The Embedded I/O Company

TEWS TECHNOLOGIES

TPMC871-SW-42

VxWorks Device Driver

1 or 2 Slot PC-Card Interface

Version 3.1.x

User Manual

Issue 3.1.1

August 2009

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7 25469 Halstenbek, Germany
Phone: +49 (0) 4101 4058 0 Fax: +49 (0) 4101 4058 19

TPMC871-SW-42

VxWorks Device Driver

1 or 2 Slot PC-Card Interface

Supported Modules:
TPMC871
TPMC872
TCP872

Issue	Description	Date
2.0	First Issue	July 2000
2.1	Extended Configuration Description	August 2001
2.2	General Revision	January 2004
2.3	Errors in description corrected	March 2004
2.4	TPMC871 V3 and TPMC872 are now supported	March 2004
2.5.0	Installation and Setup modified, TCP872 is now supported	September 8, 2004
3.0.0	General Revision, Revised Driver startup, Device Initialization, System Setup (PCI Configuration Hints) and Application Hints	October 22, 2008
3.1.0	Description for support of VxWorks versions with XBD-support (VxWorks 6.2 and newer)	November 11, 2008
3.1.1	Company address information changed	August 29, 2009

Table of Contents

1 Introduction

1.1 Device Driver

The TPMC871-SW-42 VxWorks device driver software allows the operation of the supported Modules and PC-Cards.

The TPMC871-SW-42 device driver supports the following features:

- Hardware independent access functions setting up PC-Card controllers to allow and control access to the inserted PC-Cards.
- PC-Card ATA Device Driver to use PC-Flash-Cards

The TPMC871-SW-42 supports the modules listed below:

TPMC871	1 slot PC Card Interface PMC (V1.x … V3.x)	(PMC)
TPMC872	2 slot PC Card Interface PMC	(PMC)
TCP872	2 slot PC Card Interface compact PCI Card	(compactPCI)

In this document all supported modules and devices will be called TPMC871. Specials for certain devices will be advised.

This device driver software is split into three layers:

- Hardware dependent layer
- Calling interface for hardware independent interface
- Application layer or PC-Card device driver layer (PC Card ATA Disk driver)

The hardware dependent part mainly covers the functions of the socket services defined in the PC Card standard. These functions get information about the controllers setting and status as well as the capabilities of the device interface. Other functions handle the setup of the controllers. All these functions are called through one entry point with a function code and a variable number of parameters. Functions which are not defined in the PC Card standard are called directly. These functions will initialize the device interface and PC Card controllers.

The calling interface can handle calls to different PC Card interfaces. This will allow a simple change of the hardware layer or to use different hardware layers with the same PC Card device drivers.

The application and PC Card device driver layer handles the access to the PC Cards. This layer will call the hardware independent calling interface to allow access to the PC Cards.

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

TPMC871 User Manual or the User Manual of the corresponding Module

TPMC871 Engineering manual or the Engineering manual of the corresponding Module

2 Installation

Following files are located on the distribution media:

Directory path 'TPMC871-SW-42':

tp871_ss.c	Device interface source codes
tp871_sys.c	Hardware dependent functions source code
tp871pci.c	PCI configuration and MMU mapping function source code
tp871_ss.h	Device interface setup and include file
tp871_err.h	TPMC871 specific error codes
tp871d_ss.h	Device interface include file
tp871_sys.h	Hardware dependent functions include file
pcc_mng.c	Calling interface source code
pcc_mng.h	Calling interface and application include file
ata_drv.c	PC Card ATA disk device driver source code
ata_drv.h	ATA device driver and application include file
ata_def.h	Local ATA device driver include file
tp871pciConfig.c	PCI configuration table
tp871pciConfig.h	PCI configuration table structure include file
include/tdhal.h	Hardware dependent interface functions and definitions
pccExa.c	Example application for PC Card interface software
ataExa.c	Example application for PC Card ATA Disk device driver
pccConfig.c	Assignment table PCI device to adapter number
pccConfig.h	Assignment table structure include file
pciMapShow.c	Function showing the PCI devices and a Memory- and IO-Map (useful for PCI configuration)
TPMC871-SW-42-3.1.1.pdf	PDF copy of this manual
ChangeLog.txt	Release history
Release.txt	Release information

2.1 Include device driver into project

For including the TPMC871-SW-42 device driver into a project the following topics must be in mind:

(1) Copy the files from the distribution media into a subdirectory in your project path. (For example: ./TPMC871)

(2) Add the device drivers C-files to your project.

(3) Now the driver is included in the project and will be built with the project.

For a more detailed description of the project facility please refer to the VxWorks User's Guides of the IDE. (Tornado, Workbench, etc)

For VxWorks version 6.2 and newer, XBD support must be included. Please check that *INCLUDE_XBD* is defined in the project parameters (*prjParams.h*).

2.2 BSP dependent adjustments

The driver includes a file called *include/tdhal.h* which contains functions and definitions for BSP adaptation. It may be necessary to modify it for BSP specific settings. Most settings can be made automatically by conditional compilation set by the BSP header files, but some settings must be configured manually. There are two way of modification, first you can change the *include/tdhal.h* and define the corresponding definition and its value, or you can do it, using the command line option *–D*.

There are 3 offset definitions (*USERDEFINED_MEM_OFFSET*, *USERDEFINED_IO_OFFSET*, and *USERDEFINED_LEV2VEC*) that must be configured if a corresponding warning message appears during compilation. These definitions always need values. Definition values can be assigned by command line option *-D<definition>=<value>*.

definition	description
USERDEFINED_MEM_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI memory space access
USERDEFINED_IO_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI I/O space access
USERDEFINED_LEV2VEC	The value of this definition must be set to the difference of the interrupt vector (used to connect the ISR) and the interrupt level (stored to the PCI header)

Another definition allows a simple adaptation for BSPs that utilize a *pciIntConnect()* function to connect shared (PCI) interrupts. If this function is defined in the used BSP, the definition of *USERDEFINED_SEL_PCIINTCONNECT* should be enabled. The definition by command line option is made by *-D<definition>.*

Please refer to the BSP documentation and header files to get information about the interrupt connection function and the required offset values.

2.3 System dependent Adaptations

Before compiling software some important system dependent configurations have to be checked and if necessary be done.

- Check if the PCI configuration of the TPMC871 has been done and if it has set up completely. (Refer to *6.1 Check TPMC871 PCI configuration*)
- If there is a necessary the PCI configuration and PCI bridge configuration tables have to be modified matching to the used system. (Refer to *6.2 Setup TPMC871 PCI configuration manually*)
- To use the example applications or use them to make an own application the adapter assignment table has to be modified for System and application requirements. (Refer to *6.3 Example Application Configuration*)

2.4 Special installation for Intel x86 based targets

The TPMC871 device driver is fully adapted for Intel x86 based targets. This is done by conditional compilation directives inside the source code and controlled by the VxWorks global defined macro **CPU_FAMILY**. If the content of this macro is equal to *I80X86* special Intel x86 conforming code and function calls will be included.

2.5 Include PCI setup for TPMC871

The function tp871PciInit() has to be executed during system start. The function will make some necessary PCI configuration register setups on the TPMC871. These setups include setups which make the different version TPMC871 hardware versions compatible to the driver.

Some systems (typically Intel x86 compatible) do not map a general space for PCI devices. These systems will crash with an access fault (typically a page fault) if the device hardware is accessed. To solve this problem a MMU mapping entry has to be added for the required TPMC871 PCI memory spaces prior the MMU initialization (*usrMmuInit()*) is done. This part of the function is always enabled for Intel x86 compatible systems (*CPU_FAMILY* is defined to *I80X86*).

If MMU mapping is necessary for another system type, please enable the part of the function manually by enabling the conditional compilation for the CPU_FAMILY of the system.

The C source file **tp871pci.c** contains the function *tp871PciInit().* This routine finds out all TPMC871 devices and adds MMU mapping entries for all used PCI memory spaces. Please insert a call to this function after the PCI initialization is done and prior to MMU initialization (*usrMmuInit()*).

The right place to call the function *tp871PciInit()* is at the end of the function *sysHwInit()* in **sysLib.c** (it can be opened from the project *Files* window).

Be sure that the function is called prior to MMU initialization otherwise the TPMC871 PCI spaces remains unmapped and we got an access fault during driver initialization.

Please insert the following call at a suitable place in **sysLib.c**:

```
tp871PciInit();
```

Modifying the sysLib.c file will change the sysLib.c in the BSP path. Remember this for future projects and recompilations.

Avoiding compilation a warning a function prototype should be defined at top of the calling function. There for insert:

```
void tp871PciInit();
```

The Function *tp871PciInit()* was designed for and tested on generic Pentium targets. If you use another BSP please refer to BSP documentation or contact the technical support for required adaptation.

If you got strange errors after system startup with the new build system please carry out a VxWorks *build clean* and *build all*.

2.6 Increase the Number Maximum Used Adapters

If there are more than six PC Card adapters used, there are two defines that have to be modified to allow more adapters. Simply increase the value of the definition to value higher or some the really maximum used adapters. Count one adapter for every TPMC871 and two adapters for every TPMC872 or TCP872. E.g.: You have to specify 8 adapters if you have 3 TPMC872 (3*2 adapters) and 2 TPMC871 (2*1 adapter) mounted to your system.

The first define to be modified is *TPMC871_MAXADAPTER* in *tp871_ss.h*:

```
#define    TPMC871_MAXADAPTER        6
```

The second define is *PCC_MAXADAPTER* in *pcc_mng.h*:

```
#define    PCC_MAXADAPTER            6
```

2.7 System resource requirement

System resource requirements are split into two parts, first the requirements for PC Card interface functions and second for PC Card ATA device driver. Requirements of both depend on the number of specified the maximum number of adapters.

Memory and Stack usage may differ from system to system, depending on the used compiler and its setup.

The following formula shows the way to calculate the common requirements of the driver and devices.

*<total requirement> = <driver requirement> + (<number of devices> * <device requirement>)*

The maximum usage of some resources is limited by adjustable parameters. If the application and driver exceed these limits, increase the according values in your project.

2.7.1 System resource requirements of PC Card Interface

The table gives an overview over the system resources that will be needed by the PC Card interface.

Resource	Driver requirement	Devices requirement
Memory	< 1 KB	< 1 KB
Stack	< 1 KB	---
PCI resources	---	see chapter PCI configuration

2.7.2 System resource requirements of PC Card ATA Driver

The table gives an overview over the system resources that will be needed by the driver.

Resource	Driver requirement	Devices requirement
Memory	< 1 KB	< 1 KB
Stack	< 1 KB	---

The table just shows the requirement of the PC Card ATA Driver function, requirements for VxWorks standard disk drivers are not considered.

3 PC Card Interface Functions

These functions are needed to setup the PC Card interfaces or to get information about the setup and the capabilities.

These functions are split into two parts. The first part is hardware dependent and special for the TPMC871. The other part is a device independent interface, which allows a unified access to different kind of PC Card slots.

A big part of the hardware dependent functions are the socket functions which will be described later in the chapter “Socket Functions”. These functions are defined in the PC Card standard as the socket services.

3.1 Dependent Functions

This chapter describes the functions of the TPMC871 dependent layer. This part of the driver accomplishes the access to the PC Card controller on the TPMC871.

3.1.1 tpmc871_ss_init()

NAME

tpmc871_ss_init() initializes socket interface.

SYNOPSIS

```
unsigned long     tpmc871_ss_init
(
      void
)
```

DESCRIPTION

This function initializes the local data of the TPMC871 socket interface driver. This function must be called before any other function of the hardware dependent interface is called.

EXAMPLE

```
#include           “tp871_ss.h”

unsigned long      result;

…

result = tpmc871_ss_init();
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

TP871_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter “Error Codes”)

3.1.2 tpmc871_init()

NAME

tpmc871_init() initializes adapter and gets entry point.

SYNOPSIS

```
unsigned long     tpmc871_init
(
      int               busNo,
      int               devNo,
      int               funcNo,
      ADAPTER           *adapter,
      unsigned long     (**entry)()
)
```

DESCRIPTION

This function must be called for every adapter (TPMC871 module) before any of the socket functions is called for the adapter and after the *tpmc871_ss_init()* is called. This function allocates and initializes memory for the device control block and makes a basic setup of the PC Card controller. This function reads the PCI configuration and stores the controller access areas.

PARAMETERS

The input parameters **busNo**, **devNo** and **funcNo** specify the slot where the TPMC871 is mounted to. **funcNo** must always be zero, because there is only one function implemented on the TPMC871.

This function returns the **entry** point of the socket functions and the local **adapter** number of the specified TPMC871. Both of the parameters will be needed for calling the socket functions directly and for connecting this adapter (TPMC871) with the calling interface (see chapter “Calling Interface Functions”).

EXAMPLE

```
#include          “tp871_ss.h”
#include          “pcc_mng.h”

unsigned long     result;
int               busNo, devNo, funcNo;
ADAPTER           locAdapter;
unsigned long     (*locEntry)();

…

/*-------------------------------------------------------
  Initialize TPMC871 mounted to
    PCI bus 0, PCI device 16 and PCI function 0
  -------------------------------------------------------*/
result = tpmc871_init(  0, 16, 0, &locAdapter, &locEntry);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
     /* use returned parameters to call the socket functions */
}
```

RETURNS

TP871_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter “Error Codes”)

3.1.3 Entry Point Function

NAME

tpmc871_entry() Entry Point Function for socket function of the TPMC871

SYNOPSIS

```
LOCAL unsigned long   tpmc871_entry
(
      unsigned long     function,
      ...
)
```

DESCRIPTION

This function is local to the tpmc871 dependent code. The only way to call this function is to get the entry point of this function by calling the *tpmc871_init()* function and using the returned address for a referenced function call. This function will call the specified socket function with the needed number of parameters. Therefore the parameter list of this function is variable.

PARAMETERS

The parameter **function** specifies which socket function shall be called.

The length and contents of parameter list depends on the called socket function.

The length of the parameter list will not be checked by this function.

EXAMPLE

```
#include          "tp871_ss.h"
#include          "pcc_mng.h"

unsigned long     result;
SKTBITS           Sockets;

…

/*-------------------------------------------------------
  Call the entry point locEntry and execute the socket
  function Acknowledge Interrupt for the adapter specified
  in locAdapter. locEntry and locAdapter were returned
  by tpmc871_init
  -------------------------------------------------------*/
result = locEntry( PCC_ACK_INTERRUPT,
                   (unsigned long)locAdapter,
                   (unsigned long)&Sockets);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

TP871_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter "Error Codes")

3.1.4 Socket Functions

The socket functions will be described below in a separate chapter “Socket Functions”. Not all functions are necessary for every kind of PC Card controller. Therefore some of the functions are not implemented. The following list shows the possible functions and if functions are implemented for the TPMC871 or not.

Function Code	Impl.	Function Code	Impl.
PCC_GET_ADP_CNT	Yes	PCC_GET_SS_INFO	No
PCC_INQ_ADAPTER	Yes	PCC_INQ_EDC	No
PCC_GET_ADAPTER	Yes	PCC_GET_EDC	No
PCC_SET_ADAPTER	Yes	PCC_SET_EDC	No
PCC_INQ_WINDOW	Yes	PCC_START_EDC	No
PCC_GET_WINDOW	Yes	PCC_PAUSE_EDC	No
PCC_SET_WINDOW	Yes	PCC_RESUME_EDC	No
PCC_GET_PAGE	Yes	PCC_STOP_EDC	No
PCC_SET_PAGE	Yes	PCC_READ_EDC	No
PCC_INQ_SOCKET	Yes	PCC_PRIOR_HANDLER	No
PCC_GET_SOCKET	Yes	PCC_SS_ADDR	No
PCC_SET_SOCKET	Yes	PCC_ACCESS_OFFSETS	No
PCC_GET_STATUS	Yes	PCC_ACCESS_CONFIG	No
PCC_RESET_SOCKET	Yes	PCC_INQ_BWINDOW	No
PCC_GET_VENDOR_INFO	Yes	PCC_GET_BWINDOW	No
PCC_ACK_INTERRUPT	Yes	PCC_SET_BWINDOW	No
		PCC_VEND_SPECIFIC	No

3.2 Calling Interface Functions

This chapter describes the functions of the calling interface dependent layer. This part of the software implements a calling interface which allows applications and PC Card drivers to use different PC Card controllers without changing the application or driver.

3.2.1 pcc_init()

NAME

pcc_init() initializes calling interface.

SYNOPSIS

```
unsigned long     pcc_init
(
      void
)
```

DESCRIPTION

This function initializes the local data of the calling interface. This function must be called once before any other function of the calling interface is called.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;

…

result = pcc_init();
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter “Error Codes”)

3.2.2 pcc_adapter_init()

NAME

pcc_adapter_init() connects an adapter to the calling interface.

SYNOPSIS

```
unsigned long     pcc_adapter_init
(
      ADAPTER         adapter,
      ADAPTER         adapter_mod,
      unsigned long   (*entry)()
)
```

DESCRIPTION

This function connects an adapter to the calling interface. This function must be called before the *pcc_entry()* function is called for the specified adapter, but behind the *pcc_adapter_init()* function.

PARAMETERS

The parameter **adapter** specifies the adapter number, which shall be used in the future calls.

The parameter **adapter_mod** specifies the local adapter number. **adapter_mod** parameter is not the same as the **adapter** parameter. To get the local adapter number the hardware dependent function *tpmcxxx_init()* has to be called.

The parameter **entry** specifies the local entry point of the hardware dependent layer. The entry point will be get, if the hardware dependent function *tpmcxxx_init()* is called.

EXAMPLE

```
#include           “pcc_mng.h”

unsigned long      result;

…

/*-------------------------------------------------------
  Connect an initialized adapter to the calling interface.
    The adapter shall be called as adapter 4.
    locEntry and locAdapter are values returned by
    tpmcxxx_init()
  -------------------------------------------------------*/
result = pcc_adapter_init(   4, locAdapter, locEntry);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter “Error Codes”)

3.2.3 pcc_entry()

NAME

pcc_entry() Calling interface function

SYNOPSIS

```
unsigned long     pcc_entry
(
     unsigned long     function,
     ...
)
```

DESCRIPTION

This function will call the socket function via the local entry points of the hardware dependent interfaces.

PARAMETERS

The parameter **function** specifies which socket function shall be called.

The length and contents of parameter list depends on the called socket function.

The length of the parameter list will not be checked by this function.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
SKTBITS           Sockets;

…

/*-------------------------------------------------------
  Call the Acknowledge interrupt function for the connected
  adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_ACK_INTERRUPT,
                        4,
                        (unsigned long)&Sockets);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR (0) = if no error occurred

Otherwise an appropriate error code (see below chapter “Error Codes”)

4 Socket Functions

This chapter will describe the socket functions which are supported by the TPMC871 device interface. The unsupported functions are only listed with a short description. If different PC Card interfaces are used, it may be possible, that there are different functions supported. Please refer also to the other software manuals.

These functions are called via the calling interface in the way described before in the chapter *pcc_entry().*

4.1 Supported Functions

4.1.1 GetAdapterCount

NAME

GetAdapterCount returns the number of activated PC Card adapter located in the system.

FUNCTION CODE

PCC_GET_ADP_CNT (0x80)

PARAMETERS

COUNT	*TotalAdapters
SIGNATURE	Signature

DESCRIPTION

This function identifies the PC Card socket functions and returns the number of activated PC Card adapters in the system.

PARAMETERS

The parameter **TotalAdapters** points to the count of activated PC Card adapters.

The **Signature** is a field of two characters where an ident code for the socket interfaces is returned. The value returned in signature is always *'SS'* if a socket interface is present. This field must be '0' before this function is called.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
ADAPTER           TotalAdapters;
SIGNATURE         Signature;

…

/*-------------------------------------------------------
  Get adapter count
  -------------------------------------------------------*/
Signature[0] = Signature[1] = 0;
result = pcc_entry (    PCC_GET_ADP_CNT,
                        (unsigned long)&TotalAdapters,
                        (unsigned long)Signature);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error

4.1.2 InquireAdapter

NAME

InquireAdapter Get the capabilities of the specified adapter

FUNCTION CODE

PCC_INQ_ADAPTER (0x84)

PARAMETERS

ADAPTER Adapter
PTR pBuffer
COUNT *NumSockets
COUNT *NumWindows
COUNT *NumEDCs
COUNT *NumBridgeWindows

DESCRIPTION

This function informs about the technical capabilities of the specified adapter. The information shows the possible interrupts, possible power settings and special capabilities (for example: power down mode).

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **pBuffer** is a pointer to the *AISTRUCT* structure where the adapter capabilities will be stored.

Data structure *AISTRUCT*:

WORD	wBufferLength	This is an input entry and must be set to the maximal length of the *CharTable*, *wNumPwrEntries* and the *PwrEntry* array.
WORD	wDataLength	The function will return in this entry to the real length of the *CharTable*, *wNumPwrEntries* and the *PwrEntry* array.
ACHARTBL	CharTable	This structure holds the information about the interrupt and special abilities of the PC Card controller.
WORD	wNumPwrEntries	Number of elements in the *PwrEntry* array
PWRENTRY	PwrEntry[1]	Array with *wNumPwrEntries* elements, which identifies the possible power settings and levels.

Data structure *ACHARTBL*:

FLAGS8	AdpCaps	This entry shows the abilities of the adapter. Defined flags are: PCC_AC_IND – Shared indicators PCC_AC_PWR – No individual power levels PCC_AC_DBW – Same data bus width for all windows PCC_AC_CARDBUS – All sockets are CardBus PC Card capable
BYTE	CacheLineSize	Specifies the host system cache line size in units of 32 bit words. This value is zero for non CardBus PC Card adapters.
FLAGS32	ActiveHigh	This bit field describes which interrupt routings can possibly used for high active interrupts.
FLAGS32	ActiveLow	This bit field describes which interrupt routings can possibly used for low active interrupts.

Data structure *PWRENTRY*:

PWRINDEX	PowerLevel	This entry describes the power level in 1/10V (Example: 50 means 5.0V
FLAGS8	ValidSignals	This bit field describes the power lines which can be set to the previously defined power level. Defined flags are: PCC_VCC – Power index is valid for Vcc PCC_VPP1 – Power index is valid for Vpp1 PCC_VPP2 – Power index is valid for Vpp2

The parameter **NumSockets** returns the total number of sockets provided by the specified adapter.

The parameter **NumWindows** returns the total number of windows provided by the specified adapter.

The parameter **NumEDCs** returns the total number of EDCs provided by the specified adapter.

The parameter **NumBridgeWindows** returns the total number of bridge windows provided by the specified adapter.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
char              pBuffer[50];
AISTRUCT          *pAIStruct;
COUNT             NumSockets,
                  NumWindows,
                  NumEDCs,
                  NumBridgeWindows;

…

/*-------------------------------------------------------
  Get capabilities of adapter 4
  -------------------------------------------------------*/
pAIStruct = (AISTRUCT*)pBuffer;
pAIStruct->wBufferLength = 50;        /* Setup Bufferlength */
result = pcc_entry (    PCC_INQ_ADAPTER,
                        (unsigned long)4,
                        (unsigned long)pBuffer,
                        (unsigned long)&NumSockets,
                        (unsigned long)&NumWindows,
                        (unsigned long)&NumEDCs,
                        (unsigned long)&NumBridgeWindows);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter

4.1.3 GetAdapter

NAME

GetAdapter reads the actual configuration of the specified adapter.

FUNCTION CODE

PCC_GET_ADAPTER (0x85)

PARAMETERS

ADAPTER Adapter
FLAGS8 *State
IRQ *SCRouting

DESCRIPTION

The configuration of the specified adapter is read. The parameter values are compatible to the parameters of the *SetAdapter* function.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **State** will be filled with flags specifying the actual configuration. The following flags are defined:

PCC_AS_POWERDOWN	Hardware is in power conserve mode.
PCC_AS_MAINTAIN	Adapter and socket configuration are maintained while in power conserve mode.

The parameter **SCRouting** will return the actual setup for socket interrupts. The following flags are defined:

PCC_IRQ_HIGH
The status change interrupt is active high.

PCC_IRQ_ENABLE
The status change interrupt is enabled.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
FLAGS8            State;
IRQ               SCRouting;

…

/*-------------------------------------------------------
  Get actual configuration of adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_GET_ADAPTER,
                        (unsigned long)4,
                        (unsigned long)&State,
                        (unsigned long)&SCRouting);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter

4.1.4 SetAdapter

NAME

SetAdapter configures the specified adapter.

FUNCTION CODE

PCC_SET_ADAPTER (0x86)

PARAMETERS

ADAPTER Adapter
FLAGS8 State
IRQ SCRouting

DESCRIPTION

The specified adapter will be configured with this function. This function should be called before any sockets, windows or pages are set.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **State** specifies the new configuration of the adapter. The following flags are defined:

PCC_AS_POWERDOWN	Hardware is set to power conserve mode.
PCC_AS_MAINTAIN	Adapter and socket configuration are maintained while in power conserve mode.

The parameter **SCRouting** specifies how socket interrupts will be setup. The following flags are defined:

PCC_IRQ_HIGH	The status change interrupt is active high.
PCC_IRQ_ENABLE	The status change interrupt is enabled.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;

…

/*-------------------------------------------------------
  Enable status change interrupt for adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_SET_ADAPTER,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)PCC_IRQ_ENABLE);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_IRQ	unsupported state or IRQ level specified

4.1.5 InquireWindow

NAME

InquireWindow gets the capabilities of the specified window.

FUNCTION CODE

PCC_INQ_WINDOW (0x87)

PARAMETERS

ADAPTER Adapter
WINDOW Window
PTR pBuffer
FLAGS8 *WndCaps
SKTBITS *Sockets

DESCRIPTION

This function informs about the capabilities of the specified window and for which sockets the window can be used.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Window** specifies the window on the specified PC Card adapter.

The parameter **pbuffer** is a pointer to the *WISTRUCT* structure where the window capabilities and limits are returned.

Data structure *WISTRUCT*:

WORD	wBufferLength	This is an input entry and must be set to the maximal length of the WinTable array.
WORD	wDataLength	The function will return in this entry to the real length of the CharTable array.
WINTBL	WinTable[1]	*WINTBL* is defined as a union where different information are set for memory in a *MEMWINTBL* structure or for I/O windows in a *IOWINTBL*. If the use as memory and I/O window is possible for the window, the memory description is always the first in the array.

Data structure *MEMWINTBL*:

FLAGS16	MemWndCaps	Flags specifying special abilities of the specified memory window. Defined flags are:	
		PCC_WC_BASE	Programmable base address for the window
		PCC_WC_SIZE	Programmable size for the window
		PCC_WC_WENABLE	Window can be enabled or disabled without reprogramming the characteristics.
		PCC_WC_8BIT	Window can be programmed for 8 bit data transfers.
		PCC_WC_16BIT	Window can be programmed for 16 bit data transfers
		PCC_WC_BALIGN	The base address must be aligned to a multiple of the window size.
		PCC_WC_POW2	The window size must be a value power of two.
		PCC_WC_CALIGN	The offsets must align to the size of the window.
		PCC_WC_PAVAIL	Paging is available on the hardware.
		PCC_WC_PSHARED	Paging hardware is shared with another window.
		PCC_WC_PENABLE	Paging can be enabled and disabled.
		PCC_WC_WP	The window can write protected.
BASE	FirstByte	Value defines the first accessible memory address of the specified window.	
BASE	LastByte	Value defines the last accessible memory address of the specified window.	
SIZE	MinSize	Value defines the minimum size for the specified memory window.	
SIZE	MaxSize	Value defines the maximum size for the specified memory window.	
SIZE	RepGran	Memory sizes have to be set to a multiple of the size returned in this entry.	
SIZE	RepBase	Memory bases have to start at a multiple of this value.	
SIZE	RepOffset	Memory offsets have to be set to a multiple of this value.	
SPEED	Slowest	Entry shows the slowest possible memory access. The value is specified in the data structure *SPEED* which is described below.	
SPEED	Fastest	Entry shows the fastest possible memory access. The value is specified in the data structure *SPEED* which is described below.	

Data structure *SPEED*:

BYTE	speed	This field specifies the speed or indicates that the extended speed setting shall be used. Possible values are defined for this field:

Code	Meaning
0	reserved
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5..6	reserved
7	use extended speed

BYTE	extSpeedMant	This field specifies the mantissa of the extended speed setting. How this value is coded, is described below.
BYTE	extSpeedExp	This field specifies the exponent of the extended speed setting. The values are defined as follows:

Code	extSpeedMant	extSpeedMant
0x0	reserved	1 ns
0x1	1.0	10 ns
0x2	1.2	100 ns
0x3	1.3	1 µs
0x4	1.5	10 µs
0x5	2.0	100 µs
0x6	2.5	1 ms
0x7	3.0	10 ms
0x8	3.5	
0x9	4.0	
0xA	4.5	
0xB	5.0	
0xC	5.5	
0xD	6.0	
0xE	7.0	
0xF	8.0	

Data structure *IOWINTBL*:

FLAGS16	IOWndCaps	Flags specifying special abilities of the specified I/O window. Defined flags are:	
		PCC_WC_BASE	Programmable base address for the window
		PCC_WC_SIZE	Programmable size for the window
		PCC_WC_WENABLE	Window can be enabled or disabled without reprogramming the characteristics.
		PCC_WC_8BIT	Window can be programmed for 8 bit data transfers.
		PCC_WC_16BIT	Window can be programmed for 16 bit data transfers.
		PCC_WC_BALIGN	Base address must be aligned to a multiple of the window size.
		PCC_WC_POW2	Window size must be a value power of two.
		PCC_WC_INPACK	Window allows the INPACK# signal.
		PCC_WC_EISA	Window allows EISA-like I/O mapping.
		PCC_WC_CENABLE	EISA-like common address space may be ignored.
BASE	FirstByte	Value defines the first accessible I/O address of the specified window.	
BASE	LastByte	Value defines the last accessible I/O address of the specified window.	
SIZE	MinSize	Value defines the minimum size for the specified I/O window.	
SIZE	MaxSize	Value defines the maximum size for the specified I/O window.	
SIZE	RepGran	I/O sizes have to be set to a multiple of the size returned in this entry.	
COUNT	AddrLines	Entry returns the number of interpreted address lines at the controller, typically ten (10) or sixteen (16).	
FLAGS8	EISASlot	Entry is used for EISA bus settings, so it will be unused on TPMC871 modules	

The parameter **WndCaps** will be filled with flags describing the capabilities of the window. The following flags are defined for this value:

PCC_WC_COMMON	The window can map PC Card common memory into system memory space.
PCC_WC_ATTRIBUTE	The window can map PC Card attribute memory into system memory space.
PCC_WC_IO	The window can map PC Card I/O ports into system I/O space.
PCC_WC_WAIT	The window supports the WAIT# signal.

The **Sockets** parameter is filled with bit field marking the sockets which can be accessed by the window.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
char              pBuffer[80];
WISTRUCT          *pWIStruct;
FLAGS8            WndCaps;
SKTBITS           Sockets;

…

/*-------------------------------------------------------
  Get capabilities of window 2 on adapter 4
  -------------------------------------------------------*/
pWIStruct = (WISTRUCT*) pBuffer;
pWIStruct->wBufferLength = 80;        /* Setup Bufferlength */
result = pcc_entry (    PCC_INQ_WINDOW,
                        (unsigned long)4,
                        (unsigned long)2,
                        (unsigned long)pBuffer,
                        (unsigned long)&WndCaps,
                        (unsigned long)&Sockets);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_WINDOW	invalid window

4.1.6 GetWindow

NAME

GetWindow reads the actual configuration of the specified window.

FUNCTION CODE

PCC_GET_WINDOW (0x88)

PARAMETERS

ADAPTER	Adapter
WINDOW	Window
SOCKET	*Socket
SIZE	*Size
FLAGS8	*State
SPEED	*Speed
BASE	*Base

DESCRIPTION

The configuration of the specified window is read. The parameter values are compatible to the parameters of the *SetWindow* function.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Window** specifies the window of the specified PC Card adapter.

The parameter **Socket** returns the socket number the window is currently assigned to.

The **Size** parameter returns the current window size.

The **State** flag field returns the actual window state. The value can be a combination of the following defined flags:

PCC_WS_IO	The window maps registers from the PC Card into the host I/O space.
PCC_WS_ENABLED	The window is enabled for mapping card's address space into host memory or I/O space.
PCC_WS_16BIT	The window is programmed for 16 bit data width.
PCC_WS_PAGED	The window is divided into pages.
PCC_WS_EISA	The window is set for EISA I/O mapping.
PCC_WS_CENABLE	Accesses to I/O ports generates card enables.

The parameter **Speed** is a pointer to the data structure which will be filled with data specifying the current access speed of the window.

Data structure *SPEED*:

BYTE	speed	This field specifies the speed or indicates that the extended speed setting shall be used. Possible values are defined for this field:

Code	**Meaning**
0	reserved
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5..6	reserved
7	use extended speed

BYTE	extSpeedMant	This field specifies the mantissa of the extended speed setting. How this value is coded, is described below.
BYTE	extSpeedExp	This field specifies the exponent of the extended speed setting. The values are defined as follows:

Code	**extSpeedMant**	**extSpeedMant**
0x0	reserved	1 ns
0x1	1.0	10 ns
0x2	1.2	100 ns
0x3	1.3	1 µs
0x4	1.5	10 µs
0x5	2.0	100 µs
0x6	2.5	1 ms
0x7	3.0	10 ms
0x8	3.5	
0x9	4.0	
0xA	4.5	
0xB	5.0	
0xC	5.5	
0xD	6.0	
0xE	7.0	
0xF	8.0	

The parameter **Base** specifies the base address the window is mapped to.

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
FLAGS8            State;
SOCKET            Socket;
SPEED             Speed;
SIZE              Size;
BASE              Base;

…

/*-------------------------------------------------------
  Get actual configuration window 2 on adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_GET_WINDOW,
                        (unsigned long)4,
                        (unsigned long)2,
                        (unsigned long)&Socket,
                        (unsigned long)&Size,
                        (unsigned long)&State,
                        (unsigned long)&Speed,
                        (unsigned long)&Base);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_WINDOW	invalid window

4.1.7 SetWindow

NAME

SetWindow configures the specified window.

FUNCTION CODE

PCC_SET_WINDOW (0x89)

PARAMETERS

ADAPTER	Adapter
WINDOW	Window
SOCKET	Socket
SIZE	Size
FLAGS8	State
SPEED	Speed
BASE	Base

DESCRIPTION

The specified window will be configured. This function allows the selection of a special memory area where the PC Card shall be mapped to.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Window** specifies the window of the specified PC Card adapter.

The parameter **Socket** specifies the socket the window is assigned to.

The **Size** parameter specifies the new size of the window.

The **State** flag field specifies the window configuration. The value can be a combination of the following defined flags, if they are valid for the window (see chapter “InquireWindow”):

PCC_WS_IO	The window maps registers from the PC Card into the host I/O space.
PCC_WS_ENABLED	The window shall be enabled for mapping card’s address space into host memory or I/O space. Memory access must also be enabled with the *SetPage* function (see chapter “SetPage”).
PCC_WS_16BIT	The window is programmed for 16 bit data width.
PCC_WS_PAGED	The window is divided into pages.
PCC_WS_EISA	The window is set for EISA I/O mapping.
PCC_WS_CENABLE	Accesses to I/O ports generates card enables.

The parameter **Speed** is a pointer to the data structure which will setup the access speed of the window. The TPMC871 allows access times up to 600nsec.

Data structure *SPEED*:

BYTE	speed	This field specifies the speed or indicates that the extended speed setting shall be used. Possible values are defined for this field:

Code	**Meaning**
0	reserved
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5..6	reserved
7	use extended speed

BYTE	extSpeedMant	This field specifies the mantissa of the extended speed setting. How this value is coded, is described below.
BYTE	extSpeedExp	This field specifies the exponent of the extended speed setting. The values are defined as follows:

Code	**extSpeedMant**	**extSpeedExp**
0x0	reserved	1 ns
0x1	1.0	10 ns
0x2	1.2	100 ns
0x3	1.3	1 µs
0x4	1.5	10 µs
0x5	2.0	100 µs
0x6	2.5	1 ms
0x7	3.0	10 ms
0x8	3.5	
0x9	4.0	
0xA	4.5	
0xB	5.0	
0xC	5.5	
0xD	6.0	
0xE	7.0	
0xF	8.0	

The parameter **Base** specifies the base address the window will be mapped to.

EXAMPLE

```
#include           “pcc_mng.h”

unsigned long      result;
SPEED              Speed;

…

/*-------------------------------------------------------
  Set window 2 on adapter 4 at socket 0 with the following
  capabilities:
    - Size:  0x1000
    - State: enable window
    - Speed: 200ns
    - Base:  0xfd100000
  -------------------------------------------------------*/
Speed.speed        = 0x02;
Speed.extSpeedMant = 0;
Speed.extSpeedExp  = 0;
result = pcc_entry (    PCC_SET_WINDOW,
                        (unsigned long)4,
                        (unsigned long)2,
                        (unsigned long)0,
                        (unsigned long)0x1000,
                        (unsigned long)PCC_WS_ENABLED,
                        (unsigned long)&Speed,
                        (unsigned long)0xfd100000);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	No error
PCC_BAD_ADAPTER	Invalid adapter
PCC_BAD_ATTRIBUTE	The requested state does not match to the windows capabilities.
PCC_BAD_BASE	The base address is invalid.
PCC_BAD_SIZE	The window size is invalid.
PCC_BAD_SOCKET	The window can not be assigned to the specified socket.
PCC_BAD_SPEED	Illegal speed specified
PCC_BAD_TYPE	*PCC_WS_IO* setting is invalid.
PCC_BAD_WINDOW	Invalid window

4.1.8 GetPage

NAME

GetPage reads the actual configuration of the specified page.

FUNCTION CODE

PCC_GET_PAGE (0x8A)

PARAMETERS

ADAPTER	Adapter
WINDOW	Window
SOCKET	Page
FLAGS8	*State
OFFSET	*Offset

DESCRIPTION

This function reads the configuration of the specified page for the specified window. The parameter values are compatible to the parameter values which have to be set in the *SetPage* function.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Window** specifies the window of the specified PC Card adapter.

The parameter **Page** specifies the page of the specified window.

The **State** flag field returns the actual page state. The value can be a combination of the following defined flags:

PCC_PS_ATTRIBUTE	The page maps attribute memory into the host systems common memory.
PCC_PS_ENABLED	The page is enabled. Access to the specified address area is only enabled if the window and page are enabled.
PCC_PS_WP	The paged area is write protected.

The parameter **Offset** specifies the PC Card local offset. This value specifies the local address on the PC Card.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
FLAGS8             State;
OFFSET             Offset;

…

/*-------------------------------------------------------
  Get actual configuration of page 0 for window 2 on
  adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_GET_PAGE,
                        (unsigned long)4,
                        (unsigned long)2,
                        (unsigned long)0,
                        (unsigned long)&State,
                        (unsigned long)&Offset);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_PAGE	invalid page
PCC_BAD_WINDOW	invalid window

4.1.9 Set Page

NAME

SetPage configures the specified page on the specified window.

FUNCTION CODE

PCC_SET_PAGE (0x8B)

PARAMETERS

ADAPTER	Adapter
WINDOW	Window
SOCKET	Page
FLAGS8	State
OFFSET	Offset

DESCRIPTION

This function configures the page(s) of a window, selects the start address of the mapped area of the PC Card and enables or disables the PC Card accesses.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Window** specifies the window of the specified PC Card adapter.

The parameter **Page** specifies the page of the specified window.

The **State** flag field specifies the new page state. The value can be set as a combination of the following defined flags:

PCC_PS_ATTRIBUTE	The page shall map attribute memory into the host systems common memory.
PCC_PS_ENABLED	The page shall be enabled. Access to the specified address area is only enabled if the window and page are enabled.
PCC_PS_WP	The paged area shall be write protected.

The parameter **Offset** specifies the PC Card local offset. This value specifies the local address on the PC Card.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;

…

/*-------------------------------------------------------
  Set actual configuration of page 0 for window 2 on
  adapter 4
    - Attribute mapping
    - page enable
    - offset = 0x100
  -------------------------------------------------------*/
result = pcc_entry (    PCC_SET_PAGE,
                        (unsigned long)4,
                        (unsigned long)2,
                        (unsigned long)0,
                        (unsigned long)
                             (PCC_PS_ATTRIBUTE |
                              PCC_PS_ENABLED),
                        (unsigned long)0x100);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_ATTRIBUTE	state with invalid attribute
PCC_BAD_OFFSET	offset is invalid
PCC_BAD_PAGE	invalid page
PCC_BAD_WINDOW	invalid window

4.1.10 InquireSocket

NAME

InquireSocket gets the capabilities of the specified socket.

FUNCTION CODE

PCC_INQ_SOCKET (0x8C)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket
PTR	pBuffer
FLAGS8	*SCIntCaps
FLAGS8	*SCRptCaps
FLAGS8	*CtlIndCaps

DESCRIPTION

This function describes the capabilities of the specified socket. Information about special abilities, possible interrupt routings and about the events creating interrupts or status changes will be returned.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Socket** specifies the PC Card socket on the specified adapter.

The parameter **pBuffer** is a pointer to the *SISTRUCT* structure where the socket capabilities will be stored.

Data structure *SISTRUCT*:

WORD	wBufferLength	This is an input entry and must be set to the maximal length of the *CharTable*.
WORD	wDataLength	The function will return in this entry to the real length of the *CharTable*.
SCHARTBL	CharTable	This structure holds the information about the interrupt and special abilities of the specified PC Card socket.

Data structure *SCHARTBL*:

FLAGS16	SktCaps	Flags specifying special capabilities of the specified I/O socket. Defined Flags are:	
		PCC_IF_MEMORY	Socket supports memory-only interface.
		PCC_IF_IO	Socket supports I/O and memory interface.
		PCC_IF_CB	Socket supports CardBus PC Card.
		PCC_IF_33VCC	Socket supports 3.3 V interface.
		PCC_IF_XXVCC	Socket supports X.X V interface.
		PCC_IF_VSKAY	Socket supports Low Voltage Key.
		PCC_IF_DMA	Socket supports 16 bit PC Card DMA transfers.

The parameter **SCIntCaps** returns which events can trigger the status change interrupt and the parameter **SCRptCaps** returns which events can be reported. A combination of the following defined flags will be returned:

PCC_SBM_WP	PC Card write protect
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism.
PCC_SBM_EJECT	External generated indicating a request to eject the card.
PCC_SBM_INSERT	External generated indicating a request to insert the card.
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **CtlIndCaps** returns which controls and indicators are supported for the socket. A combination of the following defined flags will be returned:

PCC_SBM_WP	PC Card write protect state
PCC_SBM_LOCKED	External generated indicator for the state of an externally mechanical or electrical for the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	Control for motor to eject a PC Card
PCC_SBM_INSERT	Control for motor to insert a PC Card
PCC_SBM_LOCK	Control for Card Lock
PCC_SBM_BATT	Indication for BVD1 and BVD2
PCC_SBM_BUSY	Indicator for showing the card in-use
PCC_SBM_XIP	Indicator for eXecution-In-Place application in progress

EXAMPLE

```
#include          “pcc_mng.h”

unsigned long     result;
char              pBuffer[50];
SISTRUCT          *pSIStruct;
FLAGS8            SCIntCaps,
                  SCRptCaps,
                  CtlIndCaps;

…

/*-------------------------------------------------------
  Get capabilities of Socket 0 on adapter 4
  -------------------------------------------------------*/
pSIStruct = (SISTRUCT*)pBuffer;
pSIStruct->wBufferLength = 50;        /* Setup Bufferlength */
result = pcc_entry (    PCC_INQ_SOCKET,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)pBuffer,
                        (unsigned long)&SCIntCaps,
                        (unsigned long)&SCRptCaps,
                        (unsigned long)&CtlIndCaps);
…
```

```
…
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_SOCKET	invalid socket

4.1.11 GetSocket

NAME

GetSocket reads the actual configuration of the specified socket.

FUNCTION CODE

PCC_GET_SOCKET (0x8D)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket
FLAGS8	*SCIntMask
PWRINDEX	*Vcontrol
PWRINDEX	*VccLevel
PWRINDEX	*VppLevels
FLAGS8	*State
FLAGS8	*CtlInd
IRQ	*IREQRouting
FLAGS8	*IFType
WORD	*IFIndex

DESCRIPTION

This function reads the actual configuration of the specified socket. The parameters are compatible to the parameters which have to be set in the *SetSocket function*.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Socket** specifies the socket on the specified PC Card adapter.

The parameter **SCIntMask** returns the events generated by status changes when they occur on the socket. The value can be a combination of the following values:

PCC_SBM_WP	PC Card write protect
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **Vcontrol** returns voltage control settings. This can be a combination of the following defined values:

PCC_VXTL_CISREAD	Vcc and Vpp are controlled by the Vcc and Vpp fields.
PCC_VCTL_OVERRIDE	If set the Vcc level does not match to the value indicated by the voltage sense

(The following values are mutually exclusive)

PCC_VCTL_50V	Use 5.0V for CIS read
PCC_VCTL_33V	Use 3.3V for CIS read
PCCVCTL_XXV	Use X.XV for CIS read

The parameter **VccLevel** specifies an index into the *PWRENTRY* array returned by the *InquireAdapter* function (see chapter “InquireAdapter”). The used index will specify the voltage which is used for Vcc.

The parameter **VppLevels** is an array of indices into the *PWRENTRY* array returned by the *InquireAdapter* function (see chapter “InquireAdapter”). The used indices will specify the voltages which are used for Vpp1 and Vpp2.

The parameter **State** returns the latched value of the state changes experienced on the specified socket. The values must be explicitly cleared by using the *SetSocket* function (see chapter "SetSocket"). The parameter is a combination of the following defined values:

PCC_SBM_WP	PC Card is write protected.
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **CtlInd** returns the current setting of the socket controls and indicators. The following values are defined for this parameter:

PCC_SBM_WP	PC Card write protect state
PCC_SBM_LOCKED	External generated indicator for the state of an externally mechanical or electrical for the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	Control for motor to eject a PC Card
PCC_SBM_INSERT	Control for motor to insert a PC Card
PCC_SBM_LOCK	Control for Card Lock
PCC_SBM_BATT	Indication for BVD1 and BVD2
PCC_SBM_BUSY	Indicator for showing the card in-use
PCC_SBM_XIP	Indicator for eXecution-In-Place application in progress

The parameter **IREQRouting** returns the actual IREQ signal. The can be a combination of the following values:

PCC_IRQ_HIGH	The PC Card interrupt is active high.
PCC_IRQ_ENABLE	The PC Card interrupt is enabled.

The parameter **IFType** returns the current interface setting. The following values are defined.

PCC_IF_MEMORY	Socket is set to memory-only interface.
PCC_IF_IO	Socket is set to I/O and memory interface.
PCC_IF_CARDBUS	Socket is set to CardBus PC Card.
PCC_IF_CUSTOM	The socket will use the custom interface described with the parameter **IFIndex**..
PCC_DREQ	A binary value will describe which DMA channel is currently used for DREQ#.

The parameter **IFIndex** returns the index into the custom interface array.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
FLAGS8             State,
                   SCIntMask,
                   CtlInd,
                   IFType;
IRQ                IREQRouting;
PWRINDEX           Vcontrol, VccLevel, VppLevels[2];
WORD               IFIndex;

…

/*-------------------------------------------------------
  Get actual configuration of socket 0 on adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_GET_SOCKET,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)&SCIntMask,
                        (unsigned long)&Vcontrol,
                        (unsigned long)&VccLevel,
                        (unsigned long)VppLevels,
                        (unsigned long)&State,
                        (unsigned long)&CtlInd,
                        (unsigned long)&IREQRouting,
                        (unsigned long)&IFType,
                        (unsigned long)&IFIndex);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_SOCKET	invalid socket

4.1.12 SetSocket

NAME

SetSocket sets a new configuration for the specified socket.

FUNCTION CODE

PCC_SET_SOCKET (0x8E)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket
FLAGS8	SCIntMask
PWRINDEX	Vcontrol
PWRINDEX	VccLevel
PWRINDEX	*VppLevels
FLAGS8	State
FLAGS8	CtlInd
IRQ	IREQRouting
FLAGS8	IFType
WORD	IFIndex

DESCRIPTION

This function reads the actual configuration of the specified socket. The parameters are compatible to the parameters which have to be set in the *SetSocket function*.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Socket** specifies the socket on the specified PC Card adapter.

The parameter **SCIntMask** sets the events which shall generate a status change interrupt when they occur on the socket. The value can be a combination of the following values:

PCC_SBM_WP	PC Card write protect
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **Vcontrol** specifies how to handle the voltage control. This can be a combination of the following defined values:

PCC_VXTL_CISREAD	Vcc and Vpp are controlled by the Vcc and Vpp fields.
PCC_VCTL_OVERRIDE	If set the Vcc level does not match to the value indicated by the voltage sense

The parameter **VccLevel** specifies an index into the *PWRENTRY* array returned by the *InquireAdapter* function (see chapter4.1.2 “InquireAdapter”). The used index will specify the voltage which will be set used for Vcc.

The parameter **VppLevels** is an array of indices into the *PWRENTRY* array returned by the *InquireAdapter* function (see chapter “InquireAdapter”). The used indices will specify the voltages which will be set for Vpp1 and Vpp2.

The parameter **State** clears the specified flags in the latched value of the state changes experienced on the specified socket. The parameter is a combination of the following defined values:

PCC_SBM_WP	PC Card is write protected.
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **CtlInd** sets up the setting of the socket controls and indicators. The following values are defined for this parameter:

PCC_SBM_WP	PC Card write protect state
PCC_SBM_LOCKED	External generated indicator for the state of an externally mechanical or electrical the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	Control for motor to eject a PC Card
PCC_SBM_INSERT	Control for motor to insert a PC Card
PCC_SBM_LOCK	Control for Card Lock
PCC_SBM_BATT	Indication for BVD1 and BVD2
PCC_SBM_BUSY	Indicator for showing the card in-use
PCC_SBM_XIP	Indicator for eXecution-In-Place application in progress

The parameter **IREQRouting** sets the IREQ# routing. The value can be a combination of the following values:

PCC_IRQ_HIGH	The PC Card interrupt is active high.
PCC_IRQ_ENABLE	The PC Card interrupt is enabled.

The parameter **IFType** specifies the interface setting. The following values are defined:

PCC_IF_MEMORY	Socket is set to memory-only interface.
PCC_IF_IO	Socket is set to I/O and memory interface.
PCC_IF_CARDBUS	Socket is set to CardBus PC Card.
PCC_IF_CUSTOM	The socket will use the custom interface described with the parameter **IFIndex**.
PCC_DREQ	A binary value will describe which DMA channel is currently used for DREQ#.

The parameter **IFIndex** specifies an index into the custom interface array. This value is only valid if *PCC_IF_CUSTOM* is specified.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
PWRINDEX           VppLevels[2];

…
```

```
…
/*-------------------------------------------------------
  Set a new configuration for socket 0 on adapter 4
    - enable status change interrupts for Card Detect
    - use PowerIndex 0 for Vcc and PowerIndex 1 for Vpp1/2
    - memory interface
  -------------------------------------------------------*/
VppLevels[0] = 0;
VppLevels[1] = 0;
result = pcc_entry (    PCC_SET_SOCKET,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)PCC_SBM_CD,
                        (unsigned long)0,
                        (unsigned long)2,
                        (unsigned long)VppLevels,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)PCC_IF_MEMORY,
                        (unsigned long)0);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_IRQ	specified IRQRouting is not defined
PCC_BAD_SOCKET	invalid socket
PCC_BAD_TYPE	invalid IFType
PCC_BAD_VCC	invalid Vcc level
PCC_BAD_VPP	invalid Vpp level defined
PCC_BAD_ATTRIBUTE	invalid *VCTL_x* combination

4.1.13 GetStatus

NAME

GetStatus reads status information of the specified socket

FUNCTION CODE

PCC_GET_STATUS (0x8F)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket
FLAGS8	*CardState
FLAGS8	*SocketState
FLAGS8	*CtlInd
IRQ	*IREQRouting
FLAGS8	*IFType

DESCRIPTION

This function reads the status of the card, socket, controls and indicators of the specified socket.

This service should not be invoked during hardware interrupt processing. It is intended to be used on application layer.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Socket** specifies the socket on the specified PC Card adapter.

The parameter **CardState** returns the actual state of the socket and the PC Card. The returned values are dependent from the capabilities of the PC Card controller. The value can be a combination of the following values:

PCC_SBM_WP	PC Card write protect
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **SocketState** returns the latched value of the state changes experienced on the specified socket. The values must be explicitly cleared by using the *SetSocket* function (see chapter “SetSocket”). The parameter is a combination of the following defined values:

PCC_SBM_WP	PC Card is write protected.
PCC_SBM_LOCKED	External generated indicating the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	External generated indicating a request to eject the card
PCC_SBM_INSERT	External generated indicating a request to insert the card
PCC_SBM_BVD1	Battery voltage detect1, indicator for battery is unserviceable
PCC_SBM_BVD2	Battery voltage detect2, indicator for battery is weak
PCC_SBM_RDYBSY	Indicator for Ready/Busy
PCC_SBM_CD	Card detect 1 and 2

The parameter **CtlInd** returns the current setting of the socket controls and indicators. The following values are defined for this parameter:

PCC_SBM_WP	PC Card write protect state
PCC_SBM_LOCKED	External generated indicator for the state of an externally mechanical or electrical the state of an electrical or mechanical lock mechanism
PCC_SBM_EJECT	Control for motor to eject a PC Card
PCC_SBM_INSERT	Control for motor to insert a PC Card
PCC_SBM_LOCK	Control for Card Lock
PCC_SBM_BATT	Indication for BVD1 and BVD2
PCC_SBM_BUSY	Indicator for showing the card in-use
PCC_SBM_XIP	Indicator for eXecution-In-Place application in progress

The parameter **IREQRouting** returns the actual IREQ signal. The value can be a combination of the following values:

PCC_IRQ_HIGH	The PC Card interrupt is active high.
PCC_IRQ_ENABLE	The PC Card interrupt is enabled.

The parameter **IFType** returns the current interface setting. The following values are defined.

PCC_IF_MEMORY	Socket is set to memory-only interface.
PCC_IF_IO	Socket is set to I/O and memory interface.
PCC_IF_CARDBUS	Socket is set to CardBus PC Card.
PCC_IF_CUSTOM	The socket will use the custom interface.
PCC_DREQ	A binary value will describe which DMA channel is currently used for DREQ#.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
FLAGS8             CardState,
                   SocketState,
                   CtlInd,
                   IFType;
IRQ                IREQRouting;

…

/*-------------------------------------------------------
  Get actual status of socket 0 on adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_GET_STATUS,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)&CardState,
                        (unsigned long)&SocketState,
                        (unsigned long)&CtlInd,
                        (unsigned long)&IREQRouting,
                        (unsigned long)&IFType);
…
```

```
…
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_SOCKET	invalid socket

4.1.14 ResetSocket

NAME

ResetSocket resets PC Card and socket hardware.

FUNCTION CODE

PCC_RESET_SOCKET (0x90)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket

DESCRIPTION

This function resets the PC Card in the specified socket and it resets the specified socket hardware to its power-on default state.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Socket** specifies the socket on the specified PC Card adapter.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long result;

…

/*-------------------------------------------------------
  Reset PC Card and hardware on socket 0 on adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_RESET_SOCKET,
                        (unsigned long)4,
                        (unsigned long)0);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_SOCKET	invalid socket
PCC_NO_CARD	no card in specified socket

4.1.15 GetVendorInfo

NAME

GetVendorInfo gets information about the vendor implementing of the socket functions.

FUNCTION CODE

PCC_GET_VENDOR_INFO (0x9D)

PARAMETERS

ADAPTER	Adapter
BYTE	Type
PTR	*pBuffer
BCD	*Release

DESCRIPTION

This function returns information about the socket functions.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Type** specifies the type of the client-supplied buffer. The only currently defined **Type** is zero (0).

The parameter **pBuffer** is a pointer to the *VISTRUCT* structure where the adapter capabilities will be stored.

Data structure *VISTRUCT*:

WORD	wBufferLength	This is an input entry and must be set to the maximal length of the *szImplementor* array.
WORD	wDataLength	The function will return in this entry to the real length of the *szImplementor* array.
char	szImplementor	This field is filled with an ASCII string, giving information about the vendor.

The parameter **Release** returns the release number of actual used socket functions. This value is BCD coded, 0x0100 means 1.00.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
char               pBuffer[80];
VISTRUCT           *pVIStruct;
BCD                Release;

…
```

```
…
/*-------------------------------------------------------
  Get vendor information for adapter 4
  -------------------------------------------------------*/
pVIStruct = (VISTRUCT*)&(pBuffer[0]);
pVIStruct->wBufferLength = 70;
result = pcc_entry (    PCC_GET_VENDOR_INFO,
                        (unsigned long)4,
                        (unsigned long)0,
                        (unsigned long)pBuffer,
                        (unsigned long)&Release);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter
PCC_BAD_SERVICE	invalid service type

4.1.16 AcknowlegdeInterrupt

NAME

AcknowledgeInterrupt gets information of interrupt source and remove interrupt.

FUNCTION CODE

PCC_ACK_INTERRUPT (0x9E)

PARAMETERS

ADAPTER	Adapter
SKTBITS	*Sockets

DESCRIPTION

This function returns information on which sockets an interrupt is pending and clears the interrupt source. This function should always be called in the status change interrupt function.

PARAMETERS

The parameter **Adapter** specifies the PC Card adapter.

The parameter **Sockets** returns a bit-field specifying the sockets where an interrupt has occurred. Bit 0 is set for Socket 0, bit 1 is set for Socket 1 and so on.

EXAMPLE

```
#include      “pcc_mng.h”

unsigned long      result;
SKTBITS            Sockets;

…

/*-------------------------------------------------------
  Acknowledge interrupts on adapter 4
  -------------------------------------------------------*/
result = pcc_entry (    PCC_ACK_INTERRUPT,
                        (unsigned long)4,
                        (unsigned long)&Sockets);
if (result)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

PCC_NO_ERROR	no error
PCC_BAD_ADAPTER	invalid adapter

4.2 Unsupported Functions

These functions will all return the error code *TP871_SS_UNIMPL_FUNCTION*.

4.2.1 GetSSInfo

NAME

GetSSInfo gets information about the socket services.

FUNCTION CODE

PCC_GET_SS_INFO (0x83)

PARAMETERS

ADAPTER	Adapter
BCD	*Complience
COUNT	*NumAdapters
ADAPETER	*FirstAdapter

4.2.2 InquireEDC

NAME

InquireEDC gets capabilities of the EDC generator.

FUNCTION CODE

PCC_INQ_EDC (0x95)

PARAMETERS

ADAPTER	Adapter
EDC	Edc
SKTBITS	*Sockets
FLAGS8	*Caps
FLAGS8	*Types

4.2.3 GetEDC

NAME

GetEDC gets actual configuration of the specified EDC.

FUNCTION CODE

PCC_GET_EDC (0x96)

PARAMETERS

ADAPTER	Adapter
EDC	Edc
SOCKET	*Socket
FLAGS8	*State
FLAGS8	*Type

4.2.4 SetEDC

NAME

SetEDC sets new configuration for the specified EDC.

FUNCTION CODE

PCC_SET_EDC (0x97)

PARAMETERS

ADAPTER	Adapter
EDC	Edc
SOCKET	Socket
FLAGS8	State
FLAGS8	Type

4.2.5 StartEDC

NAME

StartEDC starts the previously configured EDC.

FUNCTION CODE

PCC_START_EDC (0x98)

PARAMETERS

ADAPTER	Adapter
EDC	Edc

4.2.6 PauseEDC

NAME

PauseEDC pauses the specified EDC.

FUNCTION CODE

PCC_PAUSE_EDC (0x99)

PARAMETERS

ADAPTER	Adapter
EDC	Edc

4.2.7 ResumeEDC

NAME

ResumeEDC resumes the specified, paused and configured EDC.

FUNCTION CODE

PCC_RESUME_EDC (0x9A)

PARAMETERS

ADAPTER	Adapter
EDC	Edc

4.2.8 StopEDC

NAME

StopEDC	stops the specified EDC.

FUNCTION CODE

PCC_STOP_EDC (0x9B)

PARAMETERS

ADAPTER	Adapter
EDC	Edc

4.2.9 ReadEDC

NAME

ReadEDC	reads the EDC value from the specified EDC.

FUNCTION CODE

PCC_READ_EDC (0x9C)

PARAMETERS

ADAPTER	Adapter
EDC	Edc
DWORD	*Value

4.2.10 GetSetPriorHandler

NAME

GetSetPriorHandler replaces or obtains the entry point of a prior handler for the adapter.

FUNCTION CODE

PCC_PRIOR_HANDLER (0x9F)

PARAMETERS

ADAPTER	Adapter
FLAGS8	Mode
PTR	pHandler

4.2.11 GetSetSSAddr

NAME

GetSetSSAddr returns code and data area descriptions.

FUNCTION CODE

PCC_SS_ADR (0xA0)

PARAMETERS

ADAPTER	Adapter
BYTE	Mode
BYTE	Subfunc
COUNT	*NumAddData
PTR	pBuffer

4.2.12 GetAccessOffsets

NAME

GetAccessOffsets

FUNCTION CODE

PCC_ACCESS_OFFSETS (0xA1)

PARAMETERS

ADAPTER	Adapter
BYTE	Mode
COUNT	NumDesired
PTR	pBuffer
COUNT	*NumAvail

4.2.13 AccessConfigurationSpace

NAME

AccessConfigurationSpace provides an interface to read and write in CardBus configuration space.

FUNCTION CODE

PCC_READ_EDC (0xA2)

PARAMETERS

ADAPTER	Adapter
SOCKET	Socket
BYTE	Function
FLAGS8	Action
OFFSET	Location
FLAGS32	*Data

4.2.14 InquireBridgeWindow

NAME

InquireBridgeWindow reads capabilities about the specified bridge window.

FUNCTION CODE

PCC_INQ_BWINDOW (0xA3)

PARAMETERS

ADAPTER	Adapter
WINDOW	window
PTR	pBuffer
FLAGS8	*WndCaps
SKTBITS	*Sockets

4.2.15 GetBridgeWindow

NAME

GetBridgeWindow gets actual configuration of the specified bridge window.

FUNCTION CODE

PCC_GET_BWINDOW (0xA4)

PARAMETERS

ADAPTER	Adapter
WINDOW	window
SOCKET	*Socket
SIZE	*Size
FLAGS8	*State
BASE	*Base

4.2.16 SetBridgeWindow

NAME

SetBridgeWindow sets new configuration of the specified bridge window.

FUNCTION CODE

PCC_SET_BWINDOW (0xA5)

PARAMETERS

ADAPTER	Adapter
WINDOW	window
SOCKET	Socket
SIZE	Size
FLAGS8	State
BASE	Base

4.2.17 VendorSpecific

NAME

VendorSpecific Function is vendor specific.

FUNCTION CODE

PCC_READ_EDC (0xAE)

PARAMETERS

ADAPTER Adapter

5 ATA Disk Device Driver

This VxWorks ATA Disk device driver allows the operation of PC Card ATA disks with TPMC871 PC Card PMC conforming to the VxWorks block I/O system specification. The ATA Disk device driver is an addition to the TPMC871-SW-42 Device Interface Software and will not work without it. It will call the calling interface which is a unified interface for PC Card interfaces from TEWS TECHNOLOGIES, which allows running this driver on different PC Card interfaces.

Before using the ATA Disk driver the calling interface and the PC Card devices must be initialized.

This ATA Disk device driver is a standard block device and the implemented file systems of VxWorks shall be possible to be handled by the driver. More information on block devices and the file systems can be found in the VxWorks Programmer's Guide.

5.1 I/O System Functions

This chapter describes the driver level interface to the I/O system. The purpose of these functions is to install the driver in the I/O system, add and initialize devices.

5.1.1 pccAtaDrv()

NAME

pccAtaDrv() initializes ATA Disk PC Card driver.

SYNOPSIS

```
STATUS pccAtaDrv
(
    void
)
```

DESCRIPTION

This function initializes the local values of ATA Disk PC Card driver. This function must be called before any other function of this ATA Disk driver is called.

EXAMPLE

```
#include      "ata_drv.h"

STATUS        result;

...
```

```
…
result = pccAtaDrv();

if (result == ERROR)
{
     /* Handle the occurred error */
}
else
{
     /* Execution successful */
}
```

RETURNS

OK or ERROR if an error occurred. On ERROR the driver will set the global error value to an appropriate error code. For a full description of the error codes see chapter “Error Codes”.

5.1.2 pccAtaDevCreate() – (up to VxWorks 6.1)

NAME

pccAtaDevCreate() creates a device descriptor for ATA Disk in the specified PC Card adapter. This function must be used for VxWorks 6.1 and older versions.

SYNOPSIS

```
unsigned long     pccAtaDevCreate
(
      ADAPTER         adNo,
      SOCKET          sockNo,
      int             intLevel,
      int             intVector
)
```

DESCRIPTION

This function creates an ATA Disk device on the specified PC Card socket that will be serviced by the ATA Disk driver. This function must be called before accesses are made to the ATA Disk. The function will return a device descriptor, which must be specified when creating the device for the file system (for example *dosFsDevInit()*).

PARAMETERS

The parameters **adNo** and **sockNo** specify the PC Card socket where the ATA Disk device shall be created on.

The parameters **intLevel** and **intVector** specify the interrupt level and the interrupt vector for the interrupts created for the specified socket. The values of these levels are hardware and BSP dependent. The values which are specified in the BSP documentation for #INTA of the PMC slot the module is mounted have to be specified.

EXAMPLE

```
#include      “ata_drv.h”

unsigned long      dev_handle;

…

/*-------------------------------------------------------
  Create an ATA Disk device on adapter 4, socket 0. The
  interrupt level and interrupt vector are 0x19.
  -------------------------------------------------------*/
dev_handle = pccAtaDevCreate (4, 0, 0x19, 0x19);
if (dev_handle)
{
     /* Execution successful */
}
else
{
     /* Handle the occurred error */
}
```

RETURNS

This function returns a device descriptor on success and on ERROR it will return 0. On ERROR the driver will set the global error value to an appropriate error code. For a full description of the error codes see chapter “Error Codes”.

5.1.3 pccAtaXbdDevCreate() – (VxWorks 6.2 and newer)

NAME

pccAtaXbdDevCreate() creates a device descriptor for ATA Disk in the specified PC Card adapter. This function replaces *pccAtaDevCreate()* for VxWorks 6.2 and newer systems.

SYNOPSIS

```
unsigned long     pccAtaDevCreate
(
        ADAPTER         adNo,
        SOCKET          sockNo,
        int             intLevel,
        int             intVector,
        char            name
)
```

DESCRIPTION

This function creates an ATA Disk device on the specified PC Card socket that will be serviced by the ATA Disk driver. This function must be called before accesses are made to the ATA Disk. The function will return a device descriptor.

PARAMETERS

The parameters **adNo** and **sockNo** specify the PC Card socket where the ATA Disk device shall be created on.

The parameters **intLevel** and **intVector** specify the interrupt level and the interrupt vector for the interrupts created for the specified socket. The values of these levels are hardware and BSP dependent. The values which are specified in the BSP documentation for #INTA of the PMC slot the module is mounted have to be specified.

The parameter **name** specifies the base of the drive name. The name of the disk (partition) will be build by the specified name, a colon and a number specifying the partition number (normally 0) on the disk. If a disk is specified with the name “/PCC0”, the device “/PCC0:0” will be created and accessible with disk operations.

EXAMPLE

```
#include      “ata_drv.h”

unsigned long      dev_handle;

…

/*-------------------------------------------------------
  Create an ATA Disk device on adapter 4, socket 0. The
  interrupt level and interrupt vector are 0x19. The
  disk names shall be “/PCC0:0”
  -------------------------------------------------------*/
dev_handle = pccAtaXbdDevCreate (4, 0, 0x19, 0x19, “/PCC0”);
if (dev_handle)
{
     /* Execution successful */
}
else
{
     /* Handle the occurred error */
}
```

RETURNS

This function returns a device descriptor on success and on ERROR it will return 0. On ERROR the driver will set the global error value to an appropriate error code. For a full description of the error codes see chapter “Error Codes”.

5.2 I/O Interface Functions

This chapter describes the interface to the basic I/O system.

5.2.1 open()

NAME

open() opens a device or file.

SYNOPSIS

```
int open
(
        const char  *name,      /* name of the file to open */
        int         flags,      /* select access mode (O_RDONLY, O_WRONLY ...)  */
        int         mode        /* mode of file to create     */
)
```

DESCRIPTION

Before I/O can be performed to a file, a file descriptor must be opened by invoking the basic I/O function *open().*

EXAMPLE

```
/*------------------------------------------------------------
  Open the file named "example.txt" at the drive
  named "PC1:" for a read access
  ------------------------------------------------------------*/
fd = open ("PCC1:\\example.txt", O_RDONLY, 0);
if (fd == ERROR)
{
     /* Handle error */
}
```

RETURNS

A file descriptor number or ERROR (if the file does not exist or no file descriptors are available)

SEE ALSO

ioLib, basic I/O routine - *open()*

5.2.2 close()

NAME

close() – close a device or file

SYNOPSIS

```
STATUS close
(
     int          fd      /* filedescriptor */
)
```

DESCRIPTION

This function closes opened file previously opened.

EXAMPLE

```
int       fd;
STATUS    retval;

/*----------------
  close the device
  ----------------*/
retval = close(fd);
if (retval == ERROR)
{
     /* Handle error */
}
```

RETURNS

OK or ERROR. If the function fails, an error code will be stored in *errno*.

SEE ALSO

ioLib, basic I/O routine - close()

5.2.3 read()

NAME

read() reads bytes from the specified file.

SYNOPSIS

```
int read
(
    int     fd,         /* descriptor of opened file          */
    char    *buffer,    /* pointer to a buffer to receive bytes */
    size_t  maxbytes    /* max number of bytes to read        */
)
```

DESCRIPTION

This routine reads a number of bytes from the specified file and places them in **buffer**. The parameter **maxbytes** specifies the maximum number of bytes to read.

EXAMPLE

```
int  fd;
int  nbytes;
char buffer[80];

…

/*----------------------------------------------------------
  Read up to 80 bytes from the file connected with
  the file descriptor fd
  ----------------------------------------------------------*/
nbytes = read (fd, buffer, 80);
```

RETURNS

ERROR or number of bytes read

SEE ALSO

ioLib, basic I/O routine - *read()*

5.2.4 write()

NAME

write() writes bytes to the specified file on ATA Disk device.

SYNOPSIS

```
int write
(
        int             fd,             /* file descriptor on which to write        */
        char            *buffer,        /* buffer containing bytes to be written    */
        size_t          nbytes          /* number of bytes to be written            */
)
```

PARAMETER

This routine writes **nbytes** bytes from **buffer** to the specified file connected with the file descriptor **fd**.

EXAMPLE

```
int  fd;
int  nbytes;

…

/*-------------------------------------------------------
  Write 12 bytes to the file connected with the
  file descriptor fd
  -------------------------------------------------------*/
nbytes = write (fd, "Hello world!", 12);
```

RETURNS

ERROR or number of bytes written

SEE ALSO

ioLib, basic I/O routine - *write()*

5.2.5 ioctl()

NAME

ioctl() performs an I/O control function.

SYNOPSIS

```
int ioctl
(
        int         fd,         /* file descriptor                          */
        int         function,   /* function code                            */
        int         arg         /* optional function dependent argument     */
)
```

DESCRIPTION

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

The ATA Disk driver supports no special I/O operation.

RETURNS

OK or ERROR (if the device descriptor does not exist or the function code is unknown)

INCLUDE FILES

ioLib.h

SEE ALSO

ioLib, basic I/O routine - *ioctl(),*VxWorks Programmer's Guide: I/O System and Local File System

6 PCI Configuration Hints

This chapter describes how to decide if a system dependent PCI configuration is necessary and how the configuration has to be done.

The chapter also gives an advice how to modify the example configuration depending on the PCI configuration of the system.

6.1 Check TPMC871 PCI configuration

This is the first step that should be executed, to decide if extra configuration is necessary or if the system has already been setup by CPU boot code. The driver disk contains a file called *pciMapShow.c* which contains a function named *pciMapShow()* that will display the current PCI-map. For execution the file must be included to your system. After startup the function can be executed by calling:

```
pciMapShow
```

The function will now display some tables containing the PCI-bus information. 1st A map of all PCI-devices will be shown. This gives an overview where to find TPMC871 devices and bridges we may have to setup. The highlighted entries show the PCI devices which are important for us. One shows the TPMC871 mounted to PCI-bus 1 and device 2 with one supplied function on the TPMC871. (A TPMC872 or TCP872 will have 2 functions). The second highlighted device shows the PCI-PCI bridge from bus 0 to bus 1, it is found on PCI-bus 0, device 20 and function 0.

(The example shows a part of the displayed device table on a TVME8240 with PMC-Span and 1 TPMC871 running a VxWorks 5.4) There have not been run any special PCI-configurations yet)

```
+-------------------------------------------------------+
|                     PCI DEVICE LIST                   |
+-------------------------------------------------------+
| BusNo:   0 -- DevNo:  0 -- FuncNo: 0                  |
|        VendorID: 1057h                                |
|        DeviceID: 0003h                                |
|        Class Code:  060000h                           |
|        Command:     06h                               |
|        Header Type: 00h                               |
|        SubvendorID: 0000h -- SubsystemID: 0000h       |
+-------------------------------------------------------+
| BusNo:   0 -- DevNo: 13 -- FuncNo: 0                  |
|        VendorID: 10E3h                                |
|        DeviceID: 0000h                                |
|        Class Code:  068000h                           |
|        Command:     07h                               |
|        Header Type: 00h                               |
|        SubvendorID: 0000h -- SubsystemID: 0000h       |
+-------------------------------------------------------+
…
```

```
…
+-----------------------------------------------------------+
| BusNo:   0 -- DevNo: 20 -- FuncNo: 0                      |
|        VendorID: 1011h                                    |
|        DeviceID: 0022h                                    |
|        Class Code:  060400h                               |
|        Command:     07h                                   |
|        Header Type: 01h                                   |
+-----------------------------------------------------------+
| BusNo:   1 -- DevNo:  2 -- FuncNo: 0                      |
|        VendorID: 104Ch                                    |
|        DeviceID: AC56h                                    |
|        Class Code:  060700h                               |
|        Command:     03h                                   |
|        Header Type: 02h                                   |
|        SubvendorID: 1498h -- SubsystemID: 0367h           |
+-----------------------------------------------------------+
```

The second table shows PCI-I/O spaces. The table gives an overview which areas of the PCI-I/O space are already in use and how they are split to subordinate busses.

(The example shows a part of the displayed I/O address map on a TVME8240 with PMC-Span and 2 TPMC871)

No I/O space mapped from busNo 0 to busNo 1 by the PCI-PCI-bridge (identified by '*(Bridge I/O)*', nor any I/O space setup for the TPMC871 (identified by '*(CardBus I/O x)*') is displayed. Only a Legacy Register is visible which will not used by the driver. That means there is no space to access the TPMC871. Refer to *6.2 Setup TPMC871 PCI configuration manually* to find out how to make the configuration.

```
+-----------------------------------------------------------+
|                      I/O ADDRESS MAP                      |
| Bus |  Address  ||                                        |
+-----+-----------++----------------------------------------+
|   1 | FE000000h || DevNo:  2 -- FuncNo:  0                |
|     |     :     ||                  (Legacy Reg Address)  |
|     |     :     ||                          ENABLED I/O   |
|     |     :     || DeviceID:  Ven:104Ch - Dev:AC56h       |
|     | FE000001h || SubsysID:  Ven:1498h - Sys:0367h       |
|:::::|:::::::::::||::::::::::::::::::::::::::::::::::::::::|
|   0 | FE800000h || DevNo: 13 -- FuncNo:  0                |
|     |     :     ||                       (Base Address 0) |
|     |     :     ||                          ENABLED I/O   |
|     |     :     || DeviceID:  Ven:10E3h - Dev:0000h       |
|     | FE800FFFh || SubsysID:  Ven:0000h - Sys:0000h       |
|:::::|:::::::::::||::::::::::::::::::::::::::::::::::::::::|
|   0 | FE801000h || DevNo: 14 -- FuncNo:  0                |
…
```

The last table shows PCI Memory spaces. The table gives an overview which areas of the PCI-memory space are already in use and how they are split to subordinate busses.

(The example shows a part of the displayed memory address map on a TVME8240 with PMC-Span and 2 TPMC871)

No memory space mapped from busNo 0 to busNo 1 by the PCI-PCI-bridge (identified by '(Bridge Memory)'), nor any memory space setup for the TPMC871 (identified by '(CardBus Memory)') is displayed. Only the Register Address Space is visible, but not in an accessible space on busNo1. That means there is no space to access the TPMC871. Refer to *6.2 Setup TPMC871 PCI configuration manually* to find out how to make the configuration.

```
+-----------------------------------------------------------+
|                     MEMORY ADDRESS MAP                    |
| Bus |  Address  ||                                        |
+-----+-----------++----------------------------------------+
|   1 | 00000000h || DevNo:  2 -- FuncNo:  0                |
|     |     :     ||                        (Base Address 0) |
|     |     :     ||                            ENABLED MEM  |
|     |     :     || DeviceID:  Ven:104Ch - Dev:AC56h        |
|     | 00000FFFh || SubsysID:  Ven:1498h - Sys:0367h       |
|:::::|:::::::::::||::::::::::::::::::::::::::::::::::::::::|
|   0 | 80000000h || DevNo: 13 -- FuncNo:  0                |
|     |     :     ||                        (Base Address 1) |
|     |     :     ||                            ENABLED MEM  |
|     |     :     || DeviceID:  Ven:10E3h - Dev:0000h        |
|     | 80000FFFh || SubsysID:  Ven:0000h - Sys:0000h       |
|:::::|:::::::::::||::::::::::::::::::::::::::::::::::::::::|
|   0 | 80001000h || DevNo: 14 -- FuncNo:  0                |
```

…

6.2 Setup TPMC871 PCI configuration manually

If address spaces necessary for the used PC Cards are not available on the TPMC871, we have to modify the PCI configuration manually. Address spaces may not accessible because the PCI configuration of the TPMC871 has not setup the necessary spaces, or the PCI-PCI bridge or bridges from PCI bus 0 to the subordinate bus where the TPMC871 is mounted is not setup.

If the bridges do not map enough I/O and/or memory address space to the bus where the TPMC871 is connected, we have to setup the PCI-PCI bridge first. The second step is to setup the TPMC871 PCI configuration.

Configuration functions have to be included like the *tp871PciInit()* function which makes standard TPMC871 PCI initialization and MMU mapping. (Refer to *2.5 Include PCI setup for TPMC871*) If a setup function is not needed, it may not be included or the corresponding configuration table can be used without an entry.

The right order of function calls is:

```
tp871PciBridgeSetup();  /* Setup Bridge Addresses */
tp871PciSetup();        /* Setup Addresses */
tp871PciInit();         /* Init Controller */
```

6.2.1 Setup PCI-PCI bridges manually

The configuration to setup the PCI-PCI bridge has to be made in the table *tp871BridgeCfgTable[]* which is find in *tp871pciConfig.c*. The table already contains one entry which marks the end of the table and must always be the last one. New entries must be insert at the top of the table, the structure (*TP871PCIBRIDGECONFIGSTRUCT*) of the entries is defined in *tp871pciConfig.h*.

```
typedef struct
{
        int             busNo;
        int             devNo;
        int             funcNo;
        unsigned int    memBaseAddr;
        unsigned int    memSpaceSize;
        unsigned int    ioBaseAddr;
        unsigned int    ioSpaceSize;
        int             secBusNo;
        int             subOrdBusNo;
        int             comandReg;
} TP871PCIBRIDGECONFIGSTRUCT;
```

busNo

Specifies the bus number of the bridges primary bus. This value is used to select the bridge. (A value of -1 specifies the last entry in the table)

devNo

Specifies the bridges device number on the primary bus. This value is used to select the bridge. (A value of -1 specifies the last entry in the table)

funcNo

Specifies the bridges function number on the primary bus. This value is used to select the bridge, usually it is 0. (A value of -1 specifies the last entry in the table)

memBaseAddr

This entry specifies the base address of the memory addresses that shall be mapped to the secondary bus. *memBaseAddr* and *memSpaceSize* must select a memory address area that is not mapped by any other device or bridge on the primary bus. The base address must be aligned to a multiple of the space size, but at least to a 1MB (0x100000) boundary. A value of 0 specifies that the setup of the memory space shall be skipped and the current mapping shall be used.

memSpaceSize

This entry specifies the size of the memory address area that shall be mapped to the secondary bus. *memBaseAddr* and *memSpaceSize* must select a memory address area that is not mapped by any other device or bridge on the primary bus. A value of 0 specifies that the memory access to the secondary bus shall be disabled. The space size must be at least 1MB (0x100000). This value is not used if *memBaseAddr* is set to 0.

ioBaseAddr

This entry specifies the base address of I/O addresses that shall be mapped to the secondary bus. *ioBaseAddr* and *ioSpaceSize* must select an I/O address area that is not mapped by any other device or bridge on the primary bus. The base address must be aligned to a multiple of the space size, but at least to a 4KB (0x1000) boundary. A value of 0 specifies that the setup of the I/O space shall be skipped and the current mapping shall be used.

ioSpaceSize

This entry specifies the size of the I/O address area that shall be mapped to the secondary bus. *ioBaseAddr* and *ioSpaceSize* must select an I/O address area that is not mapped by any other device or bridge on the primary bus. The space size must be at least 4KB (0x1000). A value of 0 specifies that the I/O access to the secondary bus shall be disabled. This value is not used if *ioBaseAddr* is set to 0.

secBusNo

Specifies the secondary bus number. The secondary bus number must be the next unused one. If a value of -1 is specified the setup of the secondary bus number is skipped and the current value will used.

subOrdBusNo

Specifies the subordinate bus number. The subordinate bus number specifies the highest bus number that will available on the secondary bus. If there is no bridge on the secondary bus, the value must be the secondary bus number. If a value of -1 is specified the setup of the secondary bus number is skipped and the current value will used.

comandReg

Specifies the value that shall be written to bridges command register. The table below shows the most relevant bits (for all other bits refer to the PCI bridge specification):

Bit	Description
0	Enable I/O access If I/O access is disabled by *ioSpaceSize* = 0 the bit will be reset independent from this flag.
1	Enable memory access If memory access is disabled by *memSpaceSize* = 0 the bit will be reset independent from this flag.
2	Enable Bus Master

EXAMPLE

(This example shows how to make the bridge mapping for a TVME8240 with PMC-Span and a TPMC871, see also *6.1 Check TPMC871 PCI configuration*)

The memory map shows that a memory address space starting at 0x80100000 with a size of 0x00080000 (512KB) is available and unused on PCI-bus 0 and that the I/O address space starting at 0xFE001000 with a size of 0x1000 (4KB) is available and unused on PCI-bus 0. The secondary bus number 1 can be used and there will be no more subordinate busses. I/O, Memory and Bus Master access shall be enabled. This settings must be inserted with new table entry in tp871BridgeCfgTable[]. The bridge can be found on the primary bus 0 as device 20 and function 0.

```
/* This table sets up the PCI-PCI Bridge on a PMC-Span mounted to an TVME8240 */
TP871PCIBRIDGECONFIGSTRUCT tp871BridgeCfgTable[] =
{ /* B,  D, F, memBaseAdr, memSpcSize, ioBaseAddr, ioSpceSize, secB, subOrdB, cmdReg */
  {  0, 20, 0, 0x80100000, 0x00080000, 0xFE001000, 0x00000800,    1,       1,   0x07},

  /* busNo, devNo, funcNo == -1 ==> End of Table */
  { -1, -1, -1, 0x00000000, 0x00000000, 0x00000000,  0x00000000,  0,       0,   0x00}
};
```

After recompilation the PCI map should have an additional I/O and an additional memory entry showing the spaces mapped by the PCI-PCI bridge to PCI bus 1.

New I/O entry:

```
-----+-----------++---------------------------------------+
>   0 | FE001000h || DevNo: 20 -- FuncNo:  0               |
>     |     :     ||                          (Bridge I/O) |
>     |     :     ||                          ENABLED I/O  |
>     |     :     || DeviceID:  Ven:1011h - Dev:0022h      |
>     | FE001FFFh || Bridge   0 -->  1  ---  (SubOrd:   1) |
+-----+-----------++---------------------------------------+
```

New memory entry:

```
+-----+-----------++---------------------------------------+
>   0 | 80100000h || DevNo: 20 -- FuncNo:  0               |
>     |     :     ||                       (Bridge Memory) |
>     |     :     ||                          ENABLED MEM  |
>     |     :     || DeviceID:  Ven:1011h - Dev:0022h      |
>     | 8017FFFFh || Bridge   0 -->  1  ---  (SubOrd:   1) |
+-----+-----------++---------------------------------------+
```

6.2.2 Setup TPMC871 PCI configuration manually

The configuration to setup the TPMC871 PCI device has to be made in the table *tp871CfgTable[]* which is find in *tp871pciConfig.c*. The table already contains one entry which marks the end of the table and must always be the last one. New entries must be insert at the top of the table, the structure (*TP871PCICONFIGSTRUCT*) of the entries is defined in *tp871pciConfig.h*.

```
typedef struct
{
        int             busNo;
        int             devNo;
        int             funcNo;
        unsigned int    regAddr; */
        unsigned int    memBaseAddr;
        unsigned int    memSpaceSize;
        unsigned int    ioBaseAddr;
        unsigned int    ioSpaceSize;
        unsigned char   intLine;
} TP871PCICONFIGSTRUCT;
```

busNo

Specifies the bus number the TPMC871 is mounted on. This value is used to select the TPMC871. (A value of -1 specifies the last entry in the table)

devNo

Specifies the TPMC871 device number on PCI bus. This value is used to select the TPMC871. (A value of -1 specifies the last entry in the table)

funcNo

Specifies the TPMC871 device function on the PCI bus. This value is used to select the PC-Card adapter on the TPMC871. On TPMC871 there is only function number 0 present, but for TCP872 and TPMC872 there function 0 and function1 present. The function number selects the PC-Card slot. (A value of -1 specifies the last entry in the table)

regAddr

This entry specifies the base address of the TPMC871 register space in memory. This space has a fixed size of 0x1000 byte. The specified address and the following 0x1000 Byte must be available and unused by other devices on the PCI bus the TPMC871 is mounted on. A value of 0 specifies that the setup of the register space shall be skipped and the current mapping shall be used. The address must be aligned to 4KB (0x1000) boundary.

memBaseAddr

This entry specifies the base address of the memory addresses that shall be used for memory windows by the PC-Card. *memBaseAddr* and *memSpaceSize* must select a memory address area that is not mapped by any other device or bridge on the PCI bus. The base address must be aligned to a multiple of the space size, but at least to a 4KB (0x1000) boundary. A value of 0 specifies that the setup of the memory window space shall be skipped and the current mapping shall be used.

memSpaceSize

This entry specifies the size of the memory window that shall be usable for PC-Card memory windows. *memBaseAddr* and *memSpaceSize* must select a memory address area that is not mapped by any other device or bridge on the PCI bus. A value of 0 specifies that access to memory windows shall be disabled. The space size must be at least 4KB (0x1000). This value is not used if *memBaseAddr* is set to 0.

ioBaseAddr

This entry specifies the base address of the I/O addresses that shall be used for I/O windows by the PC-Card. *ioBaseAddr* and *ioSpaceSize* must select an I/O address area that is not mapped by any other device or bridge on the PCI bus. The base address must be aligned to a multiple of the space size, but at least to a 4Byte boundary. A value of 0 specifies that the setup of the I/O window space shall be skipped and the current mapping shall be used.

ioSpaceSize

This entry specifies the size of the I/O window that shall be usable for PC-Card I/O windows. *ioBaseAddr* and *ioSpaceSize* must select an I/O address area that is not mapped by any other device or bridge on the PCI bus. The space size must be at least 4Byte. A value of 0 specifies that the I/O access to the secondary bus shall be disabled. A value of 0 specifies that access to I/O windows shall be disabled. This value is not used if *ioBaseAddr* is set to 0.

intLine

This entry specified the interrupt line that is used by the TPMC871. This value is slot dependent and can not be changed. If there has been assigned an interrupt line without setting with this configuration it is recommended not keep the assigned value. A value of 0 will skip the configuration of the interrupt line.

To get the necessary value it may be helpful to mount another PMC device and read out the assigned interrupt line.

EXAMPLE

(This example shows how to make the bridge mapping for a TVME8240 with PMC-Span and a TPMC871, see also *6.1 Check TPMC871 PCI configuration*)

The bridge setup has mapped an memory address space starting at 0x80100000 with a size of 0x000800000 (512KB) and an I/O address space starting at 0xFE001000 with a size of 0x1000 (4KB) to PCI bus 0. The register space and PC-Card memory windows have to be placed into the memory address space. The address space shall be accessible at 0x80100000 and the memory windows shall be mapped into a window of 0x10000 bytes starting at address 0x80110000. The I/O window shall have 0x100 bytes and should start at 0xFE001400. The interrupt line has already been configured. This settings must be inserted with new table entry in tp871CfgTable[]. The TPMC871 can be found on PCI bus 1 as device 2 and function 0.

```
TP871PCICONFIGSTRUCT tp871CfgTable[] =
{ /* B,  D, F,    regAddr, memBaseAdr, memSpcSize, ioBaseAddr,  ioSpcSize,
intLine */
 {   1,  2, 0, 0x80100000, 0x80110000, 0x00010000, 0xFE001400, 0x00000100,
0x00},

 /* busNo, devNo, funcNo == -1 ==> End of Table */
 {  -1, -1, -1, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00}
};
```

After recompilation the PCI map should have an additional I/O and two additional memory entries showing the TPMC871 address spaces.

New I/O entry:

```
+-----+-----------++---------------------------------------+
>   1 | FE001400h || DevNo:  2 -- FuncNo:  0               |
>     |     :     ||                       (CardBus I/O 0) |
>     |     :     ||                          ENABLED I/O  |
>     |     :     || DeviceID:  Ven:104Ch - Dev:AC56h      |
>     |     :     || SubsysID:  Ven:1498h - Sys:0367h      |
>     | FE0014FFh || Bridge   0 -->  0  ---  (SubOrd:   0) |
+-----+-----------++---------------------------------------+
```

New memory entries:

```
+-----+-----------++---------------------------------------+
|   1 | 80100000h || DevNo:  2 -- FuncNo:  0               |
|     |     :     ||                       (Base Address 0) |
|     |     :     ||                          ENABLED MEM  |
|     |     :     || DeviceID:  Ven:104Ch - Dev:AC56h      |
|     | 80100FFFh || SubsysID:  Ven:1498h - Sys:0367h      |
|:::::|:::::::::::||:::::::::::::::::::::::::::::::::::::::|

+-----+-----------++---------------------------------------+
>   1 | 80110000h || DevNo:  2 -- FuncNo:  0               |
>     |     :     ||                     (CardBus Memory 0) |
>     |     :     ||                          ENABLED MEM  |
>     |     :     || DeviceID:  Ven:104Ch - Dev:AC56h      |
>     |     :     || SubsysID:  Ven:1498h - Sys:0367h      |
>     | 8011FFFFh || Bridge   0 -->  0  ---  (SubOrd:   0) |
+-----+-----------++---------------------------------------+
```

6.2.3 Delivered Configuration Examples

The PCI configuration file (*tp871pciConfig.c*) contains examples for different systems. This example can easily be activated by the definition at the beginning of the file. The current revision contains examples for the TVME8240 with PMC-Span and a MVME5100.

TVME8240 Example Configuration

The definition *TVME8240_SETUP* enables a configuration of the TPMC871 or TPMC872 boards mounted to a PMC-Span. There are four TPMC871 device configuration entries. Unmounted modules will be skipped.

DevNo	FuncNo	PC-Card-Slot
2	0	TPMC871 or 1st Slot of TPMC872
2	1	2nd Slot of TPMC872
3	0	TPMC871 or 1st Slot of TPMC872
3	1	2nd Slot of TPMC872

The definition *TVME8240_BRIDGE_SETUP* enables the configuration setting of the PCI-PCI bridge of the PMC-Span. This configuration maps the address spaces for the TPMC871 configuration before. This configuration is only necessary if the bridge is not configured or the configuration does not match.

This configurations are just examples and may need a modification if devices are mounted to the PCI bus or system address maps change.

MVME5100 Example Configuration

The definition *MVME5100_SETUP* enables a configuration of the TPMC871 or TPMC872 boards mounted to the MVME5100. There are four TPMC871 device configuration entries. Unmounted modules will be skipped.

DevNo	FuncNo	PC-Card-Slot
16	0	TPMC871 or 1st Slot of TPMC872
16	1	2nd Slot of TPMC872
17	0	TPMC871 or 1st Slot of TPMC872
17	1	2nd Slot of TPMC872

This configurations are just examples and may need a modification if devices are mounted to the PCI bus or system address maps change.

6.3 Example Application Configuration

This paragraph describes how the adapter number is assigned to a special TPMC871 slot. Both example applications (pccExa() and ataExa()) use the slot assignment in *pccConfig.c*. The table must be adapted to the used system.

For every possible PC-Card slot the table needs one entry. The TPMC871 slot is identified by PCI bus, device and function number. A 4th value is the assigned adapter number (device number). The table is completed with an entry where the PCI numbers (busNo, devNo, funcNo) are to -1.

```
typedef struct
{
        int             busNo;
        int             devNo;
        int             funcNo;
        ADAPTER         adapter;
} PCC_DEV_TAB_STRUCT;
```

busNo

Specifies the PCI bus number the TPMC871 device is mounted to.

devNo

Specifies the PCI device number the TPMC871 device is mounted to.

funcNo

Specifies the PCI function number on the TPMC871 device. (TPMC871 is always 0, for TPMC872 and TCP872 the function number is 0 for the 1st slot and 1 for the 2nd slot)

adapter

Specifies the adapter or device number of the slot. This number will be used to select the slot by the driver.

Example TVME8240

The list below makes the assignment for three PC-Card slots.

A TPMC871 is mounted on Bus 1 Device 2. It will be accessible as adapter 0. A TPMC872 is mounted ro PCI bus 1 Device 3. The 1st slot will be accessible as adapter 1, and the 2nd will be accessible as adapter 3.

```
PCC_DEV_TAB_STRUCT dev_tab[] =
{   /*  BusNo,  DevNo,  FuncNo, adapter */
    {       1,      2,      0,      0},         /* TPMC871 */
    {       1,      3,      0,      1},         /* TPMC872 (1) */
    {       1,      3,      1,      2},         /* TPMC872 (2) */
    {      -1,     -1,     -1,      0}          /* End of List */
};
```

The assignment configuration file *pccConfig.c* contains examples for TVME8240 with PMC-Span, MVME5100 and a standard PC (TEWS_PC).

7 Application Hints

This chapter gives a quick overview, what has to do before using the driver calls.

7.1 Start up PC Card Socket Functions

This part shows how to startup the socket functions to be used from the application layer.

Modify the table which makes the adapter assignment to PC-Card sockets. This is described in this manual in *6.3 Example Application Configuration*.

If the assignment is done, the socket functions of the TPMC871 have to be initialized:

```
result = tpmc871_ss_init();
if (result)
{
     printf("ERROR (%08lX)\n", result);
     return;
}
printf("OK\n");
```

The next step that has to be done is the initialization of the PC Card manager:

```
printf("Initialize PC Card manager ... ");
result = pcc_init();
if (result)
{
     printf("ERROR (%08lX)\n", result);
     return;
}
printf("OK\n");
```

After that we have to connect the TPMC871 adapters to the PC Card manager. The easiest way is to make a loop for all assigned TPMC871 modules:

```
adNo = 0;
while (dev_tab[adNo].busNo != -1)
{
```

Initialize the TPMC871 (adapter) that is placed at the position specified in the configuration table. If there is a module mounted, the function will return a local adapter number, which is used by the TPMC871 socket functions and the entry point to call the TPMC871 socket functions.

```
    result = tpmc871_init(  dev_tab[adNo].busNo,
                            dev_tab[adNo].devNo,
                            dev_tab[adNo].funcNo,
                            &locAdapter,
                            &locEntry);
    if (result)
    {
        printf("ERROR (%08lX)\n", result);
    }
    else
    {
        printf("OK\n");
```

The initialization was successful and now it has to be connected to PC Card manager. Specify adapter number used for this adapter and the two parameters received from the *tpmc871_init()* function.

```
        result = pcc_adapter_init(   dev_tab[adNo].adapter,
                                     locAdapter,
                                     locEntry);
        if (result)
        {
            printf("ERROR (%08lX)\n", result);
        }
        else
        {
            printf("OK\n");
```

The TPMC871 is now accessible via the PC Card interface by the assigned adapter number.

If intending to use interrupts, it is now a good time to connect the interrupt function into the VxWorks system. The third parameter specifies the adapter. This depends on the parameter of the interrupt service function. Read out the interrupt line from TPMC871 PCI configuration header and convert the interrupt line into an interrupt vector which is necessary to connect the interrupt function.

```
                /* Read interrupt line from PCI header */
                pciConfigInByte(   dev_tab[adNo].busNo,
                                   dev_tab[adNo].devNo,
                                   dev_tab[adNo].funcNo,
                                   PCI_CFG_DEV_INT_LINE,
                                   &intLine);
                /* Convert to vecore */
                intVector = tdh_level2vector(intLine);

                /* Connect Interrupt Function */
                /* using function from include/tdhal.h */
                if(tdh_intConnect( INUM_TO_IVEC(intVector),
                                   (VOIDFUNCPTR)pccInterrupt,
                                   (int)dev_tab[adNo].adapter) == ERROR)
                {
                     printf("    Connecting Interrupt Failed (%08Xh)!!!\n",
                     errnoGet());
                }
```

The interrupt level can now be enabled.

```
                tdh_intEnable(dev_tab[adNo].intLvl);
          }
     }
}
```

Then the unified socket functions are ready to use via the PC Card manager.

7.2 Start up the PC Card ATA Driver

7.2.1 No XBD support (VxWorks V6.1 and older)

This part shows how to startup the socket functions, PC-Card ATA disk driver and the DOS file system to be used from the application layer. The initialization is mainly identical as described before in *7.1 Start up PC Card Socket Functions*.

Modify the table which makes the adapter assignment to PC-Card sockets. This is described in *6.3 Example Application Configuration*.

If the assignment is done, the socket functions of the TPMC871 have to be initialized:

```
typedef char drv_name_struct[20];
LOCAL drv_name_struct   *pDrv_name = NULL;
LOCAL unsigned long     *pDrv_handle = NULL;
LOCAL DOS_VOL_DESC      **vol_desc = NULL;
```

Initialize the drivers and the dosFileSystem.

```
if (dosFsInit(MAX_FILES) != OK)
{
     printf("ERROR (0x%08X)\n", errnoGet());
     return;
}
```

Initialize Socket Service Handlers (refer to *7.1 Start up PC Card Socket Functions*)

```
printf("Initialize Socket Services for TPMC871 ... ");
result = tpmc871_ss_init();
…

result = pcc_init();
…

adNo = 0;
while (dev_tab[adNo].busNo != -1)
{
     result = tpmc871_init(  dev_tab[adNo].busNo,
                             dev_tab[adNo].devNo,
                             dev_tab[adNo].funcNo,
                             &locAdapter,
                             &locEntry);
     …
}

adaNum = adNo;     /* Store number of adapters */
```

Now Memory for disk device handles, ATA Disk driver names and volume descriptors have to be allocated.

```
pDrv_handle = (unsigned long*)malloc(adNo * sizeof(unsigned long));
if(pDrv_handle == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
vol_desc = (DOS_VOL_DESC**)malloc(adNo * sizeof(DOS_VOL_DESC*));
if(vol_desc == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
pDrv_name = (drv_name_struct*)malloc(adNo * sizeof(drv_name_struct));
if(pDrv_name == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
```

The next step is to initialize the ATA Disk driver:

```
result = pccAtaDrv();
if (result)
{
     printf("ERROR (%08lX)\n", result);
     return;
}
```

Now the ATA disk devices have to be created by doing a loop over all sockets. The interrupt line and vector has to be assigned with the device creation.

```
for (adNo = 0; adNo < adaNum; adNo++)
{
     pciConfigInByte(   dev_tab[adNo].busNo,
                        dev_tab[adNo].devNo,
                        dev_tab[adNo].funcNo,
                        PCI_CFG_DEV_INT_LINE,
                        &intLine);
     intVector = tdh_level2vector(intLine);
```

```
        pDrv_handle[adNo] = pccAtaDevCreate(  adNo,
                                              0,        /* sockNo always 0 */
                                              intLine,
                                              intVector);
        if (pDrv_handle[adNo])
        {
            printf("OK -- Handle: %08lX\n", pDrv_handle[adNo]);
        }
        else
        {
            printf("ERROR (%08Xh)\n", errnoGet());
        }
        /* Assigne driver name */
        sprintf(pDrv_name[adNo], "/PCC%d:", adNo);
    }
```

The last step is creating the DOS volume for all successfully initialized adapters.

```
    for (adNo = 0; adNo < adaNum; adNo++)
    {
        sockNo = 0;         /* Only one socket per adapter */

        if (pDrv_handle[adNo])
        {
            printf("    Create Volume '%s' on Adapter %ld - Socket %ld ... ",
                   pDrv_name[adNo],
                   adNo,
                   0);     /* sockNo always 0 */
            vol_desc[adNo] = dosFsDevInit (  pDrv_name[adNo],
                                             (BLK_DEV*)pDrv_handle[adNo],
                                             NULL);
            if(vol_desc[adNo] == 0)
            {
                printf("ERROR (0x%08X)\n", errnoGet());
            }
            else
            {
                printf("OK (VolDesc: 0x%08X)\n",
                       (unsigned int)vol_desc[adNo]);
            }
        }
    }
```

7.2.2 XBD support (VxWorks V6.2 and newer)

This part shows how to startup the socket functions, PC-Card ATA disk driver if VxWorks need XBD support. INCLUDE_XBD must be defined for the project (prjParams.h). This is generally done in the BSP by default. The initialization is mainly identical as described before in *7.1 Start up PC Card Socket Functions*.

Modify the table which makes the adapter assignment to PC-Card sockets. This is described in *6.3 Example Application Configuration*.

If the assignment is done, the socket functions of the TPMC871 have to be initialized:

```
typedef char drv_name_struct[20];
LOCAL drv_name_struct   *pDrv_name = NULL;
LOCAL unsigned long     *pDrv_handle = NULL;
LOCAL DOS_VOL_DESC      **vol_desc = NULL;
```

Initialize the drivers and the dosFileSystem.

```
if (dosFsLibInit(MAX_FILES, 0) != OK)
{
     printf("ERROR (0x%08X)\n", errnoGet());
     return;
}
```

Initialize Socket Service Handlers (refer to *7.1 Start up PC Card Socket Functions*)

```
printf("Initialize Socket Services for TPMC871 ... ");
result = tpmc871_ss_init();
…

result = pcc_init();
…

adNo = 0;
while (dev_tab[adNo].busNo != -1)
{
     result = tpmc871_init(  dev_tab[adNo].busNo,
                             dev_tab[adNo].devNo,
                             dev_tab[adNo].funcNo,
                             &locAdapter,
                             &locEntry);
     …
}

adaNum = adNo;     /* Store number of adapters */
```

Now Memory for disk device handles, ATA Disk driver names and volume descriptors have to be allocated.

```
pDrv_handle = (unsigned long*)malloc(adNo * sizeof(unsigned long));
if(pDrv_handle == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
vol_desc = (DOS_VOL_DESC**)malloc(adNo * sizeof(DOS_VOL_DESC*));
if(vol_desc == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
pDrv_name = (drv_name_struct*)malloc(adNo * sizeof(drv_name_struct));
if(pDrv_name == NULL)
{
     printf("Can't allocate memory\n");
     return;
}
```

The next step is to initialize the ATA Disk driver:

```
result = pccAtaDrv();
if (result)
{
     printf("ERROR (%08lX)\n", result);
     return;
}
```

Now the ATA disk devices have to be created by doing a loop over all sockets. The interrupt line and vector has to be assigned with the device creation.

```
for (adNo = 0; adNo < adaNum; adNo++)
{
     pciConfigInByte(   dev_tab[adNo].busNo,
                        dev_tab[adNo].devNo,
                        dev_tab[adNo].funcNo,
                        PCI_CFG_DEV_INT_LINE,
                        &intLine);
     intVector = tdh_level2vector(intLine);
```

```
    /* Assign drive name */
    sprintf(pDrv_name[adNo], "/PCC%d", adNo);
    pDrv_handle[adNo] = pccAtaXbdDevCreate (   adNo,
                                               0,        /* always 0 */
                                               intLine,
                                               intVector,
                                               pDrv_name[adNo]);
    if (pDrv_handle[adNo])
    {
        printf("OK -- Handle: %08lX\n", pDrv_handle[adNo]);
    }
    else
    {
        printf("ERROR (%08Xh)\n", errnoGet());
    }

    /* Add Partition number to name (always 0) */
    sprintf(pDrv_name[adNo], "%s:0", pDrv_name[adNo], adNo);
}
```

The last step “creating the disk devices” is no more necessary with XBD support.

7.3 Setup Card Access

This chapter gives a simple example how to setup a socket to allow accesses to a PC Card. The addressing depends on the used BSP and the configuration of TPMC871 modules. Valid addresses can be determined by the function call Inquire Windows (*PCC_INQ_WINDOW*, refer to *4.1.5 InquireWindow*).

7.3.1 Access PC-Card memory

Example:

Map a window to a memory card. Place the socket (0) on adapter (2), using window (1). The windows address shall be at 0xfd100000 with a size of 0x1000. The PC Card internal offset shall be 0x0. The access shall be write protected. Setup the slot with fix parameters. In complex systems it will be better to read out the capabilities or the actual setup, to avoid conflicts.

Start after the initialization of the drivers:

First set up the adapter (2). Don't use any interrupts or any special modes.

```
result = pcc_entry (    PCC_SET_ADAPTER,
                        (unsigned long)2,
                        (unsigned long)0,
                        (unsigned long)0);
if (result)
{
     /* error */
}
```

Then set up the socket (0). Don't use any interrupts or any special modes. Set Vcc to 5V (Powerindex 2), Vpp is not needed. The power setting must match with the PC-Card requirements. The PC-Card uses a memory interface.

```
VppLevels[0] = 0;       /* 0V */
VppLevels[1] = 0;       /* 0V */
result = pcc_entry (    PCC_SET_SOCKET,
                        (unsigned long)2,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)2,           /* 5 V */
                        (unsigned long)VppLevels,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)0,
                        (unsigned long)PCC_IF_MEMORY,
                        (unsigned long)0);
```

```
if (result)
{
     /* error */
}
```

Now it's time to configure the window (1). The size shall be 0x1000, access speed shall be 200ns, enable window access. Now the window is open, but the card access has to be set up with the *setPage* function.

```
speed.speed = 0x2;
result = pcc_entry(     PCC_SET_WINDOW,
                        (unsigned long)2,
                        (unsigned long)1,
                        (unsigned long)0,
                        (unsigned long)0x1000,      /* size */
                        (unsigned long)PCC_WS_ENABLE,
                        (unsigned long)&speed,
                        (unsigned long)0xFD100000); /* Baseaddr */
if (result)
{
     /* error */
}
```

At present set up the card access. Use page (0), because the controller doesn't allow paging. Read from the beginning of the card, that means offset zero (0). Enable the access and make it write protected.

```
result = pcc_entry (    PCC_SET_PAGE,
                        (unsigned long)2,
                        (unsigned long)1,
                        (unsigned long)0,
                        (unsigned long)PCC_PS_ENABLED | PCC_PS_WP,
                        (unsigned long)0); /* card offset */
```

The memory card can now be accessed at the address space from 0xfd100000 up to 0xfd100fff.

7.3.2 Access PC-Card attribute memory

Attribute memory is access is setup in the same way memory access is set up, just the *PCC_PS_ATTRIBUTE* has to be added to the page attributes.

Example:

The example allows access to the attribute space without write protection.

```
result = pcc_entry (    PCC_SET_PAGE,
                        (unsigned long)2,
                        (unsigned long)1,
                        (unsigned long)0,
                        (unsigned long)PCC_PS_ENABLED | PCC_PS_ATTRIBUTE,
                        (unsigned long)0);      /* card offset */
```

7.3.3 Access PC-Card I/O

I/O addresses on PC-Cards can be accessed via I/O windows. If a window can map I/O can be determined by the socket function Inquire Windows (Refer to *4.1.5 InquireWindow*). The current supported boards allow I/O mapping in window 5 and 6.

The setup mainly looks like the setup for memory windows. Different setup is necessary during the call of *PCC_SET_SOCKET* where the interface type *PCC_IF_MEMORY* has to be replaced by *PCC_IF_IO*. The next modification must be done when calling *PCC_SET_WINDOW*, an I/O window must be selected and the window flag *PCC_WS_IO* must be added. The last difference is that the function *PCC_SET_PAGE* has not to be called.

8 Appendix

8.1 Error Codes

This chapter gives a short description of the error codes, which can be created by this software interface.

8.1.1 TPMC871 Error Codes

TP871_NO_ERROR	0x00000000	Execution OK
TP871_NO_DEVICE	0x87100000	No TPMC871 found at specified slot
TP871_SS_NOT_INIT	0x87100001	The socket functions are not initialized yet.
TP871_SS_MAX_ADAPTER	0x87100002	Maximum number of adapters is already initialized.
TP871_SS_CANT_ALLOC_MEM	0x87100003	The socket functions can not allocate memory.
TP871_SS_ILL_FUNCTION	0x87100004	Illegal socket function code specified.
TP871_SS_UNIMPL_FUNCTION	0x87100005	Unimplemented socket function called.

8.1.2 Socket Function Error Codes

PCC_NO_ERROR	0x00000000	Execution OK
PCC_MAX_ADAPTER	0xF8710002	Maximum number of adapters is already connected.
PCC_BAD_ADAPTER	0xF8710101	Bad adapter number specified
PCC_BAD_ATTRIBUTE	0xF8710102	Bad combination of attributes
PCC_BAD_BASE	0xF8710103	Bad base specified
PCC_BAD_IRQ	0xF8710106	Bad IRQ state specified
PCC_BAD_OFFSET	0xF8710107	Bad offset specified
PCC_BAD_PAGE	0xF8710108	Bad page number specified
PCC_BAD_SIZE	0xF871010A	Bad size specified
PCC_BAD_SOCKET	0xF871010B	Bad socket number specified
PCC_BAD_TYPE	0xF871010D	Bad window type specified
PCC_BAD_VCC	0xF871010E	Bad Vcc index specified
PCC_BAD_VPP	0xF871010F	Bad Vpp index specified
PCC_BAD_WINDOW	0xF8710111	Bad window number specified
PCC_BAD_SERVICE	0xF8710115	Bad socket function specified
PCC_BAD_SPEED	0xF8710117	Bad speed specified

8.1.3 ATA Disk Driver Error Codes

S_pccAta_NOCARD	0xA8710001	No disk in the specified socket
S_pccAta_NOWINDOW	0xA8710002	Can not allocate a matching window
S_pccAta_WINSIZE	0xA8710003	Specified window size can not be mapped
S_pccAta_ATADISK	0xA8710004	No ATA Disk found at specified socket
S_pccAta_PARTERR	0xA8710005	Partition error
S_pccAta_ACCTIMEOUT	0xA8710006	Timeout while disk access
S_pccAta_ACCERROR	0xA8710007	Error occurred while disk access
S_pccAta_WINOFFSET	0xA8710008	Illegal window offset specified