Installation and operation manual
PCIe-CAN-01 / PCIe-CAN-02

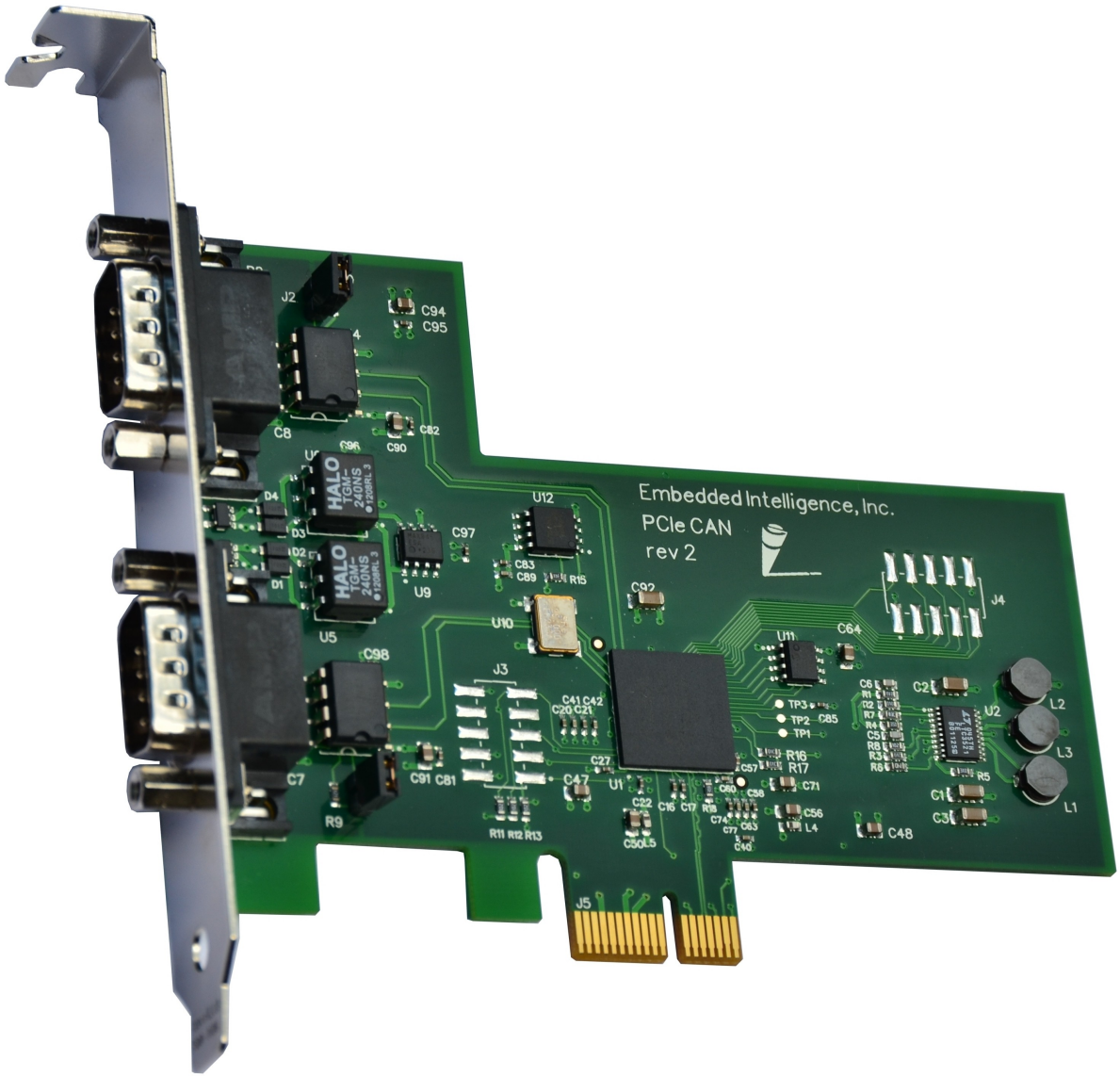


Table of Contents

Compliance information....3
FCC Compliance....3
EN 55022 Class A Warning....3
CE declaration of conformity....3
Safety note....3
Hardware description and installation....4
CAN connector pinout....5
Driver Installation....6
Windows....6
Linux....9
CAN bus wiring....10
Development libraries....13
C language....13
Functions....13
EICanOpen....13
EICanClose....14
EICanRecv....14
EICanXmit....14
EICanReadCardInfo....15
Structures....15
CanFrame....15
CanErrorInfo....16
CanCardInfo....17
Error codes....17
C++ language....19
Class Methods....19
EIcan::EIcan....19
Eican::~EIcan....19
EIcan::Open....19
EIcan::Close....19
EIcan::Recv....20
EIcan::Xmit....20
EIcan::ReadCardInfo....20

Compliance information

FCC Compliance

NOTE: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

EN 55022 Class A Warning

This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

CE declaration of conformity

Embedded Intelligence, Inc declare under our sole responsibility that the products; PCIe-CAN-01 and PCIe-CAN-02 to which this declaration relates are in conformity with the following standard(s) or other normative documents

- EU directive 2004/108/EC

May 19, 2014

Stephen Glow – President Embedded Intelligence, Inc.

Safety note

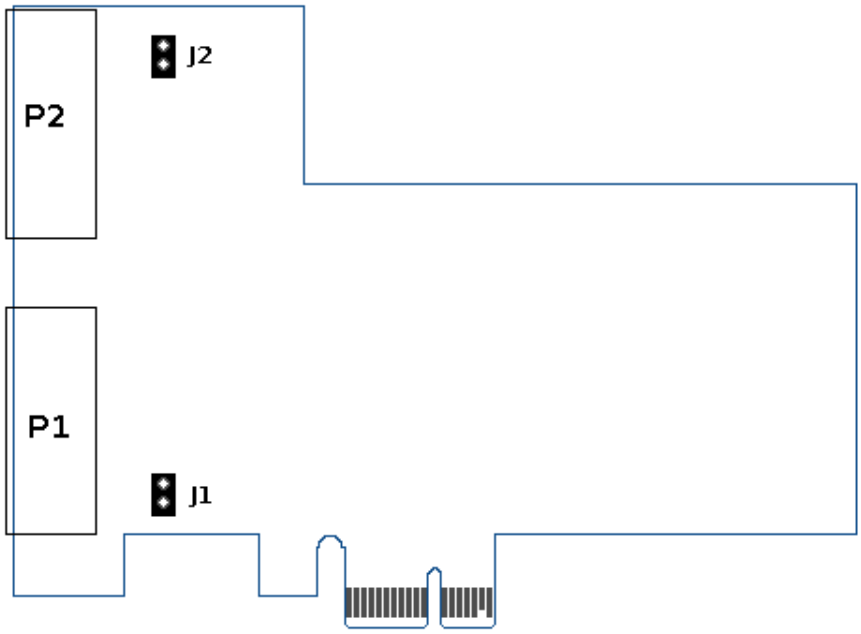
This card is intended for use in UL Listed computers or Equivalent, that have instructions detailing installation.

Hardware description and installation

The PCIe-CAN-02 is a dual channel CAN interface card for the PCI express bus. The PCIe-CAN-01 is a single channel version sized to fit in a half hight PCI express chassis. Full sized brackets are also provided for the PCIe-CAN-01, so it can be used in either a full height, or half height PCI express chassis.

On the dual channel card, the bottom CAN connector (i.e. the one located closest to the motherboard when the card is installed) is port 1. The upper connector is CAN port 2.

The card includes 120 Ohm termination resistors on board. The jumper adjacent to each CAN port connector is used to enable these termination resistors. When the jumper is in place, the CAN bus is locally terminated. When the jumper is removed the on board termination resistor is not used. Proper termination of the CAN bus is very important for the proper functioning of the network. For more details, see the section later in the document on proper CAN bus wiring.



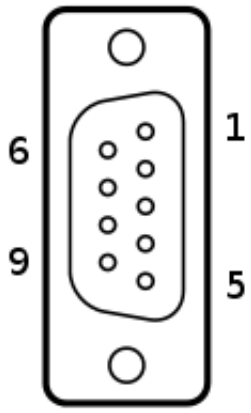
The card uses a single lane PCI express connector to interface with the host computer. The card can be installed in any PCI express slot on the host PC, both single lane slots and multi-lane slots will work equally well.

To install the card in the host PC, power down the PC and insert the card into any free PCI express slot. We recommend that a screw be used to secure the bracket to the host chassis.

CAN connector pinout

The CAN interface connector used on the card is a male DB-9 connector using standard CAN bus wiring:

Pin	Signal
1	No connection
2	CAN Low
3	CAN Ground
4	No connection
5	No connection
6	CAN Ground
7	CAN High
8	No connection
9	No connection



Driver Installation

Windows

The windows driver for the PCIe-CAN card supports Windows XP and later version of the operating system on both 32-bit and 64-bit systems.

To install the driver, first download the most recent driver file from the Embedded Intelligence web site (http://embeddedintelligence.com/download.html). The driver is distributed as a zip file and must be extracted before it can be installed. Extract the file into a folder on your desktop or other location where it can be easily found.

To install the driver, restart the PC with the CAN card installed. You must have administrator privileges to update drivers in windows, to install the driver log onto the PC using an account which has these privileges.

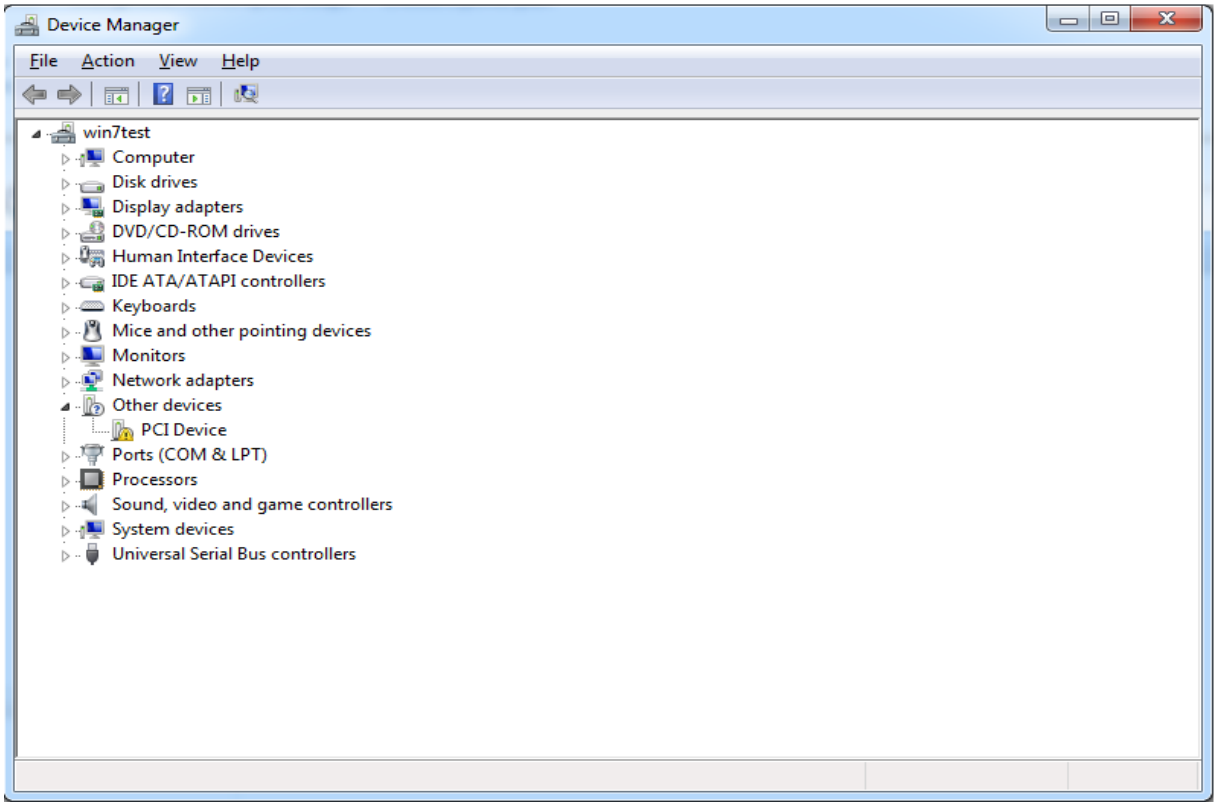
When the PC is first started with the CAN card installed, it may pop up a 'new hardware found' wizard. The driver can be installed through this wizard by selecting the folder to which the driver files were extracted. This wizard does not always show up for all version of windows, so we will detail a more reliable way to install the driver using the device manager. This method can also be used to upgrade an existing driver.

There are several ways to start the device manager program on windows. One way is to press the 'Start' button at the bottom of the screen, and enter the following command in the text field at the bottom of the startup list:

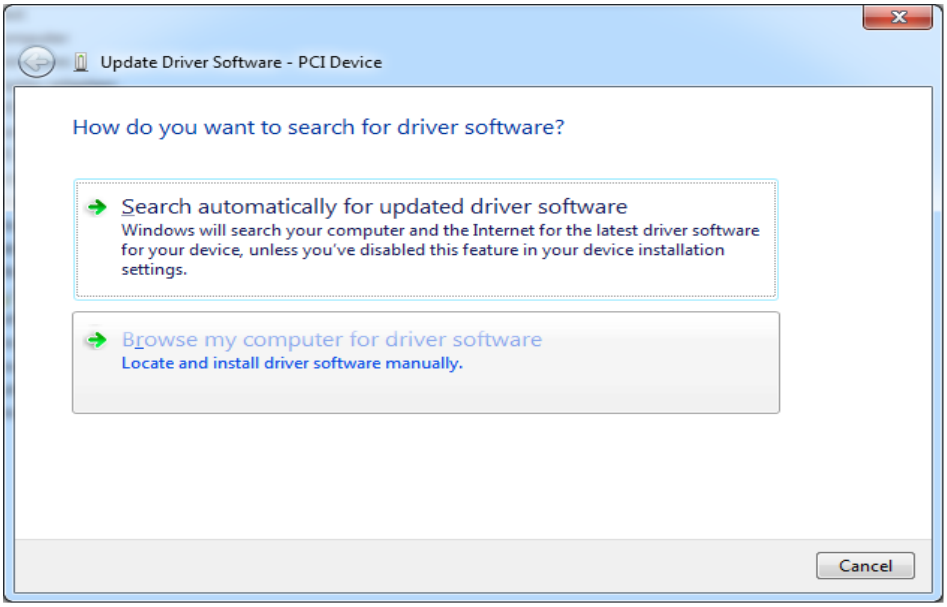
```
mmc devmgmt.msc
```

Alternately, the device manager can be found in the control panel listed under the 'Hardware and Sound' category.

The device manager gives a list of all the hardware devices available in the system. For a new install, the CAN card will show up under “other devices” listed as “PCI Device”. When upgrading the driver, the card will show up under the “CAN cards” category.

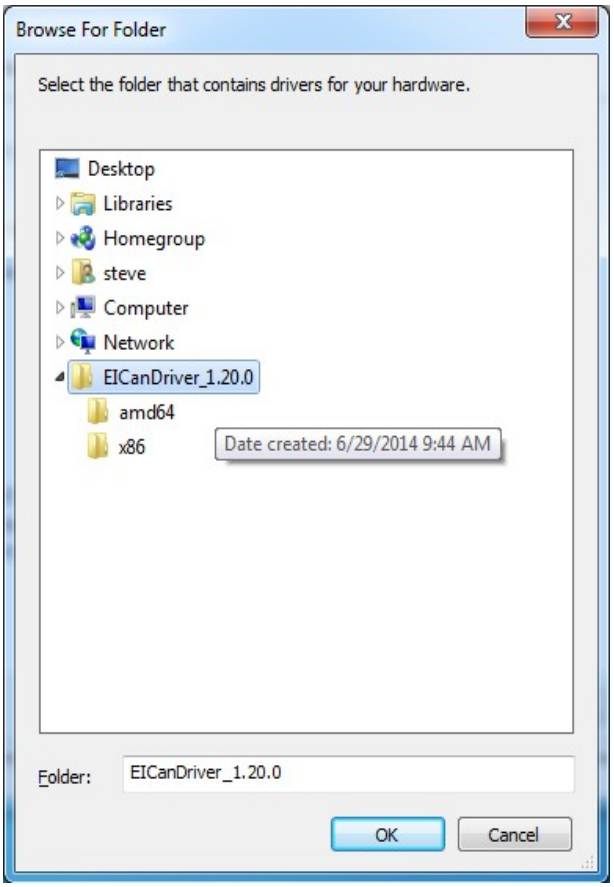


Right click on the CAN card device, and select the option to “Update driver software”. This should open a dialog box like the following:

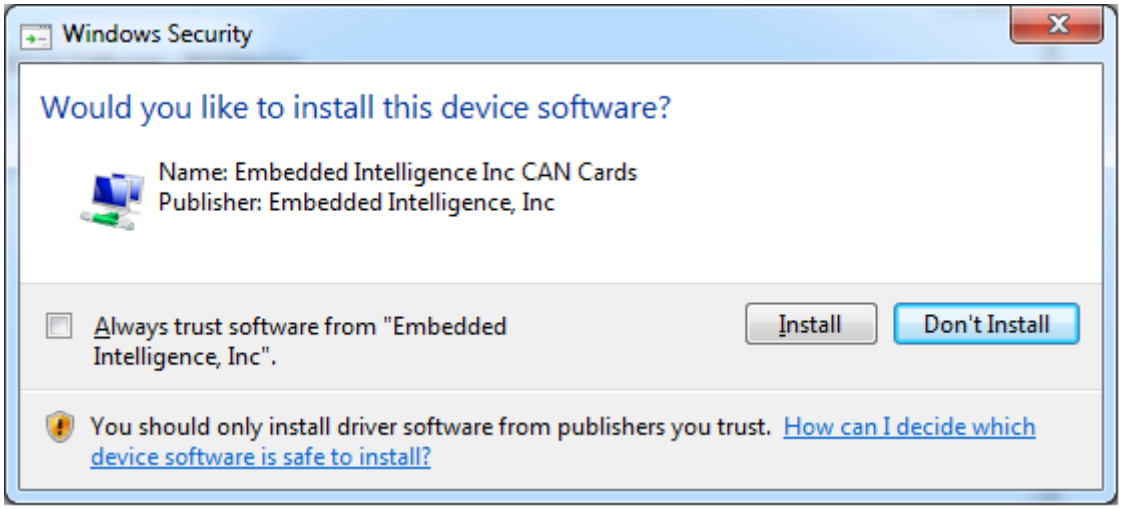


Choose “Browse my computer for driver software”.

Locate the folder containing the driver software which was previously downloaded and extracted.



Clicking “OK” should start the driver installation. Windows will then confirm whether you want to install this driver. To install the driver, select “Install”.



Linux

The Linux driver is distributed in source code format for easier installation on a variety of different Linux distributions and kernel versions.

To install the driver, first download the driver files from the downloads section of the Embedded Intelligence web site:

http://embeddedintelligence.com/download.html

To decompress the archive file, use the following command (substituting the proper archive file name if necessary):

```
tar xzf eican-1.00.0.tgz
```

This will create a new directory containing the driver source files. Enter that directory and build the driver using the following command:

```
make
```

This should compile the driver. On success, the driver file (named eican.ko) will be created in the working directory.

To insert the driver into the running kernel, execute the following command (which requires super user privileges):

sudo insmod eican.ko

To install the driver in the appropriate kernel directory so it can be automatically loaded at startup, use the following command (which also requires super user privileges):

make install

When the driver is properly installed and detects a CAN card in the system, it will create device files named /dev/eican00, /dev/eican01, etc. There will be one such file created for each CAN port, so single channel cards will create one file, dual channel cards will create two files, etc.

Normally, the device files will be created with root ownership. On Linux distributions that support the udev device manager this can be changed by creating a udev rule file. The file should be named

```
10-cancard.rules
```

and placed in the udev rules directory in the system. For Debian based distributions this folder is

```
/etc/udev/rules.d/
```

The rules file should contain the following line:

```
SUBSYSTEM=="cancard", OWNER="someuser", GROUP="somegroup", MODE="0666"
```

Change the someuser and somegroup strings to valid user and group names in the system.

Please see the Linux udev documentation for more information on writing udev rules.

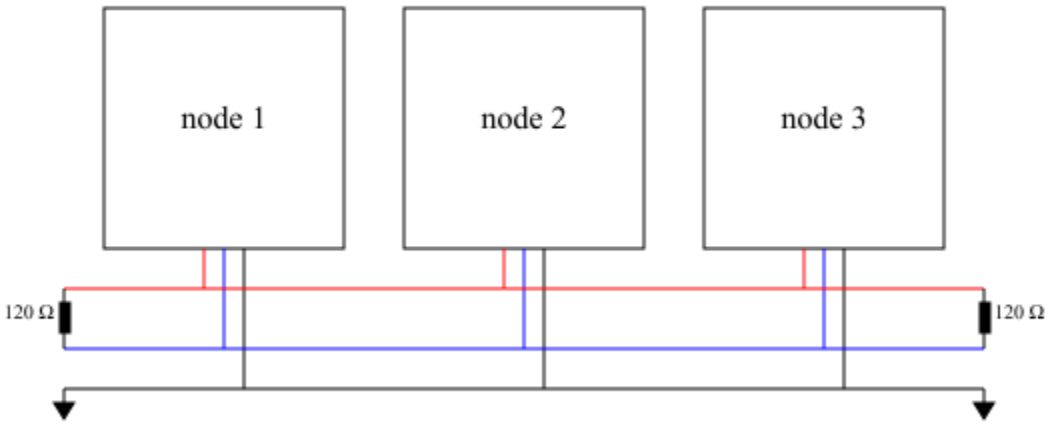
There are many different Linux distributions in the wild. Please don't hesitate to contact Embedded Intelligence support if you encounter difficulties building or installing the Linux driver on your system.

CAN bus wiring

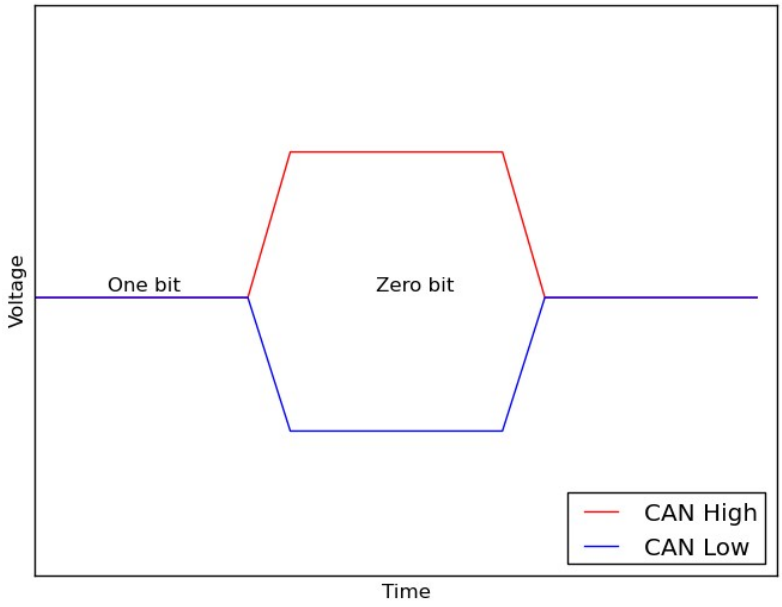
Laying out a system that uses the CAN bus requires a certain amount of care. When properly designed, a CAN network will provide a highly reliable communications link. If insufficient care is taken when designing the network layout, significant error rates can result.

CAN is a shared bus style network. All nodes on the network are directly connected to the same set of bus wires. The CAN bus uses two signal wires named CAN high and CAN low. Networks will often function with only those two signal wires shared between the nodes on the network, but it's wise to also connect a common ground wire between all the nodes on the network. This avoids large ground shifts between nodes which could exceed the common mode voltage limitations of the CAN transceivers.

When a CAN network is designed, it should take the form of a single bus line with nodes hanging off of it. The length of the stub from the main bus to the CAN node should be as small as possible for best results. Termination resistors should be placed at both ends of the main CAN bus with values that match the impedance of the network cable, 120 Ohm is the value typically used. If the PCIe-CAN card is being used at one end of the CAN network, then it's internal termination resistor may be used by inserting the jumper next to the CAN connector. If the card is not at the end of the bus, then it's internal termination resistor should be disabled by removing the jumper.



Messages are transmitted on the CAN bus as a series of bits. 0 bits (called dominate bits in CAN terminology) are signaled by applying a differential voltage to the CAN high and low bus lines. 1 bits (called recessive bits in CAN) are signaled by not driving the bus and allowing the terminating resistors to reduce the differential voltage to zero.



When a node on the CAN network needs to transmit a message, it first waits for the bus to be idle, then starts transmitting it's message after a brief delay. It's often the case that multiple nodes will attempt to transmit a message at exactly the same time. As long as all nodes transmit the same bit value at the same time, there will be no conflict. As soon as one node transmits a zero (dominate) bit while another node transmit a one (recessive) bit, the dominate bit will be the value that is actually signaled on the bus. This is a natural result of the way bits are signaled on the CAN bus. Dominate bits are signaled by nodes actively driving the bus voltage, recessive bits are signaled by nodes releasing control the bus lines. Each transmitting node also monitors the actual state of the bus, and if it detects a dominate bit value at a time that it attempted to transmit a recessive bit value, then it immediately stops transmitting it's message. This mechanism is called message arbitration, and is used to prioritize traffic on the CAN network.

In order for message arbitration to work, all transmitting nodes must also monitor the CAN bus lines to determine if the bits that they are sending are actually prevailing on the bus. This basic requirement of CAN is the reason that CAN networks are limited to a maximum data rate of 1 Megabit / second, and that the maximum length of the network is limited.

When a CAN node transmits a bit, that bit value needs to propagate through that node's local CAN transceiver, along the entire length of the network through the transceiver of the farthest node on the network, and all the way back before the end of the bit time. In fact, the transmitting node typically samples the actual bus value at around 88% of the bit time, so this round trip delay needs to happen within 88% of the bit period. This restricts both the maximum bit frequency of the CAN network, and the maximum length of the bus.

The CAN in Automation group (http://www.can-cia.org/) recommends the following maximum network lengths and stub lengths for different CAN bit rates. Stub lengths are the distance from the main bus line to the individual CAN node.

Bit rate	Max bus length	Max stub length	Max accumulated stub length
1 Mbit/sec	25m	1.5m	7.5m
800 kbit/sec	50m	2.5m	12.5m
500 kbit/sec	100m	5.5m	27.5m
250 kbit/sec	250m	11m	55m
125 kbit/sec	500m	22m	110m
50 kbit/sec	1000m	55m	275m
20 kbit/sec	2500m	137.5m	687.5m

In addition to limiting bus length, bus termination is also very important. The bus termination resistors perform two important functions in the CAN network; they speed up the transition from dominate to recessive bit values, and they minimize reflected waves caused by impedance mismatches at the end of the bus. CAN transceivers are designed to be used with 120 Ohm termination resistors on each end of the cable, and thus for best results the CAN network cable should have a characteristic impedance of 120 Ohms.

CAN is a very reliable network technology when properly implemented. For best results, the following guidelines are recommended when laying out a CAN bus:

- Use a single main bus line with nodes hanging off it via short stubs. Resist the temptation to layout a CAN bus in a star or tree style configuration.
- Pick a network bit rate that's appropriate for the length of your CAN bus using the table above. In fact, use the next lower bit rate if possible. Lower bit rates provide a lot of extra tolerance to questionable termination or line layout.
- Don't forget to terminate both ends of the CAN network! This is probably the single most common cause of network errors in CAN.
- Use cable designed for CAN with 120 Ohm characteristic impedance. CAT-5 has a characteristic impedance of 100 Ohms which is not ideal for a CAN system.

Development libraries

Software interface libraries are provided in several different programming languages. These libraries are designed to be very small, simple and easy to use.

To install the programming libraries, simply download the EICAN_API.zip file from the Embedded Intelligence web site. This zip file contains folders for every supported operating system. Each OS folder contains additional folders for each programming language supported.

C language

The C language interface is supported for all operating systems. The source code of the library consists of a single header file named eican.h, and a single c language file named eican.c. The directory also contains a very simple test program and the necessary project files to build the example.

The C language interface consists of a very small set of functions for opening/closing a port on the CAN card, and reading/writing CAN messages. The functions are all designed to be thread safe, so it's safe to send CAN messages from multiple threads of a multi-threaded program, or to read CAN messages from one thread while writing from another.

Functions

```
void *EICanOpen( int port, int baud, int *error );
```

This function is used to open the CAN port. Before any other calls can be made which access this CAN port, it must be successfully opened using this function.

port This parameter identifies the CAN port to open. CAN card ports are numbered from 0, so systems with a single dual channel card would have CAN ports numbered 0 and 1.

baud The bit rate at which to communicate on the CAN port. The value passed should be one of the #defined values in the eican header file. Options are:

EICAN_BITRATE_1000000	1 Megabit / second
EICAN_BITRATE_800000	500 Kbits / second
EICAN_BITRATE_500000	250 Kbits / second
EICAN_BITRATE_250000	250 Kbits / second
EICAN_BITRATE_125000	120 Kbits / second
EICAN_BITRATE_100000	100 Kbits / second
EICAN_BITRATE_50000	50 Kbits / second
EICAN_BITRATE_20000	20 Kbits / second

error If an error occurs during the call, an error code will be returned in the integer referenced by this pointer.

Return On success, a pointer to an internal data structure will be returned. This pointer should be passed to subsequent function calls made on this port.

On failure, a NULL pointer is returned and the error parameter is used to return an error code.

`int EICanClose( void *local );`

This function is used to close a CAN port when it is no longer needed. After closing the port, no further calls can be made which access that port unless it is opened again through another call to **`EICanOpen`**.

local	The pointer which was previously returned from the `EICanOpen` function. Once the port has been closed, this pointer is no longer valid and should not be passed to any subsequent functions.
Return	An error code is returned on failure, or zero on success.

`int EICanRecv( void *local, CanFrame *frame, int32_t timeout );`

Wait for a new message to be received on the open CAN port and return it.

local	The pointer which was previously returned from the `EICanOpen` function.
frame	A pointer to a CanFrame structure. If this function returns success, the received CAN frame will be stored in the structure referenced by this pointer.
timeout	The maximum amount of time to wait for a message before returning an error. The timeout is specified in millisecond units.
Return	An error code is returned on failure, or zero on success.

`int EICanXmit( void *local, CanFrame *frame, int32_t timeout );`

Transmit a CAN message over the open CAN port. This function adds the passed message to the cards transmit queue and returns.

local	The pointer which was previously returned from the `EICanOpen` function.
frame	A pointer to a CanFrame structure. This structure holds the CAN frame to be transmitted.
timeout	The maximum amount of time to wait for a message to be added to the transmit queue. Timeouts are specified in milliseconds.
Return	An error code is returned on failure, or zero on success.

`int EICanReadCardInfo( void *local, CanCardInfo *info );`

This function can be used to retrieve some basic information about the CAN card.

local The pointer which was previously returned from the `EICanOpen` function.

info A pointer to a CanCardInfo structure. This structure will be filled in by the function with information about the card.

Return An error code is returned on failure, or zero on success.

Structures

`CanFrame`

This structure represents a single message that can be transmitted on the CAN bus.

uint8_t	type	Identifies the type of the CAN frame. Type codes are defined in eican.h CAN_FRAME_DATA — A normal CAN data message. CAN_FRAME_REMOTE — A remote transmit request type CAN message. CAN_FRAME_ERROR — A special frame used by the card to return information about CAN bus errors. See the CanErrorInfo structure for details.
uint8_t	length	This field gives the number of bytes of data passed with the CAN message. It can range from 0 to 8 bytes.
uint32_t	id	Every CAN message has an ID value associated with it. There are two types of message IDs; standard 11-bit identifiers, and extended 29-bit identifiers. This field uses bit 29 to distinguish between standard and extended CAN ID values. When a standard CAN message is received, bits 0-10 of this field will contain the ID and bits 11-31 will be zero. When an extended CAN message is received, bits 0-28 will contain it's message ID and bit 29 will be set to identify the message as an extended format message. Bits 30-31 will be zero.
uint32_t	timestamp	This field contains a timestamp (in microseconds) for each received CAN message.
uint8_t	data[8]	This array contains the data associated with the CAN message. CAN messages can have up to 8 bytes of data associated with them.

CanErrorInfo

This structure is used to pass error information from the card. The CanErrorInfo structure is the same size as the CanFrame structure. When a CanFrame is passed from the card with a type code set to CAN_FRAME_ERROR, the CanFrame structure should be casted to a structure of this type to better identify the contents of the error.

uint8_t	type	Identifies the type of the CAN frame. For error information, this should always be CAN_FRAME_ERROR.
uint8_t	length	This field gives the number of bytes of additional error information available in the data array.
uint32_t	mask	This field contains a bit-mask identifying all the errors that are being reported by this frame. **Bit** **Description** 0 CAN bit stuffing error was detected on the bus. This means too many consecutive ones or zeros were detected on the bus. 1 CAN form error. A frame was detected on the bus in which some fixed bits were at the wrong level. 2 CAN CRC error. The CRC code of the CAN message was incorrect. This is caused by corruption of the CAN message. 3 ACK error. This occurs when a message is transmitted an no other node on the network acknowledges it. 4 Bit 0 error. A 0 bit was transmitted, but a 1 bit was received. 5 Bit 1 error. A 1 bit was transmitted, but a 0 bit was received. 15 Receiver overflow. A CAN message (or error message) could not be added to the receive queue because it was full.
uint32_t	timestamp	This field contains a timestamp (in microseconds) of when the error occurred.
uint8_t	data[8]	This array contains additional information about the state of the card: Byte Description 0 Transmit error counter. This counter increases with transmission errors and decreases with successful transmissions. 1 Receive error counter. This counter increases with receive errors and decreases with successful message receptions. 2-7 reserved

`CanCardInfo`

This structure is used to return some basic information about the CAN card.

uint32_t	serial	Card serial number
uint32_t	hwType	Card hardware type code
uint32_t	fwVer	Main firmware version. For example, version 1.02.03 would be formatted as 0x00010203.
uint32_t	bootVer	Boot loader version number.
uint32_t	fpgaVer	FPGA image version number.
uint32_t	driverVer	Driver version number.

Error codes

The following list of error codes are defined in the eican.h file.

#define	Value	Description
EICAN_ERR_OK	0	No error
EICAN_ERR_UNKNOWN_CMD	1	Passed command ID was unknown
EICAN_ERR_BAD_PARAM	2	Illegal parameter passed
EICAN_ERR_PORT_OPEN	3	Specified CAN port is already open
EICAN_ERR_PORT_CLOSED	4	Specified CAN port is not open
EICAN_ERR_CARD_BUSY	5	Card command area is busy
EICAN_ERR_INTERNAL	6	Some sort of internal device failure
EICAN_ERR_TIMEOUT	7	Card failed to respond to command
EICAN_ERR_SIGNAL	8	Signal received by driver
EICAN_ERR_MISSING_DATA	9	Not enough data was passed
EICAN_ERR_CMDMUTEX_HELD	10	The command mutex was being held
EICAN_ERR_QUEUECTRL	11	The CAN message queue head/tail was invalid
EICAN_ERR_FLASH	12	Failed to erase/program flash memory
EICAN_ERR_NOTERASED	13	Attempt to write firmware before erasing flash
EICAN_ERR_FLASHFULL	14	Too much data sent when programming flash
EICAN_ERR_UNKNOWN_IOCTL	15	Specified IOCTL code was unknown
EICAN_ERR_CMD_TOO_SMALL	16	Command passed without required header

#define	Value	Description
EICAN_ERR_CMD_TOO_BIG	17	Command passed with too much data
EICAN_ERR_CMD_IN_PROGRESS	18	Command already in progress on card
EICAN_ERR_CAN_DATA_LENGTH	19	More then 8 bytes of data sent with CAN message
EICAN_ERR_QUEUE_FULL	20	Transmit queue is full
EICAN_ERR_QUEUE_EMPTY	21	Receive queue is full
EICAN_ERR_READ_ONLY	22	Parameter is read only
EICAN_ERR_MEMORYTEST	23	Memory read/write test failure
EICAN_ERR_ALLOC	24	Memory allocation failure
EICAN_ERR_CMDFINISHED	25	Used internally by driver
EICAN_ERR_DRIVER	26	Generic device driver error
EICAN_ERR_CANT_OPEN_PORT	256	Unable to obtain handle to device driver

C++ language

The C++ language API is very similar to the C language API. The main difference is that the C language functions have been replaced by methods to a main class (**`EIcan`**) representing the CAN card port.

Class Methods

`EIcan::EIcan( void );`

The default constructor for the EIcan class. This simply initializes the classes internal data structures. The Eican::Open function needs to be called before the class can be used.

`Eican::~EIcan();`

The class destructor closes the port if it was open at the time that the object was destroyed.

`int EIcan::Open( int port, int baud );`

This method is used to open the CAN port. Before any other calls can be made which access this object, it must be successfully opened using this method.

port This parameter identifies the CAN port to open. CAN card ports are numbered from 0, so systems with a single dual channel card would have CAN ports numbered 0 and 1.

baud The bit rate at which to communicate on the CAN port. The value passed should be one of the #defined values in the eican header file. Options are:

EICAN_BITRATE_1000000	1 Megabit / second
EICAN_BITRATE_800000	500 Kbits / second
EICAN_BITRATE_500000	250 Kbits / second
EICAN_BITRATE_250000	250 Kbits / second
EICAN_BITRATE_125000	120 Kbits / second
EICAN_BITRATE_100000	100 Kbits / second
EICAN_BITRATE_50000	50 Kbits / second
EICAN_BITRATE_20000	20 Kbits / second

Return An error code is returned on any error, or zero on success.

`int EIcan::Close( void );`

This method is used to close a CAN port when it is no longer needed. After closing the port, no further calls can be made which access that port unless it is opened again through another call to **`Eican::Open`**.

Return An error code is returned on failure, or zero on success.

`int EIcan::Recv( CanFrame *frame, int32_t timeout );`

Wait for a new message to be received on the open CAN port and return it.

frame	A pointer to a CanFrame structure. If this function returns success, the received CAN frame will be stored in the structure referenced by this pointer.
timeout	The maximum amount of time to wait for a message before returning an error. The timeout is specified in millisecond units.
Return	An error code is returned on failure, or zero on success.

`int EIcan::Xmit( CanFrame *frame, int32_t timeout );`

Transmit a CAN message over the open CAN port. This function adds the passed message to the cards transmit queue and returns.

frame	A pointer to a CanFrame structure. This structure holds the CAN frame to be transmitted.
timeout	The maximum amount of time to wait for a message to be added to the transmit queue. Timeouts are specified in milliseconds.
Return	An error code is returned on failure, or zero on success.

`int EIcan::ReadCardInfo( CanCardInfo *info );`

This method can be used to retrieve some basic information about the CAN card.

info	A pointer to a CanCardInfo structure. This structure will be filled in by the function with information about the card.
Return	An error code is returned on failure, or zero on success.