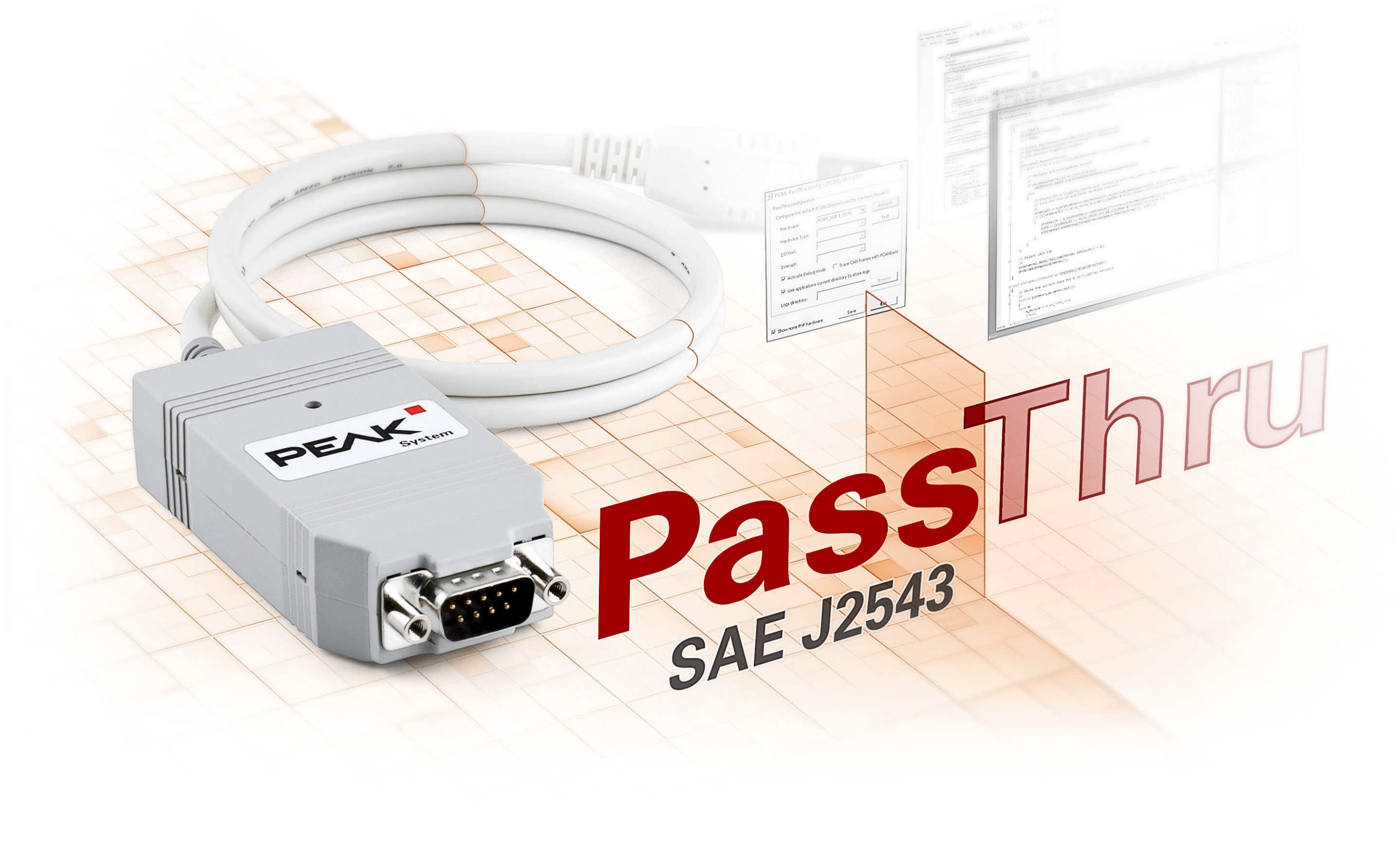
PCAN-PassThru API

Pass-Thru API and Connection of Pass-Thru Software to PEAK CAN Interfaces

User Manual



Document version 1.0.0 (2015-10-30)

PEAK-System Technik GmbH
Otto-Roehm-Strasse 69
64293 Darmstadt
Germany

Phone: +49 (0)6151 8173-20
Fax: +49 (0)6151 8173-29

www.peak-system.com
info@peak-system.com

Document version 1.0.0 (2015-10-30)

Contents

1 Introduction

For the programming of control units (ECU), there are many applications from various manufacturers which are used in the development and diagnosis of vehicle electronics. The interface for the communication between these applications and the control units is defined by the international standard SAE J2534 (Pass-Thru). Thus, the hardware for the connection to the control unit can be selected regardless of its manufacturer.

PCAN-PassThru allows the use of SAE J2534-based applications with CAN adapters from PEAK-System. The functions defined by the standard are provided by Windows DLLs for 32 and 64-bit systems. These can also be used to develop own Pass-Thru applications. The API is thread-safe. It uses mutual exclusion mechanisms to allow several threads from one or several processes to call functions of the API in a safe way.

The communication via CAN and OBD-2 (ISO 15765-4) is based on the programming interfaces PCAN-Basic and PCAN-ISO-TP. PCAN-PassThru is supplied with each PC CAN interface from PEAK-System.

Note: The SAE J2534 protocol is fully described in its norm. It is required for the development of your own Pass-Thru applications. This manual cannot supersede this API documentation.

1.1 Features

- Implementation of the international standard SAE J2534 (Pass-Thru)
- Use of SAE J2534 applications with PC CAN interfaces from PEAK-System
- Windows DLLs for the development of your own SAE J2534 applications for 32 and 64 bit
- Thread-safe API
- Physical communication via CAN and OBD-2 (ISO 15765-4) using a CAN interface of the PCAN series
- Uses the PCAN-Basic programming interface to access the CAN hardware in the computer
- Uses the PCAN-ISO-TP programming interface (ISO 15765-2) for the transfer of data packages up to 4095 bytes via the CAN bus

1.2 System Requirements

- Windows 10, 8.1, 7, Vista (32/64-bit)
- At least 512 MB RAM and 1 GHz CPU
- PC CAN interface from PEAK-System
- PCAN-Basic API
- PCAN-ISO-TP API

Note: The required APIs PCAN-Basic and PCAN-ISO-TP are installed both with the PCAN-PassThru setup.

1.3 Scope of Supply

- PCAN-PassThru API installation including
  - Interface DLLs for Windows (32/64-bit)
  - Configuration software for Windows 10, 8.1, 7, Vista (32/64-bit)
  - PCAN-Basic API
  - PCAN-ISO-TP API
- Documentation in PDF format

2 Installation

This chapter covers the setup of the PCAN-PassThru package under Windows and the use of its configuration software.

Do the following to install the package:

1. Insert the supplied DVD into the appropriate drive of the computer. Usually a navigation program appears a few moments later. If not, start the file `Intro.exe` from the root directory of the DVD.
2. In the main menu, select **Tools**, and then click on **Install**. Besides the product DVD, the setup is also available as a download from the website www.peak-system.com. Double click the included file `PCAN-PassThru Setup.exe` to start the installation.
3. Confirm the message of the User Account Control. The setup program for the package is started.
4. Follow the instructions of the program.

The setup will install the PCAN-PassThru interface DLLs for Windows 32- and 64-bit, the required APIs PCAN-Basic and PCAN-ISO-TP, and the configuration software PCAN-PassThru Configuration.

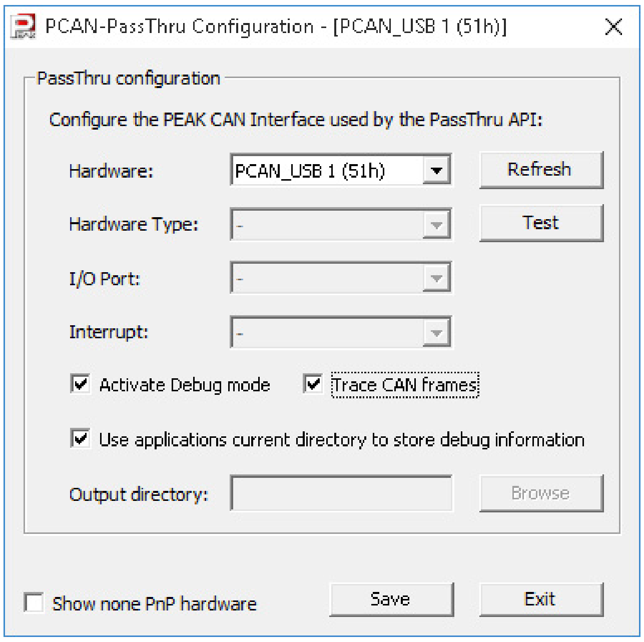
2.1 Hardware Interface Configuration

The CAN or OBD-II communication of PCAN-PassThru requires a CAN interface from PEAK-System which can be set up using the tool PCAN-PassThru Configuration. The desired PEAK CAN interface and its driver have to be successfully installed. A detailed description how to put your CAN interface into operation is included in the related documentation.

Do the following to configure a PEAK CAN interface to be used by the PCAN-PassThru API:

1. Open the Windows Start menu or the Windows Start page and select **PCAN-PassThru Configuration**.

   The dialog box for selecting the hardware and for setting the parameters appears.


PCAN-PassThru Configuration - [PCAN_USB 1 (51h)]

PassThru configuration

Configure the PEAK CAN Interface used by the PassThru API:

Hardware: PCAN_USB 1 (51h)

Hardware Type: -

I/O Port: -

Interrupt: -

[✓] Activate Debug mode [✓] Trace CAN frames

[✓] Use applications current directory to store debug information

Output directory:

[ ] Show none PnP hardware

Figure 1: PCAN-PassThru Configuration (PnP hardware selected)

2. If you like to use none PnP hardware like the PCAN-ISA, PCAN-PC/104, or PCAN-Dongle activate the **Show none PnP hardware** checkbox at the bottom on the left.
3. **Hardware:** Select the interface to be used by the PassThru API from the list. If this is a PnP hardware, the next 3 parameters are skipped.

   Note: If the adapter was connected to the computer after the tool was started, you can update the list with the button **Refresh** button.
4. **Hardware Type:** Select the type of your adapter.
5. **I/O Port and Interrupt:** Each CAN channel of none-PnP interfaces is set up with an interrupt (IRQ) and an I/O port before the hardware is installed in the computer. Select these parameters from the following drop-down lists.

   The entered parameters can be checked with the **Test** button on the top right.
6. Finally, save the configuration with **Save** or close the tool with **Exit**.

2.1.1 Additional Options

The PCAN-PassThru Configuration tool offers additional options for debugging and logging.

Activate Debug mode: Activate this checkbox to enable logging and the trace option. With this, all function calls with their parameters are saved to a csv file. The parameters are: timestamp of the function call, function name, return value, parameters, and error message.

The log file is named with the pattern:

```
PCAN_log_[Year][Month][Day][Hours][Minutes][Seconds]_[Counter].csv
```

Trace CAN frames: This option is available if **Activate Debug mode** is checked. With this enabled, the CAN traffic is traced to the configured directory. The trace file format PCAN-Trace 1.1 by PEAK-System is used. If the file reaches a size of 10 MB a new one is created. File naming follows the pattern:

```
[Year][Month][Day][Hours][Minutes][Seconds]_[Used_CAN-Channel]_[Counter].trc
```

Note: Both, creating log and trace files is done until the capacity of the hard drive is reached.

Output Directory: If **Use applications current directory to store debug information** is disabled, the directory for saving log and trace files can be chosen via **Browse**.

3 Programming Interface

3.1 Implementation

3.1.1 PassThru Functions

The following functions are available:

- **PassThruOpen**: establishes a connection and initializes the Pass-Thru device.
- **PassThruClose**: closes the connection to the Pass-Thru device.
- **PassThruConnect**: establishes a logical connection with a protocol channel on the specified SAE J2534 device.
- **PassThruDisconnect**: terminates a logical connection with a protocol channel.
- **PassThruReadMsgs**: reads messages and indications from the receive buffer.
- **PassThruWriteMsgs**: sends messages.
- **PassThruStartPeriodicMsg**: queues the specified message for transmission, and repeats at the specified interval.
- **PassThruStopPeriodicMsg**: stops the specified periodic message.
- **PassThruStartMsgFilter**: starts filtering of incoming messages.
- **PassThruStopMsgFilter**: removes the specified filter.
- **PassThruSetProgrammingVoltage**: Sets a single programming voltage on a single specific pin. This function is not supported.
- **PassThruReadVersion**: returns the version strings associated with the DLL.
- **PassThruIoctl**: read/write configuration parameters. Only the following IOCTL ID values are supported:
  - GET_CONFIG: read configuration parameters. Only the following parameter details are supported:
    - DATA_RATE: get the bit rate value (in bps).
    - LOOPBACK: states whether Tx messages are echoed (1) or not (0).
    - BIT_SAMPLE_POINT: bit sample point as a percentage of the bit time.
    - SYNC_JUMP_WIDTH: synchronization jump width (undefined unit, value from BRCan).
      Note: Use a pointer to an `SCONFIG_LIST` structure for the InputPtr parameter.
  - SET_CONFIG: write configuration parameters. Only the following parameter details are supported:
    - DATA_RATE: set the bit rate value (in bps). This will call a disconnection and a connection of the channel.
    - LOOPBACK: states whether Tx messages are echoed (1) or not (0).
      Note: Use a pointer to an `SCONFIG_LIST` structure for the InputPtr parameter.
  - CLEAR_TX_BUFFER: Tx AND Rx buffer will be cleared.
  - CLEAR_RX_BUFFER: Tx AND Rx buffer will be cleared.
  - CLEAR_PERIODIC_MSGS: clears periodic messages.
  - CLEAR_MSGS_FILTERS: clears filter messages.

3.1.2 PassThru Message Structure

The ISO-TP message has the following structure:

```
typedef struct {
      unsigned long ProtocolID;
      unsigned long RxStatus;
      unsigned long TxFlags;
      unsigned long Timestamp;
      unsigned long DataSize;
      unsigned long ExtraDataIndex;
      unsigned char Data[4128];
} PASSTHRU_MSG;
```

Data fileds:

- **ProtocolID:** supported protocols are `CAN` (0x05) or `ISO-15765` (0x06).
- **RxStatus:** only available when receiving a message. Supported flags are:
  - CAN_29BIT_ID: the value of the bit #8 states if the message is standard (11bit, value = 0) or extended (29bit, value = 1).
  - TX_MSG_TYPE: the value of the bit #0 states if the message is a transmit loopback (value = 1) or not (value = 0).
- **TxFlags:** transmit message flags. Supported flags are:
  - CAN_29BIT_ID: the value of the bit #8 states if the message is standard (11bit, value = 0) or extended (29bit, value = 1).
- **Timestamp:** timestamp in microseconds.
- **DataSize:** data size in bytes (including the 4 CAN ID bytes).
- **ExtraDataIndex:** when no extra data, the value should be equal to DataSize.
- **Data:** array of data bytes.

3.1.3 PassThruIoctl

The following structures are used with the PassThruIoctl and the Ioctl ID values GET_CONFIG and SET_CONFIG.

```
// parameter for PassThruIoCtl (used with GET_CONFIG/SET_CONFIG Ioctl IDs)
typedef struct {
      unsigned long Parameter;      // name of the parameter
      unsigned long Value;          // value of the parameter
} SCONFIG;

// list of parameters for PassThruIoCtl (used with GET_CONFIG/SET_CONFIG Ioctl IDs)
typedef struct {
      unsigned long NumOfParams;    // number of SCONFIG elements
      SCONFIG* ConfigPtr;           // array of SCONFIG
} SCONFIG_LIST;
```

3.2 Function Examples

The following example is divided in several steps demonstrating the supported PassThru functions.

3.2.1 Opening a PassThru Device

With this step a PassThru device is opened. Furthermore checking for an error is shown.

```
#define PASSTHRU_NB_MSG       10

TPTResult result;
ULONG deviceId;

result = PassThruOpen(NULL, &deviceId);
if (result != STATUS_NOERROR)
{
      char errText[BUFSIZ];
      memset(errText, 0, sizeof(BUFSIZ));

      if (STATUS_NOERROR != PassThruGetLastError(errText))
            fprintf(stderr, "Failed to get LastError.\n");
      else
            fprintf(stderr, errText);
}
```

3.2.2 Connecting to CAN

This step shows how to connect to a raw CAN channel.

```
ULONG channelId;

result = PassThruConnect(deviceId, CAN, 0, BAUDRATE_250K, &channelId);

if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.3 Writing Messages

```
PASSTHRU_MSG pMsg[PASSTHRU_NB_MSG];
ULONG pNumMsgs;
ULONG i, j;

// initialization
memset(pMsg, 0, sizeof(PASSTHRU_MSG) * PASSTHRU_NB_MSG);
pNumMsgs = PASSTHRU_NB_MSG;
for (i = 0; i < pNumMsgs; i++)
{
      // initialize each message
      j = 0;
      pMsg[i].ProtocolID = CAN;

      // set length
      pMsg[i].DataSize = min(4+i, 4+8);

      // set CAN ID
      pMsg[i].Data[j++] = 0x00;
      pMsg[i].Data[j++] = 0x00;
      pMsg[i].Data[j++] = 0x00;
      pMsg[i].Data[j++] = (unsigned char) (0xA0 + i);

      // set CAN Data
      for (; j < pMsg[i].DataSize; j++)
            pMsg[i].Data[j] = (unsigned char) (0xB0 + j);
}

// write messages
result = PassThruWriteMsgs(channelId, pMsg, &pNumMsgs, 0);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.4 Setting a Filter Message

```
PASSTHRU_MSG pMsgMask, pMsgPattern;
ULONG filterId;

// initialization
memset(&pMsgMask, 0, sizeof(PASSTHRU_MSG));
memset(&pMsgPattern, 0, sizeof(PASSTHRU_MSG));

// filter on CAN ID 11bit
pMsgMask.ProtocolID     = pMsgPattern.ProtocolID = CAN;
pMsgMask.DataSize       = pMsgPattern.DataSize = 4;

// filter ID anything like 0x????0140
pMsgMask.Data[2]        = 0xFF;
pMsgMask.Data[3]        = 0xFF;
pMsgPattern.Data[2]     = 0x01;
pMsgPattern.Data[3]     = 0x40;

// set a filter message
result = PassThruStartMsgFilter(channelId, PASS_FILTER, &pMsgMask, &pMsgPattern,
NULL, &filterId);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.5 Reading Messages

```
// initialization
memset(pMsg, 0, sizeof(PASSTHRU_MSG) * PASSTHRU_NB_MSG);
pNumMsgs = PASSTHRU_NB_MSG;

// read messages
result = PassThruReadMsgs(channelId, pMsg, &pNumMsgs, 0);
if (result == STATUS_NOERROR)
      { /* Process messages… */ }
else if (result == ERR_BUFFER_EMPTY)
      { printf("No message received"); }
else  { /* TODO … */ }
```

3.2.6 Tx Loopback Configuration

This step demonstrates how to set and get a Tx loopback configuration.

```
SCONFIG_LIST configList;
SCONFIG configs[10];
unsigned long nbParams;

// prepare parameter Tx loopack to enabled
nbParams = 0;
configs[nbParams].Parameter   = LOOPBACK;
configs[nbParams++].Value     = 1;
configList.NumOfParams        = nbParams;
configList.ConfigPtr          = configs;

// set configuration
result = PassThruIoctl(channelId, SET_CONFIG, &configList, NULL);
if (result != STATUS_NOERROR) { /* TODO … */ }

// read configuration
configs[0].Value = 0;
result = PassThruIoctl(channelId, GET_CONFIG, &configList, NULL);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.7 Periodic Messages

The step covers the setup of periodic messages.

```
PASSTHRU_MSG msg;
ULONG msgID;
ULONG timeInterval;

// set up a periodic message
memset(&msg, 0, sizeof(PASSTHRU_MSG));
msg.ProtocolID    = CAN;
msg.DataSize      = 7+4;

// set CAN ID
j = 0;
msg.Data[j++] = 0x00;
msg.Data[j++] = 0x00;
msg.Data[j++] = 0x00;
msg.Data[j++] = (unsigned char) (0xC1);

// set CAN Data
for (; j < msg.DataSize; j++)
      msg.Data[j] = (unsigned char) (0xB0 + j);

timeInterval = 100;
result = PassThruStartPeriodicMsg(channelId, &msg, &msgID, timeInterval);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.8 Disconnect from the channel

```
// disconnects from the channel
result = PassThruDisconnect(channelId);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.2.9 Close the device

```
// closes the device
result = PassThruClose(deviceId);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.3 Technical Notes

3.3.1 Rx/Tx Queues

There is no receive queue or transmit queue:

- When PassThruReadMsgs is called, the messages are directly read from PCAN-Basic via a loop.
- When PassThruWriteMsgs is called, the messages are directly written via PCAN-Basic.

3.3.2 Message Filtering

Although the PCAN-Basic API provides message filtering features, it was not used for the PCAN-PassThru API since the way you define a filter in this API differs a lot from the way used in PCAN-Basic. PCAN-PassThru uses a mask and a pattern for the CAN ID and the data. PCAN-Basic uses a range of CAN IDs instead.

3.3.3 Periodic Messages

Periodic messages are handled via a map for each channel. A unique thread periodically checks each channel every 1ms and transmits messages if needed.

4 License Information

The APIs PCAN-PassThru, PCAN-Basic, and PCAN-ISO-TP are property of the PEAK-System Technik GmbH and may be used only in connection with a hardware component purchased from PEAK-System or one of its partners. If CAN hardware of third-party suppliers should be compatible to that of PEAK-System, then you are not allowed to use the mentioned APIs with those components.

If a third-party supplier develops software based on the mentioned APIs and problems occur during the use of this software, consult that third-party supplier.