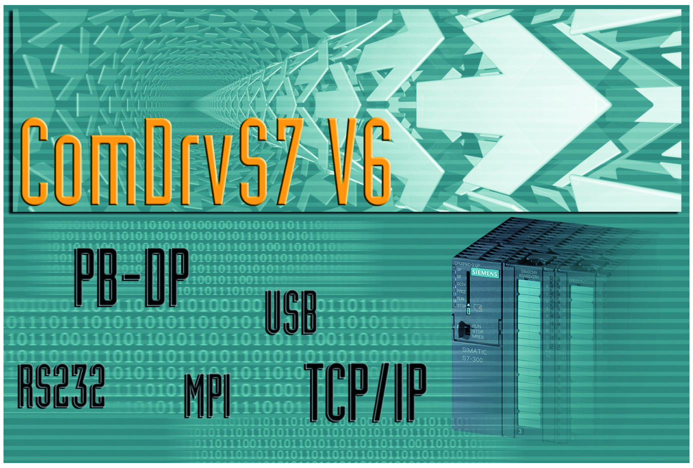


ComDrvS7 V6.2X

for LOGO!®, S7-1500®, S7-1200®, S7-300®, S7-400®,
VIPA 100V/200V/300S/SLIO

Documentation

Developers Guide

July 2015

With examples and hints to
Visual C++ , Visual-Basic, Visual-C#, .Net, Delphi and C++ Builder

MHJ-Software GmbH & Co. KG
www.mhj.de

1 General information about ComDrvS7

Brief description

The ComDrvS7 DLL enables you to transfer data to and from Siemens S7 series 1200/1500/300/400 PLCs and S7-compatible CPUs of the series VIPA 100V/200V/300V/300S. You can establish a communication link using MPI, Profibus-DP (with NetLink, NETLink PRO or SIMATIC®-NET) or TCP/IP. ComDrvS7 also supports the data-transfer to LOGO!® from 0BA7 or higher.

As of version 6.25 the PLC-family S7-1500® from Siemens is supported.
One of the functions "MPI6_OpenTcpIp_S71500" or "MPI6_OpenTcpIp_S71500Ext" is required to communicate with a PLC-1500.
Please read the chapter "Required settings in a PLC 1500 from Siemens" for the important hardware settings!

As of version 6.23 a 64 bit version of ComDrvS7 is available. You can use this DLL for 64 bit applications, on a 64 bit OS. A 64 bit application is necessary, if the limits of 32 bit applications become a problem. For example files become greater then 4 GB. The disadvantage of 64 bit applications is, that they only run on a 64 bit OS., e.g. Windows 7-64bit.
A 32 bit applications can be used on a 32 bit and 64 bit OS.

As of version 6.20, you can transfer data to and from a Siemens LOGO! (0BA7) via ethernet.

The DLL can be integrated into Windows applications to communicate with the PLC. You need a PC/MPI cable (RS232, USB), the NetLink or NETLink PRO (TCP/IP) to communicate. You can also communicate via an Ethernet-CP or an Ethernet interface that is integrated into the CPU. In this case, the units communicate via a standard Ethernet cable.

SIMATIC®-NET is supported as of version 4, provided that the drivers were installed on the PC. In this case, it is possible, for example, to address CPs 5512, 5611 and the Siemens USB-MPI adapter. In the presence of Siemens' Teleservice V6, ComDrvS7 can also acquire data via telephone lines (e.g. using the Teleservice II-Adapter).
As of version 4, a CPU can be addressed using routing. This means that the CPU to be addressed must not necessarily be connected directly to the PC, cince the request can be routed to the CPU via different bus systems.

The DLL supports the following operations:

- Read/write bytes within address ranges I, Q, M and DB (I, Q, M is not available in the Lite version). You do not need the password for a password protected CPU.
- Read/write words within address ranges I, Q, M and DB (I, Q, M is not available in the Lite version). You do not need the password for a password protected CPU.
- Read/write double words within address ranges I, Q, M and DB (I, Q, M is not available in the Lite version). You do not need the password for a password protected CPU.
- Read/write timers and counters (not supported by the Lite version).
- Reading DBs from the CPU and saving them in a WLD file (not supported by the Lite version).
- Reading DBs from a WLD file and transferring them into the CPU (not supported by the Lite version).
- Copy RAM to ROM to save current DB values.
- Determine and change the operating mode of the CPU (RUN, STOP change).
- Read and modify the time of day from the CPU.
- Read the number of DBs that are available in the PLC
- Determine the DB numbers that are available in the PLC
- Determine the length of DBs in bytes
- Read the serial number of the CPU and of an existing MMC
- Read the status of error-LEDs SF, BF1 and BF2 of a CPU
- Query the need for a password to access a CPU
- Pass a password to the CPU to provide unlimited access to a password-protected CPU.
- Identify system areas
- Determine the position of the operating mode switch and protection levels
- Read the order number from the CPU
- Determine accessible nodes on the MPI/DP network
- Functions to convert REAL, DINT and INT operands from WORD or BYTE buffers.
- **Other protocols on request.**

A 32-bit and 64-bit version of the DLL is available.

System requirements

Platform: WinXP, Vista, Win7 (32/64 bit), Win8 (32/64 bit)

1.1 Hardware requirements for the different communication channels

1.1.1 RS232/USB communications

You need a PC/MPI cable to provide the connection between the PC and
the S7 series 300/400 PLC. You may also use a USB-MPI cable if the USB driver has created a virtual serial interface.
Order numbers of the interfacing cables:

MPI adapter RS232:	**M007.001**
MPI adapter USB:	**M007.005**

1.1.2 Communications via MHJ-Netlink or MHJLink++ (incl. routing)

In this case, **MHJ-NetLink** connects the hub/switch to the S7-PLC. You require an additional crossover cable if you want to insert the MHJ-NetLink directly into the network adapter of a PC.
Order numbers of the interface cables:
MHJ-NetLink for MPI and Profibus-DP: M007.010

1.1.3 TCP/IP-direct communications (incl. routing)

If the S7 PLC is equipped with an Ethernet-CP or if the CPU has an integrated Ethernet (or Profinet) port, then you may also establish the communication link using these resources. A conventional Ethernet cable provides the connection between the PC and the CP. The PC and the CP can also be linked by means of a hub/switch. Both, the PC and the CP must be in the same subnet.

1.1.4 TCP/IP-NETlink PRO communications (incl. routing)

In this case, **NETLink PRO** is used to connect the hub/switch to the S7-PLC. You require an additional crossover cable if you want to insert the **NETLink PRO** directly onto the network adapter of a PC.
In comparison to the standard NetLink, **NETLink PRO** has the advantage that it is optimized with respect to remote maintenance via the Internet. The communication speed of the **NETLink PRO** is higher than that of the standard NetLink.
Order number of the interface cable:
NETLink PRO (Ethernet) for MPI and Profibus-DP: M007.020

1.1.5 SIMATIC®-NET communications (inkl. Routing)

TheSIMATIC®-NET drivers must have been installed on the PC. Under these circumstances the Siemens interface adapter (e.g. CP5512, CP5611 USB-MPI adapter) may be used. When the Teleservice feature V6 or higher is available, it is also possible to operate remotely via phone line. Here you can make use of the Teleservice Adapter II or compatibles (see www.mhj.de). You can also order the Teleservice software using www.mhj.de.
This communication path is only available with the 32-bit DLL.

1.2 ComDrvS7 installation

The driver is supplied on CD-ROM or you may use a download link.

You can download the current version from www.mhj.de at any time.

After installation, all DLLs are available on the hard disk. These differ in the programming languages into which they may be integrated. Please make absolutely sure that you use the DLL that is supplied for your programming language.
A DLL that has not been activated automatically reverts to a demo version.

The **demo version** displays a demo message with every open-function. This message requires confirmation. Subsequently, the demo message will appear at certain intervals or when you execute the open-function.

Please read the file README.TXT located on the CD for additional information.

1.3 Changes with respect to older versions of the ComDrvS7 DLL

1.3.1 From V6.25: implementation of the S7-1500® family

From version 6.25, the read and write functions may be applied to the S7-1500® operands of the type E, A, M and DB. For this purpose, the special initiation function "MPI6_OpenTcpIp_S71500" or "MPI6_OpenTcpIp_S71500Ext" must be executed.
The different functions are associated with explicit details to indicate which function may be used for the S7-1500®.

1.3.2 From V6.23: A 64-bit DLL is available

1.3.3 From V6.2: implementation of the LOGO!® family

From version 6.2, the read and write functions may be applied to the LOGO!® (0BA7) operands of the type I, Q, M and VM. For this purpose, the special initiation function "MPI6_OpenTcpIp_Logo" must be executed.

The access to a LOGO!® is only possible in the MICRO-Version of ComDrvS7.

1.3.4 From V6.1; implementation of the S7-1200® family

From version 6.1, the read and write functions may be applied to the S7-1200® operands of the type E, A, M and DB. For this purpose, the special initiation function "MPI6_OpenTcpIp_S71200" must be executed.
The different functions are associated with explicit details to indicate which function may be used for the S7-1200®.

1.3.5 From V6.0: speed optimised protocols for read and write functions

As of version 6, speed optimised protocols will be used to read and write operands (E, A, M, T, Z, DB). As a result, the communication speed is increased significantly in comparison to earlier versions.

1.3.6 From V6.0: new functions MixRead and MixWrite

The new functions MixRead and MixWrite can read and write different operand types in a single call. For example, you can read data from different data blocks, clock memory and input values with a single call of the MixRead. The data requests are automatically optimized by the function so that duplicate queries, overlaps, etc. are detect automatically.

1.3.7 From V6.0: read and write functions do not require a password

From version 6, you can execute the read and write functions (byte, word and double word) without knowledge of the CPU password. This means that you can run these functions without transferring the password to the CPU.

1.3.8 From V6.0: write DBs into WLD-files

From version 6, you can read DBs from the CPU to write them to a WLD-file. This type of file can then be processed with the S7-programming systems. It is also possible to create a backup of the DBs on the PC.

1.3.9 From V6.0: read DBs from WLD-files and transfer them to the CPU

From version 6, you can read DBs from a WLD-file and transfer them to the CPU. You can create a WLD-file using S7 programming systems. This means that we have the option to store, among other things, DBs with different settings on the PC and transfer them to the CPU when required (recipe management and similar).

1.3.10 From V6.0: function to copy RAM to ROM

You can use this function to save process-related data of all the DBs located in the memory of the CPU into the load memory of the CPU. This means that the data are retained even when the CPU is re-booted.

1.3.11 From V6.0: read/write the time and date of the CPU

From version 6, you can read the time and the date from a CPU and rewrite it if necessary.

1.3.12 From V6.0: change the operating mode of the CPU

As of version 6, the operating mode of the CPU can be set to RUN or STOP mode.

1.3.13 From V5.0: reading identification data of a CPU (e.g. CPU serial number)

From version 5, the identification data may be read from a CPU. These include:

- the serial number of the CPU
- the serial number of the MMC in the CPU
- The system identification (may be specified by the user when the hardware of the CPU is being configured)
- The location identification (may be specified by the user when the hardware of the CPU is being configured)
- The name of the CPU (may be specified by the user when the hardware of the CPU is being configured)
- The station identification (may be specified by the user when the hardware of the CPU is being configured)

The data can be read by Siemens S7-300? CPUs with a firmware version 2.6. or higher. ComDrvS7 has a feature that can be used to determine whether a CPU actually provides the data.

1.3.14 From V5.0: reading the status from the fault LEDs of a CPU

From version 5, the status of the fault LEDs SF (collective error), BF1 (bus-error 1) and BF2 (bus-error 2) can be read from a CPU. This enables the PC programmer to determine whether the execution of the PLC program is affected or made impossible by such an error.
The respective error can then be displayed on the PC or the program on the PC can respond appropriately.

1.3.15 From V5.0: password may be transferred to a password-protected CPU

From version 5, if the CPU features write protection, i.e. write access to the CPU is only possible using the password, the required password can be transferred to the CPU. ComDrvS7 provides two functions for this purpose. The first function can be used to check whether a password is required for write access. The second function transfers the password to the CPU (you must have the correct password for this purpose) to unlock access. The password is unlocked until the communication link with the CPU is terminated.

1.3.16 From V5.0: reading DB-data from different DBs in a single function call

The two functions MPI_A_MixReadDBByte and MPI_A_MixReadDBWort can be used to read data from different data blocks using a single function call. For example, these can be used to read byte 12 from DB10 and byte 0 from DB11. These functions are relevant if the data to be read are not located in single data block.

1.3.17 From V4.0: routing support

From version 4, the communication paths NetLink, NETLink PRO, TCP/IP-direct and SIMATIC?-NET support routing. This means that you can access CPUs that have no direct connection to the PC, provided that they are located on a network that is linked to the CPU, which is connected to the PC. The benefit is that no single CPU must provide data collection functions, but instead, that you can access each of the pooled CPUs. Only one CPU is connected directly to the PC, which forwards the request to the other CPUs. The request may be routed via different bus systems to the CPU.
The condition is that the routing data was stored in the CPUs when the hardware was configured.

1.3.18 From V4.0: additional communication path SIMATIC®-NET

SIMATIC®-NET is supported from version 4, provided that the drivers were installed on the PC (this is true if the Simatic® Manager V5.1 or higher or Teleservice V6 or higher were installed.) In this case, it is possible, for example, to address CPs 5512, 5611 and the Siemens USB-MPI adapter.

1.3.19 From V4.0: support for remote maintenance access via the Siemens Teleservice

In the presence of Siemens Teleservice V6, ComDrvS7 can also communicate with a CPU via the telephone line. This also provides support for the Siemens Teleservice Adapters II.

1.3.20 From V3.6: additional communication path NETLink PRO (TCP/IP)

From Version 3.6, ComDrvS7 can access a CPU using the NETLink PRO. The communication on the PC side runs via TCP/IP. On the CPUs side, the NETLink PRO for MPI or Profibus-DP may be employed. In this case, all baud rates up to 12 Mbaud are supported.
The advantage of the NETLink PRO over the NetLink is that this can manage a max. of 4 PC connections. The NETLink PRO also provides automatic detection of the baud rate for the MPI or Profibus-DP.

1.3.21 From V3.5: additional communication path TCP/IP-direct

From Version 3.5, ComDrvS7 can also access a CPU with Ethernet-CP or a CPU with an integrated Ethernet-interface. The function `"MPI_A_Einleitung_TCP_IP_Direct"` was implemented for this purpose.

1.3.22 From V3.x: Individual read functions ate not limited to 128 bytes (64 words)

From version 3, the read functions of the ComDrvS7-DLL are no longer limited to 128 bytes per call. This means, for instance, that it is possible to request 200 clock memories in a single call.

1.3.23 From V3.x: new function to query the opstatus of the CPU

From version 3, ComDrvS7-DLL includes the function MPI_A_IstCPUInRun. This function can be used to determine whether the status of the connected CPU is RUN.

1.3.24 From V3.x: new functions to convert Real-, DINT and INT data types

From version 3, the ComDrvS7-DLL contains auxiliary functions to generate Real, DINT and INT numbers from BYTE or WORD buffers. Furthermore, REAL, INT and DINT numbers may be saved to BYTE or WORD buffers.

1.3.25 From V3.x: Byte buffers may be transferred to read and write functions that access byte operands

From version 3, the status and control buffers of the read/write functions that access the data type BYTE was changed, provided that the functions accessed the byte operands. In previous versions, the buffers have data type WORD. The data type of the buffer being transferred must be changed if the ComDrvS7-DLL is used in applications that were created with an earlier version. In this case, the effort can be minimised by copying the current WORD buffer into a BYTE buffer before the call to the DLL function (for the write functions). For read functions, the BYTE buffer can be copied to the existing WORD buffer after the call to the DLL function.
In general, however, the change should not present a problem.
The change was necessary to improve the usability of the above-mentioned conversion functions.

1.3.26 From V2.5: managing multiple connections

From version 2.5, the DLL has capabilities to manage multiple connections.

Example 1:
Serial port COM1 of a CPU is connected to MPI address 2. Serial port COM2 of a CPU is connected to MPI address 3.
Communications may be established with both CPUs. Then you can perform actions with the CPUs without interrupting the communication link with the other CPU.

Example 2:
Serial port COM1 is connected to a MPI network with 2 CPUs. The COM1 port can be used to establish and maintain a single connection to the CPUs. Then you can perform actions with the CPUs without interrupting a communication link with the other CPU.

Both sample programs "MPI V25 Demo Single-Thread" and "Demo V25 MPI Multi-Thread" illustrate these options.

1.3.27 From V2.5: TCP/IP communication with the aid of the MHJ-NetLink

From version 2.5, the DLL can establish a connection with a S7-CPU via TCP/IP. This requires the MHJ-NetLink. This is connected to a hub/switch or directly into the network adapter of a PC (using a crossover cable).
The main advantage of this connection is the speed, which is significantly higher than that of the serial connection.

<u>Important note:</u>
Other processes running on the PC may lead to communication errors, because the response times of MPI-connection cannot be met. To avoid this situation, the communication mode must be programmed in a thread with the priority "THREAD_PRIORITY_TIME_CRITICAL".

2 Different versions of ComDrvS7

2.1 Differences between the Lite version and the full version

A so-called **Lite-Version** is available for ComDrvS7.
The Lite version comprises a multi-user license (developer license). This version only provides access to data areas located within the data blocks. Both, reading and writing of data is possible.

The Lite version cannot be used to access other operand areas (e.g., clock memories, inputs, outputs, etc.). If it is necessary to program this type of access, the functions will return the corresponding error (see the table of error codes).

Any other function, such as the conversion functions, the function to determine accessible nodes, etc., may also be executed in the Lite version.
The explanation in respect of each function indicates whether this is not available in the Lite version or the restrictions that apply when it is used in the Lite version.

2.2 The multiple license

The multiple license can be used any number of your own projects/systems. The company name of the licensee is specified during the registration.
The multiple licence is for one developer.

Please read the relevant paragraphs of the licensing agreement.

2.3 Particularity of the MICRO version

With the MICRO version of ComDrvS7 you have access to plc families S7-1200® and LOGO!® (0BA7 or higher). The MICRO version is a multiple license.

2.4 Combination of the versions

If you are owner of a multiple-licence of ComDrvS7 and you want to access to a LOGO!®, you can buy the MICRO licence and extend the functionality.
You need to execute the function "MPI6_ActivateComDrvS7" twice. First with the activation code from the multiple licence. And a second time with the activation code of the MICRO version.

2.5 Extended Version

The extended version contains functions to read and write all the blocks inside a S7-300/400. So you can implement backup and restore functions in your own applications.
Please read the "ComDrvS7-V6-Extended-English.pdf" for detailed informations about the extended functions.

2.6 License Agreement and Terms of Use for the ComDrvS7 DLL

Please read the following license agreement carefully and thoroughly before you use the "ComDrvS7 DLL" (MPIA32_V**.DLL or MPIA64_V**.DLL) on your computer. By opening the software package or by using the software become contractually bound to the terms of the following license agreement. If you do not agree to be bound by these terms, then do not install the software. In this case, you can return the package immediately after purchase or after receipt to the manufacturer for a refund of your payments. The software is never sold outright, but it is licensed for use only. You only receive ownership of storage media (CD) and the manual and any related written documents.

1. The multi-license
The purchase of a **ComDrvS7 DLL multi-licenses** entitles you to use the DLL in as many projects as necessary without requiring additional license fees. The DLL may <u>not</u> be sold on its own, but only in conjunction with software projects.
The registration number must not be disclosed under any circumstances.
The ComDrvS7 **multi-license** may not be used for software products that are in direct competition with software products supplied by MHJ-Software. The multi-license of ComDrvS7 may not be used for standard visualization software. Such projects require that a separate license model MHJ-Software be negotiated.
This license must be specified in writing in the form of a contract. ComDrvS7 may not be distributed with projects where the driver provides the main functionality.
The multi-licence is for one developer.
Only the company which have purchased the multiple-license, have the right to copy the software products which includes the ComDrvS7-driver. If the buyer of ComDrvS7 sold his software to an end user and the end user makes copies of this software, then the end user also needs a multiple licence of ComDrvS7.

<u>Note: The multi-license is a license for one developer.</u>

2. Duration of license
The license is granted for an unlimited time period. The license will automatically be revoked without the need for a termination if you should violate any provision of this agreement. In the event of a termination, you are obliged to destroy the software as well as all copies of the software. You may terminate the license agreement at any time by destroying the Software and all copies thereof.

3. Limited guarantee
For a period of 6 months from date of acquisition, MHJ-Software & Ing.-Büro Weiß guarantee that the software is essentially free from defect in material and workmanship and that the substantial functions match the accompanying product manual. In the case of a justified notification of defects, the company reserves the right to carry out repairs or in the event that the reworking of fails, rescission or reduction at user's choice. Any other warranty, in particular with regard to the software being suitable for the user's purposes, and for any direct or indirect damages (e.g., loss of profit, business interruptions) as well as the loss of data or damages that were caused by the restoration of lost data are expressly excluded, except in cases of proven intent or gross negligence on the part of MHJ-Software & Ing.-Büro Weiß. In any case, the company’s entire liability shall be limited to the amount that was paid for the software.

4. Miscellaneous
This license agreement is governed by the laws of the Federal Republic of Germany. In the event that any provision of this license agreement is rendered wholly or partially invalid, this shall not affect the validity of the remaining provisions. The invalid provision shall be substituted by one that meets the intent and purpose of the invalid provision. There are no subsidiary agreements. Any modifications to this license agreement must be in writing. The same applies to the repeal of this clause.

3 Information on using ComDrvS7 with the various programming languages

ComdrvS7 can be used with the programming languages C++, C#, VB and Delphi. The following development environments may be used:

- Visual C++ (Microsoft)
- C++ Builder (Borland, Codegear, embarcadero)
- Visual Basic (Microsoft)
- Visual C# (Microsoft)
- Delphi (Borland , Codegear, embarcadero)

Since different versions of the individual development environments may be used, the user must ensure that the proper declarations are used for the respective programming languages.

3.1 Windows CE

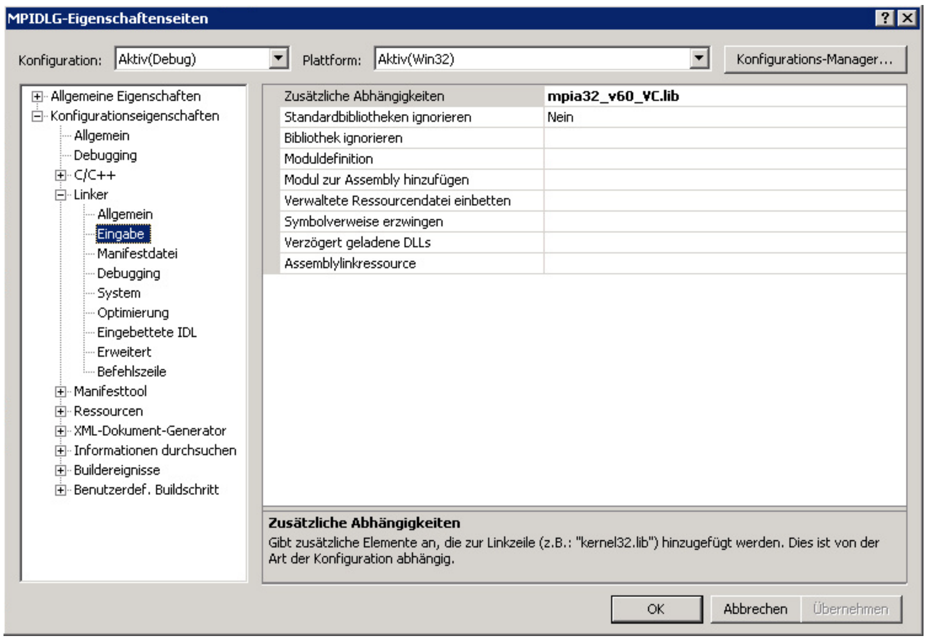
There is an example for the CE-Version of the driver as an Visual C++ 2008 solution.
You find this example in the directory "EXAMPLES Windows CE".
For WinCE-projects please use the DLLs from the directory "DLL-WinCE". Here you find the files for ARM and x86.

3.2 Visual C++

3.2.1 What must be considered?

The files in the directory "DLL Bit32\VC" or "DLL Bit64\VC" must be used. This contains the necessary DLL and LIB files. Furthermore, the header file that is present here must be linked with the VC project. These files can be used for all VC development environments from version 6 (VC98). The 64-bit DLL can be used with Visual Studio 2012 or higher.

To use the ComDrvS7-DLL in the project the lib file must be included in the project configuration. The following figure shows an example for VC 2008:



MPIDLG-Eigenschaftenseiten

Konfiguration: Aktiv(Debug) Plattform: Aktiv(Win32) Konfigurations-Manager...

- Allgemeine Eigenschaften
- Konfigurationseigenschaften
  - Allgemein
  - Debugging
  - C/C++
  - Linker
    - Allgemein
    - Eingabe
    - Manifestdatei
    - Debugging
    - System
    - Optimierung
    - Eingebettete IDL
    - Erweitert
    - Befehlszeile
  - Manifesttool
  - Ressourcen
  - XML-Dokument-Generator
  - Informationen durchsuchen
  - Buildereignisse
  - Benutzerdef. Buildschritt

Zusätzliche Abhängigkeiten	**mpia32_v60_VC.lib**
Standardbibliotheken ignorieren	Nein
Bibliothek ignorieren	
Moduldefinition	
Modul zur Assembly hinzufügen	
Verwaltete Ressourcendatei einbetten	
Symbolverweise erzwingen	
Verzögert geladene DLLs	
Assemblylinkressource	

Zusätzliche Abhängigkeiten
Gibt zusätzliche Elemente an, die zur Linkzeile (z.B.: "kernel32.lib") hinzugefügt werden. Dies ist von der Art der Konfiguration abhängig.

Fig.: LIB-file included in the VC project

In order to execute the application in the development environment the ComDrvS7-DLLs must be copied into the project directory.

3.2.2 Example for VC++

The example for VC6 is located in the directory "Example Visual C V6". This can also be used for versions < VC2008, the project is converted to the respective version when it is opened. The example for VC2008 is located in the directory "Example Visual C 2008".

Note:

The VC-examples cannot be used with the express version of Visual C, since the MFC class library is not available in the express version.

3.3 C++ Builder

3.3.1 What must be considered?

The files located in the directory "DLL Bit32\VC" or "DLL Bit64\VC" must be used. This contains the necessary DLL and LIB files. Furthermore, the header file that is present here must be linked with the C++ Builder project.
The files can be used in all Builder development environments from version 5.

In the Builder project, the LIB file is simply included in the project group of the EXE file. This is shown below.

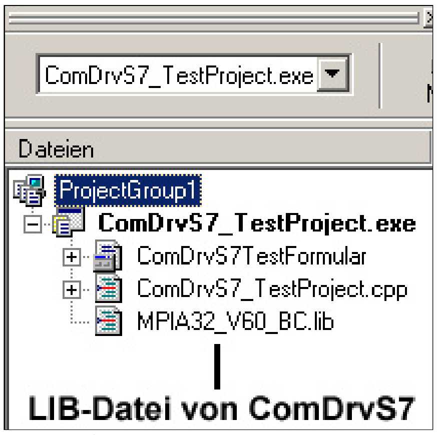


Fig.: LIB-file included in the EXE project.

In order to execute the application in the development environment the ComDrvS7-DLLs must be copied into the project directory.
If you use the 64-bit DLL, the extension of the lib file is ".a". The C++ Builder XE3 is the first version, where the 64 bit DLL can be used.

3.3.2 C++ Builder example

The installation directory of ComDrvS7 contains several examples for the C++ Builder. These are sorted in accordance with the version number of the Builder.
This includes examples for the Builder version C++Builder 5, C++Builder 2007 and C++Builder 2010. For versions < C++Builder 2007 you can use the Project C++Builder 5.

Newer versions (Builder XE, XE 2, ..) can use the example C++ Builder 2010.
There is also an 64-bit example for C++ Builder XE3 or higher.

3.4 Visual Basic

3.4.1 What Must Be Considered?

The files in the directory "DLL Bit32\VB" must be used. This contains the necessary DLL and LIB files.
This directory also contains the files "ComDrvS7V6_Declare_VB6.bas" and "ComDrvS7_Declare_VB2008.vb". These files supply the declarations of the ComDrvS7 functions.
The file "ComDrvS7_Declare_VB6.bas" must only be used for version 6 of Visual Basic!
The file "ComDrvS7V6_Declare_VB2008.vb" is used with Visual Basic versions later than VB6.
The following figure shows how this file is included in the project folder:

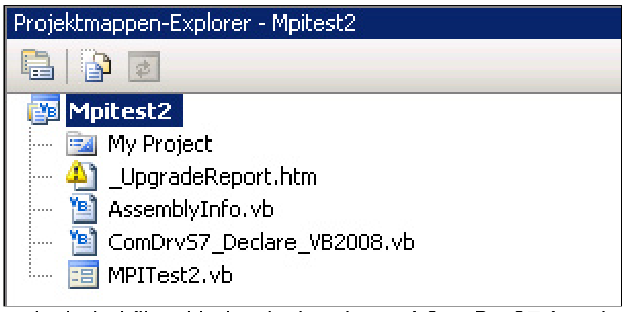


Fig.: Included file with the declarations of ComDrvS7-functions

Copy the ComDrvS7-DLLsinto the Windows-System32 directory to execute the application in the development environment.
For a **64-bit** application, you have to use the .Net-wrapper class. Please look at the example "Example Visual Basic 2012 64-Bit Wrapper".

3.4.2 Visual Basic examples

The installation directory of ComDrvS7 contains several Visual Basic examples. These are sorted according to the version number.
The example "Example Visual Basic 6" can be used with the VB6 version.
The example "Example Visual Basic 2008” can be used with VB 2008. This example can also be used with the Express version of VB2008.
Furthermore, the example "Example Visual Basic 2008 with wrapper" is available which uses the .Net-wrapper-class.

If you use VB 2010 or higher, open the example "Example Visual Basic 2010" or "Example Visual Basic 2010 with wrapper"

Important note:
If you start your application inside of Visual Studio, the DLLs of ComDrvS7 must be copied into the BIN-Directory of the solution.
Example:
If you set the configuration manager to "Debug" and "x86", you have to copy the DLLs into the directory "...\bin\x86\Debug\".

3.5 Visual C#

The files in the directory "DLL Bit32\VC" or "DLL-Bit64\VC" must be used. This contains the necessary DLL files. This directory contains another directory named "NET". This is where the DLL the wrapper-class is stored. It is named "ComDrvS7V6_Net.dll". Copy the DLLs into the project directory "..\bin\Debug".
The wrapper-class (or the ComDrvS7V6_Net.dll file) must be inserted into the project folder references. The following figure shows this situation:

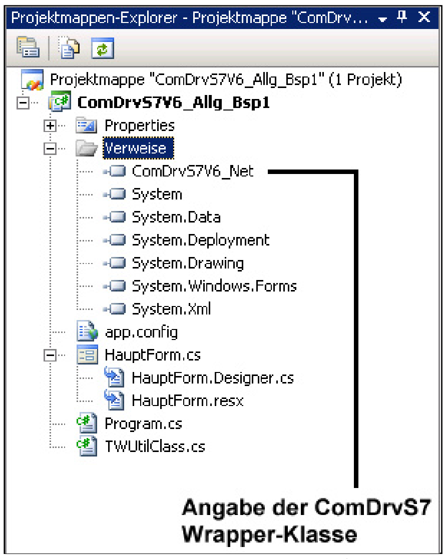


Fig.: DLL with wrapper-class inserted into the references of the project folder

Then the following using directive must be specified:

```
using MHJSW.ComDrvS7V6_Net;
```

For more information, see the comprehensive C# example.

3.5.1 Visual C# examples

The C# example that is available form the directory "Example Visual C# 2008" can be used. With Visual C# 2010 or newer use the example "Example Visual C# 2010". These are located in the ComDrvS7 installation directory.
For a **64-bit** application, please look at the example "Example Visual C# 2012 64Bit".

Refer to the example "Example Visual C# 2008 with PCPanel WPF" if you are using the PCPanel WPF Controls.

Important note:
If you start your application inside of Visual Studio, the DLLs of ComDrvS7 must be copied into the BIN-Directory of the solution.
Example:
If you set the configuration manager to "Debug" and "x86", you have to copy the DLLs into the directory "...\bin\x86\Debug\".

3.6 Delphi

3.6.1 What must be considered?

The files located in the directory "DLL Bit32\VC" or "DLL-Bit64\BC" must be used. This contains the necessary DLL and LIB files (respectively ".a"-file if 64 bit).
This also contains the files
"DelphiDeklarationFunktionen_V4_Bis_2006.TXT" and
"DelphiDeklarationFunktionen_Ab_Delphi2009.TXT".
These files supply the all the declarations for the ComDrvS7 functions. Depending on Delphi version being used, the data may be copied and included in the source code. The declarations are also available in the respective Delphi examples.

Copy the ComdrvS7 DLL files into the project directory.

3.6.2 Delphi examples

There are several examples for the different Delphi versions. The directory names indicate the Delphi version for which the respective example is intended.
The examples included are for version 4, 2006 and 2009. If you are using a Delphi version between version 4 and 2006, load the example for the version 4; this is then converted accordingly.

Newer versions of Delphi IDE can use the exampel of Delphi 2009.

4 Switching from an older versions of ComDrvS7

If you have created an application with an earlier version of ComDrvS7, you can benefit easily and quickly from the innovations of Version 6.

The read and write functions of version 6 have undergone significant optimisations to improve their speed. Furthermore, the new read and write functions can access password-protected CPUs without having to pass the password.
This is a good reason to switch to the new read-write functions. You can easily implement this change.

The procedure is as follows:

- Execute the initial function as usual.
- Call the function **MPI6_ChangeProtocolTypeForV5Functions** with parameter TakeV6Protocol set to 1.

All the old read-write functions (e.g. MPI_A_ReadMerkerByte, MPI_A_WriteMerkerByte, etc.) are converted to the new protocol type without further changes to the code.

Here the function **MPI6_ChangeProtocolTypeForV5Functions** must only be called once after you have executed the initial function. The settings apply until communications are terminated.

You can find a detailed description of the **MPI6_ChangeProtocolTypeForV5Functions** function in a different chapter.

Some of the old functions will no longer be mentioned in this manual. However, these are still included to ensure compatibility with ComDrvS7. Only function name and the parameters of the information functions have been translated into English. The descriptions in this manual are valid for these functions.

The description of the functions up to ComDrvS7 V5 is available from the earlier manual, which is also installed as a PDF file when you install ComDrvS7 V6.

5 General ComDrvS7 procedure

The image below shows the basic procedure when using ComDrvS7.

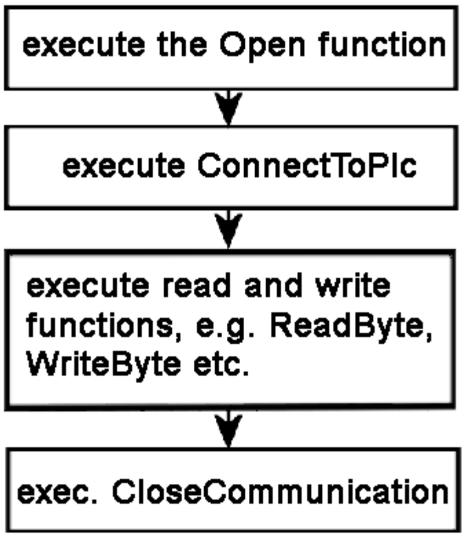

execute the Open function
execute ConnectToPlc
execute read and write functions, e.g. ReadByte, WriteByte etc.
exec. CloseCommunication

Fig.: Procedure to use ComDrvS7

Depending on the connection that was established with a CPU, one of the 6 **MPI6_Open** functions is called.
Then the **MPI6_ConnectToPLC** function is called.
If this function did not result in an error, the CPU may be accessed via the **read, write and information functions**.
If the data is exchanged cyclically with the CPU, so you can leave the connection open and regain access to the CPU at any time.
The function **MPI6_CloseCommunication** is only called at the end of the communication session (or in case of an error) to disconnect the CPU and to release the instance of ComDrvS7.

5.1 Change to another access route to the CPU

If you have developed an application that employs the MHJ-NetLink to establish a connection with the CPU, and if the new project uses a direct TCP/IP connection to the CPU, changes to your code are minimised.

In ComDrvS7, the access path to the CPU is defined via the open function. When the open function has been executed, there are no differences regarding the access path.

For example, if you must change from MHJ-NetLink to TCP/IP Direct, then the call must access "MPI6_OpenTcpIp" instead of "MPI6_OpenNetLink“. Otherwise, no changes are necessary. This also applies to all the other access paths.

5.2 Response to an error occurring in the open functions

Errors that occur when the open-functions are being processed are usually caused by a hardware error or the parameters passed to the particular Open function are not correct.
An Open function that returns an error does not require a call to that another function since the communication instance is removed in the Open function when the error occurs.

5.3 Reaction to an error that occurs in the open functions

When an error occurs in the MPI6_ConnectToPLC function or one of the read-/write functions with respect to the CPU (e.g. MPI6_ReadByte, MPI6_WriteDword, etc.), the communication instance should generally be closed by means of the function MPI6_CloseCommunication and subsequently reopened with a call to the respective open function.

It is important to issue a call to the MPI6_CloseCommunication function, otherwise the communication instance remains in memory!

Note regarding errors that are not caused by communication problems

Errors that result from faulty parameter being transferred, etc. are an exception. An example is the error that occurs when an action with a WLD file does not find the file in the specified path. In this case, of course it is usually sufficient to correct the path without a call to the MPI6_CloseCommunication function.
These errors usually occur in the development phase of the application.

See chapter "Error messages" for a list of error codes.

6 Description of the different functions

This chapter discusses the individual functions of ComDrvS7 with their tasks. Most of the descriptions are followed by a brief example that employs the respective function.

6.1 Basic information explaining the different functions of ComDrvS7

The individual functions are shown in C syntax. Because the examples are very simple, they should not pose any problems to VB, Delphi and C #programmers. The explanations of the parameters are the same for all programming languages. Any variation is detailed in the explanation of each parameter.

When using the wrapper class in .Net, the handle for the communication instance is not required, since this handle is managed by the wrapper class.

6.2 The function: MPI_A_GetDLLError or MPI_A_GetDLLErrorEng

Brief description

You can call the MPI_A_GetDLLError function to obtain a string (null terminated), which describes the error being returned in detail. Each function in the DLL returns the value '0' (FALSE) as a function value if an error has occurred during the execution. In this case, the variable ErrorCode that must be passed to each function contains the error code.
If you want to describe the error, you can call the function MPI_A_GetDLLError.
Chapter "Error messages" contains a description of the error codes.
The function MPI_A_GetDLLErrorEng returns the error messages in English.
In the .Net wrapper class, the functions have the name "MPI6_GetDLLError" or "MPI6_GetDLLErrorEng"

Description of the Parameters

Argument	C-Type	Description
Handle	INT	Das Handle der Kommunikationsinstanz welche angesprochen wird. (Entfällt bei .Net-Wrapper-Klasse)
ErrorString	CHAR*	Enthält den String für den übergebenen Errorcode.
ErrorCode	WORD	Error-Code für den der String geliefert werden soll.
Function return	BOOL	Wurde die Funktion erfolgreich ausgeführt, so wird der Wert '1' (TRUE) geliefert. Bei einem Fehler ist der Rückgabewert '0' (FALSE).

Beispiel

In den Beispielen zu den einzelnen Funktionen, wird MPI_A_GetDLLError häufig verwendet.

6.3 The function: MPI6_OpenRS232

Brief description

The MPI6_OpenRS232 function must be called to communicate with a CPU for the first time, provided that the communication link is established via a **serial interface** or a **virtual COM port on a USB adapter**.
The function opens the specified PC interface using the specified communication parameters.
In addition, MPI network data must be transferred to the function.
Execution of the function can only succeed if a CPU is connected to the specified PC interface.

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path is used (COM port, baud rate, etc.). This means that the handle defines the communication instance (not applicable for .Net wrapper class).

Description of the Parameters:

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
ComNr	INT	Definition of the PCs serial port to which the PLC is connected. E.g. '1' for COM1
BaudRate	LONG	Defines the baud rate to communicate with the PLC. This value depends on the PC/MPI cable being used. The older models only support a baud rate of 19200 baud. With the latest cables the baud rate may be set to 19200, 38400, 57600 and 115200 baud. **You may also enter: 19200, 38400, 57600, 115200** **If this value is specified incorrectly, it is impossible to establish communications.**
PGAddress	BYTE	Specifies the MPI/DP address that is used by the communication instance to log into the MPI/DP network. **It is important to note that the entered address may be used by any other device in the connected network.** By default, the programming devices are set to address '0 '.
HighestAddress	BYTE	Defines the highest address that may be used in connected MPI-network. Enter a value of **15, 31, 63 or 126**. It is important to ensure that all devices in the connected network have the same highest address.
ComWasAlreadyUsed	BOOL*	This parameter returns TRUE if the port is already in use by another communication instance. In this case, the communication parameters (e.g. baud rate and PGAddress) that were passed with the function were not used. Communication can take place however.
Error	WORD*	If the parameter returns FALSE, then the interface was still unallocated and the specified communication parameters were set.

Function return	BOOL	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
		If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALS E).

Example

The following example opens the COM2 interface using the function MPI6_OpenRS232. A transmission rate of 115200 baud was selected.

```
//Variables
int ComNr=2;          //COM2 port
long BaudRate=115200; //baud rate 115200BYTE PGMPIAdresse=0;
//address of the DLL application = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
bool SchnittstelleWarSchonAllokiert=false; //true if the
                                         //interface was
                                    //already in use
WORD Error=0;         //error variable
char ErrorString[255];//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance

//establish connection
if (!MPI6_OpenRS232(&MPIHandle, ComNr, BaudRate, PGMPIAdresse,
                            HoechsteMPI,
                            &SchnittstelleWarSchonAllokiert,
                            &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Einleitung war erfolgreich.", "",
           MB_ICONINFORMATION);
```

6.4 The function: MPI6_OpenNetLink

Brief description

The MPI6_OpenRS232 function must be called to communicate with a CPU for the first time, provided that the communication link is established via TCP/IP using a **MHJ-NetLink** or a **MHJ-NetLink++**.
The function establishes a connection with a NetLink that has the specified IP address. In addition, MPI network data must be transferred to the function.

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL To ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the NetLink that will be used to execute the communications. The address is entered in the form "172.16.130.84".
PGAddress	BYTE	Specifiy the MPI/DP address that the communication instance must use to log on to the network. **It is important to note that the entered address must not be used by any other device on the connected network.** By default, programming devices are set to the address '0'. **Attention:** this function call cannot change the PG address of the MHJ-NetLink. This must be defined using the supplied configuration utility.
HighestAddress	BYTE	Defines the highest MPI/DP address, which may be used on the connected network. Here the values **15, 31, 63 or 126** must be specified. It is important to ensure that all devices on the connected network have the same highest address. **Attention:** this function call cannot change the PG address of the MHJ-NetLink. This must be defined using the supplied configuration utility.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_OpenNetLink function to connect to a NetLink with the IP address 172.16.130.84.

```
BYTE PGMPIAdresse=0;  //MPI address of the communication instance = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
char IPAdresseStr[50]={0};
//enter the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_OpenNetLink(&MPIHandle, IPAdresseStr, PGMPIAdresse,
                              HoechsteMPI, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Einleitung erfolgreich.", "",
           MB_ICONINFORMATION);
```

Note:
It should be mentioned again that the specifying the PG-address and the highest MPI/DP address does not change the values defined in the NetLink. The NetLink settings are defined by means of the configuration utility that is included. These settings must only be entered once, thereafter the settings are stored permanently the MHJ-NetLink.

6.5 The function: MPI6_OpenTcpIp

Brief description

The MPI6_OpenTcpIp function must be called to communicate with a CPU for the first time, provided that the **communication link is established via TCP/IP with an Ethernet-CP or a CPU with an integrated Ethernet interface**.
The function establishes a connection with an Ethernet-CP that has the specified IP address. The slot number of the CPU is also required. For S7-300 systems, this must usually be defined as Slot 2.

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the Ethernet-CP or the integrated Ethernet interface of the CPU that will be used to execute the communications. The address is entered in the form "172.16.130.84".
PlcSlotNr	INT	Specifies the slot of the CPU with which you want to communicate. For S7-300 systems, this must usually be defined as slot 2.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_OpenTcpIp function to establish a connection with an Ethernet-CP with the IP address 172.16.130.84.

```
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
int CPUSlotNr=2; //slot of the CPU to be addressed in the PLC rack
char IPAdresseStr[50]={0};
//enter the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_OpenTcpIp(&MPIHandle, IPAdresseStr,
                    CPUSlotNr, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
              MB_ICONINFORMATION);
```

6.6 The function: MPI6_OpenTcpIp_S71500

Brief description

It is imperative to call the function MPI6_OpenTcpIp_S71500 to communicate for the first time with a CPU of the S7-1500® family.

The function establishes a connection with a S7-1500® that has the specified IP address.

Please read the chapter "Required settings in a PLC 1500 from Siemens" for the important hardware settings!

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the Ethernet-CP or the integrated Ethernet interface of the CPU that will be used to execute the communications. The address is entered in the form "172.16.130.84".
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_OpenTcpIp_S71500 function to establish a connection with a S7-1500? CPU with the IP address 172.16.130.84

```
WORD Error=0;          //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;      //Handle of the new communication instance
char IPAdresseStr[50]={0};
//specify the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_OpenTcpIp_S71500(&MPIHandle, IPAdresseStr, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
             MB_ICONINFORMATION);
```

6.7 The function: MPI6_OpenTcpIp_S71500Ext

Brief description

It is imperative to call the function MPI6_OpenTcpIp_S71500 to communicate for the first time with a CPU of the S7-1500® family.

The function establishes a connection with a S7-1500® that has the specified IP address.
In addition to the function "MPI6_OpenTcpIp_S71500", the function supports the selecting of the used network adapter. This function is required, when there are more than one network adapter available.

Please read the chapter "Required settings in a PLC 1500 from Siemens" for the important hardware settings!

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the Ethernet-CP or the integrated Ethernet interface of the CPU that will be used to execute the communications. The address is entered in the form "172.16.130.84".
IPAddressNetworkAdapter	CHAR*	IP address of the PC-network adapter. This adapter is connected with the PLC.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

6.8 The function: MPI6_OpenTcpIp_S71200 (Also in MICRO version)

Brief description

It is imperative to call the function MPI6_OpenTcpIp_S71200 to communicate for the first time with a CPU of the S7-1200® family.
The function establishes a connection with a S7-1200® that has the specified IP address.

The function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

If you use a plc with **firmwareversion 4 (or higher)**, please read the infos in chapter "Required settings in a PLC 1500® (and S7-1200® from firmware version 4) from Siemens"

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the Ethernet-CP or the integrated Ethernet interface of the CPU that will be used to execute the communications. The address is entered in the form "172.16.130.84".
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_OpenTcpIp_S71200 function to establish a connection with a S7-1200? CPU with the IP address 172.16.130.84

```
WORD Error=0;          //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;      //Handle of the new communication instance
char IPAdresseStr[50]={0};
//specify the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_OpenTcpIp_S71200(&MPIHandle, IPAdresseStr, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
             MB_ICONINFORMATION);
```

6.9 The function: MPI6_OpenTcpIp_Logo (Only MICRO-version)

Brief description

It is imperative to call the function MPI6_OpenTcpIp_Logo to communicate for the first time with a CPU of the LOGO!® (0BA7 or higher) family.

The function establishes a connection with a LOGO!® that has the specified IP address (Infos about these settings are described in the chapter "Configuration of the LOGO's IP address").

If you use a **LOGO!® 0BA8 (and higher)**, please read the chapter "Special behaviors of a LOGO!® 0BA8 (and higher)".

This function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddress	CHAR*	Provide the IP address of the Ethernet-CP or the integrated Ethernet interface of the CPU that will be used to execute the communications. The address is entered in the form "172.16.130.84".
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_OpenTcpIp_Logo function to establish a connection with a LOGO!® - CPU with the IP address 172.16.130.84

```
WORD Error=0;          //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;      //Handle of the new communication instance
char IPAdresseStr[50]={0};
//specify the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_OpenTcpIp_Logo(&MPIHandle, IPAdresseStr, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
             MB_ICONINFORMATION);
```

6.10 The function: MPI6_Open_NetLinkPro_TCP_AutoBaud

Brief description

The **MPI6_Open_NetLinkPro_TCP_AutoBaud** function must be called to communicate with a CPU for the first time, provided that the **communication link is established via a NETLink PRO with a MPI or Profibus-DP interface of a CPU**.
The function establishes a connection with a NETLink PRO that has the specified IP address. In addition, the PG-address and the highest address that exists in the network (MPI or Profibus) must be specified. In contrast to the function **MPI6_Open_NetLinkPro_TCP_SelectBaud** , this function is used when the NETLink PRO must automatically detect the baud rate that is in use on the bus. Here it is important to note that it may take a few seconds more to establish communications,since baud rate detection requires some time. If the communication link is established and disconnected at short intervals (e.g. to communicate with many different CPUs) you should use the **MPI6_Open_NetLinkPro_TCP_SelectBaud** function.

The **MPI6_Open_NetLinkPro_TCP_AutoBaud** function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddressStr	CHAR*	Provide the IP address of the NETLink PRO that will be used to execute the communications. The address is entered in the form "172.16.130.84".
PGAddress	BYTE	The MPI/DP address that the communication instance must use to log on to the MPI/DP network. **It is important to note that the entered MPI/DP address must not be used by any other device on the connected MPI/DP network.** By default, the programming devices are set to MPI/DP address '0'.
HighestAddress	BYTE	Defines the highest MPI/DP address, which may be used in the connected network. Here the values **15, 31, 63 or 126** must be specified. It is important to ensure that all devices on the connected MPI/DP network are set to the same highest MPI/DP address.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_Open_NetLinkPro_TCP_AutoBaud function to connect to a NETLink PRO with the IP address 172.16.130.84.

```
BYTE PGMPIAdresse=0;  //MPI address of the communication instance = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
char IPAdresseStr[50]={0};
//enter the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_Open_NetLinkPro_TCP_AutoBaud(&MPIHandle, IPAdresseStr,
                                      PGMPIAdresse, HoechsteMPI,
                                      &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
              MB_ICONINFORMATION);
```

Note:

Before the NETLink PRO is used for the first time, it must be set to the necessary communication parameters. Use the configuration software that is supplied with ComDrvS7. After the ComDrvS7 has been installed, it is located in the directory "NETLink PRO Konfigurator".
The settings you have entered (such as IP address, subnet mask, etc.) are permanently stored in the NETLink PRO. This means that the settings are still available after the supply voltage has been disconnected from of the NETLink PRO.

No additional driver is required to operate the NETLink PRO with ComDrvS7!

6.11 The function: MPI6_Open_NetLinkPro_TCP_SelectBaud

Brief description

The **MPI6_Open_NetLinkPro_TCP_SelectBaud** function must be called to communicate with a CPU for the first time, provided that the **communication link is established via a NETLink PRO with a MPI or a Profibus-DP interface of a CPU**.
The function establishes a connection with a NETLink PRO that has the specified IP address. In addition, the PG-address and the highest address that exists in the network (MPI or Profibus) must be specified. In contrast to the function **MPI6_Open_NetLinkPro_TCP_AutoBaud** , this function is used when the baud rate of the MPI/DP bus is known. In such a case, the time to detect the baud rate is not required, which speeds up the initialisation. You should use this function if the communication link is established and disconnected at short intervals (e.g. to communicate with many different CPUs). Provided however, that the baud rate of the MPI/DP bus is known. However, this should apply under most circumstances.

The **MPI6_Open_NetLinkPro_TCP_SelectBaud** function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path (IP address) is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
IPAddressStr	CHAR*	Provide the IP address of the NETLink PRO that will be used to execute the communications. The address is entered in the form "172.16.130.84".
PGAddress	BYTE	The MPI/DP address that the communication instance must use to log on to the MPI/DP network. **It is important to note that the entered MPI/DP address must not be used by any other device on the connected MPI/DP network.** By default, the programming devices are set to MPI/DP address '0'.
HighestAddress	BYTE	Defines the highest MPI/DP address, which may be used in the connected MPI/DP network. Here the values **15, 31, 63 or 126** must be specified. It is important to ensure that all devices on the connected network are set to the same highest MPI/DP address.
IsProfibusDP	BOOL	When communicating with the CPU via Profibus-DP, set this value to '1 '(TRUE). When communicating via MPI, set this value to '0' (FALSE).

BaudrateUsed	WORD	Passing the baud rate defined for the MPI/DP network. Here the following valued have been defined: 9.6 kBaud: MPIA_BAUD_96 corresponds to a value 0 19.2 kBaud: MPIA_BAUD_19_2 corresponds to a value 1 45.45 kBaud: MPIA_BAUD_45_45 corresponds to a value 2 93.74 kBaud: MPIA_BAUD_93_75 corresponds to a value 3 187.5 kBaud: MPIA_BAUD_187_5 corresponds to a value 4 500 kBaud: MPIA_BAUD_500 corresponds to a value 5 1500kBaud: MPIA_BAUD_1500 corresponds to a value 6 3000 kBaud: MPIA_BAUD_3000 corresponds to a value 7 6000 kBaud: MPIA_BAUD_6000 corresponds to a value 8 12000 kBaud: MPIA_BAUD_12000 corresponds to a value 9
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example employs the MPI6_Open_NetLinkPro_TCP_SelectBaud function to connect to a NETLink PRO with the IP address 172.16.130.84. The NETLink PRO is connected to the Profibus-DP interface of the CPU. The DP network is set to operate at 1.5MBaud.

```
BYTE PGMPIAdresse=0;  //DP address of the communication instance = 0
BYTE HoechsteMPI=31;  //highest DP address permitted in network = 31
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
char IPAdresseStr[50]={0};
bool IstProfibusDP=true;//this is a DP network
WORD VorgabeBaudrate=MPIA_BAUD_1500;//the baud rate is set mto 1.5MBaud
eingestellt
//enter the IP address
strcpy(IPAdresseStr, "172.16.130.84");
//establish connection
if (!MPI6_Open_NetLinkPro_TCP_SelectBaud(&MPIHandle, IPAdresseStr,
                                       PGMPIAdresse, HoechsteMPI,
                                         IstProfibusDP, VorgabeBaudrate
                                        &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
              MB_ICONINFORMATION);
```

Note:

Before the NETLink PRO is used for the first time, it must be set to the necessary communication parameters. Use the configuration software that is supplied with ComDrvS7. After the ComDrvS7 has been installed, it is located in the directory "NETLink PRO Konfigurator".
The settings you have entered (such as IP address, subnet mask, etc.) are permanently stored in the NETLink PRO. These settings are also retained when the supply power to the NETLink PRO is turned off.

<u>No</u> additional driver is required to operate the NETLink PRO with ComDrvS7!

6.12 The function: MPI6_Open_SimaticNet (only 32-bit, not CE)

Brief description

The **MPI6_Open_SimaticNet** function must be called to communicate with a CPU for the first time, provided that the **communication link is established via the SIMATIC® NET driver.** This communication can be used to access the Siemens USB-MPI adapter, as well as CPs 5511, 5612 etc.. Communications via a TS Adapter II are also supported.

The condition is that the SIMATIC? NET driver was installed on the PC. This driver is installed on the PC, for example, when the Simatic?-Manager (from V5.1), the driver for the SIEMENS-USB adapter or the Teleservice V6 are installed. You must select the interface to be used here in the "PG/PC interface configuration" dialog. You can access this dialog by means of the file "s7epatsx.exe" in the Windows System32 directory.
With the Siemens Teleservice V6, you can start a remote query via the telephone line using the TS-adapter II.

The function **MPI6_Open_SimaticNet** establishes a connection with the device that was selected in the dialog "PG/PC interface configuration".

The function **MPI6_Open_SimaticNet** function creates a communication instance. The variable "Handle" supplies the "identification" for this instance. This identifier must be passed to the other functions of the DLL to ensure that the specified communication path is used (not required for the .Net wrapper class).

Description of the parameters

Argument	C-Type	Description
Handle	INT*	This is where the handle of the newly generated communication instance is returned (not applicable for the .Net wrapper class).
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example for MPI6_Open_SimaticNet:

The example below employs the function MPI6_Open_SimaticNet to establish a connection with the interface that was selected in the "PG/PC interface configuration" dialog.

```
WORD Error=0;          //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;      //handle of the new communication instance

if (!MPI6_Open_SimaticNet(&DLLHandle, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
             MB_ICONINFORMATION);
```

Note:

The SIMATIC®-NET connection must not be used to establish connections MHJ-NetLink, NetLink PRO or TCP/IP-Direkt. Please use the appropriate initialisation functions (e.g. MPI6_Open_NetLinkPro_TCP_SelectBaud, MPI6_OpenTcpIp, etc.) for these communication paths.

6.13 The function: MPI6_CloseCommunication

Conditions to execute the function

The initialisation functions MPI6_OpenXXXX must have been completed successfully.

Brief description

The MPI6_CloseCommunication function must be called last in order to stop the communications with a PLC for good. This function also closes the interface that was opened by means of the function "MPI6_OpenRS232" or via the MPI6_OpenNetLink, MPI6_OpenTcpIp, MPI6_Open_SimaticNet, MPI6_Open_NetLinkPro_TCP_AutoBaud or MPI6_Open_NetLinkPro_TCP_SelectBaud functions. Furthermore, the communication instance identified by the specified handle will be eliminated.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example for MPI6_CloseCommunication:

In the example below, a previously established communication link with a CPU is terminated, the interface or socket is closed and the communication instance is eliminated.

```
WORD Error=0;
char ErrorString[255]={0};//error string to return the error

if (!MPI6_CloseCommunication(MPIHandle, &Error)){
      MPI_A_GetDLLError(ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
else {
     MessageBox(AppHandle, "communications terminated without
errors.", "",                  MB_ICONEXCLAMATION);
}//end else
```

Note:
Even if the function returns an error, the serial interface or the socket is closed and the communication instance eliminated. One exception is the error ERROR_MPIA_PARAMETER_ERROR (number 510) that does not pass a correct handle. In this case, no action can be executed (does not apply to the .Net wrapper class).

6.14 The function: MPI6_GetAccessibleNodes

Conditions to execute the function

The initialisation functions MPI6_OpenXXXX must have been completed successfully.
Important:
The function cannot be executed if the initialisation was executed by means of the "MPI6_OpenTcpIp", "MPI6_OpenTcpIp_S71200" or other TCP/IP-functions.

Brief description

The MPI6_GetAccessibleNodes function can be used to identify the MPI/DP addresses of the devices connected to an MPI network or to Profibus-DP.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Addresses	INT*	Integer array that lists the MPI/DP addresses of each node in the connected MPI/DP network. The array must accommodate 127 integer fields.
countAddresses	INT*	Returns the number of nodes connected to the MPI/DP network.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example MPI6_GetAccessibleNodes:

The example below determines the nodes that are accessible on the connected MPI-network.

```
int ComNr=2;            //COM2 port
long BaudRate=115200; //baud rate 115200
BYTE PGMPIAdresse=0;  //MPI address of the DLL application = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
bool SchnittstelleWarSchonAllokiert=false;
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
int Teilnehmer[130]={0}; //array for the nodes
int AnzahlTeilnehmer=0;
char AusgabeText[255]={0};
//establish connection
if (!MPI6_OpenRS232(&MPIHandle, ComNr, BaudRate,
                  PGMPIAdresse,
                   HoechsteMPI,
                   &SchnittstelleWarSchonAllokiert,
                   &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Einleitung war erfolgreich.", "",
           MB_ICONINFORMATION);
//determine the accessible nodes
if (!MPI6_GetAccessibleNodes(MPIHandle, Teilnehmer,
                            &AnzahlTeilnehmer,
                             &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
else {
     //display the number of nodes
     wsprintf(AusgabeText, "Es sind %i Teilnehmer am MPI-Netz
              angeschlossen", AnzahlTeilnehmer);
     MessageBox(AppHandle, AusgabeText, "", MB_ICONEXCLAMATION);
}//end else
//terminate communications
if (!MPI6_CloseCommunication(MPIHandle, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communications terminated without errors.", "",
                 MB_ICONINFORMATION);
```

6.15 The function: MPI6_SetRoutingData

Conditions to execute the function

The initialisation functions (with the exception of MPI6_OpenRS232) (e.g. MPI6_OpenNetLink, MPI6_OpenTcpIp, etc.) must have been completed successfully.

Brief description

The MPI6_SetRoutingData function must be called before you can access a CPU that is not directly connected using the MPI6_ConnectToPLCRouting function. The parameters of the function define the CPU to which you want to be routed. Routing means that you do not communicate with the CPU that is directly connected to the PC but via a different CPU that is linked with this CPU.

Important exception:

Routing can employ communication channels MHJ-NetLink, NetLink PRO, TCP/IP direct and SIMATIC?-NET. Routing is not possible with the MPI6_OpenRS232 function.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Routing example:

The MPI6_ConnectToPLCRouting function is explained by an example regarding routing.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
TargetSlotNr	BYTE	Here you must specify the slot number (i.e. the slot) of your CPU. For S7-300 systems, this must be defined as Slot 2. With S7-400 systems, you can find the slot number in the hardware configuration.
TargetRackNr	BYTE	Specifies the rack number where the CPU was installed. Normally, the CPU is installed in the rack 0, i.e. this should be set to 0.
TargetMPI_DP_Address	BYTE	If the target CPU is accessible via a MPI or a DP network, then this parameter must be set to the MPI/DP address of the target CPU in this network.
TargetIPAddressStr	char*	If the target CPU is accessible via a TCP/IP network, then this parameter must be set to the IP address of the CPU or the Ethernet CP (that is located on the rack of the CPU).
TargetSubnetID_High	WORD	High-word of the subnet ID of the S7 network that will be used to access the CPU. The subnet ID is defined in the hardware configuration.
TargetSubnetID_Low	WORD	Low-word of the subnet ID of the S7 network that will be used to access the CPU. The subnet ID is defined in the hardware configuration.
TargetNetIsMPI_DP_Net	BYTE	If the CPU is accessed via a MPI/DP network, then this value must pass a value of 1. In this case, the parameter ZielBaugruppeIPAdresseStr is ignored. The parameter ZielBaugruppeMPI_DP_Adresse must specify in the correct address. If a value of 0 is supplied, then the CPU is specified by means if the IP address, the MPI/DP address is ignored and should be set to 2.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

6.16 The function: MPI6_ConnectToPLC

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.

Brief description

The MPI6_ConnectToPLC function must be called before you can address the desired CPU with read or write access functions. This function determines which node is addressed via the connected MPI or Profibus-DP network. Here the communication partner is specified by the MPI or the Profibus address.

Important exception:

If the initialisation was executed by means of the MPI6_ConnectToPLC function, then the MPI address is a dummy value, since the CPU is already defined by the IP address and the CPU slot number. In this case, the MPI/DP address must always be specified as 2.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PlcAddress	BYTE	Contains the MPI/DP address of the required communication partner. The value of the address can range from 0 to 126. For communication path TCP-IP direct, enter a value of 2.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

If the function returns one of the following error-codes (decimal):
1046: PLC is not a S7-1500®
1045: PLC is not a LOGO
1044: PLC is not a S7-1200®
then the function "MPI6_CloseCommunicatio" was already executed inside of the function. These errors occur, if the wrong open-function was called. For example the open-function of the S7-1200 and a S7-1500 is actually plugged.

Example

The example below establishes a communication link with a PLC. The CPU has the MPI address 10.

```
int ComNr=2;           //COM2 port
long BaudRate=115200; //baud rate 115200
BYTE PGMPIAdresse=0;  //MPI address of the DLL application = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
bool SchnittstelleWarSchonAllokiert=false; //true if the
                                           //port was already
                                           //in use
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
BYTE AGMPIAdresse=10; //the MPI address of the CPU to be accessed
//establish connection
if (!MPI6_OpenRS232(&MPIHandle, ComNr, BaudRate, PGMPIAdresse,
                            HoechsteMPI,
                            &SchnittstelleWarSchonAllokiert,
                            &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
          MB_ICONINFORMATION);
//establish communications
if (!MPI6_ConnectToPLC(MPIHandle, AGMPIAdresse, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
else {
     MessageBox(AppHandle, "communications established successfully!",
"",                 MB_ICONINFORMATION);
}//end else
//terminate communications
if (!MPI6_CloseCommunication(MPIHandle, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communications terminated without errors.", "",
                 MB_ICONINFORMATION);
```

6.17 The function: MPI6_ConnectToPLCRouting

Conditions to execute the function

The initialisation functions (with the exception of MPI6_OpenRS232) (e.g. MPI6_OpenNetLink, MPI6_OpenTcpIp, etc.) must have been completed successfully. In addition, the routing data must have been transferred with the MPI6_SetRoutingData function.

Brief description

The MPI6_ConnectToPLCRouting function must be called before you can address the desired CPU with read or write access functions. In contrast to the MPI6_ConnectToPLC function, the CPU that is connected directly to the PC will not be accessed. Here a CPU will be accessed that is connected via a network link with the directly connected CPU. The CPU that will be accessed can be specified by means of the parameters of the MPI6_SetRoutingData function.

Important exception:

If the initialisation was executed by means of the MPI6_OpenRS232 function, then it is not possible to perform routing. In this case, the MPI6_ConnectToPLCRouting function must not be used.

PLC family S7-1500®, S7-1200® and LOGO!

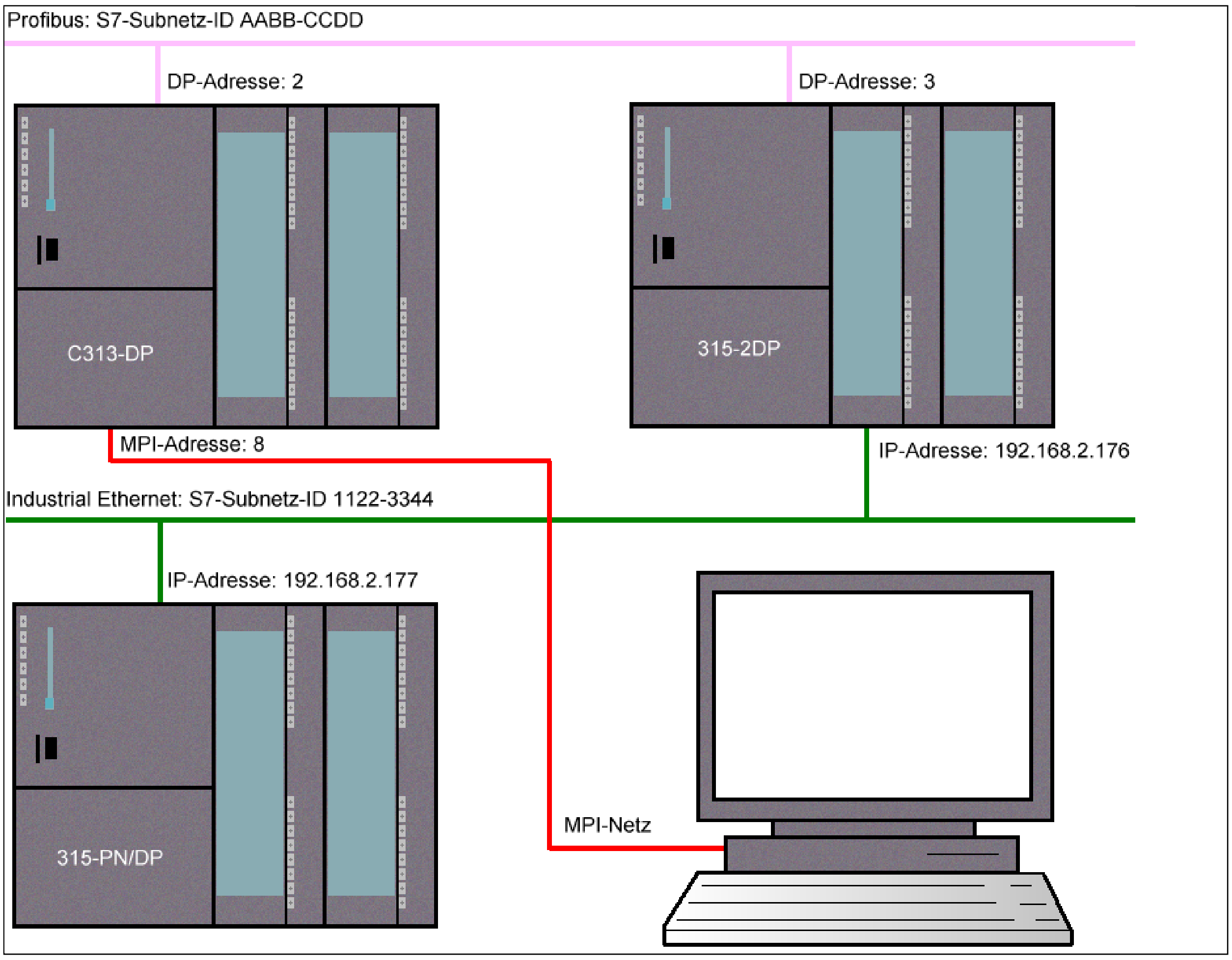
This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PlcAddress	BYTE	Contains the MPI/DP address of the directly connected communication partner. The value of the address can range from 0 to 126. For communication path TCP-IP direct, enter a value of 2.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

6.18 Routing example

The example below shows both functions, MPI6_SetRoutingData and MPI6_ConnectToPLCRouting. In the example, the PC is connected to the MPI interface of a CPU via a NetLink PRO. A network links this CPU to two other CPUs. This is shown in the following figure:


Profibus: S7-Subnetz-ID AABB-CCDD
DP-Adresse: 2
DP-Adresse: 3
C313-DP
315-2DP
MPI-Adresse: 8
IP-Adresse: 192.168.2.176
Industrial Ethernet: S7-Subnetz-ID 1122-3344
IP-Adresse: 192.168.2.177
315-PN/DP
MPI-Netz

The example below shows both functions, MPI6_SetRoutingData and MPI6_ConnectToPLCRouting. In the example, the PC is connected to the MPI interface of a CPU via a NetLink PRO. A network links this CPU to two other CPUs. This is shown in the following figure:

6.18.1 Initialisation by means of a NetLink PRO

In the first step, the initialisation function for the NetLink PRO communication path must be called.

```
BYTE PGMPIAdresse=0;  //MPI address of the communication instance = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
char IPAdresseStr[50]={0};
//enter the IP-address of the NetLink PRO
strcpy(IPAdresseStr, "192.168.2.100");
//establish connection
if (!MPI6_Open_NetLinkPro_TCP_AutoBaud(&MPIHandle, IPAdresseStr,
                                    PGMPIAdresse, HoechsteMPI,
                                   &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
      return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
              MB_ICONINFORMATION);
```

Here there is no different to a call that does not use routing. The IP address of the NetLink PRO is specified in the call. In this example, the IP address is "192.168.2.100".

6.18.2 Routing data transfer

Now the data must be passed to the actual CPU being accessed using the MPI_A_SetRoutingData function.

```
BYTE ZielBaugruppeSlotNr=2;
BYTE ZielBaugruppeRackNr=0;
BYTE ZielBaugruppeMPI_DP_Adresse=2;//address not relevant
char ZielBaugruppeIPAdresseStr[50]={0};
strcpy(ZielBaugruppeIPAdresseStr, "192.168.2.177");//IP of the 315-PN/DP
WORD ZielSubnetzID_High=0x1122;
WORD ZielSubnetzID_Low=0x3344;
BYTE ZielNetzIstMPI_DP_Netz=0;//the target network is Ethernet
//
if (!MPI6_SetRoutingData(DLLHandle, ZielBaugruppeSlotNr,
                         ZielBaugruppeRackNr, ZielBaugruppeMPI_DP_Adresse,
                         ZielBaugruppeIPAdresseStr, ZielSubnetzID_High,
                         ZielSubnetzID_Low, ZielNetzIstMPI_DP_Netz,
                         &Error)){
        //error
        MPI_A_GetDLLError(DLLHandle, DLLErrorString, Error);
        MessageBox(AppHandle, DLLErrorString, "", MB_ICONSTOP);
        //
        return;
}//end if
MessageBox(AppHandle, "Routing data transferred successfully.", "",
               MB_ICONINFORMATION);
```

Explanation of the parameters:

The "TargetSlotNr" is set to 2, because the CPU to be accessed is installed in slot 2 (as with all 300-type systems). Since the 315-PN/DP is installed in the first rack, the "TargetRackNr" is set to 0.
The "TargetMPI_DP_Address" parameter is not relevant for this example, since the 315-PN/DP is accessed via Ethernet. The IP address is specified by parameter "TargetIPAddressStr", this is the IP address of the 315-PN/DP.
The parameters "TargetSubnetID_High" and "TargetSubnetID_Low" must contain the S7 subnet ID of the network that connects the CPU being accessed with the master. In the example the 315-PN/DP is accessed using the Ethernet link that is also connected to the 315-2DP via the Ethernet CP. This has the Ethernet subnet ID "1122-3344", i.e. these values must be specified by the parameters.
Finally, the parameter "TargetNetIsMPI_DP_Net" must be allocated. In this example, this value must be set to 0, because the target network that is used to access the 315-PN/DP is an Ethernet network. This means that the IP address will be used and not the MPI/DP address.

Now the parameters are complete. This only leaves the last function MPI6_ConnectToPLCRouting to be called.

6.18.3 Call the function MPI6_ConnectToPLCRouting

The MPI6_ConnectToPLC function must not be used, since the connection will not be established with the CPU that is connected directly to the PC but with a CPU that is networked with this CPU. You must use the function MPI6_ConnectToPLCRouting to establish communications.
The example below shows this call:

```
BYTE AGMPIAdresse=8; //the MPI address of the directly connected CPU
if (!MPI6_ConnectToPLCRouting(MPIHandle, AGMPIAdresse,
                             &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
else {
     MessageBox(AppHandle, "Communications established successfully!",
"",                  MB_ICONINFORMATION);
}//end else
```

The CPU MPIAdresse must have the value 8, since this is the MPI address of CPU C313-DP to which the PC is connected. This CPU then forwards the request to the CPU that must be accessed.

6.18.4 Conclusion as to the routing example

This completes the calls, the 315-PN/DP can now be accessed by means of the read and write functions. In comparison with direct access, there are no differences when calling the functions. The example has demonstrated that the function MPI6_SetRoutingData followed by MPI6_ConnectToPLCRouting must be called after the respective initialisation functions.

When configuring the hardware of the CPUs it is important to note, that the routing data must also be transferred. In the Simatic?-Manager for example, this type of configuration must be performed by means of the NetPro. This is necessary to ensure that the CPUs are aware of the other CPUs will be accessible via them.

It should also be noted that the rate of communication when routing is less that that of a direct connection.

ComDrvS7 supports routing for the communication paths MHJ-NetLink, NetLink PRO, TCP/IP direct and SIMATIC®-NET.

6.19 The function: MPI6_ReadByte

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The function MPI6_ReadByte can be used to determine the status of input, output, flags and data block bytes.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the bytes must be read.
byteBufferPtr	BYTE*	This buffer is used to store the status information.
wCountBytes	WORD	It must be ensured that the area is sufficiently large to accommodate all the status information. The buffer must have as many fields as are required for the requested bytes.
wDBNR	WORD	Number of bytes to read. Up to a max. of 65535 bytes may be read with one call. If the bytes being read are the contents of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below retrieves the status information from the communication partner having MPI address 10 starting with memory byte 10. 30 bytes must be read.

```
int ComNr=2;           //COM2 port
long BaudRate=115200; //baud rate 115200
BYTE PGMPIAdresse=0;  //MPI address of the DLL application = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
bool SchnittstelleWarSchonAllokiert=false;
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
int MPIHandle=-1;     //handle of the new communication instance
BYTE AGMPIAdresse=10; //the MPI address of the CPU to be accessed
bool Fehler=false;
char AusgabeStr[255]={0};//string for text output
//establish connection
if (!MPI6_OpenRS232(&MPIHandle, ComNr, BaudRate, PGMPIAdresse,
                            HoechsteMPI,
                            &SchnittstelleWarSchonAllokiert,
                            &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Initialisation successful.", "",
          MB_ICONINFORMATION);
//establish communications
if (!MPI6_ConnectToPLC(MPIHandle, AGMPIAdresse, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     Fehler=true;
}//end if
else {
     MessageBox(AppHandle, "Communications established successfully!",
                  "", MB_ICONINFORMATION);
}//end else
//read data
if (!Fehler){
    BYTE StatusBuffer[100]={0};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_ReadByte(MPIHandle, Operand, 10, StatusBuffer,
                       30, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       wsprintf(AusgabeStr, "Clock memory 11 has the status: %02X",
                 StatusBuffer[1]);
       MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
```

```
//terminate communications
if (!MPI6_CloseCommunication(MPIHandle, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communications terminated without errors.", "",
                 MB_ICONINFORMATION);
```

6.20 The function: MPI6_ReadWord

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The function MPI6_ReadWord can be used to determine the status of input, output, flags and data block words.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!

This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the words must be read.
wordBufferPtr	WORD*	The status information is stored in this buffer. You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields as words that will be queried.
wCountWord	WORD	Number of words to read. Up to a max. of 65535 words may be read with one call.
wDBNR	WORD	If the words being read are the contents of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
It should be noted that the structure of the returned status values is as follows (example of word memory MW2):
Word memory 2 = clock memory 2 (HIBYTE) and clock memory 3 (LOBYTE)
It should also be noted that byte-oriented word operands overlap. For this reason, the function either reads all even or all odd words in the specified area. This depends on the value specified in "wAddress". If you specify an even number here (e.g. 10), all even-numbered words are read. If, for example, the value passed is 13, then all the odd words read.
In actual fact it only makes sense to read even-numbered words, the option was left open to cater for exceptions.

Example

In the example below, the status information of the even numbered word memories from word memory 0 are read from a communication partner. There are 5 words to be read. After the action, the status buffer contains the contents of word memories MW0, MW2, MW4, MW6 and MW8.

The example assumes that the initialisation functions (e.g. MPI6_OpenTcpIp) were completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function MPI6_CloseCommunication terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
    WORD StatusBuffer[100]={0};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_ReadWord(MPIHandle, Operand, 0, StatusBuffer,
                       5, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       char AusgabeStr[255]={0};
       wsprintf(AusgabeStr, "Word memory 2 has status (hex):
            %04X", StatusBuffer[1]);
       MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.21 The function: MPI6_ReadDword

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The function MPI6_ReadDword can be used to determine the status of input, output, flags (memory bits) and data block double words.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the double words must be read.
dwordBufferPtr	DWORD*	The status information is stored in this buffer.
wCountDword	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields as double words that will be queried.
wDBNR	WORD	Number of double words to read. Up to a max. of 65535 double words may be read with one call.
Error	WORD*	Number of double words to read. Up to a max. of 65535 double words may be read with one call. If the double words being read are the contents of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

It should be noted that the structure of the returned status values is as follows (example of double word memory MD2):
MD2 consists of word memories MW2 and MW4, where MW2 represents the Hi-word. MW2 in turn consists of the bytes MB2 and MB3. MW4 consists of the bytes MB4 and MB5. Thus, MD2 includes 4 bytes, i.e. MB2, MB3, MB4 and MB5.
It should also be noted that byte-oriented double word operands overlap. For this reason, the function reads either all even or all odd double words or all the odd double words in the specified area. This depends on the value specified in "wAddress". If you specify an even number here (e.g. 10), all even-numbered double words are read. If, for example, the value passed is 13, then all the odd double words read.
In fact, it only makes sense to read even-numbered double words, the option was left open to cater for exceptions.

Example

In the example below the status information of the even numbered double word memories from double word memory 0 are read from a communication partner. There are 5 double words to be read. After the action, the status buffer contains the contents of double word memories MD0, MD4, MD8, MD12 and MD16.

The example assumes that the initialisation functions (e.g. MPI6_OpenTcpIp) were completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function MPI6_CloseCommunication terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
    DWORD StatusBuffer[100]={0};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_ReadDword(MPIHandle, Operand, 0, StatusBuffer,
                       5, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       char AusgabeStr[255]={0};
       wsprintf(AusgabeStr, "MD4 has the status (hex):
                           %08X", StatusBuffer[1]);
       MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
```

6.22 The function: MPI6_ReadTimer (not in the Lite-version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_ReadTimer function can be used to determine the status of timer blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
wAddress	WORD	Starting address from which the timer must be read.
wordBufferPtr	WORD*	The status information is stored in this buffer.
wCountTimer	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields as timers that will be queried.
Error	WORD*	Number of timers to read. Up to a max. of 65535 timers may be read with one call. If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

The timer-word has the following structure:

Bit 0-9:	BCD-coded time factor
Bit 12+13:	time base (0=10ms, 1=100ms, 2=1s, 3=10s)

Example

In the example below, the status information of timers 0 to 4 is read from a communication partner.

The example assumes that the initialisation functions (e.g. MPI6_OpenTcpIp) were completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function MPI6_CloseCommunication terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
    WORD StatusBuffer[10]={0};
    if (!MPI6_ReadTimer(MPIHandle, 0, StatusBuffer, 5, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       char AusgabeStr[255]={0};
       wsprintf(AusgabeStr, "Timer 1 has status (hex):
                        %04X", StatusBuffer[1]);
       MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.23 The function: MPI6_ReadCounter (not in the Lite-version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_ReadCounter function can be used to determine the status of counter blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
wAddress	WORD	Starting address from which the counter must be read.
wordBufferPtr	WORD*	The status information is stored in this buffer.
wCountCounter	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields as counters that will be queried.
Error	WORD*	Number of counters to read. Up to a max. of 65535 counters may be read with one call. If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

The strcture of the counter word is as follows:
Bit 0-9: BCD-coded counter reading

Example

In the example below the status information of timers 0 to 4 is read from a communication partner.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others.
Executing the function MPI6_CloseCommunication will terminate communications and eliminate the instance.
In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
    WORD StatusBuffer[10]={0};
    if (!MPI6_ReadCounter(MPIHandle, 0, StatusBuffer, 5, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       char AusgabeStr[255]={0};
       wsprintf(AusgabeStr, "Counter 1 has the status (hex):
                              %04X", StatusBuffer[1]);
       MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.24 The function: MPI6_MixRead_2

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_MixRead_2 function can be used to read the status of the operands in areas E, A, M, DB, T and Z.
Here, the status of different operands can read by means of a single call and in no particular order. For example, it is possible to read the status of input word EW2, clock memory MB10, data word 2 of DB10 (DB10.DBW2) and timer block T2 by means of a call to the MPI6_MixRead_2 function.
The function automatically optimises the request. Overlapping operands, duplicate requests, etc. are detected and the protocol to the CPU is optimised accordingly.

This function can be used to read the status from different operand areas when different addresses of an operand area must be read (e.g. MW0, MW100, MB150, etc.).

Restrictions of the Lite version

The Lite version can only read data from data blocks.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).

Op_Type	DWORD*	Array containing the types of the operands that must be read.
		memory bits (flag) = 0x0101
		Input = 0x1101
		Output = 0x2101
		Timer = 0x5401
		Counter = 0x6401
		DB data = 0x7101

Op_Address	DWORD*	Array containing the addresses of the operands that must be read.
DBNr	DWORD*	Array containing the DB numbers when an operand consists of a DB-data. For non-DB-data, the contents of the respective Index=0.
Op_LengthByte	DWORD*	Array containing the respective lengths of the operands in bytes. Permitted values are 1, 2 and 4.
Data	DWORD*	Array containing the returned status values for the operands.
wCountParam	WORD	The number of operands specified in the arrays.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below requests the status information of the operands MB10, DB2.DBW0 and DB1.DBD100 from a communication partner.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others.
Executing the function **MPI6_CloseCommunication** will terminate communications and eliminate the instance. This takes place in the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //create the arrays
 DWORD Op_Type[3];
 DWORD Op_Address[3];
 DWORD Op_LengthByte[3];
 DWORD DBNr[3];
 DWORD Data[3];
 //enter MB10 into array index 0
 Op_Type[0]=0x0101; //clock memory
 Op_Address[0]=10;  //address 10
 Op_LengthByte[0]=1; //length 1 byte
 DBNr[0]=0; //DB number=0 since this is not DB data
 //DB2.DBW0 im Array-Index 1 eintragen
 Op_Type[1]=0x7101; //DB data
 Op_Address[1]=0;  //address 0
 Op_LengthByte[1]=2; //length 2 byte = 1 word
 DBNr[1]=2; //DB number=2
 //enter DB1.DBD100 into array-index 2
 Op_Type[2]=0x7101; //DB data
 Op_Address[2]=100;  //address 100
 Op_LengthByte[2]=4; //length 4 bytes = 1 double word
 DBNr[2]=1; //DB number=1
 //the number of operands to read
 WORD wCountParam=3; //3 operands
 //call to the MixRead function
 if (!MPI6_MixRead_2(MPIHandleV6, Op_Type, Op_Address, DBNr,
                    Op_LengthByte, Data, wCountParam, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     char AusgabeStr[255]={0};
     wsprintf(AusgabeStr, "DB2.DBW0 has the status(hex): %04X",
             LOWORD(Data[1]));
     MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
 }//end else
 //
.
.
.
```

6.25 The function: MPI6_WriteBit_2

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteByte function can control the value of input, output, flags (memory bist) and data block bits. This means that the operands can be set to the value passed by the function.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Note

The use of the MPI6_WriteBit_2 function is inefficient; it should only be used in exceptional cases. The function should only be used when it is important to control a single bit without affecting the other bits of the byte. In all other cases, the WriteByte, WriteWord or WriteDword functions are preferable.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wByteAddress	WORD	Byte address of the bit operand. Example: 10 for M10.3
bBitAddress	BYTE	Byte address of the bit operand. Example: 3 for M10.3
bValue	BYTE	Specifies the control value. Permitted values are 0 and 1.
wDBNR	WORD	If the bit being controlled is a bit of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below sets memory bit M10.3 in the communication partner to the value 1. The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    BYTE SteuerWert=1;
    WORD wByteAddress=10;
    BYTE bBitAddress=3;
    WORD wDBNR=0;
    BYTE Operand=77; //clock memory ASCII-Code 77

    if (!MPI6_WriteBit_2(MPIHandle, Operand, wByteAddress,
                         bBitAddress, SteuerWert, wDBNR, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.26 The function: MPI6_WriteByte

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteByte function can control the value of input, output, flags (memory bits) and data block bytes. This means that the operands can be set to the value passed by the function.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the bytes must be written.
byteBufferPtr	BYTE*	This is the buffer where the control information is stored.
wCountBytes	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information that must be written. The buffer must have the same number of fields as control bytes that were specified.
wDBNR	WORD	Number of bytes to written. Up to a max. of 65535 bytes may be written with one call. If the bytes being written are the contents of a data block, the number of the respective DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

In the example below, clock memory bytes 0 to 9 in the communication partner are set to the value FF (hex).
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. This takes place in the same manner as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    BYTE SteuernBuffer[10]={0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                            0xFF, 0xFF, 0xFF};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_WriteByte(MPIHandle, Operand, 0, SteuernBuffer,
                       10, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.27 The function: MPI6_WriteWord

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteWord function can control the value of input, output, flag (memory bit) and data block words. This means that the operands can be set to the value passed by the function.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the words must be written.
wordBufferPtr	WORD*	The control information is stored in this buffer.
wCountWord	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have the same number of fields as control words were specified.
wDBNR	WORD	Number of words to written. Up to a max. of 65535 words may be written with one call. If the words being written are the contents of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

It should be noted that the values specified in wordBufferPtr are written as follows (for example, clock memory MW2):

MB 2 = HIBYTE(wordBufferPtr[0])
MB 3 = LOBYTE(wordBufferPtr[0])

It should also be noted that byte-oriented word operands overlap. For this reason, the function either writes to all even or all odd words in the specified area. This depends on the value specified in "wAddress". If you specify an even number here (e.g. 10), all even-numbered words will be written. If the value passed is 13, all the odd words are written.
In fact, it only makes sense to write even-numbered words, the option was left open to cater for exceptions.

Example

In the example below, clock memory words 30 to 38 in the communication partner are set to the value 1F00(hex).
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. When the **MPI6_CloseCommunication** function was executed, communications can be terminated and the instance removed. This takes place in the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    WORD SteuernBuffer[5]={0x1F00, 0x1F00, 0x1F00, 0x1F00, 0x1F00};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_WriteWord(MPIHandle, Operand, 30, SteuernBuffer,
                       5, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.28 The function: MPI6_WriteDword

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteDword function can control the value of input, output, flag (memory bit) and data block double words. This means that the operands can be set to the value passed by the function.
The **Lite-Version** only provides access to data blocks.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
operand	BYTE	The ASCII code for letters E, A, M, D defines the operand area, which you want to read. Inputs = 69, outputs = 65, memory bits (flags) = 77, data blocks = 68, VM area (only MICRO-version) = 86.
wAddress	WORD	Starting address from which the double words must be written.
dwordBufferPtr	DWORD*	The control information is stored in this buffer.
wCountDword	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have the same number of fields as control double words were specified.
wDBNR	WORD	Number of double words to written. Up to a max. of 65535 double words may be written with one call. If the words being written are the contents of a data block, the number of the DB (1-65535) must be specified here. Otherwise, enter 0.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

It should be noted that the values specified in dwordBufferPtr are written as follows (for example, double word memory MD2):

MB 2 = HIBYTE(HIWORD(dwordBufferPtr[0]))
MB 3 = LOBYTE(HIWORD(dwordBufferPtr[0]))
MB 4 = HIBYTE(LOWORD(dwordBufferPtr[0]))
MB 5 = LOBYTE(LOWORD(dwordBufferPtr[0]))

It should also be noted that byte-oriented word operands overlap. For this reason, the function reads either all even double words or all the odd double words in the specified area. This depends on the value specified in "wAddress". If you specify an even number here (e.g. 10), all even-numbered double words are written. If the value passed is 13, all the double odd words are written.
In fact, it only makes sense to write even-numbered double words, the option was left open to cater for exceptions.

Example

In the example below, memory double words 10 and 14 in the communication partner are set to the value 11223344(hex).
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    DWORD SteuernBuffer[2]={0x11223344, 0x11223344};
    BYTE Operand=77; //clock memory ASCII-Code 77
    if (!MPI6_WriteDword(MPIHandle, Operand, 10, SteuernBuffer,
                        2, 0, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.29 The function: MPI6_WriteTimer (Not in Lite-Version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteTimer function can be used to control the value of timer blocks. This means that the timer can be set to the value passed by the function.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
wAddress	WORD	Starting address of the timer that must be controlled.
wordBufferPtr	WORD*	The control information is stored in this buffer.
wCountTimer	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields, as timers were specified to be controlled. Number of timers to write. Up to a max. of 65535 timers may be written with one call.
Error	WORD*	If the function returns '0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

The timer-word has the following structure:

Bit 0-9:	BCD-coded time factor
Bit 12+13:	time base (0=10ms, 1=100ms, 2=1s, 3=10s)

Example

The example below writes the value 100 (hex) to timers 3-9 in the communication partner. The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    WORD SteuernBuffer[7]={0x100, 0x100, 0x100, 0x100, 0x100, 0x100,
                           0x100};
    if (!MPI6_WriteTimer(MPIHandle, 3, SteuernBuffer, 7, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.30 The function: MPI6_WriteCounter (Not in Lite-Version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteCounter function can be used to control the value of counter blocks. This means that the counter can be set to the value passed by the function.
If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
wAddress	WORD	Starting address of the counter that must be controlled.
wordBufferPtr	WORD*	The control information is stored in this buffer.
wCountCounter	WORD	You must ensure that the size of the buffer is sufficient to accommodate all the status information. The buffer must have as many fields, as counters were specified to be controlled. Number of counters to write. Up to a max. of 65535 counters may be written with one call.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:

The structure of the counter word is as follows:
Bit 0-9: BCD-coded counter value

Example

The example below writes the value 10 (hex) to timers 10 to 21 in the communication partner. The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    WORD SteuernBuffer[12]={0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
                           0x10, 0x10, 0x10, 0x10, 0x10, 0x10};
    if (!MPI6_WriteCounter(MPIHandle, 10, SteuernBuffer, 12
                          &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Control was successful!", "",
             MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.31 The function: MPI6_MixWrite_2

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_MixWrite_2 function can be used to control the operands in the areas E, A, M, DB, T and Z, i.e. to set these to the specified value.
Here, the control of different operands can occur by means of a single call and in no particular order. For example, it is possible to control input word IW2, flags (memory bits) MB10, data word 2 of DB10 (DB10.DBW2) and timer block T2 by means of a single call to the MPI6_MixWrite_2 function.
The function automatically optimises the control request. Overlapping operands, duplicate requests, etc. are detected and the protocol to the CPU is optimised accordingly.

This function can be used to control different operand areas when different addresses of an operand area must be written (e.g. MW0, MW100, MB150, etc.).

Restrictions of the Lite version

The Lite version can only control data from data blocks.

If the CPU is protected by a password, this does not have to be transferred.

PLC family S7-1500®, S7-1200® and LOGO!
This function can also be used for the CPUs of the S7-1500®, S7-1200® series. However, DBs may not have been generated with the option "only symbolically addressable".
There are no Data-blocks in the LOGO!. With a LOGO!® you have access to inputs, outputs, flags and VM area. The access to LOGO! is only possible in the MICRO version of ComDrvS7.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).

Op_Type	DWORD*	Array containing the types of the operands that must be controlled.
		memory bit (flag) = 0x0101
		Input = 0x1101
		Output = 0x2101
		Timer = 0x5401
		Counter = 0x6401
		DB data = 0x7101

Op_Address	DWORD*	Array containing the addresses of the operands that must be controlled.
DBNr	DWORD*	Array containing the DB numbers when an operand consists of a DB-data. For non-DB-data, the contents of the respective Index=0.
Op_LengthByte	DWORD*	Array containing the respective lengths of the operands in bytes. Permitted values are 1, 2 and 4.
Data	DWORD*	Array with the control values for the operands.
wCountParam	WORD	The number of operands specified in the arrays.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below controls the operands MB10, DB2.DBW0 DB1.DBD100 in the CPU to set them to the target values.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 DWORD Op_Type[3];
 DWORD Op_Address[3];
 DWORD Op_LengthByte[3];
 DWORD DBNr[3];
 DWORD Data[3];
 //MB10: enter data into array index 0
 Op_Type[0]=0x0101; //clock memory
 Op_Address[0]=10;  //address 10
 Op_LengthByte[0]=1; //length 1 byte
 DBNr[0]=0; //DB number=0 since this is not DB data
 Data[0]=0x33; //control value
 //DB2.DBW0: enter data into array index 1
 Op_Type[1]=0x7101; //DB data
 Op_Address[1]=0;  //address 0
 Op_LengthByte[1]=2; //length 2 byte = 1 word
 DBNr[1]=2; //DB number=2
 Data[1]=0x1122; //control value
 //DB1.DBD100: enter data into array index 2
 Op_Type[2]=0x7101; //DB data
 Op_Address[2]=100;  //address 100
 Op_LengthByte[2]=4; //length 4 bytes = 1 double word
 DBNr[2]=1; //DB number=1
 Data[2]=0x55667788; //control value
 //Number of operands to be controlled
 WORD wCountParam=3; //3 operands
 //call to the MixWrite function
 if (!MPI6_MixWrite_2(MPIHandleV6, Op_Type, Op_Address, DBNr,
                     Op_LengthByte, Data, wCountParam, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(AppHandle, "Control executed successfully.", "",
                     MB_ICONINFORMATION);
 }//end else
 //
.
.
.
```

6.32 The function: MPI6_WriteBit (not in the Lite-version)

This function is only available for compatibility reasons. For new applications, the MPI6_WriteBit_2 function should be used!

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteBit function can be used to change the contents of a bit in operand areas input, output, clock memory and data.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Note

The use of the MPI6_WriteBit function is ineffective; it should only be used in exceptional cases. The function should only be used when it is important to control a single bit without affecting the other bits of the byte. In all other cases, the WriteByte, WriteWord or WriteDword functions are preferable.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
ByteAddress	WORD	Indicates the byte address where the bit operand is located. Specify e.g. '0'
BitAddress	WORD	Specifies the bit address that mast be accessed. Range 0 - 7.
DBNr	WORD	When a data bit is affected, the DBNummer specifies the number of the data block that contains the data bit. If this specification does not refer to a data bit, then the DBNummer must be set to '0'.
Operand	char	These parameters must pass the operand in uppercase letters, where: "E" = inputs, "A" = outputs, "M" = memory bits (flags), "D" = data bits in DB
WriteBuffer	WORD*	Specify the control value (0 or 1) in this buffer.
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below sets memory bit M10.1 in the communication partner to the value 1. The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//write data
if (!Fehler){
    WORD SteuerWert=1;
    if (!MPI6_WriteBit(MPIHandle, 10, 1, 0, 'M', &SteuerWert,
                      &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
    }//end if
    else {
       MessageBox(AppHandle, "Operand M10.1 was controlled"
         "successfully!", "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.33 The function: MPI6_WriteDBFromWldToPlc (not in the Lite- and CE-versions)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WriteDBFromWldToPlc function can be used to
transfer a data block that is contained in a WLD file into the CPU. WLD files are created and filled with blocks by S7 programming systems (e.g. Simatic-Manager and WinSPS-S7). The S7 programming systems are also able to read and edit modules from WLD files.
Together with the MPI6_ReadDBFromPlcAndWriteToWld function, you can load ComDrvS7 data blocks from the CPU, save them on the PC and write them to the CPU if required.
In addition to backing up data, this can also be used to realize a type of recipe management.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
WldFileWithPath	char*	Specification of the WLD file with path from which the DB must be retrieved to transfer it to the CPU. For example, "C:\ProjektDBs.WLD"
DBNr	WORD	The number of the DB that must be transferred to the CPU.
OverwriteIfExist	BYTE	This parameter determines whether a DB that already exists in the CPU should be overwritten. If this is set to 1, the DB is overwritten . When this is set to 0, a DB that exists in the CPU is not overwritten and the function returns error value 713.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example reads DB2 from a WLD file that is named "Test_Datei.wld" and that is located in the path "D:\MPI6_Testprojekt\" and transfers this to the CPU. If DB2 should already exist in the CPU, then this must be overwritten.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 WORD Error=0;
 char ErrorString[255]={0};
 char WldFileWithPath[555]={0};
 BYTE OverwriteIfExist=1; //overwrite a DB that exists
                          //in the CPU
 WORD wDBNR=2; //DB2 must be transferred
 //specify the WLD file with path
 strcpy(WldFileWithPath, "D:\\MPI6_Testprojekt\\Test_Datei.wld");
 //Execute the transfer of the DB from the WLD file into the CPU
 if (!MPI6_WriteDBFromWldToPlc(MPIHandleV6, WldFileWithPath, wDBNR,
                              OverwriteIfExist, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(AppHandle, "DB was written successfully.", "",
         MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.34 The function: MPI6_ReadDBFromPlcAndWriteToWld (Not in the Lite- and CE-Version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_ReadDBFromPlcAndWriteToWld function can be used to load a data block from a CPU to save it to a WLD file on your PC.
The S7 programming systems (e.g. Simatic?-Manager or WinSPS-S7) can read and edit modules from WLD files.
Together with the MPI6_WriteDBFromWldToPlc function, you can load ComDrvS7 data blocks from the CPU, save them on the PC and write them back to the CPU if required.
In addition to backing up data, this can also be used to realize a type of recipe management.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
WldFileWithPath	char*	Specification of the WLD file with path to which the DB retrieved from the CPU must be written. The path must exist; the WLD file is created if it does not exist. For example, "C:\ProjektDBs.WLD" For an existing WLD file must not contain a DB with the same number. If this is the case, the function returns the error value 717.
DBNr	WORD	Number of DB that is to be loaded from the CPU and written to the WLD file.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example reads DB2 from the CPU and writes it to a WLD file that is named "Test_Datei.wld" and that is located in the path "D:\MPI6_Testprojekt\".

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 WORD Error=0;
 char ErrorString[255]={0};
 char WldFileWithPath[555]={0};
 WORD wDBNR=2; //DB2 must be retrieved from the CPU
 //specify the WLD file with path
 strcpy(WldFileWithPath, "D:\\MPI6_Testprojekt\\Test_Datei.wld");
 //Read the DBs from the CPU and save it to the WLD file
 if (!MPI6_ReadDBFromPlcAndWriteToWld(MPIHandleV6, WldFileWithPath,
                                    wDBNR, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(Application->Handle, "Read the DBs from the CPU"
                "and saving to the WLD was successful.", "",
             MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.35 The function: MPI6_GetDBNrInWldFile (Not in the Lite- and CE-Version)

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetDBNrInWldFile function can be used to identify data blocks that exist in a WLD file.
WLD files are created and filled with blocks by S7 programming systems (e.g. Simatic-Manager and WinSPS-S7). The S7 programming systems are also able to read and edit modules from WLD files.
The MPI6_ReadDBFromPlcAndWriteToWld and MPI6_WriteDBFromWldToPlc functions you can load ComDrvS7 data blocks from a CPU, save them on the PC and write them back to the CPU if required.
In addition to backing up data, this can also be used to realize a type of recipe management.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
WldFileWithPath	char*	Specifies the WLD-file with its path in which the existing DB must be determined. This file must exist.
DBNrArray	WORD*	In this array, the numbers of the DBs in the WLD file are returned. The array must have as many or more fields as are specified in the parameter MaxDBNumbers.
wcountDB	WORD*	Number of data blocks found in the WLD file.
MaxDBNumbers	WORD	Maximum number of DB numbers to be returned. The array DBNrArray must at least have been designed for this number of DBs.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below determines the existing data blocks from the WLD file named "Test_Datei.wld" which is located in the path "D:\MPI6_Testprojekt\".

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was exexuted successfully.

```
.
.
.
 //
 WORD Error=0;
 char ErrorString[255]={0};
 char WldFileWithPath[555]={0};
//specify the WLD file with path
 strcpy(WldFileWithPath, "D:\\MPI6_Testprojekt\\Test_Datei.wld");
 //Read the DBs from the CPU and save it to the WLD file
 WORD wcountDB=0; //this variable returns the number of DBs
 WORD DBNrArray[100]; //array for existing DB numbers
 WORD MaxDBNumbers=100; //the array for a max. of 100 DB numbers
 if (!MPI6_GetDBNrInWldFile(MPIHandleV6, WldFileWithPath, DBNrArray,
                           &wcountDB, MaxDBNumbers, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     char AusgabeStr[255]={0};
     wsprintf(AusgabeStr, "The WLD file contains %u DBs.",
   wcountDB);
     MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
 }//end else
 //
.
.
.
```

6.36 The function: MPI6_ReadPlcClock

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_ReadPlcClock function can be used to read the current date and time from the CPU. The date and time is returned in the format Date-and-Time. In addition, a string specifying the function will be provided.

The structure of the Date-and-Time format is as follows:

Byte position	Description
n	year 0-99
n+1	month 1-12
n+2	day 1-31
n+3	hour 0-23
n+4	minute 0-59
n+5	second 0-59
n+6	millisecond 0-999 + weekday (1-7)
n+7	

All data is in BCD coded. Weekday: 1=Sunday, 7=Saturday.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
byteBufferPtr	BYTE*	Array that returns the date and time from the CPU in the format Date-and-Time. The length of this array must be 8 bytes or more.
DtString	char*	This supplies the date and the time as a string of the form "2009-02-25-21:58:11.478".
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below reads the date and time from the CPU.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 BYTE byteBufferPtr[8]={0}; //buffer for the format Date-and-Time
 char DTStr[255]={0}; //returned setting as string
 //
 if (!MPI6_ReadPlcClock(MPIHandleV6, byteBufferPtr, DTStr, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     char AusgabeStr[255]={0};
     wsprintf(AusgabeStr, "Datum Uhrzeit in der CPU: %s", DTStr);
     MessageBox(ApplHandle, AusgabeStr, "", MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.37 The function: MPI6_WritePlcClock

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In addition, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_WritePlcClock function can be used to set the date and time in the CPU.
The date and time must be specified in the format Date-and-Time.

The structure of the Date-and-Time format is as follows:

Byte position	Description
n	year 0-99
n+1	month 1-12
n+2	day 1-31
n+3	hour 0-23
n+4	minute 0-59
n+5	second 0-59
n+6	millisecond 0-999 + weekday (1-7)
n+7	

All data is in BCD coded. Weekday: 1=Sunday, 7=Saturday.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
byteBufferPtr	BYTE*	Array where the date and time for the CPU must be specified in the format Date-and-Time. The length of this array must be 8 bytes or more.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

In the example below, the date in the CPU is set to 31.05.2009 and the time to 12:33, 10 seconds and 333 milliseconds.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 BYTE byteBufferPtr[8]={0};
 byteBufferPtr[0]=0x09; //year 2009
 byteBufferPtr[1]=0x05; //month 5
 byteBufferPtr[2]=0x31; //day 31
 byteBufferPtr[3]=0x12; //hour 12
 byteBufferPtr[4]=0x33; //minute 33
 byteBufferPtr[5]=0x10; //second 10
 byteBufferPtr[6]=0x33; //milliseconds 333 specification 1
 byteBufferPtr[7]=0x31; //milliseconds 333 specifiaction 2 day = 1
 //
 if (!MPI6_WritePlcClock(MPIHandleV6, byteBufferPtr, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "Date and time was saved.", "",
         MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.38 The function: MPI6_CopyRamToRom

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_CopyRamToRom function can be used to transfer the actual values of the data blocks from the CPUs memory into the load memory. This means that these values are retained even when you issue a master reset to the CPU.
For example, this function may be executed if you have changed the values in a DB by means of PLC commands or by control functions via ComDrvS7 and if these values should remain active after a master reset.

This function can only be executed when the CPU is in STOP mode. If the CPU is not in STOP mode, you can set it to STOP mode with the MPI6_SetPLCToStop function.

The execution of this function may require a few minutes (depending on available memory).

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below executes the function copy RAM-to-ROM in the CPU. To start with, the CPU must be in STOP mode.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //test whether CPU is in RUN mode
 bool PlcInRun=false;
 if (!MPI6_IsPLCInRunMode(MPIHandleV6, &PlcInRun, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
 }//end if
 //
 if (PlcInRun){
     MessageBox(ApplHandle, "Action not valid in RUN mode!", "",
                MB_ICONINFORMATION);
     return;
 }//end if
 //
 if (!MPI6_CopyRamToRom(MPIHandleV6, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "The action was executed.", "",
               MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.39 The function: MPI6_PLCHotRestart or MPI6_CPUWiederanlauf

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_PLCHotRestart and MPI6_CPUWiederanlauf functions issue a restart command to the CPU. The condition is that the CPU supports a restart (S7-400 with the appropriate hardware configuration) and that the mode selector is in position RUN.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

Im folgenden Example wird bei der CPU ein Wiederanlauf ausgelöst.
Im Example wird davon ausgegangen, dass eine der Einleitungsfunktionen (z.B. MPI6_OpenTcpIp) erfolgreich ausgeführt wurde. Ebenso muss die Funktion **MPI6_ConnectToPLC** ohne Fehler ausgeführt worden sein. Nach der Aktion können weitere folgen. Durch Ausführen der Funktion **MPI6_CloseCommunication** kann die Kommunikation beendet und die Instanz beseitigt werden. So wie dies im Example für die Funktion MPI6_ReadByte gezeigt wurde.

```
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 if (!MPI6_CPUWiederanlauf(MPIHandleV6, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "Aktion wurde ausgeführt.", "",
               MB_ICONINFORMATION);
 }//end else
.
```

6.40 The function: MPI6_PLCWarmRestart or MPI6_CPUNeustart

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_PLCWarmRestart or MPI6_CPUNeustart functions result in a restart of the CPU, provided that the mode selector is in position RUN.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below triggers a restart of the CPU.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others.
Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 if (!MPI6_CPUNeustart(MPIHandleV6, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "Action was executed.", "",
           MB_ICONINFORMATION);
 }//end else
.
```

6.41 The function: MPI6_SetPLCToStop

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_SetPLCToStop functions change the status of the CPU to STOP.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
Error	WORD*	If the function returns '0', an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

In the example below, the status of the CPU is set to STOP.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully.
Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
 char ErrorString[255]={0};
 WORD Error=0;
 //
 if (!MPI6_SetPLCToStop(MPIHandleV6, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "Action was executed.", "",
           MB_ICONINFORMATION);
 }//end else
.
```

6.42 The function: MPI6_IsPLCInRunMode

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_IsPLCInRunMode function determines the status of the CPU and returns a value '1' in parameter PlcInRun if the CPU is in RUN mode.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PlcInRun	BOOL*	If this parameter contains '1', the CPU is in RUN mode.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below tests whether the CPU is in RUN mode.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //Ist die CPU in RUN
 bool PlcInRun=false;
 if (!MPI6_IsPLCInRunMode(MPIHandleV6, &PlcInRun, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
 }//end if
 //
 if (PlcInRun){
     MessageBox(ApplHandle, "CPU ist in RUN mode!", "",
                   MB_ICONINFORMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "CPU is not in RUN mode!", "",
                MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.43 The function: MPI6_GetSystemValues

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetSystemValues function can be used to read the system areas from a CPU. For example, this can provide information on the number of timers, counters and clock memories that are available for programming.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
CountByteProcessImageInputs	WORD*	This returns the number of bytes of the process image of the inputs.
CountByteProcessImageOutputs	WORD*	This returns the number of bytes of the process image of the outputs.
CountByteBitMemory	WORD*	This returns the number of memory bytes that are available in the connected PLC.
CountTimer	WORD*	This returns the number of timers that are available in the connected PLC.
CountCounter	WORD*	This returns the number of counters that are available in the connected PLC.
CountByteWorkMemory	DWORD*	Returns the number of memory bytes in the CPU. **Attention: In the old versions of ComDrvS7, this parameter had the type WORD, i.e. it was only 16 bit wide!**
CountByteLocalData	WORD*	Returns the entire local data area of the CPU in bytes.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below reads the system areas from the communication partner.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     WORD AnzahlBytePAE, AnzahlBytePAA, AnzahlMerker, AnzahlZeiten;
     WORD AnzahlZaehler;
     DWORD AnzahlByteRam;
     WORD AnzahlByteLokaldaten;
     //
     if (!MPI6_GetSystemValues(MPIHandle, &AnzahlBytePAE,
                              &AnzahlBytePAA,
                                &AnzahlMerker, &AnzahlZeiten,
                               &AnzahlZaehler,
                               &AnzahlByteRam,
                               &AnzahlByteLokaldaten,
                                &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
          wsprintf(AusgabeStr,
                "Number of bytes PAE: %03u\n"
                "Number of bytes PAA: %03u\n"
                "Number of clock memories: %03u\n"
                "Number of timers   : %03u\n"
                "Number of counters : %03u\n",
                AnzahlBytePAE, AnzahlBytePAA, AnzahlMerker,
                AnzahlZeiten, AnzahlZaehler);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONEXCLAMATION);
    }//end else
}//end if
.
.
.
```

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, A communication error may result.

6.44 The function: MPI6_GetLevelOfProtection

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetLevelOfProtection function can be used to determine the protection levels that were defined for a CPU. The position of the key switch on the CPU is also returned.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
LevelModeSwitch	WORD*	Protection level on the mode selector/key switch (possible values??: 1, 2 or 3)
ParameterizedProtectionLevel	WORD*	configured protection level (possible values: 0, 1, 2 or 3. 0 means that no password was assigned, the configured protection level is invalid)
LevelPlc	WORD*	Valid protection level of the CPU (possible values: 1, 2 or 3)
ModeSwitchPosition	WORD*	Returns the position of the mode selector/key switch of the CPU. Possible contents: 0=STOP, 1=RUN, 2=RUN-P.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, A communication error may result.

Example

The example below transfers the position of the key switch the communication partner as a string.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** will terminates communications and eliminate the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     WORD SchutzstufeSchluesselSchalter, ParametrierteSchutzstufe;
     WORD CPUSchutzstufe;
     WORD SchluesselschalterStellung=0;
     //
     if (!MPI6_GetLevelOfProtection(MPIHandle,
                             &SchutzstufeSchluesselSchalter,
                             &ParametrierteSchutzstufe,
                        &CPUSchutzstufe,
                       &SchluesselschalterStellung,
                      &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         switch(SchluesselschalterStellung){
                case 0: MessageBox(AppHandle, "Switch position:
                                 STOP", "", MB_ICONEXCLAMATION);
                       break;
                case 1: MessageBox(AppHandle, "Switch position:"
                                 "RUN", "", MB_ICONEXCLAMATION);
                        break;
                case 2: MessageBox(AppHandle, "Switch position:"
                                  "RUN-P", "", MB_ICONEXCLAMATION);
                                  break;
                default: MessageBox(AppHandle, "Switch position:"
                                   "unknown", "",
                                   MB_ICONEXCLAMATION);
         }//end switch
    }//end else
}//end if
.
.
.
```

6.45 The function: MPI6_GetOrderNrPlc

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetOrderNrPlc function can be used to determine the order number of a CPU.

LOGO!

This function is not possible.

Description of the parameters

Argument	Typ	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
OrderNrStr	char*	Here, the order number of the CPU is returned as a null-terminated string.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel".
If another communication instance should also accesses an information function of the same CPU, A communication error may result.

Example

The example shown below determines the order number of a CPU.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others.
Executing the function **MPI6_CloseCommunication** will terminates communications and eliminate the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     char BestellNr[100]={0};
     //
     if (!MPI6_GetOrderNrPlc(MPIHandle, BestellNr, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Order no.: %s", BestellNr);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.46 The function: MPI6_CanPlcSendIdentData

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_CanPlcSendIdentData function can be used to determine whether the connected CPU is capable of supplying identification data. For this reason, the function should be executed before the MPI6_GetPlcIdentData function if it cannot be guaranteed that the connected CPU supports this feature.
The Siemens S7-300? series of CPUs support this feature from firmware version 2.6.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	Typ	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PlcCanSendData	bool*	Returns a value 1, if the CPU supports the request for identification data. Otherwise, the value 0 is returned.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, A communication error may result.

Example

See the example on the MPI6_GetPlcIdentData function

6.47 The function: MPI6_GetPlcIdentData

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

If it is doubtful that the connected CPU can supply the identification data, the MPI6_CanPlcSendIdentData function must be executed first. The Siemens S7-300? series of CPUs support this feature from firmware version 2.6.

Brief description

The MPI6_GetPlcIdentData function returns the unique serial number of the connected CPU as well as the serial number of the MMC card that is available in the CPU. This enables you to identify the connected CPU clearly. It is also possible to read the text items that may be defined in the hardware configuration of the CPU, i.e. the CPU name, the station name, the plant identification and the location identifier.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PlcName	char*	Returns the station name defined in the hardware configuration of the CPU or, alternatively, an empty string if this was not defined.
ModuleName	char*	Returns the CPU name defined in the hardware configuration of the CPU or, alternatively, an empty string if this was not defined.
PlantDesignation	char*	Returns the system identification that was defined in the hardware configuration for the CPU or alternatively, an empty string, if this was not defined.
LocationIdentifier	char*	Returns the location identification that was defined in the hardware configuration for the CPU or, alternatively, an empty string, if this was not defined.
SerialNrPlcStr	char*	Returns the serial number of the connected CPU.
MMCIdentNrStr	char*	Returns the serial number of the MMC-card that was installed in the CPU.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, a communication error may result.

Example

The example below reads the identification data from the CPU, if the CPU supports this function. First, the CPU is checked whether it supports this function and, on a positive response, the data is loaded from the CPU.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
bool AbfrageMoeglich=false;
if (!MPI6_CanPlcSendIdentData(MPIHandle, &AbfrageMoeglich, &Error)){
    //display the error(s)
    MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
    MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
//
if (AbfrageMoeglich){
   char NameDerStation[100]={0};
   char NameDerCPU[100]={0};
   char AnlagenKennzeichnung[100]={0};
   char OrtsKennzeichnung[100]={0};
   char SerienNummerCPU[100]={0};
   char MMCIdentNr[100]={0};
   //
   if (!MPI6_GetPlcIdentData(MPIHandle, NameDerStation, NameDerCPU,
                             AnlagenKennzeichnung, OrtsKennzeichnung,
                             SerienNummerCPU, MMCIdentNr, &Error)){
       //display the error(s)
       MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
       MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
   }//end if
   else {
       char AusgabeStr[255]={0};
       wsprintf(AusgabeStr, "Serial number of the CPU: %s",
                   SerienNummerCPU);
       MessageBox(Handle, AusgabeStr, "", MB_ICONINFORMATION);
   }//end else
}//end if
.
.
.
```

6.48 The function: MPI6_GetPlcErrorLED

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In addition, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetPlcErrorLED function returns the status of the error LEDs SF, BF1 and BF2. This can be used to determine whether a system error or a bus error is present at the CPU. The programmer can then respond accordingly in the PC program. In spite of these errors, the CPU can be in RUN mode if the PLC program contains the respective error OBs.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
SF_LED_Status	BYTE*	This contains a value 1 if the SF LED on the CPU is on or if it flashes.
SF_LED_FlashingFrequency	BYTE*	0 = steady light, when the SF_LED_Status contains 1 1 = flashes normally at 2 Hz 2 = flashes slowly at 0.5 Hz
BUS1F_LED_Status	BYTE*	This contains a value 1 if the BF1 LED on the CPU is on or if it flashes.
BUS1F_LED_FlashingFrequency	BYTE*	0 = permanently on when BUS1F_LED_Status is set to 1 1 = flashes normally at 2 Hz 2 = flashes slowly at 0.5 Hz
BUS2F_LED_Status	BYTE*	This contains a value 1, if the BF2 LED on the CPU is on or if it flashes.
BUS2F_LED_FlashingFrequency	BYTE*	0 = permanently on when BUS2F_LED_Status is set to 1 1 = flashes normally at 2 Hz 2 = flashes slowly at 0.5 Hz
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, a communication error may result.

Example

The example below reads the status of the error LEDs on the CPU.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
BYTE SF_LED_Status=0;
BYTE SF_LED_BlinkFrequenz=0;
BYTE BUS1F_LED_Status=0;
BYTE BUS1F_LED_BlinkFrequenz=0;
BYTE BUS2F_LED_Status=0;
BYTE BUS2F_LED_BlinkFrequenz=0;
//
if (!MPI6_GetPlcErrorLED(MPIHandle,
                       &SF_LED_Status, &SF_LED_BlinkFrequenz,
                       &BUS1F_LED_Status, &BUS1F_LED_BlinkFrequenz,
                       &BUS2F_LED_Status, &BUS2F_LED_BlinkFrequenz,
                       &Error)){
    //display the error(s)
    MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
    MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
else {
    char AusgabeStr[255]={0};
    wsprintf(AusgabeStr, "Status der SF-LED: %s", SF_LED_Status);
    MessageBox(Handle, AusgabeStr, "", MB_ICONINFORMATION);
}//end else
.
.
.
```

6.49 The function: MPI6_IsPasswordRequired

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully. In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_IsPasswordRequired function can be used to determine whether read and/or write operations for the CPU require a password, i.e. whether the CPU is protected by a password. This function must be used if you are unsure whether a password must be transferred or not. An error will be produced if you should transfer a password to the CPU when this is not actually password protected.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Note

A password is not required for the ComDrvS7 functions ReadByte, ReadWord, ReadDword, ReadTimer, ReadCounter, WriteByte, WriteWord, WriteDword, WriteTimer, WriteCounter, MixRead_2 and MixWrite_2.

Description of the parameters

Argument	Typ	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PasswordRequired	bool*	Returns a value of 1 if the specified mode requires a password. Returns 0, if the specified mode requires a password.
Mode	BYTE	Transfer 'R' (or the ASCII code 82 dec.) to query the read mode. Transfer 'R' (or the ASCII code 87 dec.) to query the write mode.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note:
The data for this information function are provided by the virtual CPU. The CPU is only able to do this once "in parallel". If another communication instance should also accesses an information function of the same CPU, A communication error may result.

Example: See the example for MPI6_SendPasswordToPlc.

6.50 The function: MPI6_SendPasswordToPlc

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully. In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_SendPasswordToPlc function can be used to transfer a password to the CPU to enable read/write operations on a password-protected CPU. If you are not sure whether the access mode of the CPU is protected by a password, you must first check this condition with the MPI6_IsPasswordRequired function. If you are certain that an access password is required,

Important!

The password remains valid until the connection with the CPU is interrupted. If you are writing several times to the password-protected CPU, this means that the password must only be passed once. The condition is, that the communication link to the CPU is not interrupted.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Note

A password is not required for the ComDrvS7 functions ReadByte, ReadWord, ReadDword, ReadTimer, ReadCounter, WriteByte, WriteWord, WriteDword, WriteTimer, WriteCounter, MixRead_2 and MixWrite_2.

Description of the parameters

Argument	Typ	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
PasswordStr	char*	Password (8 characters max.) that protects the CPU. The password protection and the password itself are defined in the hardware configuration of the CPU.
PasswordIsCorrect	bool*	Returns a 1, if the password that was passed is correct. Returns a 0, if the password is bad.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The following example checks whether the CPU has a write protection feature. After that, the password is passed to the CPU.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
bool PasswortUebergabeNotwendig=false;
BYTE Modus='W'; //write mode
if (!MPI6_IsPasswordRequired(MPIHandle,
                        &PasswortUebergabeNotwendig, Modus, &Error)){

    //display the error(s)
    MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
    MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
if (PasswortUebergabeNotwendig){
   //
   char PasswortStr[20]={0};
   bool PasswortIstKorrekt=false;
   strcpy(PasswortStr, "Test"); //CPU is configured for the
                                //password 'Test'
   if (!MPI6_SendPasswordToPlc(MPIHandle, PasswortStr,
                              &PasswortIstKorrekt, &Error)){
      //display the error(s)
      MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
      MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
   }//end if
   else {
      if (PasswortIstKorrekt)
          MessageBox(Handle, "The password is correct.", "",
                   MB_ICONINFORMATION);
      else
          MessageBox(Handle, "The password is incorrect!", "",
                      MB_ICONSTOP);
   }//end else
}//end if
.
.
.
```

6.51 The function: MPI6_GetCountDB

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The function MPI6_GetCountDB can be used to determine the number of DBs in a CPU.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
CountDB	int*	This parameter returns the number of DBs that are available in the CPU.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below determines the number of data blocks in the connected communication partner.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     int AnzahlDB=0;
     //
     if (!MPI6_GetCountDB(MPIHandle, &AnzahlDB, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Number of DBs availabe: %u", AnzahlDB);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.52 The function: MPI6_GetDBInPlc

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetDBInPlc function can be used to determine the numbers of the data blocks that exist in a CPU.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
DBNumbers	WORD*	WORD-Array that is used to return the numbers of the DBs that exist in the CPU.
CountDB	INT*	Number of DBs that was entered into the WORD-Array.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below displays the number of the first DB that exists in the CPU.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     int AnzahlDB=0;
     WORD DBNummern[255]={0};
     //
     if (!MPI6_GetDBInPlc(MPIHandle, DBNummern, &AnzahlDB, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         if (AnzahlDB>0){
             wsprintf(AusgabeStr, "First available DB: %u",
                  DBNummern[0]);
             MessageBox(AppHandle, AusgabeStr, "",
                        MB_ICONINFORMATION);
         }//end if
         else
             MessageBox(AppHandle, "First available DB!", "",
                        MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.53 The function: MPI6_GetLengthDB

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.
In additions, the call to the function MPI6_ConnectToPLC or MPI6_ConnectToPLCRouting must also have been successful.

Brief description

The MPI6_GetLengthDB function can be used to determine the length of a data block that exists in the CPU. The length is returned in bytes.

PLC family S7-1500®, S7-1200® and LOGO!

This function is not possible.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
DBNr	WORD	The number of the DB of which the length is to be determined.
CountByte	WORD*	Length of the DB in bytes.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below determines the length of DB1 in the communication partner.
The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
//read data
if (!Fehler){
     WORD LaengeInByte=0;
     //
     if (!MPI6_GetLengthDB(MPIHandle, 1, &LaengeInByte, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Length of DB1: %u Byte", LaengeInByte);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
.
.
.
```

6.54 The function: MPI6_ChangeProtocolTypeForV5Functions

Conditions to execute the function

One of the initialisation functions (e.g. MPI6_OpenTcpIp, etc.) must have been completed successfully.

Brief description

The MPI6_ChangeProtocolTypeForV5Functions function can be used to convert the old read and write functions of ComDrvS7 version V5 to the new, faster protocol. This function is executed once after the initialisation functions, where the parameter TakeV6Protocol must have the value 1. This enables the users of older ComDrvS7 versions to benefit from the innovations of ComDrvS7 V6.

Description of the parameters

Argument	C-Type	Description
Handle	INT	The handle of the communication instance, which is being accessed (not required for the .Net wrapper class).
TakeV6Protocol	BYTE	When the value 1 is transferred, the new protocol will be used, even by the old read and write functions (e.g. MPI_A_ReadMerkerByte). If the function is called when the parameter has a value of 0, the old functions will be used.
Error	WORD*	If the function returns ‘0’, an error has occurred during execution. In this case, the Error parameter contains an error value.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Example

The example below reads clock memories MB0 to MB99 with the old ComDrvS7 function. Before issuing the call to the MPI_A_ReadMerkerByte function, a call to the function MPI6_ChangeProtocolTypeForV5Functions changes the procedure to the new protocol type.

The example assumes that one of the initialisation functions (e.g. MPI6_OpenTcpIp) was completed successfully. Similarly, the function **MPI6_ConnectToPLC** must have been executed without errors. The action may be followed by others. Executing the function **MPI6_CloseCommunication** terminates communications and eliminates the instance. In the same way as it was shown in the example of the MPI6_ReadByte function.

```
.
.
.
 char ErrorString[255]={0};
 WORD Error=0;
 //Auf neues V6-Protokoll umstellen
 BYTE TakeV6Protocol=1;
 //
 if (!MPI6_ChangeProtocolTypeForV5Functions(MPIHandleV6,
                                         TakeV6Protocol, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
 }//end if
 //
 BYTE ByteStatusBuffer[100];
 //Call the old function as usual, the new protocol will be used
 //automatically
 if (!MPI_A_ReadMerkerByte(MPIHandleV6, 0, 100, ByteStatusBuffer,
                         &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandleV6, ErrorString, Error);
     MessageBox(ApplHandle, ErrorString, "", MB_ICONEXCLAMATION);
 }//end if
 else {
     MessageBox(ApplHandle, "Daten wurden gelesen.", "",
               MB_ICONINFORMATION);
 }//end else
.
.
.
```

6.55 The function: MPI6_GetVersionComDrvS7

Conditions to execute the function

None.

Brief description

The MPI6_GetVersionComDrvS7 function returns the version number of the ComDrvS7 DLL as a string. This means that the version in used can easily be checked.

Description of the parameters

Argument	C-Type	Description
VersionStr	char*	String with the version number of the form "ComDrvS7 V6.XX"
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

6.56 The function: MPI_A_RealFromByteBuffer or MPI6_RealFromByteBuffer

Brief description

The MPI_A_RealFromByteBuffer function assembles a real number from a byte buffer. The function can be used when a real number was read from a DB using the ReadDBByte function.

Description of the parameters

Argument	C-Type	Description
RealValue	float*	The real number, which was determined from the first 4 bytes of the buffer.
ByteBuffer	BYTE*	The buffer whose first 4 bytes contain a real number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The real number is assembled from ByteBuffer bytes [0], ByteBuffer [1], ByteBuffer [2] and ByteBuffer [3].

6.57 The function: MPI_A_RealFromWordBuffer or MPI6_RealFromWordBuffer

Brief description

The MPI_A_RealFromWordBuffer function assembles a real number from a WORD buffer. The function can be used when a real number was read from a DB using the ReadDBWort function.

Description of the parameters

Argument	C-Type	Description
RealValue	float*	The real number, which was determined from the first 2 words of the buffer.
WordBuffer	WORD*	The buffer whose first 2 words contain a real number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The real number is assembled from the words WordBuffer[0] and WordBuffer[1].

6.58 The function: MPI_A_IntFromByteBuffer or MPI6_IntFromByteBuffer

Brief description

The MPI_A_IntFromByteBuffer function assembles an INT (16-Bit) number from a BYTE buffer. The function can be used when an Int number was read from a DB using the ReadDBByte function.

Description of the parameters

Argument	C-Type	Description
IntValue	short*	The INT number, which was determined from the first 2 bytes of the buffer.
ByteBuffer	BYTE*	The buffer whose first 2 bytes contain an Int number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The Int number is assembled from ByteBuffer bytes [0] and ByteBuffer [1].

6.59 The function: MPI_A_IntFromWordBuffer oder MPI6_IntFromWordBuffer

Brief description

Die Funktion MPI_A_IntFromWordBuffer setzt eine Int-Zahl (16-Bit) aus einem WORD-Buffer zusammen. Die Funktion kann z.B. eingesetzt werden, wenn eine Int-Zahl aus einem DB ausgelesen wurde und dabei die Funktion ReadDBWort zum Einsatz kam.

Description of the parameters

Argument	C-Type	Description
IntValue	short*	Die Int-Zahl (16-Bit) welche aus dem ersten Wort des Buffers ermittelt wurde.
WordBuffer	WORD*	Der Buffer dessen erstes Wort eine Int-Zahl enthält.
Function return	BOOL	Wurde die Funktion erfolgreich ausgeführt, so wird der Wert '1' (TRUE) geliefert. Bei einem Fehler ist der Rückgabewert '0' (FALSE).

Anmerkung

Die Int-Zahl wird aus dem Wort WordBuffer[0] ermittelt.

6.60 The function: MPI_A_DIntFromByteBuffer oder MPI6_DIntFromByteBuffer

Brief description

The MPI_A_DIntFromByteBuffer assembles a DIn (32-bit) number from a BYTE buffer. The function can be used when a DInt number was read from a DB using the ReadDBByte function.

Description of the parameters

Argument	C-Type	Description
DIntValue	int*	The DInt number, which was determined from the first 4 bytes of the buffer.
ByteBuffer	BYTE*	The buffer with the first 4 bytes containing an DInt number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The DInt number is determined from the the bytes ByteBuffer[0], ByteBuffer[1], ByteBuffer[2] and ByteBuffer[1].

6.61 The function: MPI_A_DIntFromWordBuffer or MPI6_DIntFromWordBuffer

Brief description

The MPI_A_DIntFromWordBuffer function assembles a DInt (32-bit) number from a WORD buffer. he function can be used when a DInt number was read from a DB using the ReadDBWort function.

Description of the parameters

Argument	C-Type	Description
DIntValue	int*	The DINT number, which was determined from the first 2 words of the buffer.
WordBuffer	WORD*	The buffer whose first 2 words contain a DInt number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The DInt number is assembled from the words WordBuffer[0] and WordBuffer[1].

6.62 The function: MPI_A_RealToWordBuffer or MPI6_RealToWordBuffer

Brief description

The MPI_A_RealToWordBuffer function saves a real number into a WORD buffer. The function can be used when a real number must be saved in a DB using the function WriteDBWort.

Description of the parameters

Argument	C-Type	Description
RealValue	float	The real number that is written to the first two words of the buffer.
WordBuffer	WORD*	The buffer whose first 2 words contain the real number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The real number is saved to the words WordBuffer[0] and WordBuffer[1].

6.63 The function: MPI_A_RealToByteBuffer or MPI6_RealToByteBuffer

Brief description

The function MPI_A_RealToByteBuffer saves a real number in a BYTE buffer. The function can be used when a real number must be saved in a DB using the function WriteDBByte.

Description of the parameters

Argument	C-Type	Description
RealValue	float	The real number that is written to the first 4 bytes of the buffer.
ByteBuffer	BYTE*	The buffer whose first 4 bytes contain the real number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The real number is saved to the bytes ByteBuffer[0], ByteBuffer[1], ByteBuffer[2] and ByteBuffer[3].

6.64 The function: MPI_A_IntToByteBuffer or MPI6_IntToByteBuffer

Brief description

The MPI_A_IntToByteBuffer function writes an Int (16-bit) number to a BYTE buffer. The function can be used when an Int number must be saved in a DB using the function WriteDBByte.

Description of the parameters

Argument	C-Type	Description
IntValue	short	The Int number that is written to the first 2 bytes of the buffer.
ByteBuffer	BYTE*	The buffer whose first 2 bytes contain the Int number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The Int number is saved to bytes ByteBuffer[0] and ByteBuffer[1].

6.65 The function: MPI_A_DIntToByteBuffer or MPI6_DIntToByteBuffer

Brief description

The MPI_A_DIntToByteBuffer function writes a DInt (32-Bit) number to a BYTE buffer. The function can be used when a DInt number must be saved in a DB using the function WriteDBByte.

Description of the parameters

Argument	C-Type	Description
DIntValue	int	The DInt number that is written to the first 4 bytes of the buffer.
ByteBuffer	BYTE*	The buffer whose first 4 bytes contain the DInt number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The DInt number is saved to bytes ByteBuffer[0], ByteBuffer[1], ByteBuffer[2] and ByteBuffer[3].

6.66 The function: MPI_A_DIntToWordBuffer oder MPI6_DIntToWordBuffer

Brief description

The MPI_A_DIntToWordBuffer function writes a DInt (32-bit) number to a WORD buffer. The function can be used when a DInt number must be saved in a DB using the function WriteDBByte.

Description of the parameters

Argument	C-Type	Description
DIntValue	int	The DInt number that is written to the first 2 bytes of the buffer.
WordBuffer	WORD*	The buffer whose first 2 bytes contain the DInt number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

Note

The DInt number is written to the words WordBuffer[0] and WordBuffer[1].

6.67 The function: MPI6_BcdToDecimal

Brief description

The MPI6_BcdToDecimal function converts a BCD number to a decimal number. The BCD number must be in the range 0-99.

Description of the parameters

Argument	C-Type	Description
Bcd	BYTE	BCD number in the range 0-99
Dec	BYTE*	The returned decimal number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

6.68 The function: MPI6_DecimalToBcd

Brief description

The MPI6_DecimalToBcd function converts a decimal number to a BCD number. The decimal number must be in the range 0-99.

Description of the parameters

Argument	C-Type	Description
Dec	BYTE	Decimal number in the range 0-99
BCD	BYTE*	The returned BCD number.
Function return	BOOL	If the function was executed successfully, a value '1'(TRUE) is returned. When an error has occurred, the returned value is '0' (FALSE).

7 Accessing several nodes using a serial port.

The example below is intended to show how two CPUs that are located on an MPI network can be accessed via a serial interface.
Here it must be noted that communication instances that share a communication resource must never be processed in different threads. The functions of the communication instances must be processed in a single thread, one after the other.

7.1 Executing the initialisation

To start with, the initialisation must be executed for each connection.
The respective variables are shown below. The interfacing parameters are available once only, because the baud rate is defined during the first initialisation. If the second initialisation is executed with the same COM port but with different interfacing parameters (e.g. a different baud rate), then the second set of definitions have no effect because the port has already been opened and configured.

```
int ComNr=2;           //COM2 port
long BaudRate=38400;  //baud Rate
BYTE PGMPIAdresse=0;  //MPI address of the DLL application = 0
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
bool SchnittstelleWarSchonAllokiert=false;
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
//Variables für Kommu1
int MPIHandle_1=-1;     //handle of the first communication instance
BYTE AGMPIAdresse_1=10; //the MPI address of the first CPU
bool Fehler_1=false;
//Variables für Kommu2
int MPIHandle_2=-1;     //handle of the second communication instance
BYTE AGMPIAdresse_2=11; //the MPI address of the second CPU
bool Fehler_2=false;
///////////////////////////
//establish connection 1
if (!MPI6_OpenRS232(&MPIHandle_1, ComNr, BaudRate,
                            PGMPIAdresse, HoechsteMPI,
                            &SchnittstelleWarSchonAllokiert,
                            &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "Kommu 1",
                MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Initialisation 1 was successful.", "",
           MB_ICONINFORMATION);
```

```
//establish connection 2
if (!MPI6_OpenRS232(&MPIHandle_2, ComNr, BaudRate,
                            PGMPIAdresse, HoechsteMPI,
                            &SchnittstelleWarSchonAllokiert,
                            &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "Kommu 2",
                MB_ICONEXCLAMATION);
     //close commu 1
     MPI6_CloseCommunication(MPIHandle_1, &Error);
     return;
}//end if
MessageBox(AppHandle, "Einleitung  2 war erfolgreich.", "",
           MB_ICONINFORMATION);
```

Please note that different MPI handles are transferred during the initialisation. These handles are used to specify the different communication instances.
The second call to the function "MPI6_OpenRS232" returns the parameter "SchnittstelleWarSchonAllokiert" with the value 'true'. This is because the COM2 port was already opened with the first initialisation of the first communication instance.

7.2 Establish communications with the CPUs

Now, the two communication instances can establish the connection with the respective CPU. This is done with the "MPI6_ConnectToPLC" function. The function must be called for each communication instance.

```
//establish communication 1
if (!MPI6_ConnectToPLC(MPIHandle_1, AGMPIAdresse_1, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "Commu 1",
                MB_ICONEXCLAMATION);
     Fehler_1=true;
}//end if
else
     MessageBox(AppHandle, "Commu 1 established successfully!!",
                "", MB_ICONINFORMATION);
//establish communication 2
if (!MPI6_ConnectToPLC(MPIHandle_2, AGMPIAdresse_2, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "Commu 2",
                MB_ICONEXCLAMATION);
     Fehler_2=true;
}//end if
else
     MessageBox(AppHandle, "Commu 2 established successfully!",
                "", MB_ICONINFORMATION);
```

The accessed PLC MPI address and the MPI handles that are being used in the calls to both "MPI6_ConnectToPLC" functions are different. For each communication instance is supposed to access to a different CPU.

7.3 Read data from the CPU

Now that both communication instances have established communications with the respective CPU, these can also read data from the CPUs. The communication instance with the handle "MPIHandle_1" always accesses the CPU with the MPI address "AGMPIAdresse_1", while the instance with the handle "MPIHandle_2" always accesses the MPI address "AGMPIAdresse_2". The following listing shows this situation.

```
if (!Fehler_1){
     WORD LaengeInByte=0;
     //
     if (!MPI6_GetLengthDB(MPIHandle_1, 1, &LaengeInByte, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Commu1\nlength of DB1: %u byte",
                  LaengeInByte);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
if (!Fehler_2){
     WORD LaengeInByte=0;
     //
     if (!MPI6_GetLengthDB(MPIHandle_2, 1, &LaengeInByte, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Commu2\nlength of DB1: %u bytes",
                   LaengeInByte);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
```

This determines and returns the lengths of data blocks 1 in the respective CPU.
At this point further data may be read or written from/to the CPU. The order of read and write operations is irrelevant.

7.4 Termination of communications and removal of the communication instances

To terminate communications with the CPUs, close the interface and remove the communication instances you must issue a call to the "MPI6_CloseCommunication" function for each communication instance.
If you only issue a call to the function of one communication instance, communication with the other instance remains unaffected. The interface will only be released when the second communication instance is removed by means of the "MPI6_CloseCommunication" function.

```
//terminate communication 1
if (!MPI6_CloseCommunication(MPIHandle_1, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communic. 1 terminated without errors.", "",
           MB_ICONINFORMATION);
//terminate communication 2
if (!MPI6_CloseCommunication(MPIHandle_2, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communic. 2 terminated without errors.", "",
           MB_ICONINFORMATION);
```

7.5 Notes on the example

If different serial interfaces were used in the example, e.g. COM1 for communication instance 1 and COM2 for communication instance 2, then the functions of the instances could also be invoked in different threads. The advantage would be, that the instances do not block each other.

The MPI adapter that must be used for a serial connection can usually manage multiple connections (MHJ-MPI adapter 2 connections, SIEMENS adapter 4 connections). This means that for a MHJ-MPI adapter a max. of 2 communication instances can access the communication partners via an MPI adapter (and thus a serial interface).

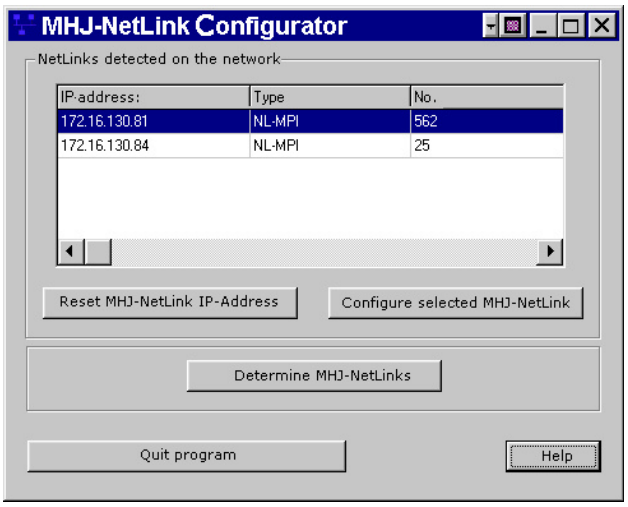
If the connection is established via TCP/IP and a CPU with an integrated Ethernet port or an Ethernet-CP, the communication instances are also independent of each other. The procedure is the same as for the serial interface, with only the "MPI6_OpenTcpIp" being used instead of "MPI6_OpenRS232".

8 Which conditions apply when using a MHJ-NetLink?

MHJ-NetLinks can manage at least two communications links. There is also the option to access multiple-MHJ NetLinks with different IP addresses.
The example below addresses two CPUs that are linked via MPI using two-MHJ NetLinks. The CPUs have the MPI address 10 and 12.

8.1 MHJ-NetLink configuration

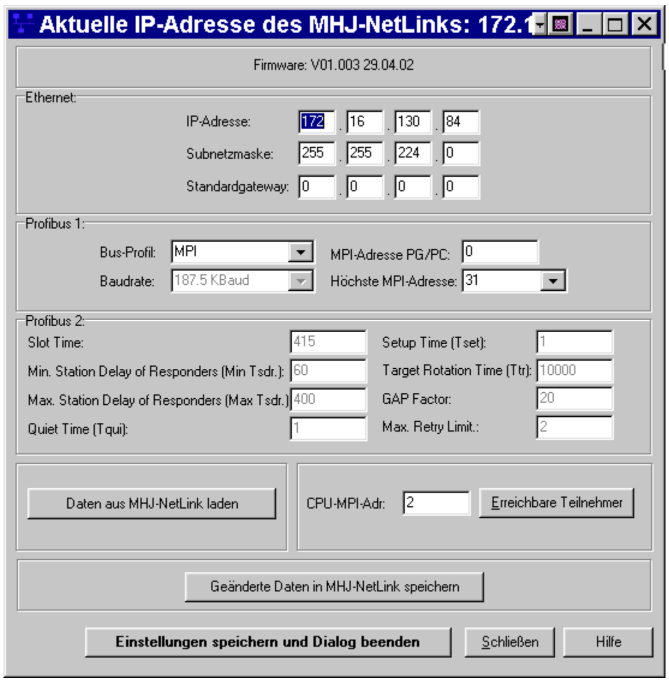
Before a MHJ-NetLink can be accessed via the MPI-DLL, it must be configured using the supplied MHJ-NetLink configurator. This can be used to change the IP address, the network settings, etc. and save them in the MHJ-NetLink. When a MHJ-NetLink was configured, the data is permanently available in the MHJ-NetLink, even after a power failure. This means that the configuration is only required once, unless you want to change the data.
The MHJ-NetLink configurator is shown below. When the program was started, the button "Determine MHJ-NetLinks" was activated.


MHJ-NetLink Configurator
NetLinks detected on the network
Reset MHJ-NetLink IP-Address
Configure selected MHJ-NetLink
Determine MHJ-NetLinks
Quit program
Help

IP-address:	Type	No.
172.16.130.81	NL-MPI	562
172.16.130.84	NL-MPI	25

Fig.: Configurator for MHJ-NetLink

This indicates that two-MHJ NetLinks were found on the network. In addition, we can see the IP address of the MHJ-NetLinks.
We will initially configure the MHJ-NetLink with the IP address 172.16.130.84. In the list, select this unit and press the button "Configure the selected MHJ-NetLink". If the MHJ-NetLink had the IP address 0.0.0.0, i.e. as delivered, a dialog would appear where the IP address must be specified. Then the dialog shown below will appear:


Aktuelle IP-Adresse des MHJ-NetLinks: 172.1

Firmware: V01.003 29.04.02

Ethernet:
IP-Adresse: 172 . 16 . 130 . 84
Subnetzmaske: 255 . 255 . 224 . 0
Standardgateway: 0 . 0 . 0 . 0

Profibus 1:
Bus-Profil: MPI
MPI-Adresse PG/PC: 0
Baudrate: 187.5 KBaud
Höchste MPI-Adresse: 31

Profibus 2:
Slot Time: 415
Setup Time (Tset): 1
Min. Station Delay of Responders (Min Tsdr.): 60
Target Rotation Time (Ttr): 10000
Max. Station Delay of Responders (Max Tsdr.) 400
GAP Factor: 20
Quiet Time (Tqui): 1
Max. Retry Limit.: 2

CPU-MPI-Adr: 2

Daten aus MHJ-NetLink laden
Erreichbare Teilnehmer
Geänderte Daten in MHJ-NetLink speichern
Einstellungen speichern und Dialog beenden
Schließen
Hilfe

Fig.: The configuration dialog

Here you can enter the network settings, the PG-MPI address, etc.
In the example, the bus-profile "MPI" is selected, which is the default setting. Furthermore, the "MPI address PG/PC" is set to the value '1'. The MPI address of the CPU is not required.
Press the button "Save modified data to MHJ-NetLink". A message is displayed indicating that the data was transferred to the MHJ-NetLink. The process is started if you confirm this message with "Yes".
A message indicates that the data was saved to the MHJ-NetLink, whereupon you must turn off the MHJ-NetLink. This is accomplished by removing and reinserting the NetLink from the PG interface of the CPU.
Now you can quit from this dialog by means of the button "Save settings and close dialog".

The same procedure must now be performed on the MHJ-NetLink with the IP address 172.16.130.81. In the list, select this unit and press the button "Configure the selected MHJ-NetLink". The dialog shown above is displayed, but with the other IP address. On this dialog, you also select "MPI" as the bus-profile. The PG-MPI-address is set to the value '3 '. Then you press the button "Save changed data in the MHJ-NetLink" and perform the steps described above. At this point, you can leave the dialog via the button "Save settings and exit the dialog."
When you have completed these steps, you can close the configurator with the button then "Quit program".

Additional information on commissioning a MHJ-NetLink can be found in the Help function for the software configuration.

8.2 Executing the initialisation

As with a connection via RS232, the open function must also be called to establish a connection via a MHJ-NetLink. However, in this case the "MPI6_OpenNetLink" should be used. The following figure shows this situation:

```
BYTE HoechsteMPI=31;  //highest address permitted in network = 31
WORD Error=0;         //error variable
char ErrorString[255]={0};//error string to return the error
//Variables für Kommu1
char IPAdresseStr_1[50]={0};
int MPIHandle_1=-1;     //handle of the first communication
BYTE PGMPIAdresse_1=1;  //MPI address of the 1st communic.instance = 1
BYTE AGMPIAdresse_1=10; //the MPI address of the first CPU
bool Fehler_1=false;
//variables for Commu2
char IPAdresseStr_2[50]={0};
BYTE PGMPIAdresse_2=3;  //MPI address of the 2nd communic.instance = 3
int MPIHandle_2=-1;     //handle of the second communication instance
BYTE AGMPIAdresse_2=12; //the MPI address the second CPU
bool Fehler_2=false;
//
strcpy(IPAdresseStr_1, "172.16.130.84");
strcpy(IPAdresseStr_2, "172.16.130.81");
///////////////////////////
//establish connection 1
if (!MPI6_OpenNetLink(&MPIHandle_1, IPAdresseStr_1,
                              PGMPIAdresse_1,
                              HoechsteMPI, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     return;
}//end if
MessageBox(AppHandle, "Einleitung  1 war erfolgreich.", "",
           MB_ICONINFORMATION);
//establish connection 2
if (!MPI6_OpenNetLink(&MPIHandle_2, IPAdresseStr_2,
                              PGMPIAdresse_2,
                              HoechsteMPI, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     //
     MPI_A_KommuBeenden(MPIHandle_1, &Error);
     return;
}//end if
MessageBox(AppHandle, "Initialisation 2 was successful.", "",
           MB_ICONINFORMATION);
```

8.3 Establish communications

The other functions do not distinguish themselves from other connections (e.g. NetLink PRO, TCP/IP or RS232). This means that the "MPI6_ConnectToPLC" function is called for each communication instance stating the respective AG-MPI address (for TCP/IP this would always be set to 2).

```
//establish communication 1
if (!MPI6_ConnectToPLC(MPIHandle_1, AGMPIAdresse_1, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     Fehler_1=true;
}//end if
else {
     MessageBox(AppHandle, "Communic. 1 established successfully!",
                "", MB_ICONINFORMATION);
}//end else
//establish communication 2
if (!MPI6_ConnectToPLC(MPIHandle_2, AGMPIAdresse_2, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     Fehler_2=true;
}//end if
else {
     MessageBox(AppHandle, "Communic. 2 established successfully!",
                "", MB_ICONINFORMATION);
}//end else
```

8.4 Read data

Now the data must be read from the CPUs. The access functions used here are also independent of the connection resource. As in the previous example, the length of DB1 will be determined in the respective CPU.

```
//read data
if (!Fehler_1){
     WORD LaengeInByte=0;
     //
     if (!MPI6_GetLengthDB(MPIHandle_1, 1, &LaengeInByte, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Commu1\nlength of DB1: %u byte",
                  LaengeInByte);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if

if (!Fehler_2){
     WORD LaengeInByte=0;
     //
     if (!MPI6_GetLengthDB(MPIHandle_2, 1, &LaengeInByte, &Error)){
         //display the error(s)
         MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
         MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
     }//end if
     else {
         char AusgabeStr[255]={0};
         wsprintf(AusgabeStr, "Kommu2\nlength of DB1: %u byte",
                  LaengeInByte);
         MessageBox(AppHandle, AusgabeStr, "", MB_ICONINFORMATION);
    }//end else
}//end if
```

At this point further data may be read or written from/to the CPU. The order of read and write operations is irrelevant.

8.5 Terminate communications

If you want to terminate communications with the respective CPU, you can accomplish this using the "MPI6_CloseCommunication" function. The communication resource is released simultaneously and the communication instance eliminated.

```
//terminate communication 1
if (!MPI6_CloseCommunication(MPIHandle_1, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_1, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Communic. 1 terminated without errors.", "",
           MB_ICONINFORMATION);
//terminate communication 2
if (!MPI6_CloseCommunication(MPIHandle_2, &Error)){
     //display the error(s)
     MPI_A_GetDLLError(MPIHandle_2, ErrorString, Error);
     MessageBox(AppHandle, ErrorString, "", MB_ICONEXCLAMATION);
}//end if
MessageBox(AppHandle, "Kommunikation 2 ohne Fehler beendet.", "",
           MB_ICONINFORMATION);
```

8.6 Notes on the example

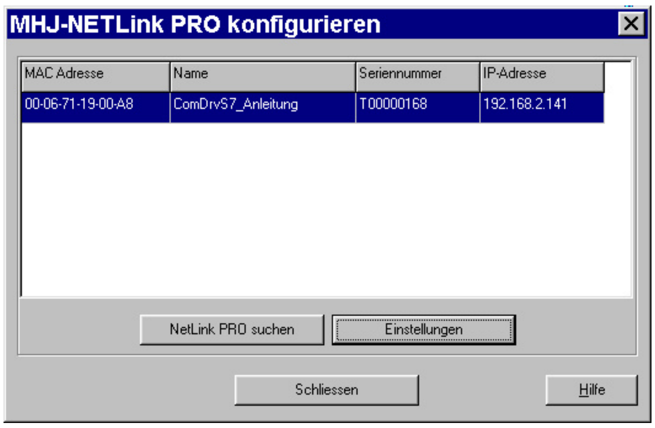
Since different serial interfaces were used in the example (two MHJ-NetLinks), the functions of the respective communication instance could also be invoked in different threads. The advantage would be, that the communication instances do not block each other.

9 Which conditions apply when using a MHJ-NetLink PRO?

A NETLink PRO enables communications from TCP/IP on the PC side to a MPI or DP interface on the CPU. In this case, the MPI/DP network supports all baud rates up to 12 Mbaud. The NETLink PRO can manage at least 4 PC connections.

9.1 Configuration of a NetLink PRO

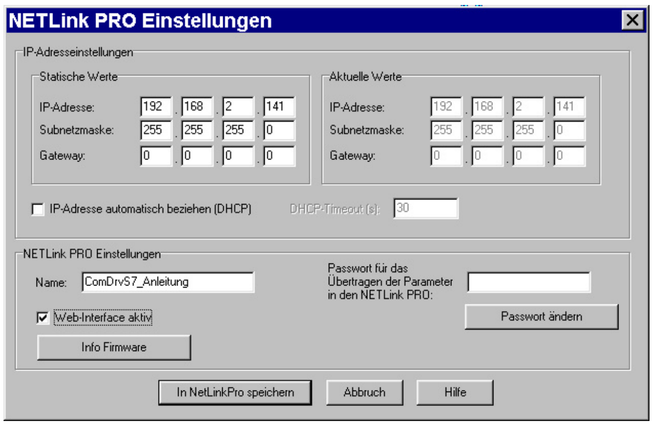
Before a NetLink PRO can be accessed via the ComDrvS7, it must be configured using the supplied NetLink PRO configurator. This can be used to change the IP address, the network settings, etc. and save them in NetLink PRO. When a NetLink PRO was configured, the data is permanently available in the NetLink PRO, even after a power failure. This means that the configuration is only required once, unless you want to change the data.
The following figure shows the NETLink PRO configurator. When the program was started, the button "Search for NETLink PRO" was activated.



MAC Adresse	Name	Seriennummer	IP-Adresse
00-06-71-19-00-A8	ComDrvS7_Anleitung	T00000168	192.168.2.141

Fig.: List of NETLink PRO units on the network

Now you can select the required NETLink Pro in the list and then press the button "Settings". As a result of the dialog, "NETLink PRO settings" will be displayed where you can define important communication parameters.



	Statische Werte	Aktuelle Werte
IP-Adresse:	192 . 168 . 2 . 141	192 . 168 . 2 . 141
Subnetzmaske:	255 . 255 . 255 . 0	255 . 255 . 255 . 0
Gateway:	0 . 0 . 0 . 0	0 . 0 . 0 . 0

☐ IP-Adresse automatisch beziehen (DHCP) DHCP-Timeout (s): 30

Name: ComDrvS7_Anleitung

Passwort für das Übertragen der Parameter in den NETLink PRO:

☑ Web-Interface aktiv

Fig.: NETLink PRO settings

Help on the individual parameters is available via the "Help" button.
If you have made the necessary adjustments, you can write them to the NETLink PRO with the "Save in NETLink PRO" button.

9.2 The two initialisation functions of the NETLink PRO

There are two initialisation functions to establish a communication instance for a NETLink PRO. These are the functions
MPI6_Open_NetLinkPro_TCP_AutoBaud
and
MPI6_Open_NetLinkPro_TCP_SelectBaud
The difference between the two functions is that the function
MPI6_Open_NetLinkPro_TCP_AutoBaud
does not require the baud rate on the MPI/DP network. During the initialisation, the NETLink PRO attempts to determine and retrieve the baud rate that was defined for the bus. The disadvantage of this version is the time that is required for the determination of the baud rate, because this may take a few seconds. The initialisation process is extended by this time.

If the baud rate is available, the function
MPI6_Open_NetLinkPro_TCP_SelectBaud
should be used, since this assigns the baud rate and eliminates the time required by the determination of the baud rate.

10 What conditions must be considered when using a SIMATIC®-NET?

SIMATIC®-NET is supported from ComDrvS7 version 4.

The condition is that the SIMATIC? NET driver was installed on the PC. This driver is installed on the PC, for example, when the Simatic?-Manager (from V5.1), the driver for the SIEMENS-USB adapter or the Teleservice V6 are installed. You must select the interface to be used here in the "PG/PC interface configuration" dialog. You can access this dialog by means of the file "s7epatsx.exe" in the Windows System32 directory. This means, for example, that the Siemens USB-MPI adapter, the CP5511 or CP5612 may be selected.
The selected interface and the respective settings will be used by ComDrvS7 if the initialisation is invoked via the **MPI6_Open_SimaticNet** function.

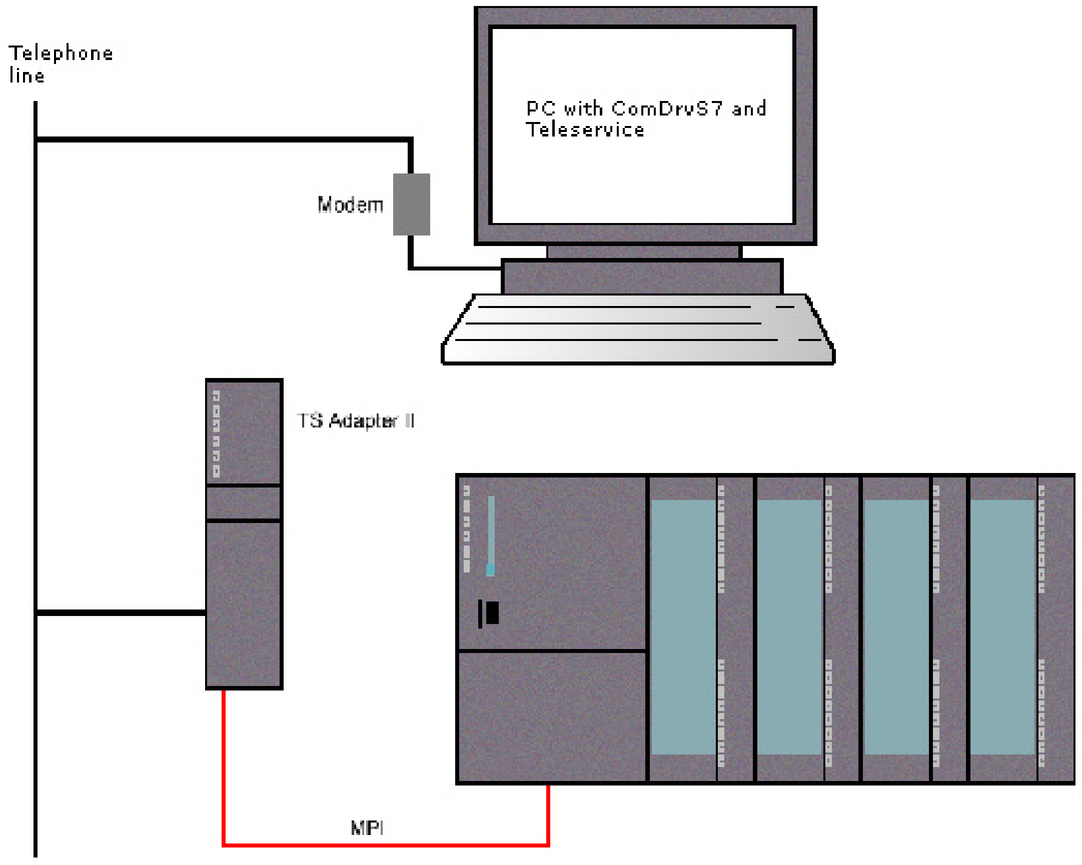
Besides ComDrvS7, the Siemens software products can continue to access the CPU, provided that the communication resources of the CPU have not been exhausted. It is thus possible, for example, that your application accesses a CPU with ComDrvS7 (using SIMATIC®-NET) while it is being accessed simultaneously via the Simatic®-Manager.

SIMATIC®-NET is **not** supported from ComDrvS7 64-Bit.

11 Remote access procedure via telephone line using ComDrvS7

As of ComDrvS7 version 4, the Siemens Teleservice (from version 6) can be used for remote access purposes via the telephone line.

Any kind of modem may be employed on the PC. The system side supports i.e. the Siemens Teleservice Adapter II.
The figure below shows the necessary components with their names:



On the PC the Siemens Teleservice from V6 must be installed. The PC is connected to the telephone line via a modem. On the system side the Siemens Teleservice Adapter II may be used. In the ComDrvS7 the communication path SIMATIC®-NET must be used, i.e. the initialisation function MPI6_Open_SimaticNet must be called. The interface in the SIMATIC®-NET driver of the TS-Adapter II that is used in the example is set to "TS Adapter". This also enables routing.

12 General notes on the ComDrvS7-DLL

12.1 What must be considered when a CPU is accessed by multiple communication instances?

Communication errors will occur when multiple communication instances access the information functions of a CPU. A communication error will result if, for example, several instances request the operating mode position in immediate succession or even in different threads but essentially in "parallel". The CPU must assemble the answer to such a request in a virtual process but it is not always possible to do this.
Status queries can be invoked in parallel but only a limited quantity. This number depends on the CPU type. The number is available from the CPU manual.

12.2 What must be considered when the next ComDrvS7 DLL or other applications are executed on the PC?

If additional applications are executed along with the ComDrvS7, then the functions of the ComDrvS7 must be invoked in one or more threads. It is important that the threads have the priority "THREAD_PRIORITY_TIME_CRITICAL" so that the response times on the MPI/DP bus or via TCP/IP are met.

12.3 When is it possible to issue calls to the functions of the individual communication instances in different threads?

When multiple communication instances are being used, i.e., a connection is established with multiple CPUs while different communication resources are being used, then the condition is satisfied, that the functions of the individual instances may be called in different threads. Each thread must possess the priority "THREAD_PRIORITY_TIME_CRITICAL".
It is, however, not possible to call functions of a communication instance from different threads.

When a communication resource is used by multiple instances, e.g. when 2 CPUs are being accessed via the COM1 port, then the functions of both instances must be executed in a single thread in succession.

13 Error messages

The following list contains the possible error numbers. One of these values may be included in the "error" variable of a DLL function when the value returned by this function is '0 '(FALSE). The functions MPI_A_GetDLLError, MPI_A_GetDLLErrorEng, or MPI6_GetDLLError MPI6_GetDLLErrorEng can be used to retrieve a descriptive string (null terminated) for the respective error.

Value (dec)	Description
54.273	The requested information is not available on the PLC!
53.825	Protection level error of the CPU!
53.409	Action not possible due to the protection level!
53.377	Select the operating mode that is necessary for this function!
53.298	The parameters passed to the PLC are faulty!
33.794	Action cannot be executed because of an incorrect status of the PLC!
33.540	Message from the module. A resource bottleneck exists!
19.718	Temporary lack of resources in the PLC. Repeat the request.
16.997	The MPI address of the PG has already been allocated on the network!
16.949	The node address of a connected PLC is too high!
16.662	The partner refuses to communicate!
1.046	PLC is not a S7-1500
1.045	PLC is not a LOGO
1.044	PLC is not a S7-1200
1.043	Failure to configure the mode for the S7-1200® family
1.030	VM area is only possible with a LOGO!®
1.029	Function is not possible with a S7-1200®
1.028	Function is not possible with a LOGO!®
1.027	Function is only possible with the MICRO version of ComDrvS7
1.026	Function is not possible with the MICRO version of ComDrvS7
1.004	Function is only possible with the Extended version of ComDrvS7
1.003	Unknown block type!
1.002	DB0 not permitted!
1.001	This feature cannot be executed in the Lite version!
1.000	Error when creating a DLL instance!
733	The given handle is not valid.
732	With at least one block, the action is not possible.
731	Error while executing compress.
729	Error, because the WLD-file already exists.
728	Failure to configure the mode for the S7-1200® family
727	Incorrect position of the mode switch or the CPU is already in the required mode.

726	The action is not possible in this mode.
725	The action is not possible in RUN mode.
724	The service is not supported by the CPU.
723	Number of bytes for the mix function is invalid.
722	Failed to accept the status data for the mix function.
721	Internal error mix function: Pointer number too high.
720	One or more control values are incorrect.
719	Die Anzahl der DBs in der WLD-Datei übersteigt den angegebenen Max-Parameter.
718	The number of DBs in the WLD file exceeds the specified max-parameter.
717	WLD action: Block is not available in the CPU.
716	The DB already exists in the WLD. Cannot overwrite.
715	The file is not a correct WLD file.
714	WLD action: Failed to test whether the block to be transferred is already present in the CPU.
713	WLD action: Block is already available in the CPU.
712	WLD file does not exist.
711	Block is not available in the WLD file.
710	WLD action: File operation error.
709	WLD action: Block is too large for the action.
708	Status of one or more operands cannot be delivered. Possible reason: One or more operands are not available in the CPU.
707	The specified operand is not permitted for this function.
706	One or more operands cannot be controlled. Possible reason: One or more operands are not available in the CPU.
614	NetLink PRO: Cannot determine the baud rate on the bus!
604	Status data of one or more requested operands not available.
603	Fault when evaluating the status data.
602	Fault - password contains too many characters.
601	Fault when converting the password.
518	An unknown error has occurred during the initialisation!
517	Fault occurred when evaluating accessible nodes!
513	Fault when closing the interface!
512	Fault when opening the interface!
511	Errors when evaluating the info-data!
510	The parameters passed are faulty!
509	Communication error has occurred!
508	Memory error. Memory could not be allocated!

500	Demo limit reached. This message appears only in the demo version of the DLL when the area of the demo version has been exceeded. In this case, please note the additional information enclosed with demo version on the permitted operand areas. Obsolete from version 6.

14 Access to operands of a LOGO!® from SIEMENS

In the help-function of the LOGOSoft-Comfort-software you find the tables with the mappings between I/O and VM (Variable Memory) addresses of the LOGO!®.
Open the help-function and search for the page named "Parameter VM Mapping".
Maybe in newer devices (higher 0BA8) the mapping is changing, so take a look at this page, to get the latest informations.

14.1 Digital inputs

The digital inputs from a LOGO!® can be read directly.
You can use the read functions (e. g. MPI6_ReadByte) with the operand area "I".
The writing of inputs is not possible.

14.1.1 Addressing the digital inputs

In the following table you find the addresses for the inputs I1 to I24.

LOGO Inputs	Addressing in ComDrvS7 0BA7/0BA8
I1	I0.0
I2	I0.1
I3	I0.2
I4	I0.3
I5	I0.4
I6	I0.5
I7	I0.6
I8	I0.7
I9	I1.0.
I10	I1.1
I11	I1.2
I12	I1.3
I13	I1.4
I14	I1.5
I15	I1.6
I16	I1.7
...	...

14.2 Analog inputs

The analog inputs from a LOGO!® can be read via the VM area.
You can use the read functions (e. g. MPI6_ReadWord) with the operand area "V".

14.2.1 Addressing the analog inputs

In the following table you find the addresses for the inputs AI1 to AI8.

LOGO analog input	Address ComDrvS7 0BA7	Address ComDrvS7 0BA8
AI1	VW926	VW1032
AI2	VW928	VW1034
AI3	VW930	VW1036
AI4	VW932	VW1038
AI5	VW934	VW1040
AI6	VW936	VW1042
AI7	VW938	VW1044
AI8	VW940	VW1046
AI8-AI16	-	VW1048-VW1062

Example 0BA7:

If you want to read the AI6, you have to specify the address 936 and the operand area "V".

Example 0BA8:

If you want to read the AI6, you have to specify the address 1042 and the operand area "V".

14.3 Digital outputs

The digital outputs from a LOGO!® can be read and write directly.
You can use the read and write functions (e. g. MPI6_ReadByte or MPI6_WriteByte) with the operand area "Q".

14.3.1 Addressing the digital outputs

In the following table you find the addresses for the outputs Q1 to Q16.

LOGO Outputs	Addressing in ComDrvS7 0BA7/0BA8
Q1	Q0.0
Q2	Q0.1
Q3	Q0.2
Q4	Q0.3
Q5	Q0.4
Q6	Q0.5
Q7	Q0.6
Q8	Q0.7
Q9	Q1.0.
Q10	Q1.1
Q11	Q1.2
Q12	Q1.3
Q13	Q1.4
Q14	Q1.5
Q15	Q1.6
Q16	Q1.7

14.4 Analog ouputs

The analog outputs from a LOGO!® can be read and write via the VM area.
You can use the read and write functions (e. g. MPI6_ReadWord or MPI6_WriteWord) with the operand area "V".

14.4.1 Addressing the analog ouputs

In the following table you find the addresses for the outputs.

LOGO analog output	Address ComDrvS7 0BA7	Address ComDrvS7 0BA8
AQ1	VW944	VW1072
AQ2	VW946	VW1074
AQ3-AQ16	-	VW1076-VW1102

14.5 Digital Flags

The digital flags from a LOGO!® can be read and write via the VM area.
You can use the read and write functions (e. g. MPI6_ReadByte or MPI6_WriteByte) with the operand area "V".

14.5.1 Addressing the digital flags

In the following table you find the addresses for the flags M1 to M27.

LOGO digital flag	Address ComDrvS7 0BA7	Address ComDrvS7 0BA8
M1	V948.0	V1104.0
M2	V948.1	V1104.1
M3	V948.2	V1104.2
M4	V948.3	V1104.3
M5	V948.4	V1104.4
M6	V948.5	V1104.5
M7	V948.6	V1104.6
M8	V948.7	V1104.7
...	...	..

14.6 Analog Flags

The analog flags from a LOGO!® can be read and write via the VM area.
You can use the read and write functions (e. g. MPI6_ReadWord or MPI6_WriteWord) with the operand area "V".

14.6.1 Addressing the analog flags

In the following table you find the addresses for the analog flags.

LOGO analog flags	Address ComDrvS7 0BA7	Address ComDrvS7 0BA8
AM1	VW952	VW1118
AM2	VW954	VW1120
AM3	VW956	VW1122
AM4	VW958	VW1124
AM5	VW960	VW1126
AM6	VW962	VW1128
AM7	VW964	VW1130
AM8	VW966	VW1132
AM9	VW968	VW1134
AM10	VW970	VW1136
AM11	VW972	VW1138
AM12	VW974	VW1140
AM13	VW976	VW1142
AM14	VW978	VW1144
AM15	VW980	VW1146
AM16	VW982	VW1148
AM17-AM63	-	VW1244

14.7 Analog and digitale network-inputs and network-outputs

In LOGO!® devices with ethernet interface (0BA7 and higher) you can use digitale and analoge network-inputs and network-outputs.
These operands are adressed inside the VM area from byte address 0 to byte address 850.
In ComDrvS7 you can use the read and write functions (e. g. MPI6_ReadWord or MPI6_WriteWord) with the operand area "V" to access these operands.

The access is possible with bit, byte, word and dword functions.

15 Configuration of the LOGO's IP address

The LOGO!® (0BA7 or higher) must have an IP address to make it accessible via Ethernet. The device must be accessible for the PC to use the LOGO programming software as well as ComDrvS7. You may connect the LOGO controller directly to the network adapter of the PC or via a network switch.

The example assumes that the PC with the programming software and ComDrvS7 has the IP address 192.168.1.90. The LOGO controller should be located on the same subnet, and to simplify matters, only in the last digit of the IP address differs. For this reason, the LOGO controller will be set to the IP address 192.168.1.196. Please remember that this IP address must not be occupied.
You can easily adjust the IP address and the subnet mask of the LOGO controller using the LOGO's display. To get to the menu of the LOGO, press the ESC key. Then, using the up and down arrow keys, select menu item "Network" and confirm by pressing the OK button. Use OK in the submenu to select the menu item "IP address". The current IP address will be visible. Press the OK button again to use the arrow keys to select the individual addresses. As mentioned, the example is set to the address 192.168.1.196. When this address was set, you can accept it with the OK button.Now for the subnet mask. In the example this is set to 255.255.255.0. From the current position the subnet mask can be selected by pressing the down arrow key and by pressing OK to start changing the subnet mask. When you have selected 255.255.255.0, complete the entry with the OK button.For further information on setting the IP addresses in the LOGO!®, please read the relevant sections in the help files of the LOGO programming software or the manual of the LOGO controller.
Now the IP settings have been applied to the LOGO controller and you can return to the main menu of LOGO by pressing the "ESC" button on the LOGO!®.

15.1 Special behaviors of a LOGO!® 0BA8 (and higher)

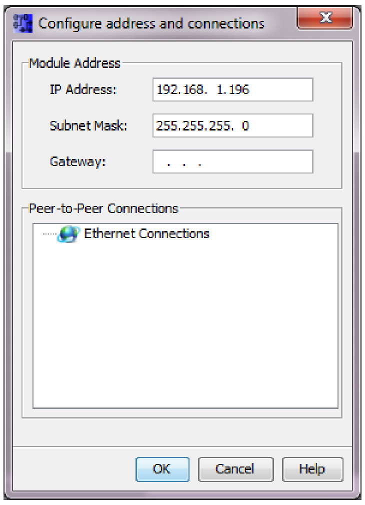
If you use a 0BA8 (or later), you have to note a special communication-behavior.
The 0BA8 closes a communication-channel after 5 seconds, when there is no communication.

<u>Example:</u>
If you read/write operands every 8 seconds, you have to perform the following steps:
1. Execute MPI6_OpenTcpIp_Logo
2. Execute MPI6_ConnectToPLC
3. Read and write the operands
4. Execute MPI6_CloseCommunication

If you read operands in time-intervals of less than 5 seconds, the closing and reopening of the communication-channel is not necessary.

15.2 Configuration of the Ethernet connection by means of the LOGO!® programming software

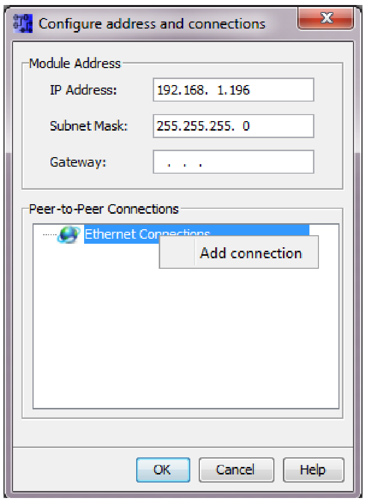
The next step is to execute menu item "Tools->Ethernet connections" in the LOGO programming software. As a result, the dialog "Configure address and Connections" is displayed. For the purposes of the current example, this is completed as follows:



IP Address: 192.168. 1.196
Subnet Mask: 255.255.255. 0
Gateway: . . .

Figure: "Configure address and Connections" dialog box

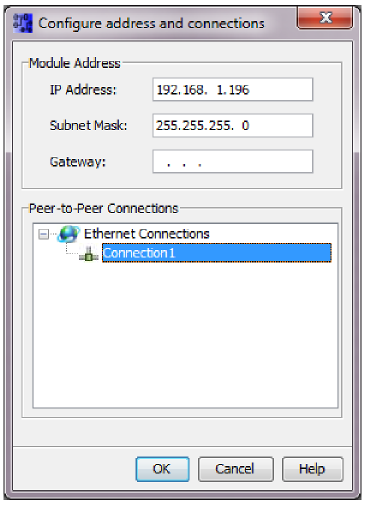
The IP address of the LOGO that will be addressed is entered into "Module Address->IP address", i.e. in the example this is 192.168.1.196. If you enter this IP address, the subnet mask is applied automatically. The next field is "Peer-to-Peer connections". Here it is necessary to create a new Ethernet connection. For this purpose, you select "Ethernet Connections" and press the **right mouse key**.



IP Address: 192.168. 1.196
Subnet Mask: 255.255.255. 0
Gateway: . . .

Figure: add a new Ethernet connection

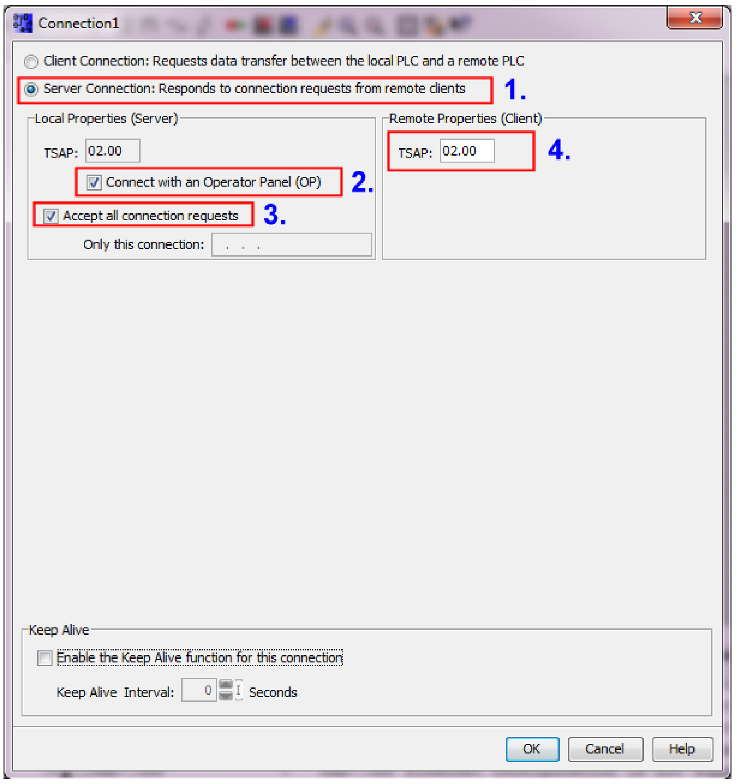
Now you must select menu item "Add Connection". Then result is displayed as follows in the dialog box:



IP Address: 192.168. 1.196
Subnet Mask: 255.255.255. 0
Gateway: . . .

Figure: new connection

Continue with a double-click on the new entry "Connection1". As a result, dialog "Connection1" will be displayed where the options must be completed as follows:



( ) Client Connection: Requests data transfer between the local PLC and a remote PLC
(●) Server Connection: Responds to connection requests from remote clients

Local Properties (Server)
TSAP: 02.00
☑ Connect with an Operator Panel (OP)
☑ Accept all connection requests
Only this connection: . . .

Remote Properties (Client)
TSAP: 02.00

Keep Alive
☐ Enable the Keep Alive function for this connection
Keep Alive Interval: 0 Seconds

Figure: connection settings

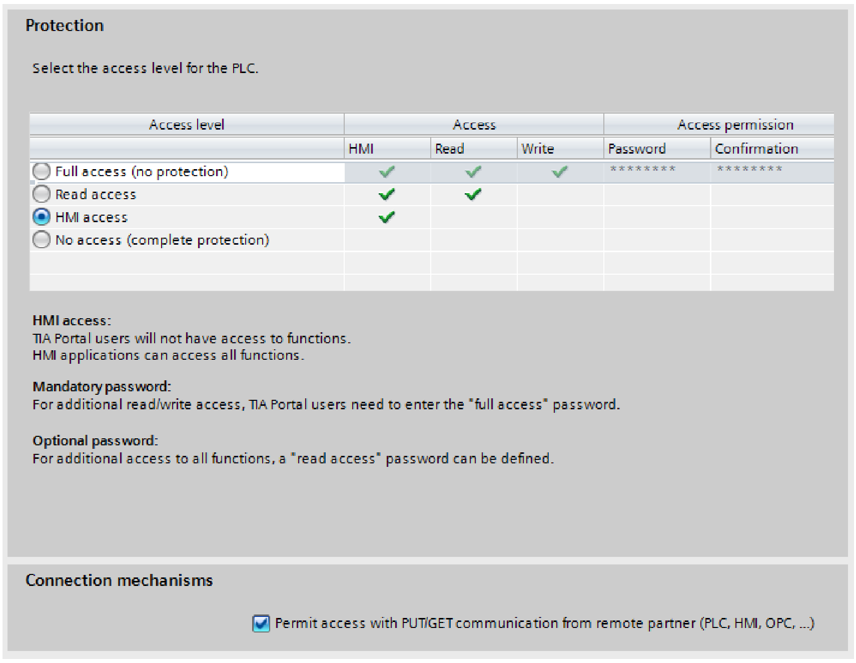
Enter the adjustments in the sequence that is shown in the figure. You can now quit from the dialog box by clicking "OK".
Confirm the resulting dialog box "Configure address and connections" and quit by pressing OK.

Now the IP settings have been completed and you can transfer the settings to the LOGO!®.

16 Required settings in a PLC 1500® (and S7-1200® from firmware version 4) from Siemens

To establish a connection between the PC and a PLC 1500 (and 1200 from firmwareversion 4) via ComDrvS7 you have to make some PLC-settings. First you have to select the option "Permit access with PUT/GET communication from remote partner (PLC, HMI, OPC ...)". You find this option inside the PLC-properties, tab sheet "General", category "Protection".
The "highest" selectable access level is named "HMI access". There, a password for the programming access is required. With this level, the changing of the PLC-blocks is protected with this password

In the following picture you can see an example of the settings:



Protection

Select the access level for the PLC.

Access level	Access			Access permission	
	HMI	Read	Write	Password	Confirmation
Full access (no protection)	✓	✓	✓	********	********
Read access	✓	✓			
[✓ HMI access]	✓				
No access (complete protection)					

HMI access:
TIA Portal users will not have access to functions.
HMI applications can access all functions.

Mandatory password:
For additional read/write access, TIA Portal users need to enter the "full access" password.

Optional password:
For additional access to all functions, a "read access" password can be defined.

Connection mechanisms

[✓] Permit access with PUT/GET communication from remote partner (PLC, HMI, OPC, ...)