Getting Service from ADLINK

Customer Satisfaction is top priority for ADLINK Technology Inc. Please contact us should you require any service or assistance.

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 this completed service form for prompt and satisfactory service.

<table>
<thead>
<tr>
<th>Company Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company/Organization</td>
</tr>
<tr>
<td>Contact Person</td>
</tr>
<tr>
<td>E-mail Address</td>
</tr>
<tr>
<td>Address</td>
</tr>
<tr>
<td>Country</td>
</tr>
<tr>
<td>TEL</td>
</tr>
<tr>
<td>FAX:</td>
</tr>
<tr>
<td>Web Site</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Model</td>
</tr>
<tr>
<td>Environment</td>
</tr>
<tr>
<td>OS:</td>
</tr>
<tr>
<td>M/B:</td>
</tr>
<tr>
<td>CPU:</td>
</tr>
<tr>
<td>Chipset:</td>
</tr>
<tr>
<td>BIOS:</td>
</tr>
</tbody>
</table>

Please give a detailed description of the problem(s):
# Table of Contents

## Table of Contents

Table of Contents......................................................................................... i

## List of Tables

List of Tables........................................................................................ iii

## List of Figures

List of Figures........................................................................................ iv

## 1 Introduction

1.1 PCI/cPCI/PM-7841 Features ......................................................... 1
1.2 Applications ............................................................................. 3
1.3 Specifications............................................................................ 4

## 2 Installation

2.1 Before Installing the PCI/cPCI/PM-7841................................. 7
2.2 Installing PCI-7841 ................................................................. 7
2.3 Installing cPCI-7841................................................................. 9
2.4 Installing PM-7841 ................................................................. 10
2.5 Jumper and DIP Switch Description .................................... 12
2.6 Base Address Setting ............................................................ 12
2.7 IRQ Level Setting ................................................................. 14

## 3 Function Reference

3.1 Functions Table ..................................................................... 15
   PORT_STRUCT structure define ........................................... 16
   CAN_PACKET structure define ......................................... 18
   Members ............................................................................. 18
   DEVICENET_PACKET structure define ......................... 19
   Members ............................................................................. 19
3.2 CAN LAYER Functions......................................................... 20
   CAN-layer Card Initialization Functions ......................... 20
   PM7841_Install(base, irq_chn, 0xd000) ......................... 20
   GetDriverVersion() ........................................................... 20
   CanOpenDriver() ................................................................ 21
   CanCloseDriver() .............................................................. 21
   CanConfigPort() ............................................................... 22
   CanDetectBaudrate() ......................................................... 23
   CanRead() .......................................................................... 24
   CanWrite() .......................................................................... 24
   CAN-layer I/O Functions ................................................... 24
Table of Contents

CanEnableReceive() ..................................................... 24
CanDisableReceive() .................................................... 25
CanSendMsg() .............................................................. 25
CanRcvMsg() ................................................................ 26
CAN-layer Status Functions ......................................... 27
CanClearOverrun() ....................................................... 27
CanClearRxBuffer() ...................................................... 28
CanClearTxBuffer() ....................................................... 28
CanGetErrorCode() ...................................................... 29
CanSetErrorWarningLimit() ........................................... 31
CanGetErrorWarningLimit() .......................................... 32
CanGetRxErrorCode() ................................................. 32
CanGetTxErrorCode() .................................................. 33
CanGetPortStatus() ...................................................... 34
CanGetLedStatus() ....................................................... 35
CanSetLedStatus() ....................................................... 35
CanGetRcvCnt() ........................................................... 36
Error and Event Handling Functions ............................. 37
CanInstallCallBack() ..................................................... 37
CanRemoveCallBack() ................................................. 38
CanCloseDriver(handle); .............................................. 40
CanGetReceiveEvent() ................................................. 40
CanInstallEvent() .......................................................... 41

Warranty Policy ..................................................................... 43
List of Tables

Table 1-1: PCI-7841 Specifications ............................................... 4
Table 1-2: cPCI-7841 Specifications .............................................. 4
Table 1-3: PM-7841 Specifications ............................................... 5
List of Figures

Figure 2-1: Default Base Address Configuration ........................ 13
Figure 2-2: IRQ Settings ............................................................ 14
1 Introduction

The PCI/cPCI/PM-7841 is a Controller Area Network (CAN) interface card used for industrial PC with PCI, Compact-PCI, and PC104 bus. It supports dual ports CAN's interface that can run independently or bridged at the same time. The built-in CAN controller provides bus arbitration and error detection with auto correction and re-transmission function. The PCI cards are plug and play therefore it is not necessary to set any jumper for matching the PC environment.

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

The PCI/cPCI/PM-7841 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-7841 Features

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

- Two independent CAN network operation
- Bridge function supports
- 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
- Powerful master interface for CANopen, DeviceNet and SDS application layer protocol
- Up to 1Mbps programmable transfer rate
- Supports standard DeviceNet data rates 125, 250 and 500 Kbps
- 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 function supports
- 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
- Powerful master interface for CANopen, DeviceNet and SDS application layer protocol
- Up to 1Mbps programmable transfer rate
- Supports standard DeviceNet data rates 125, 250 and 500 Kbps
- 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 function supports
- 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
- Powerful master interface for CANopen, DeviceNet and SDS application layer protocol
- Up to 1Mbps programmable transfer rate
- Supports standard DeviceNet data rates 125, 250 and 500 Kbps
- DIP-Switch for base address configuration
- Software Programmable Memory-Mapped Address
- PC-104 industry form factor
- DOS library and examples included
1.2 Applications

- Industry automation
- Industry process monitoring and control
- Manufacture automation
- Product testing
## 1.3 Specifications

### PCI-7841 Specification Table

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

**Table 1-1: PCI-7841 Specifications**

### cPCI-7841 Specification Table

<table>
<thead>
<tr>
<th>Ports</th>
<th>2 CAN channels (V2.0 A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN Controller</td>
<td>SJA1000</td>
</tr>
<tr>
<td>CAN Transceiver</td>
<td>82c250</td>
</tr>
<tr>
<td>Signal Support</td>
<td>CAN_H, CAN_L</td>
</tr>
<tr>
<td>Isolation Voltage</td>
<td>2500 Vrms</td>
</tr>
<tr>
<td>Connectors</td>
<td>Dual male connectors</td>
</tr>
<tr>
<td>Operation Temperature</td>
<td>0 ~ 60°C</td>
</tr>
<tr>
<td>Storage Temperature</td>
<td>-20°C ~ 80°C</td>
</tr>
<tr>
<td>Humidity</td>
<td>5% ~ 95% non-condensing</td>
</tr>
<tr>
<td>IRQ Level</td>
<td>Set by Plug and Play BIOS</td>
</tr>
<tr>
<td>I/O port address</td>
<td>Set by Plug and Play BIOS</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>400mA @5VDC (Typical)</td>
</tr>
<tr>
<td>(without external devices)</td>
<td>900mA @5VDC (Maximum)</td>
</tr>
</tbody>
</table>

**Table 1-2: cPCI-7841 Specifications**
<table>
<thead>
<tr>
<th>Size</th>
<th>132(L)mm x 98(H)mm</th>
</tr>
</thead>
</table>

**PM-7841 Specification Table**

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

**Table 1-2: cPCI-7841 Specifications**

**Table 1-3: PM-7841 Specifications**
2 Installation

This chapter describes how to install the PCI/cPCI/PM-7841. At first, the contents in the package and unpacking information that you should be careful are described.

2.1 Before Installing the PCI/cPCI/PM-7841

Your PCI/cPCI/PM-7841 card contains sensitive electronic components that can be easily damaged by static electricity.

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

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

After opening the card module carton, exact 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-7841.

2.2 Installing PCI-7841

What you have:

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

► PCI-7841 Dual Port PCI Isolated CAN Interface Card
► ADLINK CD-ROM

If any of these items is missing or damaged, contact the dealer from whom you purchased the product. Save the shipping materials and carton in case you want to ship or store the product in the future.
**PCI-7841 Layout:**

![PCIE-7841 Layout](image)

**Terminator Configuration**

A 120 Ohm terminal resistor is installed for each port, while JP1 enables the terminal resistor for port0 and JP2 enables the terminal resistor for port1.

**Connector Pin Definition**

P3 and P4 are CAN connectors; pin definition is as follows:

![DIP-9 Connector](image)
2.3 Installing cPCI-7841

What you have

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

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

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

cPCI-7841 Layout

Terminator Configuration

A 120 Ohm terminal resistor is installed for each port, while JP1 enables the terminal resistor for port0 and JP2 enables the terminal resistor for port 1
Connector Pin Definition

J1 and J2 are CAN connectors; pin definition is as follows:

![Combicon-Style Connector](image)

2.4 Installing PM-7841

What you have

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

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

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

A 120 Ohm terminal resistor is installed for each port, while JP1 enables the terminal resistor for port0 and JP2 enables the terminal resistor for port 1.
Connector Pin Define

J1 and J2 are CAN connectors; pin definition is as follows:

2.5 Jumper and DIP Switch Description

You can configure the output of each channel and base address by setting jumpers and DIP switches on the PM-7841. The card's jumpers and switches are preset at the factory. Under normal circumstances, you should not need to change the jumper settings.

A jumper switch is closed (sometimes referred to as "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.

2.6 Base Address Setting

The PM-7841 requires 16 consecutive address locations in 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 an 5 position DIP switch SW1 (refer to Table 2.1). The address settings for I/O port from Hex 200 to Hex 3F0 is described in Table 2.2 below. The default base address of your PM-7841 is set to hex 200 in the factory (see Figure below).
SW1: Base Address = 0x200

Figure 2-1: Default Base Address Configuration

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

(*): default setting ON : 0
X: don't care OFF : 1

Note: A4,…, A9 correspond to PC-104(ISA) bus address lines.
2.7 IRQ Level Setting

A hardware interrupt can be triggered by the external Interrupt signal which is from JP3 and JP4.

The jumper setting is specified as below:

**Note:** Be aware that there is no other add-on cards sharing the same interrupt level in the system.

Interrupt Default Setting = IRQ15

![Figure 2-2: IRQ Settings](image-url)
3 Function Reference

The cPCI/PCI-7841 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
- DeviceNet layer functions
- Send and Receive packet functions
- Connection establish and release functions
- DeviceNet object class functions

The particular functions associated with each function are presented in the next page.

3.1 Functions Table

CAN layer functions

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM-7841 Initial</td>
<td>PM7841_Install()</td>
</tr>
<tr>
<td></td>
<td>GetDriverVersion()</td>
</tr>
<tr>
<td></td>
<td>CanOpenDriver()</td>
</tr>
<tr>
<td></td>
<td>CanCloseDriver()</td>
</tr>
<tr>
<td></td>
<td>CanConfigPort()</td>
</tr>
<tr>
<td></td>
<td>CanDetectBaudrate()</td>
</tr>
<tr>
<td></td>
<td>_7841_Read()</td>
</tr>
<tr>
<td></td>
<td>_7841_Write()</td>
</tr>
<tr>
<td></td>
<td>CanEnableReceive()</td>
</tr>
<tr>
<td></td>
<td>CanDisableReceive()</td>
</tr>
<tr>
<td></td>
<td>CanSendMsg()</td>
</tr>
<tr>
<td></td>
<td>CanRcvMsg()</td>
</tr>
<tr>
<td></td>
<td>CanGetRcvCnt()</td>
</tr>
</tbody>
</table>
Note: only for compact PCI and PC-104 version.

**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; // Reserved
    BYTE sjw, sam; // Reserved
} PORT_STRUCT;
```
Members

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

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


accMask: Acceptance Mask for CAN controller.

baudrate: Baud rate setting for the CAN controller.

<table>
<thead>
<tr>
<th>Value</th>
<th>Baudrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>125 Kbps</td>
</tr>
<tr>
<td>1</td>
<td>250 Kbps</td>
</tr>
<tr>
<td>2</td>
<td>500 Kbps</td>
</tr>
<tr>
<td>3</td>
<td>1M bps</td>
</tr>
</tbody>
</table>

CanPortConfig(), CanGetPortStatus(), and PORT_STATUS structure

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;

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

CAN_PACKET structure define
The CAN_PACKET structure defines the packet format of CAN packet. 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()
DEVICENET_PACKET structure define

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

```c
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 packet.
- **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++
Prototype int PM7841_Install(int baseAddr, int irq_chn, int memorySpace)
Parameters baseAddr: Base Address of PM-7841 (DIP Switch)
irq_chn: IRQ channel (Jumper)
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++

```c
#include "pm7841.h"
int ret;
ret = PM7841_Install(baseAddr, irq_chn, memorySpace);
```

GetDriverVersion()
Purpose Get the version of driver
Prototype C/C++
Prototype 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 operation system, 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. And 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)
```
### Function Reference

**CanClosePort()**

| 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()**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Configure properties of a port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>C/C++</td>
</tr>
</tbody>
</table>
| 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 | PORT_STRUCT structure define |
| Usage | C/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++

```c
int CanDetectBaudrate(int handle, int miliSecs)
```

**Parameters**
- `handle`: handle retrieve from CanOpenDriver()
- `MiliSecs`: timeout time (ms)

**Return Value**
Return –1 if error occurs

**Others**
is the baudrate

<table>
<thead>
<tr>
<th>Value</th>
<th>Baudrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>125 Kbps</td>
</tr>
<tr>
<td>1</td>
<td>250 Kbps</td>
</tr>
<tr>
<td>2</td>
<td>500 Kbps</td>
</tr>
<tr>
<td>3</td>
<td>1 Mbps</td>
</tr>
</tbody>
</table>

**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. And it will return a –1 when detecting no activity on the network or time was exceeded.

**See Also**
none

**Usage**
C/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-7841.
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-7841.
See Also CanWrite()
Usage none

CanWrite()
Purpose Direct write the register of PCI-7841.
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-7841
See Also CanRead()
Usage none

CAN-layer I/O Functions

CanEnableReceive()
Purpose Enable receive 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 receive. Any packet on the network that can induce an 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 CanEnableReceive(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 is successful
CanSendMsg()

Purpose: Receive a can packet from a port

Prototype: C/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 are only 64-bytes FIFO under hardware. 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()  

CAN-layer Status Functions  

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 signalled. A Data Overrun signals, that data are 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
```c
...
CanClearOverrun(handle);
CanCloseDriver(handle);
```

### CanClearRxBuffer()

**Purpose**
Clear data in the receive buffer

**Prototype**
C/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++

```c
#include "pci7841.h
```

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

```
...
CanClearRxBuffer(handle);
CanCloseDriver(handle);
```

### CanClearTxBuffer()

**Purpose**
Clear Transmit Buffer

**Prototype**
C/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 FIRST-IN-FIRST-OUT. 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++

```c
#include "pci7841.h"

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

...
CanClearTxBuffer(handle);
CanCloseDriver(handle);
```

### CanGetErrorCode()

**Purpose**
Get the Error Code

**Prototype**
C/C++

```c
BYTE CanGetErrorCode(int handle)
```

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

**Return Value**
error code

Return error code is an 8-bit data

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Name</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>ERRC1</td>
<td>Error Code 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ERRC0</td>
<td>Error Code 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DIR</td>
<td>Direction</td>
<td>1</td>
<td>Rx error occurred during reception</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Tx error occurred during transmission</td>
</tr>
<tr>
<td>4</td>
<td>SEG4</td>
<td>Segment 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SEG3</td>
<td>Segment 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SEG2</td>
<td>Segment 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SEG1</td>
<td>Segment 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SEG0</td>
<td>Segment 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Bit interpretation of ERRC1 and ERRC2

<table>
<thead>
<tr>
<th>Bit ERRC1</th>
<th>Bit ERRC2</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>bit error</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>form error</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>stuff error</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>other type of error</td>
</tr>
</tbody>
</table>

### Bit interpretation of SEG4 to SEG 0

<table>
<thead>
<tr>
<th>SEG4</th>
<th>SEG3</th>
<th>SEG2</th>
<th>SEG1</th>
<th>SEG0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>start of frame</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>ID.28 to ID.21</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ID.20 to ID.18</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>bit SRTR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>bit IDE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ID.17 to ID.13</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ID.12 to ID.5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ID.4 to ID.0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>RTR bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>reserved bit 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>reserved bit 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Data length code</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Data field</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CRC sequence</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CRC delimiter</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>acknowledge slot</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>end of frame</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>intermission</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>active error flag</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>passive error flag</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>tolerate dominant bits</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>error delimiter</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>overload flag</td>
</tr>
</tbody>
</table>
Remarks
Get the information about the type and location of errors on the bus.
When a 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
Set 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++

```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++

```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++

```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 happened, 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++
```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 happened, 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++

```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)
```
CanSetTxErrorCount()

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
CanRcvMsg()

Usage
#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

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Led Off</td>
</tr>
<tr>
<td>1</td>
<td>Led On</td>
</tr>
</tbody>
</table>

**Remarks**
Get the status of Led
This function supports the cPCI-7841 and PM-7841.

**Usage**
C/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**:

<table>
<thead>
<tr>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Led Off</td>
</tr>
<tr>
<td>1</td>
<td>Led On</td>
</tr>
</tbody>
</table>
CanSetLedStatus()

<table>
<thead>
<tr>
<th></th>
<th>Led Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Led Off</td>
</tr>
<tr>
<td>1</td>
<td>Led On</td>
</tr>
</tbody>
</table>

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
#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++
```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
#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)

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

Parameters

- `handle`: handle retrieve from CanOpenDriver()
- `Index`: event type

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Error Warning</td>
</tr>
<tr>
<td>3</td>
<td>Data Overrun</td>
</tr>
<tr>
<td>4</td>
<td>Wake Up</td>
</tr>
<tr>
<td>5</td>
<td>Error Passive</td>
</tr>
<tr>
<td>6</td>
<td>Arbitration Lost</td>
</tr>
<tr>
<td>7</td>
<td>Bus Error</td>
</tr>
</tbody>
</table>

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

The suggested 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 CanRemoveCall-Back() or a hardware reset.

Actually, the call-back function is a part of ISR. You need to care about the DOS reen- trance 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 CanOpen- Driver()

Index : event type

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Error Warning</td>
</tr>
<tr>
<td>3</td>
<td>Data Overrun</td>
</tr>
<tr>
<td>4</td>
<td>Wake Up</td>
</tr>
<tr>
<td>5</td>
<td>Error Passive</td>
</tr>
<tr>
<td>6</td>
<td>Arbitration Lost</td>
</tr>
</tbody>
</table>
Function Reference

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-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);
Windows 95/98 Environment

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()
Heven : HANDLE point for receive event
Return Value none
Remarks Retrieve receive notify event
Under 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 the 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 CanOpen-
Driver()

Index : event type

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Error Warning</td>
</tr>
<tr>
<td>3</td>
<td>Data Overrun</td>
</tr>
<tr>
<td>4</td>
<td>Wake Up</td>
</tr>
<tr>
<td>5</td>
<td>Error Passive</td>
</tr>
<tr>
<td>6</td>
<td>Arbitration Lost</td>
</tr>
<tr>
<td>7</td>
<td>Bus Error</td>
</tr>
</tbody>
</table>

HEvent : HANDLE created from CreateEvent()(Win32 SDK)

Return Value Return 0 is successful -1 if error occurs

Remarks 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)

```c
#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
```
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. When sending in damaged products for repair, please attach an RMA application form which can be downloaded from: http://rma.adlinktech.com/policy/.

2. All ADLINK products come with a limited two-year warranty, one year for products bought in China:
   - The warranty period starts on the day the product is shipped from ADLINK’s factory.
   - Peripherals and third-party products not manufactured by ADLINK will be covered by the original manufacturers' warranty.
   - For products containing storage devices (hard drives, flash cards, etc.), please back up your data before sending them for repair. ADLINK is not responsible for any loss of data.
   - Please ensure the use of properly licensed software with our systems. ADLINK does not condone the use of pirated software and will not service systems using such software. ADLINK will not be held legally responsible for products shipped with unlicensed software installed by the user.
   - For general repairs, please do not include peripheral accessories. If peripherals need to be included, be certain to specify which items you sent on the RMA Request & Confirmation Form. ADLINK is not responsible for items not listed on the RMA Request & Confirmation Form.
3. Our repair service is not covered by ADLINK's guarantee in the following situations:

- Damage caused by not following instructions in the User's Manual.
- Damage caused by carelessness on the user's part during product transportation.
- Damage caused by fire, earthquakes, floods, lightening, pollution, other acts of God, and/or incorrect usage of voltage transformers.
- Damage caused by unsuitable storage environments (i.e. high temperatures, high humidity, or volatile chemicals).
- Damage caused by leakage of battery fluid during or after change of batteries by customer/user.
- Damage from improper repair by unauthorized ADLINK technicians.
- Products with altered and/or damaged serial numbers are not entitled to our service.
- This warranty is not transferable or extendible.
- Other categories not protected under our warranty.

4. Customers are responsible for shipping costs to transport damaged products to our company or sales office.

5. To ensure the speed and quality of product repair, please download an RMA application form from our company website: http://rma.adlinktech.com/policy. Damaged products with attached RMA forms receive priority.

If you have any further questions, please email our FAE staff: service@adlinktech.com.