INTERBUS-S

User Manual

Driver Software for Controller Boards for IBM-compatible PCs

Type: IBS PC CB SWD UM E

Revision: A

Order No.: 27 53 96 0

This manual is valid for the driver software version 0.9 for the controller boards:

IBS PC CB/I-T	Order No. 27 80 84 9
IBS PC CB/COP/I-T	Order No. 27 54 51 6
IBS PC CB/RTX486/I-T	Order No. 27 61 47 0

PHOENIX CONTACT

Please Observe the Following:

In order to guarantee that your use of this manual is as straightforward as possible and that hardware is used safely in the installation, operation and maintenance phases, we request that you carefully read and observe the following instructions:

Explanation of Symbols Used

The *attention* symbol refers to erroneous handling, which could lead to damage to the hardware or software, or, in indirect connection with dangerous process peripherals (e.g., unprotected shafts or motors with actuator functions), to light to severe personal injury. The symbol is always located to the left of the tagged text.

The *hand* symbol gives you tips and advice on the efficient use of hardware and on software optimization, to save you from performing extra work, for example. In addition, text marked in this way informs you of system-related maximum and minimum conditions that must absolutely be observed to achieve error-free operation. The hand is also found in front of clarifications of terms.

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

We are Interested in Your Opinion

We are constantly attempting to improve the quality of our manuals. Should you have any suggestions or recommendations for improvement of the contents and layout of our manuals, we would appreciate it if you would send us your comments. Please use the universal telefax form at the end of the manual for this.

Statement of Legal Authority

Phoenix Contact reserves the right to make any technical changes that serve for the purpose of technical progress.

Phoenix Contact reserves all rights in the case of a patent award or listing of a registered design. External products are always named without reference to patent rights. The existence of such rights shall not be excluded, however.

The use of products described in this manual is oriented exclusively to qualified application programmers and software engineers familiar with automation technology and the applicable national standards. Phoenix Contact assumes no liability for erroneous handling of or damage to Phoenix Contact or external products resulting from disregard of information contained in this manual.

A Table of Contents

Section 1

Driver Software Overview

This section provides information on the driver software available for different

- operating systems;
- programming languages;
- compilers.

1 Driver Software Overview

Driver software for various operating systems is available for the host controller boards. In addition, you can select from several compilers for the programming languages C and Pascal. This section shows the possible combinations of operating systems and compilers on your PC and the coprocessor boards of the IBS PC CB/COP/I-T and IBS PC CB/RTX486/I-T.

Table 1-1: Operating systems

Operating system	Version	Processor
Microsoft® DOS	5.0, 6.2	PC (host)
SMA TDOS®	3.5a	COP386
Technosoft RTXDOS-16®	6.1	COP486
Microsoft Windows®	3.1	Host
IBM OS/2®	2.1	Host

Table 1-2: Compiler

Compiler	Version
Borland C/Turbo C®	3.0 / 3.1
Microsoft C/C++®	7.0
Microsoft Visual C++®	1.0
IBM C Set/2®	2.0
Borland Turbo Pascal®	6.0 / 7.0
Microsoft Visual Basic®	3.0

See the sections referred to in Tables 1-3 and 1-4 for a detailed description.

Table 1-3: Compiler/operating system combinations on the host

		Operating systems		
		Microsoft DOS®	Microsoft Windows®	IBM OS/2®
Compiler	Borland C/Turbo C®	Section 2	Sections 2 and 4	—
	Microsoft C/C++®			
	Microsoft Visual C++®			
	IBM C®	—	—	Sections 2 and 5
	Borland Turbo Pascal®	Section 3	Sections 3 and 4	—
	Microsoft Visual Basic®	In preparation		

Table 1-4: Compiler/operating system combinations on the coprocessor board

		Operating systems	
		TDOS®	RTXDOS®
Compiler	Borland C/Turbo C®	Section 2	
	Microsoft C/C++®		
	Borland Turbo Pascal®	Section 3	

Coprocessor board (COP) programming under DOS does not differ from host programming under DOS. Therefore, a program created on the host under DOS can also be executed on the coprocessor board.

Section 2

Driver Software for C Under DOS

This section provides information on

- the implementation and the functions of the device driver interfaces;
- the required include files and libraries;
- device drivers for DOS.

2 Driver Software for C Under DOS

2.1 Structure of the Driver Software on the Host PC

Link the device driver interface in the form of a library to your application programs. The device drivers for DOS are TSR programs (similar to the drivers of network adapters).

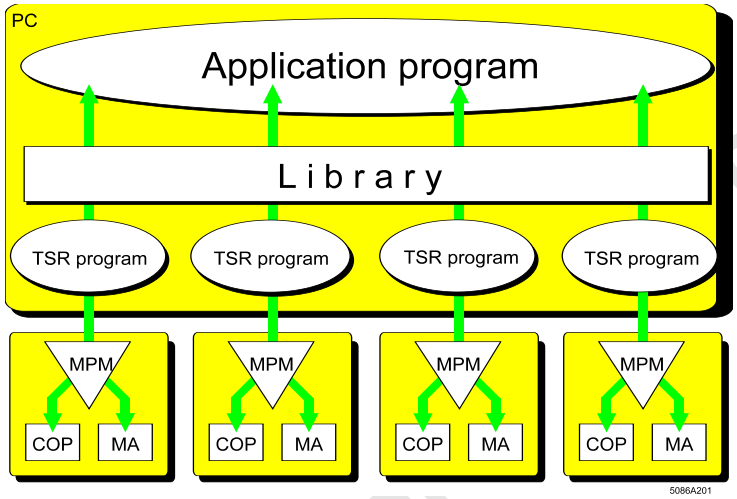

Figure 2-1: IBS driver software under Microsoft DOS® for *C*

A device driver, i.e. a TSR program, must be installed for each controller board! The device driver installation is described in Section 4 *Installation and First Startup* of the *IBS PC CB UM E* manual.

2.2 Structure of the Driver Software on the COP

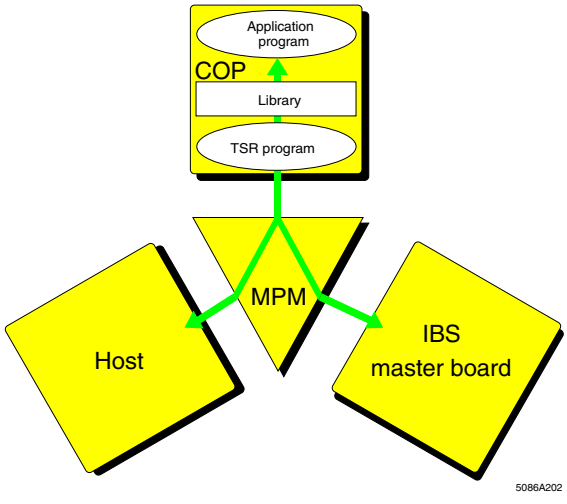

Figure 2-2: Driver software for C on the coprocessor board

Coprocessor board (COP) programming does not differ from host programming under DOS. From driver software version 0.9 onwards, linking with special libraries, too, is no longer required for the processor board, because the same libraries are used for the operation on the host and on the COP. Therefore, a program created on the host under DOS can be executed on the coprocessor board.

For the operation on the COP, use the TSR program IBSCOP.EXE as device driver, instead of IBSPCCB.EXE.

2.3 Libraries and Include Files

Libraries

To facilitate the user's work with the driver software, versions 0.9 and higher combine all required DDI and auxiliary functions in one library. This library is used on the host PC as well as on the coprocessor board. Therefore, an application program can be run on the host PC or on the coprocessor board without previous recompiling or linking. A prerequisite is that the appropriate drivers (TSR programs) are loaded on both systems.

Only the functions for reading from and writing to the coprocessor board's SRAM are not available on the host PC, where the error message *ERR_INVLD_CMD* ($008C_{hex}$) is returned when they are called. Now as before, the library is available as large or medium model (Microsoft C Version 7.0 and higher, and Borland C++ Version 3.0 and higher).

Include files

To simplify also the handling of the include files, only the include file *IBS_DOS.H* needs to be incorporated with driver software 0.9 and higher. All other required include files are called from this include file. However, you may also call the required include files individually.

In addition, when the *IBS_DOS.H* is used it is no longer necessary to manually enter compiler switches in the program or in the compiler's command line. The required constant declaration (IBS_DOS_DRV) then takes place within *IBS_DOS.H*.

Table 2-1: Libraries and include files

Memory model:	Library:	Include file
medium	MDDI_TSR.LIB	IBS_DOS.H (calls STDTYPES.H, COMPILER.H, IBS_CM.H, DDI_USR.H, DDI_ERR.H, DDI_LIB.H, PC_UTIL.H and DDI_MACR.H)
large	LDDI_TSR.LIB	

The include file *DDI_MACR.H* allows the use of the macro functions described in Section 4.2.4. The macros are defined in this file.

If you do not want to use *IBS_DOS.H* and call the required include files separately in the program, insert the instruction *#define IBS_DOS_DRV* before incorporating the include files (either in the program text or as a compiler option).

Examples:

```
#define IBS_DOS_DRV
#include "stdtypes.h"
#include "ddi_usr.h"
...
```

or

```
cl /C+  /O+ /DIBS_DOS_DRV ...
```

2.4 Functions for DOS

Table 2-2: Overview of the DDI functions for DOS

Function	Task	Page
DDI_DevOpenNode	Opens a data channel to a node	2-6
DDI_DevCloseNode	Closes a data channel to a node	2-7
DDI_MXI_SndMessage	Writes a message to the MPM	2-8
DDI_MXI_RcvMessage	Reads a message from the MPM	2-9
DDI_DTI_ReadData	Reads data from the MPM	2-10
DDI_DTI_WriteData	Writes data to the MPM	2-11

Table 2-3: Overview of the hardware control functions

Function	Task	Page
COP_WriteStaticRAM	Writes a number of bytes to the SRAM of the COP	2-14
COP_ReadStaticRAM	Reads a number of bytes from the SRAM of the COP	2-15
GetDIPSwitch	Reads out the settings of the DIP switches for setting the boot configuration	2-13
GetSysFailRegister	Reads the contents of the SysFail register	2-14
EnableWatchDog	Enables a watchdog	2-16
TriggerWatchDog	Triggers a watchdog	2-16
GetWatchDogState	Reads out the state of a watchdog	2-17
ClearWatchDog	Resets the state of a watchdog	2-17
GetIBSDiagnostic	Evaluates the IBS master board state	2-18

2.4.1 Device Driver Interface Functions

2.4.1.1 Data Channel Management Functions

DDI_DevOpenNode

Task: This function opens a data channel to a node.

Synopsis: INT16 DDI_DevOpenNode(CHAR *devName,INT16 perm,INT16 *nodeHd);

Parameters:

*devName	Device name of the device to be accessed. The name identifies the controller board (board number) and the MPM node on this board (see Section 5.2.3.1 *Data Channel Management* in the *IBS PC CB UM E* manual.)
perm	The access permission specifies with which access rights the data channel may be accessed. A distinction can be made between read, write and read/write accesses.
*nodeHd	The node handle is a value assigned by the DDI, which is used to find an allocation to the opened node in all other functions.

Positive acknowledgment: Node handle

Negative acknowledgment:

DDI error code	Gives a more detailed specification of the error found when opening the data channel to a node (see Section 8 *DDI Error Messages*).
Cause:	- unknown device name - node does not exist

Call syntax:

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

Constants for the access rights:

DDI_READ	Read access only
DDI_WRITE	Write access only
DDI_RW	Read and write access

In the current version of the driver software (V 0.9), the same data channel (same device name) between two MPM nodes (e.g. host and COP) may be opened only once. If the same data channel between two MPM nodes is open more than once at a time, the data of one data channel overwrites the data of another, as it uses the same MPM memory area. In this case **no** error message is output.

Examples: Wrong:

```
DDI_DevOpenNode("IBB1N1_D", ... , nodeHandle1)
DDI_DevOpenNode("IBB1N1_D", ... , nodeHandle2)
```

Correct:

```
DDI_DevOpenNode("IBB1N1_D", ... , nodeHandle1)
DDI_DevOpenNode("IBB1N2_D", ... , nodeHandle2)
```

(See Section 5.2.3.1 *Data Channel Management* in the *IBS PC CB UM E* manual)

DDI_DevCloseNode

Task: This function closes a data channel to a node which had been opened with *DDI_DevOpenNode()* before. After this function has been called successfully, the device is no longer "connected“ with the calling program, and the node handle is no longer valid.

Synopsis: INT16 DDI_DevCloseNode(INT16 nodeHd);

Parameter:	nodeHd	The node handle specifies the node to be closed.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	DDI error code	Gives a more detailed specification of the error that occurred when the function was called (see Section 8 *DDI Error Messages*).
	Cause:	- Invalid device name - Invalid node handle - ode does not exist

2.4.1.2 Mailbox Interface Functions

DDI_MXI_SndMessage

Task: This function places a message or a command in a mailbox.

Synopsis:
INT16 DDI_MXI_SndMessage
(INT16 node_Hd,T_DDI_MXI_ACCESS *ddi_mxi_acc);

Parameters:	nodeHd	The node handle is the logical number (handle) of a channel previously opened on the device driver interface.
	*ddi_mxi_acc	Pointer to a data structure of the type T_DDI_MXI_ACCESS (see below).
T_DDI_MXI_ACCESS:	Data structure with the elements required to transfer a message/command	
Structure elements:	msgType	The firmware 3.x does not yet support the structure element *message type*. Set it to 0.
	msgLength	The structure element *message length* contains the total length of the message to be sent in bytes. The maximum permissible total length (see below) is 1024.
	*msgBlk	The structure element **msgBlk* is a pointer to a message block containing the message to be sent in mailbox syntax. The mailbox syntax format is described in the *IBS PC CB UM E* manual.
	DDIUserID	The structure element *DDIUserID* is not yet supported by the firmware 3.x. Set *DDIUserID* to 0.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	DDI error code	Specifies why the function could not be executed (see Section 7 *DDI Error Messages*)
	Cause	invalid node handle; no suitable mailbox found; the message exceeds the maximum usable mailbox length (1020 bytes = 1024 bytes minus 2 command code bytes minus 2 parameter count bytes).

Call syntax:

```
INT16  DDI_MXI_SndMessage(
               INT16 node_Hd,                  /* IN : node handle     */
               T_DDI_MXI_ACCESS * ddi_mxi_acc);/* IN : pointer to
                                                  mailbox access
                                                  structure            */
```

Format of the structure *T_DDI_MXI_ACCESS*:

```
typedef struct {
               INT16 msgType        /* Message type                    */
               USIGN16 msgLength;   /* Message length                  */
               USIGN16 DDIUserID;   /* DDI_User_ID                     */
               USIGN8 *msgBlk;      /* pointer to array for the message */
} T_DDI_MXI_ACCESS;
```

DDI_MXI_RcvMessage

Task:	This function fetches a message from a mailbox. It is used, for example, to fetch the acknowledgment of an IBS command. It does not wait for the acknowledgment! If there is no acknowledgment, a message to this effect is output in the *DDI error code* parameter.	
Prerequisite:	The length of the provided receive buffer **must** be entered in the *msgLength* component of the *T_DDI_MXI_ACCESS* structure. The driver checks the size of the receive buffer before reading in and generates the error message *ERR_MSG_TO_Long* ($009A_{hex}$) if the received message is longer than the memory space available.	
Synopsis:	USIGN16 DDI_MXI_RcvMessage (USIGN16 nodeHd,T_DDI_MXI_ACCESS *ddi_ mxi_acc);	
Parameters:	nodeHd	The node handle is the logical number of a channel previously opened on the DDI.
	*ddi_mxi_acc	Pointer to a data structure of the type T_DDI_MXI_ACCESS (see below).
T_DDI_MXI_ACCESS:	Data structure with the elements required to transfer a message/command	
Structure elements:	msgType	The fimware 3.x does not yet support the structure element *message type*. Set it to 0.
	msgLength	Before calling the function *DDI_MXI_RcvMessage*, enter the size of the available input buffer in bytes, using the structure element *message length*. After a message has been received successfully, the structure element *message length* contains the actual length of the message in bytes.
	*msgBlk	The structure element *msgBlk is a pointer to a message block containing the received message in mailbox syntax. The *IBS PC CB UM E* manual describes the structure of the mailbox syntax.
	DDIUserID	The firmware 3.x does not yet support the structure element *DDIUserID*. Set *DDIUserID* to 0.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	DDI error code	Specifies why the function could not be executed (see section 7 *DDI Error Messages*).
	Cause	- invalid node handle - receive buffer too small - no message available

Call syntax:

```
USIGN16  DDI_MXI_RcvMessage(
              USIGN16 nodeHd,                  /* IN : node handle     */
              T_DDI_MXI_ACCESS *ddi_ mxi_acc);/* OUT: pointer to
                                               mailbox access structure*/
```

Format of the structure *T_DDI_MXI_ACCESS*:

```
typedef struct {
              INT16 msgType;       /* message type                    */
              USIGN16 msgLength;   /* message length                  */
              USIGN16 DDIUserID;   /* DDI_User_ID                     */
              USIGN8 *msgBlk;/* pointer to array for the message*/
} T_DDI_MXI_ACCESS;
```

2.4.1.3 Data Interface Functions

DDI_DTI_ReadData

Task: This functions reads data from the MPM via the data interface. It places the data in Motorola format in the specified buffer.

Comment: Before the data is further processed, use the macros for converting input data. These macros convert the input data from the Motorola format to the Intel format (see Section 6).

Synopsis: INT16 DDI_DTI_ReadData
(INT16 node_Hd,T_DDI_DTI_ACCESS *ddi_dti_acc);

Parameters:

nodeHd	This parameter specifies the node.
*ddi_dti_acc	Pointer to a data structure of the type T_DDI_DTI_ACCESS (see below).

T_DDI_DTI_ACCESS: Data structure with the elements required for transferring process data

Structure elements:

length	The structure element *length* contains the number of data to be read in bytes. The maximum number is 1024 bytes.
address	The structure element *address* specifies the DTI address of a process data word in the MPM (see Section *Organization of the MPM* in the *IBS PC CB UM E* manual).
dataCons	The structure element *data consistency* specifies the data consistency to be used for the access.
*data	This structure element is a pointer to the buffer where the data to be read is to be stored.

Constants for the possible data consistency areas:

DTI_DATA_BYTE	: Byte data consistency (1 byte)
DTI_DATA_WORD	: Word data consistency (2 bytes)
DTI_DATA_LWORD	: Long-word data consistency (4 bytes)
DTI_DATA_48 Bit	: 48-bit data consistency (6 bytes)

The data consistency areas *DTI_DATA_LWORD* and *DTI_DATA_48BIT* are only possible for an access to the IBS master board from firmware 3.72

onwards.

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment:
DDI error code: Specifies in more detail the error that has occurred during process data reading (see Section 7 *DDI Error Messages*).
Cause:
- invalid node handle
- invalid parameters
- data area boundaries exceeded

Call syntax:

```
INT16  DDI_DTI_ReadData(
               INT16 node_Hd,                    /* IN : node handle     */
               T_DDI_DTI_ACCESS *ddi_dti_acc); /* IN : dti access
                                                    structure       */
```

Format of the structure *T_DDI_DTI_ACCESS*:

```
typedef struct {
               USIGN16 length;    /* number of bytes to read/write     */
               USIGN16 address;   /* address to read/write process data*/
               INT16 dataCons;    /* data consistency of the access    */
               USIGN8 *data;      /* pointer to data to read/write     */
} T_DDI_DTI_ACCESS;
```

DDI_DTI_WriteData

Task: This function writes data via the data interface to the MPM. For this purpose, the function requires data in the Motorola format.

Comment: Before writing data to the MPM, use the output data conversion macros, which convert the output data from the Intel format to the Motorola format (see Section 6).

Synopsis:
INT16 DDI_DTI_WriteData
(INT16 nodeHd,T_DDI_DTI_ACCESS *ddi_dti_acc);

Parameters:
nodeHd: The node handle specifies the node.
*ddi_dti_acc: Pointer to a data structure of the type T_DDI_DTI_ACCESS (see below).

T_DDI_DTI_ACCESS: Data structure with the elements required for the transfer of process data

Structure elements:
length: The structure element *length* contains the number of data to be written in bytes. The maximum number is 1024 bytes (1 kbyte).
address: The structure element *address* specifies the DTI address of a process data word in the MPM (see Section *Organization of the MPM* in the *IBS PC CB UM E* manual).
dataCons: The structure element *data consistency* specifies the

	data consistency for the access.
*data	Pointer to the buffer from which the data to be written is to be taken.

Constants for possible data consistency areas:

DTI_DATA_BYTE	: Byte data consistency (1 byte)
DTI_DATA_WORD	: Word data consistency (2 bytes)
DTI_DATA_LWORD	: Long-word data consistency (4 bytes)
DTI_DATA_48 Bit	: 48-bit data consistency (6 bytes)

The data consistency areas *DTI_DATA_LWORD* and *DTI_DATA_48BIT* are only possible for access to the master board from firmware 3.72 onwards.

Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	DDI error code	Specifies in more detail an error that occurred during a process data write process (see Section 7 *DDI Error Messages*).
	Cause	- invalid node handle - invalid parameters - the data area boundaries are exceeded

Call syntax:

```
INT16  DDI_DTI_WriteData(
               INT16 nodeHd,                    /* IN : node handle     */
               T_DDI_DTI_ACCESS *ddi_dti_acc); /* IN : dti access
                                                          structure*/
```

Format of the structure *T_DDI_DTI_ACCESS*:

```
typedef struct {
               USIGN16 length;    /* number of bytes to read/write     */
               USIGN16 address;   /* address to read/write process data*/
               INT16 dataCons;    /* data consistency of the access    */
               USIGN8 *data;      /* pointer to data to read/write     */
} T_DDI_DTI_ACCESS;
```

2.4.2 Hardware Control Functions

2.4.2.1 DIP Switch Inquiry

GetDIPSwitch

Task: The function *GetDIPSwitch* writes the settings of the DIP switch for boot configuration setting to the variable referenced by *dataPtr*. Therefore, the user can determine the boot configuration during program execution, e.g. to ascertain which board (PC or COP) controls the InterBus-S system. The eight switches are mapped to bits 0 to 7 of the word, e.g. DIP switch 1 is assigned to bit 0, DIP switch 2 to bit 1, etc., of the word. When a switch is in the ON position, the associated bit is zero. The unused bits 8 to 15 of the word are in any case in status one.

Synopsis: INT16 FAR GetDIPSwitch(USIGN16 boardNumber, USIGN16 FAR *dataPtr)

Parameters:

boardNumber	Board number (PC: 1 to 4, COP: 1)
*dataPtr	Pointer to a variable where the ascertained switch settings are entered).

Positive acknowledgment:

ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment:

ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning: An invalid board number was specified.
ERR_TSR_NOT_LOADED ($008B_{hex}$)
Meaning: The specified board has not been installed, or the driver for it has not been loaded.)

2.4.2.2 SysFail Register Monitoring

GetSysFailRegister

Task: The *GetSysFailRegister* function writes the contents of the SysFail register to the variable referenced by *sysFailRegPtr*. Bits 0, 4, 8 and 12 of the register indicate whether the SysFail signal of the corresponding board (PC, IBS master and COP) has been activated or not. In the event of a malfunction of an MPM node (e.g. watchdog has initiated a reset), the associated bit in the SysFail register is activated, i.e. set to one. This bit then remains set until the end of the malfunction. The individual bits of the register are assigned as follows to the MPM nodes:
Bit 0 --> host PC
Bit 4 --> IBS master board (MA)
Bit 8 --> coprocessor board (COP)
Bit 12 is not used, as there are only three MPM nodes.

Synopsis: INT16 FAR GetSysFailRegister
(USIGN16 boardNumber,USIGN16 FAR *sysFailRegPtr)

Parameters:		
	boardNumber	Board number (PC: 1 to 4, COP: 1)
	*sysFailRegPtr	Pointer to a variable where the contents of the SysFail register are entered.

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning: An invalid board number was specified.
ERR_TSR_NOT_LOADED ($008B_{hex}$)
Meaning: The specified board has not been installed, or the driver for it has not been loaded.)

2.4.2.3 SRAM Access Functions

(only for IBS PC CB/COP/I-T and IBS PC CB/486RTX/I-T)

COP_WriteStaticRAM

Task: Writes the specified number of bytes from the given address onwards to the static RAM of the COP. The lowest possible address is 0.

Synopsis: INT16 FAR COP_WriteStaticRAM
(USIGN32 address, USIGN16 length, USIGN8 FAR *data)

Parameters:		
	address	Start address in the static RAM
	length	Data record length (number of bytes to be written)
	*data	Pointer to the buffer from which the function is to take the data to be written.

Positive acknowldegment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	ERR_AREA_EXCDED (0096_{hex})	
	Meaning	The data record to be read is too long. The function can read a maximum of 64 kbytes in one call.
	Remedy	Call the function twice to transfer larger data volumes block by block. Increase the start address by 64 kbytes in the second call.
	Meaning	The upper boundary of the area has been exceeded. The static RAM has a size of 128 kbytes.
	Remedy	Ensure that the total ofstart address and data record length does not exceed the area boundary.

COP_ReadStaticRAM

Task:	Reads the specified number of bytes from the specified address onwards from the static RAM of the COP.	
Synopsis:	INT16 FAR COP_ReadStaticRAM(USIGN32 address, USIGN16 length, USIGN8 FAR *data)	
Parameters:	address	Start address in the static RAM
	length	Data record length (number of bytes to be read)
	*data	Pointer to the buffer where the function is to store the data to be read.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	ERR_AREA_EXCDED (0096_{hex})	
	Meaning	The data record to be written is too long. The function can write a maximum of 64 kbytes in one call.
	Remedy	Call the function twice to transfer larger data quantities block by block. Increase the start address by 64 kbytes in the second call.
	Meaning	The upper boundary of the area has been exceeded. The static RAM has a size of 128 kbytes.
	Remedy	Ensure that the total of start address and data record length does not exceed the area boundary.

2.4.2.4 Watchdog Control Functions

EnableWatchDog()

Task: The function enables the watchdog.

Synopsis: INT16 FAR EnableWatchDog(USIGN16 boardNumber)

Parameter: boardNumber — Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning — The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning — An invalid board number was specified.
Remedy — Enter a valid board number.

Comment: After the function has been called, the watchdog must be triggered at regular intervals.
PC: Trigger interval less than 146 ms, otherwise a bit is set in the SysFail register.
COP: Trigger interval less than 125 ms, otherwise a bit resetting the COP is set in the SysFail register.

TriggerWatchDog()

Task: The function triggers the watchdog.

Synopsis: INT16 FAR TriggerWatchDog(USIGN16 boardNumber)

Parameter: boardNumber — Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning — The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning — An invalid board number was specified.
Remedy — Enter a valid board number.

Comment: This call must be repeated at regular intervals, to ensure that the watchdog does not initiate a reset.
PC: Trigger interval less than 146 ms, otherwise a bit is set in the SysFail register.
COP: Trigger interval less than 125 ms, otherwise resetting the COP is set in the SysFail register.

GetWatchDogState()

Task: This function allows you to inquire from your application program whether the watchdog has initiated a reset. If the application program is running on the host, the function inquires automatically the host watchdog state. If the application program is running on the COP, the function inquires automatically the COP watchdog state.

Synopsis: INT16 FAR GetWatchDogState(USIGN16 boardNumber)

Parameter: boardNumber Board number (PC: 1 to 4, COP: 1)

Positive aknowledgment: -

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})

Meaning: An invalid board number was specified.

Remedy: Specify a valid board number.

Return value: Coprocessor board watchdog state:
1 The watchdog initiated the last COP warmstart (software reset).
0 The watchdog did not initiate the last COP warmstart (software reset).
Host watchdog state:
1 The host watchdog initiated a reset.
0 The host watchdog did not initiate a reset.

The return values are no longer available after a hardware reset of the controller board or the host.

ClearWatchDog()

Task: The function resets the watchdog state.

Synopsis: INT16 FAR ClearWatchDog (USIGN16 boardNumber)

Parameter: boardNumber Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning An invalid board number was specified.
Remedy Enter a valid board number.

2.4.3 IBS Diagnostic Function

GetIBSDiagnostic();

Task: Using the GetIBSDiagnostic() function you can evaluate the state of the IBS master board and, therewith, the state of the IBS system.

Synopsis: INT16 GetIBSDiagnostic(USIGN16 boardNumber,T_IBS_DIAG FAR *diagInfo);

Parameters:

boardNumber	Board number (PC: 1 to 4, COP: 1)
*ErrInfo:	Pointer to the structure with error details

T_IBS_DIAG Structure with diagnostic details

Structure elements:

State: The bits of the *state* structure element describe the bus state. Masking (ANDing) the state structure element with the following constants allows to evaluate the state of the IBS system:

```
DIAG_IBS_READY     IBS is ready
DIAG_IBS_RUN       IBS has started and is running
DIAG_IBS_SYS_FAIL  A bit in the SysFail register was
                   set(e.g. by a watchdog)
DIAG_IBS_BSA       A bus segment is disabled
DIAG_IBS_ERROR     IBS master board indicated error
```

errType: The bits of the *errType* structure elements describe error conditions in more detail. Masking (ANDing) the *errType* structure element with the following constants allows you to evaluate the error type:

```
DIAG_CNTRL_ERR     Controller error
DIAG_RMT_BUS_ERR   Remote bus error
                   (e.g. defective remote bus cable)
DIAG_LCL_BUS_ERR   Local bus error
                   (e.g. defective local bus cable)
DIAG_MDL_ERR       IBS module error
                   (e.g. interrupted I/O supply
                   voltage, output overload)
```

diagPara: The evaluation of the structure element *diagPara* differs according to its value and to the structure element *errType*:

- If the structure element *errType* indicates a remote bus, local bus or IBS device error and the value of the structure element *diagPara* is in the range from 0 to 255, *diagPara* specifies the number of the bus segment where the error has occurred. Output the bus segment in decimal notation.
- If the structure element *errType* indicates a remote bus, local bus or IBS device error and the value of the structure element *diagPara* is higher than 255, *diagPara* specifies an error number (E01, E02, E04, E05 or E06). See the description of the message *Bus_Error_Information_Indication* ($80C4_{hex}$) in the controller board manual (IBS PC CB UM E).
- If the structure element *errType* indicates a controller error, the structure element *diagPara* specifies a controller error number (see the list of the controller error numbers). Output the controller error number in hexadecimal notation.

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment: ERR_INVLD_BOARD_NUM (0080_{hex})
Meaning: Invalid board number

Negative acknowledgment: ERR_NODE_NOT_READY (0087_{hex})
Meaning: IBS master board cannot be accessed.
Remedy: Wait a moment and then try again.

Negative acknowledgment: MPM NOT AVAILABLE (0099_{hex})
Meaning: The MPM cannot be accessed.
Remedy: Reinstall the driver.

Evaluate the return values only when the function was executed successfully (positive acknowledgment ERR_OK (0000_{hex})). On return of a negative acknowledgment it is not ensured that the return values reflect the IBS master board state!

Call syntax:

```
INT16 GetIBSDiagnostic(
                USIGN16 BoardNumber,    /*Board number          */
                T_IBS_DIAG FAR *diagInfo);/*Pointer to structure
                                            with error details  */
```

Format of the structure *T_IBS_DIAG FAR*:

```
typedef struct {
                USIGN16  state;     /* Bus state, Ready, Run etc.      */
                USIGN16  errType;   /* Error type, remote bus, local
                                       bus or controller error         */
                USIGN16  diagPara;  /* Supplementary information,
                                       see parameter description on the
previous page                     */
} T_IBS_DIAG;
```

Example: Refer to the next page for a program detail for evaluating the *state* structure element by masking (ANDing) with specified constants:

```
void Diagnose (void)
{
GetIBSDiagnostic(boardNumber, &Errinfo);
                if ((Errinfo.state & DIAG_IBS_READY) = = DIAG_IBS_READY)
                   {
                    printf(„IBS Ready“)
                   }
                if ((Errinfo.state & DIAG_IBS_RUN) = = DIAG_IBS_RUN)
                   {
                    printf(„IBS Run“)
                   }
                    else
                   {
                    printf(„IBS Stop!“)
                   }
}
```

Section 3

Driver Software for Pascal Under DOS

This section provides information on

- the implementation and the functions of the device driver interface;
- the required units;
- device drivers for DOS.

3 Driver Software for Pascal Under DOS

3.1 Structure of the Driver Software on the Host PC

Link the device driver interface in the form of a unit to your application program. The device drivers for DOS are TSR programs (similar to the drivers of network adapters).

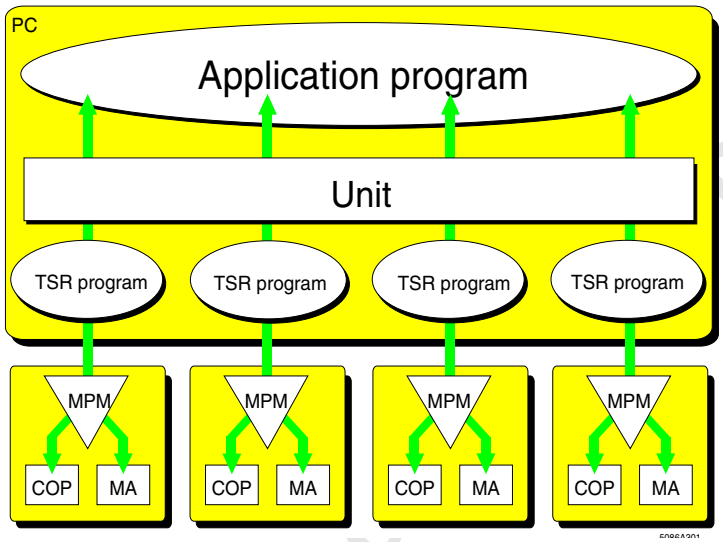

Figure 3-1: IBS driver software under Microsoft-DOS® for *Pascal*

A device driver, i.e. a TSR program, must be installed for each controller board! The device driver installation is described in Section *4 First Installation and Startup* of the *IBS PC CB UM E* manual.

3.2 Structure of the Driver Software on the COP

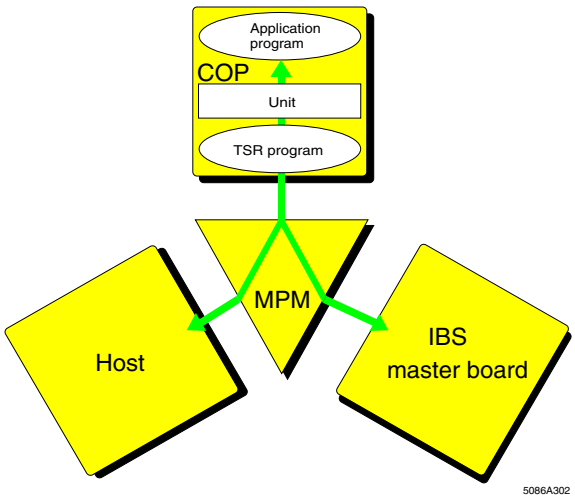

Figure 3-2: IDriver software for *Pascal* on the coprocessor board

Coprocessor board (COP) programming does not differ from host programming under DOS. From driver software version 0.9 onwards, linking with special units, too, is no longer required for the processor board, because the same units are used for the operation on the host and on the COP. Therefore, a program created on the host under DOS can be executed on the coprocessor board.

For the operation on the COP, call the TSR program IBSCOP.EXE as device driver, instead of IBSPCCB.EXE.

3.3 Units

To facilitate the user's work with the driver software, versions 0.9 and higher combine all required DDI and auxiliary functions in the unit DDI_DRV.PAS. This unit is used on the host PC as well as on the coprocessor board. Therefore, an application program can be run on the host PC or on the coprocessor board without previous recompiling or linking. A prerequisite is that the appropriate drivers (TSR programs) are loaded on both systems.

Only the functions for reading from and writing to the coprocessor board's SRAM are not available on the host PC, where the error message *ERR_INVLD_CMD* ($008C_{hex}$) is returned when they are called.

The unit *DDI_VAR.PAS* contains the definitions of the constants and global variables. This unit also permits the use ot the macros described in Section 6. For Pascal, they are declared as functions in this file.

3.4 Functions for DOS

Table 3-1: Overview of the DDI functions for DOS

Function	Task	Page
DDI_DevOpenNode	Opens a data channel to a node	3-6
DDI_DevCloseNode	Closes a data channel to a node	3-7
DDI_MXI_SndMessage	Writes a message to the MPM	3-8
DDI_MXI_RcvMessage	Reads a message from the MPM	3-9
DDI_DTI_ReadData	Reads data from the MPM	3-10
DDI_DTI_WriteData	Writes data to the MPM	3-11

Table 3-2: Overview of the hardware control functions

Function	Task	Page
COP_WriteStaticRAM	Writes a number of bytes to the SRAM of the COP	3-14
COP_ReadStaticRAM	Reads a number of bytes from the SRAM of the COP	3-15
GetDIPSwitch	Reads out the settings of the DIP switch for setting the boot configuration	3-13
GetSysFailRegister	Reads out the contents of the SysFail register	3-14
EnableWatchDog	Enables a watchdog	3-16
TriggerWatchDog	Triggers a watchdog	3-16
GetWatchDogState	Reads out the state of a watchdog	3-17
ClearWatchDog	Resets the state of a watchdog	3-17
GetIBSDiagnostic	Evaluates the IBS master board state	3-18

3.4.1 Functions of the Device Driver Interface

3.4.1.1 Data Channel Management Functions

DDI_DevOpenNode

Task: This function opens a data channel to a node.

Synopsis:
DDI_DevOpenNode
(DevName:StringPtr ;perm:INT16;NodeHd:INT16Ptr):INT16;

Parameters:

DevName	The device name is the name of the device to be accessed. It identifies the controller board (board number) and the MPM node on this controller board (see Section 5.2.3.1 *Data Channel Management* in the *IBS PC CB UM E* manual.
perm	The access permission specifies with which access rights the data channel may be accessed. A distinction can be made between read, write, and read/write accesses.
NodeHd	The node handle is a value assigned by the DDI, which is used to find an allocation to the opened node in all other functions.

Positive acknowledgment: Node handle

Negative acknowledgment:

DDI error code	Gives a more detailed specification of the error found when opening the data channel to a node (see Section 7 *DDI Error Messages*).
Cause	- unknown device name - node does not exist

Call syntax:

```
DDI_DevOpenNode(
            DevName:StringPtr;  { IN:  device name           }
            perm:INT16;         { IN:  access permission    }
            NodeHd:INT16Ptr)    { OUT: address of node handle }
            :INT16;
```

Constants for the access rights:

DDI_READ	Read access only
DDI_WRITE	Write access only
DDI_RW	Read and write access

In the current verion of the driver software (V 0.9), the same data channel (same device name) between two MPM nodes (e.g. host and COP) may be opened only once. If the same data channel between two MPM nodes is open more than once at a time, the data of one data channel overwrites the data of another, as it uses the same MPM memory area. In this case **no** error message is output.

Examples:

Wrong:

```
var ret : USIGN16;
var s : string;

s:= 'IBB1N1_D' +#0;
ret:= DDI_DevOpenNode(addr(s[1]),...., @nodeHd1)

s:= 'IBB1N1_D' +#0;
ret:= DDI_DevOpenNode(addr(s[1]),...., @nodeHd2)
```

Correct:

```
var ret : USIGN16;
var s : string;

s:= 'IBB1N1_D' +#0;
ret:= DDI_DevOpenNode(addr(s[1]),...., @nodeHd1)

s:= 'IBB1N2_D' +#0;
ret:= DDI_DevOpenNode(addr(s[1]),...., @nodeHd2)
```

(See Section 5.2.3.1 *Data Channel Management* in the *IBS PC CB UM E* manual)

The term *(addr(s[1])* is required, as the device driver was written in the programming language "*C*" . It ensures a C-compatible transfer structure.

DDI_DevCloseNode

Task: This function closes a data channel to a node which had been opened with *DDI_DevOpenNode()* before. After this function has been called successfully, the device is no longer "connected" with the calling program, and the node handle is no longer valid.

Synopsis: DDI_DevCloseNode(NodeHd : INT16):INT16;

Parameter:	*nodeHd	The node handle specifies the node to be closed.
Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.
Negative acknowledgment:	DDI error code	Gives a more detailed specification of the error that occurred when the function was called (see Section 8 *DDI Error Messages*).
	Cause	- unknown device name - invalid node handle - node does not exist

3.4.1.2 Mailbox Interface Functions

DDI_MXI_SndMessage

Task: This function places a message or a command in a mailbox.

Synopsis: DDI_MXI_SndMessage
(NodeHd : INT16;mxiAcc : P_DDI_MXI_ACCESS): INT16;

(INT16 node_Hd,T_DDI_MXI_ACCESS *ddi_mxi_acc);

Parameters:	NodeHd	The node handle is the logical number (handle) of a channel previously opened at the DDI.
	mxiAcc	Pointer to a data structure of the type T_DDI_MXI_ACCESS (see below).

T_DDI_MXI_ACCESS: Data structure with the elements required to transfer a command.

Structure elements:	msgType	The firmware 3.x does not support the structure element *message type*. Set it to 0.
	msgLength	The structure element *message length* contains the total length of the message to be sent in bytes. The maximum permissible total length (see below) is 1024.
	msgBlk	The structure element *msgBlk* is a pointer to a message block that contains in mailbox syntax the message to be sent. The mailbox syntax format is described in the *IBS PC CB UM E* manual.
	DDIuserID	The structure element *DDIUserID* is not yet supported by the firmware 3.x. Set *DDIUserID* to 0.

Positive acknowledgment:	ERR_OK (0000_{hex})	
	Meaning	The function was executed successfully.

Negative acknowledgment:	DDI error code	Specifies why the function could not be executed (see Section 7 *DDI Error Messages*)
	Cause	invalid node handle; no suitable mailbox found; the message exceeds the maximum usable mailbox length (1020 bytes = 1024 bytes minus 2 command code bytes minus 2 parameter count bytes).

Call syntax:

```
DDI_MXI_SndMessage(
                NodeHd : INT16;              {IN : node handle          *}
                mxiAcc : P_DDI_MXI_ACCESS) {IN : pointer to mailbox
                                                 access structure     *}
                :INT16;
```

Format of the structure *T_DDI_MXI_ACCESS*:

```
P_DDI_MXI_ACCESS = ^T_DDI_MXI_ACCESS;
T_DDI_MXI_ACCESS = record
                   msgType   : INT16;    {message type                    }
                   msgLength : USIGN16;  {message length                  }
                   DDIUserID : USIGN16;  {DDI_User_ID                     }
                   MsgBlk    : USIGN8Ptr;{pointer to array for the message}
                   end;
```

DDI_MXI_RcvMessage

Task: This function fetches a message from a mailbox. It is used, for example, to fetch the acknowledgment of an IBS command. It does not wait for the acknowledgment! If there is no acknowledgment, a message to this effect is output in the *DDI error code* parameter.

Prerequisite: The length of the provided receive buffer **must** be entered in the *msgLength* component of the *T_DDI_MXI_ACCESS* structure. The driver checks the size of the receive buffer before reading in, and generates the error message *ERR_MSG_TO_Long* ($009A_{hex}$) if the received message is longer than the memory space available.

Synopsis: DDI_MXI_RcvMessage(NodeHd:INT16;mxiAcc:P_DDI_MXI_ACCESS):INT16;

Parameters:

NodeHd	The node handle is the logical number of a channel previously opened at the device driver interface.
mxiAcc	Pointer to a data structure of a type T_DDI_MXI_ACCESS (see below).

T_DDI_MXI_ACCESS: Data structure with the elements required for the transfer of a message/command.

Structure elements:

msgType	The firmware 3.x does not support the structure element *message type*. Set it to 0.
msgLength	Before calling the function *DDI_MXI_RcvMessage*, enter the size of the available input buffer in bytes, using the structure element *message length*. After a message has been received successfully, the structure element *message length* contains the actual length of the message in bytes.
msgBlk	The structure element **msgBlk* is a pointer to a message block that contains in mailbox syntax the message to be sent. The mailbox syntax format is described in the *IBS PC CB UM E* manual.
DDIuserID	The structure element *DDIUserID* is not yet supported by the firmware 3.x. Set *DDIUserID* to 0.

Positive acknowledgment:

ERR_OK (0000_{hex})	
Meaning	The function was executed successfully.

Negative acknowledgment:

DDI error code	Specifies why the function could not be executed (see Section 7 *DDI Error Messages*).

	Cause	- invalid node handle - receive buffer too small for the received message - no message available

Call syntax:

```
DDI_MXI_RcvMessage(
                NodeHd:INT16;              { IN : node handle           }
                mxiAcc:P_DDI_MXI_ACCESS)   { OUT: pointer to mailbox
                                                  access structure      }
                :INT16;
```

Format of the structure *P_DDI_MXI_ACCESS*:

```
P_DDI_MXI_ACCESS = ^T_DDI_MXI_ACCESS;
T_DDI_MXI_ACCESS = record
                msgType   : INT16;    { Message type                    }
                msgLength : USIGN16;  { Message length                  }
                DDIUserID : USIGN16;  { DDI_User_ID                     }
                MsgBlk    : USIGN8Ptr;{ pointer to array for the message}
                end;
```

3.4.1.3 Data Interface Functions

DDI_DTI_ReadData

Task: This functions reads data from the MPM via the data interface. It places the data in Motorola format in the specified buffer.

Comment: Before the data is further processed, use the macros for converting input data. These macros convert the input data from the Motorola format to the Intel format (see Section 6).

Synopsis: DDI_DTI_ReadData(NodeHd:INT16;dtiAcc:P_DDI_DTI_ACCESS):INT16;

Parameters:

NodeHd	The node handle specifies the node.
dtiAcc	Pointer to a data structure of the type T_DDI_MXI_ACCESS (see below).

T_DDI_DTI_ACCESS: Data structure with the elements required for transferring process data.

Structure elements:

length	The structure element *length* contains the number of data to be read in bytes. The maximum number is 1024 bytes.
address	The structure element *address* specifies the DTI address of a process data word in the MPM (see Section *Segmentation of the MPM* in the *IBS PC CB UM E* manual).
dataCons	The structure element *data consistency* specifies the data consistency to be used for the access.
data	Pointer to the buffer where the data to be read is to be stored.

Constants for the possible data consistency areas:

DTI_DATA_BYTE	: Byte data consistency (1 byte)
DTI_DATA_WORD	: Word data consistency (2 bytes)

DTI_DATA_LWORD : Long-word data consistency (4 bytes)
DTI_DATA_48 Bit : 48-bit data consistency (6 bytes)

The data consistency areas *DTI_DATA_LWORD* and *DTI_DATA_48BIT* for access to the IBS master board are only possible from firmware 3.72 onwards.

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment:
DDI error code: Specifies in more detail the error that has occurred during process data reading (see Section 7 *DDI Error Messages*).
Cause:
- invalid node handle
- invalid parameters
- data area boundaries exceeded

Call syntax:

```
DDI_DTI_ReadData(
               NodeHd:INT16;              { IN : node handle          }
               dtiAcc:P_DDI_DTI_ACCESS)   { IN : dti access structure }
               :INT16;
```

Format of the structure *T_DDI_DTI_ACCESS*:

```
P_DDI_DTI_ACCESS = ^T_DDI_DTI_ACCESS;
T_DDI_DTI_ACCESS = record
               length   : USIGN16;  { number of bytes to read/write     }
               address  : USIGN16;  { address to read/write process data}
               dataCons : INT16;    { data consistency of the access    }
               Data     : USIGN8Ptr;{ pointer to data to read/write     }
              end;
```

DDI_DTI_WriteData

Task: This function writes data via the data interface to the MPM. For this purpose, the function requires data in the Motorola format.

Synopsis: DDI_DTI_WriteData(NodeHd:INT16;dtiAcc:P_DDI_DTI_ACCESS):INT16;

Comment: Before writing data to the MPM, use the output data conversion macros. These macros convert the output data from the Intel format to the Motorola format (see Section 6).

Parameters:
NodeHd: The node handle specifies the node.
dtiAcc: Pointer to a data structure of the type T_DDI_DTI_ACCESS (see below).

T_DDI_DTI_ACCESS: Data structure with the elements required for the transfer of process data.

Structure elements:
length: The structure element *length* contains the number of data to be read in bytes. The maximum number is 1024 bytes (1 kbyte).
address: The structure element *address* specifies the DTI

	address of a process data word in the MPM (see Section *Segmentation of the MPM* in the *IBS PC CB UM E* manual).
dataCons	The structure element *data consistency* specifies the data consistency for the access.
data	Pointer to the buffer from which the data to be written is to be taken.

Constants for possible data consistency areas:

DTI_DATA_BYTE	: Byte data consistency (1 byte)
DTI_DATA_WORD	: Word data consistency (2 bytes)
DTI_DATA_LWORD	: Long-word data consistency (4 bytes)
DTI_DATA_48 Bit	: 48-bit data consistency (6 bytes)

The data consistency areas *DTI_DATA_LWORD* and *DTI_DATA_48BIT* are possible for accesses to the IBS master board only from firmware 3.72 onwards.

Positive acknowledgment:

ERR_OK (0000_{hex})	
Meaning	The function was executed successfully.

Negative acknowledgment:

DDI error code	Specifies in more detail an error that occurred during a process data write process (see Section 7 *DDI Error Messages*).
Cause	- invalid node handle - invalid parameters - the data area boundaries are exceeded

Call syntax:

```
DDI_DTI_WriteData(
                NodeHd:INT16;              { IN : node handle          }
                dtiAcc:P_DDI_DTI_ACCESS)   { IN : dti access structure }
                :INT16;
```

Format of the structure *T_DDI_DTI_ACCESS*:

```
P_DDI_DTI_ACCESS = ^T_DDI_DTI_ACCESS;
T_DDI_DTI_ACCESS = record
                length    : USIGN16;  { number of bytes to read/write     }
                address   : USIGN16;  { address to read/write process data}
                dataCons  : INT16;    { data consistency of the access    }
                Data      : USIGN8Ptr;{ pointer to data to read/write     }
                end;
```

3.4.2 Hardware Control Functions

3.4.2.1 DIP Switch Inquiry

GetDIPSwitch

Task: The function *GetDIPSwitch* writes the settings of the DIP switch for boot configuration setting to the variable referenced by *dataPtr*. Therefore, the user can determine the boot configuration during program execution, e.g. to ascertain which board (PC or COP) controls the InterBus-S system. The eight switches are mapped to bits 0 to 7 of the word, e.g. DIP switch 1 is assigned to bit 0, DIP switch 2 to bit 1, etc., of the word. When a switch is in the ON position, the associated bit is zero. The unused bits 8 to 15 of the word are in any case in status one.

Synopsis: GetDIPSwitch(boardNumber : USIGN16; dataPtr : USIGN16Ptr):INT16;

Parameters:

boardNumber	Board number (PC: 1 to 4, COP: 1)
dataPtr	Pointer to a variable where the ascertained switch settings are entered).

Positive acknowledgment:

ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment:

ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning: An invalid board number was specified.
ERR_TSR_NOT_LOADED ($008B_{hex}$)
Meaning: The specified board has not been installed, or the driver for it has not been loaded.

3.4.2.2 SysFail Register Monitoring

GetSysFailRegister

Task: The *GetSysFailRegister* function writes the contents of the SysFail register to the variable referenced by *sysFailRegPtr*. Bits 0, 4, 8 and 12 of the register indicate whether the SysFail signal of the corresponding board (PC, IBS master and COP) has been activated or not. In the event of a malfunction of an MPM device (e.g. watchdog has initiated a reset), the associated bit in the SysFail register is activated, i.e. set to one. This bit then remains set until the end of the malfunction. The individual bits of the register are assigned as follows to the MPM nodes:
Bit 0 --> host PC
Bit 4 --> IBS master board (MA)
Bit 8 --> coprocessor board (COP)
Bit 12 is not used, as there are only three MPM nodes.

Synopsis: GetSysFailRegister
(boardNumber : USIGN16; SysFailReg : USIGN16Ptr):INT16;

Parameters:

boardNumber	Board number (PC: 1 bis 4, COP: 1)
sysFailRegPtr	Pointer to a variable where the contents of the SysFail register are entered.

Positive acknowledgment:

ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment:

ERR_INVALID_BOARD_NUM (0080_{hex})
Meaning: An invalid board number was specified.
ERR_TSR_NOT_LOADED ($008B_{hex}$)
Meaning: The specified board has not been installed, or the driver for it has not been loaded.

3.4.2.3 SRAM Access Functions

(only for IBS PC CB/COP/I-T and IBS PC CB/486RTX/I-T)

COP_WriteStaticRAM

Task: Writes the specified number of bytes from the given address onwards to the static RAM of the COP. The lowest possible address is 0.

Synopsis: COP_WriteStaticRam
(address:USIGN32;length:USIGN16;Data:Pointer):INT16;

Parameters:

address	Start address in the static RAM
length	Data record length (number of bytes to be written)
data	Pointer to the buffer from which the function is to take the data to be written.

Positive acknowldegment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: ERR_AREA_EXCDED (0096_{hex})

Meaning	The data record to be read is too long. The function can read a maximum of 64 kbytes in one call.
Remedy	Call the function twice to transfer larger data volumes block by block. Increase the start address by 64 kbytes in the second call.
Meaning	The upper boundary of the area has been exceeded. The static RAM has a size of 128 kbytes.
Remedy	Ensure that the total of start address and data record length does not exceed the area boundary.

COP_ReadStaticRAM

Task: Reads the specified number of bytes from the specified address onwards from the static RAM of the COP.

Synopsis: COP_ReadStaticRam (address:USIGN32;length:USIGN16;Data:Pointer):INT16;

Parameters:

address	Start address in the static RAM
length	Data record length (number of bytes to be read)
data	Pointer to the buffer where the function is to store the data to be read.

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: ERR_AREA_EXCDED (0096_{hex})

Meaning	The data record to be written is too long. The function can write a maximum of 64 kbytes in one call.
Remedy	Call the function twice to transfer larger data quantities block by block. Increase the start address by 64 kbytes in the second call.
Meaning	The upper boundary of the area has been exceeded. The static RAM has a size of 128 kbytes.
Remedy	Ensure that the total of start address and data record length does not exceed the area boundary.

3.4.2.4 Watchdog Control Functions

EnableWatchDog()

Task: The function enables the watchdog.

Synopsis: EnableWatchDog(boardNumber : USIGN16):INT16;

Parameter:

	boardNumber	Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})

Meaning	An invalid board number was specified.
Remedy	Enter a valid board number.

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

PC	Trigger interval less than 146 ms, otherwise IBS master board reset
COP	Trigger interval less than 125 ms, otherwise COP and IBS master board reset

TriggerWatchDog()

Task: The function triggers the watchdog.

Synopsis: INT16 FAR TriggerWatchDog(USIGN16 boardNumber)

Parameter:

	boardNumber	Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})

Meaning	An invalid board number was specified.
Remedy	Enter a valid board number.

Comment: This call must be repeated at regular intervals, to ensure that the watchdog does not initiate a reset.

PC:	Trigger interval less than 146 ms, otherwise IBS master board reset
COP:	Trigger interval less than 125 ms, otherwise COP and IBS master board reset

GetWatchDogState()

Task: This function allows you to inquire from your application program whether the watchdog has initiated a reset. If the application program is running on the host, the function inquires automatically the host watchdog state. If the application program is running on the COP, the function inquires automatically the COP watchdog state.

Synopsis: GetWatchDogState(boardNumber : USIGN16):INT16;

Parameter:

boardNumber	Board number (PC: 1 to 4, COP: 1)

Positive aknowledgment: -

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})

Meaning	An invalid board number was specified.
Remedy	Specify a valid board number.

Return value: Coprocessor board watchdog state:
1 The watchdog initiated the last COP warmstart (software reset).
0 The watchdog did not initiate the last COP warmstart (software reset).
Host watchdog state:
1 The host watchdog initiated a reset.
0 The host watchdog did not initiate a reset.

The return values are no longer available after a hardware reset of the controller board or the host.

ClearWatchDog()

Task: The function resets the watchdog state.

Synopsis: INT16 FAR ClearWatchDog (USIGN16 boardNumber)

Parameter:

boardNumber	Board number (PC: 1 to 4, COP: 1)

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: ERR_INVALID_BOARD_NUM (0080_{hex})

Meaning	An invalid board number was specified.
Remedy	Enter a valid board number.

3.4.3 IBS Diagnostic Function

GetIBSDiagnostic();

Task: The GetIBSDiagnostic() function is used to evaluate the IBS master board state and, therefore, the state of the IBS system.

Synopsis: GetIBSDiagnostic(boardNumber : USIGN16; Info : P_IBS_DIAG):INT16;

Parameters:

boardNumber	Board number (PC: 1 to 4, COP: 1)
Info	Pointer to structure with error details

P_IBS_DIAG Structure with error details

Structure elements:

state: The bits of the structure element *state* describe the bus state. Masking (ANDing) the structure element *state* with the following constants allows to evaluate the state of the IBS system:

```
DIAG_IBS_READY     IBS is ready
DIAG_IBS_RUN       IBS has started and is running
DIAG_IBS_SYS_FAIL  A bit was set in the SysFail
                   register (e.g. by a watchdog)
DIAG_IBS_BSA       A bus segment has been disabled
DIAG_IBS_ERROR     IBS master board indicated error
```

errType: The bits of the *errType* structure element describe error conditions in more detail. Masking (ANDing) the structure element *errType* with the following constants allows you to evaluate the error type:

```
DIAG_CNTRL_ERR     Controller error
DIAG_RMT_BUS_ERR   Remote bus error
                   (e.g. defective remote bus cable)
DIAG_LCL_BUS_ERR   Local bus error
                   (e.g. defective local bus cable)
DIAG_MDL_ERR       IBS module error
                   (e.g. interrupted I/O supply
                   voltage, output overload)
```

diagPara: The evaluation of the structure element *diagPara* differs according to its value and to the structure element *errType*:

- If the structural element *errType* indicates a remote bus, local bus or IBS device error and the value of the structure element *diagPara* is in the range from 0 to 255, *diagPara* specifies the number of the bus segment where the error has occurred. Output the bus segment in decimal notation.
- If the structure element *errType* indicates a remote bus, local bus or IBS device error and the value of the structural element *diagPara* is higher than 255, *diagPara* specifies an error number (E01, E02, E04, E05 or E06). See the description of the message *Bus_Error_Information_Indication* ($80C4_{hex}$) in the controller board manual (IBS PC CB UM E).
- If the structure element *errType* indicates a controller error, the structural element *diagPara* specifies a controller error number (see the list of the controller error numbers). Output the controller error number in hexadecimal notation.

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment: ERR_INVLD_BOARD_NUM (0080_{hex})
Meaning: Invalid board number

Negative acknowledgment: ERR_NODE_NOT_READY (0087_{hex})
Meaning: IBS master board is not accessible (e.g. is booting)
Remedy: Wait a moment and then try again

Negative acknowledgment: MPM NOT AVAILABLE (0099_{hex})
Meaning: The MPM is not accessible
Remedy: Reinstall the driver

Evaluate the diagnostic information only when the function was successfully executed (positive acknowledgment ERR_OK (0000_{hex})). On return of a negative acknowledgment **no** valid diagnostic information is available!

Call syntax:

```
GetIBSDiagnostic(
                boardNumber : USIGN16; { controller board number       }
                Info : P_IBS_DIAG)     { pointer to structure
                                         with error details            }
                :INT16;
```

Format of the structure *P_IBS_DIAG*:

```
P_IBS_DIAG = ^T_IBS_DIAG;
T_IBS_DIAG = record
                state    : USIGN16; { state of the bus e.g. Ready, Run }
                errType  : USIGN16; { type of the error, e.g. remote,
                                      local bus or I/O periphery error }
                diagPara : USIGN16; { supplementary information, see the
                                      parameter description on the
                                      previous page                    }
                end;
```

Example: Refer to the next page for a program detail for evaluating the structure element *state* by masking (ANDing) with specified constants:

```
Procedure Diagnose;
  Result    :  INT16;
  IBS_Info  :  T_IBS_DIAG

begin
         Result:=GetIBSDiagnostic(boardNumber, @IBS_Info);

    if Result=ERR_OK then begin
      if (IBS_Info.state AND DIAG_IBS_READY) = DIAG_IBS_READY
         then writeln('IBS Ready');
      if (IBS_Info.state AND DIAG_IBS_RUN) = DIAG_IBS_RUN
         then writeln('IBS Run');
      else
         writeln('IBS Stop!')
    end
end;
```

Section 4

Additions to the Driver Software for Windows®

This section provides information for

- the implementation of the device driver interface and the device drivers for Windows;
- the functions to be used;
- the required include files and libraries.

4 Additions to the Driver Software for Windows

4.1 Structure of the Driver Software on the Host (PC)

The driver software for Microsoft Windows® is provided as a **D**ynamic **L**ink **L**ibrary (DLL) (IBSPCCB.DLL). This DLL incorporates the device driver interface and device drivers for four controller boards.

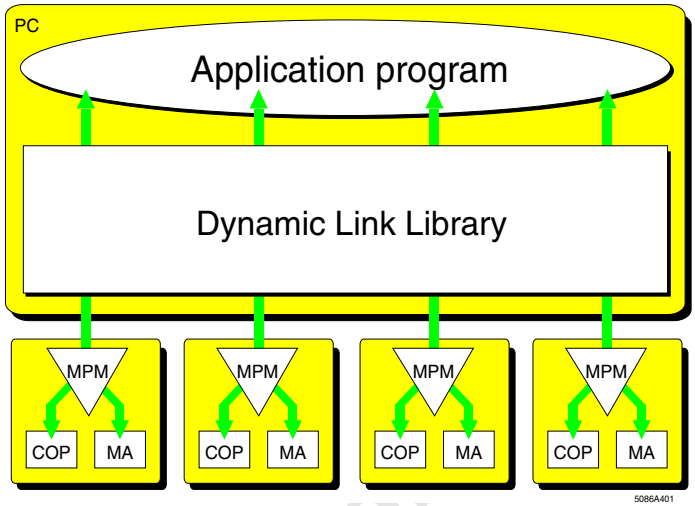

Figure 4-1: IBS driver software under Microsoft Windows®

The *IBSPCCB.DLL* file incorporates the device drivers for four controller boards, which have to be entered and parameterized in the *IBSPCCB.INI* file.

4.2 Notification Mode

The operation in the Notification Mode allows to inform the application program by means of a Windows message of the arrival of a message (e.g. message from the IBS master board) in the MPM. The cyclic call of the function DDI_MXI_RcvMessage() from the application program is in that case not required under Microsoft Windows®.

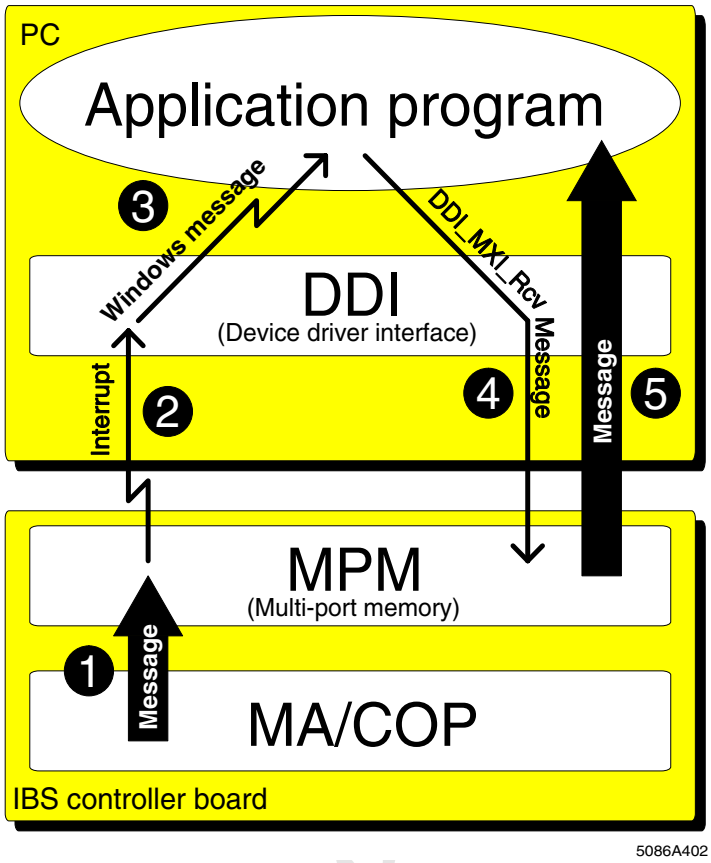

Figure 4-2: Notification Mode under Microsoft Windows®

Operation of the Notification Mode under Microsoft Windows®:

1 The IBS master board (MA) or the coprocessor board (COP) place a message in the multi-port memory (MPM).

2 The arrival of the message in the MPM causes an interrupt, which is evaluated by the mailbox interface of the DDI.

3 When the Notification Mode between the IBS master board or the coprocessor board and the application program is enabled, the DDI generates by means of the *PostMessage* procedure a Windows message informing the application program of the arrival of a message in the MPM.

4 Thereupon, the *DDI_MXI_RcvMessage* is called in the application program.

5 The message is supplied to the application program in a buffer.

The Windows message indicates only that there is a message in the MPM. To read out the message, use the *DDI_MXI_RcvMessage* function.

To obtain a quick reaction it is recommended to assign the node handle value to the parameter *firstParam* (example: *firstParam* = *nodeHd*). When the Windows message is received, this parameter, which corresponds to *wParam,* can be used for reading out the message. When the value for the Windows message (*msg*) is assigned, it is recommended to use the Windows constant WM_USER (example: msg = WM_USER + 255).

The *DDI_SetMsgNotification* function activates the Notification Mode between a Windows window and an MPM user. To deactivate the Notification Mode, call the function *DDI_ClrMsgNotification*. You can activate and deactivate the

Notification Mode for each MPM user separately.

The Notification Mode is automatically terminated when the data channel assigned to the node handle is closed, i.e. the node handle concerned is cleared.

4.3 Library Under Windows

The operation of the controller boards under Microsoft Windows® requires only a **D**ynamic **L**ink **L**ibrary (DLL). This DLL *(IBSPCCB.DLL*) created in the protected mode incorporates the same DDI functions as provided by the driver software for DOS. In addition, two functions for indicating the arrival of messages were implemented in the mailbox interface (Notification Mode).

Using the normal Windows copying procedure, copy the file *IBSPCCB.DLL* to the directory where your application program is located, or to the Windows root directory.

4.4 Include Files for *"C"*

From driver software version 0.9 onwards you only need to incorporate the include file *IBS_WIN.H.* All other required include files (STDTYPES.H, COMPILER.H, IBS_CM.H, DDI_USR.H, DDI_ERR.H, DDI_LIB.H, PC_UTIL.H and DDI_MACR) are called from this include file. However, like in earlier driver software versions, you can also call the required include files separately.

When the include file *IBS_WIN.H* is used, it is not necessary to manually enter compiler switches in the program or in the compiler's command line. The required constant declaration (IBS_WIN_DRV) then takes place within *IBS_WIN.H*.

The include file *DDI_MACR.H* allows the use of the macro functions described in Section 7. The macros are defined in this file.

If you do not want to use IBS_WIN.H but call the required include files separately in the program, insert the instruction *#define IBS_WIN_DRV* before incorporating the include files. This can be done either in the program text or as a compiler option.

Examples:

```
#define IBS_WIN_DRV
#include "stdtypes.h"
#include "ddi_usr.h"
...
```
or:
```
cl /C /Ox /DIBS_WIN_DRV ...
```

4.5 Units for *Pascal*

The only unit to be incorporated in addition to the units required under DOS (*DDI_DRV.PAS* and *DDI_VAR.PAS*), is the unit *TPPCCB.PAS*. This unit is the Pascal interface to the library *IBSPCCB.DLL*. The Notification Mode control functions are also declared in this unit.

4.6 Initialization File Under Windows

The Windows root directory must also contain the IBSPCCB.INI file, which is used for parameterization. Enter the parameters required for controller board initialization (I/O address, MPM address and interrupt number) in the *IBSPCCB.INI* file. The following example shows the entries for the operation of a controller board.

```
[GENERAL]
EnableInitErrorMessage=TRUE

[BOARD1]
BoardInUseFlag=TRUE
IOAddress=120
MPMAddress=D000
IRQ=10

[BOARD2]
BoardInUseFlag=FALSE
IOAddress=120
MPMAddress=D100
IRQ=11

[BOARD3]
BoardInUseFlag=FALSE
IOAddress=120
MPMAddress=D200
IRQ=12

[BOARD4]
BoardInUseFlag=FALSE
IOAddress=120
MPMAddress=D300
IRQ=15
```

Figure 4-3: Examples of entries in the *IBSPCCB.INI* file

Set, for example, the entry *BoardInUseFlag* to *TRUE* for board 1 to ensure that the controller board no. 1 is recognized during the initialization phase. Otherwise, the controller board will be marked as not installed and will not be initialized, even if it does exist.

If you enter an invalid value, the DLL initialization will **not** be aborted.

- In the case of the entry *EnableInitErrorMessage=TRUE*, an error message will be output to a Windows message box when the DLL is loaded (start of the application program).
- In the case of the entry *EnableInitErrorMessage=FALSE*, **no** error message will be output when the DLL is loaded (start of the application program). In this case you can see only from the driver or DDI functions (e.g. when a data channel is opened for the first time) that an initialization error has taken place.

Except for the restriction that the values for *MPMAddress* may only be in the range from $A0000_{hex}$ to $FF000_{hex}$, the values are the same as for the driver software for DOS.

4.7 Functions for Windows

Table 4-1: Overview of the DDI functions for Microsoft Windows®

Function	Task	Page
DDI_DevOpenNode	Opens a data channel to a node	4-8
DDI_DevCloseNode	Closes a data channel to a node	4-8
DDI_MXI_SndMessage	Writes a message to the MPM	4-8
DDI_MXI_RcvMessage	Reads a message from the MPM	4-8
DDI_DTI_ReadData	Reads data from the MPM	4-8
DDI_DTI_WriteData	Writes data to the MPM	4-8
DDI_SetMsgNotification	Activates the Notification Mode for Windows	4-9
DDI_ClrMsgNotification	Deactivates the Notification Mode for Windows	4-11

Table 4-2: Overview of the hardware control functions

Function	Task	Page
COP_WriteStaticRAM	Writes a number of bytes to the SRAM of the COP	*
COP_ReadStaticRAM	Reads a number of bytes from the SRAM of the COP	*
GetDIPSwitch	Reads out the settings of the DIP switch for setting the boot configuration	4-13
GetSysFailRegister	Reads out the contents of the SysFail register	4-13
EnableWatchDog	The use of the watchdog for host monitoring is not recommended. It is not ensured that your program can trigger the watchdog within the required time. One example: Move the frame of a window using your mouse. The program will halt as long as you are holding the frame of the window with your mouse!	—
TriggerWatchDog		—
GetWatchDogState		—
ClearWatchDog		—
GetIBSDiagnostic	Evaluates the IBS master board state	4-13

* These are functions for the coprocessor board. They are defined exclusively for use under the DOS-compatible operating systems (RTXDOS, TDOS) of the coprocessor board and described in Section 2 (C) and Section 3 (Pascal).

4.7.1 Device Driver Interface Functions

The DDI functions for the operation under Microsoft Windows® do not differ basically from those used under DOS. They have the same functionality and use the same parameters. Differences result from the fact that it is possible under Windows to get informed of the arrival of a message in the MPM by means of a Windows message. This feature is effected by the Notification Mode.

Declaration of the DDI functions available for Microsoft Windows®

In contrast to the driver software for DOS, the DDI functions of the driver software for Microsoft Windows® are declared in the DLL *IBSPCCB.DLL* as *FAR PASCAL* to allow the use of this DLL with different programming languages. The functions for Microsoft Windows® are declared as follows:

```
INT16 FAR PASCAL  DDI_DevOpenNode(CHAR FAR *devName, INT16 perm, INT16 FAR *nodeHd)
INT16 FAR PASCAL  DDI_DevCloseNode(INT16 nodeHd)
INT16 FAR PASCAL  DDI_DTI_WriteData(INT16 nodeHd, T_DDI_DTI_ACCESS FAR *dtiAcc)
INT16 FAR PASCAL  DDI_DTI_ReadData(INT16 nodeHd, T_DDI_DTI_ACCESS FAR *dtiAcc)
INT16 FAR PASCAL  DDI_MXI_SndMessage(INT16 nodeHd, T_DDI_MXI_ACCESS FAR *mxiAcc)
INT16 FAR PASCAL  DDI_MXI_RcvMessage(INT16 nodeHd, T_DDI_MXI_ACCESS FAR *mxiAcc)
```

These functions are, with the exception of the declarations, identical with the driver software for DOS and are described in Section 2 (C) and Section 3 (Pascal). The parameters to be transferred correspond to the descriptions in those sections.

4.7.2 Notification Mode Management Functions

Two additional functions are available for activating and deactivating the Notification Mode.

DDI_SetMsgNotification

Task: This function activates the Notification Mode for a data channel.

Prerequisite: The node handle (*nodeHd*) also transferred to the function must belong to an opened data channel of the mailbox interface, otherwise a general DDI error message is output.

Synopsis (C): INT16 DDI_SetMsgNotification
(INT16 nodeHd, T_IBS_WIN_NOTIFY FAR *notifyInfoPtr)

Synopsis (Pascal) :DDI_SetMsgNotification
(NodeHd : INT16;notifyInfoPtr: P_IBS_WIN_NOTIFY):INT16;

Parameters:		
	nodeHd	The node handle is the logical number of a channel that had previously been opened on the DDI.
	*notifyInfoPtr	Pointer to a data structure of the type *T_IBS_WIN_NOTIFY* (see below).

T_IBS_WIN_NOTIFY: Data structure with the elements required for activating the Notification function.

Structure elements:		
	hWnd	The Windows handle specifies to which Windows box the Windows message is to be directed.
	msg	Code which the user has to assign to the message
	FirstParam	First optional parameter
	secondParam	Second optional parameter

Positive acknowledgment: ERR_OK (0000_{hex})
Meaning: The function was executed successfully.

Negative acknowledgment: General DDI error code
Cause: See the description of the DDI error messages

Negative acknowledgment: EROR_NODE_IN_USE ($00B0_{hex}$)
Cause: You attempted to activate the Notification Mode for a node for which the Notification Mode had already been activated.

DDI_SetMsgNotification in the programming language "*C*"

Call syntax:

```
INT16 DDI_SetMsgNotification(
               INT16  nodeHd,                          /* IN: node handle*/
               T_IBS_WIN_NOTIFY FAR *notifyInfoPtr); /* IN: pointer to
                                                          WIN notify
                                                          structure  */
```

Format of the structure *T_IBS_WIN_NOTIFY*:

```
typedef struct {
               HWND hWnd;           /* handle of the message
                                       destination window         */
               UINT msg;            /* message code of the message*/
               WPARAM firstParam;   /* first parameter (wParam)   */
               LPARA MsecondParam;  /* second parameter (lParam)  */
} T_IBS_WIN_NOTIFY;
```

Assignment of the Windows procedure components to the components of the structure:

```
WndProc(
               HWND hWnd;           /* HWND hWnd            */
               WORD Message;        /* UINT msg             */
               WORD wParam;         /* WPARAM firstParam    */
               LONG lParam;)        /* LPARA MsecondParam   */
```

DDI_SetMsgNotification in the programming language *Pascal*

Call syntax:

```
DDI_SetMsgNotification(
               NodeHd : INT16;                    { IN: node handle    }
               notifyInfoPtr : P_IBS_WIN_NOTIFY)  { IN: pointer to
                                                        WIN notify
                                                        structure      }
               :INT16;
```

Format of the structure *T_IBS_WIN_NOTIFY*:

```
type P_IBS_WIN_NOTIFY = ^T_IBS_WIN_NOTIFY;
     T_IBS_WIN_NOTIFY = record
               H_Wnd          : HWND;
               msg            : USIGN16;
               firstParam     : USIGN16;
               secondParam    : USIGN32;
               end;
```

Assignment of the Windows procedure components to the components of the structure:

```
WndProc(
               HWND hWnd;           { HWND hWnd            }
               WORD Message;        { UINT msg             }
               WORD wParam;         { WPARAM firstParam    }
               LONG lParam;)        { LPARA MsecondParam   }
```

DDI_ClrMsgNotification

Task: This function deactivates the Notification Mode for a data channel.

Synopsis (C): INT16 DDI_ClrMsgNotification
(INT16 nodeHd, T_IBS_WIN_NOTIFY FAR *notifyInfoPtr)

Synopsis (Pascal): DDI_ClrMsgNotification
(NodeHd : INT16;notifyInfoPtr: P_IBS_WIN_NOTIFY):INT16;

Parameters:

nodeHd	The node handle is the logical number of a channel previously opened at the DDI.
*notifyInfoPtr:	Pointer to a data structure of the type *T_IBS_WIN_NOTIFY* (see below).

T_IBS_WIN_NOTIFY: Data structure with the elemetns required for deactivating the Notification function.

Structure elements:

hWnd	As a check, enter the Windows box which you opened for this data channel when activting the Notification Mode.
msg	Code which the user has to assign to the message
firstParam	First optional parameter
seciondParam	Second optional parameter

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: General DDI error code

Cause	See the description of the DDI error messages.

Negative acknowledgment: ERR_INVLD_NODE_HD

Cause	You specified a wrong node handle.

Negative acknowledgment: ERR_INVLD_HWND

Cause	You specified a wrong Windows handle.
Remedy	When deactivating the Notification Mode, use the same parameters *(nodeHd, hWnd*) as when the Notification Mode is activated.
Comment	When a wrong node or Windows handle is used, the Notification Mode is not deactivated. The parameters *msg*, *firstParam* and *secondParam* are not checked.

DDI_ClrMsgNotification in the programming language "*C*"

Call syntax:

```
INT16 DDI_ClrMsgNotification(
               INT16  nodeHd,                          /* IN: node handle*/
               T_IBS_WIN_NOTIFY FAR *notifyInfoPtr); /* IN: pointer to
                                                          WIN notify
                                                          structure  */
```

Format of the structure *T_IBS_WIN_NOTIFY*:

```
typedef struct {
               HWND hWnd;          /* handle of the message
                                      destination window         */
               UINT msg;           /* message code of the message*/
               WPARAM firstParam;  /* first parameter (wParam)   */
               LPARA MsecondParam; /* second parameter (lParam)  */
} T_IBS_WIN_NOTIFY;
```

Assignment of the Windows procedure components to the components of the structure:

```
WndProc(
               HWND hWnd;          /* HWND hWnd          */
               WORD Message;       /* UINT msg           */
               WORD wParam;        /* WPARAM firstParam  */
               LONG lParam;)       /* LPARA MsecondParam */
```

DDI_ClrMsgNotification in the programming language *Pascal*

Call syntax:

```
DDI_ClrMsgNotification(
               NodeHd : INT16;                    { IN: node handle }
               notifyInfoPtr : P_IBS_WIN_NOTIFY)  { IN: pointer to
                                                        WIN notify
                                                        structure   }
               :INT16;
```

Format of the structure *T_IBS_WIN_NOTIFY*:

```
type P_IBS_WIN_NOTIFY = ^T_IBS_WIN_NOTIFY;
     T_IBS_WIN_NOTIFY = record
               H_Wnd          : HWND;
               msg            : USIGN16;
               firstParam     : USIGN16;
               secondParam    : USIGN32;
               end;
```

Assignment of the Windows procedure components to the components of the structure:

```
WndProc(
               HWND hWnd;           { HWND hWnd           }
               WORD Message;        { UINT msg            }
               WORD wParam;         { WPARAM firstParam   }
               LONG lParam;)        { LPARA MsecondParam  }
```

4.7.3 Hardware Control Functions

For hardware control, the functions described under DOS are also available, with the exception of the watchdog function.

Declaration of the control functions available for Microsoft Windows®

In contrast to the driver software for DOS, the DDI functions of the driver software for Microsoft Windows® are declared in the DLL *IBSPCCB.DLL* as *FAR PASCAL*, to allow the use of this DLL with different programming languages. The functions for Microsoft Windows® are declared as follows:

```
INT16 FAR PASCAL  GetDIPSwitch(USIGN16 boardNumber, USIGN16 FAR *dataPtr)
INT16 FAR PASCAL  GetSysFailRegister(USIGN16 boardNumber, USIGN16 FAR  *sysFailRegPtr)
INT16 FAR PASCAL  GetIBSDiagnostic(USIGN16 boardNumber, T_IBS_DIAG FAR *infoPtr)
```

These functions are, with the exception of the declaration, identical with the driver software for DOS and are described in Section 2 (C) and Section 3 (Pascal). The parameters to be transferred are in accordance with the descriptions given there.

As under Windows it cannot be ensured, even when it is running without errors, that the watchdog is triggered at sufficiently short intervals (<146ms), the watchdog functions (EnableWatchDog, TriggerWatchDog, ClearWatchDog and GetWatchDogState) were not implemented.

The incorporation of the above functions in a separate program can be carried out via the import list in the DEF file of the application program, or via the supplied import library (IBSPCCB.LIB). The entries required in the DEF file are:

```
IMPORTS
DDI_DevOpenNode=ibspccb.2
DDI_DevCloseNode=ibspccb.3
DDI_DTI_WriteData=ibspccb.4
DDI_DTI_ReadData=ibspccb.5
DDI_MXI_SndMessage=ibspccb.6
DDI_MXI_RcvMessage=ibspccb.7
DDI_SetMsgNotification=ibspccb.8
DDI_ClrMsgNotification=ibspccb.9
GetIBSDiagnostic=ibspccb.10
GetSysFailRegister=ibspccb.11
GetDIPSwitch=ibspccb.12
```

4.8 Use of the Driver Software with C++

The driver software for Windows can also be used with C++. You only need to ensure that the include files are incorporated with *extern "C"* . Otherwise, the application cannot be linked.

Example:

```
extern "C" {
#include  ibs_win.h"
}
```

Section 5

Additions to the Driver Software for OS/2®

This section provides information on

- the implementation of the device driver interface and the device drivers for OS/2;
- the functions to be used.

5 Additions to the Driver Software for OS/2

5.1 Structure of the Driver Software on the Host (PC)

The device driver interface for IBM OS/2® is a **D**ynamic **L**ink **L**ibrary (DLL) (IBSPCCB.DLL). Under OS/2, the device drivers are implemented as OS/2 device drivers.

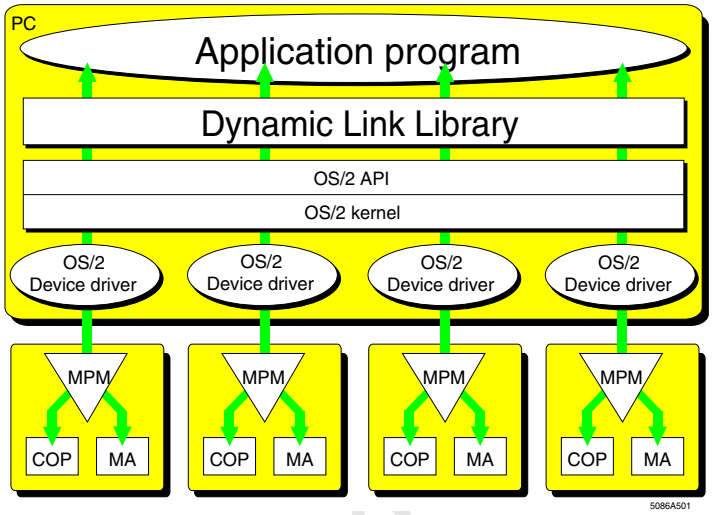

Figure 5-1: IBS driver software for IBM OS/2®

An OS/2 device driver must be installed for each controller board! This requires an entry in the CONFIG.SYS file of your OS/2 system for each. The device driver installation is described in the IBS PC CB UM E manual (Order No. 27 54 75 2), in Section 4 (Installation and Parameterization).

5.2 Notification Mode Under OS/2

Blocked Mode

Under OS/2, the *Blocked Mode* can be used as Notification Mode. The *Blocked Mode* makes it possible to have a thread wait for the arrival of a message (e.g. IBS master board message). While it is waiting, the thread is in the *Sleep* state *(Blocked Mode)*, i.e. no processor time is "wasted". This means for a thread which cyclically calls the *DDI_MXI_RcvMessage* function that, while in the *Block Mode*, it is in the *Sleep* state if there is no message in the MPM.

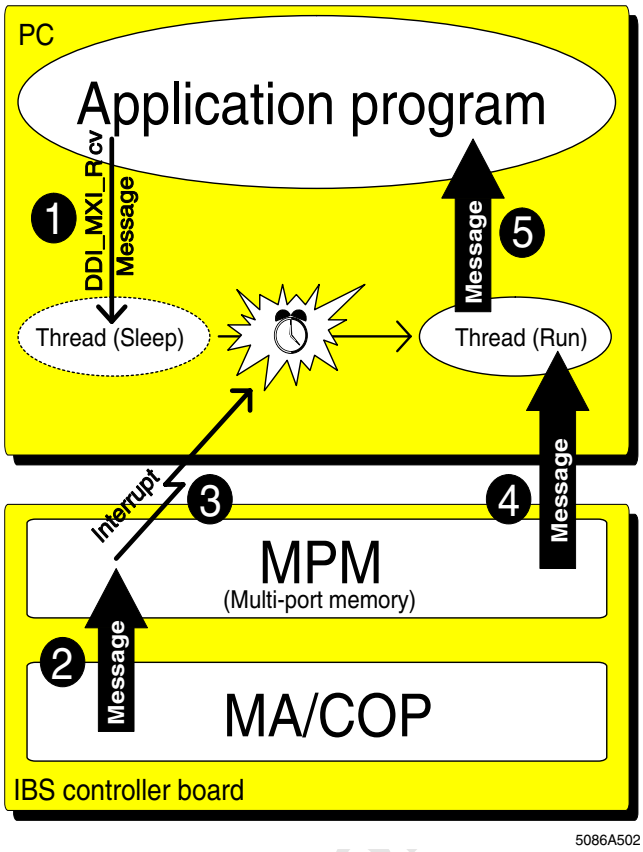

Figure 5-2: Blocked Mode under OS/2®

Mode of operation of the Blocked Mode under OS/2®:

1 The application program starts a thread with the *DDI_MXI_RcvMessage* function. If there is currently no message in the MPM, the thread goes into the *Sleep* state and does not keep the processor busy.

2 The IBS master board (MA) or the coprocessor board (COP) place a message in the multi-port memory (MPM).

3 The arrival of the message in the MPM causes an interrupt which "wakes up" the "sleeping" thread and places it back in the *Run* state.

4 Using the *DDI_MXI_RcvMessage* function, the thread fetches the message from the MPM.

5 The thread returns with the message.

After accepting the message, the application program immediately restarts the thread. Thus, the thread fetches all available messages from the MPM using the *DDI_MXI_RcvMessage* function. As soon as there are no more messages, the thread returns to the *Sleep* state.

The *Blocked Mode* is activated with the *DDI_SetMsgNotification* function, whereas the *DDI_ClrMsgNotification* function deactivates it. You can activate and deactivate the *Blocked Mode* separately for each MPM node.

Should you try to terminate a process while a thread of this process still is in the *Sleep* state, your application program will get hung at this point. To prevent this, first deactivate the *Blocked Mode* for this node using the function *DDI_ClrMsgNotification* (see function description). The thread will return without

a message, but with the error *ERR_NO_MSG*. It will be restarted immediately afterwards. As this start does not take place in the *Blocked Mode*, the thread will not "go to sleep" again when there is no message in the MPM!

Terminate the thread to ensure that it does not keep polling cyclically for arrived messages after the *Blocked Mode* has been deactivated.

5.3 Library and Include Files for OS/2

Running the controller board under IBM OS/2 requires only a **D**ynamic **L**ink **L**ibrary (DLL). This 32-bit DLL *(IBSPCCB.DLL*) declares the same DDI functions as provided by the driver software for DOS. In addition, two functions to indicate the arrival of messages were integrated in the mailbox interface (DDI_SetMsgNotification, DDI_ClrMsgNotification).

You can include the IBSPCCB.DLL functions via the IMPORTS list in the DEF file or via the supplied IMPORT library in the application program. The DEF file entries of an OS/2 program are as follows:

```
IMPORTS
IBSPCCB.DDI_DevOpenNode
IBSPCCB.DDI_DevCloseNode
IBSPCCB.DDI_DTI_WriteData
IBSPCCB.DDI_DTI_ReadData
IBSPCCB.DDI_MXI_SndMessage
IBSPCCB.DDI_MXI_RcvMessage
IBSPCCB.DDI_SetMsgNotification
IBSPCCB.DDI_ClrMsgNotification
IBSPCCB.GetIBSDiagnostic
IBSPCCB.GetDIPSwitch
IBSPCCB.GetSysFailRegister
```

Using the customary OS/2 copy procedure, copy *IBSPCCB.DLL* to the directory where your application program is located, or to the *OS2/DLL* directory.

Include file

To keep the handling of the include files easy, driver software version 0.9 requires only to incorporate include file *IBS_OS2.H*. However, you can also call the individual required files separately.

In addition, when the include file *IBS_OS2.H* is used it is not required to manually enter compiler switches in the program or in the compiler command line. The required constant declaration (IBS_OS2_DRV) then takes place within *IBS_OS2.H*.

Table 5-1: Library and include files

DLL	Include file
IBSPCCB.DLL	IBS_OS2.H (calls up STDTYPES.H, COMPILER.H, IBS_CM.H, DDI_USR.H, DDI_ERR.H, DDI_LIB.H, PC_UTIL.H and DDI_MACR.H)

The include file *DDI_MACR.H* allows the use of the macro functions described in Section 6. The macros are defined in this file.

If you do not want to use IBS_OS2.H but call the required include files individually, insert the instruction *#define IBS_OS2_DRV* before incorporating the include files. This can be done either in the program text or as a compiler option.

Examples:

```
#define IBS_OS2_DRV
#include "stdtypes.h"
#include "ddi_usr.h"
...
```

or

```
icc /C+  /O+ /DIBS_OS2_DRV ...
```

5.4 CONFIG.SYS Under OS/2

The OS/2 device drivers for the controller boards must be loaded when the host is started. To ensure this, enter for each controller board (max. four) an OS/2 device driver in the *CONFIG.SYS* file of your host.

If there are several host controller boards (3 in our example), the examples, for instance, may be as follows:

```
DEVICE=OS2_IBS.DRV
DEVICE=OS2_IBS.DRV BN= 2 IO=120 MPM=D100 IRQ=11
DEVICE=OS2_IBS.DRV BN= 3 IO=120 MPM=D200 IRQ=12
```

As no parameters are specified for the first driver in this example, the following standard values will be used as default:

```
DEVICE=OS2_IBS.DRV BN= 1 IO=120 MPM=D000 IRQ=15
```

The parameter values used are indicated when the driver is loaded.

The individual parameters have the following meanings:

OS2_IBS.DRV:	This is the name of the actual driver. Specify the complete driver if the driver is not in the OS/2 root directory.
BN=1:	The **b**oard **n**umber (BN) specifies for which controller board the driver is to be loaded. The default value is 1, i.e. if the driver for the controller board no.1 (settable with DIP switch) is to be loaded, the parameter BN=1 does not need to be specified. Valid values for the board number are 1, 2, 3 and 4.
IO=120:	This parameter stands for the I/O address under which the controller board can be accessed in the I/O address area of the PC. Set the I/O address also on the DIP switches. If the address set there does not match the specified address, the initialization of the board is aborted, and an error message is output. The default value is 120_{hex}. Refer to the *IBS PC CB UM E* manual for information on alternative I/O addresses.
MPM=D000:	This parameter stands for the address in the memory area of the PC where the controller board is to be found. The controller board occupies an address area of 4 kbytes. Ensure that this area is not already used by other boards. A check does not take place. The default value is $D000_{hex}$. Refer to the *IBS PC CB UM E* manual for information on alternative addresses.
IRQ=15:	Assign a free PC interrupt to each controller board. OS/2 does not allow on the ISA bus that an interrupt is used by more than one controller board. Ensure that the interrupt used has not already been otherwise assigned, or else the initialization will be aborted and an error message (Error at SetIRQ) will be output. The default value for the interrupt is 15. Refer to the *IBS PC CB UM E* manual for information on alternative interrupts.

If you do not enter a parameter, the default value will be used for this parameter.

5.5 Compiler Options

When using the DDI functions and the utility functions, ensure that the DLL *IBSPCCB.DLL* was compiled with the compiler option */Sp1* (byte alignment) after its creation. All compiler options relevant for the user are listed in the following:

/Gm+	Use multithread libraries
/Se	Allow all c Set/2 language extensions except migrations
/Sp1	Byte alignment
/Mp	Use optlink linkage for functions

5.6 Functions for OS/2

Table 5-2: Overview of the DDI functions for OS/2

Function	Task	Page
DDI_DevOpenNode	Opens a data channel to a node	5-9
DDI_DevCloseNode	Closes a data channel to a node	5-9
DDI_MXI_SndMessage	Writes a message to the MPM	5-9
DDI_MXI_RcvMessage	Reads a message from the MPM	5-9
DDI_DTI_ReadData	Reads data from the MPM	5-9
DDI_DTI_WriteData	Writes data to the MPM	5-9
DDI_SetMsgNotification	Activates the Blocked Mode for OS/2	5-10
DDI_ClrMsgNotification	Deactivates the Blocked Mode for OS/2	5-12

Table 5-3: Overview of the hardware control functions

Function	Task	Page
COP_WriteStaticRAM	Writes a number of bytes to the SRAM of the COP	*
COP_ReadStaticRAM	Reads a number of bytes from the SRAM of the COP	*
GetDIPSwitch	Reads out the settings of the DIP switches used for setting the boot configuration	5-13
GetSysFailRegister	Reads out the contents of the SysFail register	5-13
EnableWatchDog	Enables a watchdog	5-13
TriggerWatchDog	Triggers a watchdog	5-13
GetWatchDogState	Reads out the state of a watchdog	5-13
ClearWatchDog	Resets the state of a watchdog	5-13
GetIBSDiagnostic	Evaluates the IBS master board state	5-13

* These are functions for the coprocessor board. They are defined exclusively for use under the DOS-compatible operating systems (RTXDOS, TDOS) of the coprocessor board, and are described in Section 2.

5.6.1 Device Driver Interface Functions

The DDI function for operation under IBM OS/2® are not basically different from those used for the operation under DOS. They have the same functionality and use the same parameters. Differences result from the fact that under OS/2 it is possible to make a thread of the application program wait for the arrival of a message. The Blocked Mode and special functions are used for this purpose.

Declaration of the DDI functions available for IBM OS/2®

The DDI functions of the driver software for OS/2 are declared in the DLL *IBSPCCB.DLL* in the same way as in the driver software for DOS. The only difference is that under OS/2 you do not need to enter memory models (e.g. FAR).

```
INT16   DDI_DevOpenNode(CHAR *devName, INT16 perm, INT16 *nodeHd)
INT16   DDI_DevCloseNode(INT16 nodeHd)
INT16   DDI_DTI_WriteData(INT16 nodeHd, T_DDI_DTI_ACCESS  *dtiAcc)
INT16   DDI_DTI_ReadData(INT16 nodeHd, T_DDI_DTI_ACCESS  *dtiAcc)
INT16   DDI_MXI_SndMessage(INT16 nodeHd, T_DDI_MXI_ACCESS  *mxiAcc)
INT16   DDI_MXI_RcvMessage(INT16 nodeHd, T_DDI_MXI_ACCESS  *mxiAcc)
```

These functions are, with the exception of the declaration, identical with the driver software for DOS and are described in Section 2. The parameters to be transferred conform to the descriptions in Section 2.

Example:

```
void  threadRcvMessage(void *ulp)
{
    T_MXI_ACC  mxiAcc;

    mxiAcc.msgLength = sizeof(rcvBuf);
    mxiAcc.msgBlk = rcvBuffer;

/* Read messages until it is signaled from the outside that          */
/* no more messages are to be fetched.                               */

    while (stopRcvMsg == FALSE)
    {
        mxiAcc.msgLength = sizeof(rcvBuf);
        if ((ret = DDI_MXI_RcvMessage(nodeHd, &mxiAcc)) == ERR_OK)
        {

        /* Evaluate received message at this point                   */
        /* and initiate further processing.                          */
            . . .
        }
        else
        {
            /* Error when receiving the message                      */
                . . .
        }
    }
    _endthread();
}
```

Enter the length of the receive buffer available in the *msgLength* component of the *T_MXI_ACC* structure! The driver checks by means of operating system calls whether the receive buffer memory is valid and belongs to the calling process. If this is not the case, the program is terminated with an *OS/2 General Protection Fault*.

5.6.2 Blocked Mode Management Functions

In addition, there are two functions for activating and deactivating the Blocked Mode:

DDI_SetMsgNotification

Task: The function activates the Blocked Mode for one data channel.

Prerequisite: The node handle (*nodeHd*) also transferred to the function must belong to an opened data channel of the mailbox interface; otherwise a general DDI error message will be output.

Synopsis: DDI_SetMsgNotification(INT16 nodeHd, T_IBS_OS2_NOTIFY *infoPtr):

Parameters:

nodeHd	The node handle is the logical number of a channel previously opened at the DDI.
*infoPtr	Pointer to a data structure of the type *T_IBS_OS2_NOTIFY* (see below).

T_IBS_OS2_NOTIFY: Data structure with the elements required for activating the Blocked Mode

Structural elements:

msgNotifyMode	Initialize the structure element *msgNotifyMode* with the value *IBS_BLOCKED_MODE* to activate the Blocked Mode. Other values are currently not permissible.
processId	The structure element *processId* contains the process identifier of the process activating the Blocked Mode. The process identifier must have the same value for activation and deactivation. An exception is the value zero (00000000_{hex}) for deactivation. In this case the Blocked Mode is deactivated for the respective node without further checks.
timeout	Enter in the structural element *timeout* the time which a thread is to wait for a message (in milliseconds). If a message arrives before this period of time elapses, the thread will return with the message. Otherwise, the thread will be terminated with the error message *ERR_BLOCK_TIMEOUT* ($00CB_{hex}$).

Positive acknowledgment:

ERR_OK (0000_{hex})	
Meaning	The function was executed successfully.

Negative acknowledgment:

General DDI error code:	
Cause	See the description of the DDI error messages

Call syntax:

```
DDI_SetMsgNotification(
                INT16 nodeHd,                       /* IN: node handle     */
                T_IBS_OS2_NOTIFY *InfoPtr);         /* IN: pointer to
                                                       OS2 notify structure*/
```

Format of the data structure *T_IBS_OS2_NOTIFY*:

```
typedef struct {
                USIGN16 msgNotifyMode; /* blocked mode                */
                USIGN32 processId;     /* process identifier          */
                USIGN32 timeout;       /* timeout in milliseconds     */
} T_IBS_OS2_NOTIFY;
```

DDI_ClrMsgNotification

Task: This function deactivates the Blocked Mode for a data channel.

Synopsis: DDI_ClrMsgNotification(INT16 nodeHd, T_IBS_OS2_NOTIFY *infoPtr):

Parameters:

Node handle	This parameter is the logical number of a channel opened at the DDI before.
*infoPtr	Pointer to a data structure of the type *T_IBS_OS2_NOTIFY* (see below).

T_IBS_OS2_NOTIFY: Data structure with the elements required for deactivating the Blocked Mode.

Structural elements:

MsgNotifyMode	Initialize the structure element *msgNotifyMode* with the value *IBS_BLOCKED_MODE* to deactivate the Blocked Mode. Other values are currently not permissible.
ProcessId	The structure element *processId* contains the process identifier of the process deactivating the Blocked Mode. The process identifier must have the same value for activation and deactivation. An exception is the value zero (00000000_{hex}) for deactivation. In this case the Blocked Mode will be deactivated for the respective node without further checks.
Timeout	This parameter is irrelevant for the function *DDI_ClrMsgNotification* and is not evaluated.

Positive acknowledgment: ERR_OK (0000_{hex})

Meaning	The function was executed successfully.

Negative acknowledgment: General DDI error code

Cause	See the description of the DDI error messages

Negative acknowledgment: ERR_INVLD_NODE_HD (0085_{hex})

Cause	You specified an incorrect node handle.

Negative acknowledgment: ERR_INVLD_process identifier ($00C0_{hex}$)

Cause	You specified a wrong process identifier.
Remedy	When deactivating the Blocked Mode, use the same parameters *(nodeHd,* process identifier) as for activating the Blocked Mode.
Comment	When a wrong node handle or process identifier is specified, the Blocked Mode will not be deactivated.

Call syntax:

```
DDI_ClrMsgNotification(
                INT16 nodeHd,                    /* IN: node handle     */
                T_IBS_OS2_NOTIFY *notifyInfoPtr);/* IN: pointer to
                                                    OS2 notify structure*/
```

Format of the data structure *T_IBS_OS2_NOTIFY* :

```
typedef struct {
               USIGN16 msgNotifyMode; /* Blocked Mode                */
               USIGN32 processId;     /* process identifier          */
               USIGN32 timeout;       /* timeout in milliseconds     */
} T_IBS_OS2_NOTIFY;
```

5.6.3 Hardware Control Functions

For hardware control, too, the functions described under DOS are available. The only difference is that no memory models (e.g. FAR) need to be specified under OS/2®.

Declaration of the control functions available for IBM OS/2®

```
INT16 GetIBSDiagnostic(USIGN16 boardNumber, T_IBS_DIAG  *infoPtr)
INT16 GetDIPSwitch(USIGN16 boardNumber, USIGN16  *dataPtr)
INT16 GetSysFailRegister(USIGN16 boardNumber, USIGN16  *sysFailRegPtr)
INT16 EnableWatchDog(USIGN16 boardNumber)
INT16 TriggerWatchDog(USIGN16 boardNumber)
INT16 GetWatchDogState(USIGN16 boardNumber)
INT16 ClearWatchDog (USIGN16 boardNumber)
INT16 ClearWatchDog (USIGN16 boardNumber)
```

With the exception of the declarations, these functions are identical with the driver software for DOS; they are described in Section 2 (C). The parameters to be transferred conform to the descriptions in that section.

For OS/2, the functions for using the watchdogs will only supported from driver version 0.91 onwards.

The inclusion of the above functions in a separate program can take place by means of the import list in the DEF file of the application program or the supplied import library (IBSPCCB.LIB). The entries required in the DEF file are:

```
IMPORTS
DDI_DevOpenNode=ibspccb.2
DDI_DevCloseNode=ibspccb.3
DDI_DTI_WriteData=ibspccb.4
DDI_DTI_ReadData=ibspccb.5
DDI_MXI_SndMessage=ibspccb.6
DDI_MXI_RcvMessage=ibspccb.7
DDI_SetMsgNotification=ibspccb.8
DDI_ClrMsgNotification=ibspccb.9
GetIBSDiagnostic=ibspccb.10
GetSysFailRegister=ibspccb.11
GetDIPSwitch=ibspccb.12
```

5.7 Use of the Driver Software with C++

The driver software for OS/2 can also be used with C++. Only make sure that the include files are included with *extern "C"*. Otherwise the application cannot be linked.

Example:

```
extern "C" {
#include  ibs_OS2.h"
}
```

Section 6

Macros for Programming Support

This section provides information on macros

- which simplify the exchange of data records between the IBS master board (InterBus-S master protocol chip) and the host or coprocessor board (Intel processor).

6 Macros for Programming Support

6.1 Data Conversion Macros

The following macros (for Pascal: macro functions) simplify the transfer of data (commands, messages, process data) between the host or the coprocessor board on the one side and the IBS master board on the other side.

- The InterBus-S Master Protocol Chip (IPMS) of the host controller board uses the Motorola format (68xxx family) when placing its data in the MPM, and expects this format also when reading.
- The host and COP processors process data in the Intel format, which is typical for IBM-compatible PCs.

The Motorola format and the Intel format use opposite orders of numbering of words and bytes within a data field. The macros convert the data between the Motorola format and the Intel format, and write it to the specified buffer; this allows you to set up a process image in the Intel format.

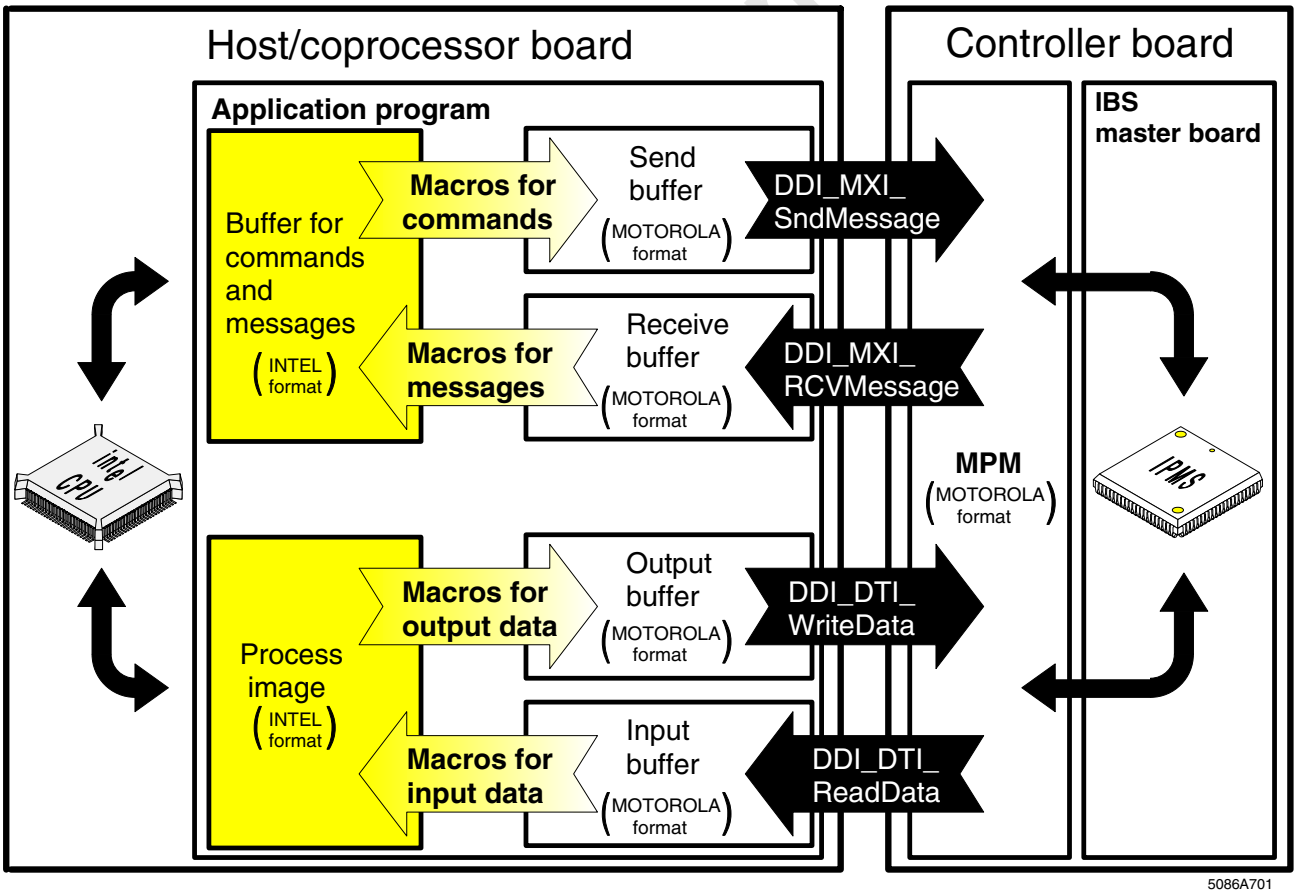

Figure 6-1: Use of the data conversion macros

Table 6-1: Overview of the data conversion macros

Macro	Task	Page
IB_SetCmdCode	Enters the command code (16 bits) in the specified send buffer	6-5
IB_SetParaCnt	Enters the parameter count (16 bits) in the specified send buffer	6-5
IB_SetParaN	Enters a parameter (16 bits) in the specified send buffer	6-6
IB_SetParaNHiByte	Enters the high byte (bits 8 to 15) of a parameter in the specified send buffer	6-6
IB_SetParaNLoByte	Enters the low byte (bits 0 to 7) of a parameter in the specified send buffer	6-8
IB_SetBytePtrHiByte	Returns the address of a parameter entry, starting at the high byte (bits 8 to 15)	6-6
IB_SetBytePtrLoByte	Returns the address of a parameter entry, starting at the low byte (bits 0 to 7)	6-6
IB_GetMsgCode	Reads a message code (16 bits) from the specified receive buffer	6-7
IB_GetParaCnt	Reads the parameter count (16 bits) from the specified receive buffer	6-7
IB_GetParaN	Reads a parameter (16 bits) from the specified receive buffer	6-7
IB_GetParaNHiByte	Reads the high byte (bits 8 to 15) of a parameter from the specified receive buffer	6-7
IB_GetParaNLoByte	Reads the low byte (bits 0 to 7) of a parameter from the specified receive buffer	6-8
IB_GetBytePtrHiByte	Returns the address of a parameter address, starting at the high byte (bits 8 to 15) of a parameter entry	6-8
IB_GetBytePtrLoByte	Returns the address of a parameter entry, starting at the low byte (bits 0 to 7) of a parameter entry	6-8
IB_PD_GetLongDataN	Reads at the specified position a long-word (32 bits) from the input buffer	6-8
IB_PD_GetDataN	Reads at the specified position a word (16 bits) from the input buffer	6-9
IB_PD_GetDataNHiByte	Reads the high byte (bits 8 to 15) of a word from the input buffer	6-9
IB_PD_GetDataNLoByte	Reads the low byte (bits 0 to 7) of a word from the input buffer	6-9
IB_PD_GetBytePtrHiByte	Returns the address of a word, starting at the high byte (bits 8 to 15).	6-9
IB_PD_GetBytePtrLoByte	Returns the address of a word, starting at the low byte (bits 0 to 7)	6-9

Table 6-1: Overview of the data conversion macros

Macro	Task	Page
IB_PD_SetLongDataN	Writes a long-word (32 bits) to the output buffer	6-10
IB_PD_SetDataN	Writes a word (16 bits) to the output buffer	6-10
IB_PD_SetDataNHiByte	Writes the high byte (bits 8 to 15) of a word to the output buffer	6-10
IB_PD_SetDataNLoByte	Writes the low byte (bits 0 to 7) of a word to the output buffer	6-10
IB_PD_SetBytePtrHiByte	Returns the address of a word, starting at the high byte (bits 8 to 15).	6-11
IB_PD_SetBytePtrLoByte	Returns the address of a word, starting at the low byte (bits 0 to 7).	6-11

The macros are so defined in the device driver interfaces for the various operating systems and compilers that they can be universally used.

The include files and libraries or units required for the use of the macros are described in Sections 2, 3, 4, and 5.

6.1.1 Macros for Converting the Data Block of a Command

IB_SetCmdCode(n, m)

Task: This macro converts a command code (16 bits) into the Motorola format and enters it in the send buffer.

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

IB_SetParaCnt(n, m)

Task: This macro converts the parameter count (16 bits) to the Motorola format and enters it in the specified send buffer. The call is required only for commands with parameters. The parameter count specifies the number of subsequent parameters in words.

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

IB_SetParaN(n, m, o)

Task: This macro converts a parameter (16 bits) into the Motorola format and enters it in the send buffer. The call is only required for commands with parameters.

Parameters:
n(USIGN8 FAR *): Pointer to the send 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 (bits 8 to 15) of a parameter into the Motorola format and enters it in the specified send buffer (see also IB_SetParaN).

Parameters:
n(USIGN8 FAR *): Pointer to the send 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 (bits 0 to 7) of a parameter into the Motorola format and enters it in the specified buffer (see also IB_SetParaN).

Parameters:
n(USIGN8 FAR *): Pointer to the send 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 at the high byte (bits 8 to 15). The data type of the address is *USIGN8 FAR **.

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

Return value:
(USIGN8 FAR *): Address of the high byte of the parameter in the send buffer.

IB_SetBytePtrLoByte(n, m)

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

Parameters:
n(USIGN8 FAR *): Pointer to send buffer
m(USIGN16): Parameter no. (word no.)

Return value:
(USIGN8 FAR *): Address of the low byte of the parameter in the send buffer

6.1.2 Macros for Converting the Data Block of a Message

IB_GetMsgCode(n)

Task:	This macro reads a message code (16 bits) from a receive buffer and converts it into the Intel format.	
Parameter:	n(USIGN8 FAR *):	Pointer to the receive buffer
return value:	(USIGN16):	Message code

IB_GetParaCnt(n)

Task:	This macro reads the parameter count (16 bits) from the data block of the message and converts it into the Intel format. The parameter count specifies the number of subsequent parameters in words.	
Parameter:	n(USIGN8 FAR *):	Pointer to the receive buffer
Return value:	(USIGN16):	Parameter count
Comment:	Read out the parameter count only for the messages which really have parameters.	

IB_GetParaN(n, m)

Task:	This macro reads a parameter value (16 bits) from the data block of the message and converts it into the Intel format.	
Parameters:	n(USIGN8 FAR *): m(USIGN16):	Pointer to the receive buffer Parameter no. (word no.)
Return value:	(USIGN16):	Parameter value
Comment:	Read out parameters only for the messages which really have parameters.	

IB_GetParaNHiByte(n, m)

Task:	This macro reads the high byte (bits 8 to 15) of a parameter from a receive buffer and converts it into the Intel format.	
Parameters:	n(USIGN8 FAR *): m(USIGN16):	Pointer to the receive buffer Parameter no. (word no.)
Return value:	(USIGN8):	Parameter value (byte)
Comment:	Read out parameters only for the messages which really have parameters.	

IB_GetParaNLoByte(n, m)

Task: This macro reads the low byte (0 to 7) of a parameter from the specified receive buffer and converts it into the Intel format.

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

Return value: (USIGN8): Parameter value (byte)

Comment: Read out parameters only for the messages which really have parameters.

IB_GetBytePtrHiByte(n, m)

Task: This macro returns the address of a parameter entry, starting at the high byte (bits 8 to 15).

The data type of the address is *USIGN8 FAR **.

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 at the low byte (bits 0 to 7).

The data type of the address is *USIGN8 FAR *.*

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.

6.1.3 Macros for Converting Input Data

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

IB_PD_GetLongDataN(n, m, o)

Task: This macro reads at the specified position a long-word from the input buffer and converts it into the Intel format. The word index in the input buffer is used for the position. Therefore, the macro reads the long-word by reading two words from the specified address onwards.

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

IB_PD_GetDataN(n, m)

Task: This macro reads at the specified position a word (16 bits) from the input buffer, and converts it into the Intel format.

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

Return value: (USIGN8): Process data item (16 bits)

IB_PD_GetDataNHiByte(n, m)

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

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

Return value: (USIGN8): Process data item (8 bits)

IB_PD_GetDataNLoByte(n, m)

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

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

Return value: (USIGN8): Process data item (8 bits)

IB_PD_GetBytePtrHiByte(n, m)

Task: This macro returns the address of a word, starting at the high byte.

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 at the low byte (bits 0 to 7).

Parameter:
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.

6.1.4 Macros for Converting Output Data

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

IB_PD_SetLongDataN(n, m, o)

Task: This macro converts a long-word (32 bits) into the Motorola format and writes it at the specified position to the output buffer. The word index in the output buffer is used for the position. Therefore, the macro writes the long-word by writing two words from the specified address onwards.

Parameters :

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

IB_PD_SetDataN(n, m, o)

Task: This macro converts a word (16 bits) into the Motorola format and writes it at the specified position into the output buffer.

Parameters:

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

IB_PD_SetDataNHiByte(n, m, o)

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

Parameters:

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

IB_PD_SetDataNLoByte(n, m, o)

Task: This macro converts the low byte (bits 0 to 7) of a word into the Motorola format and writes it at the specified position into the output buffer.

Parameters:

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

IB_PD_SetBytePtrHiByte(n, m)

Task:	This macro returns the address of a word, starting at the high byte (bits 8 to 15).	
Parameter:	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 at the low byte (bits 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 7

Driver Software Diagnostics

This section provides information on the driver software messages and error messages.

- Meaning
- Causes
- Remedy

7 Driver Software Diagnostics

Table 7-1: Overview of the DDI error messages

Code	Error message	Cause	Page
0000_{hex}	ERR_OK	The function was executed successfully	7-4
0080_{hex}	ERR_INVLD_BOARD_NUM	Invalid board number	7-4
0081_{hex}	ERR_INVLD_IO_ADDR	Invalid I/O address	7-4
0082_{hex}	ERR_INVLD_MPM_ADDR	Invalid address for the MPM window	7-5
0083_{hex}	ERR_INVLD_INTR_NUM	Illegal interrupt	7-5
0084_{hex}	ERR_INVLD_CARD_CODE	Incorrect board code	7-5
0085_{hex}	ERR_INVLD_NODE_HD	Invalid node handle specified	7-6
0086_{hex}	ERR_INVLD_NODE_STATE	Node handle of an already closed data channel specified	7-6
0087_{hex}	ERR_NODE_NOT_READY	Selected node not ready	7-6
0088_{hex}	ERR_WRONG_DEV_TYP	Wrong node handle	7-6
0089_{hex}	ERR_DEV_NOT_READY	IBS master board not yet ready	7-6
$008A_{hex}$	ERR_INVLD_PERM	Invalid channel access type	7-6
$008B_{hex}$	ERR_TSR_NOT_LOADED	Device driver not loaded	7-10
$008C_{hex}$	ERR_INVLD_CMD	Utility functions not supported by driver version 0.9	7-7
$008D_{hex}$	ERR_INVLD_PARAM		7-7
0090_{hex}	ERR_NODE_NOT_PRES	Node does not exist	7-7
0091_{hex}	ERR_INVLD_DEV_NAME	Unknown device name used	7-7
0092_{hex}	ERR_NO_MORE_HNDL	Device driver out of resources	7-7
0096_{hex}	ERR_AREA_EXCDED	The access exceeds the boundary of the selected data area	7-10
0097_{hex}	ERR_INVLD_DATA_CONS	Specified data consistency is invalid	7-10
0099_{hex}	ERR_MPM_NOT_AVALBL	Access to the MPM not possible	7-6
$009A_{hex}$	ERR_MSG_TO_LONG	Message or command contains too many parameters	7-8
$009B_{hex}$	ERR_NO_MSG	No message available	7-8
$009C_{hex}$	ERR_NO_MORE_MAILBOX	No free mailbox of the required size	7-8
$009D_{hex}$	ERR_SVR_IN_USE	Send vector register in use	7-8
$009E_{hex}$	ERR_SVR_TIMEOUT	Invalid node called	7-8
$009F_{hex}$	ERR_AVR_TIMEOUT	Invalid node called	7-9
$00A0_{hex}$	ERR_COP_USES_MA	IBS master bord control not enabled for PC	7-9
$00A1_{hex}$	ERR_PC_USES_MA	IBS master board control not enabled for the COP	7-9
$00B0_{hex}$	ERR_NODE_IN_USE	Notification Mode activated twice for one node (Windows)	7-11
$00C0_{hex}$	ERR_INVLD_Process-Identifier	Invalid process identifier specified	7-11
$00C1_{hex}$	ERR_BLK_MODE_IS_ENBLD	Blocked Mode already enabled	7-12

Table 7-1: Overview of the DDI error messages

Code	Error message	Cause	Page
$00C2_{hex}$	ERR_THREAD_IS_WAITING	Another thread is already using the Notification Mode for this node	7-12
$00C9_{hex}$	ERR_INVLD_MEMORY	Invalid receive buffer	7-12
$00CA_{hex}$	ERR_INVLD_NOTIFY_MODE	Invalid Notification Mode	7-12
$00CB_{hex}$	ERR_BLOCK_TIMEOUT	Waiting time for a message exceeded	7-12
$00D1_{hex}$	ERR_INVLD_HWND	Invalid Windows handle	7-11
$00D2_{hex}$	ERR_BOARD_NOT_PRES	Board not entered in *IBSPCCB.INI*	7-11
$00D3_{hex}$	ERR_INVLD_INI_PARAM	Invalid parameter in *IBSPCCB.INI*	7-11

7.1 DDI Messages

ERR_OK 0000_{hex}

Meaning: The driver software generates this message as a positive acknowledgment after a function has been executed successfully.

Cause: No error occurred when this function was executed. If a function was not executed successfully, the driver software causes one of the error messages below.

7.2 DDI Error Messages

If the device driver interface generates one of the following error messages as a negative acknowledgment, the function called before could not be executed successfully.

7.2.1 Controller Board Initialization Error Messages

ERR_INVLD_BOARD_NUM 0080_{hex}

Cause: An invalid board number was used.

Remedy: Enter a valid board number. Valid board numbers are 1, 2, 3, and 4.

ERR_INVLD_IO_ADDR0081_{hex}

Cause: The I/O address entered for the controller board is invalid.

Remedy: Enter one of the valid I/O addresses. The following I/O addresses are valid: 100_{hex}, 120_{hex}, 140_{hex}, 160_{hex}, 180_{hex}, $1A0_{hex}$, 200_{hex}, 220_{hex}, 240_{hex}, 280_{hex}, $2A0_{hex}$, 300_{hex}, 320_{hex}, 340_{hex}, 380_{hex}, $3A0_{hex}$

(see also Section 4.1 of the IBS PC CB UM E manual)

ERR_INVLD_MPM_ADDR 0082$_{hex}$

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

Remedy: Use an MPM address inside the range supported by the controller board (C0000$_{hex}$ to FF000$_{hex}$).

The controller board occupies an address area of 4 kbytes from this base address onwards. Ensure that this area is not already used by other boards. An automatic check does **not** take place. As in practice this memory area is already in use to a great extent (BIOS etc.), the address range available is as a rule limited to parts of the address segments D and E (addresses from D0000$_{hex}$ to EFFFF$_{hex}$). The default value is D0000$_{hex}$ (see Section 4.6 in the IBS PC CB UM E manual).

ERR_INVLD_INTR_NUM0083$_{hex}$

Cause: The specified interrupt is invalid.

Remedy: Valid interrupts are IRQ3, IRQ5, IRQ7, IRQ9, IRQ10, IRQ11, IRQ12, and IRQ15.

If several controller boards are used in one host, every installed controller board must use a different interrupt. On most standard PCs the interrupts IRQ10, IRQ11, IRQ12 and IRQ15 are not assigned and, therefore, can be used for the device drivers. The other interrupts are often used by standard PC components (serial ports COM1 and COM2, network adapters, etc.), and should not be used for the host controller boards.

ERR_INVLD_CARD_CODE 0084$_{hex}$

Cause: The status register of the controller board contains an unknown board code. (The driver for IBS PC CB/.../I-T expects the board code 2D$_{hex}$)

Remedy: Have the controller board checked.

7.2.2 General Error Messages

These error messages may occur when any of the DDI functions is called.

ERR_MPM_NOT_AVALBL0099$_{hex}$

Cause: The MPM cannot be accessed. A reset (reset key on the board) may have been initiated on the controller board.

Remedy: De-install the driver and restart it.

ERR_INVLD_NODE_HD0085$_{hex}$

Cause: An invalid node handle was used when the function was called.

Remedy: Use the permissible node handle of a successfully opened data channel.

ERR_INVLD_NODE_STATE0086$_{hex}$

Cause: An invalid node handle was used when the function was called. The invalid node handle was the handle of an already closed data channel.

Remedy: Open the closed data channel or use a data channel that had already been closed.

ERR_NODE_NOT_READY 0087$_{hex}$

Cause: The node with which you want to work has not yet reported "ready", i.e. the node-ready bit in the MPM status register has not been set. The cause may be, for example, a hardware fault.

Remedy:
- Check whether the controller board startup has been completed
- Reset of the PC

ERR_WRONG_DEV_TYP 0088$_{hex}$

Cause: Wrong node handle. It was attempted to access the mailbox interface with a node handle for the data interface.

ERR_DEV_NOT_READY 0089$_{hex}$

Cause: The IBS master board was accessed although it was not ready (Ready LED).

Remedy: Following a reset of the IBS master board, inquire the ready bit in the status/ control register with the *GetIBSDiagnostic()* function. Only when this bit is set is the IBS master board initialized and may be accessed.

ERR_INVLD_PERM008A$_{hex}$

Cause: An attempt was made to execute a function on a channel without previously specifying the appropriate access rights when opening the channel. This error occurs, when, for example, writing to the data interface is to take place, but only read rights (DDI_READ constant) were specified when the channel was opened.

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

ERR_INVLD_CMD008C$_{hex}$

Cause: This error message is issued when certain new help functions of the new DDI_TSR.LIB are used with an earlier driver (version < 0.9).

Remedy: Use a newer driver (version ≥ 0.9).

ERR_INVLD_PARAM008D$_{hex}$

Cause: This error message is output when certain new help functions of the new DDI_TSR.LIB are used with an earlier driver (version < 0.9).

Remedy: Use a newer driver (version ≥ 0.9).

7.2.3 Error Messages when Opening a Data Channel

ERR_NODE_NOT_PRES0090$_{hex}$

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

Remedy: Select the right node.
Possible nodes:
IBS PC CB: Node 1 = IBS master
IBS PC CB/COP: Node 0 = PC, node 1 = IBS master, node 2 = COP

ERR_INVLD_DEV_NAME0091$_{hex}$

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

Remedy: Select a correct device name according to Tables 5-2 to 5-5.

ERR_NO_MORE_HNDL0092$_{hex}$

Cause: The device driver is out of resources. No further data channels can be opened. If you terminate a program without closing the data channels used, they will remain open. The next program start will open further data channels. After several program starts, the maximum number of data channels that may be simultaneously open will finally be reached, and no further channel will be available.

Remedy: Close a data channel that is not required, or install the device driver anew. When a program has been terminated, always close all data channels used by the program.

7.2.4 Message/Command Transfer Error Messages

ERR_MSG_TO_LONG009A$_{hex}$

Cause: If the error message is output when a command is sent, the length of the command exceeds the maximum permissible number of parameters.

Remedy: Reduce the number of parameters.

Cause: If the error message is output on reception of a message, the message length exceeds the length of the specified receive buffer.

Remedy: Extend the receive buffer length.

ERR_NO_MSG009B$_{hex}$

Cause: The message is output when an attempt is made to fetch a message with the *DDI_MXI_RCV_MESSAGE* message, but there is no message from the node specified by the node handle.

ERR_NO_MORE_MAILBOX009C$_{hex}$

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

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

Cause: No free mailbox of the requested size is available. Observe the maximum mailbox size that can be used (1020 bytes).

Remedy: Select a smaller mailbox or wait until there is a free mailbox of the required size.

Cause: You attempted to access the coprocessor board although it is faulty.

Remedy: Please consult Phoenix Contact.

ERR_SVR_IN_USE009D$_{hex}$

Cause: The send vector register for the node is in use.

Remedy: Access the register once more, or wait until the register is free again.

ERR_SVR_TIMEOUT009E$_{hex}$

Desription: When a message placed in the MPM by the IBS master board is not fetched by the accessed MPM node, this node does not reset the acknowledge bit set by the IBS master board; i.e. the accessed MPM node does not indicate *Message recognized*. After a certain time (timeout), the IBS master board will generate the error message *ERR_SVR_TIMEOUT* . If this error message occurs several times in succession, this indicates that the accessed node is no longer ready to accept the message.

Cause: Call of an invalid node, e.g.:

You attempted to access the coprocessor board (COP), which, however, is faulty.

Remedy: Please consult Phoenix Contact.

ERR_AVR_TIMEOUT009F$_{hex}$

Description: When a message is read, an acknowledge bit is set to indicate to the communication partner that a message was processed and the mailbox is free again. The communication partner must reset this bit to indicate that it has recognized that the mailbox is free again. If the reset does not take place within a specified time, this error message is generated.

Cause: Call of an invalid node, e.g.:

You attempted to access the coprocessor board (COP), which is defective or does not exist.

Remedy: Please consult Phoenix Contact.

ERR_COP_USES_MA 00A0$_{hex}$

Cause: The host PC attempted to send a command to the IBS master board although, according to the position of the boot configuration switch, the IBS master board was controlled by the coprocessor board.

Remedy: Take care that the host PC does not send commands to the IBS master board when the IBS master board is controlled by the coprocessor board. Check the setting of the boot configuration switch (see Section 4 in the IBS PC CB UM E manual).

ERR_PC_USES_MA 00A100A1$_{hex}$

Cause The coprocessor board attempted to send a command to the IBS master board although, according to the setting of the boot configuration switch, the IBS master board was controlled by the host PC.

Remedy: Take care that the coprocessor board does not send commands to the IBS master board when the IBS master board is controlled by the host PC. Check the setting of the boot configuration switch (see Section 4 in the IBS PC CB UM E manual).

7.2.5 Process Data Transfer Error Messages

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

ERR_AREA_EXCDED0096$_{hex}$

Meaning: The access exceeds the boundary of the selected data area.

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

Remedy: Read or write only data records with a maximum size of 4 kbytes.

Cause: The upper area boundary (4 kbytes above the beginning of the node area) has been exceeded.

Remedy: Ensure that the total of the address offset, relative address and data length to be read does not exceed the upper area boundary. The node and data areas are described in Section 5 of the manual *IBS PC CB UM E* under *Segmentation of the MPM*.

ERR_INVLD_DATA_CONS0097$_{hex}$

Cause: An invalid value (1, 2, 4 or 6 bytes) was specified for the data consistency.

Remedy: Define a valid data consistency by specifying one of the following constants:
DTI_DATA_BYTE : Byte data consistency (1 byte)
DTI_DATA_WORD : Word data consistency(2 bytes)
DTI_DATA_LWORD : Long-word data consistency (4 bytes)
DTI_DATA_48 Bit : 48-bit data consistency (6 bytes)

7.2.6 Error Messages Under DOS

ERR_TSR_NOT_LOADED008B$_{hex}$

Cause: It was attempted to call a device driver function although the device driver had not been loaded.

Remedy: Load the TSR program *IBSPCCB.EXE* on the host, or the TSR program *IBSCOP.EXE* on the coprocessor board.

7.2.7 Error Messages Under Microsoft Windows®

ERR_NODE_IN_USE 00B0$_{hex}$

Cause: You attempted under Windows to activate the Notification Mode for a node for which the Notification Mode had already been enabled.

ERR_INVLD_HWND00D1$_{hex}$

Cause: You used an invalid Windows handle (hWnd).

Cause: The Windows handle used is not identical with the one used on enabling the Notification Mode.

Remedy: Use the correct Windows handle.

Comment: When a wrong Windows or node handle is specified, the notification mode is not disabled.

ERR_BOARD_NOT_PRES00D2$_{hex}$

Cause: When opening or closing a data channel (function *DDI_DevOpenNode* or *DDI_DevCloseNode*) with the parameter *device name*, an invalid board number was selected. According to the entry *BoardInUseFlag=False* in the file *IBSPCCB.INI* there is no controller board with this board number.

Remedy: Check whether there is a controller board with the board number which you accessed. If this is the case, correct the entry to *BoardInUseFlag=True* in the *IBSPCCB.INI* file.

ERR_INVLD_INI_PARAM00D3$_{hex}$

Cause: A parameter in the IBSPCCB.INI file is not valid.

Remedy: Check the entries in the IBSPCCB.INI file.

7.2.8 Error Messages Under OS/2

ERR_INVLD_PID00C0$_{hex}$

Cause: You specified a wrong process identifier.

Remedy: When disabling the Notification Mode, use the same parameters (nodeHd, *processId*) as for enabling the Notification Mode.

ERR_BLK_MODE_IS_ENBLD00C1$_{hex}$

Cause: The Blocked Mode has already been enabled for this node.

Remedy: The function *DDI_ClrMsgNotification* must be called before a new process can log on.

ERR_THREAD_IS_WAITING00C2$_{hex}$

Cause: Another thread is already waiting for messages from this node and is in the *sleep* mode. While a thread is waiting for messages from a node, generally no other thread can read from this node.

Remedy: If necessary, terminate the Notification Mode for the thread which has been waiting longer.

ERR_INVLD_MEMORY00C9$_{hex}$

Cause: The receive buffer transferred to the function *DDI_MXI_RcvMessage* is not valid, i.e. it cannot be used for storing a message. Otherwise, OS/2 indicates an integrity violation.

Remedy: Check the memory area provided for the receive buffer.

ERR_INVLD_NOTIFY_MODE00CA$_{hex}$

Cause: An attempt was made to enable an invalid Notification Mode. The driver software versions 0.9 and 0.91 support only the Blocked Mode.

Remedy: Use a Notification Mode supported by your driver software.

ERR_BLOCK_TIMEOUT00CB$_{hex}$

Cause: When enabling the Notification Mode for a thread, you entered in the structure element *timeout* the maximum time which a thread is to wait for a message. As no message was received within this time, the thread was terminated with the error message *ERR_BLOCK_TIMEOUT* (00CB$_{hex}$).

Remedy: Check the application program and the equipment to determine why an expected message has not arrived in time. If necessary, set a longer waiting time.

Appendix A

Document Appendix

A Appendix

A.1 Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

A.2 Tables

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

A.3 Index