PHOENIX CONTACT

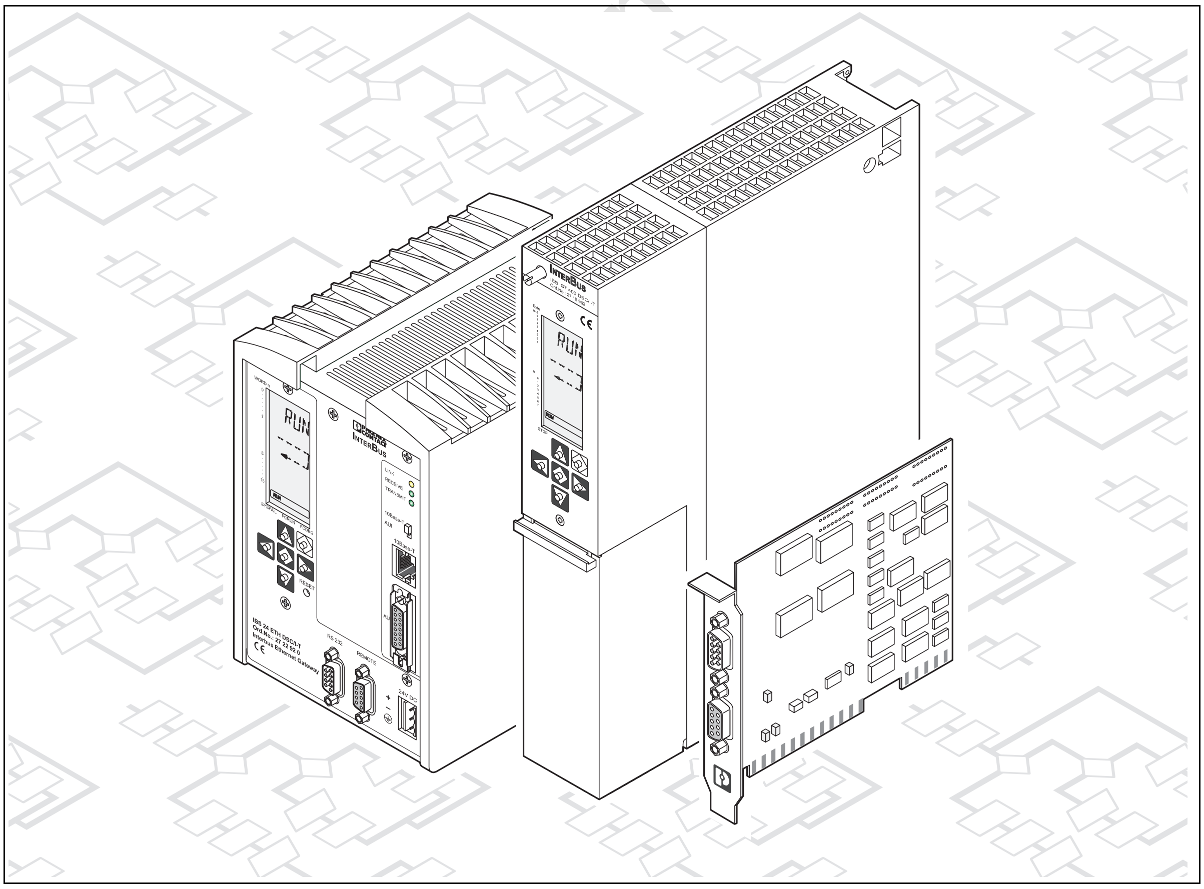
INNOVATION IN INTERFACE

User Manual

Driver Reference Manual for G4-Based Controller Boards Using PC Bus and Ethernet

Designation: IBS PC SC SWD UM E

Order No.: 27 45 17 2



Driver Reference Manual for G4-Based Controller Boards Using PC Bus and Ethernet

Designation:	IBS PC SC SWD UM E
Revision:	05
Order No.:	27 45 17 2

This manual is valid for:

INTERBUS PC controller boards

and

INTERBUS Ethernet gateways

see pages 1-5

533305

Please Observe the Following Notes

In order to guarantee the safe use of your device, we recommend that you read this manual carefully. The following notes give you information on how to use this manual.

Qualifications of the User Group

The products described in this manual should be installed/operated/ maintained only by qualified application programmers and software engineers, who are familiar with automation safety concepts and applicable national standards. Phoenix Contact assumes no liability for damage to any products resulting from disregard of information contained in this manual.

Explanation of Symbols Used

The *attention* symbol refers to an operating procedure which, if not carefully followed, could result in damage to equipment or personal injury.

The *note* symbol informs you of conditions that must strictly be observed to achieve error-free operation. It also gives you tips and advice on hardware and software optimization to save you extra work.

The *text* symbol refers you to detailed sources of information (manuals, data sheets, literature, etc.) on the subject matter, product, etc. This text also provides helpful information for the orientation in the manual.

We Are Interested in Your Opinion

We are constantly striving to improve the quality of our documents.

Should you have any suggestions or recommendations for improving the contents and layout of our documents, please send us your comments. Please use the fax form at the end of the manual for this purpose.

General Terms and Conditions of Use for Technical Documentation

Phoenix Contact GmbH & Co. KG reserves the right to alter, correct, and/or improve the technical documentation and the products described in the technical documentation at its own discretion and without giving any notice.

The receipt of technical documentation (in particular data sheets, installation instructions, manuals, etc.) does not constitute any further duty on the part of Phoenix Contact GmbH & Co. KG to furnish information on alterations to products and/or technical documentation.
Any other agreement shall only apply if expressly confirmed in writing by Phoenix Contact GmbH & Co. KG.

Although Phoenix Contact GmbH & Co. KG makes every effort to ensure that the information content is accurate, up-to-date and state-of-the-art, technical inaccuracies and/or printing errors in the information cannot be ruled out. Phoenix Contact GmbH & Co. KG does not offer any guarantees as to the reliability, accuracy or completeness of the information appearing on the Website. Phoenix Contact GmbH & Co. KG accepts no liability or responsibility for errors or omissions in the content of the technical documentation (in particular data sheets, installation instructions, manuals, etc.).

As far as is permissible by applicable jurisdiction, no guarantee or claim for liability for defects whatsoever shall be granted in conjunction with the information available in the technical documentation, whether expressly mentioned or implied. This information does not include any guarantees on quality, does not describe any fair marketable quality and does not make any claims as to quality guarantees or guarantees on the suitability for a special purpose. Phoenix Contact GmbH & Co. KG reserves the right to alter, correct, and/or improve the information and the products described in the information at its own discretion and without giving any notice.

Statement of Legal Authority

Internet

You will find current information on products from Phoenix Contact on the Internet at **www.phoenixcontact.com**.

Table of Contents

Section 1

This section informs you about
– the content of this user manual.

1 Overview

1.1 General

This user manual is a reference manual for drivers of INTERBUS controller boards for PC bus (PCI, ISA, etc.) and Ethernet (TCP/IP protocol). It contains descriptions for all function calls and the corresponding error messages.

All figures, tables, and abbreviations are listed in the Appendices. The index in the Appendix makes it easier to search for specific key terms and descriptions.

1.2 Other Software Interfaces

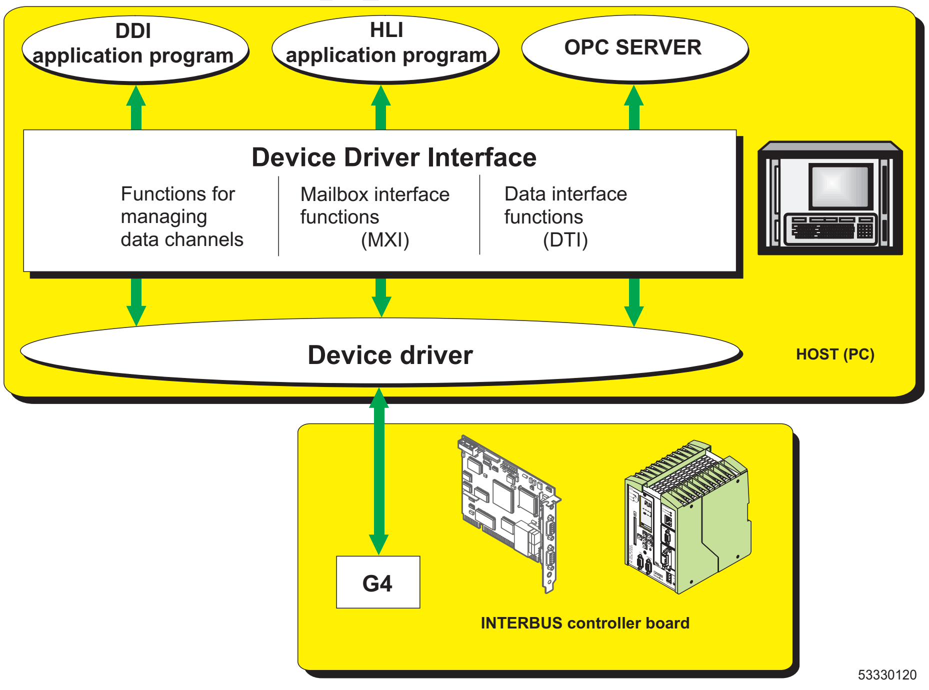

DDI
application program
HLI
application program
OPC SERVER
Device Driver Interface
Functions for managing data channels
Mailbox interface functions (MXI)
Data interface functions (DTI)
Device driver
HOST (PC)
G4
INTERBUS controller board
53330120

Figure 1-1 Structure

The driver software consists of two parts:

1. The Device Driver Interface (DDI), a compiler-specific interface to the application program.
2. The operating system-specific Device Driver (DD).
   The device driver establishes the connection between the host (PC) and the INTERBUS master or slave via the MPM.

A device driver must be installed for each controller board. The Device Driver Interface manages and controls all device drivers.

Table 1-1 Described drivers

Driver Designation	Description
PCI-DPM	PCI driver for controller boards with dual-port memory
PCI-MPM	PCI driver for controller boards with multi-port memory
ISA-DPM	ISA driver for controller boards with dual-port memory
ISA-MPM	ISA driver for controller boards with multi-port memory
PCCARD-DPM	PC card (PCMCIA) driver
TCPIP-ETH	Ethernet driver based on TCP/IP protocol

The V.24 (RS-232) driver is not included in this user manual.

The DDI is used by other software interfaces for programming. The following interfaces are available:

1.2.1 High-Level Language Interface (HLI)

Advantages of the High-Level Language Interface:

– Direct configuration with CMD
– Hardware-independent access to INTERBUS for Windows 32-bit operating systems
– Supports C/C++, Visual Basic, and Delphi programming languages
– Faster and easier data exchange using variable names

– Integrated bus and error management
– Identical access to all controller boards (IBS... SC)
– Automatic PCP communication establishment and monitoring

For additional information, please refer to the HLI data sheet (Designation: DB GB IBS PC SC HLI „High-Level Language Interface Version 2.0“, Order No. 97 88 76 3).

1.2.2 INTERBUS OPC Server

An OPC server (Designation: IBS OPC SERVER, Order No. 27 29 12 7) can also be used as a High-Level Language Interface or as an interface to any visualization system. The OPC server can be used to access INTERBUS data via a standardized software interface.

For additional information, please refer to the OPC server data sheet (Designation: DB GB IBS OPC SERVER, Order No. 97 88 06 4).

1.3 Other Operating Systems

This user manual includes descriptions for DOS and Windows. For other operating systems, such as VxWorks, a driver for individual porting is available in the source code.

Table 1-2 Device Driver Development Kit

Driver Name	Product	Order No.
ISA-MPM	IBS PC DEV KIT G4	28 36 17 5
PCI-MPM	IBS PCI DDK	27 30 27 1
TCPIP-ETH	IBS ETH DDI SWD	27 51 13 7

1.4 Supported Controller Boards

This user manual is currently valid for:

Table 1-3 Supported hardware

Hardware	Order No.	Driver Name
IBS PC 104 SC-T	27 21 70 1	ISA-MPM
IBS PC ISA SC/I-T	27 19 23 4	ISA-MPM
IBS ISA SC/RI/RT/I-T	27 29 18 5	ISA-MPM
IBS ISA SC/RI/RT-LK	27 29 19 8	ISA-MPM
IBS ISA SC/486DX/I-T	27 23 94 5	ISA-MPM
IBS ISA FC/486DX/I-T	27 22 08 5	ISA-MPM
IBS ISA RI/I-T	27 23 07 1	ISA-DPM
IBS PCCARD SC/I-T	27 24 87 6	PCCARD-DPM
IBS PCI SC/I-T	27 25 26 0	PCI-MPM
IBS PCI RI /I-T	27 30 12 9	PCI-DPM
IBS PCI RI-LK	27 04 04 5	PCI-DPM
IBS PCI SC/RI-LK	27 30 18 7	PCI-MPM
IBS PCI SC/RI/I-T	27 30 08 0	PCI-MPM
FL IL 24 BK	28 31 05 7	ETH
FL IL 24 BK-B	28 33 00 0	ETH
FL IBS SC/I-T	28 31 06 0	ETH
FC 200 PCI	27 30 66 6	PCI-MPM
FC 350 PCI ETH	27 30 84 4	ETH
RFC 430 ETH-IB	27 30 19 0	ETH
RFC 450 ETH-IB	27 30 20 0	ETH
ILC 350 ETH	27 37 20 3	ETH

1.5 Additional Documentation

Firmware Commands for Communicating With the Controller Board

The application program controls the controller board via firmware commands (e.g., start bus system, read bus configuration). The controller board operates the bus automatically after initialization and the start of the data transmission and returns appropriate messages.

The IBS SYS FW G4 UM E firmware reference manual (Order No. 27 45 18 5) describes commands, messages, and INTERBUS-specific programming such as:

- The physical addressing of INTERBUS devices
- The logical addressing of INTERBUS devices
- Combining INTERBUS devices into groups
- Determining the cycle time

Communication via INTERBUS (PCP)

The parameter data channel with the Peripherals Communication Protocol (PCP) is available for the transmission of parameterization data to intelligent INTERBUS devices or for communicating with an INTERBUS device with a V24 (RS-232) interface. PCP is a software interface based on the basic protocol of INTERBUS and enables the transmission of large non-time-critical data records without affecting the process data transmission. PCP, like firmware commands, uses the mailbox interface of the DDI.

The IBS SYS PCP G4 UM E user manual (Order No. 27 45 16 9) describes the basics and the use of the Peripherals Communication Protocol.

Support

In the event of problems, please phone our 24-hour hotline on **+49 - 52 35 - 34 18 88**.

Alternatively, you can contact our Support Department by e-mail: **interbus-support@phoenixcontact.com**

Training Courses

Our controller board training courses enable you to take advantage of the full capabilities of the connected INTERBUS system. For details and dates, please see our seminar brochure, which your local Phoenix Contact representative will be happy to mail to you. You can also find up-to-date information on the Internet at www.phoenixcontact.com.

Section 2

This section informs you about

– Interfaces between hardware and software

2 Basics of the Driver Functions

2.1 Multi-Port Memory

The central interface of controller boards is the **M**ulti-**P**ort **M**emory (MPM). The MPM is a memory on the controller board, which can be accessed by all devices (PC and INTERBUS controller board). The MPM is provided in PC card format (IBS PCCARD SC/I-T) as a **D**ual-**P**ort **M**emory (DPM) for INTERBUS slave controller boards and for the controller board). For greater clarity, the term MPM is generally used throughout this manual. MPM and DPM only differ in the number of ports (nodes) and the size of the memory. The devices store all the data that is to be shared in the MPM. The MPM is the only connection between the devices.

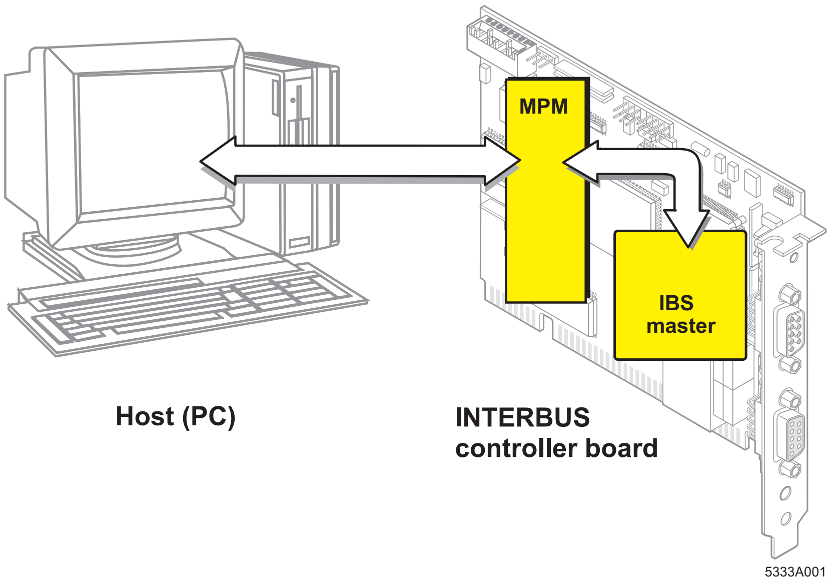


Figure 2-1 The MPM as the central interface of a controller board

Table 2-1 Memory type (MPM/DPM) of controller boards

Hardware	Memory Type	Driver Name
IBS PC ISA SC/I-T IBS PC 104 SC-T IBS ISA SC/RI/RT/I-T IBS ISA SC/RI/RT-LK IBS ISA SC/486DX/I-T	MPM	ISA-MPM
IBS PCI SC/I-T IBS PCI SC/RI-LK IBS PCI SC/RI/I-T	MPM	PCI-MPM
IBS ISA RI/I-T	DPM	ISA-DPM
IBS PCI RI /I-T IBS PCI RI-LK	DPM	PCI-DPM
IBS PCCARD SC/I-T	DPM	PCCARD-DPM
FL IBS SC/I-T IBS 24 ETH DSC/I-T RFC 430 IB RFC 450 IB	DPM	TCP/IP-ETH

The MPM may only be accessed using device driver functions. Direct reading from or writing to the MPM is not permitted.

2.2 Monitoring the Hardware

2.2.1 Watchdog for Host Monitoring

The motherboard of the controller board contains a watchdog circuit that can be used to monitor your PC program (PC system crash, program freeze). When the watchdog is triggered, the INTERBUS system is set to a defined state (reset of all outputs).

☞ The watchdog does not affect the host; a host reset, for example, is not triggered.

If you wish to use the watchdog you must activate it from the application program. It is not activated by default.

The host watchdog is activated by calling the *EnableWatchDog ()* or *SetWatchDogTimeout ()* function. Once the watchdog has been activated, it cannot be deactivated via the software and can only be deactivated by switching off the host or resetting the hardware.

The watchdog can be configured to several timeout intervals. Within the set time, the watchdog must be triggered in the application program by calling the *TriggerWatchDog ()* function. Otherwise, the watchdog causes an INTERBUS system reset.

2.2.2 The SysFail Signal

The *SysFail* signal can be evaluated for increased safety requirements. It is also available in the diagnostic status register. Each MPM device has its own reserved area within the MPM. One of these status signals is the *SysFail* (system failure) signal. It is set in the event of a system error at the corresponding device, e.g., if its watchdog is triggered. With the *GetSysFailRegister* function you can read the *SysFail* signal of any MPM device.

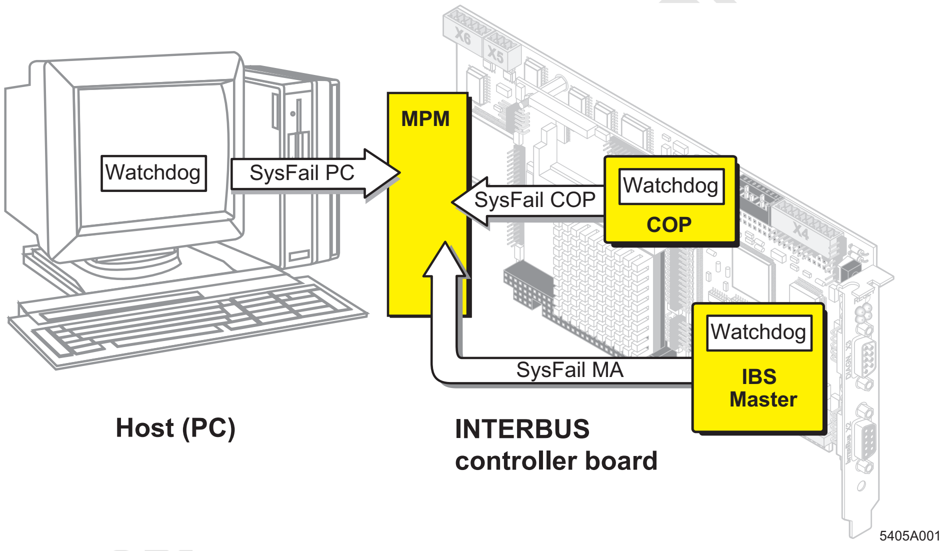


Figure 2-2 The *SysFail* signal in the MPM

2.3 Basic Information on Programming

The driver software accesses the multi-port memory (MPM) or dual-port memory (DPM) of the controller board via a memory window, which is in the memory area of the PC. For this, the following functions are available:

– "Opening and Closing Communication Channels" on page 4-3
– "Reading and Writing I/O Data" on page 4-7
– "Writing Commands and Reading Messages" on page 4-14
– "Diagnostic Functions" on page 4-20
– "Watchdog Functions" on page 4-25
– "Driver Settings and Management" on page 4-31
– "Controller Board Monitoring" on page 4-33
– "Ethernet Communication Monitoring" on page 4-35
– "Other Ethernet Settings" on page 4-50

Every INTERBUS application program should be created according to the following structure (Figure 2-3):

1. The **initialization phase** of the program and the INTERBUS system. In the **initialization phase** the required include files are integrated in the program. Global variables are provided, for example, as the receive buffer for messages from the INTERBUS master and as the buffer for input and output data in the process data area. In this phase the communication connections to the INTERBUS master are also initialized and channels for the mailbox and the data interface are opened (Section 2.4).
2. The second part of **INTERBUS startup** includes the system configuration check, the logical assignment of input and output addresses, the definition of groups, the behavior of groups in the event of an error, and INTERBUS data transmission (Section 2.5).
3. The third part of the application program is the cyclic program part. In this cyclic program part, the system regularly checks for an INTERBUS message. **Diagnostic data** is cyclically **updated** and made available. The process image of the inputs is cyclically read, the input data is linked, and the process image of the outputs is then cyclically written to the bus. This part is processed until the program is exited (Section 2.6).

4. In the fourth part, for example, **bus operation** is **stopped** and the communication channels are closed again (Section 2.7).

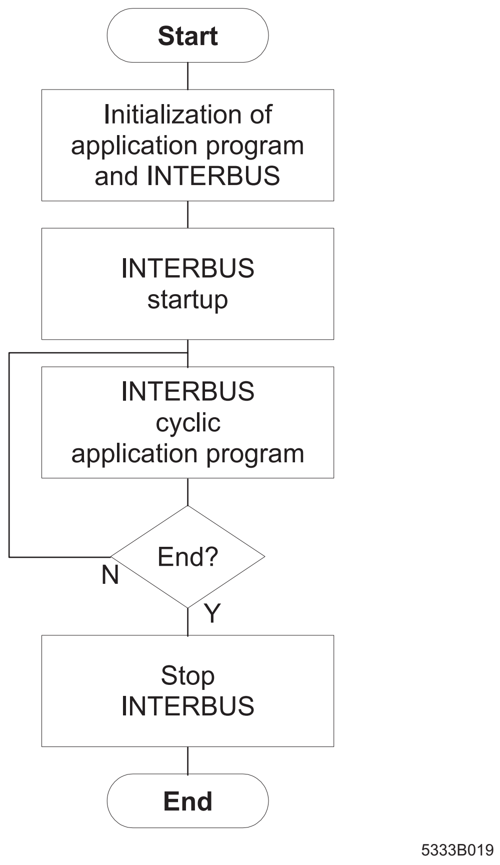

Start
Initialization of application program and INTERBUS
INTERBUS startup
INTERBUS cyclic application program
End?
N
Y
Stop INTERBUS
End
5333B019

Figure 2-3 General structure of an INTERBUS application program

This program structure is used in all example programs from Phoenix Contact. These example programs are supplied with the controller board driver software and can be downloaded from the Internet at www.phoenixcontact.com.

2.4 Initialization Phase

Global variables are provided in the **initialization phase**, for example, as the receive buffer for messages from the INTERBUS master and as the buffer for input and output data in the process data area. In this phase the communication connections to the INTERBUS master are also initialized and channels for the mailbox and data interface are opened.

When a channel is opened, a *node handle* is returned, which specifies the channel in a similar way to the *handle* on a file access and which must be specified when reading and writing data.

In this example, two node handles, one for the mailbox channel and one for the channel, are requested for connection establishment via the driver. These node handles must be specified every time the drivers are accessed.

```
/*****************************************************
* This function opens mailbox and channel for       *
* board number 1, node number 1(INTERBUS)           */

BOOL InitIBSBoard(USIGN16 BoardNo)
{
     USIGN16 ret = 0;

/* The OpenString to the data interface for INTERBUS
/* plug-in board drivers is transferred to the driver
/* and the node handle received is stored in a global
/* variable */

ret = DDI_DevOpenNode("IBB1N1_D",DDI_RW, &NodeHdDTI);
if ( ret != ERR_OK )
{
     cprintf("Error 0x%04X opening DTI: Board No.:
                                  %u \n",ret, BoardNo);
     return (FALSE);
}

/* The OpenString to the mailbox interface for INTERBUS
/* plug-in board drivers is transferred to the driver /
*
/* and the node handle received is stored in a global
/* variable */

ret = DDI_DevOpenNode("IBB1N1_M",DDI_RW,&NodeHdMXI);
if ( ret != ERR_OK )
{
     cprintf("Error 0x%04X opening MXI: Board No.:
                                  %u \n",ret, BoardNo);
     return (FALSE);
}
return (TRUE);
}
```

2.4.1 General Node Addressing

Node

An MPM device with an associated device driver is referred to as a node. The following nodes are used:

Node 0:	Host PC with associated device driver
Node 1:	INTERBUS controller board/slave controller board
Node 2:	Coprocessor board of the INTERBUS controller board
Node 3:	Reserved

Node handle

A node handle identifies a channel open to a node.

Device name

Name of the device to which a channel is to be opened. The name specifies the controller board (board number 1 to 8) or the IP address for communication with the controller board.

2.4.2 Node Addressing for PC Controller Boards

A string is provided to make operation easier when opening a mailbox/ channel for the *device name* parameter, which is used to assign the board number and MPM device.

The *device name* parameter has the following structure: IBB**x**N**y**_**z**.

x	Board number (1 to 8)
y	Node (0, 1 or 2)
z	(M) mailbox or (D) data interface

The *device name* parameter for accessing the master of controller board 3 via the data interface is "IBB3N1_D".

The following tables show examples for different situations:

Table 2-2 Opening a channel from the host to the IBS master/slave

Controller Board (Board Number)	**MPM Device**	**Interface**	**String for the *Device Name* Parameter (devName)**
Board 1	IBS master/slave	Mailbox interface	IBB1N1_M
Board 1	IBS master/slave	Data interface	IBB1N1_D
Board 2	IBS master/slave	Mailbox interface	IBB2N1_M
Board 2	IBS master/slave	Data interface	IBB2N1_D
....			
Board 8	IBS master/slave	Mailbox interface	IBB8N1_M
Board 8	IBS master/slave	Data interface	IBB8N1_D

⚠ If a PCI master controller board and a PCI slave controller board are both installed in a PC at the same time, the PCI slave controller board must be addressed with the prefix "IBS...".

Table 2-3 Opening a channel from the host to the IBS slave (only when simultaneously using PCI master controller boards and PCI slave controller boards)

Controller Board (Board Number)	**MPM Device**	**Interface**	**String for the *Device Name* Parameter (devName)**
Board 1	IBS slave (only PCI)	Mailbox interface	IBS1N1_M
Board 1	IBS slave (only PCI)	Data interface	IBS1N1_D
Board 2	IBS slave (only PCI)	Mailbox interface	IBS2N1_M
Board 2	IBS slave (only PCI)	Data interface	IBS2N1_D
....			
Board 4	IBS slave (only PCI)	Mailbox interface	IBS8N1_M
Board 4	IBS slave (only PCI)	Data interface	IBS8N1_D

Table 2-4 Opening a channel from the host to the coprocessor board

Controller Board (Board Number)	**MPM Device**	**Interface**	**String for the *Device Name* Parameter (devName)**
Board 1	Coprocessor board	Mailbox interface	IBB1N2_M
Board 1	Coprocessor board	Data interface	IBB1N2_D
Board 2	Coprocessor board	Mailbox interface	IBB2N2_M
Board 2	Coprocessor board	Data interface	IBB2N2_D
....			
Board 8	Coprocessor board	Mailbox interface	IBB8N2_M
Board 8	Coprocessor board	Data interface	IBB8N2_D

Table 2-5 Opening a channel from the coprocessor board to the host PC

Controller Board (Board Number)	**MPM Device**	**Interface**	**String for the *Device Name* Parameter (devName)**
Board 1	Host	Mailbox interface	IBB1N0_M
		Data interface	IBB1N0_D

Table 2-6 Opening a channel from the coprocessor board to the IBS master

Controller Board (Board Number)	**MPM Device**	**Interface**	**String for the *Device Name* Parameter (devName)**
Board 1	IBS master	Mailbox interface	IBB1N1_M
		Data interface	IBB1N1_D

2.4.3 Node Addressing for TCP/IP Communication

There are two forms of addressing for establishing a connection via a TCP/IP connection:

Addressing via Registry

Here, a reference to a registry entry is transmitted to the driver. In this example, it is the actual IP address.

The *device name* parameter has the following structure: IBETH**xx**N**y**_**z**.

xx	Registry entry/controller number (0 to 99)
y	Node (0, 1 or 2)
z	(M) mailbox or (D) data interface

In DDI Version 1.20 or later, all services can be sent and received with an open channel in order to reduce TCP/IP handles. No distinction is made between the mailbox and data interface.

Table 2-7 Opening an Ethernet communication channel via registry addressing

Controller Board (Controller Number)	ETH Device	Interface	String for the *Device Name* Parameter (devName)
Controller 1	Master	Mailbox interface	IBETH01N1_M
Controller 1		Data interface	IBETH01N1_D
	....		
Controller 99	Master	Mailbox interface	IBETH99N1_M
Controller 99		Data interface	IBETH99N1_D

The IP address must be entered in the registry. This entry can be easily modified using the "Comway.exe" tool.

The following registry entry is created as an example:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Phoenix
Contact\IBSETH\Parameters\1]
ConnectTimeout=08,00,00,00
DeviceNames=IBETH01N1_M IBETH01N0_M@01 IBETH01N1_D
IBETH01N0_D IBETH01N1_M@00 IBETH01N1_M@05
InUse=YES
ReceiveTimeout=08,00,00,00
IPAddress=192.168.36.205
```

Addressing via the IP Address

A string is provided to simplify operation when opening a mailbox/channel for the *device name* parameter, which contains the IP address. Here, an IP address and other communication parameters are transmitted to the driver.

The *device name* parameter has the following structure:
IBETHIP[**IP address|connect timeout, receive timeout**]N**y**_**z**

IP address	IP address without leading zeros (162.16.50.5)
Connect timeout	Connect timeout in sec (optional)
Receive timeout	Receive timeout in sec (optional)
y	Node (0, 1 or 2)
z	(M) mailbox or (D) data interface

Table 2-8 Opening a communication channel via the direct IP address

IP Address	Interface	String for the *Device Name* Parameter (devName)
162.16.50.1	Mailbox interface	IBETHIP[162.16.50.1]N1_M
	....	
162.16.50.254	Data interface	IBETHIP[162.16.50.254|30,30]N1_M

2.5 INTERBUS Startup

The second part of **INTERBUS startup** includes the system configuration check, the logical assignment of input and output addresses, the definition of groups, the behavior of groups in the event of an error, and the actual startup of INTERBUS.

This function is exchanged via the mailbox system. The mailbox interface (MXI) is a protocol-oriented interface for transmitting messages between MPM devices.
The DDI_ MXI_SendMessage and DDI_MXI_ReceiveMessage functions use the mailbox interface. The description can be found in the firmware user manual.

A detailed description of the services can be found in the "Firmware Services and Error Messages" user manual
(Designation: IBS SYS FW G4 UM E, Order No. 27 45 18 5).

```
/*****************************************************
* A transferred message buffer with all its contents*
* is packed in DDI-specific transfer *
* structures and transferred to the driver via macros*
* Handle: Node handle of the opened mailbox channel*
* MSG: Pointer to an array with the complete message*

IBDDIRET SendRequestResponse (USIGN16 Handle,
                             USIGN16 *Msg)
{
     T_DDI_MXI_ACCESS req_res;
     USIGN8 snd_buffer[1024];
     INT16 ret;
     USIGN16 i;
/* Command and parameter count are copied to the
 local output buffer and simultaneously converted into
 MOTOROLA format via macros*/
     IB_SetCmdCode(snd_buffer,Msg[0]);
     IB_SetParaCnt(snd_buffer,Msg[1]);

/*The n parameters of this message are attached*/
     for (i=1;i<=Msg[1];i++)
     {
          IB_SetParaN(snd_buffer,i,Msg[i+1]);
     }
```

```
/* Transfer structure rq_res of type T_DDI_MXI_ACCESS
is
filled ...*/
     req_res.msgType = 0;
     req_res.DDIUserID = 0;
     req_res.msgLength = (Msg[1]+2)*2;
     req_res.msgBlk = snd_buffer;

/* ... and transferred to the driver*/
     ret = DDI_MXI_SndMessage(Handle, &req_res);
     return (ret);
}

/*****************************************************
* A message is read from the driver and copied to *
* the transferred array *
* Handle: Node handle of the opened mailbox channel*
* MSG: Pointer to an array for the message received*
*/

IBDDIRET ReceiveConfirmationIndication(USIGN16 Handle,
                                       USIGN16 *Msg)
{
     T_DDI_MXI_ACCESS con_ind;
     INT16 ret;
     USIGN16 i;
     USIGN8 rcv_buffer[1024];

/* Receive structure is pre-initialized */
     con_ind.msgType = 0;
     con_ind.DDIUserID = 0;
     con_ind.msgLength = 1024;
     con_ind.msgBlk = rcv_buffer;

/*A message is fetched from the driver ...*/
     ret = DDI_MXI_RcvMessage(Handle, &con_ind);
     if ((ret != ERR_NO_MSG) && (ret == ERR_OK))
     {
/* ... Message code and parameter count are extracted
via macros */
          Msg[0] = IB_GetMsgCode(rcv_buffer);
          Msg[1] = IB_GetParaCnt(rcv_buffer);
/* The parameters are copied to the transferred array
and
converted into INTEL format
          for (i=1;i<=Msg[1];i++)
          {
               Msg[i+1] = IB_GetParaN(rcv_buffer,i);
           }  //for
     } //NO_MESSAGE
     return(ret);
}
```

2.6 Exchanging I/O and Diagnostic Data

The third part of the application program is the cyclic program part. In this program part, the system regularly checks for an INTERBUS message. **Diagnostic data** is updated and made available. The process image of the inputs is read, the input data is linked, and the process image of the outputs is then written to the bus. This part is processed until the program is exited.

The data interface (DTI) is used to transfer I/O data between MPM devices. Transmission takes place with no confirmation (acknowledgment).

Two functions (DDI_DTI_WriteData and DDI_DTI_ReadData) are provided by the DDI driver for reading and writing data. In this way, the transfer memory of the controller board can be accessed easily. In this example, the data areas for inputs and outputs have a maximum length of 512 bytes. Controller boards must be adapted specifically to the length of these data areas.

Since the controller boards expect all data to be word-orientated in Motorola format, the data is copied to the transfer buffer via completed macros (see Section 5, "Programming Support Macros"). The following program sequences explain how to access the controller boards.

```
/*****************************************************
do
   {
     /* Evaluate diagnostics */
     DiagRegIBS();

     /* Any message from Controller? */
     if( ConfirmationIndication(NodeHdMXI_MA, buffer)
      == ERR_OK )
     {
      /* Process indication */
      ………..
      }
     }

     /* Read data from IBS */
     ReadData_M2I(NodeHdDTI_MA, 0, 1, InData);

     /* Application */
     …….. 

     /* Check user keyboard actions */
     if (kbhit())
     {
         key = getch();
         switch (toupper(key))
         {
               case 'E' :      /* Exit program */
               {
                    end = TRUE;
                    break;
                 }
                case  ……..

         } /* End of switch */
     }/* End of kbhit */

     /* Write output data to IBS */
     WriteData_I2M(NodeHdDTI_MA, 0, 1, OutData);

   } while (!end);    /* End of cyclic program */
/*****************************************************
```

2.6.1 Reading Back Outputs

Supported Drivers
All

The written output data for the controller board can be read back in your application program so that several program parts can access a data area in order to assign information, for example. To do this, the IB_TO_REMOTE_DTA flag is available in addition to the DDI_DTI_ReadData function (see "DDI_DTI_ReadData" on page 4-7).

2.6.2 Bit Access

Supported Drivers
PCI-MPM (Version 2.0 or later)

Bit access is available for accessing the data areas using DDI_DTI_WriteData() and DDI_DTI_ReadData(), to gain access to bit objects (4-bit, 2-bit).

The T_DDI_DTI_ACCESS structure functions are extended accordingly:

– The *address* element indicates the byte address to be copied
– The *length* element indicates the number of bits to be copied
– In the *dataCons* element, the bit position of the specified byte is determined and the bit access itself.

In this way it is possible to either enter the corresponding constant or to write the desired bit position in bits 0 - 2 of dataCons and to assign this value using DDI_DATA_BIT.

The maximum length for bit access is limited to 16 bits.

Position of the Data in the IN Buffer (Element Data):

Length <= 8: Right-justified in the first byte of data

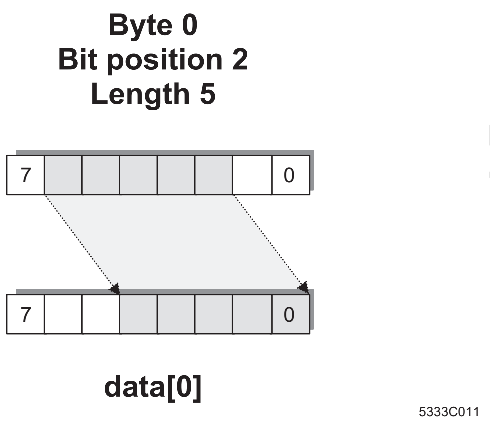


Figure 2-4 Bit access length <= 8

Length > 8: Right-justified from the second byte of data

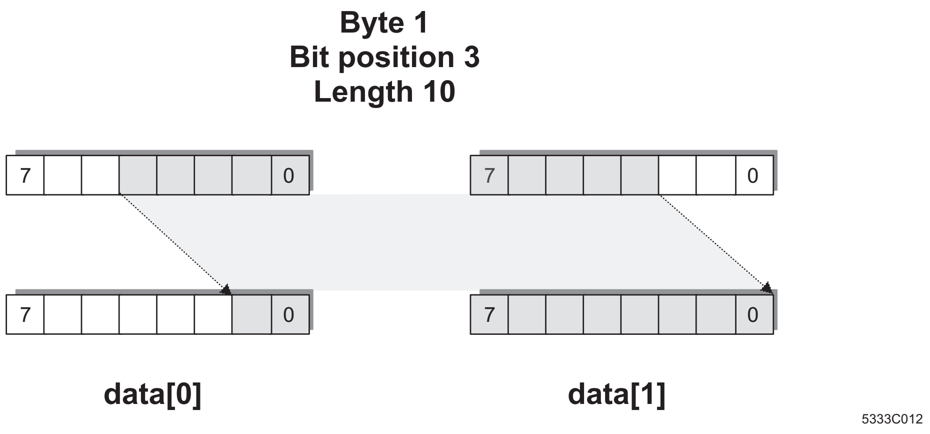


Figure 2-5 Bit access length > 8

Example

Read/write four bits from address 25 starting from position 0:

```
address = 25
length = 4
dataCons = DDI_DATA_BIT_ADDR0 or dataCons =
(DDI_DATA_BIT | 0)
```

Read/write four bits from address 25 starting from position 4:

```
address = 25
length = 4
dataCons = DDI_DATA_BIT_ADDR4 or dataCons =
(DDI_DATA_BIT | 4)
```

Read/write one bit from address 12 starting from position 2:

```
address = 12
length = 1
dataCons = DDI_DATA_BIT_ADDR2 or dataCons =
(DDI_DATA_BIT | 2)
```

The constants are defined in the DDI_USR.H file.

2.6.3 The XDTA Data Area

Supported Drivers
PCI-MPM
ISA-MPM
TCP/IP-ETH

When directly accessing data on Generation 4 Field Controllers, data can be exchanged with the IEC 61131 runtime system via the multi-port memory (MPM) of the controller board. The memory reserved for this purpose in the MPM provides the same size in kbytes for both data directions.

At present the following limitations exist:

– Only byte consistency can be ensured.
– For the length of the data area, please refer to the controller board documentation.

The same Device Driver Interface (DDI) functions are used, which are also available when accessing INTERBUS (DDI_DTI_WriteData, DDI_DTI_ReadData). Please note that different nodes should be addressed depending on the controller board and data direction:

Table 2-9 Constants for XDTA access

Controller Board	Read		Write	
	Node	Constant	Node	Constant
IBS ISA FC/I-T; FC 200 PCI	0	IB_NODE_0	1	IB_NODE_1
IBS ISA FC/486DX/I-T	2	IB_NODE_2	1	IB_NODE_1
RFC 430/450 IB	0	IB_NODE_0	1	IB_NODE_1
ILC 350 ETH	0	IB_NODE_0	1	IB_NODE_1
FC 350 PCI ETH	0	IB_NODE_0	1	IB_NODE_1

Example

```
/** process data write **/
     dtiAcc.length = 1000; /* length in bytes */
     dtiAcc.address = 0;
     dtiAcc.dataCons =
           (IB_EX_DTA | IB_NODE_1 | DTI_DATA_BYTE);
     dtiAcc.data = (USIGN8 *) outBuffer;
     ret = DDI_DTI_WriteData (dtiNodeHd1, &dtiAcc);

/** process data read **/
     dtiAcc.dataCons =
          (IB_EX_DTA | IB_NODE_0 | DTI_DATA_BYTE); //ETH
or   dtiAcc.dataCons =
          (IB_EX_DTA | IB_NODE_0 | DTI_DATA_BYTE); //FC
or   dtiAcc.dataCons =
          (IB_EX_DTA | IB_NODE_2 | DTI_DATA_BYTE); //COP
     dtiAcc.data = (USIGN8 *) inBuffer;
     ret = DDI_DTI_ReadData (dtiNodeHd1, &dtiAcc);
     Sleep (10);
```

2.6.4 Direct Inputs/Outputs

Supported Drivers
PCI-MPM

This function only applies to PCI master controller boards.

Direct inputs/outputs are available in the hardware IO registers on PCI-SC controller boards. They can be addressed from the control program on the PC without parameterizing the INTERBUS master or starting INTERBUS.

For the number of inputs/outputs, please refer to the corresponding hardware data sheet.

This information is also exchanged via the DDI_DTI_ReadData or DDI_DTI_WriteData DDI functions (see also Section 4.2).

2.6.5 INTERBUS Diagnostic Register

The diagnostic data of the controller boards is read via the GetIBSDiagnostic and GetIBSDiagnosticEx functions. The return parameters display the following information:

INTERBUS Master Diagnostics

The controller board has three registers in the MPM for analyzing error states using the application program:

– The diagnostic status register returns the operating and error states of the controller board.

– The diagnostic parameter register provides additional information about the type of error or the error location.

– The extended diagnostic parameter register provides channel-specific information about the single-channel diagnostics.

Master diagnostic status register

Each bit in the master diagnostic status register is assigned a controller board state. The states in the error bits (USER, PF, BUS, CTRL) are described in more detail using the diagnostic parameter register. Whenever an error bit is set, the diagnostic parameter register is rewritten. Otherwise, the diagnostic parameter register has the value 0000$_{hex}$.

Table 2-10 The master diagnostic register

Bit	Constant	Meaning
0	USER_BIT	Error in the application program.
1	PF_BIT	INTERBUS device has detected a peripheral fault.
2	BUS_BIT	Error in the remote bus or local bus.
3	CTRL_BIT	Controller board has an internal error.
4	DETECT_BIT	Error localization ("LOOK FOR FAIL").
5	RUN_BIT	Data cycles are being exchanged.
6	ACTIVE_BIT	Controller board is ACTIVE.
7	READY_BIT	Controller board is READY, selftest is complete.
8	BSA_BIT	One or more bus segments are switched off.
9	BASP_BIT	Controller board has activated the "SysFail signal". Output data is reset.
10	RESULT_BIT	The result of processing a service sent via standard functions was negative.
11	SYNCHRON_ RESULT_BIT	INTERBUS master does not receive a synchronization pulse. Only in synchronous operation.
12	DATA_CYCLE_ RESULT_BIT	Data cycle error. Only in synchronous operation.
13	WARN_BIT	Bus warning time has elapsed (can be parameterized).
14	QUALITY_BIT	Bus quality has deteriorated (can be parameterized).
15	SS_INFO_BIT	Message in the standard signal interface.

Operating indicators: READY, ACTIVE, RUN

The READY, ACTIVE, and RUN operating indicators show the current state of the INTERBUS system. The diagnostic parameter register is not used.

After the selftest, the controller board is ready for operation. The READY indicator bit is set (READY = 1).

If the controller board has been configured and the configuration frame activated without errors, the system indicates it is active. The READY and ACTIVE indicator bits are set (READY = 1, ACTIVE = 1).

In addition, the RUN indicator bit is set when data exchange is started (READY = 1, ACTIVE = 1 and RUN = 1).

Error indicators: DETECT, CTRL, BUS, PF, USER

The DETECT error bit indicates that an error is preventing further operation of the bus (DETECT = 1). The outputs fall back to the value ZERO. The diagnostic routine searches for the error cause.

Once the error cause has been detected, the DETECT error bit is reset (DETECT = 0) and the error is indicated in the USER, PF, BUS, and CTRL bits. The diagnostic parameter register and the extended diagnostic parameter register describe the cause of the error in more detail.

Table 2-11 Errors with bus disconnection

Error Bit/Location	Contents of the Diagnostic Parameter Register
CTRL = 1 Probably controller board/hardware error.	Error code
BUS = 1 Error on the remote bus or local bus segment.	Error location

Table 2-12 Errors without bus disconnection

Error Bit/Location	Contents of the Diagnostic Parameter Register
PF = 1 Fault on the application side of an INTERBUS device: – Short circuit at the output – Sensor/actuator supply not present	Error location
USER = 1 User error, e.g., due to incorrect parameters	Error code

Error location

For located remote or local bus errors, the diagnostic parameter register contains the error location:

5. Error on local bus: Device number of the device, e.g., "1.3" for bus segment 1; device 3
6. Error on remote bus: Device number of the bus terminal module, e.g., "1.0" for bus segment 1; device 0

Error code,
e.g., address overlap
(code $0A50_{hex}$)

Error location,
e.g., device number 3.1

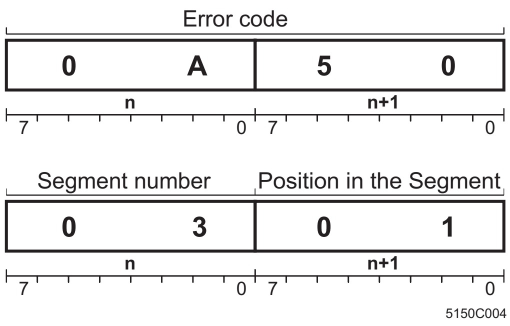


Figure 2-6 Contents of the diagnostic parameter register (example)

The precise error location is only specified if there is no interface error (bit 7 equals 0). If an interface error has occurred (bit 7 equals 1), for example, the connected bus cannot be operated, only the faulty bus segment is specified. Bit 0 indicates whether the error location is on the outgoing remote bus interface (bit 0 equals 0) or on the branching remote bus interface (bit 0 equals 1).

INTERBUS Extended Diagnostics

The extended diagnostic parameter register is used for single-channel diagnostics. A distinction is made between a channel-specific and a group-specific diagnostic message using bits 14 and 15 of the extended diagnostic parameter register.

Extended diagnostic parameter register

Binary Code	Meaning
01xx xxxx	Channel-specific diagnostic message
10xx xxxx	Group-specific diagnostic message

Channel-specific diagnostic message

Bit	7	6	5	4	3	2	1	0
	0	1	T	C	C	C	C	C

Key:

C	Channel number, encoded in bits 0 to 4
T	= 0: Error removed = 1: Error present

Group-specific diagnostic message

Bit	7	6	5	4	3	2	1	0
	1	0	0	0	G4	G3	G2	G1
	1	0	0	1	G8	G7	G6	G5
	1	0	1	0	U4	U3	U2	U1

Key:

G1 to G4	Sensor supply to groups 1 to 4
G5 to G8	Sensor supply to groups 5 to 8
U1 to U4	Voltage supply U1 to U4
	= 0: No error has occurred = 1: Error has occurred

With the group-specific diagnostic message, each bit specifies the status of a group.

INTERBUS Slave Diagnostics

The slave diagnostic register indicates the status of the higher-level INTERBUS system for slave controller boards and system couplers.

Slave diagnostic status register

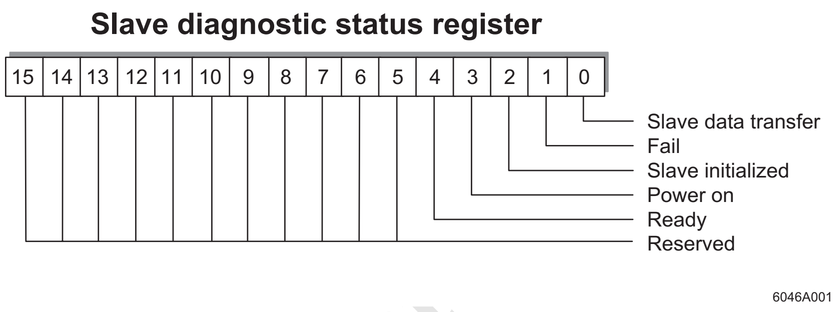


Figure 2-7 Slave diagnostic status register

Table 2-13 The slave diagnostic register

Bit	Constant	Meaning
0	SD_SLAVE_DATA _TRANSFER	A data exchange takes place with the slave controller board.
1	SD_FAIL	A peripheral fault has occurred on the slave controller board.
2	SD_SLAVE_ INITIALIZED	The controller board initialization is complete.
3	SD_POWER_ON	The supply voltage is present.
4	SD_READY	The slave controller board is in the "READY" state.
5 to 15	Reserved	-

2.7 Stopping Bus Operation and Aborting a Connection

In the fourth part, for example, **bus operation** is stopped and the communication channels are closed again.

```
// Stop INTERBUS
   RequestResponse(NodeHdMXI_MA, ALARM_STOP);
   Verify_CNF(0x1303);

// Close nodes
// Close DTI (data interface) nodes
   DDI_DevCloseNode(D_Handle);

// Close MXI (mailbox interface) nodes
   DDI_DevCloseNode(D_Handle);
```

Section 3

This section informs you about
– the operating systems supported by the drivers

3 Driver Basics

3.1 Driver Overview

The following table shows which operating systems and hardware are supported by drivers:

Table 3-1 Driver support of operating systems

Hardware (Examples)	Driver	MS-DOS	Windows 95/98	Windows NT 4.0	Windows 2000	Windows XP
IBS PC ISA SC/I-T IBS PC 104 SC-T IBS ISA SC/RI/RT/I-T IBS ISA SC/RI/RT-LK	**ISA-MPM**	✓	✓	✓	✓	-
IBS ISA SC/486DX/I-T		-	-	✓	✓	-
IBS PCCARD SC/I-T	**PCCARD -DPM**	-	-	✓	✓	✓
IBS ISA RI/I-T	**ISA-DPM**	✓	-	✓	✓	-
IBS PCI RI-LK IBS PCI RI/I-T	**PCI-DPM**	-	-	✓	✓	✓
IBS PCI SC/I-T IBS PCI SC/RI-LK IBS PCI SC/RI/I-T	**PCI-MPM**	-	-	✓	✓	✓
FL IBS SC/I-T FL IL 24 BK IBS 24 ETH DSC/I-T	**TCP/IP-ETH**	-	✓	✓	✓	✓

3.1.1 Headers

Various headers and libraries are provided to simplify programming.

IBS_DOS.H

This file is linked to the following libraries:

#include <STDTYPES.H>

#include <DDI_USR.H>

#include <DDI_ERR.H>

#include <DDI_MACR.H>

#include <IBS_UTIL.H>

#include <IBS_CM.H>

For Windows, instead of "IBS_DOS.H", "IBS_WIN.H" contains additional "DDI_WIN".H" libraries.

STDTYPES.H

This file contains standard types for programs from Phoenix Contact.

DTI_USER.H

All prototypes that belong to DTI driver functions are declared in this file.

DTI_ERR.H

This file contains the definitions of error codes that return driver functions. They are defined as symbolic constants here.

DTI_MACR.H

This file contains the macros for converting Intel format to Motorola format or for converting from Motorola format to standard Intel format used on the PC.

IBS_ULIT.H

This file contains prototypes and declarations for the diagnostics, watchdog, and static RAM utility functions.

IBS_CM.H

The file contains definitions of symbolic constants for INTERBUS commands and INTERBUS messages.

3.1.2 Supported Programming Languages

The drivers provide interfaces for various programming languages. The supported programming languages depend on the operating system used:

Table 3-2 Supported programming languages

	MS-DOS	Windows 95/98	Windows NT 4.0	Windows 2000	Windows XP
C, C++	✓	✓	✓	✓	✓
Pascal	✓	-	✓	✓	-
Delphi	-	✓	✓	✓	✓
Visual Basic	-	✓	✓	✓	✓

Only the driver syntax for "C" is described in this user manual. A description of the syntax and the interfaces for other supported programming languages can be found in the directory of corresponding libraries or units.

3.2 Drivers for the Operating System

3.2.1 Drivers for MS-DOS

Table 3-3 Drivers for MS-DOS

	Driver	Library	Controller Boards
ISA-MPM	IBSISA.EXE	LDDI_TSR.LIB	8
ISA-DPM	IBDPMDRV.EXE	DPMDRVTL.LIB	4

The ISA-MPM and ISA-DPM drivers cannot be used at the same time under MS-DOS.

Structure of the ISA-MPM/ISA-DPM Driver Software

The Device Driver Interface must be linked to your application program in the form of a library. The device driver for DOS is implemented as a TSR program. If it is to be started on every system startup, the "autoexec.bat" file should be adapted accordingly.

A device driver, i.e., a TSR program, must be installed for every controller board.

Libraries and Include Files

Libraries

The DDI and help functions are combined in one library. The library is available in the large model (Microsoft C, Version 7.0 or later and Borland C++ Version 3.0 or later).

The driver libraries are compatible with Microsoft C. You can also use the driver libraries with Borland C by converting them to a Borland-compatible format. This can be done, for example, using the Borland IMPLIB and IMPDEV tools.

Include files

To simplify the handling of include files, the *IBS_DOS.H* include file must be attached. All other required include files are called from this include file. However, you can also call the required include files individually.

Table 3-4 Libraries and include files

Memory Model	Library	Include File
Large	LDDI_TSR.LIB	IBS_DOS.H
Medium	MDTI_TSR.LIB	IBS_DOS.H

3.2.2 Drivers for Windows 95/98

Table 3-5 Drivers for Windows 95/98

	Driver	Library	Controller Boards
ISA-MPM	VIBSSCD.VXD	IBDDIW95.DLL	8
TCP/IP-ETH	IBSETHETH.DLL	IBDDIW95.DLL	256

Copy the *IBDDIW95.DLL* and *VIBSSCD.VXD* files, as normal under Windows, to the directory that contains your application program or to the Windows root directory.

Structure of the ISA-MPM Driver Software

The driver software for Microsoft Windows 95/98 is executed as a **D**ynamic **L**ink **L**ibrary (DLL) using a virtual device driver (VxD). The DLL (IBDDIW95.DLL) contains the Device Driver Interface. This DLL calls the virtual device driver (VIBSSCD.VXD) during runtime.

The device drivers for eight controller boards are integrated in the *VIBSSCD.VXD* file, which must be entered in the Windows registry database and parameterized. These entries, which are usually automatically executed by a driver setup, can also be modified using the Windows 95/98 registry editor, if required.

The path in the registry is: HKEY_LOCAL_MACHINES\System\CurrentControlSet\Services\VxD\IBSISASC. This contains a subkey (*Parameters*) in which there is another subkey for every controller board (*1, 2, ... 8*) (see Figure 3-1). All the parameters required to initialize the controller board are entered in this other subkey.

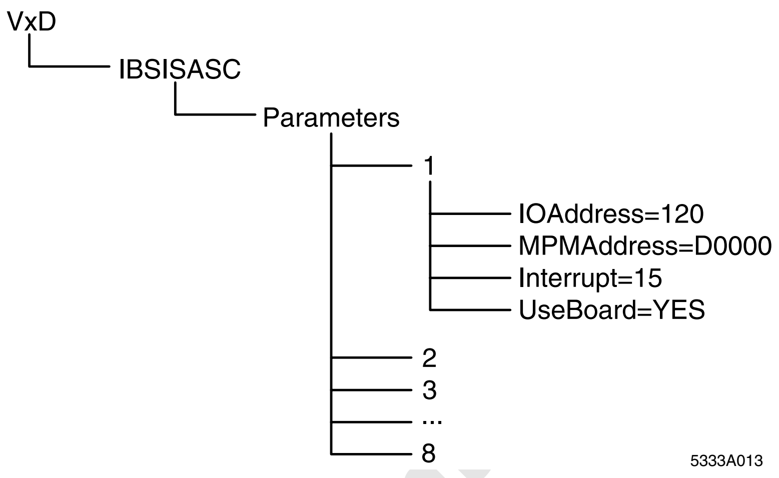


Figure 3-1 Path for parameter setting in the Windows 95/98 registry database

Structure of the ETH Driver Software

With TCP/IP-based access via Ethernet, the DDI directly accesses the TCP/IP sockets of the operating system. The reference to the current IP address is either stored in the registry or can be transmitted directly via the DDI.

Library Under Windows 95/98

Only one **D**ynamic **L**ink **L**ibrary is required to operate the controller boards under Microsoft Windows® *IBDDIW95.DLL*).

3.2.3 Drivers for Windows NT 4.0

Table 3-6 Drivers for Windows NT 4.0

	Driver	Library	Controller Boards
ISA-MPM	IBSISASC.SYS	IBDDIWNT.DLL	8*
PCI-MPM	IBPCIMPM.SYS	IBDDIWNT.DLL	8*
PCCARD-DPM	IBPCCARD.SYS	IBDDIWNT.DLL	1*
ISA-DPM	IBDPMDRV.SYS	IBDDIWNT.DLL	4*
TCP/IP-ETH	IBSETHETH.DLL	IBDDIWNT.DLL	256
PCI-DPM	IBPCIDPM.DLL CIF32DLL.DLL CIFDRV.SYS	IBDDIWNT.DLL	4*

* The maximum number of controller boards may be lower depending on the resources of the corresponding system. Should a resource conflict occur when the driver is started, the number of controller boards for this system is too high.

When using different drivers under Windows NT, different board numbers must be specified for each controller board so that every controller board can be clearly identified.

Administrator rights are required to install the kernel mode driver for Windows NT.

Structure of the ISA-MPM/PCI-MPM/PCCARD-DPM/ISA-DPM Driver Software

The driver software for Microsoft Windows NT is executed as a **D**ynamic **L**ink **L**ibrary (DLL) using a kernel mode driver. The DLL (IBDDIWNT.DLL) contains the Device Driver Interface. This DLL calls the kernel mode driver during runtime.

Structure of the PCI-DPM Driver Software

The driver software for Microsoft Windows NT is executed as a **D**ynamic **L**ink **L**ibrary (DLL) using a kernel mode driver. The DLL (IBDDIWNT.DLL) contains the Device Driver Interface. This DLL calls an interface DLL (IBPCIDPM.DLL). This interface DLL accesses a hardware communication DLL (CIF32DLL.DLL), which corresponds to the kernel mode driver (CIFDRV.SYS).

The device drivers must be entered in the Windows NT registry database and parameterized. These entries in the Windows NT registry database are automatically executed by a driver setup and can be modified by executing the driver setup again or using the Windows NT registry editor, if required.

The path in the registry is:
HKEY_LOCAL_MACHINES\System\CurrentControlSet\Services\IB... .
This contains a subkey (*Parameters*) in which there is another subkey for every controller board (*1, 2, ... 8*). All the parameters required to initialize the controller board are entered in this other subkey.

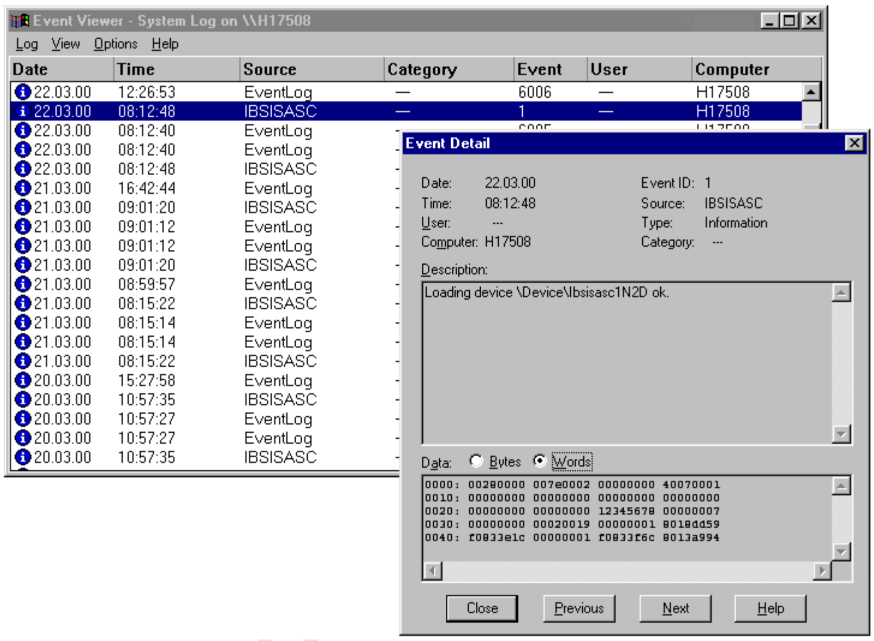
Structure of the ETH Driver Software

With TCP/IP-based access via Ethernet, the DDI directly accesses the TCP/IP sockets of the operating system. The reference to the current IP address is either stored in the registry or can be transmitted directly via the DDI.

Event Display Functions

The drivers for INTERBUS controller boards are started automatically on every system startup. The Windows NT Event Display provides information about driver startup and messages concerning errors starting the driver. The Event Display can be found in the Start menu under Programs/Management (General)/Event Display.

The following diagram shows the event details: The driver for controller board 1 has been loaded successfully.



Date	Time	Source	Category	Event	User	Computer
22.03.00	12:26:53	EventLog	—	6006	—	H17508
22.03.00	08:12:48	IBSISASC	—	1	—	H17508
22.03.00	08:12:40	EventLog	-			
22.03.00	08:12:40	EventLog	-			
22.03.00	08:12:48	IBSISASC	-			
21.03.00	16:42:44	EventLog	-			
21.03.00	09:01:20	IBSISASC	-			
21.03.00	09:01:12	EventLog	-			
21.03.00	09:01:12	EventLog	-			
21.03.00	09:01:20	IBSISASC	-			
21.03.00	08:59:57	EventLog	-			
21.03.00	08:15:22	IBSISASC	-			
21.03.00	08:15:14	EventLog	-			
21.03.00	08:15:14	EventLog	-			
21.03.00	08:15:22	IBSISASC	-			
20.03.00	15:27:58	EventLog	-			
20.03.00	10:57:35	IBSISASC	-			
20.03.00	10:57:27	EventLog	-			
20.03.00	10:57:27	EventLog	-			
20.03.00	10:57:35	IBSISASC	-			

Date:	22.03.00	Event ID:	1
Time:	08:12:48	Source:	IBSISASC
User:	---	Type:	Information
Computer:	H17508	Category:	---

Description:

Loading device \Device\Ibsisasc1N2D ok.

Data: Bytes [✓ Words]

```
0000: 00280000 007e0002 00000000 40070001
0010: 00000000 00000000 00000000 00000000
0020: 00000000 00000000 12345678 00000007
0030: 00000000 00020019 00000001 8018dd59
0040: f0833e1c 00000001 f0833f6c 8013a994
```

Figure 3-2 Example of the Event Display

Driver Settings and Diagnostics

If the IBDRVCFG DDI driver diagnostics control has been installed on your system, the "INTERBUS Driver" driver diagnostic tool can be found in the Control Panel.
This tool can be used to check and adjust the main registry settings. For more detailed information, please refer to the relevant online help.

3.2.4 Drivers for Windows 2000/XP

Table 3-7 Drivers for Windows 2000/XP

	Driver	Library	Controller Boards (2000/XP)
ISA-MPM	IBSISASC.SYS	IBDDIWNT.DLL	8/-*
PCI-MPM	IBPCIMPM.SYS	IBDDIWNT.DLL	8/8*
ISA-DPM	IBDPMDRV.SYS	IBDDIWNT.DLL	4/-*
PC CARD	IBPCCARD.SYS	IBDDIWNT.DLL	1/1
TCP/IP-ETH	IBSETH.DLL	IBDDIWNT.DLL	16/16
PCI-DPM	IBPCIDPM.DLL	IBDDIWNT.DLL	4/4*

* The maximum number of controller boards may be lower depending on the resources of the corresponding system. Should a resource conflict occur when the driver is started, the number of controller boards for this system is too high.

When using different drivers under Windows 2000/XP, different board numbers must be specified for each controller board so that every controller board can be clearly identified.

Administrator rights are required to install the kernel mode driver for Windows 2000/XP.

Structure of the ISA-MPM/PCI-MPM/PCCARD-DPM/ISA-DPM Driver Software

The driver software for Microsoft Windows NT is executed as a **D**ynamic **L**ink **L**ibrary (DLL) using a kernel mode driver. The DLL (IBDDIWNT.DLL) contains the Device Driver Interface. This DLL calls the kernel mode driver during runtime. The kernel mode driver is designed as a WDM driver.

Structure of the PCI-DPM Driver Software

The driver software for Microsoft Windows NT is executed as a **D**ynamic **L**ink **L**ibrary (DLL) using a kernel mode driver. The DLL (IBDDIWNT.DLL) contains the Device Driver Interface. This DLL calls an interface DLL (IBPCIDPM.DLL). This interface DLL accesses a hardware communication DLL (CIF32.DLL), which corresponds to the kernel mode driver (CIFDRV.SYS). The kernel mode driver is designed as a WDM driver.

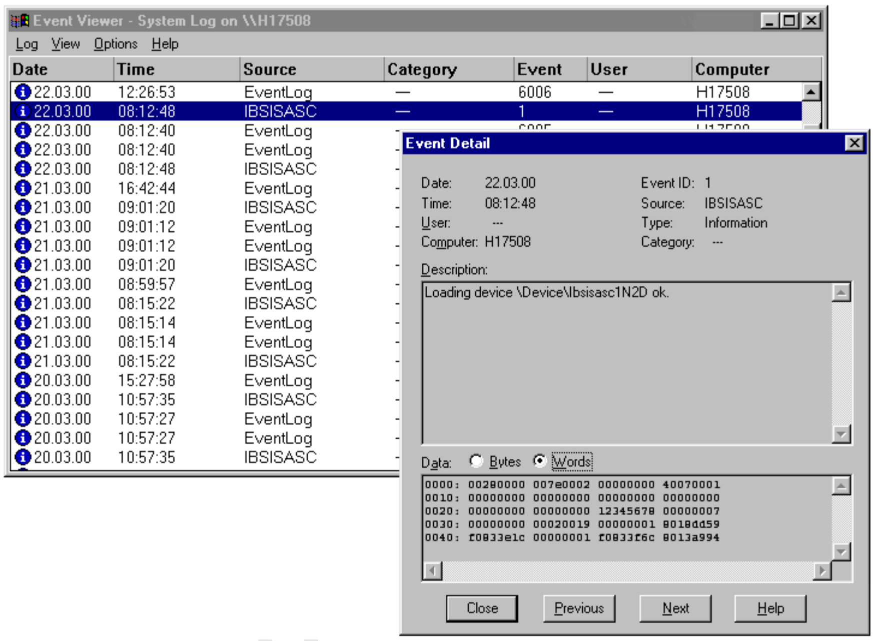
As the PCI-MPM driver for Windows 2000/XP is installed with the aid of the Found New Hardware Wizard, the include and header files must be copied manually from the "[Drive:\install\driver\win2000\ddi]" directory, for example, to an appropriate directory.

Structure of the ETH Driver Software

With TCP/IP-based access via Ethernet, the DDI directly accesses the TCP/IP sockets of the operating system. The reference to the current IP address is either stored in the registry or can be transmitted directly via the DDI.

Event Display Functions

The drivers for INTERBUS controller boards are started automatically on every system startup. Information about driver startup and messages concerning errors starting the driver can be displayed in the Event Display. For example, in Windows 2000/XP, the Event Display can be found in the control panel under "Administrative Tools".



Date	Time	Source	Category	Event	User	Computer
22.03.00	12:26:53	EventLog	—	6006	—	H17508
22.03.00	08:12:48	IBSISASC	—	1	—	H17508
22.03.00	08:12:40	EventLog	-			
22.03.00	08:12:40	EventLog	-			
22.03.00	08:12:48	IBSISASC	-			
21.03.00	16:42:44	EventLog	-			
21.03.00	09:01:20	IBSISASC	-			
21.03.00	09:01:12	EventLog	-			
21.03.00	09:01:12	EventLog	-			
21.03.00	09:01:20	IBSISASC	-			
21.03.00	08:59:57	EventLog	-			
21.03.00	08:15:22	IBSISASC	-			
21.03.00	08:15:14	EventLog	-			
21.03.00	08:15:14	EventLog	-			
21.03.00	08:15:22	IBSISASC	-			
20.03.00	15:27:58	EventLog	-			
20.03.00	10:57:35	IBSISASC	-			
20.03.00	10:57:27	EventLog	-			
20.03.00	10:57:27	EventLog	-			
20.03.00	10:57:35	IBSISASC	-			

Date: 22.03.00 Event ID: 1
Time: 08:12:48 Source: IBSISASC
User: --- Type: Information
Computer: H17508 Category: ---

Description:
Loading device \Device\Ibsisasc1N2D ok.

Data: Bytes [✓ Words]

```
0000: 00280000 007e0002 00000000 40070001
0010: 00000000 00000000 00000000 00000000
0020: 00000000 00000000 12345678 00000007
0030: 00000000 00020019 00000001 8018dd59
0040: f0833e1c 00000001 f0833f6c 8013a994
```

Figure 3-3 Example of the Event Display

Driver Settings and Diagnostics

If the IBDRVCFG DDI driver diagnostics control has been installed on your system, the "INTERBUS Driver" driver diagnostic tool can be found in the control panel.
This tool can be used to check and adjust the main registry settings. For more detailed information, please refer to the relevant online help.

3.3 Driver-Specific Information

3.3.1 Driver Functions of the ISA-MPM Driver

The following table shows which driver functions are supported by the controller boards. It also indicates the driver version from which the driver function is available.

Table 3-8 Driver functions of the ISA-MPM driver

	MS-DOS as of Version	Windows 95/98 as of Version	Windows NT as of Version	Windows 2000/XP as of Version	P.
DDI_DevOpenNode	1.00	1.00	1.00	1.15	4-3
DDI_DevCloseNode	1.00	1.00	1.00	1.15	4-6
DDI_DTI_ReadData	1.00	1.00	1.00	1.15	4-7
DDI_DTI_WriteData	1.00	1.00	1.00	1.15	4-10
DDI_MXI_SndMessage	1.00	1.00	1.00	1.15	4-14
DDI_MXI_RcvMessage	1.00	1.00	1.00	1.15	4-17
GetIBSDiagnostic	1.00	1.00	1.00	1.15	4-20
GetIBSDiagnosticEx	1.04	-	1.06	1.15	4-22
EnableWatchDog	1.00	1.00	1.00	1.15	4-25
TriggerWatchDog	1.00	1.00	1.00	1.15	4-25
GetWatchDogState	1.00	1.00	1.00	1.15	4-26
ClearWatchDog	1.00	1.00	1.00	1.15	4-27
SetWatchDogTimeout	1.00	1.00	1.00	1.15	4-27
GetWatchDogTimeout	1.00	1.00	1.00	1.15	4-29
EnableWatchDogEx	1.05	-	1.04	1.15	4-29
DDIGetInfo	1.02	-	1.04	1.15	4-31

Table 3-8 Driver functions of the ISA-MPM driver

	MS-DOS as of Version	Windows 95/98 as of Version	Windows NT as of Version	Windows 2000/XP as of Version	P.
GetSysFailRegister	1.00	1.00	1.00	1.15	4-33
ClearSysFailSignal	1.03	-	-	-	4-34
SetSysFailSignal	1.03	-	-	-	4-34

3.3.2 Driver Functions of the PCI-MPM Driver

The following table shows which driver functions are supported by the controller boards. It also indicates the driver version from which the driver function is available.

Table 3-9 Driver functions of the PCI-MPM driver

	Windows NT as of Version	Windows 2000/XP as of Version	Page
DDI_DevOpenNode	2.0	2.0	4-3
DDI_DevCloseNode	2.0	2.0	4-6
DDI_DTI_ReadData	2.0	2.0	4-7
DDI_DTI_WriteData	2.0	2.0	4-10
DDI_MXI_SndMessage	2.0	2.0	4-14
DDI_MXI_RcvMessage	2.0	2.0	4-17
GetIBSDiagnostic	2.0	2.0	4-20
GetIBSDiagnosticEx	2.0	2.0	4-22
GetSlaveDiagnostic	2.0	2.0	4-24
EnableWatchDog	2.0	2.0	4-25
TriggerWatchDog	2.0	2.0	4-25
GetWatchDogState	2.0	2.0	4-26
ClearWatchDog	1.05	1.08	4-27

Table 3-9 Driver functions of the PCI-MPM driver

	Windows NT as of Version	Windows 2000/XP as of Version	Page
SetWatchDogTimeout	1.05	1.08	4-27
GetWatchDogTimeout	1.05	1.08	4-29
EnableWatchDogEx	1.05	1.08	4-29
DDIGetInfo	1.05	1.08	4-31
GetSysFailRegister	1.05	1.08	4-33
ReadResetCounter	1.05	1.08	4-32

3.3.3 Driver Functions of the PCCARD-DPM Driver

The following table shows which driver functions are supported by the controller board. It also indicates the driver version from which the driver function is available.

Table 3-10 Driver functions of the PCCARD-DPM driver

Driver Functions	Windows NT as of Version	Windows 2000/XP as of Version	Page
DDI_DevOpenNode	1.01	1.00	4-3
DDI_DevCloseNode	1.01	1.00	4-6
DDI_DTI_ReadData	1.01	1.00	4-7
DDI_DTI_WriteData	1.01	1.00	4-10
DDI_MXI_SndMessage	1.01	1.00	4-14
DDI_MXI_RcvMessage	1.01	1.00	4-17
GetIBSDiagnostic	1.01	1.00	4-20
GetIBSDiagnosticEx	1.01	1.00	4-22
EnableWatchDog	1.01	1.00	4-25
TriggerWatchDog	1.01	1.00	4-25
GetWatchDogState	1.01	1.00	4-26

Table 3-10 Driver functions of the PCCARD-DPM driver

Driver Functions	Windows NT as of Version	Windows 2000/XP as of Version	Page
ClearWatchDog	1.01	1.00	4-27
SetWatchDogTimeout	1.01	1.00	4-27
GetWatchDogTimeout	1.01	1.00	4-29
EnableWatchDogEx	1.01	1.00	4-29
DDIGetInfo	1.01	1.00	4-31
GetSysFailRegister	1.01	1.00	4-33

⚠ The signal interface bit is always set in the standard function register for PCCARD-DPM driver mailbox access.

3.3.4 Driver Functions of the ISA-DPM Driver

The following table shows which driver functions are supported by the ISA-DPM driver. It also indicates the driver version from which the driver function is available.

Table 3-11 Driver functions of the ISA-DPM driver

	MS-DOS as of Version	Windows NT as of Version	Windows 2000 as of Version	Page
DDI_DevOpenNode	1.04	1.04	1.04	4-3
DDI_DevCloseNode	1.04	1.04	1.04	4-6
DDI_DTI_ReadData	1.04	1.04	1.04	4-7
DDI_DTI_WriteData	1.04	1.04	1.04	4-10
DDI_MXI_SndMessage	1.04	1.04	1.04	4-14
DDI_MXI_RcvMessage	1.04	1.04	1.04	4-17
GetSlaveDiagnostic	1.04	1.04	1.04	4-24
EnableWatchDog	1.04	1.04	1.04	4-25
TriggerWatchDog	1.04	1.04	1.04	4-25
GetWatchDogState	1.04	1.04	1.04	4-26
ClearWatchDog	1.04	1.04	1.04	4-27
SetWatchDogTimeout	1.04	1.04	1.04	4-27
GetWatchDogTimeout	1.04	1.04	1.04	4-29
DDIGetInfo	1.04	1.04	1.04	4-31
GetSysFailRegister	1.04	1.04	1.04	4-33

3.3.5 Driver Functions of the PCI-DPM Driver

The following table shows which driver functions are supported by the PCI-DPM driver. It also indicates the driver version from which the driver function is available.

Table 3-12 Driver functions of the PCI-DPM driver

	MS-DOS as of Version	Windows NT as of Version	Windows 2000/XP as of Version	Page
DDI_DevOpenNode	1.01	1.01	1.01	4-3
DDI_DevCloseNode	1.01	1.01	1.01	4-6
DDI_DTI_ReadData	1.01	1.01	1.01	4-7
DDI_DTI_WriteData	1.01	1.01	1.01	4-10
DDI_MXI_SndMessage	1.01	1.01	1.01	4-14
DDI_MXI_RcvMessage	1.01	1.01	1.01	4-17
GetSlaveDiagnostic	1.01	1.01	1.01	4-24
EnableWatchDog	1.01	1.01	1.01	4-25
TriggerWatchDog	1.01	1.01	1.01	4-25
GetWatchDogState	1.01	1.01	1.01	4-26
ClearWatchDog	1.01	1.01	1.01	4-27
SetWatchDogTimeout	1.01	1.01	1.01	4-27
GetWatchDogTimeout	1.01	1.01	1.01	4-29
DDIGetInfo	1.01	1.01	1.01	4-31
GetSysFailRegister	1.01	1.01	1.01	4-33

3.3.6 Driver Functions of the TCPIP-ETH Driver

The TCPIP-ETH Ethernet driver Version 2.0 or later supports all the commands in this user manual. For details of which particular services the controller boards support, please refer to the corresponding controller board documentation.

Section 4

This section informs you about
– driver functions

4 Driver Functions

The definitions of IBDDIRET, IBDDIFUNC, IBDDIHND, and IBPTR are in the *stdtypes.h* header file. These constants have been introduced to make converting the driver software to the environments of different operating systems easier.

For other programming languages the definitions can be taken from modules or units. These files can be found in the /COMMON/ directory of the example program.

☞ The driver libraries are compatible with Microsoft C. You can also use the driver libraries with Borland C by converting them to a Borland-compatible format. This can be done, for example, using the Borland IMPLIB and IMPDEV tools.

4.1 Opening and Closing Communication Channels

☞ Please note that not every function is supported by every controller board. For additional information, please refer to 3.3 "Driver-Specific Information".

4.1.1 DDI_DevOpenNode

Task:

The *DDI_DevOpenNode* function opens a data channel to the controller board specified by the device name or to a node.

The function receives the device name, the desired access rights, and a pointer to a variable for the node handle as arguments. If the function was executed successfully, a handle is entered in the variable referenced by the pointer, and this handle is used for all subsequent access to this data channel. In the event of an error, a valid value is not entered in the variable.

An appropriate error code is instead returned by the *DDI_DevOpenNode* function, which can be used to determine the cause of the error.

The node handle, which is returned to the application program is automatically generated by the DDI or controller board. This node handle has direct reference to an internal control structure, which contains all the corresponding data for addressing the relevant controller board.

The local node handle is used to obtain all the necessary parameters for addressing the controller board, such as the IP address, socket handle, node handle on the controller board, etc. from this control structure when it is subsequently accessed.

A control structure is occupied when the data channel is opened and is not released until the *DDI_DevCloseNode* function has been executed or the connection has been aborted. The maximum number of control structures is determined when the library is compiled and cannot subsequently be modified. In Windows NT/2000 there are 8 control structures per device, with a maximum of 256.

If all the control structures are occupied, another data channel cannot be opened. In this case, if *DDI_DevOpenNode* is called, it is rejected locally with the appropriate error message.

Call: DDI_DevOpenNode (devName, perm, nodeHd);

Parameters:	devName	CHAR IBPTR* The device name is the name of the device to be addressed. It specifies the controller board and the MPM device (see Section 2.4).
	perm	INT16 The access permission specifies with what access rights the data channel may be accessed. A distinction is made between read, write, and read/write access. Constants for access rights: DDI_READ: Read only access DDI_WRITE: Write only access DDI_RW: Read and write access
	nodeHd	IBDDIHND* The node handle is a pointer to a variable in which the node handle is entered. The node handle is a value assigned by the DDI, which is used to find an assignment to the open node in all other functions.

Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
Negative acknowledgment:	DDI error code	Specifies details of errors found when opening a data channel to a node (see Section 6.3 "DDI Error Messages").
	Cause	- Unknown device name - Node not available

Declaration:

```
IBDDIRET IBDDIFUNC DDI_DevOpenNode(
     CHAR IBPTR *devName,      // IN:  device name
     INT16 perm,               // IN:  access permission
     IBDDIHND IBPTR *nodeHd); // OUT: address of node
handle
```

Example:

```
// Declarations
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
...
void main(void)
{
// open DTI (data interface) nodes
ret = DDI_DevOpenNode("IBB1N1_D", DDI_RW, D_Handle);
  if (ret != ERR_OK)
  {
    // open MXI (mailbox interface) nodes
    ret = DDI_DevOpenNode("IBB1N1_M", DDI_RW,
M_Handle);
  }
...
}
```

4.1.2 DDI_DevCloseNode

Task:

This function closes a data channel or message channel to a node that was previously opened using *DDI_DevOpenNode()*. After this function has been successfully called, the device is no longer "connected" to the called program and the node handle is no longer valid.

Call:

DDI_DevCloseNode (nodeHd);

Parameters:	nodeHd	IBDDIHND* The node handle specifies the node to be closed.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of errors that occurred when calling the function (see Section 6.3 "DDI Error Messages").
	Cause	- Invalid node handle

Example:

```
// Declarations
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
IBDDIRET ret;
...
void main(void)
{
...
  // Close DTI (data interface) nodes
  ret = DDI_DevCloseNode(D_Handle);
  if (ret != ERR_OK)
  {
    // Close MXI (mailbox interface) nodes
    ret = DDI_DevCloseNode(M_Handle);
  }
}
```

4.2 Reading and Writing I/O Data

4.2.1 DDI_DTI_ReadData

Task:

This function reads data from the MPM via the data interface. It places this data in Motorola format in the specified buffer.

Bit objects (4-bit, 2-bit) can be accessed using bit access. The maximum length for bit access is limited to 16 bits.

☞ Before the data is processed further, macros should be used to convert the input data. These macros convert the input data from Motorola to Intel format (see Section 5).

Call:

DDI_DTI_ReadData (node_Hd, ddi_dti_acc);

Parameters:

nodeHd	IBDDIHND* The node handle specifies the IBDDIHND node type.
ddi_dti_acc	T_DDI_DTI_ACCESS* Pointer to a T_DDI_DTI_ACCESS data structure for reading process data.

T_DDI_DTI_ACCESS

Structure elements:

length USIGN16	The *length* structure element contains the number of bytes of data to be read. The maximum number is 1024 bytes. Bit access: The *length* element indicates the number of bits that are to be copied.
address USIGN16	The *address* structure element specifies the DTI address of a process data word in the MPM in bytes. Bit access: The *address* element indicates the byte address from which data is to be copied.
dataCons USIGN16	The *data consistency* structure element specifies the data consistency to be used for access. The bit addressing, reading back of outputs, and access to the XDTA are also activated via additional bits.

Table 4-1 Constants of the data consistency structure element

Constants	Description
DTI_DATA_BYTE DTI_DATA_WORD DTI_DATA_LWORD DTI_DATA_64BIT	Data consistency 8 bits Data consistency 16 bits Data consistency 32 bits Data consistency 64 bits
IB_EX_DTA	XDTA access (see Section 2.6.3)
IB_TO_REMOTE_DTA	Read back outputs (see Section 2.6.1)
DDI_DATA_BIT	Bit access (see Section 2.6.2). The bit position of the specified byte is determined in the dataCons element.
DTI_DATA_BIT_ADDR0 DTI_DATA_BIT_ADDR1 DTI_DATA_BIT_ADDR2 DTI_DATA_BIT_ADDR3 DTI_DATA_BIT_ADDR4 DTI_DATA_BIT_ADDR5 DTI_DATA_BIT_ADDR6 DTI_DATA_BIT_ADDR7	Bit position 0 Bit position 1 Bit position 2 Bit position 3 Bit position 4 Bit position 5 Bit position 6 Bit position 7

	data USIGN8 IBPTR*	This structure element is a pointer to the buffer in which the read data is to be stored.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of errors that occurred when reading the process data (see Section 6.3 "DDI Error Messages").
	Cause	- Invalid node handle - Invalid parameters - Limits of data area were exceeded

Declaration:

```
IBDDIRET IBDDIFUNC DDI_DTI_ReadData(
     IBDDIHND node_Hd,            //IN: node handle
     T_DDI_DTI_ACCESS IBPTR *ddi_dti_acc);
                                      //IN: dti access
                                      // structure
```

Example:

```
// Declarations
// Data structure for reading process data
T_DDI_DTI_ACCESS p_daten;
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
// Process data array
static USIGN16 Data[256];
IBDDIRET ret;
USIGN8 B_InData[DTI_BYTESIZE];
USIGN16 i;
...
void main(void)
{
...
  // main loop
  do
  {
  ...
// start reading from address
p_daten.address   = FromByteAddr;
// read number of input bytes
p_daten.length    = (USIGN16)(2*nWords);
// data consistency: WORD
p_daten.dataCons = DTI_DATA_WORD;
// address IN data buffer
p_daten.data      = B_InData;
    // read process data inputs
   if ((ret = DDI_DTI_ReadData(D_Handle, &p_daten)) ==
ERR_OK)
    {
      for (i=0; i<nWords ; i++)
      {
        Data[i] = (USIGN16)IB_PD_GetDataN(B_InData, i);
      }
    }
...
```

```
  } while (...);     // end of cyclic program
}
```

4.2.2 DDI_DTI_WriteData

Task:

This function writes data to the MPM via the data interface. This function requires data in Motorola format.

Bit objects (4-bit, 2-bit) can be accessed using bit access. The maximum length for bit access is limited to 16 bits.

Macros should be used to convert the output data before writing data to the MPM. These macros convert the output data from Intel to Motorola format (see Section 5).

So that the outputs are reset in the event of an error on the network line (e.g., faulty cable) or at the client (system crash or error in the TCP/IP protocol stack), one of the monitoring mechanisms, connection monitoring or data interface (DTI) monitoring, must be activated. If no monitoring mechanisms are activated, the last process data item remains unchanged in the event of an error.

Call:

IBDDIRET IBDDIFUNC DDI_DTI_WriteData
(nodeHd, ddi_dti_acc);

Parameters:

nodeHd	IBDDIHND* The node handle specifies the USIGN16 node type.
ddi_dti_acc	T_DDI_DTI_ACCESS IBPTR * Pointer to a T_DDI_DTI_ACCESS data structure for writing process data.

T_DDI_DTI_ACCESS

Structure elements:

length USIGN16	The *length* structure element contains the number of bytes of data to be written. The maximum number is 1024 bytes (1 kbyte). Bit access: The *length* element indicates the number of bits that are to be copied.
address USIGN16	The *address* structure element specifies the DTI address of a process data word in the MPM in bytes. Bit access: The *address* element indicates the byte address from which data is to be copied.
dataCons USIGN16	The *data consistency* structure element specifies the data consistency to be used for access. The bit addressing, reading back of outputs, and access to the XDTA are also activated via additional bits.

Table 4-2 Constants of the data consistency structure element

Constant	Description
DTI_DATA_BYTE DTI_DATA_WORD DTI_DATA_LWORD DTI_DATA_64BIT	Data consistency 8 bits Data consistency 16 bits Data consistency 32 bits Data consistency 64 bits
IB_EX_DTA	XDTA access (see Section 2.6.3)
DDI_DATA_BIT	Bit access (see Section 2.6.2). The bit position of the specified byte is determined in the dataCons element.
DDI_DATA_BIT_ADDR0 DDI_DATA_BIT_ADDR1 DDI_DATA_BIT_ADDR2 DDI_DATA_BIT_ADDR3 DDI_DATA_BIT_ADDR4 DDI_DATA_BIT_ADDR5 DDI_DATA_BIT_ADDR6 DDI_DATA_BIT_ADDR7	Bit position 0 Bit position 1 Bit position 2 Bit position 3 Bit position 4 Bit position 5 Bit position 6 Bit position 7

	data USIGN8 IBPTR*	This structure element is a pointer to the buffer from which the data to be written is taken.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of errors that occurred when writing the process data (see Section 6.3 "DDI Error Messages").
	Cause	- Invalid node handle - Invalid parameters - Limits of data area were exceeded

Declaration:

```
IBDDIRET IBDDIFUNC DDI_DTI_WriteData(
     IBDDIHND nodeHd,             // IN: node handle
     T_DDI_DTI_ACCESS IBPTR *ddi_dti_acc);
                                  // IN: dti access
                                  //     structure
```

Example:

```
// Declarations
// Data structure for reading process data
T_DDI_DTI_ACCESS p_daten;
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
// Process data array
static USIGN16 Data[256];
IBDDIRET ret;
USIGN8 B_OutData[DTI_BYTESIZE];
USIGN16 i;
...
void main(void)
{
...
  // main loop
  do
  {
  ...
     // start writing at address
     p_daten.address  = AtByteAddr;
     // write number of output bytes
     p_daten.length   = (USIGN16)(2*nWords);
```

```
        // data consistency: WORD
        p_daten.dataCons = DTI_DATA_WORD;
        // address OUT data buffer
        p_daten.data      = B_OutData;
        for (i=0; i<nWords ; i++)
        {
         IB_PD_SetDataN(B_OutData, i, Data[i]);
        }
        // write process data outputs
        ret = DDI_DTI_WriteData(D_Handle, &p_daten);
...
  } while (...);    // end of cyclic program
}
```

4.2.3 DDI_DTI_ReadWriteData

Task:

The *DDI_DTI_ReadWriteData* function is used to read and write process data in one call. This function increases performance considerably, especially when using process data services via the network, because process data is read and written in a single sequence.

⚠ So that the outputs are reset in the event of an error on the network line (e.g., faulty cable) or at the client (system crash or error in the TCP/IP protocol stack), one of the monitoring mechanisms, connection monitoring or data interface (DTI) monitoring, must be activated. If no monitoring mechanisms are activated, the last process data item remains unchanged in the event of an error.

The function is assigned the node handle and two pointers to *T_DDI_DTI_ACCESS* data structures. One structure contains the parameters for read access and the other structure contains the parameters for write access. The *T_DDI_DTI_ACCESS* structure corresponds to the general DDI specification. A plausibility check is not carried out on the user side, which means that the parameters are transmitted via the network just as they were transferred to the function.

The *nodeHd* parameter specifies the controller board in the network to which the request is to be sent. The node handle must be assigned to a process data channel, otherwise an appropriate error message is generated by the controller board.

Syntax: DDI_DTI_ReadWriteData (nodeHd, writeDTIAcc, readDTIAcc);

Parameters:	nodeHd	IBDDIHND Node handle (DTI) for the connection to which data is to be written. The node handle also determines the controller board that is to be accessed.
	writeDTIAcc	T_DDI_DTI_ACCESS IBPTR* Pointer to a T_DDI_DTI_ACCESS data structure with the parameters for write access.
	readDTIAcc	T_DDI_DTI_ACCESS IBPTR* Pointer to a T_DDI_DTI_ACCESS data structure with the parameters for read access.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

4.3 Writing Commands and Reading Messages

4.3.1 DDI_MXI_SndMessage

Task: The *DDI_MXI_SndMessage* function is used to send a message (INTERBUS command in mailbox syntax) to the controller board. The function receives a node handle and a pointer to a *T_DDI_MXI_ACCESS* data structure as parameters. The *T_DDI_MXI_ACCESS* structure contains all the parameters that are needed to send the message.

These parameters are transmitted to the controller board via the network without a plausibility check, which means that invalid parameters are first detected at the controller board and acknowledged with an error message. The *IBDDIHND nodeHd* parameter specifies the controller board in the network to which the request is to be sent.

The node handle must be assigned to a mailbox interface data channel, otherwise an appropriate error message is generated by the controller board.

Call:	DDI_MXI_SndMessage (nodeHd, ddi_mxi_acc);	
Parameters:	nodeHd	IBDDIHND* The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
	ddi_mxi_acc	T_DDI_MXI_ACCESS IBPTR* Pointer to a *T_DDI_MXI_ACCESS* data structure for sending a command.
T_DDI_MXI_ACCESS:	Structure elements:	
	msgType	The zero parameter is always assigned for an order without confirmation (unconfirmed request or unconfirmed indication).
	msgLength	The *message length* structure element contains the total length of the message to be sent in bytes. The maximum permissible length (see below) is 1024.
	DDIUserID	Reserved
	*msgBlk	The **msgBlk* structure element is a pointer to a message block, which includes the message to be sent in mailbox syntax.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies why the function could not be executed (see "DDI Error Messages" on page 6-8).

Cause	Invalid node handle No suitable mailbox found The message exceeds the maximum mailbox length that can be used (1020 bytes = 1024 bytes minus 2 bytes for command code minus 2 bytes for parameter count).

Declaration:

```
IBDDIRET IBDDIFUNC DDI_MXI_SndMessage(
     IBDDIHND nodeHd,    // IN: node handle
     T_DDI_MXI_ACCESS IBPTR *ddi_mxi_acc);
                         // IN: pointer to
                              mailbox access
                         // structure
```

Format of the structure *T_DDI_MXI_ACCESS*:

```
typedef struct {
     USIGN16 msgType          // Message type
     USIGN16 msgLength;       // Message length
     USIGN16 DDIUserID;       // DDI_User_ID
     USIGN8 IBPTR *msgBlk;    // Pointer to array for
                                   the message
} T_DDI_MXI_ACCESS;
```

Example:

```
// Declarations
// Data structure for sending commands
T_DDI_MXI_ACCESS req_res;
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
// Firmware command
static USIGN16 Msg[] = {0x0710, //
CREATE_CONFIGRURATION_REQ
                        0x0001, // Parameter counter
                        0x0001};// Frame reference
// Transmit buffer
USIGN8 snd_buffer[1024];
IBDDIRET ret;
USIGN16 i;
...
void main(void)
{
...
  IB_SetCmdCode(snd_buffer, Msg[0]);
  IB_SetParaCnt(snd_buffer, Msg[1]);
```

PHOENIX CONTACT

```
    for (i=1 ; i<=Msg[1] ; i++)
    {
      IB_SetParaN(snd_buffer, i, Msg[i+1]);
    }
    req_res.msgType   = 0;
    req_res.DDIUserID = 0;
    req_res.msgLength = (USIGN16)((Msg[1] + 2 ) * 2);
    req_res.msgBlk    = snd_buffer;
    // Transmit firmware command
    ret = DDI_MXI_SndMessage(M_Handle, &req_res);
  ...
  }
```

4.3.2 DDI_MXI_RcvMessage

Task:

This function fetches a message from a mailbox. For example, it is used to fetch a confirmation following a command. The confirmation is not expected. If no confirmation is present, a corresponding message appears in the *DDI error code* parameter.

☞ The length of the available receive buffer must be entered in the msgLength component of the T_DDI_MXI_ACCESS structure. Before reading, the driver checks the size of the receive buffer and generates the error message ERR_MSG_TOO_LONG (009A$_{hex}$) if the message received is larger than the available memory space.

Call:	IBDDIRET IBDDIFUNC DDI_MXI_RcvMessage (nodeHd,ddi_ mxi_acc);	
Parameters:	nodeHd	IBDDIHND The node handle is the logical number of a previously opened channel on the DDI interface.
	ddi_mxi_acc	T_DDI_MXI_ACCESS IBPTR* Pointer to a T_DDI_MXI_ACCESS data structure for receiving a message.
T_DDI_MXI_ACCESS	Structure elements:	
	msgType	The zero parameter is always assigned for an order without confirmation (unconfirmed request or unconfirmed indication).
	msgLength	The size of the available receive buffer should be specified in bytes before calling the *DDI_MXI_RcvMessage* function using the *message length* structure element. After the successful receipt of a message the *message length* structure element includes the actual length of the message in bytes.
	DDIUserID	The value of the DDI user ID is not relevant.
	*msgBlk	The **msgBlk* structure element is a pointer to a message block that contains the message received in mailbox syntax.
Positive acknowledgment:	ERR_OK (0000$_{hex}$) Meaning	 The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies why the function could not be executed (see "DDI Error Messages" on page 6-8).
	Cause	- Invalid node handle - Receive buffer too small - No message present

Declaration:

```
IBDDIRET IBDDIFUNC DDI_MXI_RcvMessage(
     IBDDIHND nodeHd,    // IN: node handle
     T_DDI_MXI_ACCESS IBPTR *ddi_ mxi_acc);
                         // OUT: pointer to
                              mailbox access
                         // structure
```

Format of the *T_DDI_MXI_ACCESS* structure:

```
typedef struct {
     USIGN16 msgType;          // Message type
     USIGN16 msgLength;        // Message length
     USIGN16 DDIUserID;        // DDI_User_ID
     USIGN8 IBPTR *msgBlk;     // Pointer to array for
                               // the message
} T_DDI_MXI_ACCESS;
```

Example:

```
// Declarations
// Data structure for receiving messages
T_DDI_MXI_ACCESS con_ind;
// Node handle variables
IBDDIHND D_Handle;
IBDDIHND M_Handle;
IBDDIRET ret;
USIGN16 i;
USIGN8 rcv_buffer[1024];
...
void main(void)
{
...
  Msg[0] = 0;    // set message code to "0"
  Msg[1] = 0;    // set parameter count to "0"
  rcv_buffer[2] = 0;    // set parameter count to "0"
  rcv_buffer[3] = 0;    // set parameter count to "0"
  con_ind.msgType    = 0;
  con_ind.DDIUserID = 0;
  con_ind.msgLength = 1024;
  con_ind.msgBlk     = rcv_buffer;
  // Fetch message from the controller board
  ret = DDI_MXI_RcvMessage(M_Handle, &con_ind);
  if ((ret != ERR_NO_MSG) && (ret == ERR_OK))
  {
    Msg[0] = (USIGN16)IB_GetMsgCode(rcv_buffer);
```

```
      Msg[1] = (USIGN16)IB_GetParaCnt(rcv_buffer);
      for (i=1 ; i<=Msg[1] ; i++)
      {
        Msg[i+1] = (USIGN16)IB_GetParaN(rcv_buffer, i);
      }
    } // NO_MESSAGE
...
}
```

4.4 Diagnostic Functions

4.4.1 GetIBSDiagnostic

Task:

The GetIBSDiagnostic() function is used to evaluate the operating state of the INTERBUS controller board and thus the operating state of the INTERBUS system.
After the function has been called successfully, the structure components contain the contents of the diagnostic status register and the diagnostic parameter register in processed form.

Call:

GetIBSDiagnostic (NodeHd, diagInfo);

Parameters:

NodeHd	IBDDIHND* The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
diagInfo	T_IBS_DIAG IBPTR* Pointer to a T_IBS_DIAG structure with error details.

T_IBS_DIAG

Structure elements:

state USIGN16	The bits of the *state* structure element describe the state of the bus. The state of the INTERBUS system can be evaluated by masking the *state* structure element.
diagPara USIGN16	The contents of the diagPara structure element depend on the contents of the state structure element (see Section 2.6.5):

Format of the T_IBS_DIAG structure

```
typedef struct {
     USIGN16 state;// Status of INTERBUS
     USIGN16 diagPara;
                      // Type of error (controller,
                      user, etc.)
} T_IBS_DIAG;
```

Positive acknowledgment:

ERR_OK (0000$_{hex}$)

Meaning	The function has been executed successfully.

Negative acknowledgment:

DDI error code	Specifies details of an error that has occurred. Cause: Invalid node handle

The diagnostic information should only be evaluated if the function has been executed successfully (positive acknowledgment *ERR_OK* [0000$_{hex}$]). If a negative acknowledgment is confirmed, **no** valid diagnostic information is available.

Declaration:

```
IBDDIRET IBDDIFUNC GetIBSDiagnostic(
     IBDDIHND nodeHd,    // IN: node handle
     T_IBS_DIAG IBPTR *diagInfo);
                         // Pointer to the structure
                         // with error details
```

Example:

Program section to evaluate the *state* parameter using masking (AND operation) with specified constants:

Example:

```
IBDDIRET ret;
// diagnostics structure
T_IBS_DIAG diagnostics;
...
void main(void)
{
...
  // main loop
  do
  {
  ...
    ret = GetIBSDiagnostic(NodeHdMXI, &diagnostics);
    if (ret == ERR_OK)
    {
      if (diagnostics.state & READY_BIT)
      {
        printf("IBS Ready")
      }
      if (diagnostics.state & RUN_BIT)
      {
        printf("IBS Run")
      }
      else
      {
        printf("IBS Stop!")
      }
    }
...
  } while (...);    // end of cyclic program
}
```

4.4.2 GetIBSDiagnosticEx

Task: This function is used to read the operating state of the INTERBUS master firmware and the operating state of INTERBUS. GetIBSDiagnostic has been extended to include the extended diagnostic register.

Call GetIBSDiagnosticEx (NodeHd, diagInfo);

Parameters:	NodeHd	IBDDIHND* The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
	diagInfo	T_IBS_DIAG_EX IBPTR* Pointer to a T_IBS_DIAG_EX data structure with diagnostic data.
T_IBS_DIAG_EX	Structure elements:	
	state	The bits of the "state" structure element correspond to the diagnostic bit register.
	diagPara	The "diagPara" structure element corresponds to the diagnostic parameter register.
	addInfo	The "addInfo" structure element corresponds to the Add_Error_Info parameter of the negative messages of firmware commands.
Positive acknowledgment:	ERR_OK (0000hex)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC GetIBSDiagnosticEx(
     IBDDIHND nodeHd,    // IN: node handle
     T_IBS_DIAG_EX IBPTR *diagInfo);
                         // Pointer to the structure
                         // with error details
```

Format of the *T_IBS_DIAG_EX* structure:

```
typedef struct {
     USIGN16 state;      // State of the bus:
                         // Ready, Run, etc.
     USIGN16 diagPara;   // Additional information,
                         // see parameter description
                         // on the previous page
     USIGN16 diagPara;   // Additional information,
                         // see parameter description
                         // on the previous page
} T_IBS_DIAG_EX;
```

4.4.3 GetSlaveDiagnostic

Task:	This function is used to read the operating state of the INTERBUS slave.	
	GetSlaveDiagnostic (NodeHd, diagInfo);	
Parameters:	NodeHd	IBDDIHND* The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
	diagInfo	T_IBS_DIAG IBPTR* Pointer to a T_IBS_DIAG data structure with diagnostic data.
Typ T_IBS_DIAG	Structure elements:	
	state	The bits of the "state" structure element correspond to the slave diagnostic status register (see Section 2.6.5).
	diagPara	Reserved
Positive acknowledgment:	ERR_OK (0000hex)	
	Meaning	The function has been executed successfully.
Negative acknowledgment	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC GetSlaveDiagnostic(
     IBDDIHND nodeHd,    // IN: node handle
     T_IBS_DIAG IBPTR *diagInfo);
                         // Pointer to the structure
                         // with error details
```

Format of the structure *T_IBS_DIAG*:

```
typedef struct {
     USIGN16 state;      // State of the bus:
                         // Ready, Run, etc.
     USIGN16 diagPara;   // Reserved
} T_IBS_DIAG;
```

4.5 Watchdog Functions

Please note that not every function is supported by every controller board. For additional information, please refer to 3.3 "Driver-Specific Information".

4.5.1 EnableWatchDog()

Task:	This function activates the watchdog.	
Call:	EnableWatchDog (USIGN16 IBDDIHND NodeHd)	
Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

A SysFail signal is only created locally for the ISA-DPM driver when a watchdog is triggered, but no I/O fail signal is sent to the higher-level master.

After the function has been called, the watchdog must be triggered at regular intervals.

Declaration:

```
IBDDIRET IBDDIFUNC EnableWatchDog(
     IBDDIHND nodeHd,   // IN: node handle
```

4.5.2 TriggerWatchDog()

Task: This function triggers the watchdog.

Call: TriggerWatchDog (USIGN16 IBDDIHND NodeHd)

Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

☞ This call must be repeated at regular intervals so that the watchdog does not trigger a reset.

Declaration:

```
IBDDIRET IBDDIFUNC TriggerWatchDog(
     IBDDIHND nodeHd,    // IN: node handle
```

4.5.3 GetWatchDogState()

Task: This function can be used to determine from your application program whether the corresponding host watchdog has triggered a reset. At the same time, the watchdog is triggered.

Call: IBDDIRET IBDDIFUNC GetWatchDogState (USIGN16 IBDDIHND NodeHd)

Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.

Positive acknowledgment: Return value:

Host watchdog status:
1 The host watchdog has triggered a reset.
0 The host watchdog has not triggered a reset.

Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC GetWatchDogState(
     IBDDIHND nodeHd,    // IN: node handle
```

4.5.4 ClearWatchDog()

Task:

The function resets the watchdog status and simultaneously deactivates the watchdog.

This function can only be used if the watchdog has been triggered.

Call:

IBDDIRET IBDDIFUNC ClearWatchDog (USIGN16 IBDDIHND NodeHd)

Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning:	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC ClearWatchDog(
     IBDDIHND nodeHd,    // IN: node handle
```

4.5.5 SetWatchDogTimeout()

Task:

The function sets the watchdog timeout value and activates the watchdog.

This function can only be used when the watchdog is deactivated.

The watchdog must first be triggered to set the watchdog timeout. Next the ClearWatchDog function must be used to reset the watchdog status and deactivate the watchdog and the SetWatchDogTimeout function can then be used to set the watchdog timeout.

Call:

IBDDIRET IBDDIFUNC SetWatchDogTimeout (USIGN16 IBDDIHND NodeHd, USIGN16 IBPTR *dataPtr)

Parameters:

NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
*dataPtr	Pointer to the variable with the new timeout value.

Table 4-3 Variable values for different monitoring times

Monitoring Time	Variable Value
8.2 ms	00_{hex}
16.4 ms	04_{hex}
32.8 ms	08_{hex}
65.5 ms	$0C_{hex}$
131.1 ms	10_{hex}
262.1 ms	14_{hex}
524.3 ms	18_{hex}
1048.6 ms	$1C_{hex}$

With the PCI-DPM driver, only the preset monitoring times from the PCI slave configurator can be accepted. Times cannot be set via the SetWatchdogTimeout function.

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning: The function has been executed successfully.

Negative acknowledgment: DDI error code — Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC SetWatchDogTimeout(
     IBDDIHND nodeHd,    // IN: node handle
     IBPTR *dataPtr);    // Pointer to a variable
```

4.5.6 GetWatchDogTimeout()

Task: This function reads the current timeout value.

Call: IBDDIRET IBDDIFUNC GetWatchDogTimeout (USIGN16 IBDDIHND NodeHd, USIGN16 IBPTR *dataPtr)

Parameters:

NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
*dataPtr	Pointer to the variable with the current timeout value.

Positive acknowledgment: ERR_OK (0000$_{hex}$)

Meaning:	The function has been executed successfully.

Negative acknowledgment:

DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC GetWatchDogTimeout(
     IBDDIHND nodeHd,    // IN: node handle
     IBPTR *dataPtr);    // Pointer to a variable
```

4.5.7 EnableWatchDogEx()

Task: This function sets the watchdog timeout value and activates the watchdog. After activation, the watchdog must be reset at regular intervals (TriggerWatchDog).

Declaration: IBDDIRET IBDDIFUNC EnableWatchDogEx (IBDDIHND NodeHd, USIGN16 IBPTR *dataPtr);

Parameters:

NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
dataPtr	Pointer to a variable, which contains the new timeout value.

Possible values can be found in the SetWatchDogTimeout section.

Positive acknowledgment:	ERR_OK (0000hex)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC EnableWatchDogEx(
     IBDDIHND nodeHd,    // IN: node handle
     IBPTR *dataPtr);    // Pointer to a variable
```

4.6 Driver Settings and Management

☞ Please note that not every function is supported by every controller board. For additional information, please refer to 3.3 "Driver-Specific Information".

4.6.1 DDIGetInfo()

Task: This function can be used to read a version string from the driver and Device Driver Interface (IBDDIWNT.DLL).

Call: IBDDIRET IBDDIFUNC DDI_GetInfo
(IBDDIHND NodeHd; USIGN16 cmd, VOID IBPTR *infoPtr)

Parameters:		
	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI.
	cmd	Selects the source of the version information:

Table 4-4 Constants of the cmd structure element

	Constant	**Description**
Version ID	DDI_INFO_DDI_VERSION	DDI version info
	DDI_INFO_DRV_VERSION	Driver version info

	infoPtr	Pointer to a T_DDI_VERSION_INFO data structure (see below).
T_DDI_VERSION_INFO	Structure elements:	
	vendor	CHAR vendor[32]; Vendor name "(c) Phoenix Contact Germany"
	name	CHAR name[48]; Name of the driver/interface "Windows NT 4.0 Device Driver" "Windows NT 4.0 Device Driver Interface"
	revision	CHAR revision[8]; Revision information as text "1.10"

	dateTime	CHAR dateTime[32]; Date and time stamp of creation "Thu Jul 31 15:44:44 1997" created by compiler
	revNumber	INT16 revNumber; Revision as integer value 110dec
Positive acknowledgment:	ERR_OK (0000hex)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: - Invalid node handle - Invalid parameters

4.6.2 ReadResetCounter()

Task: This function reads the reset count from the driver.

Call: IBDDIRET IBDDIFUNC ReadResetCounter
(IBDDIHND NodeHd, INT32 IBPTR *resetCount)

Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
	resetCount	Pointer to the INT32 variable to read the reset count.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

4.7 Controller Board Monitoring

4.7.1 GetSysFailRegister()

Task:

The *GetSysFailRegister* function writes the contents of the SysFail register using the variable referenced by *sysFailRegPtr*. Bits 4, 8, and 12 of the register indicate whether the SysFail signal of the corresponding board (PC and INTERBUS master) is activated or not. If a malfunction occurs in an MPM device (e.g., watchdog triggered), the relevant bit is activated in the SysFail register, i.e., set to one. This bit then remains set until the malfunction is corrected. The individual bits of the register are assigned to the MPM devices as follows:

Bit 4:	Coprocessor board (COP)
Bit 8:	INTERBUS slave controller board
Bit 12:	Host (PC)

Call:

IBDDIRET IBDDIFUNC GetSysFailRegister
(USIGN16 IBDDIHND NodeHd,USIGN16 IBPTR *sysFailRegPtr)

Parameters:

NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
*sysFailRegPtr	Pointer to a variable, in which the contents of the SysFail register are entered.

Positive acknowledgment:

ERR_OK (0000$_{hex}$)

Meaning	The function has been executed successfully.

Negative acknowledgment:

ERR_INVALID_BOARD_NUM (0080$_{hex}$)

Meaning	An invalid board number has been specified.

ERR_TSR_NOT_LOADED (008B$_{hex}$)

Meaning	The specified controller board is not available or the driver required is not loaded.

4.7.2 ClearSysFailSignal()

Task:	This function deletes the SysFail signal from the coprocessor board.	
Call:	IBDDIRET IBDDIFUNC ClearSysFailSignal (USIGN16 IBDDIHND NodeHd)	
Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC ClearSysFailSignal(
     IBDDIHND nodeHd,    // IN: node handle
```

4.7.3 SetSysFailSignal()

Task:	This function sets the SysFail signal on the coprocessor board.	
Call:	IBDDIRET IBDDIFUNC SetSysFailSignal(USIGN16 IBDDIHND NodeHd)	
Parameters:	NodeHd	The node handle is the logical number (handle) of a previously opened channel on the DDI interface.
Positive acknowledgment:	ERR_OK (0000$_{hex}$)	
	Meaning	The function has been executed successfully.
Negative acknowledgment:	DDI error code	Specifies details of an error that occurred when writing. Cause: Invalid node handle

Declaration:

```
IBDDIRET IBDDIFUNC SetSysFailSignal(

IBDDIHND nodeHd,// IN: node handle
```

4.8 Ethernet Communication Monitoring

So that the outputs are reset in the event of an error on the network line (e.g., faulty cable) or at the client (system crash or error in the TCP/IP protocol stack), one of the monitoring mechanisms, connection monitoring or data interface (DTI) monitoring, must be activated. If no monitoring mechanisms are activated, the last process data item remains unchanged in the event of an error.

Which monitoring function is used and when depends on the application program and the safety requirements.

Monitoring Mechanisms
Monitoring mechanisms require a correctly operating network. To prevent excessive network loads or to avoid using unreliable network operating modes, operation in separate automation networks or connection to another network via a firewall is recommended.

Connection Monitoring

Application

Connection monitoring can be used to determine whether there is still a connection between the bus coupler (server) and the computer (client) and whether this computer responds to requests. This monitoring function can also be used to detect the following error causes:

– Cable broken, not connected or short circuited.
– Transceiver faulty.
– Errors or faults in the Ethernet adapter of the bus coupler or in the client.
– Client system crash (workstation).
– Error in the TCP/IP protocol stack.

Activating monitoring

The *ETH_SetHostChecking* function activates the mode for monitoring the connection and the status of the client. The function is assigned a valid node handle (DTI or MXI data channel) and a pointer (*time*) to a variable with the timeout time.

This mode can be activated for all clients (workstations) with a DDI connection. A connection to a client, which only uses Ethernet management cannot be monitored. If several connections to a client are activated simultaneously, the client is only addressed once during a cycle. If the connection no longer exists, monitoring is also reset.

Echo port

Monitoring uses the echo port, which is provided on all systems that support TCP/IP. Each data telegram to this port is sent back from the receiver to the transmitter. The port is used for both connection-oriented TCP and connectionless UDP. In the case of the bus coupler, the echo port is used with UDP, to keep the resources used to a minimum.

Detecting an error

Connection monitoring sends a short data telegram to a client every 500 ms. This interval is predefined and does not change according to the number of clients to be addressed. This means that the frequency with which each client is "addressed" decreases with the number of connected clients. After the data telegram has been sent, the Inline bus coupler waits for a user-defined time for the reply to be received. If the reply is not received within this time, the bus coupler sends another data telegram to the relevant client. This process is repeated a maximum of three times. Connection monitoring then assumes that a serious error has occurred and sets the SysFail signal (outputs are set to zero).

Deactivating monitoring

If connection monitoring is no longer required, it can be deactivated using the *ETH_ClearHostChecking* function. Monitoring is only deactivated for the client and the connection that is specified by the node handle. If the same client has additional DDI connections to the bus coupler and connection monitoring was also activated for these connections, this client is still monitored via the other connections.

If a DDI connection is closed using *DDI_DevCloseNode*, monitoring for this client is also deactivated. Additional connections are treated as above; they are not reset and monitoring for these connections is not deactivated.

☞ On a PC with Windows as the operating system, an echo server is running if the TCP/IP service has been installed. You will find these services under ...\Control Panel\Network\Services. The user must ensure that the echo server responds within 500 ms in every operating state. The echo server implemented by default in Windows 2000 does not meet these requirements. For this reason, the user should use DTI monitoring for connection monitoring.

4.8.1 ETH_SetNetFailMode()

Task:

The *ETH_SetNetFailMode* routine is used to change the behavior of the controller board in the event of a NetFail. After startup, the controller board is in standard mode (*ETH_NF_STD_MODE*), which means that in the event of a NetFail, all outputs of the modules connected to the INTERBUS system are set to zero and the bus continues to run. This behavior can be changed by calling the routine. At present, the controller board supports two different modes:

– Standard mode: The controller board behavior remains the same, i.e., the outputs are set to zero in the event of an error.
– Alarm stop mode: Not only are the outputs set to zero but an alarm stop command is also sent to the controller board.

If the function is executed successfully, the routine returns the return value 0 (ERR_OK). In the event of an error, the return value is an error code (see DDI_ERR.H).

In alarm stop mode, a command is sent to the controller board but the return value is not obtained. This means that an application program will receive this message on its next read attempt.

Syntax:

```
IBDDIRET IBDDIFUNC ETH_SetNetFailMode(IBDDIHND nodeHd,
                                      T_ETH_NET_FAIL_MODE *netFailModeInfo);
```

The routine receives a valid node handle and a pointer to the structure described below as parameters. In addition to a component in which the mode to be set is entered, the structure contains a pointer to an optional parameter block, the size of which is also entered in the structure. This parameter block is purely optional and is not used for the modes that exist at present. Thus, the *numOfBytes* structure component should be set to zero.

Parameters:

IBDDIHND nodeHd	Node handle of a controller board for which the NetFail mode is to be changed.
T_ETH_NET_FAIL_MODE *netFailModeInfo	Pointer to a *T_ETH_NET_FAIL_MODE* data structure. This structure contains the parameters for setting the *NetFail mode* and, if necessary, optional parameters.

Format of the T_ETH_NET_FAIL_ MODE data structure

```
typedef struct {
     USIGN16 mode;         /* NetFail mode */
     USIGN16 numOfBytes; /* Size of the parameter
                            block in bytes      */
     VOID *miscParamPtr; /* Parameters for the
                       relevant NetFail mode */

} T_ETH_NET_FAIL_MODE;
```

The function prototypes, the type definition of the data structure, and the symbolic constants can be found in the IOCTRL.H file.

4.8.2 ETH_GetNetFailMode()

Task:

The *ETH_GetNetFailMode* function can be used to read the set *NetFail mode*. The routine expects a valid node handle and a pointer to a *T_ETH_NET_FAIL_MODE* data structure (see above) as parameters. After the routine has been called successfully, the user can read the set NetFail mode from the structure. If there are no additional parameters for the set mode, this is indicated by the *numOfBytes* structure component, which contains the value zero in this case.

Syntax:

IBDDIRET IBDDIFUNC ETH_GetNetFailMode(IBDDIHND nodeHd, T_ETH_NET_FAIL_MODE *netFailModeInfo)

Parameters:

IBDDIHND nodeHd	Node handle of a controller board from which information about the set NetFail mode is to be read.
T_ETH_NET_FAIL_MODE *netFailModeInfo	Pointer to a T_ETH_NET_FAIL_MODE data structure. If the function is called successfully, the parameters of the NetFail mode set on the controller board as well as the mode itself are entered in this structure.

Format of the structure

```
typedef struct {
    USIGN16 mode;         /* NetFail mode */
    USIGN16 numOfBytes; /* Size of the parameter
                        block in bytes      */
    VOID *miscParamPtr; /* Parameters for the
                        relevant NetFail mode */
} T_ETH_NET_FAIL_MODE;
```

Constants of the different NetFail modes

```
#define ETH_NF_STD_MODE              0
#define ETH_NF_ALARMSTOP_MODE        1
```

The function prototypes, the type definition of the data structure, and the symbolic constants can be found in the IOCTRL.H file.

4.8.3 ETH_SetHostChecking()

Task:

After the *ETH_SetHostChecking* function has been called successfully, the client (user workstation) is addressed by the bus coupler at regular intervals.

If the client does not respond within the predefined time (timeout time), three additional attempts are made to address the client. If there is still no response, the SysFail signal is set and the TCP connection is aborted by the bus coupler.

Syntax:

```
IBDDIRET IBDDIFUNC ETH_SetHostChecking (IBDDIHND nodeHd,
                                        USIGN16 *time);
```

Parameters:

IBDDIHND nodeHd	Node handle (MXI or DTI) for the bus coupler that is to be monitored.
USIGN16 *time	Pointer to a variable, which contains the desired timeout time when called. If the function has been called successfully, the actual timeout time is then entered in this variable. The smallest value for the timeout time is 330 ms, the largest value for timeout time is 65535 ms. If a smaller value is entered, the error code ERR_INVLD_PARAM is returned and "Host Checking" is not activated.

Return value:

IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Example

Unix / Windows NT/2000

```
IBDDIHND ddiHnd;
{
void CAU00yxDlg::OnButtonSetHostCheckingOn()
{
     IBDDIRET ddiRet;
     USIGN16 hcTime = 1000;
     .
     .
     .
     {
          ddiRet = ETH_SetHostChecking
          (ddiHnd, &hcTime);
          if (ddiRet == ERR_INVLD_PARAM)
          {
               // Selected hcTime is too short
               //(330 ms, minimum)
               .
               .
               .
          }
     }
     UpdateData (FALSE)
}
```

4.8.4 ETH_ClearHostChecking()

Task:

The *ETH_ClearHostChecking* function deactivates the node used to monitor the client. This function only receives the node handle as a parameter, which is also used to activate monitoring with *ETH_SetHostChecking*. After the function has been called successfully, monitoring via this channel and for this client is deactivated. Other activated monitoring channels are not affected.

Syntax:

IBDDIRET IBDDIFUNC ETH_ClearHostChecking (IBDDIHND nodeHd);

Parameters:	IBDDIHND nodeHd	Node handle (MXI or DTI) for the bus coupler for which monitoring is to be deactivated. The same node handle that was used for activating monitoring must also be used here.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Data Interface (DTI) Monitoring

Error detection and response

Client monitoring using connection monitoring can only determine whether a client can still be addressed. It is not possible to determine whether the process that controls the bus coupler (application program) is still operating correctly. An extremely serious error occurs when the controlling process is no longer operating correctly, i.e., the bus coupler is no longer supplied with up-to-date process data and as a result incorrect output data is sent to the local bus devices.

DTI monitoring can detect if a message to the data interface of the bus coupler has failed to arrive and the appropriate safety measures can be implemented. In this case, the failure of the DTI data telegram sets the SysFail signal and resets the output data for the local bus devices to zero.

Activating monitoring

Data interface (DTI) monitoring is not activated immediately after the *ETH_SetDTITimeoutCtrl* function has been called, but only after data is written to or read from the DTI for the first time using the node handle, which was also used when activating monitoring. Writing to or reading from the DTI via a connection or a node handle for which no monitoring is set does not therefore enable monitoring for another connection.

Once access has been enabled for the first time, all subsequent access must be enabled within the set timeout time, otherwise the SysFail signal is activated.

Deactivating monitoring

Monitoring is deactivated by calling the *ETH_ClearDTITimeoutCtrl* function or by closing the relevant DTI node using the *DDI_DevCloseNode* function.

If a connection is interrupted by the bus coupler as a result of DTI monitoring, the monitoring mode for this connection is deactivated and the corresponding DDI node is closed (see also "ETH_SetDTITimeoutCtrl()" on page 4-43).

If the bus coupler detects that a connection has been interrupted without the node having been closed, the SysFail signal is set. This applies especially if the controlling process (application program) is closed with an uncontrolled action (e.g., pressing Ctrl+C) and all the open data channels are closed by the operating system.

Status of the SysFail signal

The user can read the status of the SysFail signal using the *ETH_GetNetFailStatus* function. In addition to the status of the SysFail signal, a second parameter is returned, which indicates the reason if the SysFail signal has been set. An additional function for the controlled setting of the SysFail signal is provided for test purposes. This enables the behavior of the system in the event of a SysFail to be tested, especially during program development. The *ETH_SetNetFail* function only requires a valid node handle as a parameter, so that the corresponding board can be addressed in the network.

The SysFail signal can only be reset by calling the *ETH_ClrSysFailStatus* function or by executing a reset on the bus coupler.

4.8.5 ETH_SetDTITimeoutCtrl()

Task:	The *ETH_SetDTITimeoutCtrl* function activates the node for monitoring the DTI data channel specified by the node handle. After this function has been called, the monitoring function checks whether process data is received regularly. The function is assigned a valid node handle for a DTI data channel and a pointer (**time*) to a variable with the desired timeout time. After the function has been called, the timeout time calculated by the bus coupler can be found in the *USIGN16 *time* variable.	
Syntax:	IBDDIRET IBDDIFUNC ETH_SetDTITimeoutCtrl (IBDDIHND nodeHd, USIGN16 *time);	
Parameters:	IBDDIHND nodeHd	Node handle (DTI) for the bus coupler that is to be monitored.
	USIGN16 *time	Pointer to a variable, which contains the desired timeout time when called. If the function has been called successfully, the actual timeout time is then entered in this variable. The timeout time can be set to a value in the range of 110 ms to 65535 ms.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

4.8.6 ETH_ClearDTITimeoutCtrl()

Task:

The *ETH_ClearDTITimeoutCtrl* function deactivates the node for monitoring process data activity. This function only receives the node handle as a parameter, which is also used to activate monitoring. After the function has been called successfully, monitoring via this channel and for this client is deactivated. Other activated monitoring channels are not affected.

Syntax:

IBDDIRET IBDDIFUNC ETH_ClearDTITimeoutCtrl(IBDDIHND nodeHd);

Parameters:

IBDDIHND nodeHd	Node handle (DTI) for the bus coupler for which monitoring is to be deactivated. The same node handle that was used for activating monitoring must also be used here.

Return value:

IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Example

Unix / Windows NT/2000

```
IBDDIHND ddiHnd;
{
     IBDDIRET ddiRet;
     .
     .
     .
     ddiRet = ETH_ClearDTITimeoutCtrl (ddiHnd);
     .
     .
     .
}
```

Handling the SysFail Signal for the Ethernet/Inline Bus Coupler

The SysFail signal is set by writing a register to the coupling memory of the bus coupler. As soon as this signal is detected by the bus coupler, all local bus device outputs are reset and the PCP connections to the devices are interrupted.

Once the SysFail signal has been set to zero, process data can be output again. The SysFail signal is always set if the connection to the client is interrupted, the bus coupler does not write data to the DTI within the specified time or a general malfunction has been detected on the bus coupler, which prevents safe operation.

The setting of the SysFail signal is indicated by setting the SysFail bit in the control word of each data telegram, which is sent by the bus coupler. The SysFail signal can be reset using the appropriate command or, if this is no longer possible, by executing a power up.

4.8.7 ETH_SetNetFail()

Task: The *ETH_SetNetFail* function sets the SysFail signal on the bus coupler and thus prevents the further output of process data to the local bus devices. The function is assigned a node handle for a DTI or mailbox data channel of the relevant bus coupler as a parameter.

Syntax: IBDDIRET IBDDIFUNC ETH_SetNetFail (IBDDIHND nodeHd);

Parameters:	IBDDIHND nodeHd	Node handle (MXI or DTI) for the bus coupler on which the SysFail signal is to be executed.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Example **Unix / Windows NT/2000**

```
IBDDIHND ddiHnd;
{
     IBDDIRET ddiRet;
```

```
      .
      .
      .
      ddiRet = ETH_SetNetFail (ddiHnd);
      .
      .
      .
}
```

4.8.8 ETH_GetNetFailStatus()

Task:

The *ETH_GetNetFailStatus* function sends the SysFail status to the user, which is determined by the node handle of the bus coupler. The function is assigned a node handle for an open DTI or MXI data channel and a pointer to a *T_ETH_NET_FAIL* structure as parameters. After the function has been called successfully, the structure components contain the status (*status*) of the SysFail signal and an error code (*reason*) if the SysFail signal has been set.

If the SysFail signal is not set, the *status* structure component has the value 0. Otherwise *status* has the value 0xFFFF. The *reason* structure component is only valid if the SysFail signal is set. The possible values for *reason* can be found in the IOCTRL.H file or the list on page 4-48.

Syntax:

IBDDIRET IBDDIFUNC ETH_GetNetFailStatus (IBDDIHND nodeHd,
T_ETH_NET_FAIL *netFailInfo);

Parameters:

IBDDIHND nodeHd	Node handle (MXI or DTI) for the bus coupler on which the SysFail status is to be read.
T_ETH_NET_FAIL *netFailInfo	Pointer to a structure, which contains the SysFail status and the reason for the SysFail, if applicable.

Return value:

IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Format of the T_ETH_NET_FAIL structure

```
typedef struct {
     USIGN16 status; /* SysFail status */
     USIGN16 reason; /* Reason for the SysFail */
} T_ETH_NET_FAIL;
```

Possible values for the *status* structure component:

ETH_NET_FAIL_ACTIVE	0xFFFF /* SysFail signal triggered */
ETH_NET_FAIL_INACTIVE	0x0000 /* SysFail signal not triggered */

Example

Unix/Windows NT/2000

```
IBDDIHND ddiHnd;
{
     IBDDIRET ddiRet;

     T_ETH_NET_FAIL netFailInfo
     USIGN16 nfStatus;
     USIGN16 nfReason;
     .
     .
     .
     ddiRet = ETH_GetNetFailStatus (ddiHnd,
     &netFailInfo);

     if (ddiRet == ERR_OK)
     {
          nfStatus = netFailInfo.status
          nfReason = netFailInfo.reason;
     }
     .
     .
     .
}
```

Possible values for the *reason* structure component:

ETH_NF_NO_ERR	/* No error */	0x0000
ETH_NF_TASK_CREAT_ERR	/* Error when starting a task */	0x0001
ETH_NF_LISTENER_ERR	/* Listener task error */	0x0002
ETH_NF_RECEIVER_ERR	/* Receiver task error */	0x0003
ETH_NF_ACCEPT_ERR	/* Accept error */	0x0004
ETH_NF_ECHO_SERVER_ERR	/* Echo server task error */	0x0005
ETH_NF_HOST_CONTROL_ERR	/* Workstation controller task error */	0x0006
ETH_NF_DTI_TIMEOUT	/* DTI timeout occurred */	0x0007
ETH_NF_HOST_TIMEOUT	/* Workstation timeout occurred */	0x0008
ETH_NF_USER_TEST	/* Set by user */	0x0009
ETH_NF_CONN_ABORT	/* Connection aborted */	0x000A
ETH_NF_INIT_ERR	/* Initialization error */	0x000B

4.8.9 ETH_ClrNetFailStatus()

Task:

The *ETH_ClrNetFailStatus* function resets the SysFail signal. This means that process data can be output again and the status of the SysFail signal is set to 0. The function is assigned a valid node handle for a DTI or MXI data channel as a parameter.

Syntax:

IBDDIRET IBDDIFUNC ETH_ClrNetFailStatus (IBDDIHND nodeHd);

Parameters:

IBDDIHND nodeHd	Node handle (MXI or DTI) for the bus coupler on which the SysFail status is to be reset.

Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Example

Unix / Windows NT/2000

```
IBDDIHND ddiHnd;
{
IBDDIRET ddiRet;
.
.
.
ddiRet = ETH_ClrNetFailStatus (ddiHnd);
.
.
.
}
```

4.9 Other Ethernet Settings

4.9.1 ETH_InitiateManagement()

Task:	The ETH_InitiateManagement function establishes an Ethernet management connection to an Ethernet controller board. An Ethernet management connection is used to set and request specific controller board modes and parameters. An Ethernet controller board can have only one Ethernet management connection open at any one time.	
Syntax:	IBDDIRET IBDDIFUNC ETH_InitiateManagement(CHAR *server, IBDDIHND *hnd)	
Parameters:	CHAR *server	Pointer to a string with the host name or IP address.
	IBDDIHND *hnd	Pointer to a variable in which the node handle of the management connection is entered if the function is called successfully.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

4.9.2 ETH_AbortManagement()

Task:	The *ETH_AbortManagement* function aborts an existing Ethernet management connection.	
Syntax:	IBDDIRET IBDDIFUNC ETH_InitiateManagement(IBDDIHND hnd)	
Parameters:	IBDDIHND hnd	Handle of the management connection that is to be aborted.
Return value:	IBDDIRET	If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

4.9.3 ETH_HardwareReset()

Task:

The *ETH_HardwareReset* function triggers a hardware reset of the entire controller board. However, a hardware reset can only be executed if the hardware reset is enabled on the controller board. If the hardware reset is not enabled, the reset will not be executed and the function will return an error message.

Syntax:

IBDDIRET IBDDIFUNC ETH_HardwareReset(IBDDIHND hnd)

Parameters:

IBDDIHND hnd	Handle of a controller board management connection that is to be reset by a hardware reset.

4.9.4 ETH_EnableHardwareReset()

Task:

The *ETH_EnableHardwareReset* function can be used to enable the hardware reset via TCP/IP, i.e., the controller board can then execute a hardware reset using a function call.

The function does not have to be re-enabled after each reset, as the enable is stored permanently in the controller board and is not lost, even in the event of voltage failure.

The function expects a valid node handle of a management connection as a parameter.

If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Syntax:

INT32 IBDDIFUNC ETH_EnableHardwareReset(IBDDIHND hnd)

The function prototype can be found in the ETH_MNG.H file.

4.9.5 ETH_DisableHardwareReset()

Task:

The *ETH_DisableHardwareReset* function acts as a counterpart to the *ETH_EnableHardwareReset* routine. As the name indicates, this function can be used to reverse a hardware reset enable via TCP/IP, i.e., the hardware reset via TCP/IP is disabled again. The routine requires a valid handle for a management connection as a parameter.

If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Syntax:

IBDDIRET IBDDIFUNC ETH_DisableHardwareReset(IBDDIHND hnd)

The function prototype can be found in the ETH_MNG.H file.

4.9.6 ETH_GetHardwareResetMode()

Task:

The *ETH_GetHardwareResetMode* function can be used to determine whether the hardware reset via TCP/IP is enabled or disabled.

The function expects a valid node handle of a management connection as a parameter.

If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

Syntax:

INT32 IBDDIFUNC ETH_GetHardwareResetMode (IBDDIHND hnd, USIGN16 *modePtr)

The function prototype can be found in the ETH_MNG.H file.

4.9.7 ETH_SetTCPMode()

Task:

The *ETH_SetTCPMode* function modifies the time response of the TCP (Transmission Control Protocol) in the event of an error. The behavior in the event of a lost data packet is also modified.

⚠ If the time response of the TCP is modified, stable communication operation may become impossible. The settings on the controller board must be compatible with the settings of the operating system.

TCP brief display:

TCP is a transport layer protocol that provides connection-oriented protected data transfer. One example of a data protection method used by TCP is that received data packets are acknowledged positively by the receiver. This means that for every data packet received, a corresponding acknowledge message is sent by the receiver. The transmitter can use the acknowledge message to determine whether and which packet has been correctly received.

If no acknowledge message is received within a specified time, the retransmit time, the transmitter repeats the lost packet. In standard TCP implementations, the retransmit time is determined from the runtime of data packets and dynamically adjusted to changing transmission paths. The retransmit time is also increased in the event of consecutive timeouts following a fixed algorithm. The time between the individual repetitions thus increases.

If an acknowledge message has still not been received following a specific number of repetitions, the TCP connection is aborted. However, this standard behavior does not always meet the requirements to exchange process data from the controller board via Ethernet and to process the data on a workstation. Under some circumstances, long repeat times prevent this if a data packet is lost.

To meet the requirements of such applications, the TCP on the controller board has been modified accordingly and extended to include two additional modes. These modes can also be set and modified using the *ETH_SetTCPMode* service.

Syntax:

INT32 IBDDIFUNC ETH_SetTCPMode(IBDDIHND hnd, USIGN16 mode, USIGN16 value)

The *value* transfer parameter is only evaluated in mode 2 (see below). In the other modes, the value entered in *value* is not used.

If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

At present, the controller board recognizes three different TCP modes, whose time response is very different in places. A corresponding symbolic constant can be found in the ETH_MNG.H file for each of these modes.

☞ Please note that after every change of mode a hardware reset must be executed to activate the new mode.

Syntax:

```
#define ETH_FIXED_FAST_TCP_MODE    1    /* TCP
retransmit time = 36 ms    */
#define ETH_VAR_FAST_TCP_MODE      2    /* TCP
retransmit time variable */
#define ETH_STD_TCP_MODE           3    /* Standard */
```

Mode 1 (ETH_FIXED_FAST_TCP_MODE)

This mode is the **fastest** mode possible and is designed for workstations whose retransmit time can be modified to small values and is required by the application to resend lost packets as quickly as possible. The retransmit time for this mode is fixed to a value of 36 milliseconds (35.15625 milliseconds to be precise), i.e., if no acknowledgment of a sent data packet has been received within 36 milliseconds, the controller board resends the lost data packet once this time has elapsed.
If the transmitter does not receive an acknowledgment of this data packet either, the data packet is resent after 36 milliseconds. This process can be repeated up to 48 times. The data packet can therefore be sent a total of 49 times.
If an acknowledge message has still not been received after the 49th attempt, the TCP connection is aborted. Thus, the total maximum time from the data packet first being sent until the connection is aborted is:
49 x 36 milliseconds = 1.764 seconds

The time between repetitions is constant and is not dynamically adjusted.

Mode 2 (ETH_VAR_FAST_TCP_MODE)

In mode 2, the retransmit time is not fixed to a time of 47 milliseconds, but can be modified by the user in 12 ms steps. The smallest possible value for the retransmit time is 47 milliseconds and the largest possible value is 3.01 seconds. The retransmit time is not transmitted in milliseconds, but rather as an 8-bit value, which is converted accordingly in the controller board.
The smallest valid value is 3 and the largest valid value is 255. If a value (n) is transmitted that is smaller or larger than the permitted value, it is automatically corrected to the smallest possible value (3) or the largest possible value (255). The following equation can be used to calculate the retransmit time in seconds:

Retransmit time in seconds = (3/256) x (n+1) with $3 \leq n \leq 255$

As with mode 1, the retransmit time set by the user is fixed and is not dynamically adjusted. Like mode 1, a data packet can be repeated a maximum of 48 times.

Mode 3 (ETH_STD_TCP_MODE)

Mode 3 is the standard TCP mode and is thus implemented in the TCP/IP implementation as standard. This means that the retransmit time is calculated from the round trip time (RTT) and is dynamically adjusted to the existing network. In addition, the retransmit time is doubled following each consecutive failed retransmit attempt and kept at the current value after seven failed retransmit attempts.
The TCP connection is automatically aborted after a total of 12 repetitions. This mode is useful for all applications that have a slow network (modem) or whose time constraints are not so critical. A particular disadvantage of this mode is, for example, the long time before a faulty cable caused by a connection abort after the maximum number of repetitions has been exceeded is indicated to the user as an error.

TCP modes should only be modified if the user is aware of the consequences of such a modification and is also familiar with the time response in his or her network and host computer. For example, if the retransmit time selected on the controller board is too short it may result in sudden loss of connection.

Controller boards in TCP mode 2 are supplied with a retransmit time of 250 ms as standard. These settings are sufficient for most host computers and networks. An adjustment is necessary if a modem connection is to be used, for example. In this case, it is useful to set mode 3 (*ETH_STD_TCP_MODE*) so that the retransmit time is dynamically set.

4.9.8 ETH_GetTCPMode()

Task:

The *ETH_GetTCPMode* function is used to read the TCP mode that is currently set. The routines transmit pointers to variables, in which the TCP mode and the retransmit time are entered, instead of the values for the TCP mode and the retransmit time.

Syntax:

INT32 IBDDIFUNC ETH_GetTCPMode(IBDDIHND hnd, USIGN16 *modePtr, USIGN16 *valuePtr)

The TCP mode of the controller board is entered in the variable referenced by *modePtr*. The values correspond to those that are used to set the TCP mode (see *ETH_SetTCPMode*). The same applies for variables referenced by *valuePtr*. The value in these variables is only valid if TCP mode 2 is set. The retransmit time in seconds can then also be calculated using the equation specified on page 4-55.

If the function is executed successfully, the value 0 (ERR_OK) is returned. Otherwise the return value is an error code.

4.9.9 ETH_SetClientOptions()

Task:

The ETH_SetClientOptions function can be used to set different options in the driver library.

The changes made using this function are **not** saved. Please also note that this function must be called before the first DDI or management connection is opened.

The following options are available:

	Standard Value	Constant for the Command Code
Receive timeout	3 seconds	ETH_OPT_RCV_TIMEOUT
Connect timeout	3 seconds	ETH_OPT_CON_TIMEOUT
Path specification of the IBSETHA file	Current directory	ETH_OPT_IBSETHA_PATH

Syntax to activate:

IBDDIRET IBDDIFUNC ETH_SetClientOptions(int cmd, char *arg, int length)

Parameters:	int cmd	Command code
	char *arg	Pointer to command parameters
	int *length	Size of the parameters (in bytes)

Return values, error messages:	ERR_OK	0x0000	Function executed successfully
	ERR_OPT_INVLD_CMD	0x0200	Unknown command code
	ERR_OPT_INVLD_PARAM	0x0201	Error in transfer parameter

Example:

Set receive timeout to seven seconds:

```
int Timeout;
Timeout = 7
Error = ETH_SetClientOptions ( ETH_OPT_RCV_TIMEOUT,
(char *)&Timeout, sizeof ( Timeout ));
```

4.9.10 ETH_GetClientOptions()

Task:

The ETH_GetClientOptions function can be used to read the options in the driver library.

The function can be called at any time.

The following options are available:

	Constant for the Command Code
Receive timeout in seconds	ETH_OPT_RCV_TIMEOUT
Connect timeout in seconds	ETH_OPT_CON_TIMEOUT
Path specification of the IBSETHA file	ETH_OPT_IBSETHA_PATH

Syntax to activate:

IBDDIRET IBDDIFUNC ETH_GetClientOptions(int cmd, char *arg, int *length)

Parameters:	int cmd	Command code
	char *arg	Pointer to command parameters
	int *length	Size of the parameters (in bytes)

Return values, error messages:

ERR_OK	0x0000	Function executed successfully
ERR_OPT_INVLD_CMD	0x0200	Unknown command code
ERR_OPT_INVLD_PARAM	0x0201	Error in transfer parameter

Example:

Read back connect timeout:

```
int Value;
int sizeOfValue;
Error = ETH_GetClientOptions ( ETH_OPT_CON_TIMEOUT,
(char *)&Value, &sizeOfValue);
```

Section **5**

This section informs you about

– programming support macros

5 Programming Support Macros

5.1 Macros for Process Data Conversion

The following macros simplify the transmission of data (commands, messages, process data) between the host and the INTERBUS controller board.

– The INTERBUS master protocol chip (IPMS) of the controller board stores its data in the MPM in Motorola format (68xxx range) and also expects this format when reading.

– The host processor processes data in Intel format, which is typical for IBM-compatible PCs.

The numbering of words and bytes within a data field is inverted for these formats. The macros convert the data between Motorola and Intel format and write it to the specified buffer so that a process image can be easily created in Intel format, for example.

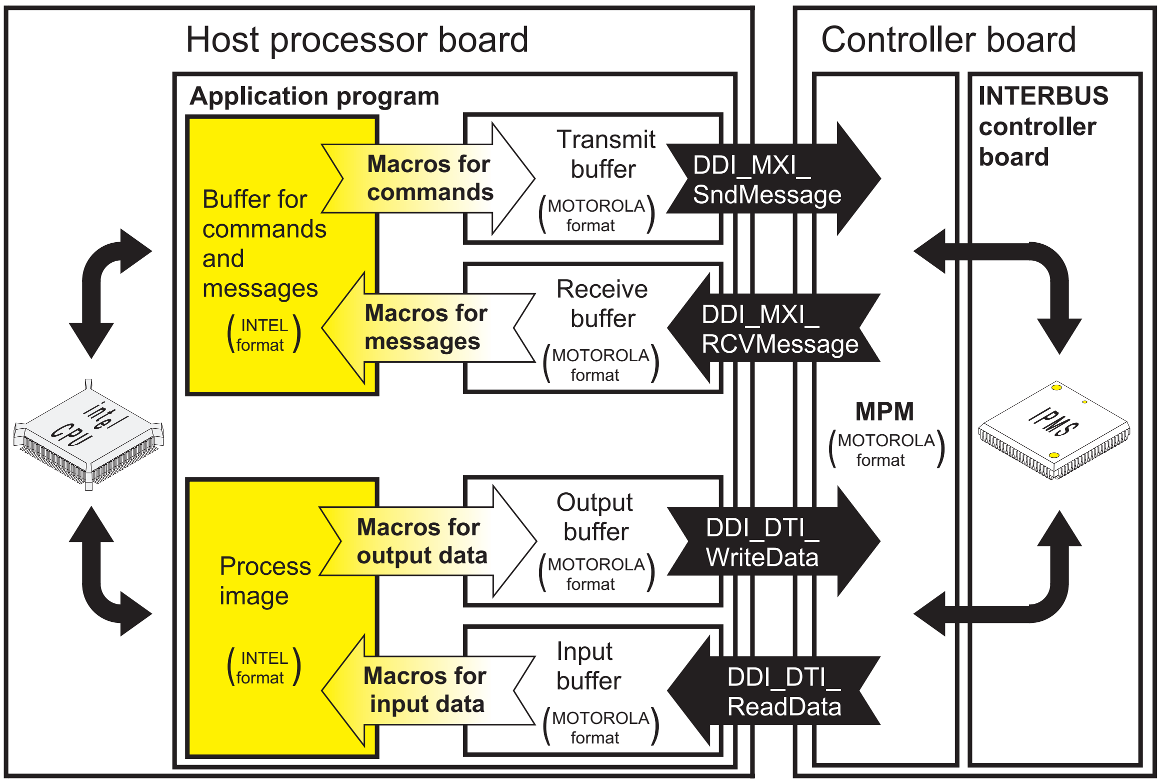


Figure 5-1 Using macros for process data conversion

Table 5-1 Overview of the macros for process data conversion

Macro	Task	Page
IB_SetCmdCode	Enters the command code (16-bit) in the specified transmit buffer	5-7
IB_SetParaCnt	Enters the parameter count (16-bit) in the specified transmit buffer	5-7
IB_SetParaN	Enters a parameter (16-bit) in the specified transmit buffer	5-7

Table 5-1 Overview of the macros for process data conversion

Macro	Task	Page
IB_SetParaNHiByte	Enters the high byte (bit 8 to 15) of a parameter in the specified transmit buffer	5-7
IB_SetParaNLoByte	Enters the low byte (bit 0 to 7) of a parameter in the specified transmit buffer	5-10
IB_SetBytePtrHiByte	Returns the address of a parameter entry starting with the high byte (bit 8 to 15)	5-8
IB_SetBytePtrLoByte	Returns the address of a parameter entry starting with the low byte (bit 0 to 7)	5-8
IB_GetMsgCode	Reads a message code (16-bit) from the specified receive buffer	5-9
IB_GetParaCnt	Reads the parameter count (16-bit) from the specified receive buffer	5-9
IB_GetParaN	Reads a parameter (16-bit) from the specified receive buffer	5-9
IB_GetParaNHiByte	Reads the high byte (bit 8 to 15) of a parameter from the specified receive buffer	5-10
IB_GetParaNLoByte	Reads the low byte (bit 0 to 7) of a parameter from the specified receive buffer	5-10
IB_GetBytePtrHiByte	Returns the address of a parameter entry starting with the high byte (bit 8 to 15)	5-10
IB_GetBytePtrLoByte	Returns the address of a parameter entry starting with the low byte (bit 0 to 7)	5-11
IB_PD_GetLongDataN	Reads a long word (32-bit) from the specified position in the input buffer	5-12
IB_PD_GetDataN	Reads a word (16-bit) from the specified position in the input buffer	5-12
IB_PD_GetDataNHiByte	Reads the high byte (bit 8 to 15) of a word from the input buffer	5-12
IB_PD_GetDataNLoByte	Reads the low byte (bit 0 to 7) of a word from the input buffer	5-12

Table 5-1 Overview of the macros for process data conversion

Macro	Task	Page
IB_PD_GetBytePtrHiByte	Returns the address of a word starting with the high byte (bit 8 to 15)	5-13
IB_PD_GetBytePtrLoByte	Returns the address of a word starting with the low byte (bit 0 to 7)	5-13
IB_PD_SetLongDataN	Writes a long word (32-bit) to the output buffer	5-14
IB_PD_SetDataN	Writes a word (16-bit) to the output buffer	5-14
IB_PD_SetDataNHiByte	Writes the high byte (bit 8 to 15) of a word to the output buffer	5-14
IB_PD_SetDataNLoByte	Writes the low byte (bit 0 to 7) of a word to the output buffer	5-15
IB_PD_SetBytePtrHiByte	Returns the address of a word starting with the high byte (bit 8 to 15)	5-15
IB_PD_SetBytePtrLoByte	Returns the address of a word starting with the low byte (bit 0 to 7)	5-15

The macros are defined for different operating systems and compilers in the Device Driver Interface so that they can be used universally.

☞ The include files, libraries, and units required to use macros are described in Sections 2, 3, 4, and 5.

☞ The first parameter no. (word no.) is always number 1.

5.1.1 Macros for Converting the Data Block of a Command

IB_SetCmdCode (n, m)

Task: This macro converts a command code (16-bit) to Motorola format and enters it in the specified transmit buffer.

Parameters:

n(USIGN8 FAR *):	Pointer to the transmit buffer
m(USIGN16):	Command code to be entered

IB_SetParaCnt (n, m)

Task: This macro converts the parameter count (16-bit) to Motorola format and enters it in the specified transmit buffer. The call is only required when dealing with a command with parameters. The parameter count specifies the number of subsequent parameters in words.

Parameters:

n(USIGN8 FAR *):	Pointer to the transmit buffer
m(USIGN16):	Parameter count to be entered

IB_SetParaN (n, m, o)

Task: This macro converts a parameter (16-bit) to Motorola format and enters it in the specified transmit buffer. The call is only required when dealing with a command with parameters.

Parameters:

n(USIGN8 FAR *):	Pointer to the transmit buffer
m(USIGN16):	Parameter no. (word no.)
o(USIGN16):	Parameter value to be entered

IB_SetParaNHiByte (n, m, o)

Task: This macro converts the high byte (bit 8 to 15) of a parameter to Motorola format and enters it in the specified transmit buffer (see also IB_SetParaN).

Parameters:

n(USIGN8 FAR *):	Pointer to the transmit buffer
m(USIGN16):	Parameter no. (word no.)
o(USIGN8):	Parameter to be entered (byte)

IB_SetParaNLoByte(n, m, o)

Task: This macro converts the low byte (bit 0 to 7) of a parameter to Motorola format and enters it in the specified transmit buffer (see also IB_SetParaN).

Parameters:	n(USIGN8 FAR *):	Pointer to the transmit buffer
	m(USIGN16):	Parameter no. (word no.)
	o(USIGN8):	Parameter to be entered (byte)

IB_SetBytePtrHiByte (n, m)

Task: This macro returns the address of a parameter entry starting with the high byte (bit 8 to 15).
The address is a *USIGN8 FAR ** data type.

Parameters:	n(USIGN8 FAR *):	Pointer to the transmit buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the high byte of the parameter in the transmit buffer.

IB_SetBytePtrLoByte (n, m)

Task: This macro returns the address of a parameter entry starting with the low byte (bit 0 to 7).
The address is a *USIGN8 FAR ** data type.

Parameters:	n(USIGN8 FAR *):	Pointer to the transmit buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the low byte of the parameter in the transmit buffer.

5.1.2 Macros for Converting the Data Block of a Message

IB_GetMsgCode (n)

Task: This macro reads the message code (16-bit) from the specified receive buffer and converts it to Intel format.

Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
Return value:	(USIGN16):	Message code

IB_GetParaCnt (n)

Task: This macro reads the parameter count (16-bit) from the data block of the message and converts it to Intel format. The parameter count specifies the number of subsequent parameters in words.

Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
Return value:	(USIGN16):	Parameter count

Remark: This macro only reads the parameter count for messages that also have parameters.

IB_GetParaN (n, m)

Task: This macro reads a parameter value (16-bit) from the data block of the message and converts it to Intel format.

Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN16):	Parameter value

Remark: This macro only reads the parameter count for messages that also have parameters.

IB_GetParaNHiByte (n, m)

Task:	This macro reads the high byte (bit 8 to 15) of a parameter from the specified receive buffer and converts it to Intel format.	
Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8):	Parameter value (byte)
Remark:	This macro only reads the parameter count for messages that also have parameters.	

IB_GetParaNLoByte (n, m)

Task:	This macro reads the low byte (bit 0 to 7) of a parameter from the specified receive buffer and converts it to Intel format.	
Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8):	Parameter value (byte)
Remark:	This macro only reads the parameter count for messages that also have parameters.	

IB_GetBytePtrHiByte (n, m)

Task:	This macro returns the address of a parameter entry starting with the high byte (bit 8 to 15). The address is a *USIGN8 FAR ** data type.	
Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the high byte of a parameter in the receive buffer.

IB_GetBytePtrLoByte (n, m)

Task: This macro returns the address of a parameter entry starting with the low byte (bit 0 to 7).

The address is a *USIGN8 FAR ** data type.

Parameters:	n(USIGN8 FAR *):	Pointer to the receive buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the low byte of a parameter in the receive buffer.

5.1.3 Macros for Converting Input Data

Macros are provided in the IBS_MACR.H file for converting long words, words, and bytes from Motorola format to Intel format. Addressing is always word-oriented here.

IB_PD_GetLongDataN(n, m, o)

Task: This macro reads a long word (32-bit) from a specified position in the input buffer and converts it to Intel format. The word index in the input buffer is used as a position. The macro reads the long word starting from the specified word address over two words.

Parameters:	n (USIGN8 FAR *)	Pointer to the input buffer
	m (USIGN16)	Parameter no. (word no.)
	o (USIGN32)	Process data item (32-bit)

IB_PD_GetDataN (n, m)

Task: This macro reads a word (16-bit) from a specified position in the input buffer and converts it to Intel format.

Parameters:	n(USIGN8 FAR *):	Pointer to the input buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8):	Parameter data item (16-bit)

IB_PD_GetDataNHiByte (n, m)

Task: This macro reads the high byte (bit 8 to 15) of a word from the input buffer and converts it to Intel format.

Parameters:	n(USIGN8 FAR *):	Pointer to the input buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8):	Parameter data item (8-bit)

IB_PD_GetDataNLoByte (n, m)

Task: This macro reads the low byte (bit 0 to 7) of a word from the input buffer and converts it to Intel format.

Parameters:	n(USIGN8 FAR *):	Pointer to the input buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8):	Parameter data item (8-bit)

IB_PD_GetBytePtrHiByte (n, m)

Task: This macro returns the address of a word starting with the high byte (bit 8 to 15).

Parameters:	n(USIGN8 FAR *):	Pointer to the input buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR*):	Address of the high byte of a word in the input buffer.

IB_PD_GetBytePtrLoByte (n, m)

Task: This macro returns the address of a word starting with the low byte (bit 0 to 7).

Parameters:	n(USIGN8 FAR *):	Pointer to the input buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the low byte of a word in the input buffer.

5.1.4 Macros for Converting Output Data

Macros are provided in the IBS_MACR.H file for converting long words, words, and bytes from Intel format to Motorola format. Addressing is always word-oriented here.

IB_PD_SetLongDataN (n, m, o)

Task: This macro converts a long word (32-bit) to Motorola format and writes it to the specified position in the output buffer. The word index in the output buffer is used as a position. The macro writes the long word starting from the specified word address over two words.

Parameters:

Parameters:	n (USIGN8 FAR *)	Pointer to the output buffer
	m (USIGN16)	Parameter no. (word no.)
	o (USIGN32)	Process data item (32-bit)

IB_PD_SetDataN (n, m, o)

Task: This macro converts a word (16-bit) to Motorola format and writes it to the specified position in the output buffer.

Parameters:	n(USIGN8 FAR *):	Pointer to the output buffer
	m(USIGN16):	Parameter no. (word no.)
	o(USIGN16):	Process data item (16-bit)

IB_PD_SetDataNHiByte(n, m, o)

Task: This macro converts the high byte (bit 8 to 15) of a word to Motorola format and writes it to the specified position in the output buffer.

Parameters:	n(USIGN8 FAR *):	Pointer to the output buffer
	m(USIGN16):	Parameter no. (word no.)
	o(USIGN8):	Process data item (8-bit)

IB_PD_SetDataNLoByte (n, m, o)

Task:	This macro converts the low byte (bit 0 to 7) of a word to Motorola format and writes it to the specified position in the output buffer.	
Parameters:	n(USIGN8 FAR *):	Pointer to the output buffer
	m(USIGN16):	Parameter no. (word no.)
	o(USIGN8):	Process data item (8-bit)

IB_PD_SetBytePtrHiByte (n, m)

Task:	This macro returns the address of a word starting with the high byte (bit 8 to 15).	
Parameters:	n(USIGN8 FAR *):	Pointer to the output buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR*):	Address of the high byte of a word in the output buffer.

IB_PD_SetBytePtrLoByte (n, m)

Task:	This macro returns the address of a word starting with the low byte (bit 0 to 7).	
Parameters:	n(USIGN8 FAR *):	Pointer to the output buffer
	m(USIGN16):	Parameter no. (word no.)
Return value:	(USIGN8 FAR *):	Address of the low byte of a word in the output buffer.

Section 6

This section informs you about
– diagnostics for driver software

6 Diagnostics for Driver Software

The driver software diagnostics use error messages and error codes for the individual functions. These error codes can be used to precisely define the cause of an error. An operating system related offset (ERR_BASE) is added to the codes listed here. This offset is already taken into consideration when using error message definitions.

6.1 Overview of DDI Messages

Table 6-1 Overview of DDI messages

Code	Constant	Error Description	Page
0000$_{hex}$	ERR_OK	The function was executed successfully	6-7
0080$_{hex}$	ERR_INVLD_BOARD_NUM	Non-permissible or invalid board number	6-8
0081$_{hex}$	ERR_INVLD_IO_ADDR	Invalid I/O address	6-8
0082$_{hex}$	ERR_INVLD_MPM_ADDR	Invalid address for the MPM window	6-8
0083$_{hex}$	ERR_INVLD_INTR_NUM	Invalid interrupt	6-9
0084$_{hex}$	ERR_INVLD_CARD_CODE	Invalid board code	6-9
0085$_{hex}$	ERR_INVLD_NODE_HD	Invalid node handle specified	6-10
0086$_{hex}$	ERR_INVLD_NODE_STATE	Node handle of a data channel that is already closed specified	6-10
0087$_{hex}$	ERR_NODE_NOT_READY	Desired node not ready	6-10
0088$_{hex}$	ERR_WRONG_DEV_TYP	Incorrect node handle	6-10
0089$_{hex}$	ERR_DEV_NOT_READY	Local bus master not ready yet	6-10
008A$_{hex}$	ERR_INVLD_PERM	Access type not enabled for channel	6-11
008B$_{hex}$	ERR_TSR_NOT_LOADED	Device driver not loaded	6-18

Table 6-1 Overview of DDI messages

Code	Constant	Error Description	Page
008C$_{hex}$	ERR_INVLD_CMD	Utility function is not supported by driver Version 0.9	6-11
008D$_{hex}$	ERR_INVLD_PARAM	Command contains invalid parameter	6-11
0090$_{hex}$	ERR_NODE_NOT_PRES	Node not available	6-12
0091$_{hex}$	ERR_INVLD_DEV_NAME	Unknown device name used	6-12
0092$_{hex}$	ERR_NO_MORE_HNDL	Device driver resources used up	6-12
0093$_{hex}$	ERR_NO_MORE_SUBNODE	There are no handles available for this subnode	6-12
0094$_{hex}$	ERR_SUBNODE_IN_USE	Subnode is already in use	6-13
0095$_{hex}$	ERR_PARA_OUT_OF_RANGE	The parameter has invalid values	6-13
0096$_{hex}$	ERR_AREA_EXCDED	Access exceeds limit of selected data area	6-15
0097$_{hex}$	ERR_INVLD_DATA_CONS	Specified data consistency is not permitted	6-15
0098$_{hex}$	ERR_DTA_NOT_PRESENT	The selected data area is not available	6-9
0099$_{hex}$	ERR_MPM_NOT_AVALBL	It is not possible to access the MPM	6-10
009A$_{hex}$	ERR_MSG_TO_LONG	Message or command contains too many parameters	6-13
009B$_{hex}$	ERR_NO_MSG	No message present	6-13
009C$_{hex}$	ERR_NO_MORE_MAILBOX	No further mailboxes of the required size free	6-14
009D$_{hex}$	ERR_SVR_IN_USE	Send vector register in use	
009E$_{hex}$	ERR_SVR_TIMEOUT	Invalid node called	6-14
009F$_{hex}$	ERR_AVR_TIMEOUT	Invalid node called	6-14

PHŒNIX CONTACT

Table 6-1 Overview of DDI messages

Code	Constant	Error Description	Page
00A2$_{hex}$	ERR_SYSFAIL	SysFail warning is active, however data has been exchanged	
00A3$_{hex}$	ERR_MSG_EVENT_IN_USE	The event description specified is already in use.	
00A4$_{hex}$	ERR_MSG_EVENT_ADDRESS	The address specified is not in the permitted area.	
00A5$_{hex}$	ERR_MSG_EVENT_LENGTH	The length specified is not permitted. (0 or greater than the permitted area)	
00A6$_{hex}$	ERR_MSG_EVENT_AREA	The memory area to be monitored was exceeded.	
00A7$_{hex}$	ERR_MSG_EVENT_MAX_ITEM	The number of event descriptions has been exceeded.	
00A9$_{hex}$	ERR_PLUG_PLAY	Invalid write access to process data in plug and play mode	
00B0$_{hex}$	ERR_NODE_IN_USE	Notification mode activated twice for one node (Windows)	
00C0$_{hex}$	ERR_INVLD_PID	Incorrect process identifier specified	
00C1$_{hex}$	ERR_BLK_MODE_IS_ENBLD	Blocked mode is already enabled	
00C2$_{hex}$	ERR_THREAD_IS_WAITING	Another thread is already using the notification mode of this node	
00C8$_{hex}$	ERR_INVLD_NODE	Invalid node number	
00C9$_{hex}$	ERR_INVLD_MEMORY	Invalid receive buffer	
00CA$_{hex}$	ERR_INVLD_NOTIFY_MODE	Invalid notification mode	
00CB$_{hex}$	ERR_BLOCK_TIMEOUT	Waiting time for a message exceeded	
00CC$_{hex}$	ERR_INVLD_NOTIFY_STATE	Invalid notification status	
00D1$_{hex}$	ERR_INVLD_HWND	Invalid Windows handle	
00D2$_{hex}$	ERR_BOARD_NOT_PRES	Board not entered in *IBDDIWIN.INI*	
00D3$_{hex}$	ERR_INVLD_INI_PARAM	Invalid parameter in *IBDDIWIN.INI*	

Table 6-1 Overview of DDI messages

Code	Constant	Error Description	Page
00E1$_{hex}$	ERR_WRONG_BOARD_REV	The controller board does not support the command used	
00E2$_{hex}$	ERR_DRV_INVLD_FUNC	Function not supported by the driver	6-11
00E3$_{hex}$	ERR_SEC_PERM_DENIED	Access not permitted, incorrect password	
00EB$_{hex}$	ERR_DTI_IN_USE	DPM driver: The handshake for exchanging process data is not yet complete. Remedy: Repeat access	
00EC$_{hex}$	ERR_DTI_NOT_RDY	DPM driver: Error during handshake for exchanging process data. Remedy: Repeat access	
0100$_{hex}$	ERR_STATE_CONFLICT	This service is not permitted in the selected operating mode of the controller	
0101$_{hex}$	ERR_INVLD_CONN_TYPE	Service called via an invalid connection	
0102$_{hex}$	ERR_ACTIVATE_PD_CHK	Process IN data monitoring could not be activated	
0103$_{hex}$	ERR_DATA_SIZE	The data volume is too large	
0110$_{hex}$	ERR_NUMBER_OF_DRIVERS _EXCEEDED	Maximum number of drivers already loaded (IBDDIWNT.DLL)	6-18
0111$_{hex}$	ERR_LOAD_DLL_FAILED	Driver DLL could not be found (IBDDIWNT.DLL)	6-18
0200$_{hex}$	ERR_OPT_INVLD_CMD	Unknown command	
0201$_{hex}$	ERR_OPT_INVLD_PARAM	Invalid parameter	
1010$_{hex}$	ERR_IBSETH_OPEN	The IBSETHA file cannot be opened	
1013$_{hex}$	ERR_IBSETH_READ	The IBSETHA file cannot be read	
1014$_{hex}$	ERR_IBSETH_NAME	The device name cannot be found in the file	
1016$_{hex}$	ERR_IBSETH_INTERNET	The system cannot read the computer name/host address	

6.2 Positive DDI Message

ERR_OK **0000_{hex}**

Meaning: After successful execution of a function, the driver software generates this message as a positive acknowledgment.

Cause: No errors occurred when executing the function. If a function is not executed successfully, the driver software generates one of the following listed error messages.

6.3 DDI Error Messages

If the Device Driver Interface generates one of the following error messages as a negative acknowledgment, the function called previously was not processed successfully.

6.3.1 Error Messages When Initializing the Controller Board

ERR_INVLD_BOARD_NUM **0080_{hex}**

Cause: A non-permissible or invalid board number was used.

Remedy: Specify a valid board number. For example, 1, 2, 3 to 8 are permitted depending on the driver used.

ERR_INVLD_IO_ADDR **0081_{hex}**

Cause: The I/O address specified for the controller board is not permitted.

Remedy: Specify a valid I/O address. The following values are permitted: 100_{hex}, 108_{hex}, 110_{hex}, 118_{hex}, ... , $3E8_{hex}$, $3F0_{hex}$, $3F8_{hex}$

ERR_INVLD_MPM_ADDR **0082_{hex}**

Cause: The base address specified in the memory area of the PC for the 4 kbyte MPM window (MPM address) is outside the area supported by the controller board.

Remedy: Specify an MPM address within the area supported by the controller board ($A0000_{hex}$ to $FF000_{hex}$).

The controller board uses an address area of 4 kbytes from this base address onwards. Ensure that this area is not already being used by other boards. An automatic check is **not** carried out. Since this memory area is already used extensively (BIOS, etc.), in practice, the available address area is usually limited to parts of address segments D and E (addresses $D0000_{hex}$ to $EFFFF_{hex}$). The standard value (default) is $D0000_{hex}$.

ERR_INVLD_INTR_NUM **0083_{hex}**

Cause: The specified interrupt is not permitted.

Remedy: Interrupts IRQ3, IRQ5, IRQ7, IRQ9, IRQ10, IRQ11, IRQ12, and IRQ15 are permitted.

☞ When several controller boards are used in one host, another interrupt must be used for every installed controller board. Interrupts IRQ10, IRQ11, and IRQ12 are not usually used on a standard PC and are thus available for the device driver. The other interrupts are frequently used by standard PC components (serial interfaces COM1 and COM2, network cards, etc.) and therefore should not be used for controller boards.

ERR_INVLD_CARD_CODE **0084_{hex}**

Cause: An incorrect board number was detected for a controller board (for example, PCCB).

Remedy:
– Check the controller board hardware

6.3.2 General Error Messages

These error messages can occur when calling any DDI function.

ERR_DTA_NOT_PRESENT **0098_{hex}**

Cause: The specified data area does not exist. The area of a node was selected that is not available.

Remedy:
– Check the parameters entered
– Check that the selected node is actually available (hardware)

	ERR_MPM_NOT_AVALBL **0099_{hex}**
Cause:	It is not possible to access the MPM. A reset may have been triggered on the controller board.
Remedy:	Uninstall the driver and restart.
	ERR_INVLD_NODE_HD **0085_{hex}**
Cause:	An invalid node handle was used when calling the function.
Remedy:	Use the valid node handle of a successfully opened data channel.
	ERR_INVLD_NODE_STATE **0086_{hex}**
Cause:	An invalid node handle was used when calling the function. This is the handle of a data channel that has already been closed.
Remedy:	Open the data channel or use one that is already open.
	ERR_NODE_NOT_READY **0087_{hex}**
Cause:	The node to be used has not yet indicated it is "Ready", i.e., the node ready bit has not been set in the MPM status register. The cause of this can be, e.g., a hardware fault.
Remedy:	- Check that the controller board has been started - Check whether the PC is in the reset state - Check the BIOS settings; double-assigned IRQs
	ERR_WRONG_DEV_TYPE **0088_{hex}**
Cause:	Incorrect node handle. An attempt was made, e.g., to access the mailbox interface with the node handle of the data interface.
	ERR_DEV_NOT_READY **0089_{hex}**
Cause:	The INTERBUS controller board was addressed, even though it was not ready ("READY" LED).

ERR_INVLD_PERM **$008A_{hex}$**

Cause: An attempt was made to execute a function on a channel for which the relevant access rights were not logged in when opening the data channel. This error occurs, e.g., if you attempt to write to the data interface, but read-only rights were specified when opening the channel (DDI_READ constant).

Remedy: Close the channel and open it again with modified access rights.

ERR_INVLD_CMD **$008C_{hex}$**

Cause: This error message is generated when certain new help functions of the new LDDI_TSR.LIB are used with an old driver (Version < 0.9).

Remedy: Use a more up-to-date driver (Version ≥ 0.9).

ERR_INVLD_PARAM **$008D_{hex}$**

Cause: This error message is generated when certain new help functions of the new LDDI_TSR.LIB are used with an old driver (Version < 0.9).

Remedy: Use a more up-to-date driver (Version ≥ 0.9).

ERR_DRV_INVLD_FUNC **$00E2_{hex}$**

Cause: The used function is not supported by the DDI and driver.

Remedy: Use a more up-to-date driver, which supports these parameter values.

6.3.3 Error Messages When Opening a Data Channel

ERR_NODE_NOT_PRES **0090_{hex}**

Cause: An attempt was made to open a data channel to a node that does not exist.

Remedy: Select the correct node.
Available nodes:

Host:	Node 0
IBS master:	Node 1
Coprocessor board:	Node 2

ERR_INVLD_DEV_NAME **0091_{hex}**

Cause: An unknown device name was specified as a parameter when opening a data channel.

Remedy: Select a correct device name according to Table 2-1 on page 2-9.

ERR_NO_MORE_HNDL **0092_{hex}**

Cause: Device driver resources used up. No further data channels can be opened. If you exit a program without closing the data channels in use, they will remain open. Additional data channels will be opened the next time the program is started. After this program has been started a number of times, the maximum permitted number of data channels that can be opened simultaneously will be reached and no more will be available.

Remedy: Close a data channel that is not required or reinstall the device driver. Every time you exit a program you should close all data channels that have been used.

ERR_NO_MORE_SUBNODE **0093_{hex}**

Cause: The maximum number of handles to a node have already been opened.

Remedy: Close data channels that are no longer required (on exiting a program).

ERR_SUBNODE_IN_USE **0094_{hex}**

Cause: Opening using the specified subnode was rejected, since there is already an open connection with this subnode, and only one connection is permitted.

Remedy: Close the previously opened connection.

ERR_PARA_OUT_OF_RANGE **0095_{hex}**

Cause: A parameter has a value, which is not permitted.

Remedy: Check your parameters for the function used or use a more up-to-date driver, which supports these parameter values.

6.3.4 Error Messages Relating to the Transfer of Messages/Commands

ERR_MSG_TOO_LONG **$009A_{hex}$**

Cause: If an error message occurs when sending a command, then the length of the command exceeds the maximum number of permitted parameters.

Remedy: Reduce the number of parameters.

Cause: If an error message occurs when receiving a message, then the length of the message exceeds the length of the receive buffer specified.

Remedy: Increase the length of the receive buffer.

ERR_NO_MSG **$009B_{hex}$**

Cause: This message occurs if an attempt has been made to retrieve a message using the *DDI_MXI_RCV_MESSAGE* function, but no messages are present for the node specified by the node handle.

ERR_NO_MORE_MAILBOX **$009C_{hex}$**

Cause: You have requested too many mailboxes within a short space of time.

No further mailboxes of the required size are available. Note the maximum useful mailbox size (1020 bytes).

Remedy: Increase the time interval between individual mailbox requests and try again.

Select a smaller mailbox or wait until a mailbox of the required size is free again.

ERR_SVR_TIMEOUT **$009E_{hex}$**

Meaning: If a message placed in the MPM by the INTERBUS controller board is not retrieved by the MPM device addressed, it does not reset the acknowledge message bit set by the INTERBUS controller board, i.e., the MPM device addressed does not indicate "*Message detected*". After a specific time has expired ("timeout"), the INTERBUS controller board generates the error message *ERR_SVR_TIMEOUT*. If this error message occurs repeatedly, it must be assumed that the node being addressed is no longer ready to accept the message.

Cause: Invalid node called, for example:

An attempt was made to address the coprocessor board (COP), but it is faulty.

Remedy: Please get in touch with Phoenix Contact.

ERR_AVR_TIMEOUT **$009F_{hex}$**

Meaning: An acknowledge message bit was set when reading a message to indicate to the communication partner that a message has been processed and the mailbox is free again. This bit must be reset by the communication partner to indicate that it has recognized that the mailbox is free again. If this reset does not take place within a set time, this error message is generated.

Cause: Invalid node called.

Remedy: Please get in touch with Phoenix Contact.

6.3.5 Error Messages Relating to Process Data Transfer

These errors only occur when accessing the data interface (DTI).

ERR_AREA_EXCDED **0096_{hex}**

Meaning: Access exceeds the upper limit of the selected data area.

Cause: The data record to be read or written is too large. The function can read a maximum of 4 kbytes in one call.

Remedy: Only read or write data records with a maximum size of 4 kbytes.

Cause: The upper area limit (4 kbytes over the start of the device area) has been exceeded.

Remedy: Make sure that the total of address offset, relative address, and data length to be read does not exceed the upper area limit.

ERR_INVLD_DATA_CONS **0097_{hex}**

Cause: An invalid value was specified for the data consistency (1 byte).

Remedy: Specify a permissible data consistency with one of the following constants:

DTI_DATA_BYTE	:Byte data consistency (1 byte)

ERR_PLUG_PLAY **$00A9_{hex}$**

Cause: An attempt was made to gain write access to process data in plug and play mode. This is not permitted for security reasons.

Remedy: Deactivate plug and play mode using the "Set_Value" command with the value "0" or switch to read access.

	ERR_STATE_CONFLICT	**0100$_{hex}$**
Cause:	A service was called, which is not permitted in this operating mode.	
Remedy:	Switch to an operating mode in which the desired call can be executed.	

	ERR_INVLD_CONN_TYPE	**0101$_{hex}$**
Cause:	A service was called, which cannot be executed via the selected connection.	
Remedy:	Select a connection type via which the service can be executed.	

	ERR_ACTIVE_PD_CHK	**0102$_{hex}$**
Cause:	Process IN data monitoring failed to activate.	

	ERR_DATA_SIZE	**0103$_{hex}$**
Cause:	The data volume to be transmitted exceeds the maximum permissible size.	
Remedy:	Transmit the data in several cycles.	

	ERR_OPT_INVLD_CMD	**0200$_{hex}$**
Cause:	An attempt was made to execute an unknown (invalid) command.	
Remedy:	Select a valid command.	

ERR_OPT_INVLD_PARAM **0201_{hex}**

Cause: An attempt was made to execute a command with unknown (invalid) parameters.

Remedy: Enter valid parameters.

ERR_ETH_RCV_TIMEOUT **1001_{hex}**

Cause: The time limit for receiving a data telegram was exceeded.

Remedy: The Ethernet connection was interrupted or an incorrect IP address was entered. Increase the timeout value.

ERR_IBSETH_OPEN **1010_{hex}**

Cause: The IBSETHA file cannot be opened.

Remedy: The IBSETHA file does not exist or is in the wrong directory.

ERR_IBSETH_READ **1013_{hex}**

Cause: The IBSETHA file cannot be read.

Remedy: The file exists but cannot be read. You may not have read access.

ERR_IBSETH_NAME **1014_{hex}**

Cause: The device name cannot be found in the file.

Remedy: The name, which was transferred to the DDI_DEVOPEN_NODE () function, is not in the IBSETHA file.

ERR_IBSETH_INTERNET **1016_{hex}**

Cause: The system cannot read the computer name/host address.

Remedy: The IP address entered in the IBSETHA file is incorrect or the symbolic name cannot be found in the host file.

6.3.6 Error Messages Under DOS

ERR_TSR_NOT_LOADED **$008B_{hex}$**

Cause: The device driver is not yet loaded, however an attempt was made to access it.

Remedy: Load the *IBSISA.EXE* TSR program as the device driver on the host.

6.3.7 Error Messages Under Microsoft Windows

ERR_NUMBER_OF_DRIVERS_EXCEEDED **0110_{hex}**

Cause: The maximum number of external drivers is already loaded.

Remedy: Close the connections to drivers/devices that are currently no longer required.

ERR_LOAD_DLL_FAILED **0111_{hex}**

Cause: A required DLL driver could not be found.

Remedy:

– Check whether all required drivers are available
– Check the registry database for the correct path information to the drivers

A Index

We Are Interested in Your Opinion!

We would like to hear your comments and suggestions concerning this document.

We review and consider all comments for inclusion in future documentation.

Please fill out the form on the following page and fax it to us or send your comments, suggestions for improvement, etc. to the following address:

Phoenix Contact GmbH & Co. KG
Marketing Services
Dokumentation INTERBUS
32823 Blomberg
GERMANY

Phone	+49 - (0) 52 35 - 3-00
Telefax	+49 - (0) 52 35 - 3-4 18 08
E-Mail	tecdoc@phoenixcontact.com

PHŒNIX CONTACT

FAX Reply

Phoenix Contact GmbH & Co. KG
Marketing Services
Dokumentation INTERBUS

Date: ______

Fax No: +49 - (0) 52 35 - 3-4 18 08

From:

Company:		Name:	
		Department:	
Address:		Job function:	
City, ZIP code:		Phone:	
Country:		Fax:	

Document:

Designation:	IBS PC SC SWD UM E	Revision:	05	Order No.:	27 45 17 2

My Opinion on the Document

Form	**Yes**	**In part**	**No**
Is the table of contents clearly arranged?	☐	☐	☐
Are the figures/diagrams easy to understand/helpful?	☐	☐	☐
Are the written explanations of the figures adequate?	☐	☐	☐
Does the quality of the figures meet your expectations/needs?	☐	☐	☐
Does the layout of the document allow you to find information easily?	☐	☐	☐
Contents	**Yes**	**In part**	**No**
Is the phraseology/terminology easy to understand?	☐	☐	☐
Are the index entries easy to understand/helpful?	☐	☐	☐
Are the examples practice-oriented?	☐	☐	☐
Is the document easy to handle?	☐	☐	☐
Is any important information missing? If yes, what?	☐	☐	☐

______

Other Comments:

______

______

533305