NuCOM â

PCI-7841/cPCI-7841/PM-7841

Dual-Port Isolated

CAN Interface Card

User's Guide

Recycled Paper

Manual Rev. 2.20: June. 6, 2001

Part No : 50-11109-100

Trademarks

PCI-7841, cPCI-7841, and PM-7841 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.

Table of Contents

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

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 ?? 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

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

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 Installation 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

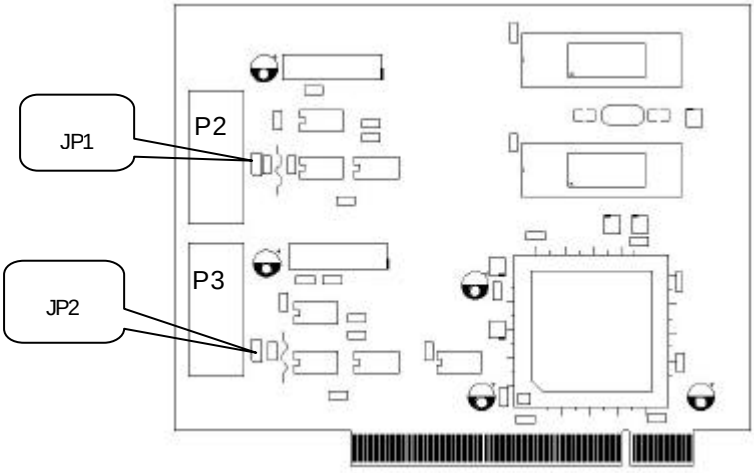
What do you have

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

- PCI-7841 Dual Port PCI Isolated CAN Interface Card
- ADLINK All-xxxxx 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

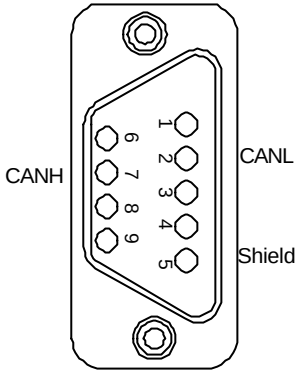


Terminator Configuration

A 120 Ω 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

The P3 and P4 are CAN connector, the below picture is their pin define



DIP-9 Connector

2.3 Installing cPCI-7841

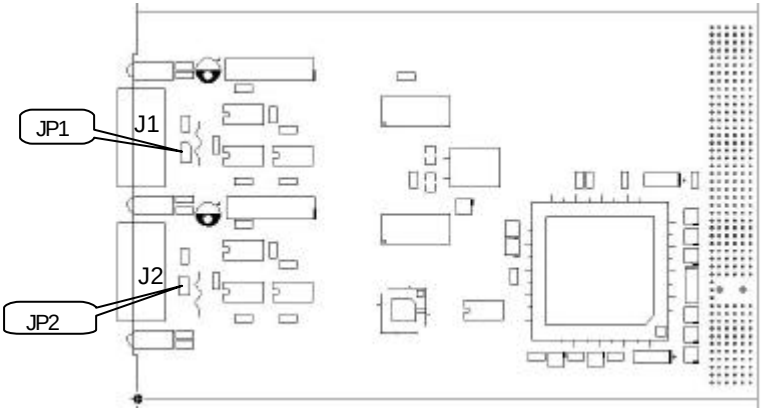
What do you have

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

- cPCI-7841 Dual Port Compact-PCI Isolated CAN Interface Card
- ADLINK All-xxxx 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

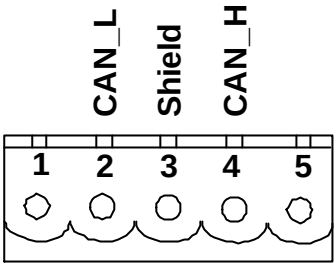


Terminator Configuration

A 120 Ω 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

The J1 and J2 are CAN Connector, the below picture is their pin define



Combicon-Style Connector

2.4 Installing PM-7841

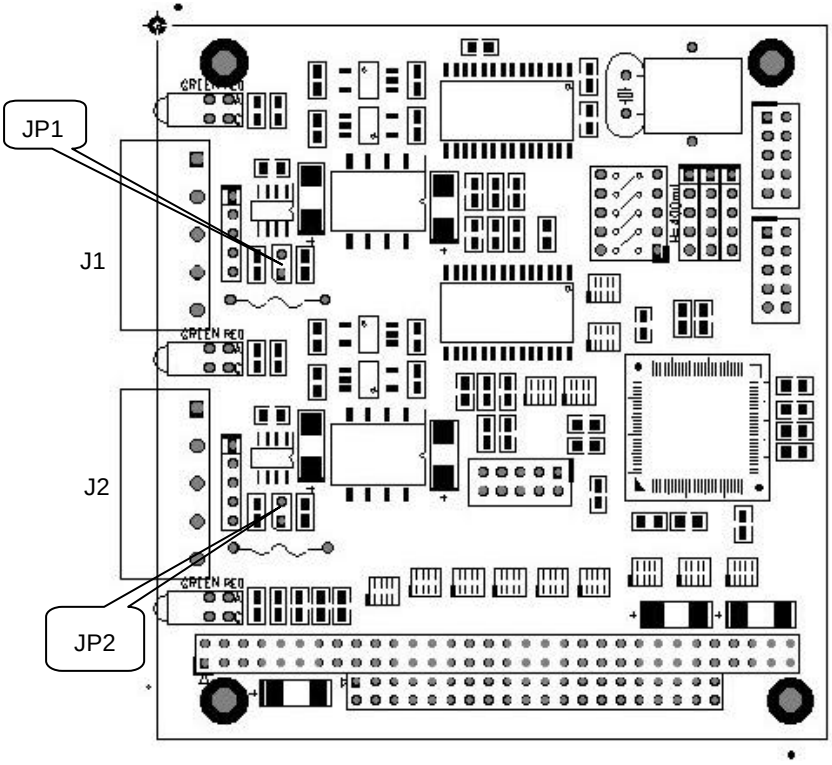
What do you have

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

- PM-7841 Dual Port PC-104 Isolated CAN Interface Card
- ADLINK All-xxxxx 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.

PM-7841 Layout



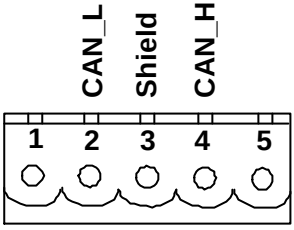
Terminator Configuration

A 120 Ω 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

The J1 and J2 are CAN Connector, the below picture is their pin define.



2.4 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.

2.5 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

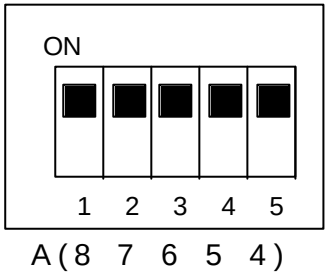


Figure Default Base Address Configuration

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.

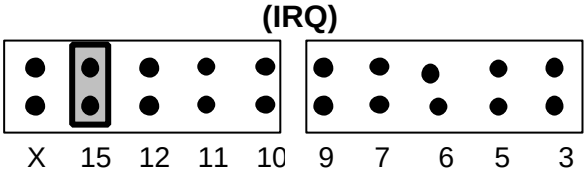
2.6 IRQ Level Setting

A hardware interrupt can be triggered by the external Interrupt signal which is from JP3 ad 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



IRQ Setting

3

Function Reference

The cPCI/PCI-7841 functions are organize 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 next page.

3.1 Functions Table

CAN layer functions		
Function Type	**Function Name**	**Page**
PM-7841 Initial	PM7841_Install()	25
	GetDriverVersion()	25
	CanOpenDriver()	27
	CanCloseDriver()	28
	CanConfigPort()	29
	CanDetectBaudrate()	30
	_7841_Read()	32
	_7841_Write()	32
	CanEnableReceive()	33
	CanDisableReceive()	34
	CanSendMsg()	35
	CanRcvMsg()	36
	CanGetRcvCnt()	51
	CanClearOverrun()	37
	CanClearRxBuffer()	38
	CanClearTxBuffer()	39
	CanGetErrorCode()	40
	CanGetErrorWarningLimit()	40
	CanSetErrorWarningLimit()	43
	CanGetRxErrorCount()	45
	CanGetTxErrorCount()	45
	CanSetTxErrorCount()	47
	CanGetPortStatus()	48
	CanGetLedStatus()[1]	49
	CanSetLedStatus()[1]	50

Error and Event handling functions		
Operation System	**Function Name**	**Page**
DOS	CanInstallCallBack()	52
	CanRemoveCallBack()	54
Windows 95/98/NT	CanInstallEvent()	58

Note : only for compact PCI and PC-104 version.

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.

baudrate: Baud rate setting for the CAN controller.

Value	Baudrate
0	125 Kbps
1	250 Kbps
2	500 Kbps
3	1M Kbps
4	User-Defined

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

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 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()

3.1.4 DEVICENET_PACKET structure define

The **DEVICENET_PACKET** structure defines the packet format of DeviceNet packet. 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 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++**

int PM7841_Install(int baseAddr, int irq_chn, int memorySpace)

Visual Basic(Windows 95/98/NT)

Parameters baseAddr: Base Address of PM-7841(DIP Switch)

Irq_chn: IRQ channel (Jumpper)

MemorySpace: Memory Mapping Range

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 “pm7841.h”

WORD version = GetDriverVersion();
majorVersion = version >> 8;
minorVersion = version & 0x00FF;
```

Visual Basic(Windows 95/98/NT)

GetDriverVersion()

Purpose	Get the version of driver
Prototype	**C/C++** *WORD GetDriverVersion(void)* **Visual Basic(Windows 95/98/NT)**
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; **Visual Basic(Windows 95/98/NT)**

CanOpenDriver()

Purpose Open a specific port, and initialize driver.

Prototype **C/C++**

int CanOpenDriver(int card, int port)

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

CanCloseDriver()

Purpose	Close an opened port, and release driver.
Prototype	**C/C++** int CanCloseDriver(int handle) **Visual Basic(Windows 95/98/NT)**
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)*

Visual Basic(Windows 95/98/NT)

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 **Can**ConfigPort()

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);
```

Visual Basic(Windows 95/98/NT)

CanDetectBaudrate()

Purpose Perform auto-detect baud rate algorithm.

Prototype **C/C++**

int CanDetectBaudrate(int handle, int miliSecs)

Visual Basic(Windows 95/98/NT)

Parameters handle : handle retrieve from *CanOpenDriver()*

MiliSecs : timeout time(ms)

Return Value Return –1 if error occurs

Others is the baudrate

Value	Baudrate
0	125 Kbps
1	250 Kbps
2	500 Kbps
3	1M Kbps

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++**

#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)* **Visual Basic(Windows 95/98/NT)**
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	***Can****Write()*
Usage	none

CanWrite()

Purpose	Direct write the register of PCI-7841.
Prototype	**C/C++** void CanWrite(int handle, int offset, BYTE data) **Visual Basic(Windows 95/98/NT)**
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

CanEnableReceive()

Purpose	Enable receive of a CAN port.
Prototype	**C/C++** *void CanEnableReceive(int handle);* **Visual Basic(Windows 95/98/NT)**
Parameters	handle : handle retrieve from *CanOpenDriver()*
Return Value	none
Remarks	Call this function to enable receive. 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	Enable receive of a CAN port.
Prototype	**C/C++** void CanEnableReceive(int handle); **Visual Basic(Windows 95/98/NT)**
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);

Visual Basic(Windows 95/98/NT)

Parameters handle : handle retrieve from *CanOpenDriver()*

Packet : *CAN_PACKET* data

Return Value Return 0 is 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);
```

Visual Basic(Windows 95/98/NT)

CanRcvMsg()

Purpose Receive a can packet from a port

Prototype **C/C++**

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

Visual Basic(Windows 95/98/NT)

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 fom 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) **Visual Basic(Windows 95/98/NT)**
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* *….* *CanClearOverrun(handle);* *CanCloseDriver(handle);* **Visual Basic(Windows 95/98/NT)**

CanClearRxBuffer()

Purpose Clear data in the receive buffer

Prototype **C/C++**

void CanClearRxBuffer(int handle)

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

CanClearTxBuffer()

Purpose Clear Transmit Buffer

Prototype **C/C++**

void CanClearTxBuffer(int handle)

Visual Basic(Windows 95/98/NT)

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++**

```
#include “pci7841.h

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

….

CanClearTxBuffer(handle);

CanCloseDriver(handle);
```

Visual Basic(Windows 95/98/NT)

CanGetErrorCode()

Purpose **Get the Error Code**

Prototype **C/C++**

BYTE CanGetErrorCode(int handle)

Visual Basic(Windows 95/98/NT)

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 ransmission
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);

Visual Basic(Windows 95/98/NT)

CanSetErrorWarningLimit()

Purpose Set the Error Warning Limit

Prototype **C/C++**

void CanSetErrorWarningLimit(int handle, BYTE value)

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

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 none

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);
```

Visual Basic(Windows 95/98/NT)

CanGetRxErrorCount()

Purpose Get the current value of the receive error counter

Prototype C/C++

BYTE CanGetRxErrorCount(int handle)

Visual Basic(Windows 95/98/NT)

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++**

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

Visual Basic(Windows 95/98/NT)

CanGetTxErrorCount()

Purpose Get the current value of the transmit error counter

Prototype **C/C++**

BYTE CanGetTxErrorCount(int handle)

Visual Basic(Windows 95/98/NT)

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++**

```
#include “pci7841.h

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

….

BYTE error_count = CanGetRxErrorCount(handle);

CanCloseDriver(handle);
```

Visual Basic(Windows 95/98/NT)

CanSetTxErrorCount()

Purpose Set the current value of the transmit error counter

Prototype **C/C++**

void CanSetTxErrorCount(int handle, BYTE value)

Visual Basic(Windows 95/98/NT)

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

value : a byte value

Return Value value

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
….
CanSetRxErrorCount(handle, 0);
CanCloseDriver(handle);
```

Visual Basic(Windows 95/98/NT)

CanGetPortStatus()

Purpose Get Port Status

Prototype **C/C++**

int CanGetPortStatus(int handle, PORT_STATUS *PortStatus)

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

CanGetLedStatus()

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

Prototype **C/C++**

BYTE CanGetLedStatus (int card, int index);

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

CanSetLedStatus()

Purpose Set the Led Status of cPCI-7841

Prototype C/C++

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

Visual Basic(Windows 95/98/NT)

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);
```

Visual Basic(Windows 95/98/NT)

CanGetRcvCnt()

Purpose	Get the how many message in the FIFO
Prototype	**C/C++** int _stdcall CanGetRcvCnt(int handle) **Visual Basic(Windows 95/98/NT)**
Parameters	card : card number
Return Value	How many messages…
Remarks	Get the unread message count in the FIFO
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);. **Visual Basic(Windows 95/98/NT)**

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++**

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

Visual Basic(Windows 95/98/NT)

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++**

```
#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++

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

Visual Basic(Windows 95/98/NT)

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-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);
```

CanGetReceiveEvent()

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

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

void CanGetReceiveEvent(int handle, HANDLE *hevent);

Visual Basic(Windows 95/98/NT)

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;

…
```

```
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);

Visual Basic(Windows 95/98/NT)

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
```

Product Warranty/Service

ADLINK warrants that equipment furnished will be free from defects in material and workmanship for a period of one year from the date of shipment. During the warranty period, we shall, at our option, either repair or replace any product that proves to be defective under normal operation.

This warranty shall not apply to equipment that has been previously repaired or altered outside our plant in any way as to, in the judgment of the manufacturer, affect its reliability. Nor will it apply if the equipment has been used in a manner exceeding its specifications or if the serial number has been removed.

ADLINK does not assume any liability for consequential damages as a result from our product uses, and in any event our liability shall not exceed the original selling price of the equipment. The remedies provided herein are the customer's sole and exclusive remedies. In no event shall ADLINK be liable for direct, indirect, special or consequential damages whether based on contract of any other legal theory.

The equipment must be returned postage-prepaid. Package it securely and insure it. You will be charged for parts and labor if the warranty period is expired or the product is proves to be misuse, abuse or unauthorized repair or modification.