau specialties added; some formatting and translation corrections		15.10.2004
1.2	PLCHandler methods updated and described (ActiveX-Control,App.D), Return and status values described (new App.B,C), ActiveX-Control (App.D)		08.02.2005
1.3	Complete Update	MM	19.05.2010
1.4	Formal Review and Rework	MN	20.05.2010
1.5	New option CheckDataChangeOnPlc added	MM	31.05.2010
1.6	New method GetApplicationInfo and update of GetProjectInfo. Some chapters resorted.	MM	08.06.2010
2.0	Release after formal review and rework	MN	09.06.2010
2.1	CDS-18063: Setting to disable the Reconnect-Thread, new return value of Disconnect() (CDS-18216), description for setting the V3 buffer size corrected, description for ClientAddressResolution completed. CDS- 19453: Chapter 4.1.8 added to describe the new method ConnectViaGateway3Ex(). CDS-18782: Return values of all methods checked and if necessary adapted. CDS-19919: PLCHandler should support CmpBlkDrvTcp. CDS-3856: PLCHandler: DeviceLogin information should be available.	MM	29.11.2010
3.0	Release after formal review	MN	20.11.2010
3.1	CDS-20795: PLCHandler: Sporadic deadlock during intensive use of SyncWriteVarsToPlc, if option WriteThroughReadCache activated: Chapters 3.8.5 to 3.8.8 updated.	MM	21.12.2010
4.0	Release after formal review	MN	09.03.2011
4.1	General build hints regarding the version handling added to chapter 2. Several Subchapters for the following new features added: CDS-7306: PLCHandler: Interface Gateway3: ReloadBootproject not supported. CDS-12878: PLCHandler should have a function to reset the PLC. CDS-22953: PLCHandler: Address information must be available for mapped variables via online-service. Corrections: CDS-23819: PLCHandler, Docu: Bug in the description of the function SyncWriteVarsToPlc and SyncReadVarsFromPlc (chapters 3.9.1 and 3.9.3), CDS-23642: Docu / PLC Handler: time unit missing for cyclic update methods (chapters 3.8.1, 3.8.10 and 3.8.11).	MM	15.07.2011
5.0	Release after formal review	MN	15.07.2011
5.1	CDS-10369: PLCHandler: Add/Remove single items to a cyclic list should be possible: New chapters 3.8.16 and 3.8.17 added. CDS-17250: PLCHandler: CycReadVars: Timestamp and quality of the read variable values are not proper set by all	MM	12.12.2011

Version	Description	Editor	Date
	<p>interfaces: Description of structure PlcVarValue updated (chapter 5.1.2).</p> <p>CDS-17260: PLCHandler: Changes of the updatarate of a cyclic list should be adapted at once: Description of CycSetUpdateRate improved (chapter 3.8.10).</p> <p>CDS-19179: PLCHandler: Interfaces ARTI3 and Gateway3: PLCHandler should not load symbols, if symbol browsing is not needed: Additional return value for affected methods <i>GetAllItems()</i>, <i>GetItem()</i> and <i>GetAddressOfMappedItem()</i> added (chapters 3.7.4 to 3.7.6). Description of the setting DontLoadSymbolsFromPlc added (chapter 5.1.2)</p> <p>CDS-22914: PLCHandler: Provide a solution to prevent deadlocks, if the DataChange or UpdateReady callback implementation of the PLCHandler application needs to enter a global semaphore: New chapters 3.13.6 and 3.13.7 inserted.</p> <p>CDS-23119: PLCHandler: Interfaces ARTI3 and Gateway3: Missing DataChangeCallbacks, if both options "WriteThroughReadCache" and "CheckDataChangeOnPlc" are used: Additional comments added to the desription of this options (chapters 3.8.7 and 3.9.1).</p> <p>CDS-23874: PLCHandler: Expanding arrays of complex types should be optionally deactivated: Description of the setting DontExpandComplexTypeArrays added (chapter 5.1.2)</p>		
6.0	Release after formal review	MN	12.12.2011