PCI-7841/cPCI-7841/PM-7841/
PMC-7841/PMC-7841G

Dual-Port Isolated
CAN Interface Card
User's Guide

Recycled Paper

Manual Rev. 2.21: October 14, 2003

Part No: 50-11109-100

Trademarks

PCI-7841, cPCI-7841, PM-7841, PMC-7841 and PMC-7841G are registered trademarks of ADLINK Technology Inc. Other product names mentioned herein are used for identification purposes only and may be trademarks and/or registered trademarks of their respective companies.

Getting Service from ADLINK

Customer Satisfaction is top priority for ADLINK TECHNOLOGY INC. If you need any help or service, please contact us.

ADLINK TECHNOLOGY INC.			
Web Site	http://www.adlinktech.com		
Sales & Service	Service@adlinktech.com		
TEL	+886-2-82265877	FAX	+886-2-82265717
Address	9F, No. 166, Jian Yi Road, Chungho City, Taipei, 235 Taiwan		

Please email or FAX your detailed information for prompt, satisfactory, and consistent service.

Detailed Company Information			
Company/Organization			
Contact Person			
E-mail Address			
Address			
Country			
TEL		FAX	
Web Site			
Questions			
Product Model			
Environment	OS: Computer Brand: M/B: CPU: Chipset: BIOS: Video Card: NIC: Other:		
Detail Description			
Suggestions for ADLINK			

Table of Contents

1

Introduction

The PCI/cPCI/PM/PMC-7841(G) is a Controller Area Network (CAN) interface card used for industrial PCs with PCI, Compact-PCI, and PC104 bus that supports dual port CAN interfaces running independently or bridged simultaneously. The built-in CAN controller provides bus arbitration and error detection with auto correction and re-transmission functions. The PCI cards are Plug and Play therefore it is not necessary to set any jumpers for matching the PC environment.

The CAN (Controller Area Network) is a serial bus system originally developed by Bosch for use in automobiles, and is increasingly becoming the standard used in industry automation. It's multi-master protocol, real-time capability, error correction, and high noise immunity make it especially suited for intelligent I/O devices control networks.

The PCI/cPCI/PM/PMC-7841(G) is programmed by using the ADLINK's software library. The programming of this PCI card is as easy as AT bus add-on cards.

1.1 PCI/cPCI/PM/PMC-7841(G) Features

The PCI-7841 is a Dual-Port Isolated CAN Interface Card with the following features:

- Two independent CAN network operation
- Bridge support
- Compatible with CAN specification 2.0 parts A and B
- Optically isolated CAN interface (up to 2500Vrms isolation protection)
- Direct memory mapping to the CAN controllers
- Up to 1Mbps programmable transfer rate
- PCI bus Plug and Play
- DOS library and examples included

The cPCI-7841 is a Dual-Port Isolated CAN Interface Card with the following features:

- Two independent CAN network operation
- Bridge support
- Compatible with CAN specification 2.0 parts A and B
- Optically isolated CAN interface (up to 2500Vrms isolation protection)
- Direct memory mapping to the CAN controllers
- Up to 1Mbps programmable transfer rate
- PCI bus Plug and Play
- compact-PCI industry bus
- DOS library and examples included

The PM-7841 is a Dual-Port Isolated CAN Interface Card with the following features:

- Two independent CAN network operation
- Bridge support
- Compatible with CAN specification 2.0 parts A and B

- Optically isolated CAN interface (up to 2500 Vrms isolation protection)
- Direct memory mapping to the CAN controllers
- Up to 1Mbps programmable transfer rate
- DIP-Switch for base address configuration
- Software Programmable Memory-Mapped Address
- PC-104 industry form factor
- DOS library and examples included

The PMC-7841(G) is a Dual-Port Isolated CAN Interface Card with the following features:

- Two independent CAN network operation
- Bridge support
- Compatible with CAN specification 2.0 parts A and B
- Optically isolated CAN interface (up to 2500 Vrms isolation protection)
- Direct memory mapping to the CAN controllers
- Up to 1Mbps programmable transfer rate
- PCI bus Plug and Play
- Specifically designed for use in GEME embedded systems

1.2 Applications

- Industry automation
- Industry process monitoring and control
- Manufacture automation
- Product testing

1.3 Specifications

PCI-7841 Specification Table

Ports	2 CAN channels (V2.0 A, B)
CAN Controller	SJA1000
CAN Transceiver	82c250
Signal Support	CAN_H, CAN_L
Isolation Voltage	2500 Vrms
Connectors	Dual DB-9 male connectors
Operation Temperature	0 – 60° C
Storage Temperature	-20° – 80° C
Humidity	5% – 95% non-condensing
IRQ Level	Set by Plug and Play BIOS
I/O port address	Set by Plug and Play BIOS
Power Consumption (without external devices)	400mA @5VDC (Typical) 900mA @5VDC (Maximum)
Size	132(L)mm x 98(H)mm

cPCI-7841 Specification Table

Ports	2 CAN channels (V2.0 A, B)
CAN Controller	SJA1000
CAN Transceiver	82c250
Signal Support	CAN_H, CAN_L
Isolation Voltage	2500 Vrms
Connectors	Dual TB 5P connectors
Operation Temperature	0 – 60° C
Storage Temperature	-20° – 80° C
Humidity	5% – 95% non-condensing
IRQ Level	Set by Plug and Play BIOS
I/O port address	Set by Plug and Play BIOS
Power Consumption (without external devices)	400mA @5VDC (Typical) 900mA @5VDC (Maximum)
Size	132(L)mm x 98(H)mm

PM-7841 Specification Table

Ports	2 CAN channels (V2.0 A, B)
CAN Controller	SJA1000
CAN Transceiver	82c250/82c251
Signal Support	CAN_H, CAN_L
Isolation Voltage	1000 Vrms
Connectors	Dual 5 male connectors
Operation Temperature	0 ~ 60° C
Storage Temperature	-20° – 80° C
Humidity	5% – 95% non-condensing
IRQ Level	Set by Jumper
I/O port address	Set by DIP Switch
Memory Mapped Space	128 Bytes by Software
Power Consumption (without external devices)	400mA @5VDC (Typical) 900mA @5VDC (Maximum)
Size	90.17(L)mm x 95.89(H)mm

PMC-7841 (G) Specification Table

Ports	2 CAN channels (V2.0 A, B)
CAN Controller	SJA1000
CAN Transceiver	82c250
Signal Support	CAN_H, CAN_L
Isolation Voltage	2500 Vrms
Connectors	Dual DB-9 male connectors
Operation Temperature	0 – 60° C
Storage Temperature	-20° – 80° C
Humidity	5% – 95% non-condensing
IRQ Level	Set by Plug and Play BIOS
I/O port address	Set by Plug and Play BIOS
Power Consumption (without external devices)	400mA @5VDC (Typical) 900mA @5VDC (Maximum)
Size	PMC-7841: 156(L)mm x 74(H)mm PMC-7841G: 149(L)mm x 74(H)mm

2

Installation

This chapter describes how to install the PCI/cPCI/PM/PMC-7841(G). Please carefully review the package contents and unpacking information.

2.1 Before Installing the PCI/cPCI/PM/PMC-7841(G)

The PCI/cPCI/PM/PMC-7841(G) card contains sensitive electronic components that can be easily damaged by static electricity.

The card should be used on a grounded anti-static mat. The operator should be wearing an anti-static wristband, grounded to the same point as the anti-static mat.

Inspect the card module carton for obvious damage. Shipping and handling may cause damage to the module. Be sure there is no shipping and handing damage on the module before processing.

After opening the card module carton, extract the system module and place it only on a grounded anti-static surface, component side up.

Note: DO NOT APPLY POWER TO THE CARD IF IT HAS BEEN DAMAGED.

You are now ready to install your PCI/cPCI/PM/PMC-7841(G).

2.2 Installing the PCI/PMC-7841(G)

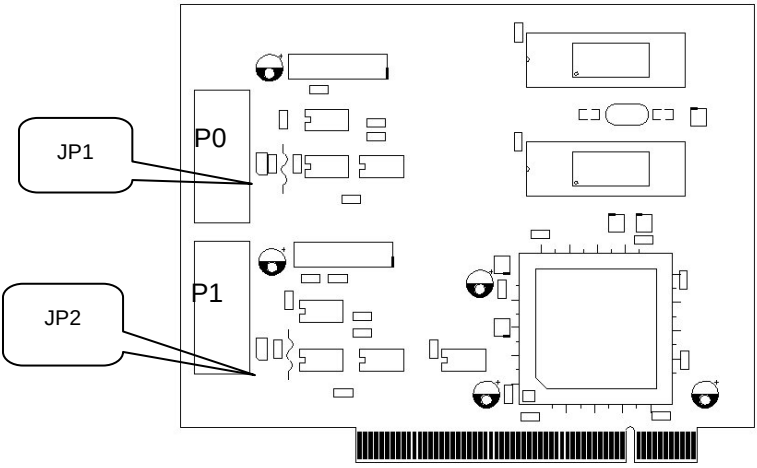
What's Included

In addition to this *User's Manual*, the package includes the following items:

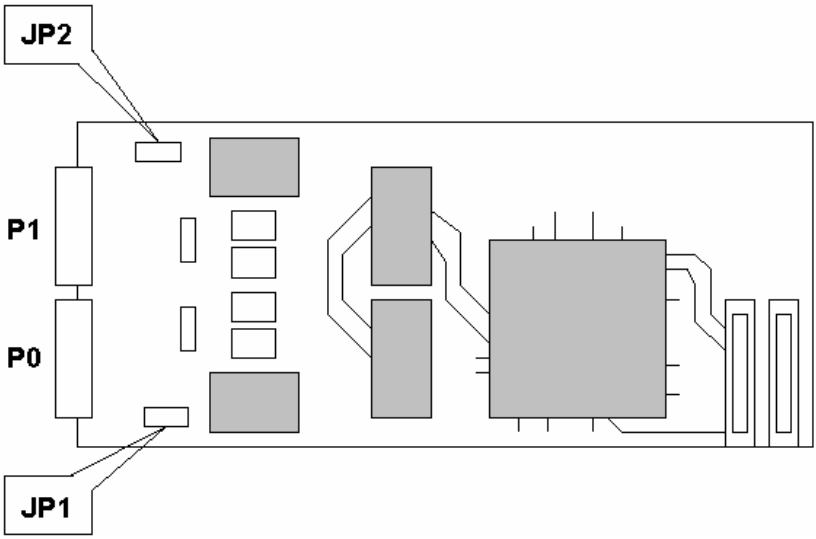
- PCI/PMC-7841 Dual Port PCI Isolated CAN Interface Card
- ADLINK All-In-One CD-ROM

If any of these items are missing or damaged, contact the dealer from whom you purchased the product. Save shipping materials and carton to ship or store the product in the future.

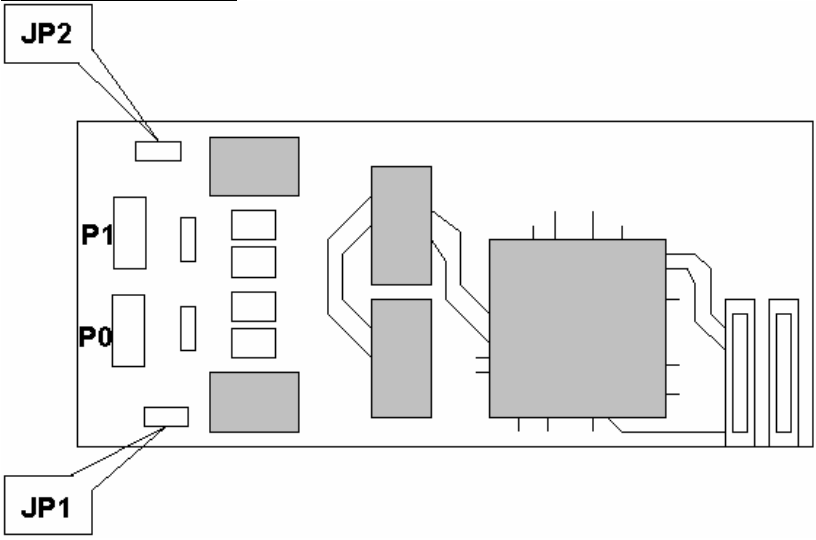
PCI-7841 Layout



PMC-7841 Layout



PMC-7841G Layout

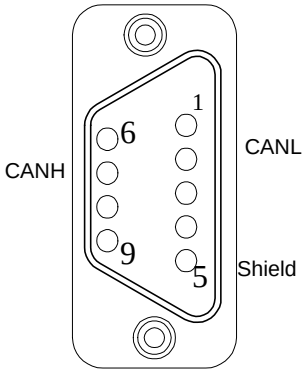


Terminator Configuration

A 120Ω terminal resistor is installed for each port; JP1 enables the terminal resistors for p0 and JP2 enables the terminal resistors for p1

Connector Pins

P0 and P1 are CAN connectors as shown below:



DIP-9

2.3 Installing the cPCI-7841

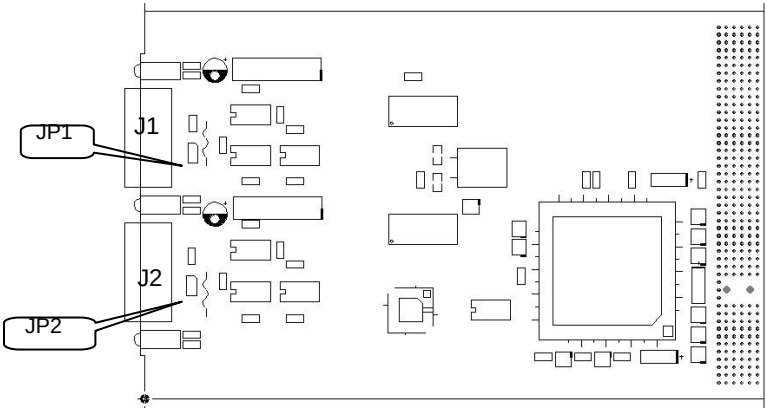
What's Included

In addition to this *User's Manual*, the package includes the following items:

- cPCI-7841 Dual Port Compact-PCI Isolated CAN Interface Card
- ADLINK All-In-One CD-ROM

If any of these items are missing or damaged, contact the dealer from whom you purchased the product. Save shipping materials and carton to ship or store the product in the future.

cPCI-7841 Layout

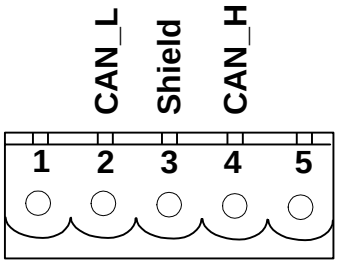


Terminator Configuration

A 120Ω terminal resistor is installed for each port; JP1 enables the terminal resistor for J1 and JP2 enables the terminal resistor for J2.

Connector Pin Define

J1 and J2 are CAN connectors as shown below:


CAN_L
Shield
CAN_H
1
2
3
4
5

2.4 Installing the PM-7841

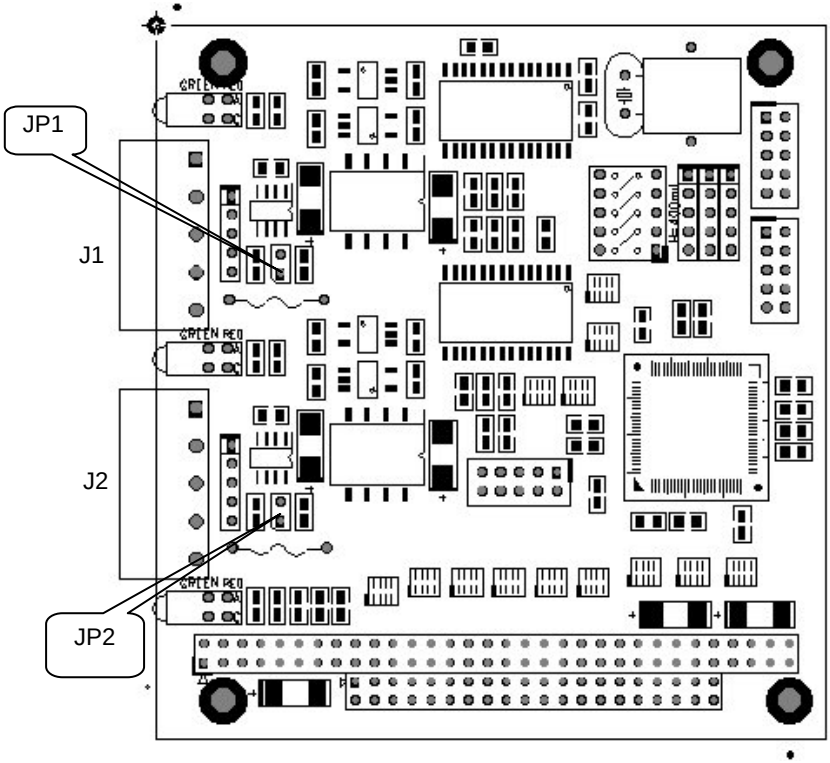
What's Included

In addition to this *User's Manual*, the package includes the following items:

- PM-7841 Dual Port PC-104 Isolated CAN Interface Card
- ADLINK All-In-One CD-ROM

If any of these items are missing or damaged, contact the dealer from whom you purchased the product. Save shipping materials and carton to ship or store the product in the future.

PM-7841 Layout

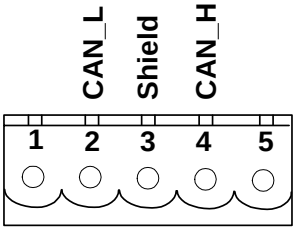


Terminator Configuration

A 120Ω terminal resistor is installed for each port, while JP1 enables the terminal resistor for J1 and JP2 enables the terminal resistor for J2.

Connector Pin Define

J1 and J2 are CAN connectors as shown below:



2.4 Jumper and DIP Switches

The output of each channel and base address are configurable by setting jumpers and DIP switches on the PM-7841. The card's jumpers and switches are preset at the factory. Under normal circumstances, there jumper settings should not need adjustment.

A jumper switch is closed ("shorted") with the plastic cap inserted over two pins of the jumper. A jumper is open with the plastic cap inserted over one or no pin(s) of the jumper.

2.5 Base Address Setting

The PM-7841 requires 16 consecutive address locations in the I/O address space. The base address of the PM-7841 is restricted by the following conditions.

1. The base address must be within the range 200hex to 3F0hex.
2. The base address should not conflict with any PC reserved I/O address.

The PM-7841's I/O port base address is selectable by a 5 position DIP switch SW1 (refer to Table 2.1). The address settings for I/O ports from Hex 200 to Hex 3F0 are described in Table 2.2 below (active low). The default base address of the PM-7841 is set to **hex 200** in the factory (see Figure below).

SW1 : Base Address = 0x200

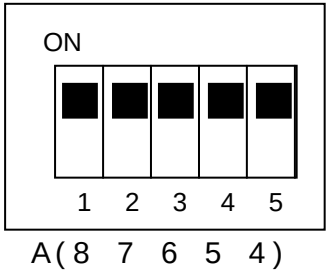


Figure Default Base Address Configuration

I/O port address(hex)	fixed A9	1 A8	2 A7	3 A6	4 A5	5 A4
200-20F	OFF (1)	ON (0)	ON (0)	ON (0)	ON (0)	ON (0)
210-21F	OFF (1)	ON (0)	ON (0)	ON (0)	ON (0)	OFF (1)
:						
(*) 2C0-2CF	OFF (1)	ON (0)	OFF (1)	OFF (1)	ON (0)	ON (0)
:						
300-30F	OFF (1)	OFF (1)	ON (0)	ON (0)	ON (0)	ON (0)
:						
3F0-3FF	OFF (1)	OFF (1)	OFF (1)	OFF (1)	OFF (1)	OFF (1)

(*): default setting ON : 0

X: don't care OFF : 1

Note: A4, ..., A9 correspond to PC-104(ISA) bus address lines.

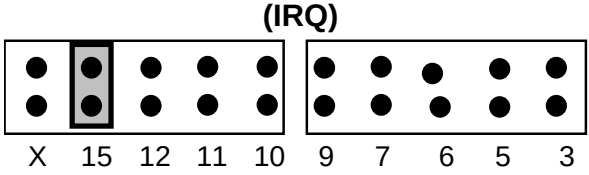
2.6 IRQ Level Setting

A hardware interrupt can be triggered by the external Interrupt signal (JP3 and JP4).

The jumper setting is specified as below:

Note: Be certain that there are no other add-on cards sharing the same interrupt level in the system.

Interrupt Default Setting = IRQ15



IRQ Setting

3

Function Reference

The cPCI/PCI/PMC-7841(G) functions are organized into the following sections:

- **CAN layer functions**
  - Card Initialization and configuration functions
  - CAN layer I/O functions
  - CAN layer status functions
  - CAN layer Error and Event Handling functions

Specific associated functions are presented in this chapter.

3.1 Functions Table

CAN layer functions		
Function Type	**Function Name**	**Page**
PM-7841 Initial	PM7841_Install()	29
	GetDriverVersion()	29
	CanOpenDriver()	31
	CanCloseDriver()	32
	CanConfigPort()	33
	CanDetectBaudrate()	34
	CanEnableReceive()	37
	CanDisableReceive()	38
	CanSendMsg()	39
	CanRcvMsg()	40
	CanGetRcvCnt()	55
	CanClearOverrun()	41
	CanClearRxBuffer()	42
	CanClearTxBuffer()	43
	CanGetErrorCode()	44
	CanGetErrorWarningLimit()	44
	CanSetErrorWarningLimit()	47
	CanGetRxErrorCount()	49
	CanGetTxErrorCount()	49
	CanSetTxErrorCount()	51
	CanGetPortStatus()	52
	CanGetLedStatus()[1]	53
	CanSetLedStatus()[1]	54

Error and Event handling functions		
Operation System	**Function Name**	**Page**
DOS	CanInstallCallBack()	56
	CanRemoveCallBack()	58
Windows 95/98/NT	CanInstallEvent()	62

Note: Only for the compactPCI and PC-104 versions.

3.1.1 PORT_STRUCT structure define

The **PORT_STRUCT** structure defines the mode of id-mode, acceptance code, acceptance mask, and baud rate of a physical CAN port. It is used by the ***CanPortConfig()***, and ***CanGetPortStatus()*** functions.

```
typedef struct _tagPORT_STRUCT
{
  int mode;                  // 0    for 11-bit;      1 for 29-bit
    DWORD accCode, accMask;
    int baudrate;
    BYTE brp, tseg1, tseg2;      // Used only if baudrate = 4
    BYTE sjw, sam;               // Used only if baudrate = 4
    }PORT_STRUCT;
```

Members

mode: 0 means using 11-bit in CAN-ID field

1 means using 29-bit in CAN-ID field.

accCode: Acceptance Code for CAN controller.

accMask: Acceptance Mask for CAN controller.

accCode and accMask is used to assign the accept ID for CAN controller.

Example 1: You want to accept all IDs.
Can ID=accCode ^ accMask=0xff.

Example 2: You want to accept only ID1 and ID2,
ID=ID1+ID2=0x01 + 0x02=0x03;
So you may set as follows,
acccode=0x03;
accMask=0x00;

Example 3: You want to accept all ID except ID1 and ID2
ID1=0x01,ID2=0x02
ID=ID1+ID2=0x03
accCode=0
accMask=0x7fc

baudrate: Baudrate settings for the CAN controller.

Value	Baudrate
0	125kbps

1	250kbps
2	500kbps
3	1Mbps
4	User-Defined

brp, tseg1, tseg2, sjw, sam: Use for User-Defined Baudrate

Bit interpretation of bus timing register0

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
sjw.1	siw.0	brp.5	brp.4	brp.3	brp.2	brp.1	brp.0

* Baud Rate Prescaler (BRP, brp): The period of CAN system clock Tscl is programmable and determines the individual bit timing. The CAN system clock is calculated using the following equation: Tscl = 2 * Tclk * (32* brp.5 + 16*brp.4 + 8*brp.3 + 4*brp.2 + 2*brp.1 + brp.0 + 1)
where Tclk = time period of the XTAL frequency =1/16MHz
brp=32* brp.5 + 16*brp.4 + 8*brp.3 + 4*brp.2 + 2*brp.1 + brp.0

* Synchronization Jump Width (SJW, sjw)
Tsjw = Tscl* (2*sjw.1 + sjw.0 + 1)
sjw=2*sjw.1+sjw.0

Bit interpretation of bus timing register1

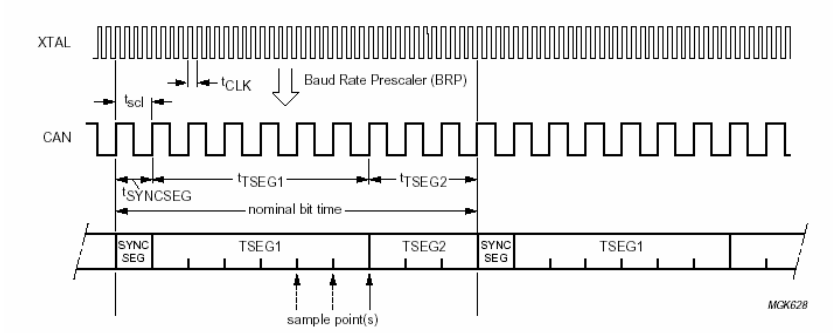
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
sam	tseg2.2	tseg2.1	tseg2.0	tseg1.3	tseg1.2	tseg1.1	tseg1.0

Sampling (SAM)

BIT	VALUE	Function
SAM	1	triple; the bus is sampled three times; recommended for low/medium speed buses (class A and B) where filtering spikes on the bus line is beneficial
	0	single; the bus is sampled once; recommended for high speed buses (SAE class C)

Sam=SAM;
* Time Segment 1 (TSEG1) and Time Segment 2 (TSEG2)
TSEG1 and TSEG2 determine the number of clock cycles per bit period and the location of the sample point, where:
Tsyncseg = 1* Tscl
Ttseg1 = Tscl * (8* tseg1.3 + 4* tseg1.2 + 2*tseg1.1 + tseg1.0 + 1)
Ttseg2 = Tscl * (4* tseg2.2 + 2 *tseg2.1 + tseg2.0 + 1)

For example:



Possible values are brp=000001, tseg1=0101, and tseg2=010.

For example:
If you set brp=000001, tseg1=0101, tseg2=010, then
Tscl=2 * Tclk * (32* brp.5 + 16*brp.4 + 8*brp.3 + 4*brp.2 + 2*brp.1 + brp.0 + 1)
=2 * Tclk * (1 + 1)
=1/4MHz
Ttseg1= Tscl * (8* tseg1.3 + 4* tseg1.2 + 2*tseg1.1 + tseg1.0 + 1)
= Tscl * (4+ 1+1)
=Tscl * 6
Ttseg2= Tscl * (4* tseg2.2 + 2 *tseg2.1 + tseg2.0 + 1)
= Tscl * (2+1)
= Tscl * 3
Can controller Baudrate=1/(Tsyncseg+Ttseg1+Ttseg2)
=1/Tscl(1+6+3)
=4MHz/10=250kHz

See Also

CanPortConfig(),CanGetPortStatus(), and PORT_STATUS structure

3.1.2 PORT_STATUS structure define

The **PORT_STATUS** structure defines the status register and **PORT_STRUCT** of CAN port. It is used by the ***CanGetPortStatus()*** functions.

```
typedef struct _tagPORT_STATUS
 {
      PORT_STRUCT port;
      PORT_REG status;
  }PORT_STATUS;
```

Members

port: **PORT_STRUCT data**

status: status is the status register mapping of CAN controller.

```
typedef union _tagPORT_REG
   {
                struct PORTREG_BIT bit;
                unsigned short reg;
   }PORT_REG;
   struct PORTREG_BIT
   {
       unsigned short RxBuffer: 1;
       unsigned short DataOverrun: 1;
       unsigned short TxBuffer: 1;
       unsigned short TxEnd: 1;
       unsigned short RxStatus: 1;
       unsigned short TxStatus: 1;
       unsigned short ErrorStatus: 1;
```

```
unsigned short BusStatus: 1;
unsigned short reserved: 8;
};
```

See Also

CanGetPortStatus(), and PORT_STATUS structure

3.1.3 CAN_PACKET structure define

The **CAN_PACKET** structure defines the packet format of CAN packets. It is used by the ***CanSendMsg()***, and ***CanRcvMsg()*** functions.

typedef struct _tagCAN_PACKET

{

DWORD CAN_ID;

BYTE rtr;

BYTE len;

BYTE data[8]

DWORD time;

BYTE reserved

}CAN_PACKET;

Members

CAN_ID	:CAN ID field (32-bit unsigned integer)
rtr	:CAN RTR bit.
len	:Length of data field.
data	:Data (8 bytes maximum)
time	:Reserved for future use
reserved	:Reserved byte

See Also

CanSendMsg(), and CanRcvMsg()

3.1.4 DEVICENET_PACKET structure define

The **DEVICENET_PACKET** structure defines the packet format of DeviceNet packets. It is widely used by the DeviceNet layer functions.

```
typedef struct _tagDEVICENET_PACKET
{
    BYTE Group;
    BYTE MAC_ID;
    BYTE HostMAC_ID;
    BYTE MESSAGE_ID;
    BYTE len;
    BYTE data[8];
    DWORD time;
    BYTE reserved;
    }DEVICENET_PACKET;
```

Members

Group:	Group of DeviceNet packets.
MAC_ID:	Address of destination.
HostMAC_ID:	Address of source.
MESSAGE_ID:	Message ID of DeviceNet packet.
len:	Length of data field.
data:	Data (8 bytes maximum).

See Also

SendDeviceNetPacket(), and **RcvDeviceNetPacket()**

3.2 CAN LAYER Functions

CAN-layer Card Initialization Functions

PM7841_Install(base, irq_chn, 0xd000)

Purpose	Get the version of driver
Prototype	**C/C++** *int PM7841_Install(int baseAddr, int irq_chn, int memorySpace)*
Parameters	baseAddr: Base Address of PM-7841(DIP Switch) Irq_chn: IRQ channel (Jumpper) MemorySpace: Memory Mapping Range
Return Value	A signed integer 0: Successful -1: Failed
Remarks	PM7841 is PC104(ISA) CAN interface card. It will need 32-bytes I/O space and 1K memory space.
See Also	none
Usage	**C/C++** #include “pm7841.h” int ret; ret = PM7841_Install(baseAddr, irq_ch, memorySpace);

GetDriverVersion()

Purpose	Get the version of driver
Prototype	**C/C++** *WORD GetDriverVersion(void)*
Parameters	none
Return Value	A 16-bit unsigned integer High byte is the major version Low byte is the major version
Remarks	Call this function to retrieve the version of current using driver. This function is for your program to get the version of library and dynamic-linked library.
See Also	none
Usage	C/C++ #include “pci7841.h” WORD version = GetDriverVersion(); majorVersion = version >> 8; minorVersion = version & 0x00FF;

CanOpenDriver()

Purpose	Open a specific port, and initialize driver.
Prototype	**C/C++** *int CanOpenDriver(int card, int port)***)**
Parameters	card: index of card port: index of port
Return Value	Return a handle for open port -1 if error occurs
Remarks	Call this function to open a port Under DOS, you will receive –1 if there is not enough memory. If writing program for the Windows system. It will return -1, if you want to open a port had been opened. You must use *CanCloseDriver()* to close the port after using.
See Also	CanCloseDriver()
Usage	**C/C++** *#include "pci7841.h"* *int handle = CanOpenDriver();* *CanSendMsg(handle, &msg);* *CanCloseDriver(handle);*

CanCloseDriver()

Purpose	Close an opened port, and release driver.
Prototype	**C/C++** int CanCloseDriver(int handle)
Parameters	handle: handle retrieve from *CanOpenDriver()* Port: index of port
Return Value	Return 0 if successful -1 if error occurs
Remarks	Call this function to close a port.
See Also	***CanOpenDriver()***
Usage	See usage of ***CanOpenDriver().***

CanConfigPort()

Purpose Configure properties of a port

Prototype **C/C++**

*int CanConfigPort(int handle, PORT_STRUCT *ptrStruct)*

Parameters handle: handle retrieve from *CanOpenDriver()*

ptrStruct: a pointer of *PORT_STRUCT* type

Return Value Return 0 is successful

-1 if error occurs

Remarks Configure a port that had been opened.

The properties of a CAN port such as baud rate, acceptance code, acceptance mask, operate mode. After configuration is over, the port is ready to send and receive data.

See Also 3.1.1 PORT_STRUCT structure define

Usage **C/C++**

```
#include “pci7841.h

PORT_STRUCT port_struct;

int handle = CanOpenDriver(0, 0);  // Open port 0 of card 0

port_struct.mode = 0;              // CAN2.0A (11-bit CAN id)

port_struct.accCode = 0;           // This setting of acceptance code and

port_struct.accMask = 0x7FF;       // mask enable all MAC_IDs input

port_struct.baudrate = 0;          // 125K bps

CanConfigPort(handle, &port_struct);

CanCloseDriver(handle);
```

CanDetectBaudrate()

Purpose Perform auto-detect baud rate algorithm.

Prototype **C/C++**

int CanDetectBaudrate(int handle, int miliSecs)

Parameters handle: handle retrieve from *CanOpenDriver()*

MiliSecs: timeout time (ms)

Return Value Return –1 if error occurs

Otherwise the baudrate

Value	Baudrate
0	125kbps
1	250kbps
2	500kbps
3	1Mbps

Remarks Call this function to detect the baud rate of a port.

The function performs an algorithm to detect your baud rate. It needs that there are activities on the network. It will return a –1 when detecting no activity on the network or time was exceeded.

See Also none

Usage **C/C++**

#include "pci7841.h

PORT_STRUCT port_struct;"

int handle = CanOpenDriver();

port_struct.mode = 0; // CAN2.0A (11-bit CAN id)

port_struct.accCode = 0; // This setting of acceptance code and

```
port_struct.accMask = 0x7FF; //      mask enable all
MAC_IDs input
port_struct.baudrate = CanDetectBaudrate(handle,
1000):

CanConfigPort(handle, &port_struct);

CanCloseDriver(handle);
```

Visual Basic(Windows 95/98/NT)

CanRead()

Purpose	Direct read the register of PCI/PMC-7841(G).
Prototype	**C/C++** *BYTE CanRead(int handle, int offset)*
Parameters	handle: handle retrieve from *CanOpenDriver()* offset: offset of register
Return Value	Return data read from port.
Remarks	Direct read the register of PCI/PMC-7841(G).
See Also	***Can****Write()*
Usage	none

CanWrite()

Purpose	Direct write the register of PCI/PMC-7841(G).
Prototype	**C/C++** void CanWrite(int handle, int offset, BYTE data)
Parameters	handle: handle retrieve from *CanOpenDriver()* Offset: offset of register data: data write to the port
Return Value	none
Remarks	Call this function to directly write a register of PCI/PMC-7841(G)
See Also	*CanRead()*
Usage	none

CanEnableReceive()

Purpose	Enable receiving of a CAN port.
Prototype	**C/C++** *void CanEnableReceive(int handle);*
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	none
Remarks	Call this function to enable receiving. Any packet on the network that can induce a interrupt on your computer. If that packet can pass your acceptance code and acceptance mask setting. So if your program doesn't want to be disturbed. You can call *CanDisableReceive()* to disable receive and *CanEnableReceive() to enable receives.*
See Also	*CanDisableReceive()*
Usage	none

CanDisableReceive()

Purpose	Disable receive of a CAN port.
Prototype	**C/C++**
	void CanDisableReceive(int handle);
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	none
Remarks	Please refer the *CanEnableReceive()*
See Also	*CanEnableReceive()*
Usage	none

CanSendMsg()

Purpose Send can packet to a port

Prototype **C/C++**

int CanSendMsg(int handle, CAN_PACKET *packet);

Parameters handle: handle retrieve from *CanOpenDriver()*

packet: *CAN_PACKET* data

Return Value Return 0 if successful

-1 if error occurs

Remarks Send a message to an opened CAN port.

Actually, this function copies the data to the sending queue. Error occurs when the port has not been opened yet or the packet is a NULL pointer. You can use the Error and Event handling functions to handle the exceptions.

See Also CanRcvMsg()

Usage **C/C++**

```
#include “pci7841.h
PORT_STRUCT port_struct;
CAN_PACKET sndPacket, rcvPacket;
int handle = CanOpenDriver(0, 0); // open the port 0 of card 0
CanConfigPort(handle, &port_struct);
CanSendMsg(handle, &sndPacket);
if(CanRcvMsg(handle, &rcvPacket) == 0)
{
}
CanCloseDriver(handle);
```

CanRcvMsg()

Purpose Receive a can packet from a port

Prototype **C/C++**

*int CanSendMsg(int handle, CAN_PACKET *packet);*

Parameters handle: handle retrieve from *CanOpenDriver()*

packet: *CAN_PACKET* data

Return Value Return 0 is successful

-1 if error occurs

Remarks Receive a message from an opened CAN port.

There is only a 64-byte FIFO. It can store from 3 to 21 packets. So there are memory buffer under driver. When data comes, the driver would move it from card to memory. It starts after your port configuration is done. This function copies the buffer to your application. So if your program has the critical section to process the data on the network. We suggest that you can call the *CanClearBuffer()* to clear the buffer first. Error would be happened most under the following conditions:

1. You want to access a port that has not be opened.
2. Your packet is a NULL pointer.
3. The receive buffer is empty.

You can use the Status handling functions to handle the exceptions.

See Also ***CanSendMsg()***

Usage See the *CanSendMsg()*

CanClearOverrun()

Purpose	**Clear data overrun status**
Prototype	**C/C++** void CanClearOverrun(int handle)
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	**none**
Remarks	**Clear the data overrun status** Sometimes if your system has heavy load, and the bus is busy, the data overrun would be signaled. A Data Overrun signals that data is lost, possibly causing inconsistencies in the system.
See Also	CanRcvMsg()
Usage	**C/C++** *#include "pci7841.h* *int handle = CanOpenDriver(0, 0); // open the port 0 of card 0* *….* *CanClearOverrun(handle);* *CanCloseDriver(handle);*

CanClearRxBuffer()

Purpose Clear data in the receive buffer

Prototype **C/C++**

```
void  CanClearRxBuffer(int handle)
```

Parameters handle: handle retrieve from ***CanOpenDriver()***

Return Value none

Remarks Clear the data in the receive buffer

There are 2-type of buffer defined in the driver. First one is the FIFO in the card; the second one is the memory space inside the driver. Both of them would be cleared after using this function.

See Also CanRcvMsg()

Usage **C/C++**

```
#include “pci7841.h
int handle = CanOpenDriver(0, 0);  //  open the port 0 of card 0
….
CanClearRxBuffer(handle);
CanCloseDriver(handle);
```

CanClearTxBuffer()

Purpose Clear Transmit Buffer

Prototype **C/C++**

void CanClearTxBuffer(int handle)

Parameters handle: handle retrieve from ***CanOpenDriver()***

Return Value none

Remarks Clear the data in the transmit buffer.

Under a busy DeviceNet Network, your transmit request may not be done due to the busy in the network. The hardware will send it automatically when bus is free. The un-send message would be stored in the memory of the driver. The sequence of outgoing message is the FIFO. According this algorithm, if your program need to send an emergency data, you can clear the transmit buffer and send it again.

See Also CanRcvMsg()

Usage **C/C++**

```
#include “pci7841.h
int handle = CanOpenDriver(0, 0); // open the port 0 of card 0
….
CanClearTxBuffer(handle);
CanCloseDriver(handle);
```

Purpose **Get the Error Code**

Prototype **C/C++**

BYTE CanGetErrorCode(int handle)

Parameters handle: handle retrieve from ***CanOpenDriver()***

Return Value **error code**

Return error code is an 8-bit data

Bit	Symbol	Name	Value	Function
7	ERRC1	Error Code 1		
6	ERRC0	Error Code 0		
5	DIR	Direction	1	Rx error occurred during reception
			0	Tx error occurred during transmission
4	SEG4	Segment 4		
3	SEG3	Segment 3		
2	SEG2	Segment 2		
1	SEG1	Segment 1		
0	SEG0	Segment 0		

Bit interpretation of ERRC1 and ERRC2

Bit ERRC1	Bit ERRC2	Function
0	0	bit error
0	1	form error
1	0	stuff error
1	1	other type of error

Bit interpretation of SEG4 to SEG 0

SEG4	SEG3	SEG2	SEG1	SEG0	Function
0	0	0	1	1	start of frame
0	0	0	1	0	ID.28 to ID.21
0	0	1	1	0	ID.20 to ID.18
0	0	1	0	0	bit SRTR
0	0	1	0	1	bit IDE
0	0	1	1	1	ID.17 to ID.13
0	1	1	1	1	ID.12 to ID.5
0	1	1	1	0	ID.4 to ID.0
0	1	1	0	0	RTR bit
0	1	1	0	1	reserved bit 1
0	1	0	0	1	reserved bit 0
0	1	0	1	1	Data length code
0	1	0	1	0	Data field
0	1	0	0	0	CRC sequence
1	1	0	0	0	CRC delimiter
1	1	0	0	1	acknowledge slot
1	1	0	1	0	end of frame
1	0	0	1	0	intermission
1	0	0	0	1	active error flag
1	0	1	1	0	passive error flag
1	0	0	1	1	tolerate dominant bits
1	0	1	1	1	error delimiter
1	1	1	0	0	overload flag

Remarks

Get the information about the type and location of errors on the bus.

When bus error occurs, if your program installed the call-back function or error-handling event. The error-bit position would be captured into the card. The value would be fixed in the card until your program read it back.

See Also

CanGetErrorWarningLimit(),

CanSetErrorWarningLimit()

Usage

C/C++

```
#include “pci7841.h

int handle = CanOpenDriver(0, 0);    //    open    the
port 0 of card 0

….

BYTE data = CanGetErrorCode();

CanCloseDriver(handle);
```

CanSetErrorWarningLimit()

Purpose Set the Error Warning Limit

Prototype **C/C++**

void CanSetErrorWarningLimit(int handle, BYTE value)

Parameters handle: handle retrieve from ***CanOpenDriver()***

value: Error Warning Limit

Return Value none

Remarks Sets the error warning limit if your program has installed the error warning event or call-back function. The error warning will be signaled after the value of error counter passing the limit you set.

See Also *CanGetErrorWarningLimit()*

Usage **C/C++**

```
#include “pci7841.h
int handle = CanOpenDriver(0, 0); // open the port 0 of card 0
….
CanSetErrorWarning(handle, 96);
CanCloseDriver(handle);
```

CanGetErrorWarningLimit()

Purpose	Get the Error Warning Limit
Prototype	**C/C++** BYTE CanGetErrorWarningLimit(int handle) Visual Basic(Windows 95/98/NT)
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	0-255 (Error warning limit value)
Remarks	Get the error warning limit
See Also	*CanSetErrorWarningLimit()*
Usage	**C/C++** #include “pci7841.h int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 …. BYTE limit = CanClearOverrun(handle); CanCloseDriver(handle);

CanGetRxErrorCount()

Purpose	Get the current value of the receive error counter
Prototype	C/C++ BYTE CanGetRxErrorCount(int handle)
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	value
Remarks	This function reflects the current of the receive error counter. After hardware reset, the value returned would be initialized to 0. If a bus-off event occurs, the returned value would be 0.
See Also	CanRcvMsg()
Usage	**C/C++** #include “pci7841.h int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 …. BYTE error_count = CanGetRxErrorCount(); *CanCloseDriver(handle);*

CanGetTxErrorCount()

Purpose	Get the current value of the transmit error counter
Prototype	**C/C++** BYTE CanGetTxErrorCount(int handle)
Parameters	handle: handle retrieve from *CanOpenDriver()*
Return Value	value
Remarks	This function reflects the current of the transmit error counter. After hardware reset, the value would set to 127. A bus-off event occurs when the value reaches 255. You can call the *CanSetTxErrorCount()* to set the value from 0 to 254 to clear the bus-off event.
See Also	CanRcvMsg()
Usage	**C/C++** #include “pci7841.h int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 …. BYTE error_count = CanGetTxErrorCount(handle); *CanCloseDriver(handle);*

CanSetTxErrorCount()

Purpose	Set the current value of the transmit error counter
Prototype	**C/C++** void CanSetTxErrorCount(int handle, BYTE value)
Parameters	handle: handle retrieve from ***CanOpenDriver()*** value: a byte value
Return Value	None
Remarks	This function set the current of the transmit error counter. Please see the remark of *CanGetTxErrorCount()*.
See Also	CanRcvMsg()
Usage	**C/C++** #include “pci7841.h int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 …. CanSetTxErrorCount(handle, 0); *CanCloseDriver(handle);*

CanGetPortStatus()

Purpose	Get Port Status
Prototype	**C/C++** int CanGetPortStatus(int handle, PORT_STATUS *PortStatus)
Parameters	handle: handle retrieve from ***CanOpenDriver()*** PortStatus: Pointer of PORT_STATUS structure
Return Value	No Error: 0 Error: -1
Remarks	Get Port Status(See the structure define for detailed description)
See Also	
Usage	**C/C++** #include “pci7841.h PORT_STATUS port_status; int handle = CanOpenDriver(0, 0);// open the port 0 of card 0 CanGetPortStatus(&port_status); CanClearOverrun(); CanCloseDriver(handle);

CanGetLedStatus()

Purpose Get the LED status of cPCI-7841 and PM-7841

Prototype **C/C++**

BYTE CanGetLedStatus (int card, int index);

Parameters card: card number

Index: index of LED

Return Value **status of Led**

Value	Function
0	Led Off
1	Led On

Remarks Get the status of Led

This function supports the cPCI-7841 and PM-7841.

See Also *CanSetLEDStatus()*

Usage **C/C++**

```
#include “pci7841.h
int handle = CanOpenDriver(0, 0); // open the port 0 of card 0
….
BYTE flag = CanGetLedStatus(0, 0);;
CanCloseDriver(handle);
```

CanSetLedStatus()

Purpose Set the Led Status of cPCI-7841

Prototype C/C++

void CanSetLedStatus(int card, int index, int flashMode);

Parameters card: card number

index: index of Led

flashMode:

Value	Function
0	Led Off
1	Led On

Return Value none

Remarks Set Led status of cPCI-7841 and PM-7841

This function supports the cPCI-7841 and PM-7841

See Also CanRcvMsg()

Usage C/C++

#include “pci7841.h

int handle = CanOpenDriver(0, 0); // open the port 0 of card 0

….

CanSetLedStatus(0, 0, 2); // Set Led to flash

CanCloseDriver(handle);

CanGetRcvCnt()

Purpose	Get the how many message in the FIFO
Prototype	**C/C++** int _stdcall CanGetRcvCnt(int handle)
Parameters	handle : handle retrieve from ***CanOpenDriver()***
Return Value	value indicates the left unread messages in the FIFO.
Remarks	Get the unread message count in the FIFO. Because the interrupt would be very busy while CAN bus is busy. There is possibility to lost the event in Windows system. A way to solve to this problem is to call this function at free time while program running. You also can call this function to make sure that receiving FIFO is empty.
See Also	*CanGetReceiveEvent()*
Usage	**C/C++** #include “pci7841.h int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 ….. int count = CanGetRcvCnt(handle);.

Error and Event Handling Functions

When the exception occurs, your program may need to take some algorithm to recover the problem. The following functions are operation-system depended functions. You should care about the restriction in the operation-system.

- DOS Environment

CanInstallCallBack()

Purpose Install callback function of event under DOS environment

Prototype **C/C++ (DOS)**

void far*CanInstallCallBack(int handle, int index, void (far* proc)());

Parameters handle: handle retrieve from *CanOpenDriver()*

Index: event type

Index	Type
2	Error Warning
3	Data Overrun
4	Wake Up
5	Error Passive
6	Arbitration Lost
7	Bus Error

void (far *proc)() : Call-back function

The suggest prototype of the call-back function is like void (far ErrorWarning)();

Return Value Previous call back function (NULL when there is no Call back installed)

Remarks Install the call-back function for event handling

In normal state, all hardware interrupt of cPCI/PCI-7841 wouldn't be set except receive and

transmit interrupt. After calling the *CanInstallCallBack()*, the corresponding interrupt would be activated. The interrupt occurs when the event happened. It will not be disabled until using *CanRemoveCallBack()* or a hardware reset.

Actually, the call-back function is a part of ISR. You need to care about the DOS reentrance problem, and returns as soon as possible to preventing the lost of data.

See Also CanRemoveCallBack()

Usage **C/C++(DOS)**

```
#include “pci7841.h

void (far ErrorWarning)();

int handle = CanOpenDriver(0, 0);

//      open the port 0 of card 0

…

//      Installs the ErrorWarning handling event and stores the previous one.

void (far *backup) = CanInstallCallBack(0, 2, ErrorWarning);

CanRemoveCallBack(0, 2, NULL); //      Remove the call-back function

CanCloseDriver(handle);
```

CanRemoveCallBack()

Purpose

Remove the callback function of event under DOS environment

Prototype

C/C++(DOS)

int CanRemoveCallBack(int handle, int index, void (far* proc)());

Parameters

handle: handle retrieve from *CanOpenDriver()*

Index: event type

Index	Type
2	Error Warning
3	Data Overrun
4	Wake Up
5	Error Passive
6	Arbitration Lost
7	Bus Error

void (far *proc)() : Previous call-back function

Return Value

Return 0 is successful

-1 if error occurs

Remarks

Install the call-back function for event handling

In normal state, all hardware interrupt of cPCI/PCI/PMC-7841(G) wouldn't be set except receive and transmit interrupt. After calling the *CanInstallCallBack()*, the corresponding interrupt would be activated. The interrupt occurs when the event happened. It will not be disabled until using *CanRemoveCallBack()* or a hardware reset.

Actually, the call-back function is a part of ISR. You need to care about the DOS reentrance problem, and returns as soon as possible to preventing the lost of data.

See Also CanRemoveCallBack()

Usage **C/C++ (DOS)**

```
#include “pci7841.h

void (far ErrorWarning)();

int handle = CanOpenDriver(0, 0); // open the port 0 of card 0

…

// Installs the ErrorWarning handling event and stores the previous one.

void (far *backup) = CanInstallCallBack(0, 2, ErrorWarning);

CanRemoveCallBack(0, 2, NULL); // Remove the call-back function

CanCloseDriver(handle);
```

CanGetReceiveEvent()

Purpose	Install the event under Windows 95/98/NT system
Prototype	**C/C++ (Windows 95/98/NT)** void CanGetReceiveEvent(int handle, HANDLE *hevent);
Parameters	handle: handle retrieve from *CanOpenDriver()* hevent: HANDLE point for receive event
Return Value	none
Remarks	Retrieve receive notify event Under the Windows 95/98/NT environment, your program can wait the input message by waiting an event. You can refer to following program to use this function. But the CAN system is a heavy-load system. Under full speed(of course, it depends on your system), the hardware receives the message faster than the event occurs. Under this condition, the event could be combined by OS. So the total count of event may be less than actually receive. You can call the *CanGetRcvCnt()* to retrieve the unread message in the driver's FIFO.
See Also	***CanGetRcvCnt()***
Usage	**C/C++ (Windows 95/98/NT)** #include "pci7841.h HANDLE recvEvent0; int handle = CanOpenDriver(0, 0); // open the port 0 of card 0 int count1;

```
CanGetReceiveEvent(handle,    rcvEvent0);
if(WaitForSingleObject(rcvEvent0, INFINITE)
                              == WAIT_OBJECT_0)
{
        //      You need not to call ResetEvent()…..
        err=CanRcvMsg(handle,&rcvMsg[0]
                              [rcvPatterns[0]]);
        rcvPatterns[0]++;
}
 cout1 = CanGetRcvCnt(handle[0]);
 //      To retrieve number of unread
 //      in the FIFO
```

CanInstallEvent()

Purpose Install the event under Windows 95/98/NT system

Prototype C/C++ (Windows 95/98/NT)

int CanInstallEvent(int handle, int index, HANDLE hEvent);

Parameters handle: handle retrieve from *CanOpenDriver()*

Index: event type

Index	Type
2	Error Warning
3	Data Overrun
4	Wake Up
5	Error Passive
6	Arbitration Lost
7	Bus Error

hEvent: HANDLE created from *CreateEvent()(Win32 SDK)*

Return Value Return 0 is successful

-1 if error occurs

Remarks Install the notify event

Unlike the Dos environment, there is only one error handling function under Windows 95/98/NT environment. First you need to create an event object, and send it to the DLL. The DLL would make a registry in the kernel and pass it to the VxD(SYS in NT system). You can't release the event object you created, because it was attached to the VxD. The VxD would release the event object *when you installed another event. One way to disable the event handling is that you install another event which handle is NULL (ex: CanInstallEvent(handle, index, NULL)). And you can create a thread to handle the error event.*

See Also **CanRemoveCallBack(),CanInstallCallBack()**

Usage **C/C++ (Windows 95/98/NT)**

```
#include “pci7841.h
int handle = CanOpenDriver(0, 0);
//      open the port 0 of card 0
…
//      Installs the ErrorWarning handling event and stores the previous one.
HANDLE hEvent = CreateEvent(NULL, FALSE, TRUE, “ErrorWarning”);
CanInstallEvent(0, 2, hEvent);
//..create a thread ….

        Thread function
        WaitForSingleObject(hEvent, INFINITE);
        ResetEvent(hEvent);
//      Event handling
```

Warranty Policy

Thank you for choosing ADLINK. To understand your rights and enjoy all the after-sales services we offer, please read the following carefully:

1. Before using ADLINK's products please read the user manual and follow the instructions exactly.
2. When sending in damaged products for repair, please attach an RMA application form.
3. All ADLINK products come with a two-year guarantee, repaired free of charge.
   - The warranty period starts from the product's shipment date from ADLINK's factory.
   - Peripherals and third-party products not manufactured by ADLINK will be covered by the original manufacturers' warranty.
   - End users requiring maintenance services should contact their local dealers. Local warranty conditions will depend on local dealers.
4. This warranty will not cover repair costs due to:
   a. Damage caused by not following instructions.
   b. Damage caused by carelessness on the users' part during product transportation.
   c. Damage caused by fire, earthquakes, floods, lightening, pollution, other acts of God, and/or incorrect usage of voltage transformers.
   d. Damage caused by unsuitable storage environments (i.e. high temperatures, high humidity, or volatile chemicals.
   e. Damage caused by leakage of battery fluid.
   f. Damage from improper repair by unauthorized technicians.
   g. Products with altered and/or damaged serial numbers.
   h. Other categories not protected under our guarantees.
5. Customers are responsible for shipping costs to transport damaged products to our company or sales office.
6. To ensure the speed and quality of product repair, please download a RMA application form from our company website: www.adlinktech.com. Damaged products with attached RMA forms receive priority.

For further questions, please contact our FAE staff.

ADLINK: service@adlinktech.com