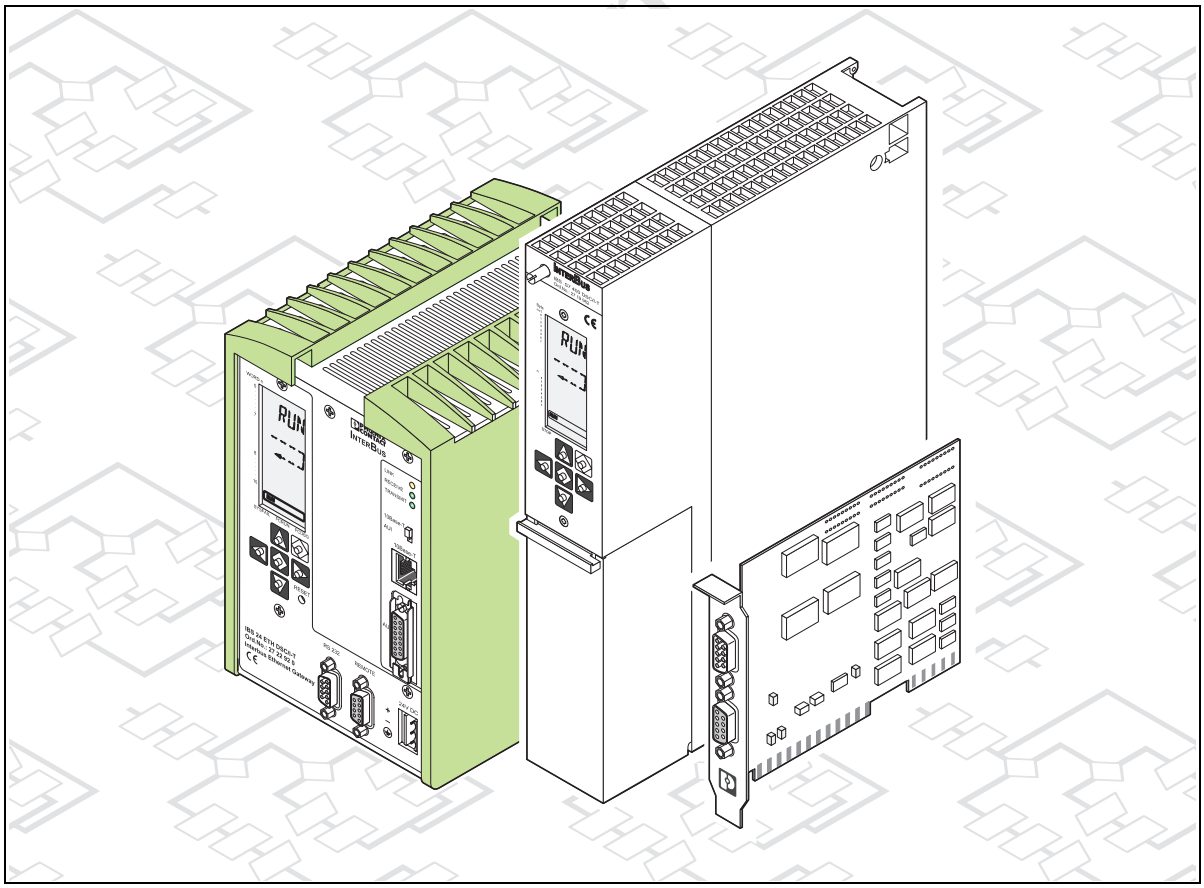
INTERBUS

User Manual

Device Driver Development Kit for Controller Boards in PC Systems With PCI Bus

Designation: IBS PCI DDK UM E

Order No.: 26 98 16 4



INTERBUS

User Manual

Device Driver Development Kit for Controller Boards in PC Systems With PCI Bus

Designation: IBS PCI DDK UM E

Revision: A

Order No.: 26 98 16 4

This user manual is valid for:

IBS PCI SC/I-T	Order No. 27 25 26 0
IBS PCI SC/RI/I-T	Order No. 27 30 08 0
IBS PCI SC/RI-LK	Order No. 27 30 18 7

6189A

PHOENIX CONTACT

Please Observe the Following Notes:

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

Requirements of the User Group

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

Explanation of Symbols Used

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

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

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

We Are Interested in Your Opinion

We are constantly 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 fax 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 the purpose of technical progress.

All product names used are trademarks of the respective organizations.

Internet

Up-to-date information on Phoenix Contact products can be found on the Internet at **www.phoenixcontact.com**.

Terms of Delivery and Payment for Software

Using the DDK sources indicates your acceptance of the following Terms and Conditions. If you do not wish to accept these Terms and Conditions, please send the complete unopened diskette package and all accompanying material (including written material and packing) against reimbursement of the payment made. Phoenix Contact points out that no General Terms of Contract of the customer (hereinafter referred to as the Registered User) can be recognized. The inclusion of such General Terms of Contract is hereby explicitly contradicted.

Terms of Contract

§ 1 Subject matter

The subject matter of this contract is the program stored on the data carrier including the written material also supplied (program description, operating instructions etc.), hereinafter referred to as "Software".

§ 2 Scope of the license

1.In the event of acquisition of a single license, the following paragraph regulates the scope of the license.

1. Phoenix Contact grants the Registered User the single, non-exclusive right (hereinafter referred to as license) to use the Software on one single computer. The Registered User may also install the Software on more than one computer. However, the Software may only be used on one computer at a time. Any use exceeding this scope is not permitted unless agreements have been reached concerning the acquisition of supplementary licenses or a company license (cf. Items 2 and 3).
   The license is a single license if described as such in acknowledgement of order and invoice.
2. In the event of acquisition of supplementary licenses, the following paragraph regulates the scope of the license.
   The Registered User is granted the right to make copies of previously acquired Software (single license) and to use these copies. The number of permitted copies is derived from the number of supplementary licenses acquired by the Registered User. The Registered User undertakes to identify the Software products placed at his disposal and put into circulation by him. To identify the data carriers, the Registered User shall receive a set of appropriate serial numbers. The number of serial numbers corresponds to the number of permitted copies. Each copy may only be used once at a time (cf. § 2 Item 1). Supplementary licenses may only be acquired by Registered Users that are in possession of a basic license for the Software in question. The Registered User is obliged to keep records of the whereabouts of all copies and to allow Phoenix Contact to inspect these records on request.
   The license is a supplementary license if described as such in acknowledgement of order and invoice.
3. In the event of acquisition of a company license, the following paragraph regulates the scope of the license.
   Registered Users that have acquired a company license are granted the right to make any number of copies of the Software placed at their disposal in their own company, and to use it on any number of computers at the same time.
   The license is a company license if described as such in acknowledgement of order and invoice. Use is only permitted in one location.

§ 3 Copyright

1. The Software is the property of Phoenix Contact. The acquisition of rights to the Software itself going beyond the license regulated in this contract is not associated with this. In particular, Phoenix Contact reserves all rights to copying, publication, processing and exploitation.
   The Registered User recognizes that he acquires no rights whatsoever to the Software products described going beyond this agreement, in particular that he acquires no commercial rights to source code.
2. The Software is legally protected against copying. The Registered User may make up to three backup copies for archiving purposes. No alphanumerical identification marks, brand names, trademarks or copyright notices may be removed; these must be transferred to the copies. Neither manuals nor written materials may be copied, unless Phoenix Contact has explicitly authorized the Registered User to do so within the framework of a supplementary or company license. With the exception of the provisions stated in the following paragraph, the Software may be neither loaned, hired nor otherwise passed on to third parties.

3. The rights ensuing from this licensing agreement may be transferred to a third party on condition that the terms and conditions of this licensing agreement are explicitly recognized in writing and that the Registered User hands over the complete program package including all diskettes (and including any backup copies made) and written materials, and that the Registered User completely deletes the program version from all hard disks in his computer. The transfer must comprise the current version (update) and all previous versions. Phoenix Contact must be notified of the transfer of the licensing rights, with the name and address of the recipient. A copy of the declaration of acceptance must be enclosed with the notification. Insofar as no other written agreement has been reached, this obligation also extends to subsidiary companies, other locations, other places of business and other companies of the Registered User.
4. The Registered User must not make any information of a technical or commercial nature obtained from Phoenix Contact, hereinafter referred to as "Information", or any development results accessible to third parties.
   Documentation supplied by Phoenix Contact, as well as the Software and the products derived from said Software may only be passed on within the framework of the above licensing agreement.
   The confidentiality obligation also applies after termination of the contract. However, it does not apply to information which is generally known or which on receipt could be proved to be part of the internal state of technology of the Registered User or which became incorporated later in this internal state of technology, irrespective of the present contract.

§ 4 Decompilation and modifications to the program

1. The decompilation of the program code provided into other code forms as well as other means of retracing the various programming stages of the Software (reverse engineering), including a program modification, are permitted for the Registered User's own use, in particular for purposes of fault correction, for establishing the interoperability and compatibility or expansion of the range of functions. Within the meaning of this provision, own use includes in particular private use by the Registered User. Own use, however, also includes use for professional or profit-making purposes, insofar as this is restricted to the Registered User's or his employees' own use and is not commercially exploited externally in any way.
2. The appropriate actions may only be carried out by commercial third parties in potential competition with Phoenix Contact if Phoenix Contact does not wish to carry out the required program modifications against reasonable compensation. Phoenix Contact must be given an adequate period of time to check the acquisition of the order by the third party, and must be informed of the name of the third party.
3. Insofar as the stated actions are carried out for commercial reasons, they are only permitted if they are essential for the creation, maintenance or proper functioning of an independently created interoperable program and the necessary information has not yet been published and is not otherwise accessible.
4. Under no circumstances may copyright notices, serial numbers or other characteristics for identifying the program be removed or modified.

§ 5 Guarantee

1. This Software has been produced with the greatest possible care. It is, however, known to the Registered User that with the present state of technology it is not possible to produce Software in such a way that it works without errors in all applications and combinations.
2. The guarantee period is six months, beginning with the delivery of the Software.
3. Phoenix Contact guarantees that at the time of delivery, the material of the data carriers on which the Software is stored and the documentation supplied are free from faults. Should the data carriers or the documentation supplied be faulty, the Registered User can demand a replacement delivery during the agreed guarantee period as stated above, on condition that he returns the defective material, including any backup copies made, and the written material.
4. If during the use of the Software by the Registered User for the contractually agreed and intended purpose considerable deviations from the performance description prepared by Phoenix Contact should result, Phoenix Contact has, at its own discretion, the right to rectify faults twice or to provide replacements twice. If Phoenix Contact does not succeed in eliminating the considerable deviations from the performance description within a reasonable period of time through rectification or replacement, or in avoiding them in such a way that the Registered User can use the Software in accordance with the contract, the Registered User can demand a reduction of payment (price reduction) or cancellation of this contract (rescission). In the latter case the Registered User must hand back any copies he may have made. _Failure of the rectification or replacement can only be presumed if the Registered User provides Phoenix Contact with verifiable documentation concerning the type and occurrence of deviations from the performance description (reproducible fault log) and if in spite of this documentation rectification is not possible, is refused by Phoenix Contact or is unreasonably delayed, if there are justified doubts concerning the prospects of success or if unacceptability prevails for other reasons.

5. Fault diagnosis and rectification within the framework of the guarantee shall be effected on the premises of the Registered User or of Phoenix Contact, at the discretion of Phoenix Contact. If a repair or service agreement has been reached between the Registered User and Phoenix Contact, the fault diagnosis and rectification may also be effected at the installation site of the computer on which the Software is being run in accordance with § 2 and § 3, by arrangement with the Registered User._Phoenix Contact shall be given the documentation and information which is in the possession of the Registered User and which is required to rectify the fault. If Phoenix Contact is to rectify the fault on the premises of the Registered User, the Registered User shall provide, free of charge, the necessary hardware and software as well as the other necessary operating conditions together with suitable operating personnel in such a way that the work can be carried out quickly. If no repair or service agreement has been reached, the Registered User shall reimburse Phoenix Contact for expenses for travelling and board for personnel sent to the installation site of the computer on which the licensed Software is being run._No other guarantee claims can be accepted.
6. Notwithstanding the above agreements, it is agreed that software identified as "BETA" or "ALPHA" is completely excluded from the guarantee, since this is only a preliminary or test version which does not correspond to the final product.

§ 6 Liability

1. Phoenix Contact accepts no liability for the Software meeting the requirements of the Registered User or working together with other programs selected by him.
2. Phoenix Contact accepts unlimited liability for damage due to deficiencies in title and absence of warranted qualities. Liability for initial inability is restricted to the purchase price as well as to such damage, which can typically be expected within the framework of a software transfer. In other respects, Phoenix Contact only accepts liability for intention and gross negligence, unless an obligation whose observance is of particular importance for the achievement of the contractual purpose (cardinal obligation) is violated. In the event of violation of a cardinal obligation, the limitation of liability for initial inability is applicable.
   If damage has been caused intentionally or through gross negligence, the amount of our liability is limited to the damage foreseeable under normal circumstances as a consequence of this obligation violation.
3. Notwithstanding the above agreements, it is agreed that no liability whatsoever can be accepted for software identified as "BETA" or "ALPHA" (cf. § 5.6)

§ 7 Prices

1. In accordance with its current price list, Phoenix Contact shall charge separately for
   – support in commissioning the Software
   – support in the analysis and rectification of faults arising through improper handling or for other reasons not arising from the Software.

§ 8 Compensation

The Registered User is fully liable to Phoenix Contact for all damage resulting from violations of this licensing agreement or of copyright.

§ 9 Termination

1. The Registered User can terminate the contract, insofar as it is unlimited in time, in whole or in part, with a period of notice of six months as from the end of each month. If a single license fee was paid for the transfer of the Software, this is not returnable.
2. On termination of the contract, the Registered User is obliged to return to Phoenix Contact the original as well as all copies and part-copies, as well as modified copies and copies combined with other program material, or verifiably to destroy them. The same applies for the program documentation and other documents provided.
3. The retention of an archive copy for backup purposes requires a separate written agreement.

§ 10 Export

The export of software, including the relevant data and documents, may require official approval - e.g. because of its type or its intended purpose. In the event of the sale of software to third parties, the Registered User shall in each case obtain the required export authorization on his own responsibility and effect delivery only in accordance with such authorization.

§ 11 Ancillary agreements

Ancillary agreements and alterations to the licensing conditions must be made in writing.

§ 12 Place of jurisdiction, choice of applicable law

1. For all disputes arising from the contractual relationship, the court at the headquarters of Phoenix Contact is competent if the Registered User is a merchant registered in the Commercial Register, a legal person under public law or if a special fund under public law is concerned.
2. The law of the Federal Republic of Germany applies for all legal relationships between the parties ensuing from the contract.
3. The utilization of the UN Sales Convention is barred.
4. Moreover, our Terms of Delivery and Payment for non-software products apply ab initio.

Phoenix Contact GmbH & Co.

Table of Contents

Section 1

This section informs you about

– the objectives and structure of this user manual.

1 Introduction

This manual and the associated C driver source code can be used to develop specific device drivers to link Generation 4 INTERBUS controller boards to any PC operating system. IBS PCI SC/I-T, IBS PCI SC/RI/I-T, and IBS PCI SC/RI-LK controller boards (referred to as IBS PCI controller boards in the following) are used to interface the open fieldbus system (standardized as IEC 61158) to PC systems (referred to as host system in the following) with PCI bus.

The host system and INTERBUS are coupled by means of a shared memory area designed as a 2-device Multi-Port Memory (MPM). An independent INTERBUS master, based on a MC68332 microcontroller and a protocol chip, is incorporated on the controller board.

This document first explains the basic, specific functions of the controller boards, and goes on to explain the general architecture of the MPM. It concludes with some notes on creating a device driver.

1.1 Files on the Disk

The disk contains the following directories:

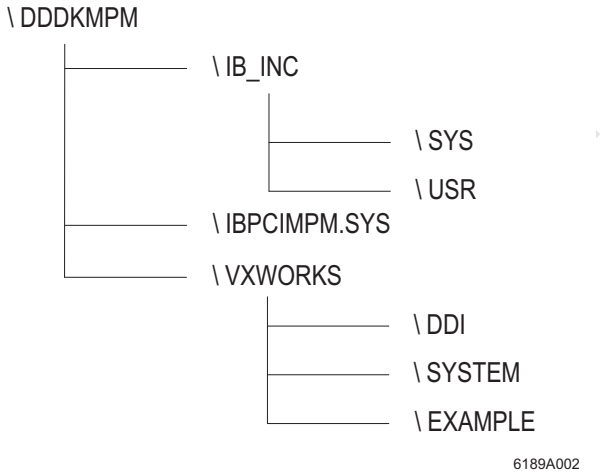


Figure 1-1 Disk directory structure

Table 1-1 Directory contents

Directory Name	Contents
IBPCIMPM.SYS	Contains the C source code for the driver.
SYS	Contains adaptations to various systems
USR	Contains include files for various systems
DDI	Contains the Device Driver Interface for VxWorks
SYSTEM	VxWorks system initialization for the IBS PCI SC/..
EXAMPLE	Contains example programs that use a finished driver.

Table 1-2 Files in the "IBPCIMPM.SYS" directory

File Name	Contents
startup.c	Initialization of hardware and software
drvinits.c	Driver initialization routines
vxwsys.c	Operating system functions/routines for VxWorks
Ibpcidrv.c	Functions/routines for PCI access
evthndg.c	Functions/routines for event handling
drvlimit.h	Limitation constants (maximum values, etc.)
errlog.h	Macros, constants, etc. for error logging and debugging
Ibpcidrv.c	Functions/routines for PCI access
os_adapt.h	Operating system-dependent data structures, macros, and constants
os_inc.h	Load operating system-dependent headers
pci_drv.h	Constants and structures for working with PCI devices
pcimsg.h	Constants for error messages
vxw_data.h	Constants and data structures for VXWorks tasks and priorities

Table 1-3 Files in the "IB_INC\SYS" directory

File Name	Contents
compiler.h	Compiler setting definitions
mpm40.h	Definitions of the MPM register addresses and their bit masks
Ioctl40.h	Definitions for the devIOCtrl driver function
drv_dbg.h	Defines, typedef, and prototypes for DriverDebugInfo commands

Table 1-4 Files in the "IB_INC\USR" directory

File Name	Contents
stdtypes.h	Generally valid standard type definitions
ddi_err.h	Definitions of possible driver error messages
ddi_usr.h	Defines, typedef, prototypes for DDI basic functions (without I/O control and management)
ibs_util.h	Declarations of the additional DDI functions, e.g., watchdog, read diagnostic registers, etc.
ibddivxw.h	DDI for VxWorks
ddi_macr.h	Macros for messages/data conversion
Svc_code.h	IBS service codes (send/receive)

Table 1-5 Files in the "VXWORX\EXAMPLE" directory

File Name	Contents
simple.c	Example program for application of DDI functions

Table 1-6 Files in the "VXWORX\SYSTEM" directory

File Name	Contents
ibspciinit.c	VxWorks system initialization

Table 1-7 Files in the "VXWORX\DDI" directory

File Name	Contents
ibddivxw.c	Device Driver Interface (DDI) for VxWorks

1.2 Conditions of Support

Upon purchasing this product, you are entitled to comprehensive support during implementation. This support is available to you free of charge, if required, for 30 days following registration of the product. This free support will only be provided via a personal e-mail address, which will be created for you.

How Can You Obtain This Support?

1. Register at the following Internet address:
   http://request.phoenixcontact.com/req_eldo/pciddk_d/ddkreg.htm
2. Please complete the registration form in full.
3. As soon as we receive your registration, you will receive confirmation together with your personal support e-mail address via e-mail.

Points to Note

Our free support is available during normal office hours (Germany) and only via your personal e-mail address.

Free support is available for 30 calendar days, commencing on the date of the confirmation of your registration. The support period may be extended for a fee. For details, please refer to Section ”Chargeable Support Services” on page 1-8.

Please ensure that you only register the product when you wish to work continuously on the implementation.

In general, a fee is charged for telephone and on-site support. For details, please refer to Section ”Chargeable Support Services” on page 1-8.

In some individual cases, it may be necessary for Phoenix Contact to telephone you, however, you will not be charged for this.

Our support can only be provided for sources supplied by Phoenix Contact. As we are not familiar with, and generally do not have access to, your development environment, Phoenix Contact does not accept any responsibility or liability for any implementation work affected by this environment.

Phoenix Contact only accepts liability for products supplied within the scope of legal requirements for product liability or according to the general delivery and payment conditions. Phoenix Contact does not accept any liability for driver implementation developed by you or by a third party.

Phoenix Contact does not accept any responsibility for development, testing, validation, release, and support of your driver implementation.

1.2.1 Chargeable Support Services

Using the free support service available via your personal e-mail address, you can, if necessary, take advantage of other support services. A fee is charged for these services and we require you to place a written or verbal order.

Please contact the relevant person in the sales department or your representative. A direct order cannot be made via your e-mail address.

– Extension of the e-mail account for an additional 30 days
  Order No. 27 10 26 2
  Order Designation IBS PCI DDK SUPPORT EXT

If necessary, we can offer you additional support services tailored to your requirements. Please contact the relevant person in the sales department or your representative. We will then be happy to supply you with a personal quotation.

Section 2

This section informs you about
– the PCI register,
– the host interface and
– the MPM for INTERBUS controller boards.

2 Structure and Interfaces of INTERBUS Controller Boards

PCI controller boards are directly connected with the host system using PCI connectors. This hardware interface has (according to the PCI bus specification 2.2) a 32-bit address and data bus. It has control lines in addition to the address and data lines. The controller boards support interrupt operation.

A detailed description of the controller board hardware and its installation in the host system can be found in the *IBS PCI SC QS UM E* Quick Start Guide *(Order No. 26 98 14 8).*

The software interface of the controller boards to the host system consists of a 16-byte window in the I/O area and a 256-kbyte window in the standard address area (memory area). The base addresses of both windows are specified by the system BIOS.

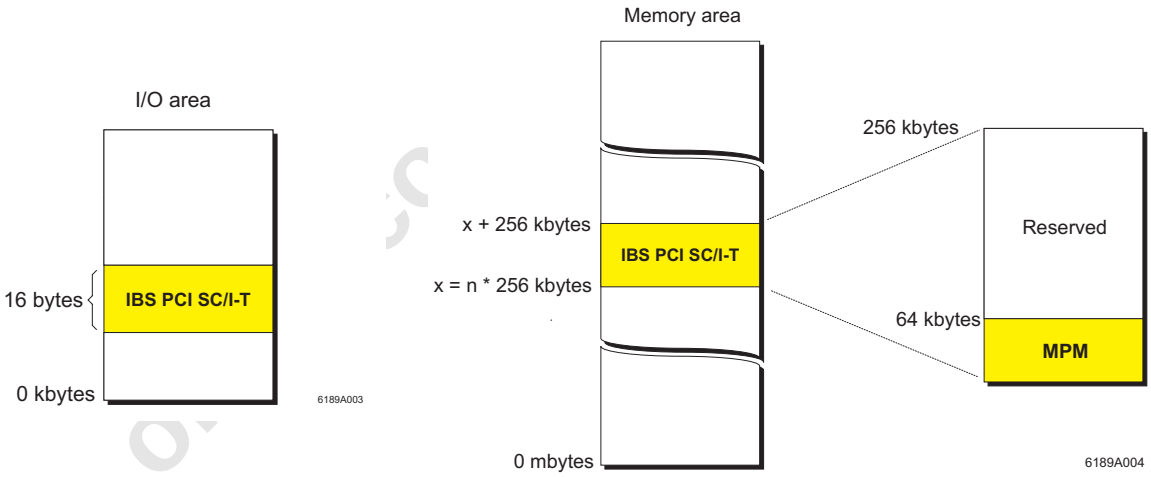


Figure 2-1 Controller board address windows in the host system

2.1 PCI Register

PCI registers integrate the controller boards into a PCI bus system. The following table shows the arrangement of the PCI registers and their controller board-specific settings.

Table 2-1 PCI register

00_{hex}	Device ID (0002_{hex} for IBS PCI SC...)		Vendor ID (1442_{hex} for Phoenix Contact)	
04_{hex}	Status register		Command register	
08_{hex}	Class code ($0C_{hex}$)			Revision ID (HW Rev)
$0C_{hex}$	Bits	Header type	Latency timer	Cache line size
10_{hex}	Base address register 0 (here: I/O address for the host interface register)			
14_{hex}	Base address register 1 (here: addresses for MPM and NV-RAM, 256 kbytes)			
18_{hex}	Base address register 2 (not used)			
$1C_{hex}$	Base address register 3 (not used)			
20_{hex}	Base address register 4 (not used)			
24_{hex}	Base address register 5 (not used)			
28_{hex}	Reserved (not used)			
$2C_{hex}$	Reserved (not used)			
30_{hex}	Base address expansion memory (expansion ROM) (not used)			
34_{hex}	Reserved (not used)			
38_{hex}	Reserved (not used)			
$3C_{hex}$	Max-Lat (00_{hex})	Min-Gnt (00_{hex})	Interrupt pin	Interrupt line

Comments for the table

The **vendor ID** 1442_{hex} is the approved manufacturer identifier for Phoenix Contact.

The **device ID** of boards with the file designation IBS PCI SC... is 0002_{hex}.

The **revision ID** indicates the hardware revision of the controller board.

Class code $0C_{hex}$ indicates a serial bus controller.

The PCI interface uses two base address registers:

Base address register 0 contains the I/O addresses for the host interface register. The length of the I/O address area is 16 bytes.

Base address register 1 is used for the MPM address area of the master and NV-RAM. A 128-kbyte memory address area is provided for the MPM and NV-RAM respectively.

The **MPM** acts as a data interface between the host system and the INTERBUS master (see also Section 2.3, "Multi-Port Memory (MPM)").

Only interrupt line A is used.

2.2 Host Interface for INTERBUS Controller Boards

The host interface forms the interface between the host system and the controller board MPM. The host interface also contains several registers.

2.2.1 Hardware Settings of Controller Boards

IBS PCI controller boards have DIP switches for setting controller board numbers (board number) and the test mode (ON/OFF). The transmission rate (500 kbaud/2 Mbaud) can also be set for the IBS PCI SC/RI-LK controller board.

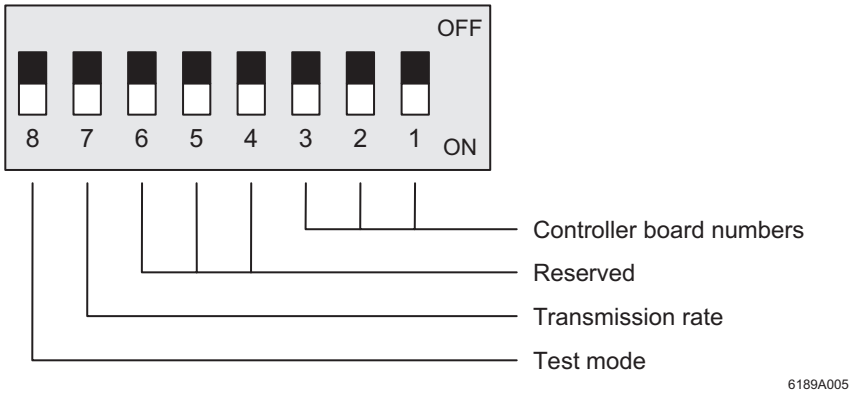


Figure 2-2 DIP switch assignment

As default upon delivery, all DIP switches are in the OFF position. DIP switches 4 to 6 are reserved for later function expansions and should be left in the OFF position.

Board number

The controller board number (board number) is specified using DIP switches **1** to **3**. Table 2-2 indicates possible settings:

Table 2-2 Possible settings for controller board numbers

Board Number	DIP Switch 1	DIP Switch 2	DIP Switch 3
1	OFF	OFF	OFF
2	OFF	OFF	ON
3	OFF	ON	OFF
4	OFF	ON	ON
5	ON	OFF	OFF
6	ON	OFF	ON
7	ON	ON	OFF
8	ON	ON	ON

Test mode

If DIP switch **8** is in the ON position, after startup the controller board will automatically switch to test mode and start up any connected bus. In test mode, data exchange is not possible in either direction between the host system and the MPM. No outputs are set.

Test mode may only be used for checking the INTERBUS system connected to the controller board. Test mode should not be used in normal system operation.

For the IBS PCI SC/RI-LK only:

Transmission rate

The transmission rate of the INTERBUS system is set, during operation via optical fiber cable, using DIP switch **7**. If the DIP switch is in the OFF position, the transmission rate is 500 kbaud. If the DIP switch is in the ON position, INTERBUS data is transmitted at 2 Mbaud.

This switch has no significance for controller boards IBS PCI SC/I-T and IBS PCI SC/RI/I-T and should remain in the OFF position. The baud rate setting is made automatically.

2.2.2 I/O Register

The controller board occupies 16 I/O addresses of the host system. The first eight addresses are occupied with control registers, which can be both read and written. After the control registers come the I/O registers for direct inputs and outputs.

Table 2-3 Addresses of the I/O registers

Offset	Name	Function
0	Board_Number	Read current set controller board number
1	Not_Used	Reserved
2	IRQ_Control_Host	Interrupt enable
3	WDT_Control_Host	Status, enabling, and reset of the watchdog timer
4	Reset_Control_Host	Master software reset
5	Not_Used	Reserved
6	Not_Used	Reserved
7	Status	Read and write status register
8	Direct_IN	Read inputs Direct IN 1 to Direct IN 6
9	Direct_OUT	Read and write outputs Direct OUT 1 and Direct OUT 2
10	Not_Used	Reserved
11	Not_Used	Reserved
12	Not_Used	Reserved
13	Not_Used	Reserved
14	Not_Used	Reserved
15	Not_Used	Reserved

2.2.2.1 Board_Number

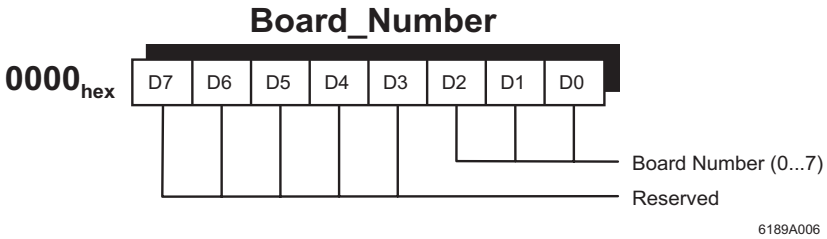


Figure 2-3 Structure of the Board_Number register

Read access

The Board_Number register reads the current controller board numbers (Board_Number) set using DIP switches **1** to **3**. Reserved bits are reported with a "0".

2.2.2.2 IRQ_Control_Host

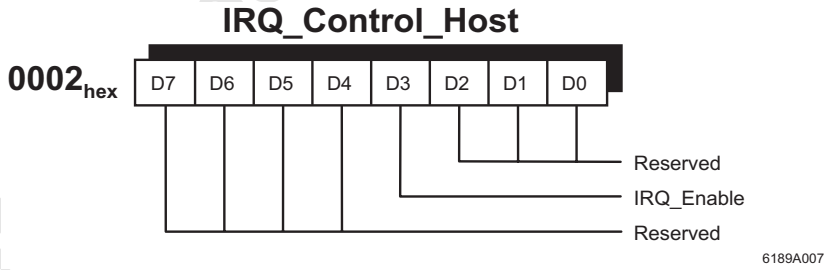


Figure 2-4 Structure of the IRQ_Control_Host register

Write access

In the IRQ_Control_Host register, the interrupt request line, which is used by the host system, is enabled. The interrupt is completed by resetting the host interface and can - if necessary - be enabled by the driver software. The selected interrupt request line is enabled if the IRQ_Enable bit receives the value "1".

The reserved bits must contain the value "0".

Read access

The IRQ_Enable bit can be read back. All other bits receive the value "0".

2.2.2.3 WDT_Control_Host

Write access

The host interface watchdog timer can be switched on, operated, and reset using the WDT_Control_Host register. The reserved bits must contain the value "0".

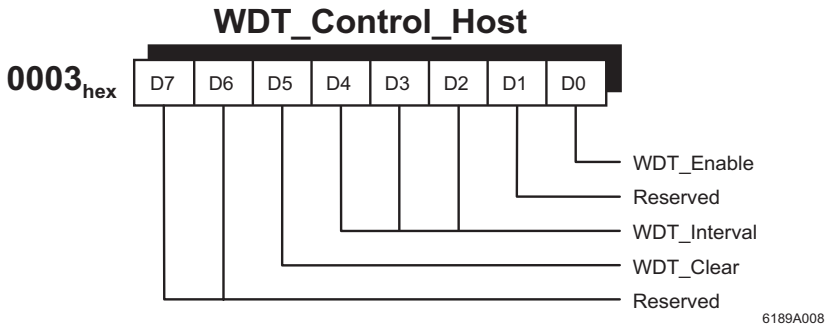


Figure 2-5 Structure of the WDT_Control_Host register during write access

Setting the WDT_Enable bit activates the watchdog timer. This timer is active and runs until it overflows or is switched off by a hardware reset of the host system. It **cannot** be deactivated by resetting the WDT_Enable bit. The monitoring time is determined by a bit combination in the "WDT_interval" (D2 to D4). The contents of this field are saved when the watchdog timer is switched on and can be modified while the timer is active. The watchdog timer is triggered by the read and write access of the WDT_Control_Host register. The WDT_Clear status is not saved, i.e., the watchdog timer cannot be switched off by setting this bit. WDT_Clear only resets the WDT_Status bit and the HF LED on the controller board after a watchdog timer has been triggered.

In the WDT_Interval (data bits 2 to 4) the following monitoring times can be set:

Table 2-4 WDT_Interval

D4	D3	D2	Monitoring Time
0	0	0	8.2 ms
0	0	1	16.4 ms
0	1	0	32.8 ms
0	1	1	65.5 ms
1	0	0	131.1 ms
1	0	1	262.1 ms
1	1	0	524.3 ms
1	1	1	1048.6 ms

Read access

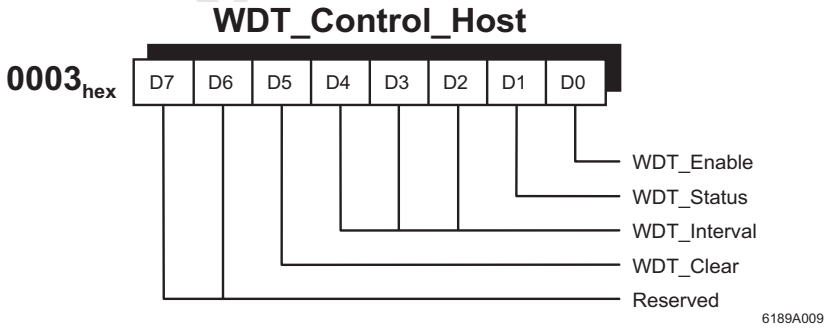


Figure 2-6 Structure of the WDT_Control_Host register during read access

During a read access, the WDT_Enable bit indicates whether the watchdog timer is running. This bit cannot be reset by a write access. The WDT_Status bit indicates the status of the monitoring output. If the bit has the value "0", the watchdog timer has always been triggered appropriately since power up. If the bit has the value "1", the watchdog timer was not reset in the specified time. In this case, the interrupt SRQ2L(0) is

generated. The HF LED on the controller board indicates a SysFail of the host system. This bit remains set until WDT_Clear is set or the timer is restarted. The set time interval WDT_Interval can be read back.

2.2.2.4 Reset_Control_Host

Write access

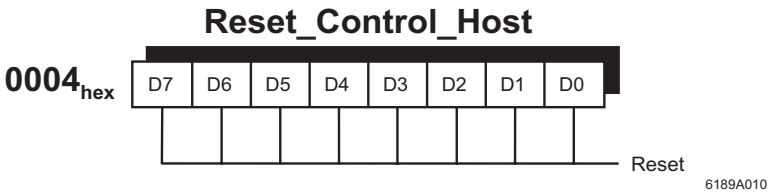


Figure 2-7 Structure of the Reset_Control_Host register during write access

A reset of the controller board master is triggered by writing the reset bit field using the bit pattern in Table 2-5. This means that both the firmware and INTERBUS are reset. After a reset the master starts a selftest with a "power on". The MPM is not immediately reset with a reset. The data remains in the MPM until the master checks the MPM during its test routine. Only then will the data be cleared and the register reset.

Table 2-5 Bit pattern for the master reset

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	0	1	0	1	0

Read access

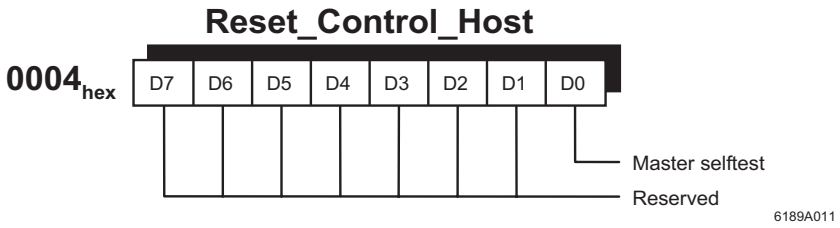


Figure 2-8 Structure of the Reset_Control_Host register during read access

During read access, data bit D0 indicates that the master is currently in the selftest mode and therefore cannot enable the MPM. If the bit is set to "0", the master is again ready to operate.

2.2.2.5 Status

Write access

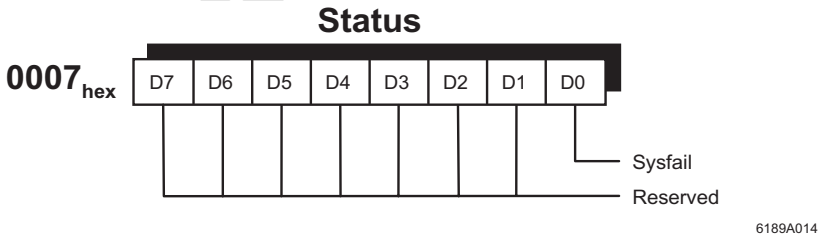


Figure 2-9 Structure of the status register during write access

Setting the SysFail bit indicates to the MPM and the INTERBUS master that an error has occurred in the host system.

This bit is set after startup. If the bit is set, no output data can be sent to the bus. Therefore this bit must be reset by the driver software.

Read access

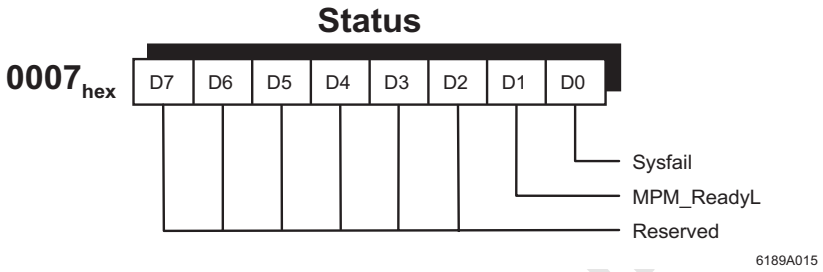


Figure 2-10 Structure of the status register during read access

The SysFail bit and MPM_ReadyL bit can be read back. All other bits are reserved and reported with a "0".

2.2.2.6 Direct_IN

Read access

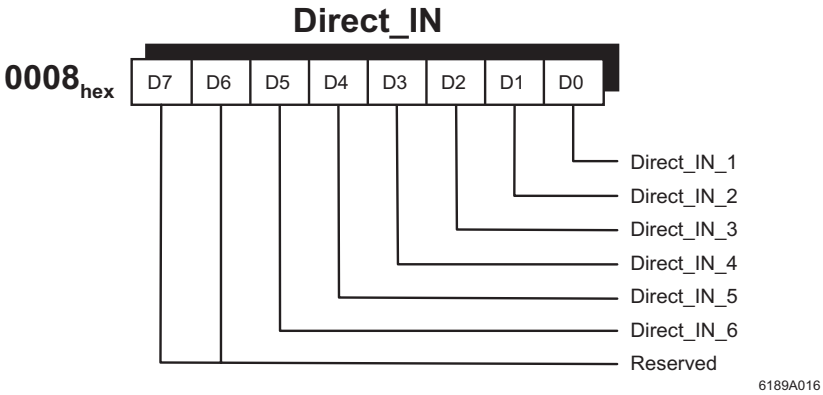


Figure 2-11 Structure of the Direct_IN register

The status of the direct inputs can be read back using this register.

2.2.2.7 Direct_OUT

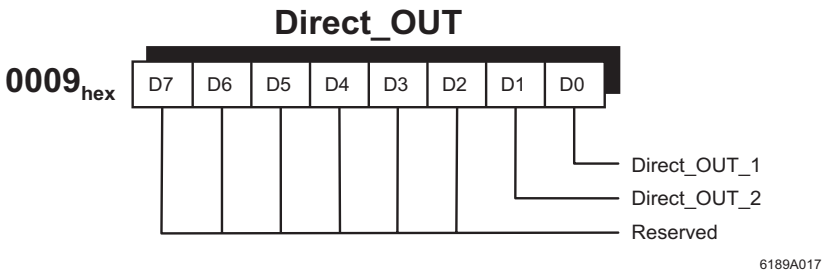


Figure 2-12 Structure of the Direct_OUT register

Write access Direct outputs can be set using this register.

Read access Direct outputs can also be read back.

2.3 Multi-Port Memory (MPM)

Along with the INTERBUS master and the host interface, the multi-port memory (MPM) is a central component of INTERBUS controller boards. As described above, all information exchanged between the host system and the INTERBUS master is routed through the MPM.

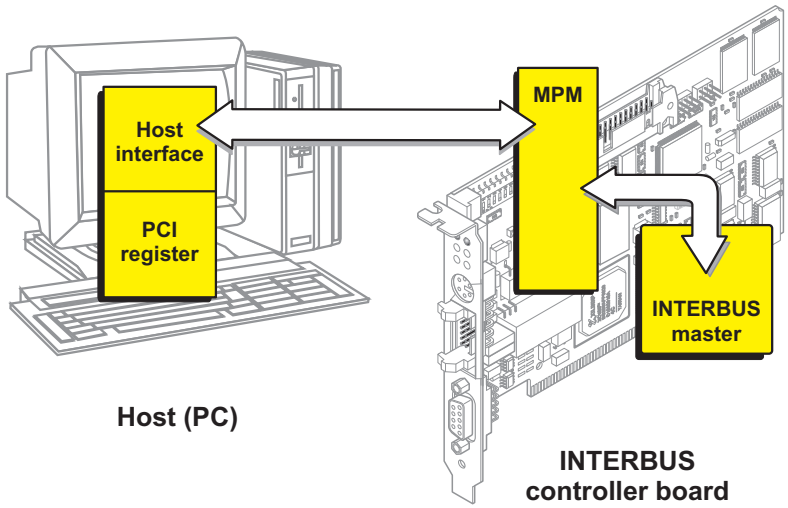


Figure 2-13 The MPM as the central interface of the controller board

The MPM control logic ensures prioritized assignment of access to the MPM. Time monitoring is started in parallel with accessing of the MPM. If a device takes too long to access or fails to access, a timeout is generated. This is indicated in an MPM register and access to the MPM is simultaneously re-enabled for other devices.

All data in the MPM is stored in Intel format ("little-endian" or "low-high" format). Appropriate conversion is therefore required on host system access.

In addition to the hardware registers, the MPM has a series of software registers, which are used specifically to control mailbox communication between the individual devices. The number and location of these registers are not predetermined by the hardware.

From the viewpoint of the user, the MPM therefore consists of the following functional components:

– Static RAM (SRAM)
– Memory manager
– Status and control register
– Serial data channel

Static RAM

The static RAM (SRAM) uses MPM address area 0000$_{hex}$ through FFFF$_{hex}$ and therefore occupies a special position within the MPM. This area is always available and cannot be blocked by the MPM memory manager. The SRAM contains the data area (DTA) and the mailbox area (MXA), i.e., all data exchanged between the individual nodes is routed via the SRAM. The segmentation and size of the individual areas is specified by the MPM firmware manager rather than being predetermined by the hardware.

Memory manager

The entire MPM address area is a maximum of 512 kbytes. Address area 00000$_{hex}$ through 0FFFF$_{hex}$ (64 kbytes) occupies a special position. This area contains the static RAM (SRAM) and the hardware registers. Up to 256 pages of any size can be displayed in the remaining area (10000$_{hex}$ –7FFFF$_{hex}$). It is possible to switch between the individual pages using a special hardware register in the MPM. Each node can display the relevant page independently of the other nodes. Address area 0000$_{hex}$ through FFFF$_{hex}$ (64 kbytes) is not switched with the rest and is thus available for every page.

Status and control register

MPM address area 3F90$_{hex}$ through 3FFF$_{hex}$ (i.e., within the SRAM area) contains a series of hardware registers. These registers contain, for example, MPM status information or are used to evaluate and generate signals (interrupts) between the individual nodes. The registers are available to all nodes. Registers that present the same contents to all nodes are distinguished from those that are dedicated to each individual node. However, the same number of registers at the same addresses are always available to all nodes.

Serial data channel

I/O shift registers and a serial EEPROM can be connected to the MPM serial data channel. The serial data channel is used to read and save or output configuration data. A distinction is made between access to the serial EEPROM and access to the shift registers. For example, the switches, the motherboard ID, and the MPM configuration are read via the shift registers. The serial data channel is accessed using four hardware registers in the MPM.

2.3.1 MPM Address Area

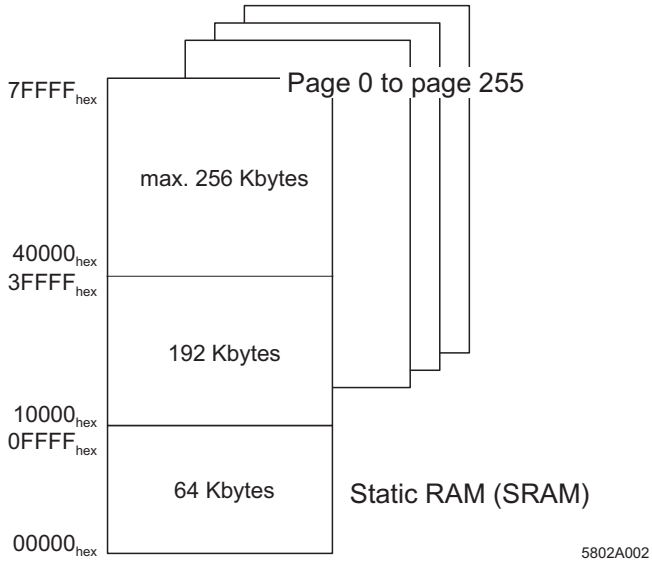


Figure 2-14 MPM address area

The diagnostic registers, for example, have the following addresses:

Register	Address
Diagnostic status register	3520_{hex}
Diagnostic parameter register	3522_{hex}
Ext. diagnostic parameter register	$37E6_{hex}$
Slave diagnostic status register	$37EC_{hex}$

For additional information on the diagnostic registers, please refer to the *Firmware Services and Error Messages* User Manual *IBS SYS FW G4 UM E, Order No. 27 45 18 5*.

2.3.2 Segmentation of the SRAM in the MPM

The SRAM is located from address 0000_{hex} through address $FFFF_{hex}$ in the MPM.

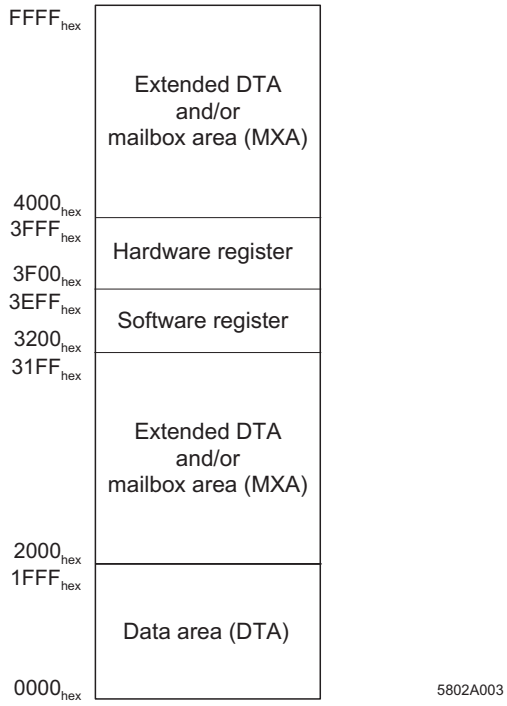


Figure 2-15 Segmentation of the SRAM

This document provides a general description of the MPM, i.e., the registers and MPM functions of a complete MPM with **four** nodes are described.

The IBS PCI controller board MPM is only designed for **two** nodes:

Node 0:	Host CPU (application program)
Node 1:	INTERBUS master (firmware)

All MPM registers and bits are only available to nodes 0 and 1. Accessing the registers for nodes 2 and 3 has no effect.

2.3.3 MPM Communication Options

The MPM is used to exchange information between devices (nodes). This communication can be related to the exchange of data or notification of an event. Both options are offered by the MPM and are supported by corresponding mechanisms. The MPM SRAM is available for exchanging data. For exchanging events, there are four sources of interrupts for each node, which can be evaluated independently of each other. It is also possible for both forms to be used in combination, as, for example, in the mailbox interface.

Communication Methods Used:

– Data interface
– Mailbox interface
– SysFail requests
– Signal interface
– Synchronization requests

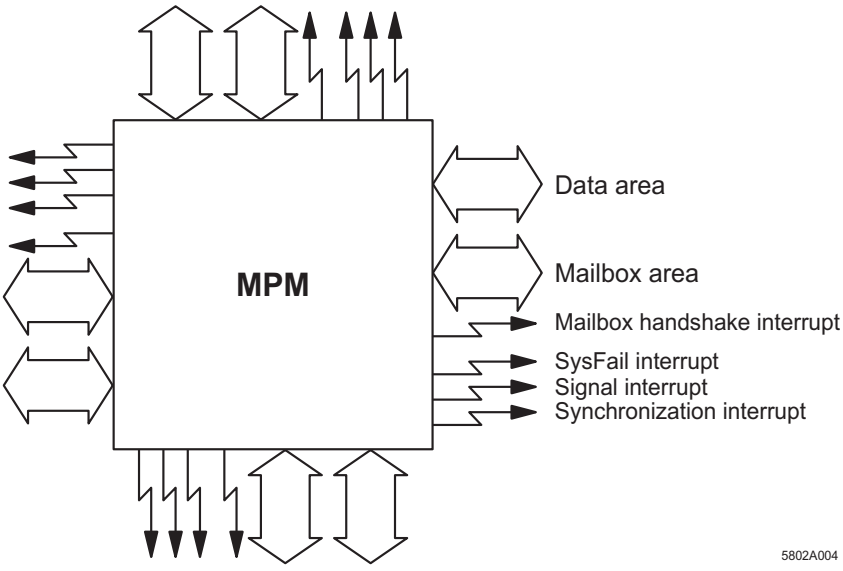


Figure 2-16 Diagram of the various MPM communication options

2.3.3.1 Data Interface (DTI)

Process data is exchanged with the INTERBUS master via the data interface.

Various areas are defined within the data area (DTA). The data interface (DTI) consists of both the "normal" data area, which contains process data and the extended data area, in which user-specific data can be stored.

No handshake is specified for exchanging data via the DTI. This means that in contrast to message exchange via the mailbox interface, data may be overwritten by one node while it is being read by another. The user must use their own transmission protocol. For IBS PCI controller boards, a protocol of this type is activated by default by the firmware, and is supported by the driver.

A detailed description of the data interface can be found in Section 3, "Data Exchange via the Data Area".

2.3.3.2 Mailbox Interface (MXI)

The mailbox interface (MXI) is a protocol-oriented interface via which messages can be exchanged between the nodes.

The mailbox interface consists of the mailbox area (MXA) and a number of hardware and software registers for each node. The area occupied by the mailbox area and the register addresses is predefined and must not be changed by the user. The size and location of the mailbox area and the software registers are specified in the MPM descriptor, which is created by the MPM master for each node (see Section "Structure of the MPM Descriptor" on page 2-49).

The mailbox area in turn is divided into a number of blocks ("mailboxes"). Each of these can take one message. When sending, the address of the mailbox containing the message is placed in a software register in the MPM and the receiving node is notified (see also Section 4, "Communication via the Mailbox Interface"). The mailbox is managed by the node driver. The length of one mailbox is 1 kbyte.

MPM registers used:

– Set HS Ax/Bx register
– Status register 1
– Handshake register A / handshake register B
– Send vector and acknowledge vector registers (software registers)

A detailed description of the mailbox interface and the handshake protocol used can be found in Section 4, "Communication via the Mailbox Interface".

2.3.3.3 SysFail Request

The SysFail logic can be used to immediately notify other nodes of serious system errors. The SysFail signal can be triggered either by hardware, via the corresponding MPM control lines or by software, by writing to a special MPM register. An interrupt is always generated at all nodes (apart from the initiating one) and indicated in an MPM register (status register 1). In the case of a SysFail initiated by the hardware, the logical status of the initiating line is indicated in a different MPM register (status SysFail register). The response to a SysFail interrupt depends on the particular node. In the event of a SysFail interrupt, the INTERBUS master, for example, sets all output data to zero. A node must acknowledge that it has detected a SysFail interrupt by writing to an MPM register (clear status bit register). This resets the corresponding bit in status register 1.

MPM registers used:

– Status register 1
– Status SysFail register
– Clear-status-bit register
– Set SysFail request register

2.3.4 MPM Hardware Register

MPM hardware registers are displayed in the MPM SRAM address area in $3F90_{hex}$ through $3FFF_{hex}$. A distinction is made between read and write hardware registers.

Write registers

The write registers are word registers that can initiate or pass on information or actions. The contents of the write registers can only be read back in summarized form in the read registers.

Read registers

The read registers are word registers. They can be used to read both the written data in the write registers and additional status and configuration data.

Table 2-6 Write register addresses in the MPM

Address	Register	Page	Address	Register	Page
$3F90_{hex}$	Set MPM node par ready 0	2-31	$3FCA_{hex}$	Set HS A15	2-37
$3F92_{hex}$	Set MPM node par ready 1	2-31	$3FCC_{hex}$	Set HS B7	2-37
$3F94_{hex}$	Set MPM node par ready 2	2-31	$3FCE_{hex}$	Set HS B15	2-37
$3F96_{hex}$	Set MPM node par ready 3	2-31	$3FD0_{hex}$	Set HS A2	2-37
$3F98_{hex}$	Switch memory	2-42	$3FD2_{hex}$	Set HS A10	2-37
$3F9C_{hex}$	Set sync req	2-41	$3FD4_{hex}$	Set HS B2	2-37
$3FA0_{hex}$	Set SysFail req	2-33	$3FD6_{hex}$	Set HS B10	2-37
$3FA2_{hex}$	Program bits	2-45	$3FD8_{hex}$	Set HS A6	2-37
$3FA4_{hex}$	Serial data	2-44	$3FDA_{hex}$	Set HS A14	2-37
$3FA6_{hex}$	Serial address	2-43	$3FDC_{hex}$	Set HS B6	2-37
$3FA8_{hex}$	Clear status bit 0	2-34	$3FDE_{hex}$	Set HS B14	2-37
$3FAA_{hex}$	Clear status bit 1	2-34	$3FE0_{hex}$	Set HS A1	2-37
$3FAC_{hex}$	Clear status bit 2	2-34	$3FE2_{hex}$	Set HS A9	2-37
$3FAE_{hex}$	Clear status bit 3	2-34	$3FE4_{hex}$	Set HS B1	2-37
$3FB0_{hex}$	Set MPM node SG int 0	2-38	$3FE6_{hex}$	Set HS B9	2-37

Table 2-6 Write register addresses in the MPM

Address	Register	Page	Address	Register	Page
$3FB2_{hex}$	Set MPM node SG int 1	2-38	$3FE8_{hex}$	Set HS A5	2-37
$3FB4_{hex}$	Set MPM node SG int 2	2-38	$3FEA_{hex}$	Set HS A13	2-37
$3FB6_{hex}$	Set MPM node SG int 3	2-38	$3FEC_{hex}$	Set HS B5	2-37
$3FB8_{hex}$	Set MPM node ready0	2-30	$3FEE_{hex}$	Set HS B13	2-37
$3FBA_{hex}$	Set MPM node ready1	2-30	$3FF0_{hex}$	Set HS A0	2-37
$3FBC_{hex}$	Set MPM node ready2	2-30	$3FF2_{hex}$	Set HS A8	2-37
$3FBE_{hex}$	Set MPM node ready3	2-30	$3FF4_{hex}$	Set HS B0	2-37
$3FC0_{hex}$	Set HS A3	2-37	$3FF6_{hex}$	Set HS B8	2-37
$3FC2_{hex}$	Set HS A11	2-37	$3FF8_{hex}$	Set HS A4	2-37
$3FC4_{hex}$	Set HS B3	2-37	$3FFA_{hex}$	Set HS A12	2-37
$3FC6_{hex}$	Set HS B11	2-37	$3FFC_{hex}$	Set HS B4	2-37
$3FC8_{hex}$	Set HS A7	2-37	$3FFE_{hex}$	Set HS B12	2-37

Table 2-7 Read register addresses in the MPM

Address	Register	Page	Address	Register	Page
$3F90_{hex}$	MPM configuration	2-26	$3FB2_{hex}$	Status SysFail	2-32
$3F98_{hex}$	Read memory page	2-43	$3FB4_{hex}$	Status node SG inf	2-40
$3FA2_{hex}$	RDY bits	2-46	$3FB6_{hex}$	Status register 2	2-29
$3FA4_{hex}$	Serial data	2-44	$3FC0_{hex}$	Handshake register A	2-35
$3FB0_{hex}$	Status register 1	2-27	$3FC2_{hex}$	Handshake register B	2-35

2.3.4.1 MPM Configuration Register

The MPM configuration register enables the nodes to read the board configuration. The register contains information about the size of the MPM and the status of the nodes.

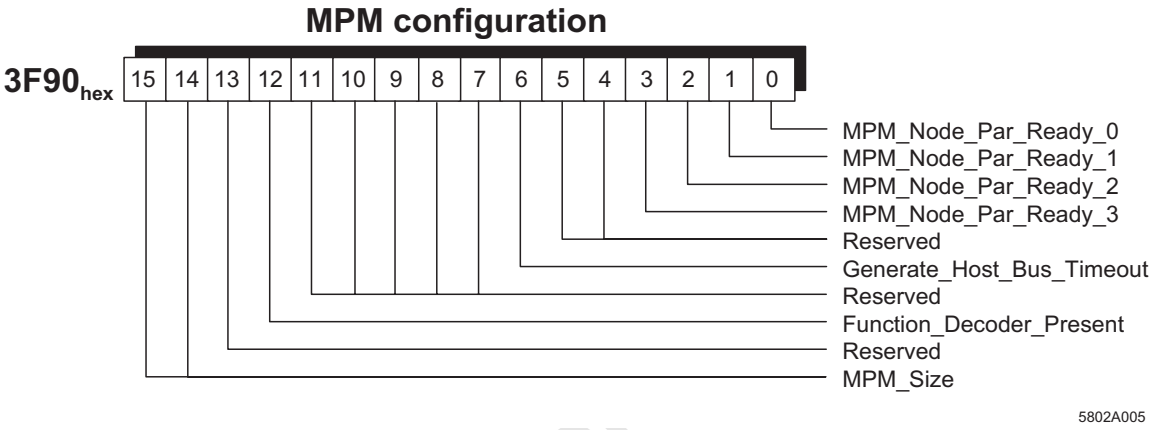


Figure 2-17 Bit assignment for the MPM configuration register (address $3F90_{hex}$)

MPM_Size

Bits 14 and 15 in the MPM configuration register indicate the size of the SRAM available for the MPM:

Table 2-8 Setting the MPM size

Address Area	Bit 14	Bit 15
16 kbytes	0	0
64 kbytes	0	1
256 kbytes	1	0
512 kbytes	1	1

MPM_Node_Par_-Ready_x

The *MPM_Node_Ready_x* bit indicates that the node has successfully completed its selftest (bit = 1: selftest successful). These bits are set by the nodes by writing the value 8000_{hex} to the set MPM node ready x register and reset with the value 0000_{hex}.

2.3.4.2 Status Register 1

Status register 1 contains information about the status of the individual nodes. It contains four bits for each node:

Node 0:	Bit 12 - bit 15
Node 1:	Bit 8 - bit 11
Node 2:	Bit 4 - bit 7
Node 3:	Bit 0 - bit 3

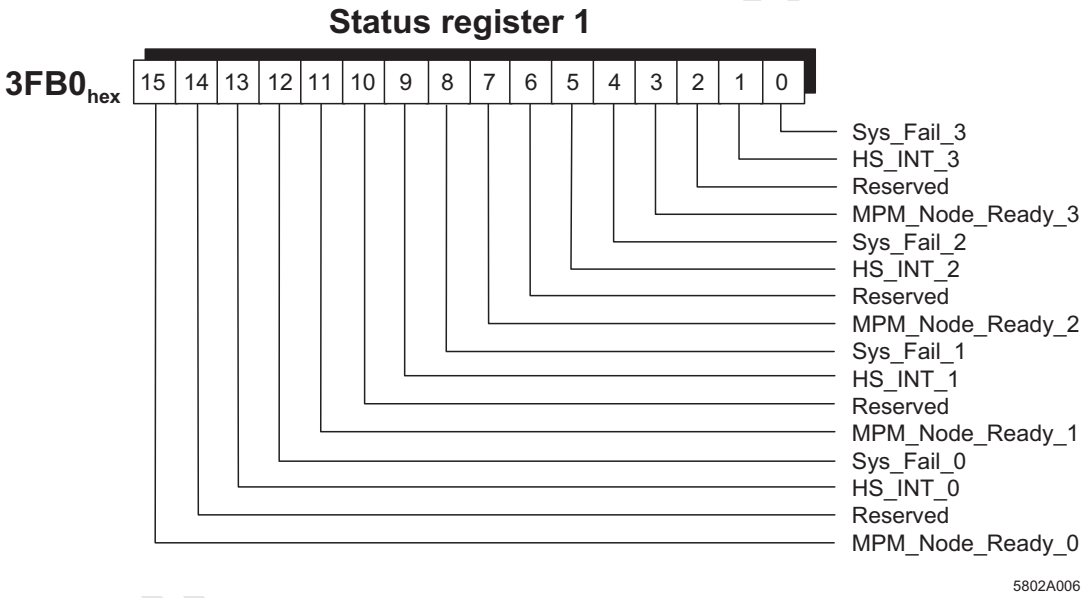


Figure 2-18 Bit assignment for status register 1 (address $3FB0_{hex}$)

MPM_Node_Ready_-x

The *MPM_Node_Ready_x* bit indicates that the node has successfully completed its selftest (bit = 1: selftest successful). These bits are set by the nodes by writing the value 8000_{hex} to the set MPM node ready x register and reset with the value 0000_{hex}.

HS_Int_x

Setting the *HS_Int_x* bit in status register 1 indicates to the node that there is a message ready for it in the MPM (see Section 4, "Communication via the Mailbox Interface"). The bit can be used in polling mode or in an interrupt routine to locate the cause of the interrupt more precisely and is a summary of the handshake bits for the node.

Sys_Fail_x

The *Sys_Fail_x* bit is used to indicate a serious malfunction of a node. The *Sys_Fail_x* bit is set by the hardware or the set SysFail register and has to be reset by the nodes by writing the value 0000_{hex} to the clear status SysFail x registers. The status SysFail register can be used to determine which node has signaled the system error.

The *Sys_Fail_x* bits do not indicate the status of the line that has initiated the SysFail.

2.3.4.3 Status Register 2

Status register 2 contains additional information about the status of the individual nodes. For each node, four bits are provided in the register:

Node 0:	Bit 12 - bit 15
Node 1:	Bit 8 - bit 11
Node 2:	Bit 4 - bit 7
Node 3:	Bit 0 - bit 3

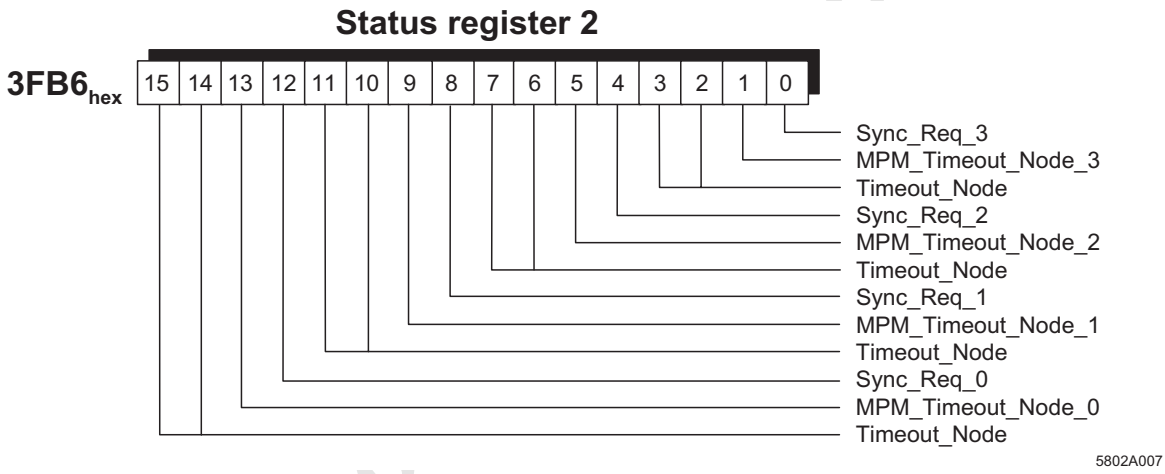


Figure 2-19 Bit assignment for status register 2 (address $3FB6_{hex}$)

Sync_Req_x

Setting the *Sync_Req_x* bit (bit = 1) in status register 2 indicates that the sync interrupt has been generated for the associated node. It is not possible to determine which node has caused the sync interrupt. The sync_req bit is reset by writing set bit 14 to the clear status bit register. Each node can only clear its own bit, i.e., node 1, for example, can only reset the *Sync_Req_1* bit.

2.3.4.4 The Set MPM Node Ready x Register

Writing the value 8000_{hex} (most significant bit = 1) to a Set MPM node ready x register sets the corresponding *MPM_Node_Ready_x* bit in MPM status register 1. The bit is reset by writing the value 0000_{hex} (most significant bit = 0) to the register. By setting the bit, the node signals that it is ready to operate. Each node has its own set MPM node ready x register.

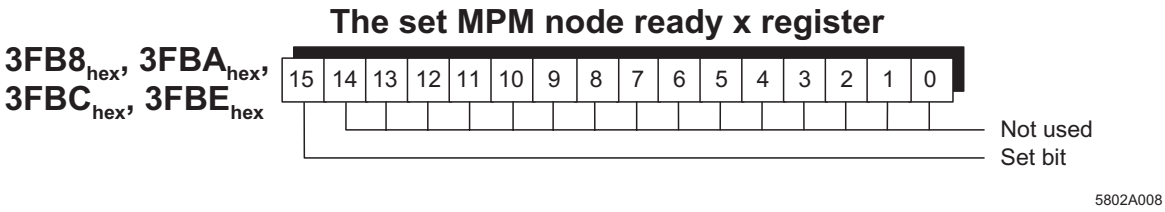


Figure 2-20 Bit assignment for the set MPM node ready x register

Table 2-9 The set MPM node ready x register addresses

Address	Register
$3FB8_{hex}$	Set MPM node ready 0
$3FBA_{hex}$	Set MPM node ready1
$3FBC_{hex}$	Set MPM node ready2
$3FBE_{hex}$	Set MPM node ready3

2.3.4.5 The Set MPM Node Par Ready x Register

After successful parameterization, a node sets the corresponding *MPM_Node_Par_Ready_x* bit in the MPM configuration register. This is achieved by writing a value with a set most significant bit, e.g., the value 8000_{hex}, to the set MPM node par ready x register. Writing a reset bit 15 (bit = 0) to the register also resets the corresponding *MPM_Node_Par_Ready_x* bit in the MPM configuration register. Each node has its own register, which corresponds to the node number (for example, node 1 uses the set MPM node par ready 1 register).

$3FB8_{hex}$, $3FBA_{hex}$, $3FBC_{hex}$, $3FBE_{hex}$

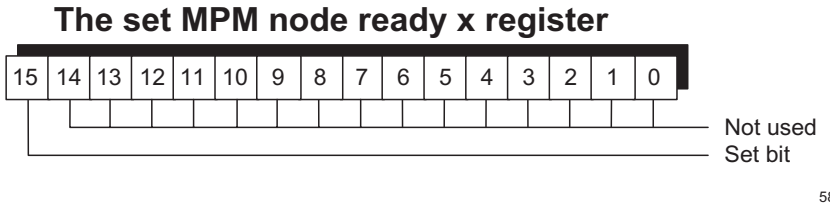


Figure 2-21 Bit assignment for the set MPM node par ready x register

Table 2-10 The set MPM node par ready x register addresses

Address	Register
$3F90_{hex}$	Set MPM node par ready 0
$3F92_{hex}$	Set MPM node par ready 1
$3F94_{hex}$	Set MPM node par ready 2
$3F96_{hex}$	Set MPM node par ready 3

2.3.4.6 Status SysFail Register

The status SysFail register can be used to determine which node has signaled a system error (bit = 1: node system error). The bits in the register correspond to the status of the corresponding SysFail line.

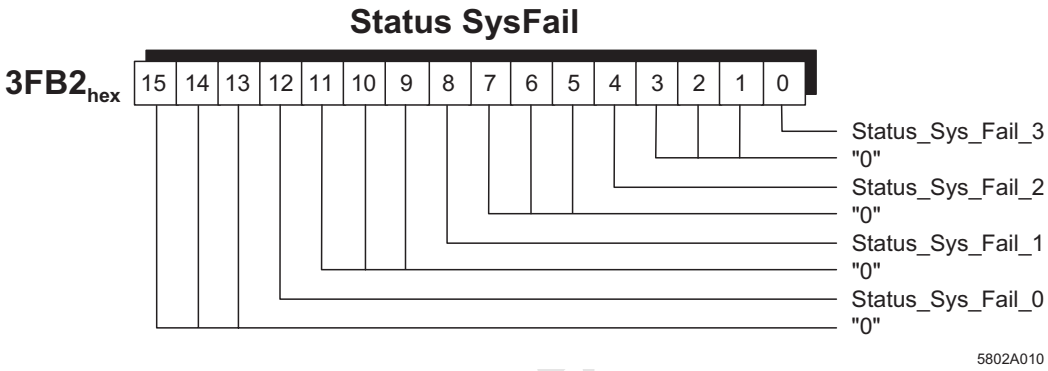


Figure 2-22 Bit assignment for the status Sy^sFail register (address $3FB2_{hex}$)

Status_Sys_Fail_x The *Status_Sys_Fail_x* bit corresponds directly to the hardware signal used to indicate the system error. It is not possible to reset the bit by writing to an MPM register. This is only possible once the signal (hardware line) indicating the error has been withdrawn.

2.3.4.7 The Set SysFail Request Register

The SysFail signal can be generated in two different ways: a hardware signal to the MPM control logic or the software writing to the set SysFail request register. As with other registers, writing with bit 15 (the most significant bit) set initiates the signal. This generates a SysFail interrupt on all other nodes.

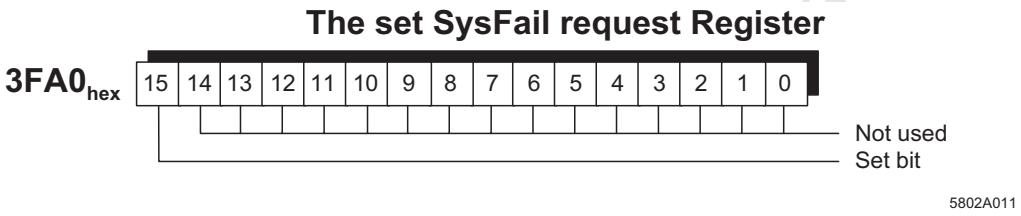


Figure 2-23 Bit assignment for the set SysFail request register (address $3FA0_{hex}$)

2.3.4.8 The Clear Status Bit x Registers

The clear status bit x registers are used to reset certain signals sent between the nodes. Each node has its own clear status bit x register, i.e., for a complete MPM there are four clear status bit x registers.

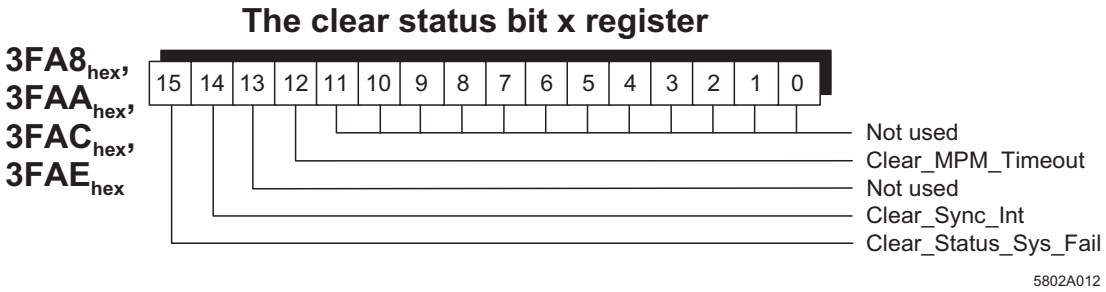


Figure 2-24 Bit assignment for the clear status bit x registers

Table 2-11 Clear status bit x register addresses

Address	Register
$3FA8_{hex}$	Clear status bit 0
$3FAA_{hex}$	Clear status bit 1
$3FAC_{hex}$	Clear status bit 2
$3FAE_{hex}$	Clear status bit 3

As can be seen in Figure 2-24, the following signals can be reset using the clear status bit x register:

– MPM timeout
– Sync signal
– Status SysFail signal

For each signal to be reset, enter a "1" in the corresponding bit position. This enables the signals to be reset individually or simultaneously.

2.3.4.9 Handshake Register A and Handshake Register B

32 handshake bits are defined for exchanging acknowledged messages between the nodes. The handshake bits are used by the mailbox handshake protocol for sending, receiving, and acknowledging messages between the nodes.

By evaluating handshake registers A and B, a node can determine which other node has caused which handshake interrupt (see also Section 4, "Communication via the Mailbox Interface").

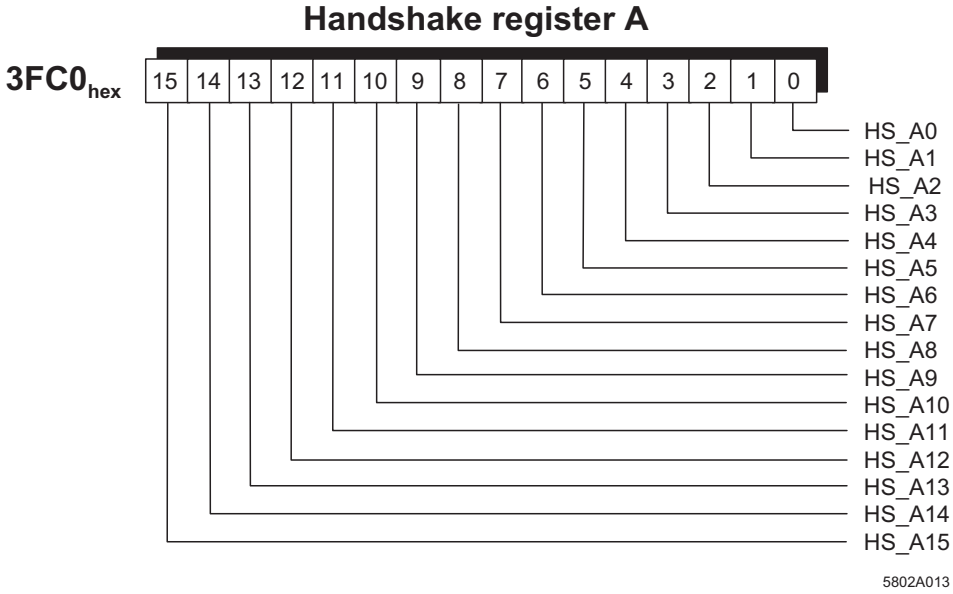


Figure 2-25 Bit assignment for handshake register A (address $3FC0_{hex}$)

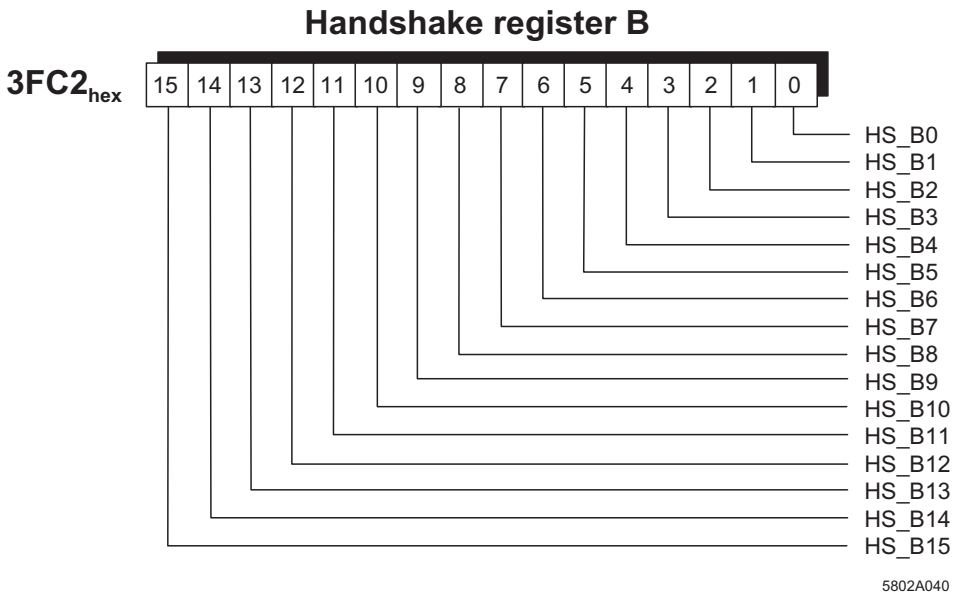


Figure 2-26 Bit assignment for handshake register B (address $3FC2_{hex}$)

2.3.4.10 The Set HS Ax Register and the Set HS Bx Register

These 32 registers are used for implementing the mailbox handshake protocol. The assignment and meaning of the registers are described in detail in the mailbox handshake protocol. Writing set bit 15 (bit = 1) to a set HS Ax/Bx register activates an interrupt at the corresponding node, the *HS_INT_x* bit in status register 1 changes to "1", and the associated bit is set in one of the two handshake registers. Writing reset bit 15 (bit = 0) to a set HS Ax/Bx register resets the associated bit in the handshake register and terminates the interrupt request.

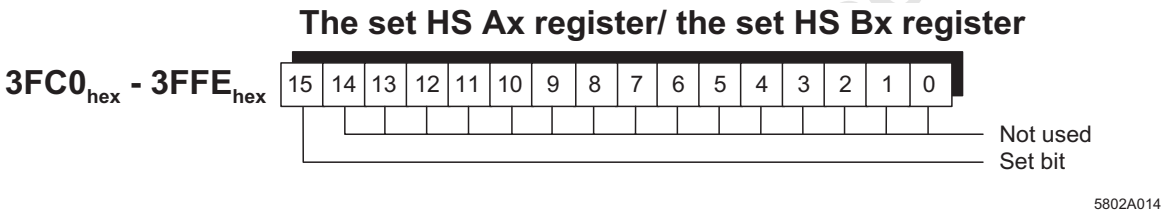


Figure 2-27 Bit assignment for the set HS Ax register and set HS Bx register

Table 2-12 The set HS Ax register and the set HS Bx register addresses

Address	Register	Address	Register	Address	Register	Address	Register
$3FC0_{hex}$	Set HS A3	$3FD0_{hex}$	Set HS A2	$3FE0_{hex}$	Set HS A1	$3FF0_{hex}$	Set HS A0
$3FC2_{hex}$	Set HS A11	$3FD2_{hex}$	Set HS A10	$3FE2_{hex}$	Set HS A9	$3FF2_{hex}$	Set HS A8
$3FC4_{hex}$	Set HS B3	$3FD4_{hex}$	Set HS B2	$3FE4_{hex}$	Set HS B1	$3FF4_{hex}$	Set HS B0
$3FC6_{hex}$	Set HS B11	$3FD6_{hex}$	Set HS B10	$3FE6_{hex}$	Set HS B9	$3FF6_{hex}$	Set HS B8
$3FC8_{hex}$	Set HS A7	$3FD8_{hex}$	Set HS A6	$3FE8_{hex}$	Set HS A5	$3FF8_{hex}$	Set HS A4
$3FCA_{hex}$	Set HS A15	$3FDA_{hex}$	Set HS A14	$3FEA_{hex}$	Set HS A13	$3FFA_{hex}$	Set HS A12
$3FCC_{hex}$	Set HS B7	$3FDC_{hex}$	Set HS B6	$3FEC_{hex}$	Set HS B5	$3FFC_{hex}$	Set HS B4
$3FCE_{hex}$	Set HS B15	$3FDE_{hex}$	Set HS B14	$3FEE_{hex}$	Set HS B13	$3FFE_{hex}$	Set HS B12

2.3.4.11 The Set MPM Node SG Int x Register

The set MPM node SG int x registers can be used to send a signal (interrupt) to any other nodes. The meaning of this signal is left entirely to the user. Each node has its own register, which corresponds to the node number (for example, node 1 uses the set MPM node par ready 1 register).

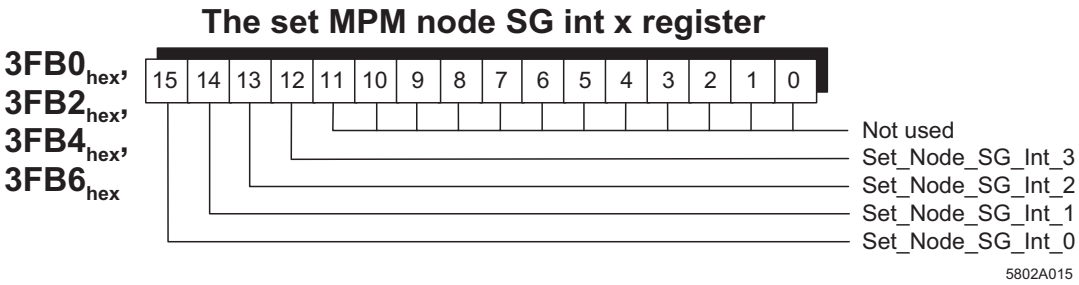


Figure 2-28 Bit assignment for the set MPM node SG int x register

Table 2-13 The set MPM node SG int x register addresses

Address	Register
$3FB0_{hex}$	Set MPM node SG int 0
$3FB2_{hex}$	Set MPM node SG int 1
$3FB4_{hex}$	Set MPM node SG int 2
$3FB6_{hex}$	Set MPM node SG int 3

The nodes to which a signal is to be sent are selected by the bit pattern written to the register. A set bit (bit = 1) activates an interrupt at the corresponding node and sets the associated bit in the status node SG inf register (see also Section 2.3.4.12, "Status Node SG Inf Register"). A signal can be sent to one or more nodes simultaneously.

The bits in the status node SG int x register are reset and the interrupt request cleared by writing a reset bit (bit = 0) to the bit position of the required node.

Table 2-14 Assignment of register bits to nodes

Bit	Function Executed
Bit 15 = 1	Signal to node 0
Bit 14 = 1	Signal to node 1
Bit 13 = 1	Signal to node 2
Bit 12 = 1	Signal to node 3
Bit 15 = 0	Delete signal to node 0
Bit 14 = 0	Delete signal to node 1
Bit 13 = 0	Delete signal to node 2
Bit 12 = 0	Delete signal to node 3

A node cannot send a signal to itself.

2.3.4.12 Status Node SG Inf Register

The status node SG inf register (status node signal interface register), can be used to determine from the bit(s) set which node has sent the MPM node SG int signal to which other node(s).

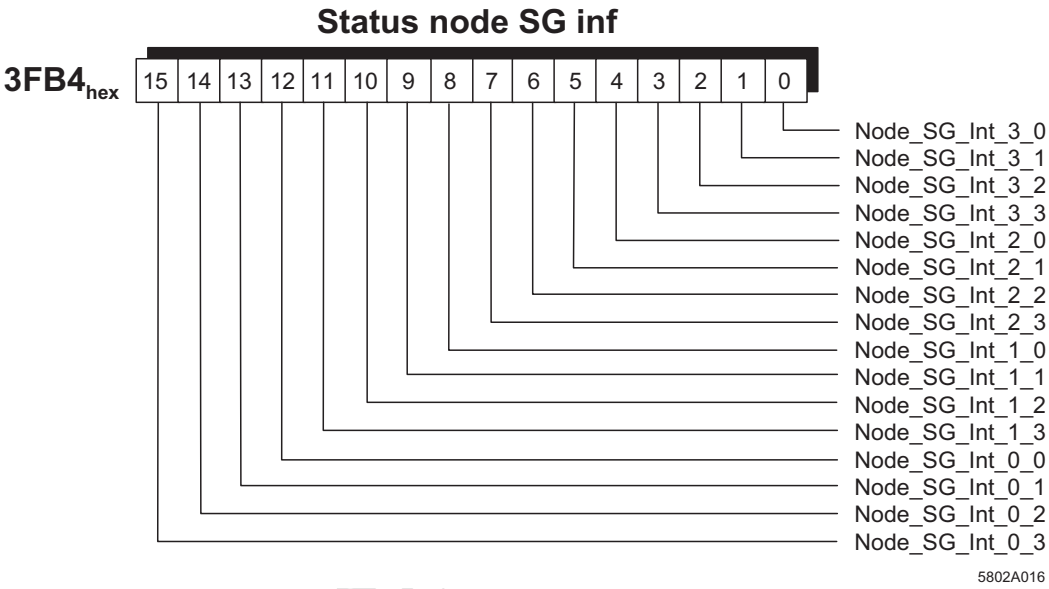


Figure 2-29 Bit assignment for the status node SG inf register (address $3FB4_{hex}$)

In Figure 2-29 the data bits are "Node SG int x y", where "y" designates the originator and "x" the receiver of the MPM node SG int signal.

2.3.4.13 The Set Sync Register

The set sync register can be used to send a further signal (interrupt) to any other nodes. The meaning of this signal is left entirely to the user. One possible application, as implied by the name, is synchronization during data exchange between the nodes. (In firmware version 4.x this signal is already used for synchronous mode.) In addition to activating an interrupt, the corresponding *Sync_Req_x* bits in status register 2 are set. The *Sync_Req_x* bits in status register 2 are reset using the clear status bit x registers.

There is only one register of this type in the MPM. The nodes are, therefore, always accessed through the same register.

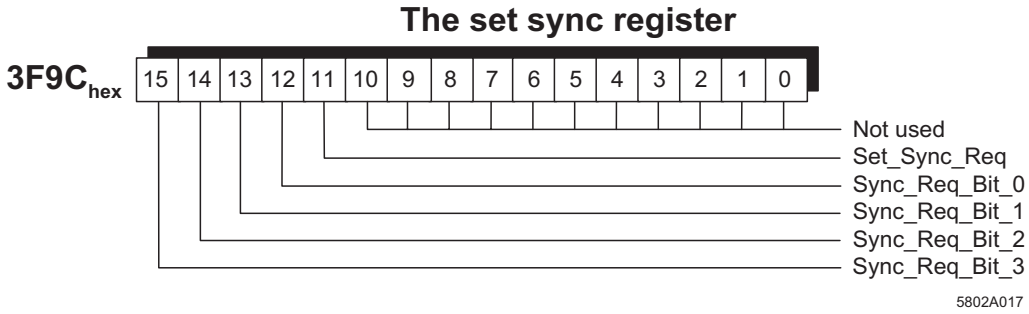


Figure 2-30 Bit assignment for the set sync register (address $3F9C_{hex}$)

Set_Sync_Req

Setting this bit to "1" causes the sync signal (interrupt) to be sent to the nodes according to the sync req bit mask set. The *Set_Sync_Req* bit can be set at the same time as the selection bit pattern, i.e., one write access to the register is sufficient. When the signal is initiated the *Set_Sync_Req* bit is automatically withdrawn, whereas the selection mask (bit 15 - bit 12) remains unchanged.

Sync_Req_Bit_x

The nodes to which a signal is to be sent are selected by the bit pattern in the four most significant bits of the register. A set bit (bit = 1) activates a sync interrupt at the corresponding node and sets the associated bit in status register 2 (see also Section 2.3.4.3). A signal can be sent to one or more nodes simultaneously.

Table 2-15 Assignment of the sync req bits to the nodes

Bit	Function Executed
Bit 12 = 1	Signal to node 0
Bit 13 = 1	Signal to node 1
Bit 14 = 1	Signal to node 2
Bit 15 = 1	Signal to node 3

2.3.4.14 The Switch Memory Register

The MPM address area above $FFFF_{hex}$ can be switched between several pages. The switch memory register is used to specify which page is displayed. The pages can be displayed by each node independently of the other nodes. Thus, for example, node 1 can display page 3, while node 0 works with page 0. It is possible to select from a maximum of 256 pages. The number of pages depends on the controller board.

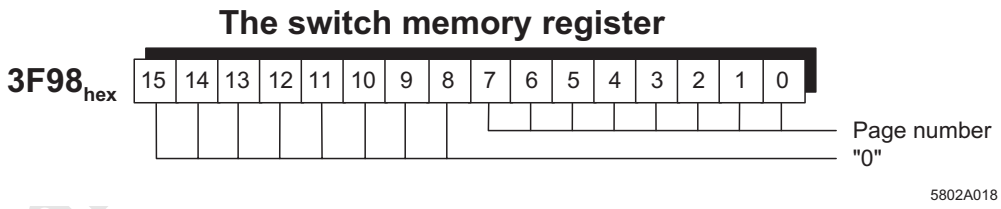


Figure 2-31 Bit assignment for the switch memory register (address $3F98_{hex}$)

The function decoder is also displayed using the switch memory register.

2.3.4.15 The Read Memory Page Register

As described above, the MPM address area above $FFFF_{hex}$ can be switched between several pages. For each node, the read memory page register indicates which page is currently displayed. Each individual node can display different pages. Thus, for example, node 1 can display page 3, while node 0 works with page 0.

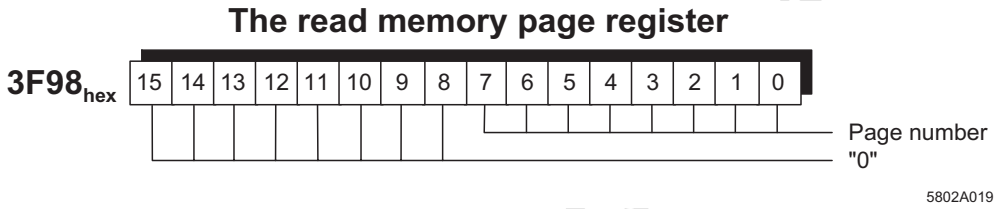


Figure 2-32 Bit assignment for the read memory page register (address $3F98_{hex}$)

2.3.4.16 Serial Address Register

The serial address register is a write register. The register contains the address for the next serial access. It can be an EEPROM or serial IN/OUT address. The address is entered as a byte address. When accessing the serial EEPROM the least significant address bit (0) is not evaluated, as EEPROM data exchange is word-oriented. However, byte addresses are always used for serial IN or serial OUT access.

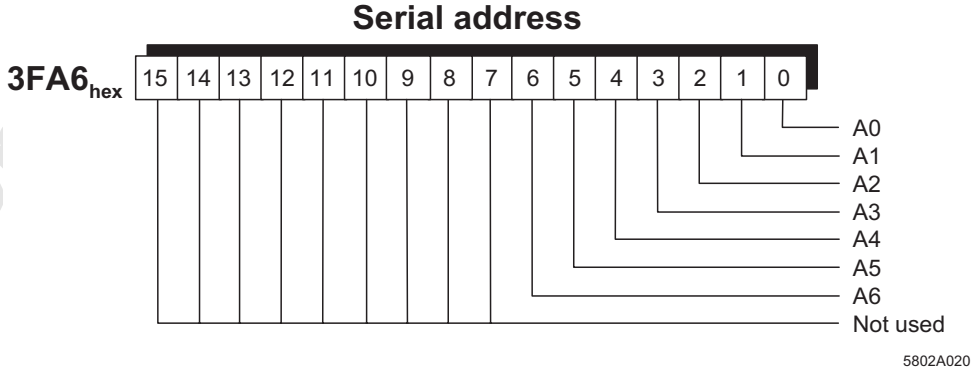


Figure 2-33 Bit assignment for the serial address register (address $3FA6_{hex}$)

The following serial address registers are available:

Table 2-16 Serial address register

Address	Register	Page
0002_{hex}	Configuration data	2-46

Only the register address is entered in the serial address register. The value of the register entered in the serial address register is given in the serial data register (see page 2-44).

2.3.4.17 Serial Data Register

The serial data register is used for both write and read access to the MPM serial channel. It is, therefore, a read/write register. The register contains either the data to be written or the data that has been read. All 16 bits are valid when accessing the serial EEPROM. However, when serial IN or serial OUT is accessed, only the low-order byte (bit 0 to bit 7) is valid.

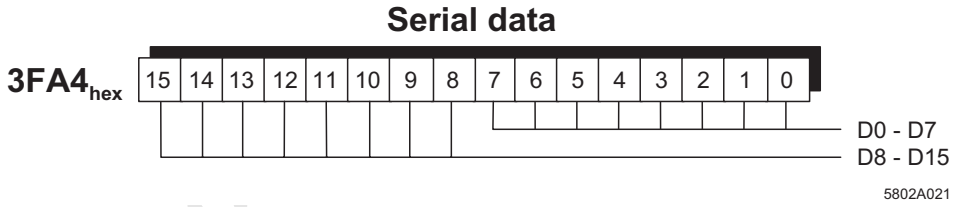


Figure 2-34 Bit assignment for the serial data register (address $3FA4_{hex}$)

2.3.4.18 The Program Bits Register

The program bits register is a write register whose four most significant bits indicate how the serial data channel is to be accessed.

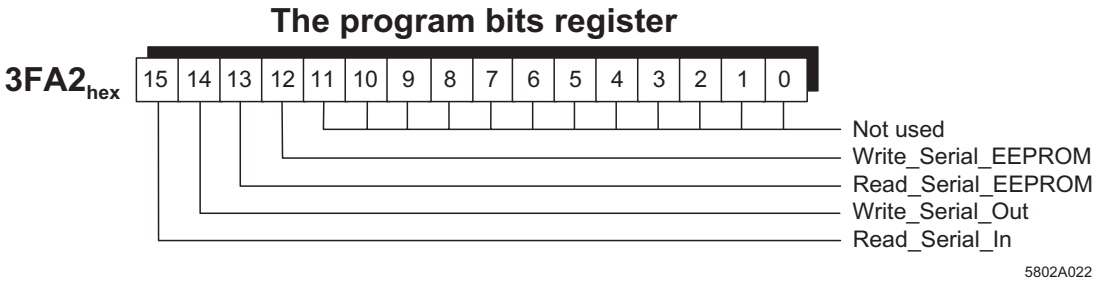


Figure 2-35 Bit assignment for the program bits register (address $3FA2_{hex}$)

Table 2-17 Meaning of the bits

Bit	Function Executed
Bit 15 = 1	Read data from serial IN
Bit 14 = 1	Write data to serial OUT
Bit 13 = 1	Read data from EEPROM
Bit 12 = 1	Write data to EEPROM

Several bits can be set simultaneously. In this case, the individual accesses are processed according to a specified **priority list**:

1. Read serial IN (highest priority)
2. Read EEPROM
3. Write EEPROM
4. Write serial OUT (lowest priority)

2.3.4.19 Ready Bits Register

The ready bits register is a read register whose bits indicate which accesses to the MPM serial interface (for example, serial EEPROM) have been completed. A distinction is made between access to the EEPROM and access to the serial IN/OUT interface. Each set bit (bit = 1) indicates that the access has been completed.

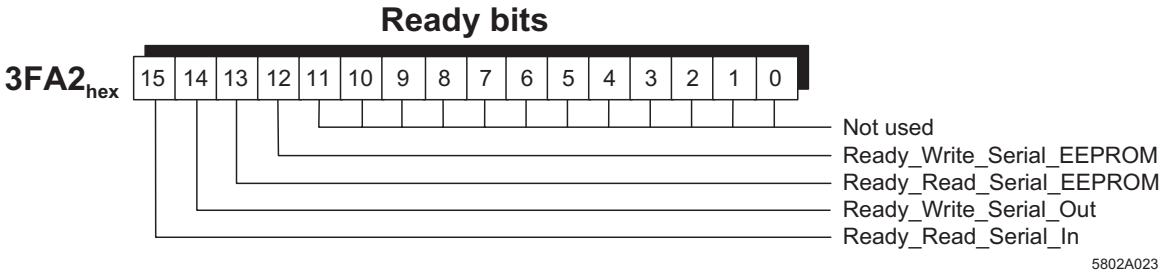


Figure 2-36 Bit assignment for the ready bits register (address $3FA2_{hex}$)

2.3.4.20 Configuration Data Register

Information about which nodes are present can be obtained by reading the serial IN data channel. Information about the current system configuration can be found at address 2 in the serial IN data channel.

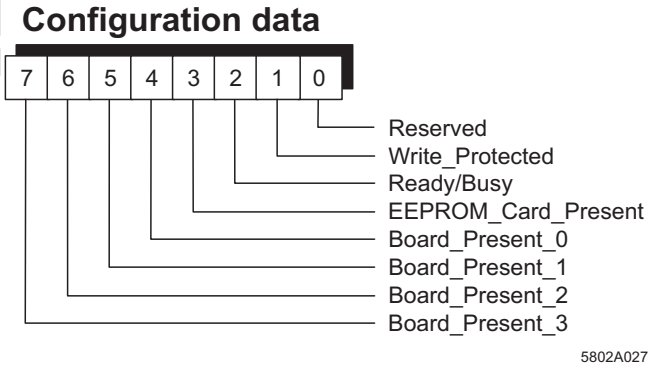


Figure 2-37 Bit assignment for the configuration data register (**address 2**)

Write_Protected

Indicates whether write protection is activated on a memory card.

Bit 1 = 1: Write protection active

Bit 1 = 0: Write protection deactivated

Ready/busy

Indicates whether a write access to a memory card has been completed.

Bit 2 = 1: Access completed (RDY)

Bit 2 = 0: Access not yet completed (BSY)

EEPROM_Card_-Present

Indicates whether there is a memory card (EEPROM, Flash, etc.) on the controller board.

Bit 3 = 1: No memory card present

Bit 3 = 0: Memory card present

Board_Present_x

These four bits indicate which nodes are present. A deleted bit indicates that the relevant node is available.

Table 2-18 Board present bits

Node	Bit 7	Bit 6	Bit 5	Bit 4
0	0	X	X	X
1	X	0	X	X
2	X	X	0	X
3	X	X	X	0

2.3.5 MPM Software Register

In addition to the hardware registers, the MPM contains a series of software registers, whose meaning and function are closely related to the INTERBUS master and the mailbox handshake protocol. The number and location of these registers are not predetermined by the hardware, but are managed by the INTERBUS master.

Information about the available registers and their locations is stored in the MPM descriptor, which is created for each node.

Information about the following classifications is stored in the MPM descriptor:

– Data Area (DTA)
– Extended Data Area (ExDTA)
– Mailbox Area (MXA)
– Send vector and acknowledge vector registers

Data area

Each node has a data area, which it uses to exchange data with the other nodes without using a protocol. The start address and length of the node data area are specified in the MPM descriptor. Here, each node can also find the start addresses of the data areas for other nodes to which it has read access here.

Extended data area

The extended data area is similar in structure to the "standard" data area. The entries in the MPM descriptor for the extended data area also contain information about its start address and its length.

Mailbox area

The entries relating to the mailbox area also contain information about its start address in the MPM and its length. They do not contain any information about the division of the mailbox area into individual mailboxes. This division is left entirely to the user or creator of the device driver.

Send vector and acknowledge vector registers

A total of eight registers are assigned to each node: four send vector registers and four acknowledge vector registers. Each node has its own register at different addresses. The addresses of the send vector and acknowledge vector registers are entered in the MPM descriptor. These defaults are obligatory and must not be changed by the user.

Sub node register

The sub node can be used for additional addressing of various applications of **one** node.

2.3.5.1 Structure of the MPM Descriptor

The MPM descriptor is generated by the MPM manager for every node and is stored in the MPM. Each node uses its slot number to determine where its valid MPM descriptor is stored.

Address of the MPM descriptor for node 0:
3200_{hex} + (node no. * 320_{hex}) + 240_{hex}

Address of the MPM descriptor for nodes 1, 2, and 3:
3200_{hex} + (node no. * 320_{hex}) + $26A_{hex}$

This equation can be used for each node to calculate and evaluate the address of the MPM descriptor. This task is generally carried out by the device driver during initialization.

Start Addresses of the MPM Descriptor for the Nodes:

- Node 0: 3440_{hex}
- Node 1: $378A_{hex}$
- Node 2: $3AAA_{hex}$
- Node 3: $3DCA_{hex}$

The application program (host CPU) is defined as node 0 for IBS PCI controller boards. The INTERBUS master is implemented as node 1.

Implementation of the MPM Descriptor:

```
typedef struct {
     USIGN16   startAddrDTA;      /* Start address of the data area                    */
     USIGN16   lengthDTA;         /* Length of the data area in bytes                  */
     USIGN16   startAddrExDTA;    /* Start address of the extended data area           */
     USIGN16   lengthExDTA;       /* Length of the extended data area in bytes         */
     USIGN16   startAddrMXA;      /* Start address of the mailbox area                 */
     USIGN16   lengthMXA;         /* Length of the mailbox area in bytes               */
     T_DTA_ENTRY DATA[2][4]       /* Division of the data area                         */
     USIGN16   SVR[2][4];         /* Addresses of the send vector registers            */
     USIGN16   AVR[2][4];         /* Addresses of the acknowledge vector registers
                                  */
     USIGN16   SNR[2][4];         /* Addresses of the subnode registers                */
} T_NODE_INFO;

typedef struct {
     USIGN16   startAddrDTA;      /* Start address of a DTA subrange                   */
     USIGN16   lengthDTA;         /* Length of a DTA subrange                          */
} T_DTA_ENTRY;
```

The MPM descriptor only contains the start addresses (offsets) of the different areas and vector registers.

MPM Descriptor Node 0 (3440hex):

```
startAddrDTA        0000 -> length 1024
startAddrExDTA      5C00 -> length 5120
startAddrMXA        4000 -> length 7168
data[0][0] startAddrDTA      FFFF -> length 0000
data[0][1] startAddrDTA      0000 -> length 1024
data[0][2] startAddrDTA      0000 -> length 1024
data[0][3] startAddrDTA      0000 -> length 1024
data[1][0] startAddrDTA      FFFF -> length 0000
data[1][1] startAddrDTA      1000 -> length 1024
data[1][2] startAddrDTA      1000 -> length 1024
data[1][3] startAddrDTA      1000 -> length 1024
/*   Node            0     1    2    3    */
svr[0][0-3] ->      0000 342A 342C 0000
                         /* SVR sending nodes 0-3 */

svr[1][0-3] ->      0000 3772 3A92 0000
                         /* SVR receiving nodes 0-3 */

avr[0][0-3] ->      0000 3432 3434 0000
                         /* AVR sending nodes 0-3 */

avr[1][0-3] ->      0000 377A 3A9A 0000
                         /* AVR receiving nodes 0-3 */

snr[0][0-3] ->      0000 343A 343C 0000
                         /* SNR sending nodes 0-3 */

snr[1][0-3] ->      0000 3782 3AA2 0000
                         /* SNR receiving nodes 0-3 */
```

MPM Descriptor Node 1 (378A$_{hex}$):

```
startAddrDTA        1000 -> length 1024
startAddrExDTA      8C00 -> length 5120
startAddrMXA        7000 -> length 7168
data[0][0] startAddrDTA      1000 -> length 0000
data[0][1] startAddrDTA      FFFF -> length 1024
data[0][2] startAddrDTA      0400 -> length 1024
data[0][3] startAddrDTA      1C00 -> length 1024
data[1][0] startAddrDTA      0000 -> length 0000
data[1][1] startAddrDTA      FFFF -> length 1024
data[1][2] startAddrDTA      1400 -> length 1024
data[1][3] startAddrDTA      0C00 -> length 1024
/*   Node           0      1    2    3    */
svr[0][0-3] ->      3772 0000  37760000
                         /* SVR sending nodes 0-3 */

svr[1][0-3] ->      342A 0000 3A94 0000
                         /* SVR receiving nodes 0-3 */

avr[0][0-3] ->      377A 0000 377E 0000
                         /* AVR sending nodes 0-3 */

avr[1][0-3] ->      3432 0000 3A9C 0000
                         /* AVR receiving nodes 0-3 */

snr[0][0-3] ->      3782  00003786 0000
                         /* SNR sending nodes 0-3 */

snr[1][0-3] ->      343A 0000 3AA4 0000
                         /* SNR receiving nodes 0-3 */
```

2.3.5.2 DTA, Extended DTA, and MXA

For the data area, extended data area, and mailbox area, the start address and the length in bytes are indicated. Within these areas, the corresponding node has write access.

The data area is further subdivided into subranges set aside for communication with the other nodes. The location and number of these subranges are stored in the two-dimensional *Data* array.

Table 2-19 Assignment of the array elements

Array	Assignment
Data[0][0]	Data area node n to node 0
Data[0][1]	Data area node n to node 1
Data[0][2]	Data area node n to node 2
Data[0][3]	Data area node n to node 3
Data[1][0]	Data area node 0 to node n
Data[1][1]	Data area node 1 to node n
Data[1][2]	Data area node 2 to node n
Data[1][3]	Data area node 3 to node n

The extended data area is not further subdivided and it is fully available to the user. The mailbox area is also coherent, but should be divided by the user into several mailboxes. The recommended minimum mailbox size is 1 kbyte. However, division and management are left entirely to the user or the device driver.

2.3.5.3 Send Vector Registers (SVR)

The send vector and acknowledge vector registers are software registers with a data word width of 16 bits. These registers are used for communication via the mailbox interface. The address (offset) of the MPM mailbox that contains the actual message is entered in the send vector register (SVR). The address is calculated as an offset, beginning with the starting point of the MPM (see also "MPM Address Area" on page 2-19).

Every node has an SVR for every other node (including for itself). This makes four SVRs per node.

The two-dimensional array contains both the addresses of the SVRs that the nodes use for sending and the addresses of the SVRs that the other nodes use for communication with it.

Table 2-20 Assignment of the array elements for SVR[ ][ ]

Array	Assignment	Abbreviation
SVR[0][0]	SVR node n to node 0	SVR_n_0
SVR[0][1]	SVR node n to node 1	SVR_n_1
SVR[0][2]	SVR node n to node 2	SVR_n_2
SVR[0][3]	SVR node n to node 3	SVR_n_3
SVR[1][0]	SVR node 0 to node n	SVR_0_n
SVR[1][1]	SVR node 1 to node n	SVR_1_n
SVR[1][2]	SVR node 2 to node n	SVR_2_n
SVR[1][3]	SVR node 3 to node n	SVR_3_n

The abbreviation SVR_0_1 therefore stands for a send vector register that is used to enable nodes 0 and 1 to communicate, in the direction from node 0 to node 1.

2.3.5.4 Acknowledge Vector Registers (AVR)

The acknowledge vector registers are also software registers with a data word width of 16 bits. These registers are used for communication via the mailbox interface. The AVR contains the address of a MPM mailbox that has been read by the communication partner and can therefore be re-enabled. Like the SVR, the address is a 16-bit offset, calculated in relation to the starting point of the MPM. Four AVRs are also available for each node.

The two-dimensional array contains both the addresses of the AVRs that the nodes use for sending and the addresses of the AVRs that the other nodes use for communication with it.

Table 2-21 Assignment of the array elements for AVR[ ][ ]

Array	Assignment	Abbreviation
AVR[0][0]	AVR node n to node 0	AVR_n_0
AVR[0][1]	AVR node n to node 1	AVR_n_1
AVR[0][2]	AVR node n to node 2	AVR_n_2
AVR[0][3]	AVR node n to node 3	AVR_n_3
AVR[1][0]	AVR node 0 to node n	AVR_0_n
AVR[1][1]	AVR node 1 to node n	AVR_1_n
AVR[1][2]	AVR node 2 to node n	AVR_2_n
AVR[1][3]	AVR node 3 to node n	AVR_3_n

The abbreviation AVR_0_1 therefore stands for an acknowledge vector register that is used to enable nodes 0 and 1 to communicate, in the direction from node 0 to node 1.

2.3.5.5 Subnode Register (SNR)

Subnode registers are only required if, for example, an INTERBUS master and an INTERBUS slave operate on the same node (MPM slot). These registers then contain information about the controller board (INTERBUS master or INTERBUS slave) to which the message is to be sent.

Table 2-22 Assignment of the array elements for SNR[ ][ ]

Array	Assignment	Abbreviation
SNR[0][0]	SNR node n to node 0	SNR_n_0
SNR[0][1]	SNR node n to node 1	SNR_n_1
SNR[0][2]	SNR node n to node 2	SNR_n_2
SNR[0][3]	SNR node n to node 3	SNR_n_3
SNR[1][0]	SNR node 0 to node n	SNR_0_n
SNR[1][1]	SNR node 1 to node n	SNR_1_n
SNR[1][2]	SNR node 2 to node n	SNR_2_n
SNR[1][3]	SNR node 3 to node n	SNR_3_n

The subnode registers are not used on IBS PCI controller boards.

Section 3

This section informs you about

– data exchange via the data area (DTA).

3 Data Exchange via the Data Area

The data area (DTA) is used to facilitate data exchange via the MPM. As already mentioned, each node only has write permission within its data area and extended data area, i.e., the sending node writes the data to a certain area within its DTA and the receiving node reads the data from there. To enable data to be exchanged independently between individual nodes, the data area is divided into several subranges belonging exclusively to a destination node. This segmentation is a software solution and has no effect on the MPM hardware architecture. Each data area to a different node is 1 kbyte. Also available is an extended data area, which the user can implement as required. The MPM descriptor specifies the length of this extended data area.

Example 1

Node 0 sends data via the DTA to node 1:

1. Node 0 writes the data to be sent to DTA area *DTA Node 0 to Node 1*, i.e., it starts entering the data at offset 0000_{hex} (the offset address can be determined using the MPM descriptor).
2. Starting at address 0000_{hex}, node 1 reads the data from the DTA of node 0.

Example 2

Data is sent from node 1 to node 0 in the following sequence:

1. Node 1 writes the data to be sent to its DTA area *DTA Node 1 to Node 0*. Working from the start of the MPM, the start address is 1000_{hex}.
2. Starting at MPM address 1000_{hex}, node 0 reads the data from the DTA of node 1 (the offset address can be determined using the MPM descriptor).

As no hardware restrictions are used for data exchange via the DTA, there is always a risk of simultaneous or overlapping read/write access to the same memory location. This means that a data block is already being overwritten by the other node during reading. The reading node, therefore, might receive data from two different write cycles.

Depending on the type of access (read or write), different areas (blocks) are accessed in the MPM. A write access always takes place in the area of the writing node, i.e., if node 0 wants to send data to node 2, it writes the data to its own area. If, however, data is to be read from a different node, access takes place in the corresponding block of the node that has stored the data there. This also enables a node to read its own DTA (see Figure 3-1).

"Asynchronous mode" is not supported by the driver for IBS PCI SC/... controller boards. "Asynchronous mode with synchronization pulse" is used here to prevent any data consistency problems (see Section "Operating Modes" on page 3-5)

Access options when reading from and writing to the DTA (node 0):

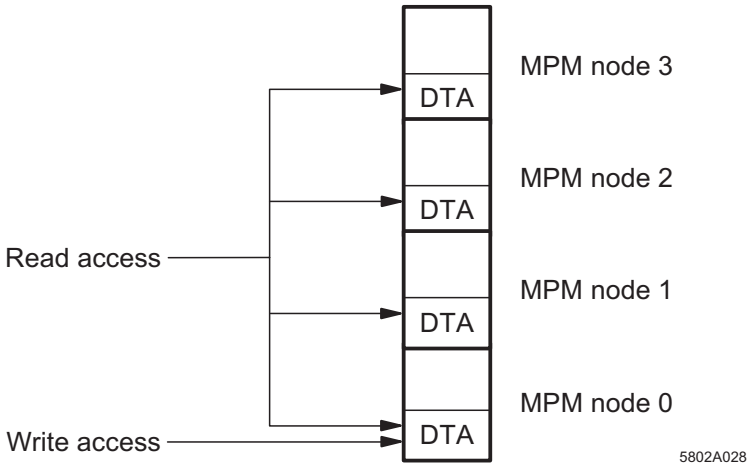


Figure 3-1 Read and write access to the DTA

3.1 Operating Modes

3.1.1 Asynchronous Mode

Asynchronous to the INTERBUS data cycles, a node accesses the I/O data stored in the DTA. This means that a data block is already being overwritten by node 1 (master firmware) during reading (node n). The reading node, therefore, might receive data from two different write cycles, because data from the current INTERBUS data cycle and the previous data cycle can be accessed. This can result in an inconsistency in the read data. Consequently, this operating mode is not the standard operating mode for the controller board and driver. Figure 3-2 ”Asynchronous access of the application to INTERBUS data” shows 2 critical points, which illustrate the problem of "simultaneous access".

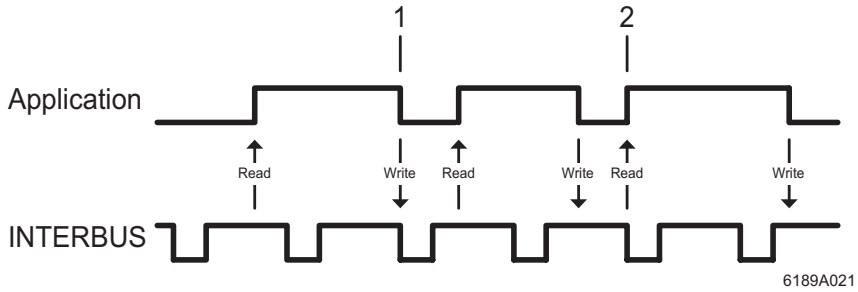


Figure 3-2 Asynchronous access of the application to INTERBUS data

3.1.2 Asynchronous Mode With Synchronization Pulse

Node 1 (master firmware) generates a synchronization pulse after the completion of the INTERBUS cycle. From this point on, INTERBUS process data can be transmitted consistently from, e.g., node 0. The next INTERBUS cycle is started by a synchronization pulse generated from node 0.

This operating mode is set as default by the master firmware and also supported by the driver. Using this operating mode, INTERBUS I/O data is consistently transmitted.

3.1.2.1 Asynchronous With Synchronization Pulse (Implementation in the Driver)

3 buffers (IN and OUT) are implemented in the driver, which ensure that any access by an application, which is always asynchronous, is made to the most recent IN buffer and to the next free OUT buffer. The driver controls which buffer is made available to the application.

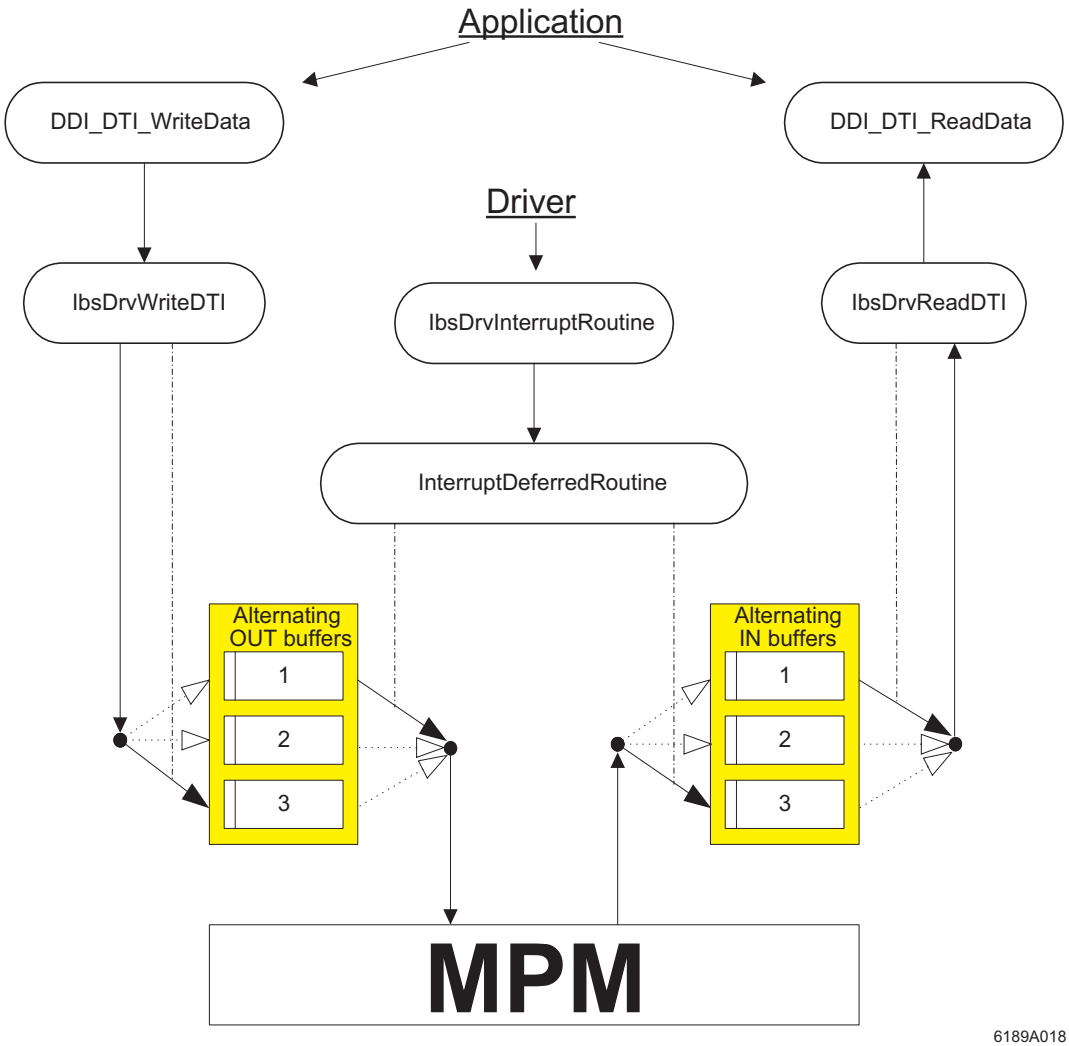


Figure 3-3 Asynchronous with synchronization pulse (implementation in the driver)

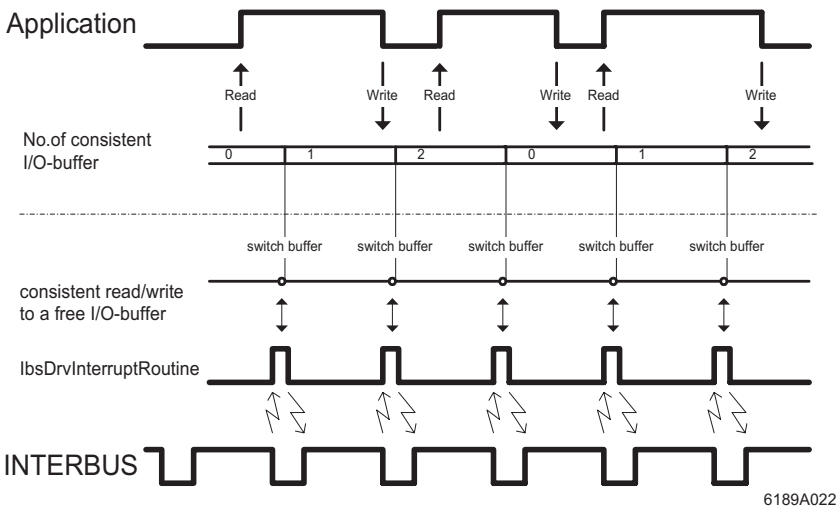


Figure 3-4 Asynchronous, consistent access of the application to INTERBUS data

Section 4

This section informs you about

– data exchange via the mailbox interface (MXI).

4 Communication via the Mailbox Interface

All messages (including commands) are generally exchanged between the individual nodes via the mailbox interface (MXI). This involves a peer to peer exchange process. A message can be sent from any node to any other node.

The messages are transferred to mailboxes, each of which represents a memory area within the MPM. The structure of these messages corresponds to the mailbox syntax specified by the INTERBUS master.

To ensure messages are transmitted correctly, the hardware and software registers in the MPM are used to provide a handshake protocol. Both the presence of a message in a mailbox and a mailbox becoming free in the MPM are signaled to the other MPM nodes.

The following MPM registers are used during handshaking:

– Set HS Ax register and set HS Bx register (hardware registers)
– Handshake registers A and B (hardware registers)
– Status register (hardware register)
– Send vector register (software register)
– Acknowledge vector register (software register)

The 32 handshake registers are hardware registers displayed in the MPM address area (see also "Handshake Register A and Handshake Register B" on page 2-35 and "The Set HS Ax Register and the Set HS Bx Register" on page 2-37). A distinction is made between "Message present" and "Mailbox free" in these registers.

Assignment of the individual bits of handshake registers A and B, and the 32 set HS Ax/Bx registers to the individual nodes and their meanings:

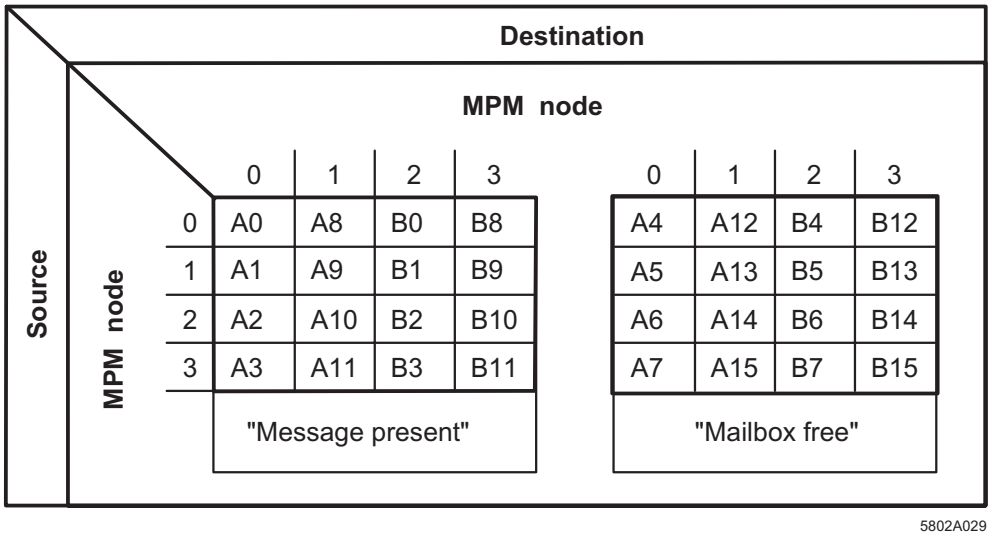


Figure 4-1 Assignment of the handshake bits

Key for Figure 4-1:

Source	Node that wants to send a message
Destination	Node that should receive this message
"Message present"	Message ready in a mailbox
"Mailbox free"	Message has been read from the mailbox and the mailbox is free again

MPM status register

In status register 1, the *HS_Int_x* bits should be evaluated to determine which node has reported something.

MPM handshake registers A and B

Handshake registers A and B summarize the 32 set HS Ax/Bx registers. These two registers can be evaluated to precisely determine the node and the type of message ("Message present" or "Mailbox free"). The bits in the two registers also indicate whether a send vector register or an acknowledge vector register is occupied (bit = 1) or free (bit = 0).

Send vector register (SVR)

The address (offset) of the MPM mailbox that contains the actual message is entered in the send vector register. The address is calculated as an offset, beginning with the starting point of the MPM (see also "Multi-Port Memory (MPM)" on page 2-16).

Acknowledge vector register (AVR)

The acknowledge vector register contains the address of the MPM mailbox that has been read by the communication partner and can therefore be re-enabled. Like the SVR, the address is a 16-bit offset, calculated in relation to the starting point of the MPM. Four AVR registers are available for each MPM node.

The process involved and the use of the different registers will now be explained in greater detail using two examples.

Example 1

Send message from node 0 to node 1:

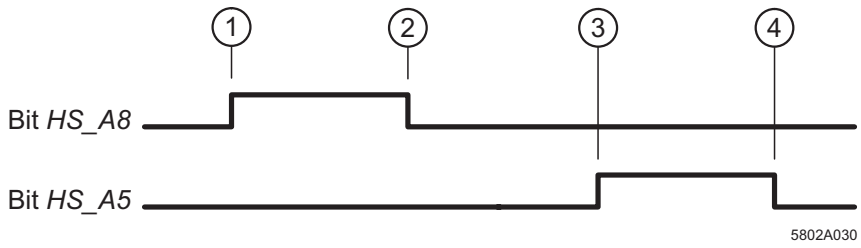


Figure 4-2 Timing diagram for handshake bits

The above diagram shows the timing and status of the handshake bits involved when a message is sent from node 0 to node 1.

The mailbox handshake protocol used for this has four significant events:

1. **"Message present"**:
   Node 0 sets bit *HS_A8* (meaning: message present; source = node 0; destination = node 1). This indicates that there is a message from node 0 to node 1. Its address has been entered by node 0 in the relevant send vector register (SVR [0] [1]).
2. **"Message present detected"**:
   Node 1 resets bit *HS_A8*. It has detected the presence of a message and read the mailbox address from the send vector register (SVR [0] [1]). The SVR can then be reused by node 0.
   Resetting bit *HS_A8* does not mean that the mailbox is free.
3. **"Mailbox free"**:
   Node 1 sets bit *HS_A5* (meaning: message free; source = node 1; destination = node 0). In this way, node 1 informs node 0 that it has read the message from the mailbox and the mailbox is now free. The mailbox address is in the acknowledge vector register (AVR [1] [0]).
4. **"Mailbox free detected"**:
   Node 0 resets bit *HS_A5*. In this way, node 0 informs node 1 that the mailbox is free. At the same time node 1 recognizes that it can reuse the acknowledge vector register (AVR_1_0).

As explained above, the handshake bits are set by writing a data item with the most significant bit set (bit = 1) to the corresponding set HS Ax/Bx registers. Conversely, the handshake bits are reset by writing a data item with the most significant bit reset (bit = 0) to the corresponding set HS Ax/Bx registers. Handshake registers A and B can be read to determine which handshake bits are set or reset.

Setting a handshake bit generally leads to an interrupt at the destination node. This offers the opportunity of enabling the mailboxes in an interrupt routine. The send vector registers can also be read within the interrupt routine.

Detailed Process for Transmitting a Message From Node 0 to Node 1:

Node 0

1. Node 0 checks whether send vector register "SVR [0] [1]" is free. To do this it reads MPM handshake register A and checks whether bit *HS_A8* is zero. If the bit is set, the SVR is occupied and cannot be used.
2. The mailbox address (offset) is entered in the SVR in the MPM.
3. The value 8000_{hex} is written to the set HS A8 register (thereby initiating an interrupt at node 1).

Node 1 (interrupt routine)

1. Node 1 uses the set bit *HS_Int_1* in the MPM status register to detect that the cause of the interrupt is a mailbox handshake protocol interrupt.
2. It reads handshake register A to determine which node initiated the interrupt. From the bit set there, *HS_A8*, the receiver (node 1) recognizes it as a message from node 0. At the same time it recognizes that a "Message present" event is involved (see also Figure 2-18 "Bit assignment for status register 1").
3. Node 1 determines the SVR address of the sender (SVR [0] [1]), reads the register and "notes" its contents, which represent the address of the mailbox with the message. (The address is in Motorola format.)
4. It resets the bit in handshake register A by writing the value 0000_{hex} to the set HS A8 register.
5. Having "noted" its address, the receiver (node 1) can now read the mailbox. As soon as the message has been copied from the mailbox, it should inform the sender (node 0) that the mailbox is free and can be reused.

Node 1 (routine for reading the message)

1. Node 1 determines whether the acknowledge vector register is free. To do this it reads handshake register A and checks whether bit *HS_A5* is zero. If it is not, it waits until bit *HS_A5* has been reset or outputs an error message. Otherwise, it continues with step 2.
2. Node 1 enters the mailbox address in the AVR (AVR [1] [0]) from which the message has been read.
3. Node 1 writes the value 8000_{hex} to the set HS A5 register, initiating an interrupt at node 0. This indicates to node 0 that the mailbox is free again.

Node 0 (interrupt routine)

1. Node 0 uses the set bit *HS_Int_0* in status register 1 to detect that the cause of the interrupt is a mailbox handshake protocol interrupt.
2. It reads handshake register A to determine which node initiated the interrupt. At the same time it recognizes that a "Mailbox free" event is involved.
3. Node 0 determines the AVR address of the receiver (AVR [1] [0]), reads the register and "notes" its contents (address of the mailbox that has become free again).
4. It resets the bit *HS_A5* in handshake register A by writing the value 0000_{hex} to the set HS A5 register.
5. In its internal mailbox management system, it marks the mailbox as free.

Example 2:

Send message from node 1 to node 0:

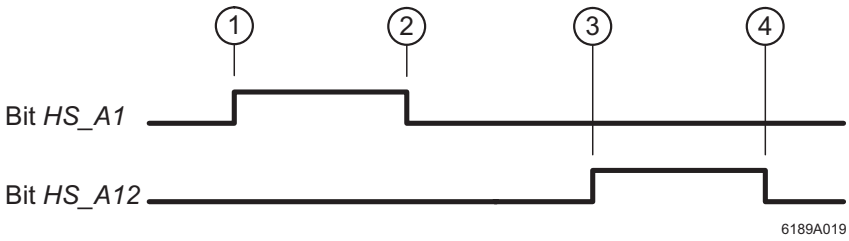


Figure 4-3 Timing diagram for handshake bits

The above diagram shows the timing and status of the handshake bits involved when a message is sent from node 1 to node 0.

The mailbox handshake protocol used for this has four significant events:

1. "**Message present**":
   Node 1 sets bit HS_A1 (meaning: message present; source = node 1; destination = node 0). This indicates that there is a message from node 1 to node 0. Its address has been entered by node 1 in the relevant send vector register (SVR [1] [0]).
2. "**Message present detected**":
   Node 0 resets bit HS_A1. It has detected the presence of a message and read the mailbox address from the send vector register (SVR [1] [0]). The SVR can then be reused by node 1.
   Resetting bit HS_A1 does not mean that the mailbox is free.

3. "**Mailbox free**":
   Node 0 sets bit HS_A12 (meaning: message free; source = node 0; destination = node 1). In this way, node 0 informs node 1 that it has read the message from the mailbox and the mailbox is now free. The mailbox address is in the acknowledge vector register (AVR [0] [1]).
4. "**Mailbox free detected**":
   Node 1 resets bit HS_A12. In this way, node 1 informs node 0 that the mailbox is free. At the same time node 0 recognizes that it can reuse the acknowledge vector register (AVR [0] [1]).

As explained above, the handshake bits are set by writing a data item with the most significant bit set (bit = 1) to the corresponding set HS Ax/Bx registers. Conversely, the handshake bits are reset by writing a data item with the most significant bit reset (bit = 0) to the corresponding set HS Ax/Bx registers. Handshake registers A and B can be read to determine which handshake bits are set or reset.

Setting a handshake bit leads to an interrupt at the destination node. This offers the opportunity of enabling the mailboxes in an interrupt routine. The send vector registers can also be read within the interrupt routine.

Detailed Process for Transmitting a Message From Node 1 to Node 0:

Node 1

1. Node 1 checks whether the send vector register SVR [1] [0] is free. To do this it reads MPM handshake register A and checks whether bit HS_A1 is zero. If the bit is set, the SVR is occupied and cannot be used.
2. The mailbox address (offset) is entered in the SVR in the MPM.
3. The value 8000hex is written to the set HS A1 register (thereby initiating an interrupt at node 1).

Node 0 (interrupt routine)

1. Node 0 uses the set bit HS_Int_0 in MPM status register 1 to detect that the cause of the interrupt is a mailbox handshake protocol interrupt.
2. It reads handshake register A to determine which node initiated the interrupt. From the bit set there, HS_A1, the receiver (node 0) recognizes it as a message from node 1. At the same time it recognizes that a "Message present" event is involved (see also Figure 2-18 on page 2-27).
3. Node 0 determines the SVR address of the sender (SVR [1] [0]), reads the register and "notes" its contents, which represent the address of the mailbox with the message. (The address is in Motorola format.)
4. It resets the bit in handshake register A by writing the value 0000hex to the set HS A1 register.
5. Having "noted" its address, the receiver (node 0) can now read the mailbox. As soon as the message has been copied from the mailbox, it must inform the sender (node 1) that the mailbox is free and can be reused.

Node 0 (routine for reading the message)

1. Node 0 determines whether the acknowledge vector register is free. To do this it reads handshake register A and checks whether bit HS_A12 is zero. If it is not, it waits until bit HS_A12 has been reset or outputs an error message. Otherwise, it continues with step 2.
2. Node 0 enters the mailbox address in the AVR (AVR [0] [1]) from which the message has been read.
3. Node 0 writes the value 8000hex to the set HS A12 register, initiating an interrupt at node 1. This indicates to node 1 that the mailbox is free again.

Node 1 (interrupt routine)

1. Node 1 uses the set bit HS_Int_1 in status register 1 to detect that the cause of the interrupt is a mailbox handshake protocol interrupt.
2. It reads handshake register A to determine which node initiated the interrupt. At the same time it recognizes that a "Mailbox free" event is involved.
3. Node 1 determines the AVR address of the receiver (AVR [0] [1]), reads the register and "buffers" its contents (address of the mailbox that has become free again).
4. It resets bit HS_A12 in handshake register A by writing the value 0000hex to the set HS A12 register.
5. In its internal mailbox management system, it marks the mailbox as free.

Tips and Notes

As the above examples show, the process of exchanging a message between two nodes via the mailbox interface is split into two phases. In the first, the receiver is informed that there is a message. In the second, the receiver informs the sender that the mailbox used is free again. Each phase requires its own set HS Ax/Bx, send vector, and acknowledge vector registers.

The two events do not have to be processed using interrupt routines. They can also be detected by cyclically checking (polling) the status register and the handshake registers. However, a solution with interrupts should be given preference, since it creates much less load on the system.

Although the protocol may appear complex, it provides a faster and more reliable method of exchanging messages between the individual nodes. It also enables the use of several mailboxes for each node, provided the driver has functions for managing multiple mailboxes.

Section 5

This section informs you about
– the basic structure of a device driver
– the creation of necessary functions.

5 Architecture and Structure of a Device Driver

The previous sections only describe the hardware used. In the following, the basic structure of the driver software for the IBS PCI controller boards will be described with reference to examples. The resulting interface corresponds to the specified device driver interface (DDI).

5.1 General

To cover the range of driver applications, the description of the device driver for the IBS PCI controller boards is simplified by limiting it to reading and writing data using the relevant protocol. If possible, the parameters transferred are also checked for validity and plausibility. The structure and design of the device driver interface enable relatively problem-free porting to different operating systems. Device drivers for various operating systems have already been created using the basic structure described below. Each of these drivers has the same interface conforming to the DDI specification, and is supplemented by some additional functions for the particular operating system. However, the basic functions are always identical, thus enabling the easy porting of application programs.

The basic tasks of a driver can be defined as follows:

– Initializing the INTERBUS controller board
– Reading and writing data
– Using the mailbox to send and receive messages (services)
– Checking the validity and plausibility of the parameters transferred
– Transferring INTERBUS diagnostic data to the application program.

5.2 Basic Structure of the Driver

A device driver for the INTERBUS controller boards for all operating systems supported by Phoenix Contact is based on seven elementary functions:

– DDI_DevOpenNode (openDevice)
– DDI_DevCloseNode (closeDevice)
– DDI_DTI_WriteData (writeDTI)
– DDI_DTI_ReadData (readDTI)
– DDI_MXI_SndMessage (writeMXI)
– DDI_MXI_RcvMessage (readMXI)
– DDI_DevIOCtrl (devIOCtrl)

These functions are used to access all of the functions provided by the driver. The drivers in the core are accessed by the application program either directly via the DDI functions or via the standard "open", "close", "read", "write", and "devIOCtrl" operating system functions.

In addition to the above functions, the driver contains three important routines that are not directly accessible to the user.

– Init device driver routine (initDDI)
– Interrupt deferred routine (intrServiceFunction)
– Release device driver routine (releaseDDI)

The init device driver routine must be called when the driver is started or loaded. The interrupt service routine, however, implements an important part of the mailbox handshake protocol under interrupt control, but remains completely invisible to the user. The release device driver routine must be called when the driver is stopped.

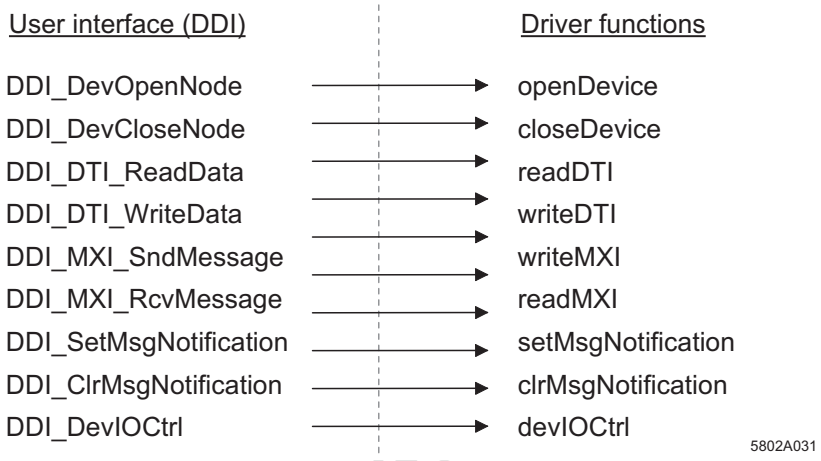


Figure 5-1 Integrating the device driver into any operating system

The device driver can be divided into two distinct parts:

– The actual driver with its functions for accessing the hardware (this part can be fully integrated into the core of the operating system)
– A small library for adapting the calls to the actual driver functions according to the DDI specification

When porting to operating systems such as UNIX, the library with the function calls according to the DDI specification can contain an adaptation to the standard calls ("open", "close", "read", "write", etc.) (see Figure 5-2).

Example of Integration Into an Operating System Such as UNIX

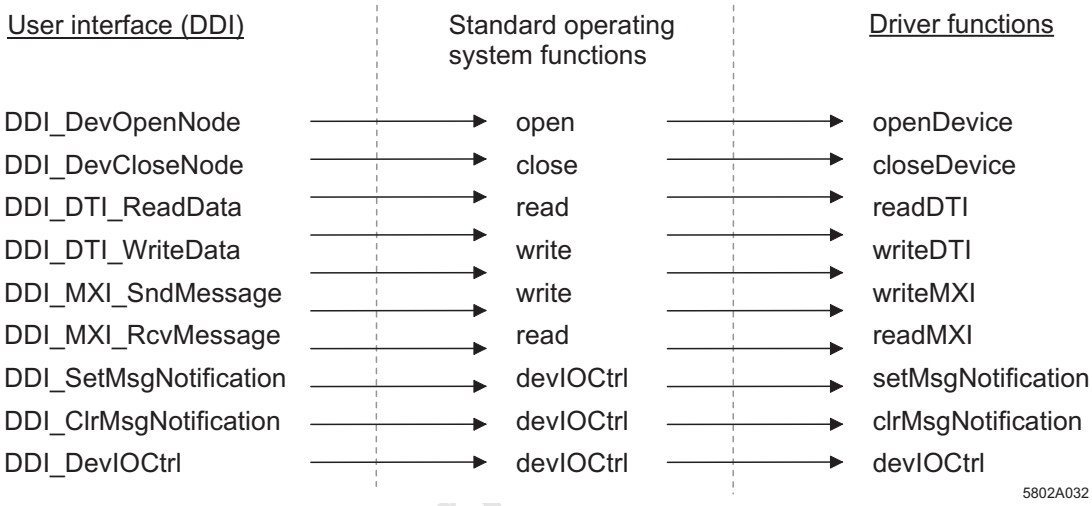


Figure 5-2 Integrating the device driver under UNIX

Example of Integration Into an Operating System Such as Windows NT

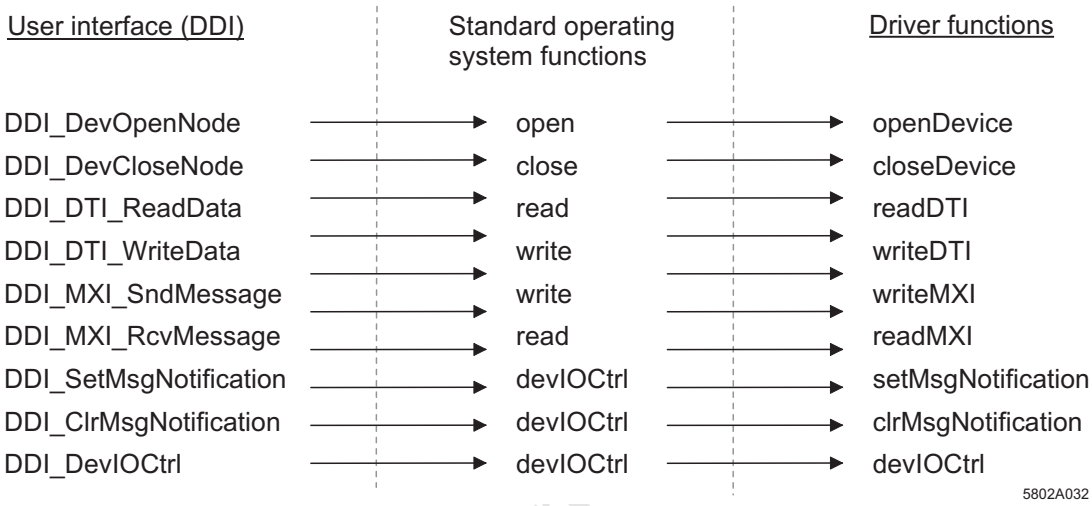


Figure 5-3 Integrating the device driver under Windows NT

5.3 Description of Functions

Before considering each of these functions in greater detail in the following sections, the basic tasks they perform are outlined below.

openDevice Opens a data channel to the mailbox or data interface. The node handle obtained is required for all further services.

closeDevice Closes the data channel to a node, preventing further access via this channel.

writeDTI Writes data to the data area of the IBS PCI controller board MPM.

readDTI Reads data from the MPM data area.

writeMXI Sends a message via the mailbox interface or mailbox area to another MPM node, following the mailbox handshake protocol.

readMXI Reads a message from another MPM node from the mailbox interface or the mailbox area of the MPM, following the mailbox handshake protocol.

devIOCtrl Used to perform all services that are not provided by the functions listed above. This includes setting and resetting notification mechanisms, reading and writing the entire MPM, reading the INTERBUS master diagnostic words, etc. Some of the services, e.g., notification handling and reading diagnostic words, are also made available to the user. Other services are "hidden" from the user.

Init device driver routine This function initializes the device driver structures and variables and performs the basic initialization of the IBS PCI controller boards.

Interrupt service routine

The interrupt service routine performs the majority of tasks needed for the mailbox handshake protocol. It receives the "Message present" and "Mailbox free" events, i.e., it also manages the mailbox. Furthermore, the interrupt service routine receives the SysFail and MPM node SG int x interrupts and carries out preprocessing for these events.

Release device driver routine

The release device driver routine closes the driver.

5.3.1 Initialization (initBoard)

Initialization depends on the operating system used and takes place when the computer or the operating system is started. However, it can also take place during normal operation. Initialization must always be completed successfully before other driver functions can be used by application programs.

Actions during the initialization phase:

– Checking that the board parameters transferred (e.g., short page address, interrupt level, and MPM memory address) are valid and that there is a controller board at the address specified.
– Initialization of the device driver internal structures and variables, evaluation of the MPM descriptor and transfer of the information it contains to internal structures.
– Entering basic settings for the IBS PCI controller boards.

Initialization will be aborted with an error message if an error occurs in one of the actions listed above. In this case, the driver cannot be used.

5.3.2 Open Data Channel (openDevice)

The openDevice function is used to set up a data channel for a particular IBS PCI controller board, a particular node (0 to 3), and a particular interface (data interface or mailbox interface). The device name determines which controller board, which node, and which interface is addressed. It is structured as follows:

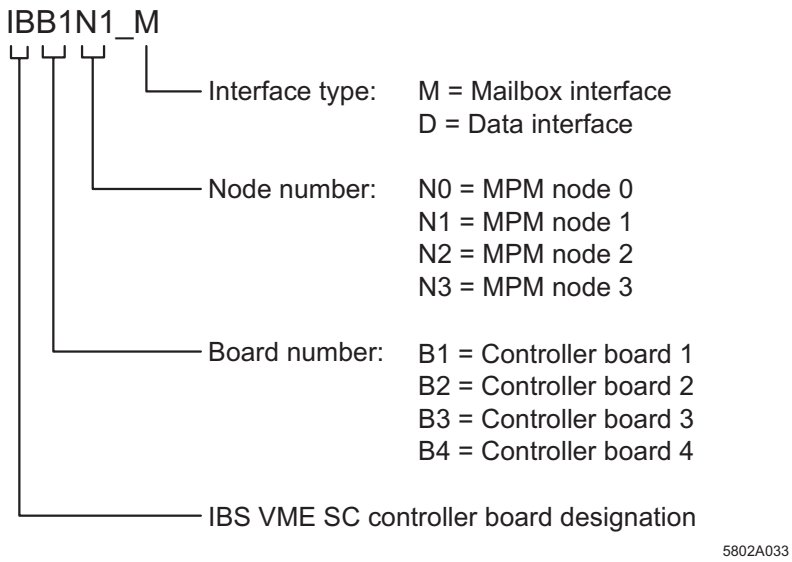


Figure 5-4 Device name structure

From the device name, the openDevice function determines the node to which the data channel is to be opened. The *Board_Pres_x* bit in the MPM configuration register is evaluated to check whether the desired node actually exists. The parameters transferred to the openDevice function, such as access right, interface type, etc., are also checked. If an error occurs in one of these steps or the device name specified is not known, the function is terminated with an appropriate error message.

If all steps up to this point have been completed successfully, the function calls another routine to generate a node handle, which is then transferred to the user. If there are no more node handles available, the function is terminated with an appropriate error message.

A selection of error messages can be found in the "ddi_err.h" file on the disk containing the C source code supplied with the DDK.

The node handle is an index in the *nodeHdCtrl* array (T_NODE_HD_CTRL data type). To avoid having to start with the index value zero, a fixed offset is added to the actual index value. This value (index + offset) is then transferred to the user as the node handle. The node handle does not enable the user to draw any conclusions about the node addressed and type of interface. The *openDevice* function stores all of the parameters belonging to the node handle, such as access right, node number, and interface type, in the *nodeHdCtrl* control structure (T_NODE_HD_CTRL data type). These parameters are then used for subsequent access with the node handle for test purposes and "determining the routing".

The assignment and management of node handles can be reduced or completely eliminated in operating systems that issue their own handle or file descriptor (e.g., UNIX). In these cases, the only check made is for the presence of the desired node. However, as already indicated, the procedure depends on the operating system used.

5.3.3 Close Data Channel (closeDevice)

The closeDevice function is the counterpart of the openDevice function. It is used to close an open data channel again, so that no more services can be executed via this channel. The corresponding *nodeHdCtrl* node handle control structure (T_NODE_HD_CTRL data type) is marked "free" and is therefore available again. Errors can only occur if the node handle transferred is invalid, i.e., its value is outside the valid range, or the data channel has already been closed.

This function has no other effect on the device driver or controller board.

5.3.4 Write Process Data (writeDTI)

The writeDTI function is used to write data to the required location in the data area (DTA) of the local node (in the host system, for example, this would be the data area of node 0). With firmware 4.xx the length of the data area is limited to a maximum of 4 kbytes. The *writeDTI* function basically checks the parameters transferred in the T_DDI_DTI_ACCESS structure and copies the data to the required data consistency area.

The following checks are made:

– Status of the control structure
– Existence of access right to the data channels
– Interface type is "data interface"
– Compliance with permissible data area when writing data (sum of offset and number of bytes less than or equal to data area)

If one of these requirements cannot be met, the function is terminated with an error message.

If all of the requirements are met, the data is copied. The DTA start address of the writing node is determined first. The DTA start address in the MPM depends on the MPM node and is stored in the MPM descriptor. The data is then written to the MPM in a loop.

After the data has been copied successfully, the function is terminated with the return value *ERR_OK*.

5.3.5 Read Process Data (readDTI)

The process involved for reading from the data area corresponds to that of writing to the DTA. Only the data transfer direction changes.

The readDTI function is used to read data from the data area (DTA) of another node or the local node. The node handle determines the node from which data is read.

The function basically checks the parameters transferred in the T_DDI_DTI_ACCESS structure and copies the data from the required area to a memory area provided by the application program.

The following checks are made:

– Status of the control structure
– Existence of access right to the data channels
– Interface type is "data interface"
– Compliance with permissible data area when reading data (sum of offset and number of bytes less than or equal to data area)

If one of these requirements cannot be met, the function is terminated with an error message.

If all of the requirements are met, the data is copied. Once again the start address should be taken from the MPM descriptor. The data is then read from the MPM in a loop.

After the data has been read successfully, the function is terminated with the return value *ERR_OK*.

5.3.6 Send Message (writeMXI)

The writeMXI function is used to send messages to any other MPM node. Unlike writing to the DTA, transmission of the message must follow a certain protocol. The mailbox handshake protocol used has already been described in Section 4, "Communication via the Mailbox Interface", and will not be explained in further detail at this point.

As with the other functions, the parameters transferred and the status of the controller board are checked:

– Validity of the node handle used
– Write access to the data channels permitted
– Interface type is "mailbox interface"
– MPM still displayed in host system memory area and enabled
– Message does not exceed the maximum permissible length of a mailbox
– Destination node is marked "ready", i.e., the corresponding *MPM_Node_Ready_x* bit in status register 1 is set.

All of the following steps result from the mailbox protocol used:

– Check whether the send vector register is free, either by reading the handshake registers in the MPM, or by reading the *svrState* structure component in the relevant node control structure. If the SVR is free, it will then be marked as occupied.
– Determine whether the SVR has already been occupied for a long time. (Check SVR timeout counter for the node). If it has, the function is terminated with the error message *ERR_SVR_TOUT*.
– Use a help function to identify a free mailbox. If there are no more free mailboxes available, enable SVR and terminate the function with an error message.
– Enter the start address of the mailbox to be used for the message in the SVR.
– Copy message to the mailbox.
– Set corresponding handshake bit by writing to a set HS Ax/Bx register.
– Restore the original contents of the controller board I/O registers and terminate the function with the return value *ERR_OK*.

There is no need to wait for the mailbox to be enabled at this point, since this task is performed by the interrupt service routine. There is also no need to wait for the send vector register to be enabled, because the relevant bit in the handshake register is checked before a message is sent to rule out the risk of overwriting. Any mailbox address within the permissible range (mailbox area) can be chosen for communicating with the other nodes.

The SVR to be used and its location in the MPM are determined using the node handle or the node number.

The addresses of the handshake and set HS Ax/Bx registers can be determined using the globally available *svrCtrl* structure (T_AVR_CTRL data type). The *Source_Node_Number* (row value) and *Destination_Node_Number* (column value) parameters required for this are contained in the *nodeHdCtrl* node handle control structure and the *devCtrlCB* general device driver control structure (T_DEV_CTRL data type).

5.3.7 Read Message (readMXI)

The readMXI function is used for reading a message from the MPM. It is the counterpart of the writeMXI function. In addition to reading the actual message, the function also handles the mailbox handshake protocol.

With DOS drivers this function is generally used in polling mode, i.e., the application program calls this function cyclically to determine whether a message is present. With other operating systems, such as OS-2, it can be implemented as a blocked function, which is to be left waiting (blocked) until a message arrives.

The function is notified of the presence of a message indirectly through the *msgCtrl* message control structure (T_MSG_CTRL data type). If the value of the structure component *msgCnt* is greater than zero, a message is present. A message is always only read by the MPM node specified by the node handle.

As with the other functions, before the message is read, the specified parameters specified (node handle, length of the buffer for the message, etc.) and the status of the controller board are checked:

– Validity of the node handle
– Read access permitted when opening the data channel
– Interface type is "mailbox interface"
– Destination node is marked "ready", i.e., the corresponding *MPM_Node_Ready_x* bit in status register 1 is set.

If the parameters correspond to the values expected, the *msgCtrl* message control structure (T_MSG_CTRL data type) is then evaluated to determine whether a message is present. The chkMailbox function is called if there is no message. This routine is also used by the interrupt service routine. It detects and processes events that affect the mailbox handshake protocol. If, after the chkMailboxes function has been called, the message counter of the message control structure is still zero, the function is terminated with the return value *ERR_NO_MSG*.

If a message is present, a check is made to determine whether the MPM is still displayed on the host system memory area and enabled.

If it is still possible to access the MPM, the address of the mailbox in the MPM is determined. This information is used to calculate the MPM segment to be displayed and the offset within this segment for the message.

Reading the Message:

– Evaluate the message parameter counter (the parameter counter is the second word of the message, see also the *"Firmware Services and Error Messages" User Manual, IBS SYS FW G4 UM E*).
– Calculate the total length of the message in bytes ((parameter counter + 2) * 2) and compare with the length of the message buffer. If the message received is longer than the buffer available, the function is terminated with an error message. In this case, the message is not copied.
– Copy data from the mailbox to the message buffer.
– Enter the length of the message in bytes ((parameter counter + 2) * 2) in the variable referenced by **msgLength*.

The following steps are again identical for both mailbox directions, and are used to execute the mailbox handshake protocol:

– Decrement message counter in the message control structure *msgCtrl*.
– Enter address of the read mailbox in the corresponding acknowledge vector register.
– By writing to the corresponding set HS Ax/Bx register, set the relevant bit in one of the handshake registers, thereby indicating to the sending node that the mailbox is free again.
– Terminate the function with the return value *ERR_OK*.

There is no need to wait for the "Mailbox free" bit to be reset. This task is performed by the sending node. As long as only one mailbox is used between the nodes, a new message cannot be transmitted until the sending node has itself detected that the mailbox is free. There is therefore no risk that a valid value in the AVR will be overwritten. It is only necessary to poll for a free AVR if several mailboxes are used between two nodes.

The addresses of the handshake and set HS Ax/Bx registers can be determined using the globally available *avrCtrl* structure (T_AVR_CTRL data type). The *Source_Node_Number* (row value) and *Destination_Node_Number* (column value) parameters required for this are contained in the *nodeHdCtrl* node handle control structure (T_NODE_HD_CTRL data type) and the *devCtrl* general device driver control structure (T_DEV_CTRL data type).

5.3.8 Interrupt Service Routine (intrServiceFunction)

The interrupt service routine is a central part of the device driver. In particular, parts of the mailbox protocol (e.g., enabling the mailboxes used, receiving messages, reading the send vector registers, etc.) are handled within this routine. The SysFail and MPM node SG int x interrupts are also processed by this routine.

The sequence of the routine is as follows:

– Read the MPM status register to determine interrupt cause(s). Possible causes are: SysFail interrupt, MPM node SG int x interrupt, and handshake interrupt.

– Determine exact cause of interrupt:
  SysFail interrupt: Which node signaled the SysFail?
  --> Evaluate MPM status register
  MPM node SG int x interrupt: Which node caused the interrupt? --> Evaluate MPM status register 1
  Handshake interrupt: Which node caused the interrupt?
  --> Evaluate handshake registers to determine which node has signaled what ("Message present" or "Mailbox free")
– "Message present": Use the node numbers (source and destination) to locate and read the relevant SVR. Then reset the handshake bit ("Message present detected"). Save the contents of the SVR (mailbox address) and increment the message counter for the node. Also call the notification function if there is one and it has been activated.
– "Mailbox free": Use the node numbers (source and destination) to locate and read the relevant AVR. Then reset the handshake bit by writing to the set HS Ax/Bx register ("Mailbox free detected"). Mark the mailbox "free" again in the local control structures.

During the interrupt service routine all other interrupts are disabled. They are only enabled again when the routine is terminated. In the current version, processing of the SysFail interrupt and the MPM node SG int x interrupt is limited to merely recording the events. There are no other actions.

The interrupt service routine is divided into two functions: the actual interrupt service routine, which is entered in the interrupt vector, and an independent function that performs all of the actions needed for the mailbox handshake protocol. The two routines can nevertheless be regarded as functionally related. However, this division also makes it possible to call the routine for the mailbox handshake protocol outside the interrupt service routine (see also implementation of the writeMXI and the readMXI function).

5.3.9 Device IO Control (devIOCtrl)

The devIOCtrl function is used to perform special services that cannot be assigned to the "normal" functions. The possible tasks covered by this function range from additional diagnostics, through reading the MPM, to various utility functions (enable watchdog, trigger watchdog, etc.). The advantage of this method is that only one function is required to access the tasks to be carried out in the INTERBUS master.

Use of the devIOCtrl function for various services:

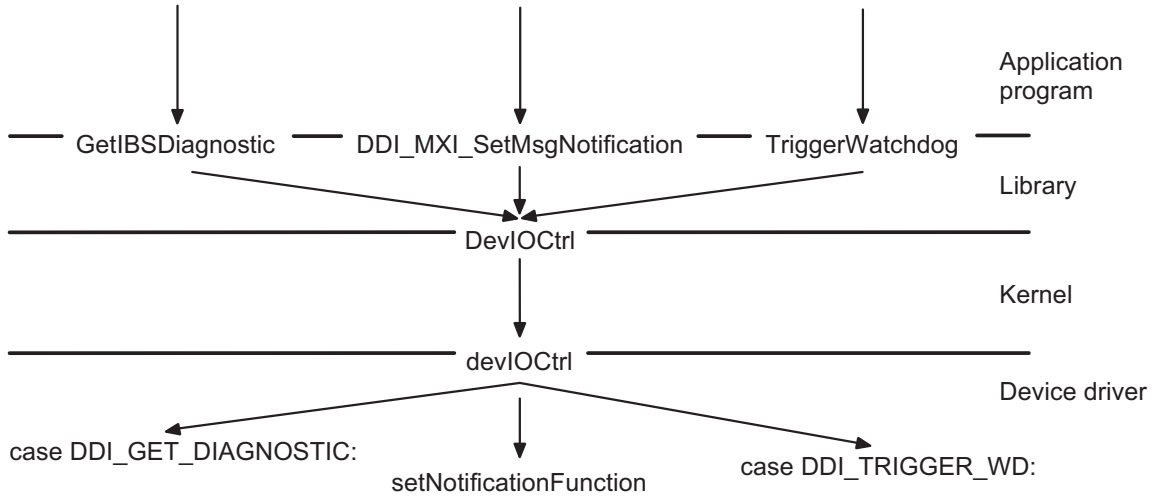


Figure 5-5 devIOCtrl function

The function is provided with three values as parameters:

– Node handle (nodeHd)
– Command code (cmd)
– Pointer to a buffer for reading or writing (dataPtr)

The *nodeHd* parameter is neither checked nor used. Its further use can be defined by the creator of the driver.

The *cmd* (command code) parameter is used to determine which command or special task is to be carried out with the routine. The value must always be unique.

The parameters required for the task or the result of the function are transferred or returned using the *dataPtr* pointer. The type of data to which this pointer refers depends on the desired task and is not restricted.

5.3.10 Utilities

In addition to functions that are a direct mapping of the DDI, some extra utilities are also provided. They are used to read the diagnostic registers, manage the watchdog functions, and set and reset the notification functions. The actual task is implied by the name of the function. See the "vmeutil.c" file for a more detailed description. The following table contains some utilities:

Table 5-1 Additional utilities

Function	Task	Page
GetIBSDiagnostic	Evaluates the operating state of the INTERBUS master	5-22
GetIBSDiagnosticEx	Evaluates the operating state of the INTERBUS master (like *GetIBSDiagnostic*, additional structure element *addinfo*)	5-23
GetSlaveDiagnostic	Evaluates the operating state of the slave	5-24
GetSysFailRegister	Reads the contents of the SysFail register	5-25
EnableWatchDog	Switches a watchdog on	5-26
TriggerWatchDog	Triggers the watchdog	5-26
GetWatchDogState	Reads the status of a watchdog	5-27
ClearWatchDog	Resets the status of a watchdog	5-27
SetWatchDogTimeout	Sets the timeout value and switches the watchdog on	5-28
GetWatchDogTimeout	Reads the timeout value	5-28

5.3.10.1 GetIBSDiagnostic

Task

The *GetIBSDiagnostic()* function reads the diagnostic status register and diagnostic parameter register. The function receives a valid node handle and a pointer to a *T_IBS_DIAG* data structure as parameters. After the function has been called successfully, the components of the structure contain the contents of the diagnostic status register and the diagnostic parameter register.

Syntax

IBDDIRET IBDDIFUNC GetIBSDiagnostic (IBDDIHND nodeHd,
T_IBS_DIAG IBPTR *diagInfo);

Parameter

nodeHd	Node handle (MXI or DTI) of the controller board from which the diagnostic status register and diagnostic parameter register are to be read.
diagInfo	Pointer to a T_IBS_DIAG data structure.
T_IBS_DIAG	The contents of the register are entered in this structure.

Positive acknowledgment:

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

Negative acknowledgment

DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

T_IBS_DIAG **structure elements**

state	The bits of the *state* structure element correspond to the diagnostic status register.
diagPara	The *diagPara* structure element corresponds to the diagnostic parameter register.

Format of the T_IBS_DIAG **structure**

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

5.3.10.2 GetIBSDiagnosticEx

Task

The *GetIBSDiagnosticEx()* function reads the diagnostic status register and diagnostic parameter register. It is used to display the operating state of the INTERBUS controller board or that of INTERBUS. The function corresponds to the *GetIBSDiagnostic()* function. However, the *addInfo* structure element has been added (this is a software register).

Syntax

IBDDIRET IBDDIFUNC GetIBSDiagnosticEx (IBDDIHND nodeHd,
T_IBS_DIAG_EX IBPTR *diagInfo);

Parameter

nodeHd	Node handle (MXI or DTI) of the controller board from which the diagnostic status register and diagnostic parameter register are to be read.
diagInfo	Pointer to a T_IBS_DIAG_EX data structure.
T_IBS_DIAG_EX	This data structure contains all the elements required for diagnostic purposes.

Positive acknowledgment

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

Negative acknowledgment

DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

T_IBS_DIAG_EX **structure elements**

state	The bits of the *state* structure element correspond to the diagnostic status register.
diagPara	The *diagPara* structure element corresponds to the diagnostic parameter register.
addInfo	The *addInfo* structure element corresponds to the *Add_Error_Info* parameter for negative messages.

Format of the T_IBS_DIAG_EX **structure**

```
typedef struct {
    USIGN16 state;      /* Status of INTERBUS      */
    USIGN16 diagPara;   /* Type of error           */
                        /* (controller, user,      */
                           etc.)
    USIGN16 addInfo;    /* Additional information  */
                        /* on the error cause      */
} T_IBS_DIAG_EX;
```

5.3.10.3 GetSlaveDiagnostic

Task

The *GetSlaveDiagnostic()* function reads the slave diagnostic status register and the slave diagnostic parameter register of an INTERBUS system coupler. It is used to display the operating state of the slave part or connected INTERBUS.

Although the *diagPara* structure element is implemented in the T_IBS_DIAG structure, it is not currently supported by the controller board firmware.

Syntax

IBDDIRET IBDDIFUNC GetSlaveDiagnostic (IBDDIHND nodeHd,
T_IBS_DIAG IBPTR *diagInfo);

Parameter

nodeHd	Node handle (MXI or DTI) of the system coupler from which the slave diagnostic status register and slave diagnostic parameter register are to be read.
diagInfo	Pointer to a T_IBS_DIAG data structure.
T_IBS_DIAG	This data structure contains all the elements required for diagnostics purposes.

Positive acknowledgment

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

Negative acknowledgment

DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

T_IBS_DIAG **structure elements**

state	The bits of the *state* structure element correspond to the slave diagnostic status register.
diagPara	The diagPara structure element corresponds to the slave diagnostic parameter register.

Format of the T_IBS_DIAG **structure**

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

5.3.10.4 GetSysFailRegister

Task

The *GetSysFailRegister()* function writes the contents of the MPM status SysFail register to the variable referenced by the *sysFailRegPtr* parameter. Bits 0, 4, 8, and 12 of the register indicate whether the SysFail signal of the controller board is activated or not. If a malfunction occurs (e.g., watchdog triggered) in one of the two bits (8 and 12), the relevant bit is activated in the status SysFail register, i.e., set to one. This bit then remains set until the malfunction is corrected. The individual bits of the register are assigned to the MPM as follows:

– Bit 12: Host system
– Bit 8: INTERBUS controller board

Syntax

```
IBDDIRET IBDDIFUNC GetSysFailRegister (IBDDIHND nodeHd,
                                       USIGN16 IBPTR *sysFailRegPtr);
```

Parameter

nodeHd	Node handle (MXI or DTI) of the controller board from which the SysFail register is to be read.
sysFailRegPtr	Pointer to a variable to which the contents of the SysFail register are to be transferred.

Positive acknowledgment

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

Negative acknowledgment

DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

5.3.10.5 EnableWatchDog

Task

The watchdog is activated with the *DDI_EnableWatchDog()* function. After activation, the watchdog must be reset at regular intervals (TriggerWatchDog).

Once activated, a watchdog can only be deactivated by switching off the host system or by means of a hardware reset.

The default watchdog time is 128 ms.

Syntax

IBDDIRET IBDDIFUNC EnableWatchDog (IBDDIHND nodeHd);

Parameter	nodeHd	Node handle (MXI or DTI) of the controller board from which the watchdog is to be activated.
Positive acknowledgment	ERR_OK (0000_{hex})	The function was executed successfully.
Negative acknowledgment	DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

5.3.10.6 TriggerWatchDog

Task

The watchdog is reset with the *TriggerWatchDog()* function.

Syntax

IBDDIRET IBDDIFUNC TriggerWatchDog (IBDDIHND nodeHd);

Parameter	nodeHd	Node handle (MXI or DTI) of the controller board from which the watchdog is to be reset.
Positive acknowledgment	ERR_OK (0000_{hex})	The function was executed successfully.
Negative acknowledgment	DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

5.3.10.7 GetWatchDogState

Task	The *GetWatchDogState()* function can be used to check whether the watchdog has been triggered (return = 1) or not (return = 0). This is determined by reading the watchdog register. The watchdog is also triggered in the process.	
Syntax	IBDDIRET IBDDIFUNC GetWatchDogState (IBDDIHND nodeHd);	
Parameter	nodeHd	Node handle (MXI or DTI) of the controller board from which the watchdog is to be read.
Positive acknowledgment	Return =1: Return = 0:	Watchdog triggered. Watchdog not triggered.
Negative acknowledgment	DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

5.3.10.8 ClearWatchDog

Task	The watchdog status is reset with the *ClearWatchDog()* function.	
Syntax	IBDDIRET IBDDIFUNC ClearWatchDog (IBDDIHND nodeHd);	
Parameter	nodeHd	Node handle (MXI or DTI) of the controller board from which the watchdog is to be reset.
Positive acknowledgment	ERR_OK (0000_{hex})	The function was executed successfully.
Negative acknowledgment	DDI error code	Describes an error that has occurred. Cause: – Invalid node handle

5.3.10.9 SetWatchDogTimeout

Task

The *SetWatchDogTimeout()* function sets the watchdog timeout time and activates the watchdog. After activation, the watchdog must be reset at regular intervals (TriggerWatchDog).

Once set, a watchdog timeout cannot be modified. An activated watchdog can only be deactivated by switching off the host system or by means of a hardware reset.

Syntax

IBDDIRET IBDDIFUNC SetWatchDogTimeout (IBDDIHND nodeHd,
USIGN16 IBPTR *dataPtr);

Parameter

nodeHd	Node handle (MXI or DTI) of the controller board from which the watchdog timeout is to be set and activated.
dataPtr	Pointer to a variable, which contains the new timeout value.

Positive acknowledgment

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

Negative acknowledgment

DDI error code Describes an error that has occurred.
Cause: – Invalid node handle

5.3.10.10 GetWatchDogTimeout

Task

The *GetWatchDogTimeout()* function reads the current value of the timeout.

Syntax

IBDDIRET IBDDIFUNC GetWatchDogTimeout (IBDDIHND nodeHd,
USIGN16 IBPTR *dataPtr);

Parameter

nodeHd	Node handle (MXI or DTI) of the controller board from which the timeout value is to be read.
dataPtr	Pointer to a variable to which the timeout value is written.

Positive acknowledgment

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

Negative acknowledgment

DDI error code Describes an error that has occurred.
Cause: – Invalid node handle

5.3.11 Data Structures Used

All of the data structures used will now be described and explained in order to provide a clear picture of the driver and its organization.

5.3.11.1 T_DEV_CTRL devCtrl

This structure contains all the parameters, which are of relevance to the controller board. They are as follows:

stateOfDevice:	Logical status of the host system (ready, error, etc.)
boardNumber:	Logical number of the controller board addressed
localNodeNumber:	Node number on the MPM (a "0" represents the driver itself.)
addrMPM:	Controller board memory address in the host system address area.
addrHostRegs:	Host register address
intrVector:	Interrupt vector used by driver
intrLevel:	Interrupt level used by driver
sndMXA:	Mailbox area location and length
startAddrExDTA:	Extended data area start address
lengthExDTA:	Length of the extended data area in bytes
sndMXA.offset...outData.length:	Location and length of the DTA and MXA in the MPM

The structure is configured during the initialization phase. During operation, it is accessed primarily for read purposes.

5.3.11.2 T_NODE_CTRL nodeCtrl[ ]

A *nodeCtrl* structure (T_NODE_CTRL data type) is set up and initialized for each node (maximum of four possible) on the MPM. This structure primarily contains information assigned specifically to one node.

nodeNumber:	Node number for which the structure is used
devNameDTI:	Pointer to the relevant device name for the DTI

devNameMXI:	Pointer to the relevant device name for the MXI
curDTIOpenCnt:	Number of data channels open to the DTI
maxDTIOpenCnt:	Maximum number of data channels to the DTI
curMXIOpenCnt:	Number of data channels open to the MXI
maxMXIOpenCnt:	Maximum number of data channels to the MXI
svr.addrSVRToRem:	Address of the SVR for sending to the relevant node
svr.stateSVRToRem:	SVR status (free or occupied, has no significance at present)
avr.addrAVRToRem:	Address of the AVR for notifying ("Mailbox free") for the relevant node
avr.stateAVRToRem:	AVR status (free or occupied, has no significance at present)
avr.addrAVRFromRem:	Address of the AVR of the relevant node (message to local node)
avr.stateAVRFromRem:	AVR status (free or occupied, has no significance at present)
dta.addrDTAToRem:	Address of the data area to the relevant node
dta.lengthDTAToRem:	Length of the data area to the relevant node
dta.addrDTAFromRem:	Address of the data area from the relevant node
dta.lengthDTAFromRem:	Length of the data area from the relevant node
exDta.addrExDTAToRem:	Address of the extended data area to the relevant node
exDta.lengthExDTAToRem:	Length of the extended data area to the relevant node
exDta.addrExDTAFromRem:	Address of the extended data area from the relevant node
exDta.lengthExDTAFromRem:	Length of the extended data area from the relevant node

5.3.11.3 T_NODE_HD_CTRL nodeHdCtrl[ ]

T_NODE_HD_CTRL data structures are provided based on the total number of node handles available to the user. The number can be adjusted to meet individual requirements. The structures are initialized when the driver is started or loaded. During operation they are occupied or enabled again when the data channels are opened and closed. The structure contains parameters that enable the node handle to be easily assigned to the node it specifies. The structure also contains parameters for the type of data channel and access rights.

stateOfNodeHd:	Node handle status (free, in use, etc.)
nodeNumber:	Number of node on which the handle operates
devType:	Type of interface (mailbox or data)
perm:	Interface access rights (read, write or both)
flagNotify:	Notification for this handle activated

5.3.11.4 T_NOTIFY_CTRL notifyCtrl[ ]

A T_NOTIFY_CTRL data structure is available for each node (maximum of four). The structures are initialized when the driver is started or loaded. During operation these structures are used by the notification functions and the interrupt service routine. The index in the array corresponds to the node number.

nodeHd:	The node handle identifies a data channel open to a node.
flagNotify:	Indicates whether notification has been activated for this node.
notifyFunc:	Address of the function to be called. The call is made from the interrupt service routine.

5.3.11.5 T_MSG_CTRL msgCtrl[ ]

For each node the incoming messages are detected using this structure. In its current form, only one message can be detected with this structure. However, since each node uses only one mailbox, there is no risk of any problems arising.

state:	Indicates whether a message is present or not.
addrOfMsg:	Address (offset) of the message in the MPM.
remNodeNumber:	Number of the node from which the message has been received.

5.3.11.6 T_SVR_CTRL svrCtrl[4][4]

This structure is already initialized at the time of translation. It is used when processing the mailbox handshake protocol to determine the following parameters:

– Addresses of the MPM handshake registers in the MPM
– Addresses of the set HS Ax/Bx registers in the MPM
– Mask for masking out the MPM handshake register (already in Intel format)

Access to this two-dimensional array is based on the row for the source node and the column for the destination node. This enables quick access to the parameters required for the mailbox handshake protocol.

5.3.11.7 T_AVR_CTRL avrCtrl[4][4]

The information given for the *svrCtrl* structure also applies to this structure. The parameters for the acknowledge vector registers are entered here.

5.3.11.8 CHAR devNamesDTI[ ][ ]

This structure contains the strings of all valid device names for the data interface. The device names for each node are determined using the local node number. The pointers of the *nodeCtrl* data structure (T_NODE_CTRL data type) refer to these strings. The strings are created in static form and are not modified.

5.3.11.9 CHAR devNamesMXI[ ][ ]

Contains the strings of all valid device names for the mailbox interface. Otherwise the same comments as for devNamesDTI apply.

5.3.11.10 USIGN16 avrToutCnt[4]

A counter is implemented here for each node of the MPM, containing the number of consecutive unsuccessful attempts to access the AVR for a node. After a certain value has been exceeded, the error message T_AVR_TIMEOUT is returned.

This counter is not in use at present.

5.3.11.11 USIGN16 svrToutCnt[4]

A corresponding counter for the send vector registers. If a specified value is exceeded, the error message ERR_SVR_TIMEOUT appears when the sendMXI function is called.

5.3.11.12 USIGN16 msgCnt[4]

This is an array with the message counters for each node. Incremented when a message is received and decremented when a message is read.

Section 6

This section informs you about
– adaptation to operating systems

6 ToDo (Adaptation to Operating Systems)

6.1 File Structures in the Driver

In order to give an overview of the dependency relationships within the driver, the "file path" for VxWorks is described here as an example. The adaptation for Windows NT is also shown. The example demonstrates the differences between different operating systems and gives a rough idea of which adaptations are necessary for other operating systems. A description of the individual "c" and "h" files can be found in Section 1.1 ”Files on the Disk”.

It is recommended that the file names are selected to correspond to the operating system, as shown here. This enables the file to be clearly identified and retains the link to the structure described here.

The calls and functions described here in relation to Windows NT are only given for the purpose of clarification and are **not** part of the IBS PCI DDK.

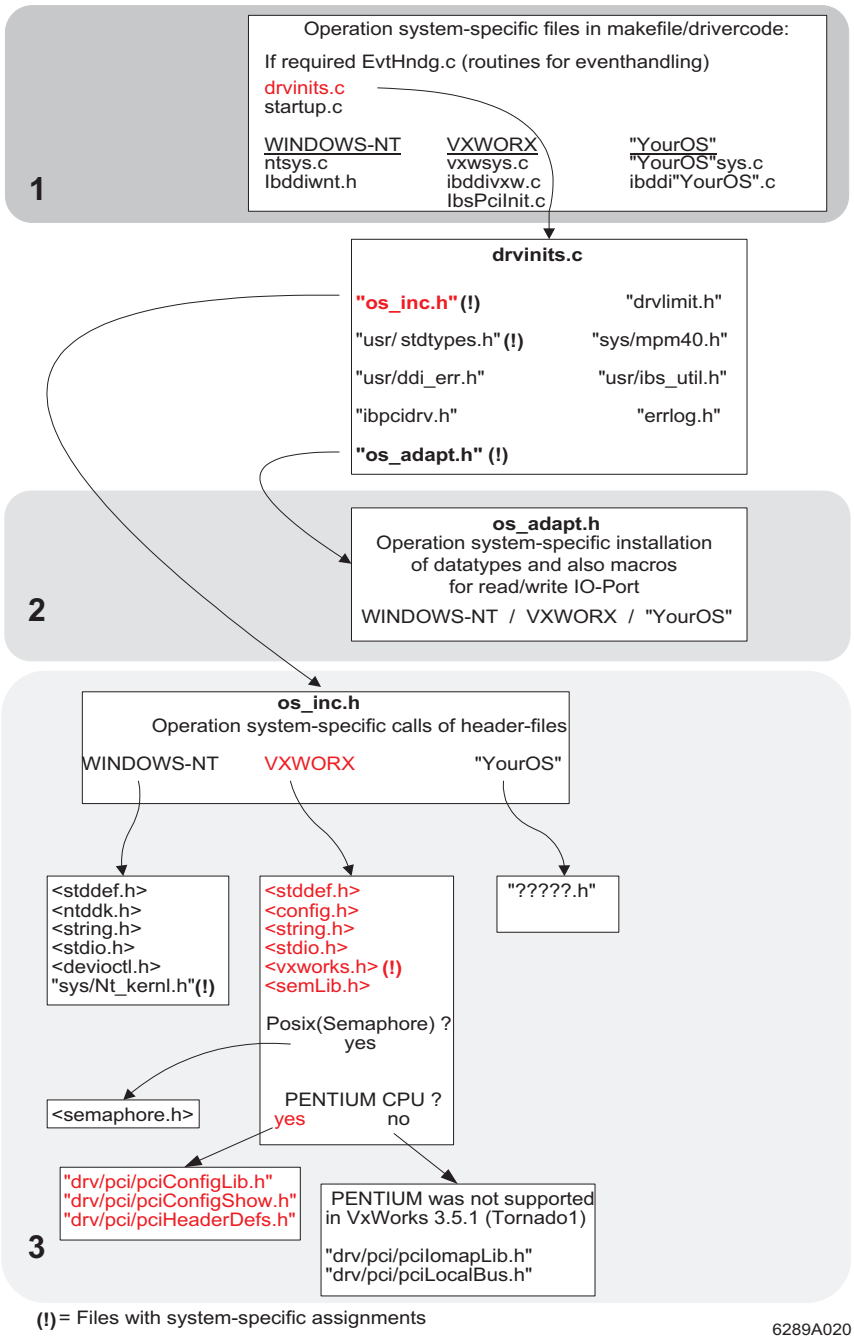


Figure 6-1 File structures in the driver

"OS?"sys.c

The "OS?"sys.c files (ntsys.c, vxwsys.c, and soon also "YourOS"sys.c) represent the interface to the relevant operating system. The system function calls are made here. The data structures are also adapted to the operating system.

The following functions are executed in "OS?"sys.c:
IBD_IoConnectInterrupt(PT_BOARD_INFO pBoardInfo);

– Connects the interrupt with the operating system

VxWorks function:

```
if ((pciIntConnect(INUM_TO_IVEC (((int) pBoardInfo->queryInfo.uInterrupt) +
    INT_NUM_IRQ0), IbsInterruptServiceRoutine, (int)pBoardInfo)) == ERROR)
    {
    printf("Connect interrupt failed\n");

    return(RETURN_ERR);
    }
sysIntEnablePIC((int) pBoardInfo->queryInfo.uInterrupt);
```

Windows NT function:

```
status = IoConnectInterrupt(&pBoardInfo->pInterruptObject,

    IbsInterruptServiceRoutine,

    pBoardInfo, NULL, pBoardInfo->uMappedIrqVector,

    pBoardInfo->Irql, pBoardInfo->Irql, LevelSensitive,

    TRUE,  /* share interrupt */ pBoardInfo->Affinity, FALSE);
```

ReadIOAddressBoardNumber(PT_PCI_INFO PciInfo);

– Reads the first I/O register of the PCI controller board. This contains the board number that is set using the DIP switches. All 8 possible board numbers are compared, and if two boards have the same address, these are indicated as "invalid".

IbsInterruptServiceRoutine(PT_BOARD_INFO pBoardInfo);

– Link to the InterruptServiceRoutine

ScanPciDevices(PT_PCI_INFO PciInfo);

– Searches in the PCI address area for the IBS PCI controller board and enters the relevant data structure.

The "ScanPciDevices" function is used to clarify the different readings of the required parameters. The sections of text marked in ***bold/italics*** are PCI system function calls. The "pPciInfo" structure, with the relevant data entered, is then available for other functions.

Windows NT: (ntsys.c)

```
void ScanPciDevices(PT_PCI_INFO PciInfo)
{
    …
    …
    …
    for (bus = 0; flag; bus++)
    {
      for (i = 0; i < PCI_MAX_DEVICES  &&  flag; i++)
      {
      SlotNumber.u.bits.DeviceNumber = i;
      for (f = 0; f < PCI_MAX_FUNCTION; f++)
      {
        SlotNumber.u.bits.FunctionNumber = f;
        j = HalGetBusData (PCIConfiguration, bus, SlotNumber.u.AsULONG,
              PciData, PCI_COMMON_HDR_LENGTH);
        if (j == 0)
        { // out of buses
          flag = FALSE;
          break;
        }
        if (PciData->VendorID == PCI_INVALID_VENDORID)
        { // skip to next slot
          break;
        }
        if (PciData->VendorID == PCI_PHOENIX_VENDOR_ID)
        {
          switch (PciData->DeviceID)
          {
          case PCI_PHOENIX_DEVICE_ID_SC:
            pciInfo->boardNumber = boardNumber;
            ++boardNumber;
            pciInfo->bus = (USIGN16)bus;
            pciInfo->pciDevice = (USIGN16)i;
            pciInfo->pciFunction = (USIGN16)f;
            pciInfo->vendorId = PciData->VendorID;
            pciInfo->deviceId = PciData->DeviceID;
            …
            …
            …
```

VxWorks: (vxwsys.c)

```
void ScanPciDevices(PT_PCI_INFO PciInfo)
{
    …
    …
    for (bus = 0; bus < 3; bus++)
    {
      for (sDeviceNo = 0; sDeviceNo < sDevices; sDeviceNo++)
      {
        pciConfigInWord (bus, sDeviceNo, 0, PCI_CFG_VENDOR_ID,
            &vendorId);
        pciConfigInWord (bus, sDeviceNo, 0, PCI_CFG_DEVICE_ID, &deviceId);
        if (vendorId != 0xffff)
        {
          if (vendorId == PCI_PHOENIX_VENDOR_ID)
          {
            switch (deviceId)
            {
              case PCI_PHOENIX_DEVICE_ID_SC:
              pPciInfo->boardNumber = uBoardNumber;
              ++uBoardNumber;
              pPciInfo->bus = (USIGN16)bus;
              pPciInfo->pciDevice = (USIGN16)i;
              pPciInfo->pciFunction = PCI_PHOENIX_FUNCTION_NUM;
              pPciInfo->vendorId = vendorId;
              pPciInfo->deviceId = deviceId;
              pciConfigInLong (bus, sDeviceNo,
                PCI_PHOENIX_FUNCTION_NUM,
                PCI_CFG_BASE_ADDRESS_1,
                &pPciInfo->memBaseAddress);
              pciConfigInLong (bus, sDeviceNo,
                PCI_PHOENIX_FUNCTION_NUM,
                PCI_CFG_BASE_ADDRESS_0,
              &pPciInfo->ioBaseAddress);
              pPciInfo->ioBaseAddress = (pPciInfo->ioBaseAddress & (~1));
              pciConfigInByte (bus, sDeviceNo,
                PCI_PHOENIX_FUNCTION_NUM, PCI_CFG_BRG_INT_LINE,
              &pPciInfo->interruptLine);
              …
              …
            }
          }
        }
      } /* for device */
    } /* for bus */
}
```

Example YourOS: ("YourOS"sys.c)

```
void ScanPciDevices(PT_PCI_INFO PciInfo)
{
    …
    for (bus = 0; bus < 3; bus++)
    {
       for (sDeviceNo = 0; sDeviceNo < sDevices; sDeviceNo++)
       {
          vendorId = YourFunction(get PCI_Bus_Data.Vendor);
                   /* Here for example,
                   all data can be read at the same time */
          deviceId = YourFunction(get PCI_Bus_Data.Device);
          if (vendorId != 0xffff)
          {
             if (vendorId == PCI_PHOENIX_VENDOR_ID)
             {
                switch (deviceId)
                {
                case PCI_PHOENIX_DEVICE_ID_SC:
                   pPciInfo->boardNumber = uBoardNumber;
                   ++uBoardNumber;
                   pPciInfo = YourFunction(get PCI_Bus_Data);   /* Here for
                   example, all data can be read at the same time */
                }
             }
          }
       } /* for device */
    } /* for bus */
}
```

os_adapt.h

The pointers, structures, and macros defined here are operating system-dependent. Thus, for example, the byte read access to the I/O ports in Windows NT is gained using the "***READ_PORT_UCHAR***(addr)" function and in VxWorks using "***sysInByte***(addr)". The function that makes *"YourOS"* available should be used here.

Windows NT:

```
#ifdef IBS_WIN_NT_VERSION          /*selects the WindowsNT-funtionality */
typedef int T_BIN_SEMAPHORE;
typedef int T_MUTEX_SEMAPHORE;
typedef USIGN8 *IOPORT8;                                  /* io-port access */
typedef DEVICE_OBJECT *P_DEVICE_OBJECT;          /* device object pointer */
typedef DRIVER_OBJECT *P_DRIVER_OBJECT;          /* driver object pointer */
typedef UNICODE_STRING P_DEVICE_NAME;                  /* device name */
…
…
/*
// General macros for reading and writing the I/O ports of the Interbus board.
*/
#define IbsReadIOPort(addr) (READ_PORT_UCHAR((addr)))
#define IbsReadIOPort16(addr) (READ_PORT_USHORT((addr)))
#define IbsWriteIOPort(addr, dataByte) (WRITE_PORT_UCHAR((addr),
(UCHAR)dataByte))
#define IbsWriteIOPort16(addr, dataByte) (WRITE_PORT_USHORT((addr),
(USHORT)dataByte))
…
…
```

VxWorks:

```
#ifdef IBS_VXWORKS_VERSION          /* selects the VxWorks-funtionality */
typedef SEM_ID T_BIN_SEMAPHORE;
typedef SEM_ID T_MUTEX_SEMAPHORE;
typedef short int IOPORT8;                                /* io-port access */
/* device object pointer */
typedef struct {void *DeviceExtension; } BUF_P_DEVICE_OBJECT,
*P_DEVICE_OBJECT;
typedef int *P_DRIVER_OBJECT;                    /* driver object pointer */
typedef char *P_DEVICE_NAME;                           /* device name */
…
…
/*
// General macros for reading and writing the I/O ports of the Interbus board.
*/
#define IbsReadIOPort(addr) (sysInByte((addr)))
#define IbsReadIOPort16(addr) (sysInWord((addr)))
```

```
#define IbsWriteIOPort(addr, dataByte) (sysOutByte((addr), (USIGN8)dataByte))
#define IbsWriteIOPort16(addr, dataByte) (sysOutWord((addr), (USIGN16)
dataByte))

typedef USIGN32 NTSTATUS;
...
...
```

Your OS:

```
#ifdef IBS_ YourOS_VERSION              /* selects the "YourOS"-funtionality */
typedef "YourDatatyp" T_BIN_SEMAPHORE;
...
...
#define IbsReadIOPort(addr) ("YourReadIOPortFunktion"(addr)))
...
...
```

your os_inc.h

The "distribution" to the various header files, depending on the operating system, takes place here.

Windows NT:

```
#ifdef IBS_WIN_NT_VERSION
    /**/
    #include <stddef.h>
    #include <ntddk.h>
    #include <string.h>
    #include <stdio.h>
    #include <devioctl.h>
    #include "sys/nt_kernl.h"
    /**/
#endif
```

VxWorks:

```
#ifdef IBS_VXWORKS_VERSION
    /**/
    #include <vxworks.h>
    #include <stddef.h>
    #include <config.h>
    #include <string.h>
    #include <stdio.h>
    #include <semLib.h>
    /*#include <vxcpu.h>*/
    #ifdef PENTIUM
        #include "drv/pci/pciConfigLib.h"
        #include "drv/pci/pciConfigShow.h"
        #include "drv/pci/pciHeaderDefs.h"
    #else
        /* PENTIUM was not supported in VxWorks 3.5.1 (Tornado1) */
        #include "drv/pci/pciIomapLib.h"
        #include "drv/pci/pciLocalBus.h"
    #endif
    /*
    set INCLUDE_POSIX_SEM to include Posix named semaphore handling
    */
    #ifdef INCLUDE_POSIX_SEM
        #include <semaphore.h>
    #endif
#endif
```

Your OS:

```
#ifdef IBS_YourOS_VERSION
    /**/
    #include <stddef.h>
    #include <string.h>
    #include <stdio.h>
    #include <?????>
    ...
    ...
    /**/
#endif
```

A 1 List of Figures

Section 1

Section 2

List of Figures

Section 3

Section 4

Section 5

Section 6

List of Figures

A 2 List of Tables

Section 1

Section 2

Section 5

A 3 Index

T

Index

We Are Interested in Your Opinion!

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

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

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

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

Phone +49 - (0) 52 35 - 3-00
Telefax +49 - (0) 52 35 - 3-4 20 66
E-Mail tecdoc@phoenixcontact.com

FAX Reply

Phoenix Contact GmbH & Co. KG
Marketing Services
Dokumentation INTERBUS

Date: _____

Fax No: +49 - (0) 52 35 - 3-4 20 66

From:

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

Document:

Designation: IBS PCI DDK UM E Revision: A Order No.: 26 98 16 4

My Opinion on the Document

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

Other Comments: