The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC821-SW-82

Linux Device Driver

INTERBUS Master G4 PMC

Version 1.0.x

User Manual

Issue 1.0.0

July 2007

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7
25469 Halstenbek, Germany
www.tews.com
Phone: +49 (0) 4101 4058 0
Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com

TEWS TECHNOLOGIES LLC
9190 Double Diamond Parkway,
Suite 127, Reno, NV 89521, USA
www.tews.com
Phone: +1 (775) 850 5830
Fax: +1 (775) 201 0347
e-mail: usasales@tews.com

TPMC821-SW-82

Linux Device Driver

INTERBUS Master G4 PMC

Supported Modules:
TPMC821

Issue	Description	Date
1.0.0	First Issue	July 24, 2007

Table of Contents

1 Introduction

The TPMC821-SW-82 Linux device driver allows the operation of TPMC821 devices conforming to the Linux I/O system specification. This includes a device-independent basic I/O interface with open(), close() and ioctl() functions.

Special I/O operation that do not fit to the standard I/O calls will be performed by calling the ioctl() function with a specific function code and an optional function dependent argument.

The TPMC821-SW-82 device driver supports the following features:

- Asynchronous mode with consistency locking
- Asynchronous mode without consistency locking
- Bus synchronous mode
- Program synchronous mode
- Standard function bit commands
- Mailbox commands
- Reading and writing process data
- Reading diagnostic information
- Creates devices with dynamically allocated or fixed major device numbers
- DEVFS and SYSFS (UDEV) support for automatic device node creation

The TPMC821-SW-82 device driver supports the modules listed below:

TPMC821	INTERBUS Master G4 PMC	(PMC)

To get more information about the features and use of TPMC821 devices it is recommended to read the manuals listed below.

TPMC821 User manual

TPMC821 Engineering Manual

Interbus specification

2 Installation

Following files are located on the distribution media:

Directory path 'TPMC821-SW-82':

TPMC821-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
TPMC821-SW-82-1.0.0.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

For installation the files have to be copied to the desired target directory.

The GZIP compressed archive TPMC821-SW-82-SRC.tar.gz contains the following files and directories:

Directory path './tpmc821/':

tpmc821.c	Driver source code
tpmc821def.h	Driver include file
tpmc821.h	Driver include file for application program
include/tpxxxhwdep.c	Hardware dependent library
include/tpxxxhwdep.h	Hardware dependent library header file
include/tpmodule.c	Driver independent library
include/tpmodule.h	Driver independent library header file
include/config.h	Driver independent library header file
Makefile	Device driver make file
makenode	Script to create device nodes in the file system
example/tpmc821exa.c	Example application
example/Makefile	Example application make file

In order to perform an installation, extract all files of the archive TPMC821-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TPMC821-SW-82-SRC.tar.gz' will extract the files into the local directory.

2.1 Build and install the device driver

- Login as *root*
- Change to the target directory
- To create and install the driver in the module directory */lib/modules/<version>* enter:

  # make install

- Only after the first build we have to execute *depmod* to create a new dependency description for loadable kernel modules. This dependency file is later used by *modprobe* to automatically load dependent kernel modules.

  # depmod –aq

2.2 Uninstall the device driver

- Login as *root*
- Change to the target directory
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

  # make uninstall

- Update kernel module dependency description file

  # depmod –aq

2.3 Install the device driver in the running kernel

- To load the device driver into the running kernel, login as root and execute the following commands:

  # **modprobe tpmc821drv**

- After the first build or if you are using dynamic major device allocation it is necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a device file system (devfs or sysfs with udev) then you have to skip running the *makenode* script. Instead of creating device nodes from the script the driver itself takes creating and destroying of device nodes in its responsibility.

  # sh makenode

On success the device driver will create a minor device for each TPMC821 device found. The first TPMC821 device can be accessed with device node /dev/tpmc821_0, the second with /dev/tpmc821_1, the third with /dev/tpmc821_2 and so on.

The assignment of device nodes to physical TPMC821 modules depends on the search order of the PCI bus driver.

2.4 Remove device driver from the running kernel

- To remove the device driver from the running kernel login as root and execute the following command:

  # modprobe tpmc821drv –r

If your kernel has enabled devfs or sysfs (udev), all /dev/tpmc821_x nodes will be automatically removed from your file system after this.

Be sure that the driver isn't opened by any application program. If opened you will get the response "*tpmc821drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe –r* again.

2.5 Change Major Device Number

The TPMC821 device driver uses dynamic allocation of major device numbers by default. If this isn't suitable for the application it is possible to define a major number for the driver. If the kernel has enabled devfs the driver will not use the symbol TPMC821_MAJOR.

To change the major number edit the file *tpmc821def.h*, change the following symbol to an appropriate value and enter **make install** to create a new driver.

TPMC821_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TPMC821_MAJOR       122
```

Be sure that the desired major number isn't used by other drivers. Please check *proc/devices* to see which numbers are free.

3 Device Input/Output functions

This chapter describes the interface to the device driver I/O system.

3.1 open()

NAME

open() - open a file descriptor

SYNOPSIS

#include <fcntl.h>

int **open** (const char *filename, int flags)

DESCRIPTION

The open function creates and returns a new file descriptor for the file named by *filename*. The *flags* argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the | operator in C).

See also the GNU C Library documentation for more information about the open function and open flags.

EXAMPLE

```
int fd;

...

fd = open(“/dev/tpmc821_0”, O_RDWR);
if (fd < 0)
{
     /* handle error condition */
}
```

RETURNS

The normal return value from open is a non-negative integer file descriptor. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

E_NODEV	The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during open. For more information about open error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.2 close()

NAME

close() – close a file descriptor

SYNOPSIS

#include <unistd.h>

int **close** (int *filedes*)

DESCRIPTION

The close function closes the file descriptor *filedes*.

EXAMPLE

```
int fd;
...
if (close(fd) != 0)
{
     /* handle close error conditions */
}
```

RETURNS

The normal return value from close is 0. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

E_NODEV	The requested minor device does not exist.

This is the only error code returned by the driver, other codes may be returned by the I/O system during close. For more information about close error codes, see the *GNU C Library description – Low-Level Input/Output*.

SEE ALSO

GNU C Library description – Low-Level Input/Output

3.3 ioctl()

NAME

ioctl() – device control functions

SYNOPSIS

#include <sys/ioctl.h>

int **ioctl**(int *filedes*, int *request* [, void *argp*])

DESCRIPTION

The ioctl function sends a control code directly to a device, specified by *filedes*, causing the corresponding device to perform the requested operation.

The argument *request* specifies the control code for the operation. The optional argument *argp* depends on the selected request and is described for each request in detail later in this chapter.

The following ioctl codes are defined in tpmc821.h:

Function	Description
TPMC821_IOCX_READ	Read process data out of the “DTA IN” area
TPMC821_IOCX_WRITE	Write new process data to the “DTA OUT” area
TPMC821_IOCS_BITCMD	Execute a standard function bit command
TPMC821_IOCX_MBOXCMD	Execute a mailbox command and wait for confirmation
TPMC821_IOCX_MBOXCMD_NOWAIT	Execute a mailbox command without waiting for confirmation
TPMC821_IOCG_GETDIAG	Get diagnostic information from the device
TPMC821_IOCS_CONFIG	Configure the device driver
TPMC821_IOC_HOSTFAIL_SET	Set a serious host system failure interrupt
TPMC821_IOC_HOSTFAIL_RESET	Reset the host system failure interrupt
TPMC821_IOC_HWFAIL_RESET	Reset the device hardware failure flag

See behind for more detailed information on each control code.

To use these TPMC821 specific control codes the header file tpmc821.h must be included in the application

RETURNS

On success, zero is returned. In the case of an error, a value of –1 is returned. The global variable *errno* contains the detailed error code.

ERRORS

EINVAL	Invalid argument. This error code is returned if the requested ioctl function is unknown. Please check the argument *request*.

Other function dependant error codes will be described for each ioctl code separately. Note, the TPMC821 driver always returns standard Linux error codes.

SEE ALSO

ioctl man pages

3.3.1 TPMC821_IOCX_READ

NAME

TPMC821_IOCX_READ – Read process data out of the “DTA IN” area

DESCRIPTION

This ioctl function reads process data out of the *DTA IN* area. A pointer to the caller’s message buffer (*TPMC821_RWBUF*) is passed by the parameter *argp* to the driver. This buffer contains variable length segments of data type *TPMC821_SEGMENT*. Each segment holds an exact description of the embedded data like data type, number of data items and offset in the *DTA IN* area. On entrance of this control function, every segment contains a description of the data items to read; on exit the driver fills the data union with the desired process data.

This relative complex mechanism has two advantages. First you can pick up occasional placed data items without copying the whole *DAT IN* buffer and second, word and long word organized data items can be automatically byte swapped by the driver. Remember Intel x86 CPU’s use little-endian and Motorola respective the INTERBUS use big-endian alignment of data words

```
typedef struct
{
        unsigned long           Size;
        TPMC821_SEGMENT         SegmentBuffer;
} TPMC821_RWBUF, *PTPMC821_RWBUF;
```

Size

Receives the total size of the variable structure buffer.

SegmentBuffer

This is the variable sized data buffer, specified as follows:

```
typedef struct {
        unsigned short      ItemNumber;
        unsigned short      ItemType;
        unsigned short      DataOffset;
        union {
                unsigned char       byte[1];
                unsigned short      word[1];
                unsigned long       lword[1];
        } u;
} TPMC821_SEGMENT, *PTPMC821_SEGMENT;
```

ItemNumber

Specifies the number of items of the specified type in the data array. In other words it specifies the size of the array u.byte[], u.word[] or u.lword[].

ItemType

Specifies the data type of the embedded process data. Note, every data item in the segment must have the same type. The following values are possible:

Value	Description
TPMC821_END	Specifies the last segment in the list. No data follows.
TPMC821_BYTE	Specifies a segment with byte data. The union part byte[] will be used.
TPMC821_WORD	Specifies a segment with word data. The union part word[] will be used and all words of the array will be byte swapped.
TPMC821_LWORD	Specifies a segment with long word data. The union part lword[] will be used and all long words will be byte swapped.

DataOffset

Specifies a byte offset from the beginning of the *DTA IN* area. The driver start reading data items from this offset and stores the requested number of items in the data union of the segment structure.

u

The union *u* contains three arrays. The size of these dynamic expandable arrays depends on the number of data items to read. Because the size of this arrays is only well-known at run-time you should never use the *sizeof()* function to determine the size of the segment structure.
The macro TPMC821_*SEGMENT_SIZE(pSeg)* (defined in tpmc821.h) delivers the correct structure size. The macro TPMC821_*NEXT_SEGMENT(pSeg)* (also defined in tpmc821.h) calculates a pointer to the beginning of the following segment in the buffer. Both macros in combination should be used to assemble a read data buffer for the desired read request. The end of the read buffer is specified by a segment with type of *TPMC821_END*.
Please refer to the following example to see how to assemble a correct read buffer.

EXAMPLE

```
#include “tpmc821.h

int                fd;
int                result;
unsigned long      size;
TPMC821_SEGMENT*   pSeg;
TPMC821_RWBUF*     pReadBuf;

pReadBuf = (TPMC821_RWBUF*)malloc( 100 );

// [1]
size.= 0;
pSeg = (TPMC821_SEGMENT*)&pReadBuf->SegmentBuffer;

// [2]
pSeg->ItemType     = TPMC821_BYTE;
pSeg->ItemNumber   = 4;
pSeg->DataOffset   = 0;
size += TPMC821_SEGMENT_SIZE(pSeg);        //  add size of this segment

// [3]
pSeg = TPMC821_NEXT_SEGMENT(pSeg);
pSeg->ItemType     = TPMC821_WORD;         //  same data read as word
pSeg->ItemNumber   = 2;
pSeg->DataOffset   = 0;
size += TPMC821_SEGMENT_SIZE(pSeg);

pSeg = TPMC821_NEXT_SEGMENT(pSeg);         //  same data read as longword
pSeg->ItemType     = TPMC821_LWORD;
pSeg->ItemNumber   = 1;
pSeg->DataOffset   = 0;
size += TPMC821_SEGMENT_SIZE(pSeg);

// [4]
pSeg = TPMC821_NEXT_SEGMENT(pSeg);         //  End segment
pSeg->ItemType     = TPMC821_END;
pSeg->ItemNumber   = 0;
pSeg->DataOffset   = 0;
size += TPMC821_SEGMENT_SIZE(pSeg);
```

```
// [5]
pReadBuf->Size = size;
result = ioctl(fd, TPMC821_IOCX_READ, (char*)pReadBuf);

if (result >= 0) {
     /* read operation successful. */
} else {
     /* read operation failed. */
}
```

This example reads the first four bytes of the *DTA IN* area within three segments with different types (byte, word and longword). If the first 4 bytes of the *DTA IN* area contain significant values you can realize the effect of byte swapping words and longwords (see also tpmc821exa.c).

[1] After opening the device, the variable ***size*** is initialized with 0 and the segment pointer is set to begin of the segment buffer. Be sure that the size of the buffer is large enough to hold all segments.

[2] The first segment contains 4 bytes read from offset 0 of the *DTA IN* area. After initializing of the segment we update the buffer size using the *TPMC821_SEGMENT_SIZE* macro. Do not use *sizeof()* instead.

[3] Before initializing the next segment we calculate a new segment pointer using the *TPMC821_NEXT_SEGMENT* macro. This macro simply adds the size of the previous segment to the previous segment pointer and returns the new segment pointer. The new segment starts without a gap direct after the previous segment.

[4] The end of the segment list is specified by a segment with item type *TPMC821_END*. If this segment is missing the read request fails. Be sure that the size of the end segment is included in the total size of the segment list.

[5] The ioctl() call transfers the request to the driver. The driver interprets the segment list and fills the corresponding data arrays with new process data.

ERRORS

EBUSY	Device is busy with a pending job. Try again later.
EIO	Hardware error (Interbus not running), or the driver initialization was not completed.
EFAULT	Error copying data to or from user area.
ENOMEM	Error allocating internal memory to hold user data.
EINVAL	Supplied data buffer contains invalid segment data. The total size may be exceeded, or an unknown item type was specified.
ETIME	The allowed time to finish the request has elapsed.

3.3.2 TPMC821_IOCX_WRITE

NAME

TPMC821_IOCX_WRITE – Write new process data to the “DTA OUT” area

DESCRIPTION

This ioctl function writes new data to the *DTA OUT* area. A pointer to the caller’s message buffer (*TPMC821_RWBUF*) is passed by the parameter *argp* to the driver. This buffer contains variable length segments of data type *TPMC821_SEGMENT*. Each segment holds an exact description of the embedded data like data type, number of data items, offset in the *DTA OUT* area and the data items to write.

This relative complex mechanism has two advantages. First you can write occasional placed data items without writing data in the whole *DAT OUT* buffer and second, word and long word organized data items can be automatically byte swapped by the driver. Remember Intel x86 CPU’s use little-endian and Motorola respective the INTERBUS use big-endian alignment of data words.

```
typedef struct
{
        unsigned long           Size;
        TPMC821_SEGMENT         SegmentBuffer;
} TPMC821_RWBUF, *PTPMC821_RWBUF;
```

Size

Receives the total size of the variable structure buffer.

SegmentBuffer

This is the variable sized data buffer, specified as follows:

```
typedef struct {
        unsigned short      ItemNumber;
        unsigned short      ItemType;
        unsigned short      DataOffset;
        union {
                unsigned char       byte[1];
                unsigned short      word[1];
                unsigned long       lword[1];
        } u;
} TPMC821_SEGMENT, *PTPMC821_SEGMENT;
```

ItemNumber

Specifies the number of items of the specified type in the data array. In other words it specifies the size of the array u.byte[], u.word[] or u.lword[].

ItemType

Specifies the data type of the embedded process data. Note, every data item in the segment must have the same type. The following values are possible:

Value	Description
TPMC821_END	Specifies the last segment in the list. No data follows.
TPMC821_BYTE	Specifies a segment with byte data. The union part byte[] will be used.
TPMC821_WORD	Specifies a segment with word data. The union part word[] will be used and all words of the array will be byte swapped.
TPMC821_LWORD	Specifies a segment with long word data. The union part lword[] will be used and all long words will be byte swapped.

DataOffset

Specifies a byte offset from the beginning of the DTA OUT area where the new data should be written.

u

The union u contains three arrays. The size of these dynamic expandable arrays depends on the number of data items to write. Because the size of this arrays is only well-known at run-time you should never use the sizeof() function to determine the size of the segment structure.
The macro TPMC821_SEGMENT_SIZE(pSeg) (defined in tpmc821.h) delivers the correct structure size. The macro TPMC821_NEXT_SEGMENT(pSeg) (also defined in tpmc821.h) calculates a pointer to the begin of the following segment in the buffer. Both macros in combination should be used to assemble a write data buffer for the desired write request. The end of the write buffer is specified by a segment with type of TPMC821_END.
Please refer to the next example to see how to assemble a correct write buffer.

EXAMPLE

```
#include “tpmc821.h”

int                fd;
int                result;
unsigned long      size;
TPMC821_SEGMENT*   pSeg;
TPMC821_RWBUF*     pWriteBuf;

pWriteBuf = (TPMC821_RWBUF*)malloc( 100 );

// [1]
size = 0;
pSeg = (TPMC821_SEGMENT*)&pWriteBuf->SegmentBuffer;

// [2]
pSeg->ItemType     = TPMC821_BYTE;
pSeg->ItemNumber   = 4;
pSeg->DataOffset   = 0;
pSeg->u.byte[0]    = 1;
pSeg->u.byte[1]    = 2;
pSeg->u.byte[2]    = 3;
pSeg->u.byte[3]    = 4;
size += TPMC821_SEGMENT_SIZE(pSeg);        //  add size of this segment

// [3]
pSeg = TPMC821_NEXT_SEGMENT(pSeg);         //  calculate next pointer
pSeg->ItemType     = TPMC821_LWORD;
pSeg->ItemNumber   = 1;
pSeg->DataOffset   = 4;
pSeg->u.lword[0]   = 0xAA55BB66;
size += TPMC821_SEGMENT_SIZE(pSeg);

// [4]
pSeg = TPMC821_NEXT_SEGMENT(pSeg);         //  End segment
pSeg->ItemType     = TPMC821_END;
pSeg->ItemNumber   = 0;
pSeg->DataOffset   = 0;
size += TPMC821_SEGMENT_SIZE(pSeg);
```

```
// [5]
pWriteBuf->Size = size;
result = ioctl(fd, TPMC821_IOCX_WRITE, (char*)pWriteBuf);
if (result >= 0) {
     /* write operation successful. */
} else {
     /* write operation failed. */
}
```

This example does the following (see also tpmc821exa.c).

[1] The variable *size* is initialized with 0 and the segment pointer is set to the beginning of the segment buffer. Be sure that the size of the buffer is large enough to hold all segments.

[2] The first segment contains 4 bytes to write from offset 0 of the *DTA OUT* area. After initializing of the segment we update the buffer size using the TPMC821_*SEGMENT_SIZE* macro. Do not use *sizeof()* instead.

[3] Before initializing the next segment we calculate a new segment pointer using the *TPMC821_NEXT_SEGMENT* macro. This macro simply adds the size of the previous segment to the previous segment pointer and returns the new segment pointer. The new segment starts without a gap direct after the previous segment. The long word data item will be byte-swapped before writing to the *DTA OUT* area.

[4] The end of the segment list is specified by a segment with item type *TPMC821_END*. If this segment is missing the read request fails. Be sure that the size of the end segment is included in the total size of the segment list.

[5] The ioctl() call transfers the request to the driver. The driver interprets the segment list and writes the contents of the data array to the specified locations in the *DTA OUT* area.

The following example displays the memory layout of the segment buffer and the *DTA OUT* area after a successful write operation.

Segment values:

1st segment:

ItemNumber:	4
ItemType:	TPMC821_BYTE
ItemOffset:	0x0
data:	0x01, 0x02, 0x03, 0x04

2nd segment:

ItemNumber:	1
ItemType:	TPMC821_LWORD
ItemOffset:	0x0
data:	0xAA55BB66

End segment:

ItemNumber:	0
ItemType:	TPMC821_END
ItemOffset:	0x0
data:	(none)

The segment buffer has the following layout:

Offset	+0	+1	+2	+3	+4	+5	+6	+7
+0x00	0x04	0x00	0x01	0x00	0x00	0x00	0x01	0x02
+0x08	0x03	0x04	0x01	0x00	0x04	0x00	0x04	0x00
+0x10	0x66	0xBB	0x55	0xAA	0x00	0x00	0x00	0x00
+0x18	0x00	0x00	x	x	x	x	x	x

The *DTA OUT* area of the TPMC821 (after writing):

Offset	+0	+1	+2	+3	+4	+5	+6	+7
+0x00	0x01	0x02	0x03	0x04	0xAA	0x55	0xBB	0x66
+0x08	x	x	x	x	x	x	x	x

ERRORS

EBUSY	Device is busy with a pending job. Try again later.
EIO	Hardware error (Interbus not running), or the driver initialization was not completed.
EFAULT	Error copying data to or from user area.
ENOMEM	Error allocating internal memory to hold user data.
EINVAL	Supplied data buffer contains invalid segment data. The total size may be exceeded, or an unknown item type was specified.
ETIME	The allowed time to finish the request has elapsed.

3.3.3 TPMC821_IOCS_BITCMD

NAME

TPMC821_IOCS_BITCMD - Execute a standard function bit command

DESCRIPTION

This ioctl function allows the execution of various frequently used commands and command sequences without using mailbox commands. A pointer to the caller's parameter buffer (*TPMC821_BITCMDBUF*) is passed by the argument pointer *argp* to the driver.

typedef struct {
    unsigned short FunctionBit;
    unsigned short FunctionParam;
} TPMC821_BITCMDBUF, *PTPMC821_BITCMDBUF;

FunctionBit

Specifies the bit number [0...6] of the standard function to execute.

FunctionParam

Specifies an optional parameter for the standard function.

Additional information about standard function bits and parameter values can be found in the User Manual – *INTERBUS Generation 4 Master Board*, which is part of the TPMC821 Engineering Manual.

EXAMPLE

```
#include “tpmc821.h”

int                fd;
int                result;
TPMC821_BITCMDBUF  BitCmdBuf;

BitCmdBuf.FunctionBit       = 0;      // Start_Data_Transfer_Req (1<<0)
BitCmdBuf.FunctionParam     = 0;      // none

result = ioctl(fd, TPMC821_IOCX_BITCMD, (char*)&BitCmdBuf);

if (result >= 0) {
     /* operation successful. */
} else {
     /* operation failed. */
}
```

ERRORS

EBUSY	Device is busy with a pending job. Try again later.
EIO	Hardware error (Interbus not running), or the driver initialization was not completed.
EFAULT	Error copying data to or from user area.
EINVAL	Invalid Bit number specified.
ETIME	The allowed time to finish the request has elapsed.

3.3.4 TPMC821_IOCX_MBOXCMD, _NOWAIT

NAME

TPMC821_IOCX_MBOXCMD - Execute a mailbox command with or without waiting for confirmation

DESCRIPTION

This ioctl function is used to transmit messages from the host system to the IBS master (SSGI box 0). If an answer message is expected the received message (SSGI box 1) is copied direct to the user receive buffer.

A pointer to the caller's parameter buffer (*unsigned short array*) is passed by the parameter pointer *argp* to the driver.

Transmit and receive buffers are organized as follows (valid for all services):

Word 1	Service_Code
Word 2	Parameter_Count
Word 3	Parameter
Word 4	Parameter
...	...
	Parameter

The control function *TPMC821_IOCX_MBOXCMD_NOWAIT* returns immediately to the caller without waiting for an answer. This control function is used only for reset and unconfirmed PCP services.

Additional information about supported services can be found in the IBS User Manuals which are part of the TPMC821 Engineering Manual.

EXAMPLE

```
#include “tpmc821.h”

int               fd;
int               result;
unsigned short    RequestPar[100];

...

RequestPar[0] = 0x0710;          // Create Configuration Service
RequestPar[1] = 1;               // one parameter follow
RequestPar[2] = 1;               // number of frames to generate

result = ioctl(fd, TPMC821_IOCX_MBOXCMD, (char*)&RequestPar);

if (result >= 0) {
     /* operation successful. */
} else {
     /* operation failed. */
}
```

ERRORS

EBUSY	Device is busy with a pending job. Try again later.
EIO	Hardware error (Interbus not running), or the driver initialization was not completed.
EFAULT	Error copying data to or from user area.
ENOMEM	Error allocating internal memory to hold user data.
EINVAL	Invalid parameter specified, max. size exceeded.
ETIME	The allowed time to finish the request has elapsed.

3.3.5 TPMC821_IOCG_GETDIAG

NAME

TPMC821_IOCG_GETDIAG - Get diagnostic information from the device

DESCRIPTION

This ioctl function returns a structure with various diagnostic information to the caller.

A pointer to the caller's parameter buffer (*TPMC821_DIAGBUF*) is passed by the parameter pointer *argp* to the driver.

```
typedef struct {
        unsigned short      SysfailReg;
        unsigned short      ConfigReg;
        unsigned short      DiagReg;
        unsigned short      HardwareFailure;
        unsigned short      InitComplete;
} TPMC821_DIAGBUF, *PTPMC821_DIAGBUF;
```

SysfailReg, ConfigReg, DiagReg

Returns the actual values of the corresponding hardware register in the coupling memory: *Status Sysfail Register, Configuration Register and Master Diagnostic Status Register.* The meaning of every bit in these registers is described in the User Manual – *INTERBUS Generation 4 Master Board.*

HardwareFailure

If the content is TRUE the IBS master has detected a hardware error. In this case the driver will not accept data transfer or message box commands until this state is left by the *TPMC821_HWFAIL_RESET* command.
Note. A hardware failure could also occur after execution of the mailbox command *Reset_Controller_Board*.

InitComplete

This parameter is TRUE if the INTERBUS firmware has completed initialization.

EXAMPLE

```
#include “tpmc821.h”

int                fd;
int                result;
TPMC821_DIAGBUF    DiagBuf;


result = ioctl(fd, TPMC821_IOCG_GETDIAG, (char*)&DiagBuf);

if (result >= 0) {

     printf( "\nRead Diagnostic Information successful\n" );
     printf( "Status Sysfail Register    : %04Xhex\n",
          DiagBuf.SysfailReg );
     printf( "Configuration Register     : %04Xhex\n",
          DiagBuf.ConfigReg );
     printf( "Master Diagnostic Register : %04Xhex\n",
          DiagBuf.DiagReg );
     printf( "Hardware Failure           : %s\n",
          DiagBuf.HardwareFailure ? "TRUE" : "FALSE" );
     printf( "Initialization done        : %s\n",
          DiagBuf.InitComplete ? "TRUE" : "FALSE" );
} else {
     // process error
}
```

ERRORS

EFAULT	Error copying data to or from user area.

3.3.6 TPMC821_IOCS_CONFIG

NAME

TPMC821_IOCS_CONFIG - Configure the device driver

DESCRIPTION

This ioctl function announces a new operating mode to the driver and changes timeout values for mailbox and data transfer functions. Every time the host changes the operating mode (SetValue mailbox message) the driver must be introduced about that so it can handle following data transfer message in the right manner (see also *Automatic Configuration* in the example application)..

A pointer to the caller's parameter buffer (*TPMC821_CONFIGBUF*) is passed by the parameter pointer *argp* to the driver.

```
typedef struct {
        unsigned short      OperatingMode;
        unsigned long       DataTimeout;
        unsigned long       MailboxTimeout;
} TPMC821_CONFIG, *PTPMC821_CONFIG;
```

OperationMode

Specifies the new operating mode. Valid operating modes are:

Value	Description
TPMC821_ASYNC	In asynchronous operating mode, the process data is updated by the INTERBUS firmware synchronously with the INTERBUS data cycles, but asynchronously with hosts' access to the process image. This operating mode is default after RESET
TPMC821_ASYNC_LOCK	In this asynchronous operating mode the hosts' access to the process data is locked for reading and writing consistent data.
TPMC821_BUS_SYNC	Bus synchronous operating mode
TPMC821_PROG_SYNC	Program synchronous operating mode

Additional information about operating modes can be found in the User Manual – *INTERBUS Generation 4 Master Board*, which is part of the TPMC821 Engineering Manual.

DataTimeout

Specifies a new timeout value (in seconds) for all following read and write commands from and to the *DTA IN* and *DTA OUT* area. The default timeout value is 2 seconds.

MailboxTimeout

Specifies a new timeout value (in seconds) for all following mailbox and function bit commands. The default timeout value is 10 seconds.

EXAMPLE

```
#include “tpmc821.h”

int                fd;
int                result;
TPMC821_CONFIGBUF  ConfigBuf;

// Setup new operating mode in the IBS firmware ...

ConfigBuf.OperatingMode     = TPMC821_ASYNC_LOCK;
ConfigBuf.DataTimeout       = 1;
ConfigBuf.MailBoxTimeout    = 20;

result = ioctl(fd, TPMC821_IOCS_CONFIG, (char*)&ConfigBuf);

if (result < 0) {
  /* handle ioctl error */
}
```

ERRORS

EFAULT	Error copying data to or from user area.
EINVAL	Invalid parameter specified.

3.3.7 TPMC821_IOC_HOSTFAIL_SET

NAME

TPMC821_IOC_HOSTFAIL_SET - Set a serious host system failure interrupt

DESCRIPTION

This ioctl function signals a serious host system failure to the TPMC821. On assertion of this host fail interrupt the TPMC821 resets all INTERBUS outputs and switch on the HF LED on the TPMC821 control panel.

If the driver was terminated the host system failure is automatically set by the driver.

The optional argument pointer can be omitted for this ioctl function.

EXAMPLE

```
#include “tpmc821.h”

int  fd;
int  result;

...

result = ioctl(fd, TPMC821_IOC_HOSTFAIL_SET);

if (result < 0) {
  /* handle ioctl error */
}
```

3.3.8 TPMC821_IOC_HOSTFAIL_RESET

NAME

TPMC821_IOC_HOSTFAIL_RESET - Reset the host system failure interrupt

DESCRIPTION

This ioctl function resets the host system failure state in the TPMC821.

The optional argument pointer can be omitted for this ioctl function.

EXAMPLE

```
#include “tpmc821.h”

int  fd;
int  result;
...

result = ioctl(fd, TPMC821_IOC_HOSTFAIL_RESET);

if (result < 0) {
  /* handle ioctl error */
}
```

3.3.9 TPMC821_IOC_HWFAIL_RESET

NAME

TPMC821_IOC_HWFAIL_RESET - Reset the device hardware failure flag

DESCRIPTION

This ioctl function resets the hardware failure flag in the device driver. The hardware failure flag was set after reception of a service interrupt request from the TPMC821.

The optional argument pointer can be omitted for this ioctl function.

Additional information about the service interrupt request can be found in the TPCM821 User Manual and User Manuals for the INTERBUS Generation 4 Master Board which is part of the TPMC821 Engineering Manual.

EXAMPLE

```
#include “tpmc821.h”

int  fd;
int  result;

result = ioctl(fd, TPMC821_IOC_HWFAIL_RESET);

if (result < 0) {
  /* handle ioctl error */
}
```

4 Diagnostic

If the TPMC821 device driver does not work properly it is helpful to get some status information from the driver respective kernel. To get debug output from the driver enable the following symbols in '*tpmc821.c*' by replacing "#undef" with "#define":

```
#define DEBUG_TPMC821
#define DEBUG_TPMC821_INTR
```

The Linux */proc* file system provides information about kernel, resources, driver, devices and so on. The following screen dumps display information of a correct running TPMC821 driver (see also the proc man pages).

```
# tail –f /var/log/messages /* before modprobing the TPMC821 driver */


TEWS TECHNOLOGIES - TPMC821 INTERBUS Master G4 PMC -  version 1.0.0 (2007-
07-24)
TPMC821: Probe new device (vendor=0x1498, device=0x0335, type=821)...

/* after modprobing the TPMC821 driver */

# cat /proc/tews-tpmc821 /* advanced status information */
TEWS TECHNOLOGIES - TPMC821 INTERBUS Master G4 PMC -  version 1.0.0 (2007-
07-24)
Supported modules: TPMC821

Registered TPMC821 modules:
/dev/tpmc821_0
  Operating Mode: TPMC821_ASYNC
  DataTimeout   : 2
  MailboxTimeout: 10

# lspci -v
.../* TPMC821 */
  Bus  2, device   8, function  0:
    Class 0280: PCI device 1498:0136 (rev 0).
      IRQ 177.
      Non-prefetchable 32 bit memory at 0xff5fe400 [0xff5fe47f].
      I/O at 0xa800 [0xa87f].
      Non-prefetchable 32 bit memory at 0xff5fe000 [0xff5fe00f].
      Non-prefetchable 32 bit memory at 0xff5fdc00 [0xff5fddff].
...
```

```
# cat /proc/interrupts
           CPU0
  0:     871862   IO-APIC-edge      timer
  1:         10   IO-APIC-edge      i8042
  6:          5   IO-APIC-edge      floppy
  7:          0   IO-APIC-edge      parport0
  8:          1   IO-APIC-edge      rtc
  9:          0   IO-APIC-fasteoi   acpi
 12:        129   IO-APIC-edge      i8042
 14:      15701   IO-APIC-edge      ide0
 15:      30854   IO-APIC-edge      ide1
 17:     852832   IO-APIC-fasteoi   radeon@pci:0000:01:00.0, TPMC821
 18:       8895   IO-APIC-fasteoi   eth0
...

# cat /proc/iomem
...
  /* TPMC821 */
eb021000-eb02107f : 0000:00:0c.0
  eb021000-eb02107f : TPMC821
eb022000-eb022fff : 0000:00:0c.0
  eb022000-eb022fff : TPMC821
eb023000-eb02300f : 0000:00:0c.0
  eb023000-eb02300f : TPMC821
...
```