PCAN-PassThru API 05.00
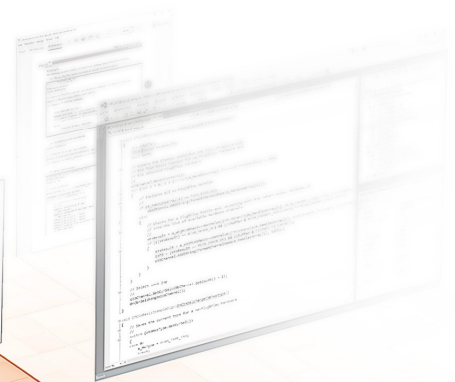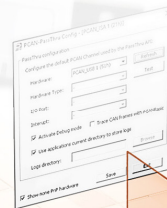
Pass-Thru API and Connection of Pass-Thru
Software to PEAK CAN Interfaces

User Manual

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 version 05.00 (Pass-Thru)
- Implementation of SAE J2534 extension J2535-2:2019 (adapted to match version 05.00 differences) to support CAN FD, and General Motors' extension GMW17753 to support messages extension
- Use of SAE J2534 applications with PC CAN interfaces from PEAK-System
- Windows DLLs for the development of your own SAE J2534 applications for the platforms Windows® 11 (x64/ARM64), 10 (x86/x64)
- 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 (or 32768 with GMW17753 extension) bytes via the CAN bus

1.2 System Requirements

- Windows 11 (x64/ARM64), Windows 10 (x64)
- For the CAN bus connection: PC CAN interface from PEAK-System
- PCAN-Basic API
- PCAN-ISO-TP API

> **Note:** The required API PCAN-ISO-TP is installed with the PCAN-PassThru setup. The PCAN-Basic API must be installed using the PEAK-Drivers Setup.

1.3 Scope of Supply

⌐ PCAN-PassThru API installation including

- Interface DLLs for Windows (x86/x64) for protocol version 4.04 and 5.00

- Configuration software for Windows

- 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. The setup is available as a download from the website www.peak-system.com. Double click the included file `PCAN-PassThru Setup.exe` to start the installation.

2. Confirm the message of the User Account Control. The setup program for the package is started.

3. 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.
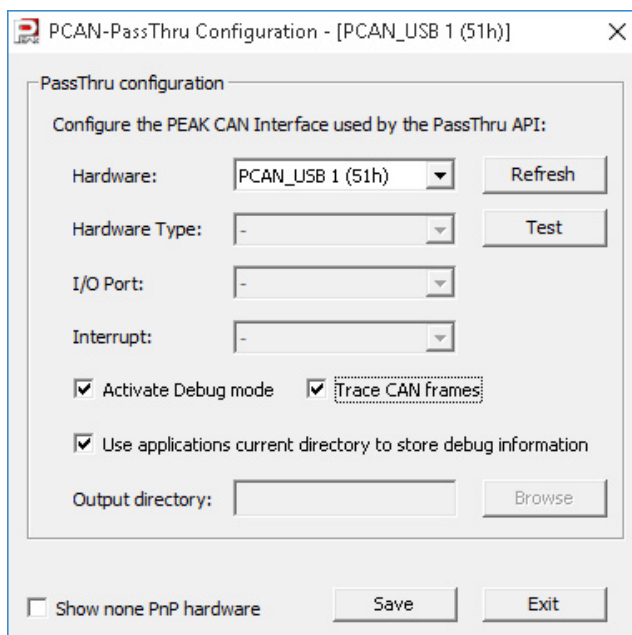
Figure 1: PCAN-PassThru Configuration (PnP hardware selected)

2. If you like to use non-PnP hardware like the PCAN-ISA, PCAN-PC/104, or PCAN-Dongle activate the **Show non-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 non-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]_[Device Name]_[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 connection with a "physical" protocol channel on the specified SAE J2534 device.

⌐ **PassThruDisconnect**: terminates a connection with a "physical" protocol channel.

⌐ **PassThruLogicalConnect**: establishes a connection with a "logical" protocol channel on the specified physical channel.

⌐ **PassThruLogicalDisconnect**: terminates a connection with a "logical" protocol channel.

⌐ **PassThruReadMsgs**: reads messages and indications from the receive buffer.

⌐ **PassThruQueueMsgs**: 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.

⌐ PassthruGetLastError: get the description of the last error.

⌐ PassthruSelect: selects specific channels for available messages.

⌐ PassthruScanForDevices: generates a list of accessible Passthru devices.

⌐ PassthruGetNextDevice: one at a time, returns the accessible Passthru devices.

⌐ **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) (physical channels).
  - ECHO_PHYSICAL_CHANNEL_TX: states whether Tx messages are echoed (1) or not (0) (physical channels).
  - BIT_SAMPLE_POINT: bit sample point as a percentage of the bit time (physical channels).
  - SYNC_JUMP_WIDTH: synchronization jump width (undefined unit, value from BRCan) (physical channels).
  - FD_CAN_DATA_PHASE_RATE: get the CAN-FD data phase rate (physical channels).
  - J1962_PINS: get the pins for Pin-Switch protocols (physical channels).

- ISO15765_WAIT_LIMIT: get the ISO-TP N_WFTmax parameter (logical channels).
- ISO15765_PAD_VALUE, ISO15765_PAD_VALUE_2: get the pad byte value (logical channels).
- ISO15765_BS: get the ISO-TP block size (BS) parameter (logical channels).
- ISO15765_STMIN: get the ISO-TP separation time (STmin) parameter (logical channels).
- N_AS_MAX: get the ISO-TP timeout As parameter (logical channels).
- N_AR_MAX: get the ISO-TP timeout Ar parameter (logical channels).
- N_BS_MAX: get the ISO-TP timeout Bs parameter (logical channels).
- N_CR_MAX, N_CR_MAX_2: get the ISO-TP timeout Cr parameter (logical channels).
- BS_TX: get the ISO-TP transmit block size parameter (logical channels).
- STMIN_TX: get the ISO-TP transmit separation time (logical channels).
- FD_ISO15765_TX_DATA_LENGTH: get the ISO-TP maximum data length code (logical channels on CAN-FD).

Note: Use a pointer to a `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 (physical channels).
  - ECHO_PHYSICAL_CHANNEL_TX: states whether Tx messages are echoed (1) or not (0) (physical channels).
  - FD_CAN_DATA_PHASE_RATE: set the CAN-FD data phase rate (physical channels).
  - J1962_PINS: assigns the pins for Pin-Switch protocols (physical channels).
  - ISO15765_WAIT_LIMIT: sets the ISO-TP N_WFTmax parameter (logical channels).
  - ISO15765_PAD_VALUE, ISO15765_PAD_VALUE_2: sets the pad byte value (logical channels).
  - ISO15765_BS: sets the ISO-TP block size (BS) parameter (logical channels).
  - ISO15765_STMIN: sets the ISO-TP separation time (STmin) parameter (logical channels).
  - N_AS_MAX: sets the ISO-TP timeout As parameter (logical channels).
  - N_AR_MAX: sets the ISO-TP timeout Ar parameter (logical channels).
  - N_BS_MAX: sets the ISO-TP timeout Bs parameter (logical channels).
  - N_CR_MAX, N_CR_MAX_2: sets the ISO-TP timeout Cr parameter (logical channels).
  - BS_TX: sets the ISO-TP transmit block size parameter (logical channels).
  - STMIN_TX: sets the ISO-TP transmit separation time (logical channels).
  - FD_ISO15765_TX_DATA_LENGTH: sets the ISO-TP maximum data length code (logical channels on CAN-FD).

Note: Use a pointer to an `SCONFIG_LIST` structure for the InputPtr parameter.

- **CLEAR_TX_QUEUE** on physical channels (Tx and Rx queues will be cleared, and queues of the logical channels connected on the physical channel will be cleared).
- **CLEAR_TX_QUEUE** on logical channels.
- **CLEAR_RX_QUEUE** on physical channels (Tx and Rx queues will be cleared, and queues of the logical channels connected on the physical channel will be cleared).
- **CLEAR_RX_QUEUE** on logical channels.

- CLEAR_PERIODIC_MSGS: clears periodic messages.

- CLEAR_MSGS_FILTERS: clears filter messages (physical channels).

- BUS_ON: restores the controller as it was before it moved to BUS OFF (physical channels).

- GET_DEVICE_INFO: acquires the general capabilities of the device.

- GET_PROTOCOL_INFO: acquires the protocol specific capabilities of the devices.

- READ_MSG_EXTENSION: reads the message extension (logical channels on FD protocols)

- WRITE_MSG_EXTENSION: writes the message extension (logical channels on FD protocols)

3.1.2   Pass-Thru Message Structure

The Pass-Thru message has the following structure:

```
typedef struct {
    unsigned long ProtocolID;
    unsigned long MsgHandle;
    unsigned long RxStatus;
    unsigned long TxFlags;
    unsigned long Timestamp;
    unsigned long DataLength;
    unsigned long ExtraDataIndex;
    unsigned char *DataBuffer;
    unsigned long DataBufferSize;
} PASSTHRU_MSG;
```

Data fields:

- **ProtocolID**: supported protocols are CAN (0x05), CAN_PS (0x8004), FD_CAN_PS (0x8011), or ISO15765_LOGICAL (0x200).

- MsgHandle: a number assigned by the application, to uniquely identify the message. Setting this number to 0 will disable reception of TxDone/TxFailed indications.

- **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).

  - FD_CAN_FORMAT: the value of the bit #20 states if the message is a CAN-FD message (value = 1) or not (value = 0).

  - FD_CAN_BRS: the value of the bit #19 states if the message is a FD bit rate switch (value = 1) or not (value = 0).

  - FD_CAN_ESI: the value of the bit #21 states if the message is a FD error-state indicator (value = 1) or not (value = 0).

  - ISO15765_ADDR_TYPE: the value of the bit #7 states if the message has the ISO-TP extended addressing format (value = 1) or not (value = 0).

  - BUFFER_OVERFLOW: the value of the bit #6 states if an overrun error has been detected (value = 1) or not (value = 0).

  - RX_MSG_TRUNCATED: the value of the bit #27 states if the message is truncated and the message extension should be read (value = 1) or not (value = 0).

- TX_SUCCESS: the value of the bit #3 states if the message is a TxDone indication (value = 1) or not (value = 0).

- TX_FAILED: the value of the bit #9 states if the message is a TxFailed indication (value = 1) or not (value = 0).

- START_OF_MESSAGE: the value of the bit #1 states if the message is a Start indication (value = 1) or not (value = 0).

- ERROR_INDICATION: the value of the bit #5 states if the message is an Error indication (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).

- FD_CAN_FORMAT: the value of the bit #20 states if the message is a CAN-FD message (value = 1) or not (value = 0).

- FD_CAN_BRS: the value of the bit #19 states if the message is a FD bit rate switch (value = 1) or not (value = 0).

- ISO15765_ADDR_TYPE: the value of the bit #7 states if the message has the ISO-TP extended addressing format (value = 1) or not (value = 0).

└ **Timestamp:** timestamp in microseconds.

└ **DataBuffer:** array of data bytes, allocated and freed by your application.

└ **DataBufferSize:** total number of bytes in the data buffer.

└ **DataLength:** number of bytes in the data buffer actually used for message data (including the 4 CAN ID bytes). The DataLength should never be larger than the DataBufferSize. The only exception is when the message is a truncated message, then the DataLength reflects the whole "message length", including the extension length.

└ **ExtraDataIndex:** when no extra data, the value should be equal to DataLength, except is the Pass-Thru message is an Indication.

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;
```

The following structures are used with the PassThruIoctl and the Ioctl ID values GET_DEVICE_INFO and GET_PROTOCOL_INFO.

```
// parameter for PassThruIoCtl (used with GET_DEVICE_INFO/GET_PROTOCOL_INFO Ioctl
IDs)
typedef struct {
```

```
      unsigned long Parameter;       // name of the parameter
      unsigned long Value;           // value of the parameter
      unsigned long Supported;       // support for parameter
} SPARAM;
// list of parameters for PassThruIoCtl (used with
GET_DEVICE_INFO/GET_PROTOCOL_INFO Ioctl IDs)
typedef struct {
      unsigned long NumOfParams;     // number of SCONFIG elements
      SPARAM* ParamPtr;              // array of SPARAM
} SPARAM_LIST;
```

The following structure is used with the PassThruIoctl and the Ioctl ID values WRITE_MSG_EXTENSION and READ_MSG_EXTENSION.

```
typedef struct {
      unsigned long NumOfBytes;
      unsigned char* BytePtr;
} SBYTE_ARRAY;
```

3.1.4   Other structures

The following structure is used with the PassThruGetNextDevice function.

```
typedef struct {
      char DeviceName[80]; // Device name
      unsigned long DeviceAvailable;     // indicates whether the device is
                                         // currently open or not
      unsigned long DeviceDLLFWStatus;   // indicates the status of the DLL /
                                         // Firmware compatibility.
      unsigned long DeviceConnectMedia;  // indicates the type of media used to
                                         // connect to the Pass-Thru Device
      unsigned long DeviceConnectSpeed;  // indicates the connection speed in bits
                                         // per second that the Pass-Thru Device
                                         // detects.
      unsigned long DeviceSignalQuality; // indicates the signal quality that the
                                         // device detects.
      unsigned long DeviceSignalStrength; // indicates the signal strength the
                                         // device detects.
} SDEVICE;
```

The following structure is used with the PassThruConnect and PassthruSetPogrammingVoltage functions.

```
typedef struct
{
      unsigned long Connector;           // connector identifier
      unsigned long NumOfResources;      // number of resources pointed to by
                                         // ResourceListPtr
      unsigned long *ResourceListPtr;    // pointer to list of resources
} RESOURCE_STRUCT;
```

The following structure is used with the PassThruSelect function.

```
typedef struct
{
      unsigned long ChannelCount;        // number of ChannelList elements
```

```
      unsigned long ChannelThreshold;    // minimum number of channels that must
                                         // have messages
      unsigned long *ChannelList;        // pointer to an array of Channel IDs to
                                         // be monitored
} SCHANNELSET;
```

The following structure is used with the PassThruLogicalConnect function.

```
typedef struct
{
      unsigned long LocalTxFlags;   // TxFlags for the Local Address
      unsigned long RemoteTxFlags;  // TxFlags for the Remote Address
      char LocalAddress[5];         // Address for J2534 Device ISO 15765 end point
      char RemoteAddress[5];        // Address for remote/vehicle ISO 15765 end
                                    // point
} ISO15765_CHANNEL_DESCRIPTOR;
```

3.2   Function Examples on CAN

The following example is divided in several steps demonstrating the supported PassThru functions on a physical CAN channel.

3.2.1   Opening a PassThru Device

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

```
#define BUFSIZ          255

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);
}
```

To connect to a specific device, the "pName" parameter must be formatted like such:

⌐ J2534-2:PEAK {PCANHandle}

⌐ Where {PCANHandle} is one of the following PCAN Device Channel Handle:

Value	Description	Value	Description
0x41	PCAN-PCI interface, channel 1	0x50A	PCAN-USB interface, channel 10
0x42	PCAN-PCI interface, channel 2	0x50B	PCAN-USB interface, channel 11
0x43	PCAN-PCI interface, channel 3	0x50C	PCAN-USB interface, channel 12
0x44	PCAN-PCI interface, channel 4	0x50D	PCAN-USB interface, channel 13
0x45	PCAN-PCI interface, channel 5	0x50E	PCAN-USB interface, channel 14
0x46	PCAN-PCI interface, channel 6	0x50F	PCAN-USB interface, channel 15
0x47	PCAN-PCI interface, channel 7	0x510	PCAN-USB interface, channel 16
0x48	PCAN-PCI interface, channel 8	0x61	PCAN-PC Card interface, channel 1
0x409	PCAN-PCI interface, channel 9	0x62	PCAN-PC Card interface, channel 2
0x40A	PCAN-PCI interface, channel 10	0x801	PCAN-LAN interface, channel 1
0x40B	PCAN-PCI interface, channel 11	0x802	PCAN-LAN interface, channel 2
0x40C	PCAN-PCI interface, channel 12	0x803	PCAN-LAN interface, channel 3
0x40D	PCAN-PCI interface, channel 13	0x804	PCAN-LAN interface, channel 4
0x40E	PCAN-PCI interface, channel 14	0x805	PCAN-LAN interface, channel 5
0x40F	PCAN-PCI interface, channel 15	0x806	PCAN-LAN interface, channel 6
0x410	PCAN-PCI interface, channel 16	0x807	PCAN-LAN interface, channel 7
0x51	PCAN-USB interface, channel 1	0x808	PCAN-LAN interface, channel 8
0x52	PCAN-USB interface, channel 2	0x809	PCAN-LAN interface, channel 9
0x53	PCAN-USB interface, channel 3	0x80A	PCAN-LAN interface, channel 10
0x54	PCAN-USB interface, channel 4	0x80B	PCAN-LAN interface, channel 11
0x55	PCAN-USB interface, channel 5	0x80C	PCAN-LAN interface, channel 12
0x56	PCAN-USB interface, channel 6	0x80D	PCAN-LAN interface, channel 13
0x57	PCAN-USB interface, channel 7	0x80E	PCAN-LAN interface, channel 14
0x58	PCAN-USB interface, channel 8	0x80F	PCAN-LAN interface, channel 15

| 0x509 | PCAN-USB interface, channel 9 | 0x810 | PCAN-LAN interface, channel 16 |

With this step, the 3ʳᵈ PEAK device connected on USB is opened.

```
TPTResult result;
ULONG deviceId;

result = PassThruOpen("J2534-2:PEAK 0x53", &deviceId);
if (result != STATUS_NOERROR)
{
      [...]
}
```

3.2.2   Connecting a physical channel to CAN

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

```
ULONG channelId;
RESOURCE_STRUCT RscStr;
RscStr.Connector = J1962_CONNECTOR;
RscStr.NumOfResources = 2;
unsigned long Rsc[2];
RscStr.ResourceListPtr = Rsc;
RscStr.ResourceListPtr[0] = 6;
RscStr.ResourceListPtr[1] = 14;
result = PassThruConnect(deviceId, CAN, 0, BAUDRATE_250K, RscStr, &channelId);

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

3.2.3   Writing Messages

```
#define PASSTHRU_NB_MSG 10
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++)
{
      // Initializes each message.
      j = 0;
      pMsg[i].ProtocolID = CAN;
      pMsg[i].DataBuffer = new unsigned char[50];
      pMsg[i].DataBufferSize = 50;
      pMsg[i].MsgHandle = 0;

      // Sets a length.
      pMsg[i].DataLength = min(4+i, 4+8);

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

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

// Writes the messages.
result = PassThruQueueMsgs(channelId, pMsg, &pNumMsgs);
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));
pMsgMask.DataBuffer = new unsigned char[4]{};
pMsgPattern.DataBuffer = new unsigned char[4]{};
pMsgMask.DataBufferSize = pMsgMask.DataLength = 4;
pMsgPattern.DataBufferSize = pMsgPattern.DataLength = 4;

// Filters on a 11-bit CAN ID.
pMsgMask.ProtocolID    = pMsgPattern.ProtocolID = CAN;

// Filters on an ID anything like 0x????0140.
pMsgMask.DataBuffer[2]       = 0xFF;
pMsgMask.DataBuffer[3]       = 0xFF;
pMsgPattern.DataBuffer[2]    = 0x01;
pMsgPattern.DataBuffer[3]    = 0x40;

// Sets a filter message.
```

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

3.2.5   Reading Messages

```
// initialization
pNumMsgs = PASSTHRU_NB_MSG;

// Reads the 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;

// Prepares a parameter Tx loopack to enable.
nbParams = 0;
configs[nbParams].Parameter  = ECHO_PHYSICAL_CHANNEL_TX;
configs[nbParams++].Value    = 1;
configList.NumOfParams       = nbParams;
configList.ConfigPtr         = configs;

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

// Reads the 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;
unsigned char data[11];
ULONG msgID;
ULONG timeInterval;

// Sets up a periodic message.
memset(&msg, 0, sizeof(PASSTHRU_MSG));
msg.DataBuffer = data;
msg.ProtocolID = CAN;
msg.DataBufferSize = msg.DataLength = 7+4;

// Sets a CAN ID
j = 0;
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = (unsigned char) (0xC1);

// Sets a CAN Data
for (; j < msg.DataLength; j++)
      msg.DataBuffer[j] = (unsigned char) (0xB0 + j);

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

3.2.8   Disconnect from the physical channel

```
// Disconnect 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   Function Examples on CAN FD and extended messages

The following example is divided in several steps demonstrating the supported PassThru functions on a logical ISO15765 on CAN FD channel with messages extension.

3.3.1   Opening a PassThru Device

With this step the default PassThru device is opened. See section 3.2.1 for more details.

```c
#define BUFSIZ          255

TPTResult result;
ULONG deviceId;

result = PassThruOpen(NULL, &deviceId);
if (result != STATUS_NOERROR) {
      /* TODO … */
}
```

3.3.2   Connecting a physical channel to CAN FD

This step shows how to connect to a physical CAN FD channel.

As we use a "PS" protocol and as pins are defined in the PassThruConnect function, there is no further need to configure the pins (this would be ignored).

```c
ULONG channelId;
RESOURCE_STRUCT RscStr;
RscStr.Connector = J1962_CONNECTOR;
RscStr.NumOfResources = 2;
unsigned long Rsc[2];
RscStr.ResourceListPtr = Rsc;
RscStr.ResourceListPtr[0] = 6;
RscStr.ResourceListPtr[1] = 14;
result = PassThruConnect(deviceId, FD_CAN_PS, 0, BAUDRATE_500K, RscStr,
&channelId);

if (result != STATUS_NOERROR) {
      SCONFIG_LIST configList;
      SCONFIG configs[1];

      // set data bitrate
      configs[0].Parameter = FD_CAN_DATA_PHASE_RATE;
      configs[0].Value = 2000000;
      configList.NumOfParams = 1;
      configList.ConfigPtr = configs;
      result = PassThruIoctl(channelId, SET_CONFIG, &configList, NULL);
      if (result != STATUS_NOERROR) { /* TODO … */ }

}
```

An alternative step below shows how to connect to a physical CAN FD channel, like in PassThru 2004.

As we use a "PS" protocol and as pins are not defined in the PassThruConnect function, we need to configure the pins.

```
ULONG channelId;
RESOURCE_STRUCT RscStr;
memset(&RscStr, 0, sizeof(RESOURCE_STRUCT));
result = PassThruConnect(deviceId, FD_CAN_PS, 0, BAUDRATE_500K, RscStr,
&channelId);

if (result != STATUS_NOERROR) {
    SCONFIG_LIST configList;
    SCONFIG configs[2];

    // set data bitrate
    configs[0].Parameter = FD_CAN_DATA_PHASE_RATE;
    configs[0].Value = 2000000;
    configs[1].Parameter = J1962_PINS;
    configs[1].Value = 0x60E;
    configList.NumOfParams = 2;
    configList.ConfigPtr = configs;
    result = PassThruIoctl(channelId, SET_CONFIG, &configList, NULL);
    if (result != STATUS_NOERROR) { /* TODO … */ }

}
```

3.3.3　Connecting a logical channel

This step shows how to connect a logical channel on the physical channel.

```
ULONG logicalChannelId;
ISO15765_CHANNEL_DESCRIPTOR chDescr;
memset(&chDescr, 0, sizeof(ISO15765_CHANNEL_DESCRIPTOR));
chDescr.RemoteAddress[3] = 0xA2;
chDescr.LocalAddress[3] = 0xA1;
result = PassThruLogicalConnect(channelId, ISO15765_LOGICAL, FULL_DUPLEX, &chDescr,
&logicalChannelId);

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

3.3.4   Writing an extended message on the logical channel

This step shows how to write a message larger than 4128 bytes, using message extension.

```
ULONG pNumMsgs;
ULONG i, j;
PASSTHRU_MSG msg;

// initialization
memset(&msg, 0, sizeof(PASSTHRU_MSG));
pNumMsgs = 1;

j = 0;
msg.ProtocolID = ISO15765_LOGICAL;
msg.DataBuffer = new unsigned char[4128];
msg.DataBufferSize = 4128;
msg.MsgHandle = 0;

// Sets a length.
msg.DataLength = 4128 + 28640;

// Sets CAN ID.
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = 0x00;
msg.DataBuffer[j++] = 0xA2;

// Sets CAN Data.
for (; j < msg.DataLength; j++)
     msg.DataBuffer[j] = (unsigned char) (0xB0 + j);

// Initialize message extension.
SBYTE_ARRAY pInput = {};
pInput.NumOfBytes = 28640;
unsigned char tByte[28640]{ /* Put some data here */ };
pInput.BytePtr = tByte;
// Writes message extension.
result = PassThruIoctl(logicalChannelId, WRITE_MSG_EXTENSION, &pInput, NULL);
if (result == STATUS_NOERROR) {
     // Writes the message.
     result = PassThruQueueMsgs(logicalChannelId, &msg, &pNumMsgs);
     if (result != STATUS_NOERROR) { /* TODO … */ }
}
```

3.3.5   Reading an extended message on the logical channel

```
// initialization
memset(&msg, 0, sizeof(PASSTHRU_MSG));
pNumMsgs = 1;

// Reads the message.
result = PassThruReadMsgs(channelId, &msg, &pNumMsgs, 0);
if (result == STATUS_NOERROR) {
     // Checks if the message is extended
     if ((msg.RxStatus & RX_MSG_TRUNCATED) == RX_MSG_TRUNCATED) {
          // Initializes the message extension.
          SBYTE_ARRAY pOutput;
          memset(&pOutput, 0, sizeof(SBYTE_ARRAY));
```

```
            unsigned char outData[28640];
            memset(outData, 0, 28640);
            pOutput.BytePtr = outData;
            pOutput.NumOfBytes = msg.DataLength - 4128;
            unsigned long indexOfReadMsg = 0;
            // Read the extension.
            result = PassThruIoctl(logicalChannelId, READ_MSG_EXTENSION,
                    &indexOfReadMsg, &pOutput);
            if (result != STATUS_NOERROR) { /* TODO … */ }
    }
}
```

3.3.6   Disconnect from the logical channel

```
// Disconnect from the channel.
result = PassThruLogicalDisconnect(logicalChannelId);
if (result != STATUS_NOERROR) { /* TODO … */ }
```

3.3.7   Disconnect from the physical channel

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

3.3.8   Close the device

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

3.4   Technical Notes

3.4.1   Rx/Tx Queues

There is no transmit queue:

⌐ When PassThruQueueMsgs is called, the messages are directly queued in PCAN-ISO-TP.

3.4.2   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.

4 License Information

The use of this software is subject to the terms of the End User License Agreement of PEAK-System Technik GmbH.

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.

Appendix A Supported Features from SAE J2534 and GM Extensions

A.1   SAE J2534-1 Revised OCT2015

A.1.1   Communication Protocols

The following figure lists the supported communication protocols:

Protocol	Supported?
CAN	Yes
ISO 15765	Yes
ISO 9141	No
ISO 14230	No
SAE J1850	No
SAE J2610	No

A.1.2   Other Features

Feature	Supported?
PROGRAMMABLE POWER SUPPLY	Software pin assignment is not supported. Function **PassThruSetProgrammingVoltage** always returns ERR_NOT_SUPPORTED.
USE PASSTHRUOPEN WITH A DEVICE NAME	YES. The **pName** argument in **PassThruOpen** function can start with J2534-1:PEAK or J2534-2:PEAK followed by the handle of the PEAK device that shall be opened (ex. 0x51).
FULL_DUPLEX/HALF_DUPLEX	YES, but logical channels connected on the same physical channel, cannot mix HALF_DUPLEX and FULL_DUPLEX : in this case, the logical connection ends with an error.

A.1.3   PASSTHRUIOCTL

The following figure lists the supported IOCTL IDs for the function PassThruIOCTL:

IOCTL ID	Supported?
GET_CONFIG	Yes
SET_CONFIG	Yes
FIVE_BAUD_INIT	No
FAST_INIT	No
CLEAR_TX_QUEUE	Yes
CLEAR_RX_QUEUE	Yes
CLEAR_PERIODIC_MSGS	Yes
CLEAR_MSG_FILTERS	Yes
BUS_ON	Yes
CLEAR_FUNCT_MSG_LOOKUP_TABLE	No
ADD_TO_FUNCT_MSG_LOOKUP_TABLE	No
DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE	No
READ_PROG_VOLTAGE	No

READ_PIN_VOLTAGE	No

The following figure lists the supported parameters associated with the IOCTL IDs « GET_CONFIG » and « SET_CONFIG »:

Parameter	Supported ?
DATA_RATE	**Yes**
NODE_ADDRESS	No
NETWORK_LINE	No
P1_MIN	No
P1_MAX	No
P2_MIN	No
P2_MAX	No
P3_MIN	No
P3_MAX	No
P4_MIN	No
P4_MAX	No
W0_MIN	No
W1_MIN	No
W1_MAX	No
W2_MIN	No
W2_MAX	No
W3_MIN	No
W3_MAX	No
W4_MIN	No
W4_MAX	No
W5_MIN	No
TIDLE	No
TINIL	No
TWUP	No
PARITY	No
T1_MAX	No
T2_MIN	No
T3_MAX	No
T4_MAX	No
T5_MIN	No
ISO15765_WAIT_LIMIT	**Yes**
ISO15765_PAD_VALUE	**Yes**
ISO15765_BS	**Yes**
ISO15765_STMIN	**Yes**
N_CS_MIN	No
N_AS_MAX	**Yes**
N_AR_MAX	**Yes**
N_BR_MIN	No
N_BS_MAX	**Yes**
N_CR_MAX	**Yes**
BS_TX	**Yes**
STMIN_TX	**Yes**
DATA_BITS	No
FIVE_BAUD_MOD	No
ECHO_PHYSICAL_CHANNEL_ TX	**Yes**

A.2 SAE J2534-2 Revised JAN2019

The following figure lists the supported features from optional PassThru extension SAE J2534-2:

Feature	Supported?
Enabling SAE J2534-2 features and accessing multiple SAE J2534 devices	Yes
PIN SELECTION	No
ACCESS TO ADDITIONAL CHANNELS	No
MIXED FORMAT FRAMES ON A CAN NETWORK	Yes CAN_MIXED_FORMAT_ALL_FRAMES only
Single Wire CAN	No
Analog Inputs	No
GM UART (SAE J2740)	No
UART Echo Byte Protocol	No
Honda Protocol	No
Repeat Messaging	No
Extended Programming Voltage Support	No
J1939 Protocol	No
J1708 Protocol	No
Extended PassThru IOCTL for Device Configuration Parameters	Yes
TP2.0 Protocol	No
Fault-Tolerant CAN	No
CAN FD protocol	**Yes**
ISO15765 on CAN FD	**Yes**
Discovery mechanism	**Yes**

The following figure lists the supported parameters associated with the IOCTL IDs « GET_CONFIG » and « SET_CONFIG »:

Parameter	Supported ?
J1962_PINS	**Yes**
FD_CAN_DATA_PHASE_RATE	**Yes**
FD_ISO15765_TX_DATA_LENGTH	**Yes**
HS_CAN_TERMINATION	No (but for compatibility reasons, this IOCTL always returns STATUS_OK)
N_CR_MAX	**Yes**
ISO15765_PAD_VALUE	**Yes**
SYNC_JUMP_WIDTH	**Yes**
BIT_SAMPLE_POINT	**Yes**

A.3 GM Extension GMW17753

ℹ **Note:** PEAK-System does not provide documentation about this extension. This must be acquired separately in any store online for standards.

The following table lists the supported features included in GMW17753.

Feature	Supported?
PIN SELECTION (CAN_FD_PS, ISO15765_FD_PS)	No

Feature	Supported?
Protocols: CAN_FD_PS, ISO15765_FD_PS	No
Tx Flags: CAN_FD_BRS, CAN_FD_FORMAT	No
Rx Status: CAN_FD_BRS, CAN_FD_FORMAT, CAN_FD_ESI, ERROR_IND.	No
RX Status: RX_MSG_TRUNCATED	**Yes**
Mixed format with ISO15765_FD	No
Message Extension (DataSize > 4128)	**Yes**
Dynamic Flow CONTROL	No
IOCTL: CAN_FD_PHASE_RATE, CAN_FD_TC_DATA_LENGTH, CAN_FD_TYPE (ISO CAN FD), N_CR_MAX (as GM value)	No
IOCTL: CAN_FD_TYPE (BOSCH CAN FD)	No
IOCTL: CAN_FD_TERMINATION	No
ERROR_IND with 4 bytes data	No

A.4   Supported Bit Rates

The following tables lists the valid bit rate combinations specified in SAE J2535-1, SAE J2535-2 and GMW17753 and the corresponding register value sets.

Nominal Bit Rate	Data Bit Rate (bits per second)				
500 000	500 000	1 000 000	2 000 000	4 000 000	5 000 000 (Default)
250 000	250 000	500 000	1 000 000	2 000 000	2 500 000 (Default)
125 000	125 000	250 000	500 000	1 000 000	1 250 000 (Default)

Bit Rate Combination	Register Value Set								
	f_clock	nom. BRP	nom. TSEQ1	nom. TSEQ2	nom. SJW	data BRP	data TSEQ1	data TSEQ2	data SJW
1 Mbit/s | 1 MBit/s	80000000	20	2	1	1	20	2	1	1
1 Mbit/s | 2 Mbit/s	80000000	10	5	2	2	10	2	1	2
1 Mbit/s | 4 Mbit/s	80000000	5	11	4	4	5	2	1	1
1 Mbit/s | 5 Mbit/s	80000000	4	14	5	5	4	1	2	2
500 kbit/s | 500 kbit/s	80000000	10	12	3	1	4	29	10	1
500 kbit/s | 1 Mbit/s	80000000	10	12	3	1	2	29	10	1
500 kbit/s | 2 Mbit/s	80000000	2	63	16	16	2	15	4	4
500 kbit/s | 4 Mbit/s	80000000	10	12	3	1	1	14	5	1
500 kbit/s | 5 Mbit/s	80000000	1	127	32	32	1	11	4	4
250 kbit/s | 250 kbit/s	80000000	2	139	20	1	8	29	10	1
250 kbit/s | 500 kbit/s	80000000	2	139	20	1	4	29	10	1
250 kbit/s | 1 Mbit/s	80000000	2	139	20	1	2	29	10	1
250 kbit/s | 2 Mbit/s	80000000	2	139	20	1	1	29	10	1
250 kbit/s | 2.5 Mbit/s	80000000	2	139	20	1	1	23	8	1
250 kbit/s | 4 Mbit/s	80000000	5	55	8	8	5	1	2	2
250 kbit/s | 5 Mbit/s	80000000	4	69	10	10	4	1	2	2
125 kbit/s | 125 kbit/s	80000000	4	139	20	1	16	29	10	1
125 kbit/s | 250 kbit/s	80000000	4	139	20	1	8	29	10	1
125 kbit/s | 500 kbit/s	80000000	4	139	20	1	4	29	10	1
125 kbit/s | 1 Mbit/s	80000000	4	139	20	1	2	29	10	1
125 kbit/s | 1.25 Mbit/s	80000000	4	139	20	1	2	23	8	1
125 kbit/s | 2 Mbit/s	80000000	10	55	8	8	10	1	2	2

| 125 kbit/s | 4 Mbit/s | 80000000 | 5 | 111 | 16 | 16 | 5 | 1 | 2 | 2 |
| 125 kbit/s | 5 Mbit/s | 80000000 | 4 | 139 | 20 | 20 | 4 | 1 | 2 | 2 |