SAE International™

SURFACE VEHICLE RECOMMENDED PRACTICE

SAE J2534-1

REV. DEC2004

Issued 2002-02
Revised 2004-12

Superseding J2534 FEB2002

(R) Recommended Practice for Pass-Thru Vehicle Programming

Foreword

The use of reprogrammable memory technology in vehicle electronic control units (ECUs) has increased in recent years, and is expected to continue in the future. Use of this technology has increased the flexibility of being able to use a single ECU hardware part to be used in many different vehicle configurations, with the only difference being the software and calibrations programmed into the unit. Reprogramming of those ECUs in the service environment also allows for ease of field modification of system operation and calibrations. Variations in reprogramming capability and the multiple tools necessary to reprogram vehicles are a burden on aftermarket repair facilities that service different makes of vehicles.

This document describes a standardized system for programming that includes a standard personal computer (PC), standard interface to a software device driver, and an interface that connects between the PC and a programmable ECU in a vehicle. The purpose of this system is to facilitate programming of ECUs for all vehicle manufacturers using a single set of programming hardware. Programming software from multiple vehicle manufacturers will be able to execute on this set of hardware to program their unique ECUs.

The U.S. Environmental Protection Agency (EPA) and the California Air Resources Board (ARB) have been working with vehicle manufacturers to provide the aftermarket with increased capability to service emission-related ECUs for all vehicles with a minimal investment in hardware needed to communicate with the vehicles. Both agencies have issued regulations that will require standardized programming tools to be used for all vehicle manufacturers. The Society of Automotive Engineers (SAE) developed this Recommended Practice to satisfy the intent of the U.S. EPA and the California ARB.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org
SAE WEB ADDRESS: **http://www.sae.org**

TABLE OF CONTENTS

1. Scope

This SAE Recommended Practice provides the framework to allow reprogramming software applications from all vehicle manufacturers the flexibility to work with multiple vehicle data link interface tools from multiple tool suppliers. This system enables each vehicle manufacturer to control the programming sequence for electronic control units (ECUs) in their vehicles, but allows a single set of programming hardware and vehicle interface to be used to program modules for all vehicle manufacturers.

This document does not limit the hardware possibilities for the connection between the PC used for the software application and the tool (e.g., RS-232, RS-485, USB, Ethernet…). Tool suppliers are free to choose the hardware interface appropriate for their tool. The goal of this document is to ensure that reprogramming software from any vehicle manufacturer is compatible with hardware supplied by any tool manufacturer.

U.S. Environmental Protection Agency (EPA) and the California Air Resources Board (ARB) "OBD service information" regulations include requirements for reprogramming emission-related control modules in vehicles for all manufacturers by the aftermarket repair industry. This document is intended to conform to those regulations for 2004 and later model year vehicles. For some vehicles, this interface can also be used to reprogram emission-related control modules in vehicles prior to the 2004 model year, and for non-emission related control modules. For other vehicles, this usage may require additional manufacturer specific capabilities to be added to a fully compliant interface. A second part to this document, SAE J2534-2, is planned to include expanded capabilities that tool suppliers can optionally include in an interface to allow programming of these additional non-mandated vehicle applications. In addition to reprogramming capability, this interface is planned for use in OBD compliance testing as defined in SAE J1699-3. SAE J2534-1 includes some capabilities that are not required for Pass-Thru Programming, but which enable use of this interface for those other purposes without placing a significant burden on the interface manufacturers.

Additional requirements for future model years may require revision of this document, most notably the inclusion of SAE J1939 for some heavy-duty vehicles. This document will be reviewed for possible revision after those regulations are finalized and requirements are better understood. Possible revisions include SAE J1939 specific software and an alternate vehicle connector, but the basic hardware of an SAE J2534 interface device is expected to remain unchanged.

2. References

2.1 Applicable Publications

The following publications form a part of this specification to the extent specified herein. Unless otherwise indicated, the latest version of SAE publications shall apply.

2.1.1 SAE PUBLICATIONS

Available from SAE, 400 Commonwealth Drive, Warrendale, PA 15096-0001.

SAE J1850—Class B Data Communications Network Interface
SAE J1939—Truck and Bus Control and Communications Network (Multiple Parts Apply)
SAE J1962—Diagnostic Connector
SAE J2610—DaimlerChrysler Information Report for Serial Data Communication Interface (SCI)

2.1.2 ISO DOCUMENTS

Available from ANSI, 25 west 43rd Street, New York, NY 10036-8002.

ISO 7637-1:1990—Road vehicles—Electrical disturbance by conduction and coupling—Part 1: Passenger cars and light commercial vehicles with nominal 12 V supply voltage
ISO 9141:1989—Road vehicles—Diagnostic systems—Requirements for interchange of digital information
ISO 9141-2:1994—Road vehicles—Diagnostic systems—CARB requirements for interchange of digital information
ISO 11898:1993—Road vehicles—Interchange of digital information—Controller area network (CAN) for high speed communication
ISO 14230-4:2000—Road vehicles—Diagnostic systems—Keyword protocol 2000—Part 4: Requirements for emission-related systems
ISO/FDIS 15765-2—Road vehicles—Diagnostics on controller area networks (CAN)—Network layer services
ISO/FDIS 15765-4—Road vehicles—Diagnostics on controller area networks (CAN)—Requirements for emission-related systems

3. Definitions

3.1 Registry

A mechanism within Win32 operating systems to handle hardware and software configuration information.

4. Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CRC	Cyclic Redundancy Check
DLL	Dynamic Link Library
ECU	Electronic Control Unit
IFR	In-Frame Response
IOCTL	Input / Output Control
KWP	Keyword Protocol
OEM	Original Equipment Manufacturer
PC	Personal Computer
PWM	Pulse Width Modulation
SCI	Serial Communications Interface
SCP	Standard Corporate Protocol
USB	Universal Serial Bus
VPW	Variable Pulse Width

5. Pass-Thru Concept

Programming application software supplied by the vehicle manufacturer will run on a commonly available generic PC. This application must have complete knowledge of the programming requirements for the control module to be programmed and will control the programming event. This includes the user interface, selection criteria for downloadable software and calibration files, the actual software and calibration data to be downloaded, the security mechanism to control access to the programming capability, and the actual programming steps and sequence required to program each individual control module in the vehicle. If additional procedures must be followed after the reprogramming event, such as clearing Diagnostic Trouble Codes (DTC), writing part numbers or variant coding information to the control module, or running additional setup procedures, the vehicle manufacturer must either include this in the PC application or include the necessary steps in the service information that references reprogramming.

This document defines the following two interfaces for the SAE J2534 pass-thru device:

a. Application program interface (API) between the programming application running on a PC and a software device driver for the pass-thru device
b. Hardware interface between the pass-thru device and the vehicle

The manufacturer of an SAE J2534 pass-thru device shall supply connections to both the PC and the vehicle. In addition to the hardware, the interface manufacturer shall supply device driver software, and a Windows installation and setup application that will install the manufacturer's SAE J2534 DLL and other required files, and also update the Windows Registry. The interface between the PC and the pass-thru device can be any technology chosen by the tool manufacturer, including RS-232, RS-485, USB, Ethernet, or any other current or future technology, including wireless technologies.

All programming applications shall utilize the common SAE J2534 API as the interface to the pass-thru device driver. The API contains a set of routines that may be used by the programming application to control the pass-thru device, and to control the communications between the pass-thru device and the vehicle. The pass-thru device will not interpret the message content, allowing any message strategy and message structure to be used that is understood by both the programming application and the ECU being programmed. Also, because the message will not be interpreted, the contents of the message cannot be used to control the operation of the interface. For example, if a message is sent to the ECU to go to high speed, a specific instruction must also be sent to the interface to go to high speed.

The OEM programming application does not need to know the hardware connected to the PC, which gives the tool manufacturers the flexibility to use any commonly available interface to the PC. The pass-thru device does not need any knowledge of the vehicle or control module being programmed. This will allow all programming applications to work with all pass-thru devices to enable programming of all control modules for all vehicle manufacturers.

Figure 1 shows the relationship between the various components required for pass-thru programming and responsibilities for each component:

Vehicle	Cable	Pass-thru interface	Cable	Programming PC		
				Interface driver	API	Application
Vehicle manufacturer determines programming sequence and security access	Tool supplier defines interface with pass-thru device SAE J1962 on vehicle end 5 meter max. length	Capabilities defined in SAE J2534 Hardware supplied by tool supplier Could be implemented in scan tool	Tool supplier defines cable requirements, if any, between PC and interface	Tool supplier device driver, installation, and setup procedure based on hardware supported Examples are: RS-232, USB, PCMCIA, Ethernet, IEEE1394, Bluetooth	SAE J2534	Vehicle manufacturer programming application controls user interface, vehicle software and calibration selection, programming sequence, and security access
Vehicle manufacturer	Tool supplier				SAE J2534	Vehicle manufacturer

FIGURE 1—SAE J2534 OVERVIEW

6. Pass-Thru System Requirements

A "fully" compliant SAE J2534 interface shall support all communication protocols and capabilities defined in this document. The interface's registry entry "Protocols Supported" (see section 9.2-Win32 Registry) specifies the protocols fully supported by the interface.

6.1 PC Requirements

Generic PC running a Win32 Operating System (e.g., Windows 95/Windows 98/Windows NT/Windows Millennium Edition, Windows 2000, Windows XP, …). The PC should be capable of connection to the Internet.

6.2 Software Requirements and Assumptions

Reprogramming applications can assume that the PC will be connected to the Internet, although not all applications will require this. The OEM application is limited to a single thread for communication with the tool manufacturer DLL/API. Multiple protocols may be connected and used from the single application thread (see Section 6.6).

The interface will not handle the tester present messages automatically. The OEM application is responsible to handle tester present messages.

6.3 Connection to PC

The interface between the PC and the pass-thru device shall be determined by the manufacturer of the pass-thru device. This can be RS-232, USB, Ethernet, IEEE1394, Bluetooth or any other connection that allows the pass-thru device to meet all other requirements of this document, including timing requirements. The tool manufacturer is also required to include the device driver that supports this connection so that the actual interface used is transparent to both the PC programming application and the vehicle.

6.4 Connection to Vehicle

The interface between the pass-thru device and the vehicle shall be an SAE J1962 connector for serial data communications. The maximum cable length between the pass-thru device and the vehicle is five (5) meters. The interface shall include an insulated banana jack that accepts a standard 0.175" diameter banana plug as the auxiliary pin for connection of programming voltage to a vehicle specific connector on the vehicle.

If powered from the vehicle, the interface shall:

a. operate normally within a vehicle battery voltage range of 8.0 to 18.0 volts D.C.,
b. survive a vehicle battery voltage of up to 24.0 volts D.C. for at least 10 minutes,
c. survive, without damage to the interface, a reverse vehicle battery voltage of up to 24.0 volts D.C. for at least 10 minutes.

6.5 Communication Protocols

The following communication protocols shall be supported:

6.5.1 ISO 9141

The following specifications clarify and, if in conflict with ISO 9141, override any related specifications in ISO 9141:

a. The maximum sink current to be supported by the interface is 100 mA.
b. The range for all tests performed relative to ISO 7637-1 is –1.0 to +40.0 V.
c. The default bus idle period before the interface shall transmit an address, shall be 300 ms.
d. Support following baud rate with ±0.5% tolerance: 10400.
e. Support following baud rate with ±1% tolerance: 10000.
f. Support following baud rates with ±2% tolerance: 4800, 9600, 9615, 9800, 10870, 11905, 12500, 13158, 13889, 14706, 15625, and 19200.
g. Support other baud rates if the interface is capable of supporting the requested value within ±2%.
h. The baud rate shall be set by the application, not determined by the SAE J2534 interface. The interface is not required to support baud rate detection based on the synchronization byte.
i. Support odd and even parity in addition to the default of no parity, with seven or eight data bits. Always one start bit and one stop bit.

j. Support for timer values that are less than or greater than those specified in ISO 9141 (see Figure 30 in Section 7.3.2).
k. Support ability to disable automatic ISO 9141-2 / ISO 14230 checksum verification by the interface to allow vehicle manufacturer specific error detection.
l. If the ISO 9141 checksum is verified by the interface, and the checksum is incorrect, the message will be discarded.
m. Support both ISO 9141 5-baud initialization and ISO 14230 fast initialization.
n. Interface shall not adjust timer parameters based on keyword values.

6.5.2 ISO 14230-4 (KWP2000)

The ISO 14230 protocol has the same specifications as the ISO 9141 protocol as outlined in the previous section. In addition, the following specifications clarify and, if in conflict with ISO 14230, override any related specifications in ISO 14230:

a. The pass-thru interface will not automatically handle tester present messages. The application needs to handle tester present messages when required.
b. The pass-thru interface will not perform any special handling for the $78 response code. Any message received with a $78 response code will be passed from the interface to the application. The application is required to handle any special timing requirements based on receipt of this response code, including stopping any periodic messages.

6.5.3 SAE J1850 41.6 KBPS PWM (PULSE WIDTH MODULATION)

The following additional features of SAE J1850 must be supported by the pass-thru device:

a. Capable of 41.6 kbps and high speed mode of 83.3 kbps.
b. Recommend Ford approved SAE J1850PWM (SCP) physical layer

6.5.4 SAE J1850 10.4 KBPS VPW (VARIABLE PULSE WIDTH)

The following additional features of SAE J1850 must be supported by the pass-thru device:

a. Capable of 10.4 kbps and high speed mode of 41.6 kbps
b. 4128 byte block transfer
c. Return to normal speed after a break indication

6.5.5 CAN

The following features of ISO 11898 (CAN) must be supported by the pass-thru device:

a. 125, 250, and 500 kbps
b. 11 and 29 bit identifiers
c. Support for 80% ± 2% and 68.5% ± 2% bit sample point
d. Allow raw CAN messages. This protocol can be used to handle any custom CAN messaging protocol, including custom flow control mechanisms.

6.5.6 ISO 15765-4 (CAN)

The following features of ISO 15765-4 must be supported by the pass-thru device:

a. 125, 250, and 500 kbps
b. 11 and 29 bit identifiers
c. Support for 80% ± 2% bit sample point
d. To maintain acceptable programming times, the transport layer flow control function, as defined in ISO 15765-2, must be incorporated in the pass-thru device (see Appendix A). If the application does not use the ISO 15765-2 transport layer flow control functionality, the CAN protocol will allow for any custom transport layer.
e. Receive a multi-frame message with an ISO15765_BS of 0 and an ISO15765_STMIN of 0, as defined in ISO 15765-2.
f. No single frame or multi-frame messages can be received without matching a flow control filter. No multi-frame messages can be transmitted without matching a flow control filter.
g. Periodic messages will not be suspended during transmission or reception of a multi-frame segmented message.

6.5.7 SAE J2610 DAIMLERCHRYSLER SCI

Reference the SAE J2610 Information Report for a description of the SCI protocol.

When in the half-duplex mode (when SCI_MODE of TxFlags is set to {1} Half-Duplex), every data byte sent is expected to be "echoed" by the controller. The next data byte shall not be sent until the echo byte has been received and verified. If the echoed byte received doesn't match the transmitted byte, or if after a period of T1 no response was received, the transmission will be terminated. Matching echoed bytes will not be placed in the receive message queue.

6.6 Simultaneous Communication On Multiple Protocols

The pass-thru device must be capable of supporting simultaneous communication on multiple protocols during a single programming event. Figure 2 indicates which combinations of protocols shall be supported. If SCI (SAE J2610) communication is not required during the programming event, the interface shall be capable of supporting one of the protocols from data link set 1, data link set 2, and data link set 3. If SCI (SAE J2610) communication is required during the programming event, the interface shall be capable of supporting one of the SCI protocols and one protocol from data link set 1.

	DATA LINK SET 1	DATA LINK SET 2	DATA LINK SET 3
Without SCI	SAE J1850 VPW SAE J1850 PWM	ISO 9141 ISO 14230	CAN ISO 15765
With SCI	SAE J1850 VPW SAE J1850 PWM	SCI A SCI B	None

FIGURE 2—SIMULTANEOUS COMMUNICATION OPTIONS

6.7 Programmable Power Supply

The interface shall be capable of supplying between 5 and 20 volts to one of the following pins (6, 9, 11, 12, 13 or 14) on the SAE J1962 diagnostic connector, or to an auxiliary pin which would need to be connected to the vehicle via a cable that is unique to the vehicle. The auxiliary pin on the interface shall be a female banana jack (see Section 6.4- Connection to Vehicle). As well, short to ground capability on pin 15 is required. The following requirements shall be met by the power supply:

a. Minimum 5 V DC
b. Maximum 20 V DC
c. Resolution 0.1V DC
d. Accuracy ±2% of requested voltage
e. Maximum source current 150 mA
f. Maximum sink current 300mA (only for SHORT_TO_GROUND on pin 15).
g. Maximum 1 ms settling time (required for SCI protocol only, reference SAE J2610 Information Report)
h. Pin assignment software selectable

6.8 Pin Usage

Figure 3 indicates the possible uses for each pin of the SAE J1962 connector and for the auxiliary pin. This figure also indicates the default condition for each pin, which is the required condition when the interface is connected to the vehicle, and the condition to return to when the pin is no longer used to supply programming voltage, short to ground, or serial data communication. For the following table, high impedance is defined as greater than 500 kΩ impedance relative to signal ground, and as greater than 500 kΩ impedance relative to chassis ground.

PIN NO.	Possible Uses	Default
Aux	Auxiliary programming voltage (not part of SAE J1962 connector)	High impedance
1		High impedance
2	SAE J1850 (+)	SAE J1850 (+)
3		High impedance
4	Chassis Ground	Chassis Ground
5	Signal Ground	Signal Ground
6	ISO 15765-4/CAN High Programming Voltage SCI A Engine (Rx)	High impedance
7	ISO 9141/ ISO 14230 K-line SCI A engine (Tx) SCI A Trans (Tx) SCI B Engine (Tx)	High impedance
8		High impedance
9	SCI B Trans (Rx) Programming Voltage	High impedance
10	SAE J1850 (-)	SAE J1850 (-)
11	Programming Voltage	High impedance
12	SCI B engine (Rx) Programming Voltage	High impedance
13	Programming Voltage	High impedance
14	ISO 15765-4/ CAN Low Programming Voltage SCI A Trans (Tx)	High impedance
15	ISO 9141/ ISO 14230 L-line Short to Ground SCI B Trans (Tx)	High impedance
16	Unswitched battery voltage	Unswitched battery voltage

FIGURE 3—PIN USAGE

6.9 Data Buffering

The interface/API shall be capable of receiving 8 simultaneous messages. For ISO 15765 these can be multi-frame messages. The interface/API shall be capable of buffering a maximum length (4128 byte) transmit message and a maximum length (4128 byte) receive message.

6.10 Error Recovery

6.10.1 Device Not Connected

If the DLL returns ERR_DEVICE_NOT_CONNECTED from any function, that error shall continue to be returned by all functions, even if the device is reconnected. An application can recover from this error condition by closing the device (with PassThruClose) and re-opening the device (with PassThruOpen, getting a new device ID).

6.10.2 Bus Errors

All devices shall handle bus errors in a consistent manner. There are two error strategies: Retry and Drop.

The Retry strategy will keep trying to send a packet until successful or stopped by the application. If loopback is on and the message is successfully sent after some number of retries, only one copy of the message shall be placed in the receive queue. Even if the hardware does not support retries, the firmware/software must retry the transmission. If the error condition persists, a blocking write will wait the specified timeout and return ERR_TIMEOUT. The DLL must return the number of successfully transmitted messages in pNumMsgs. The DLL shall not count the message being retried in pNumMsgs. After returning from the function, the device does not stop the retries. The only functions that will stop the retries are PassThruDisconnect (on that protocol), PassThruClose, or PassThruIoctl (with an IoctlID of CLEAR_TX_BUFFER).

Devices shall use the Retry strategy in the following scenarios:

- All CAN errors, such as bus off, lack of acknowledgement, loss of arbitration, and no connection (lack of terminating resistor)
- SAE J1850PWM or SAE J1850VPW bus fault (bus stuck passive) or loss of arbitration (bus stuck active)

The Drop strategy will delete a message from the queue. The message can be dropped immediately on noticing an error or at the end of the transmission. PassThruWriteMsg shall treat dropped messages the same as successfully transmitted messages. However, if loopback is on, the message shall not be placed in the receive queue.

Devices shall use the Drop strategy in the following scenarios:

- If characters are echoed improperly in SCI
- Corrupted ISO 9141 or ISO 14230 transmission
- SAE J1850PWM lack of acknowledgement (Exception: The device must try sending the message 3 times before dropping)

7. Win32 Application Programming Interface

7.1 API Functions – Overview

To conform to this document a vendor supplied API implementation (DLL) must support the functions included in Figure 4.

Function	Description
PassThruOpen	Establish a connection with a Pass-Thru device.
PassThruClose	Terminate a connection with a Pass-Thru device.
PassThruConnect	Establish a connection with a protocol channel.
PassThruDisconnect	Terminate a connection with a protocol channel.
PassThruReadMsgs	Read message(s) from a protocol channel.
PassThruWriteMsgs	Write message(s) to a protocol channel.
PassThruStartPeriodicMsg	Start sending a message at a specified time interval on a protocol channel.
PassThruStopPeriodicMsg	Stop a periodic message.
PassThruStartMsgFilter	Start filtering incoming messages on a protocol channel.
PassThruStopMsgFilter	Stops filtering incoming messages on a protocol channel.
PassThruSetProgrammingVoltage	Set a programming voltage on a specific pin.
PassThruReadVersion	Reads the version information for the DLL and API.
PassThruGetLastError	Gets the text description of the last error.
PassThruIoctl	General I/O control functions for reading and writing protocol configuration parameters (e.g. initialization, baud rates, programming voltages, etc.).

FIGURE 4—SAE J2534 API FUNCTIONS

7.2 API Functions – Detailed Information

7.2.1 PASSTHRUOPEN

This function is used to establish a connection and intialize the Pass-Thru Device. This function must be called one time before any other function with the exception of PassThruGetLastError. Any function called before a successul call to PassThruOpen must return ERR_INVALID_DEVICE_ID. If the function is successful, a value of STATUS_NOERROR is returned. The Device ID returned is used as a handle to the initialized SAE J2534 device.

7.2.1.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruOpen
(
        void *pName
        unsigned long *pDeviceID
)
```

7.2.1.2 Parameters

pName Must be NULL (reserved for future use with multiple devices).

pDeviceID Pointer to location for the device ID that is assigned by the DLL.

7.2.1.3 Return Values – See Figure 5

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with the device
ERR_DEVICE_IN_USE	Device is currently open
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required
ERR_FAILED	Undefined error, use PassThruGetLastError for text description

FIGURE 5—RETURN VALUES

7.2.2 PASSTHRUCLOSE

This function is used to close the connection to a Pass-Thru Device. All periodic messages will be stopped, filters will be cleared, and all pins will return to their default state (see Section 6.8). This function must be called before an application exits. The DLL can use this function to de-allocate data structures and deactivate any device drivers. If the function is successful, a value of STATUS_NOERROR is returned. After this call, all active protocols will be disconnected, Channel Ids will no longer be valid, and any function call other than PassThruOpen will result in the error ERR_INVALID_DEVICE_ID being returned (with the exception of PassThruGetLastError).

7.2.2.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruClose
(
    unsigned long DeviceID
)
```

7.2.2.2 Parameters

DeviceID DeviceID returned from PassThruOpen

7.2.2.3 Return Values – See Figure 6

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device.
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_FAILED	Undefined error, use PassThruGetLastError for text description

FIGURE 6—RETURN VALUES

7.2.3 PASSTHRUCONNECT

This function is used to establish a logical connection with a protocol channel on the specified SAE J2534 device. After this function is called, the value pointed to by pChannelID is used as the logical identifier for the combination of Device ID and Protocol ID. If the function is successful, a value of STATUS_NOERROR is returned and a valid channel ID will be placed in <pChannelID>. All future interactions with the protocol channel will be done using the pChannelID. Note that the interface will block all received messages on this channel until a filter is set.

7.2.3.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruConnect
(
        unsigned long DeviceID,
        unsigned long ProtocolID,
        unsigned long Flags,
        unsigned long BaudRate,
        unsigned long *pChannelID
)
```

7.2.3.2 Parameters

DeviceID Device ID returned from PassThruOpen

ProtocolID Protocol ID,

Flags Connection flags,

BaudRate Initial baud rate

pChannelID Pointer to location for the channel ID that is assigned by the DLL.

7.2.3.3 Connect Flag Values – See Figure 7

Definition	Flags Bit(s)	Description	Value
	31-24	Unused	Tool manufacturer specific- shall be set to 0
	23-16	Unused	Reserved for SAE J2534-2-shall be set to 0
	15-13	Unused	Reserved for SAE - shall be set to 0
ISO9141_K_LINE_ONLY	12	L line usage for ISO9141 and ISO14230 Initialization address	0 = use L-line and K-line for initialization address 1 = use K-line only line for initialization address
CAN_ID_BOTH	11	CAN ID support type for CAN and ISO 15765 (also see bit 8)	0 = either standard or extended CAN ID types used – CAN ID type defined by bit 8 1 = both standard and extended CAN ID types used – if the CAN controller allows prioritizing either standard (11 bit) or extended (29 bit) CAN ID's then bit 8 will determine the higher priority ID type
	10	Unused	Reserved for SAE - shall be set to 0
ISO9141_NO_CHECKSUM	9	Checksum control for ISO9141 and ISO14230	0 = The interface will generate and append the checksum as defined in ISO 9141-2 and ISO 14230-2 for transmitted messages, and verify the checksum for received messages. 1 = The interface will not generate and verify the checksum-the entire message will be treated as data by the interface
CAN_29BIT_ID	8	CAN ID type for CAN and ISO 15765 (also see bit 11)	0 = Receive standard CAN ID (11 bit) 1 = Receive extended CAN ID (29 bit)
	7	Unused	Reserved for SAE- shall be set to 0

FIGURE 7—FLAG VALUES

7.2.3.4 Protocol ID Values – See Figure 8

Definition	Description	Value(s)
J1850VPW	GM / DaimlerChrysler CLASS2	0x01
J1850PWM	Ford SCP	0x02
ISO9141	ISO 9141 and ISO 9141-2	0x03
ISO14230	ISO 14230-4 (Keyword Protocol 2000)	0x04
CAN	Raw CAN (flow control not handled automatically by interface)	0x05
ISO15765	ISO 15765-2 flow control enabled (see Appendix A for high level description)	0x06
SCI_A_ENGINE	SAE J2610 (DaimlerChrysler SCI) configuration A for engine	0x07
SCI_A_TRANS	SAE J2610 (DaimlerChrysler SCI) configuration A for transmission	0x08
SCI_B_ENGINE	SAE J2610 (DaimlerChrysler SCI) configuration B for engine	0x09
SCI_B_TRANS	SAE J2610 (DaimlerChrysler SCI) configuration B for transmission	0x0A
Reserved	Reserved for SAE use	0x0B – 0x7FFF
Reserved	Reserved for SAE J2534-2	0x8000 - 0xFFFF
Unused	Tool manufacturer specific	0x10000 – 0xFFFFFFFF

FIGURE 8—PROTOCOL ID VALUES

7.2.3.5 Return Values – See Figure 9

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_NOT_SUPPORTED	Device cannot support a protocol, or a particular (requested) flag on a protocol mandated by this document. Device is not fully SAE J2534 complaint
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_PROTOCOL_ID	Invalid ProtocolID value, unsupported ProtocolID, or there is a resource conflict (i.e. trying to connect to multiple protocols that are mutually exclusive such as J1850PWM and J1850VPW or CAN and SCI_A, etc.).
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required
ERR_INVALID_FLAGS	Invalid flag values.
ERR_INVALID_BAUDRATE	The desired baud rate cannot be achieved within the tolerance specified in Section 6.5
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_CHANNEL_IN_USE	Channel number is currently connected.

FIGURE 9—RETURN VALUES

7.2.4 PASSTHRUDISCONNECT

This function is used to terminate a logical connection with a protocol channel. If the function is successful, a value of STATUS_NOERROR is returned. After this call, all filters associated with the channel will be cleared, all periodic messages associated with the channel will be stopped, the associated pins will return to their default state (see Section 6.8) and the Channel ID will no longer be valid.

7.2.4.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruDisconnect
(
        unsigned long ChannelID
)
```

7.2.4.2 Parameters

ChannelID The channel ID assigned by the PassThruConnect function.

7.2.4.3 Return Values – See Figure 10

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device.
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_FAILED	Undefined error, use PassThruGetLastError for text description.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.

FIGURE 10—RETURN VALUES

7.2.5 PASSTHRUREADMSGS

This function reads messages and indications from the receive buffer. All messages and indications shall be read in the order that they occurred on the bus. If a transmit message generated a loopback message and TxDone indication, the TxDone indication shall always be queued first. Except for loopback messages and indications, no messages shall be queued for reception without matching a PASS_FILTER (for non-ISO 15765) or FLOW_CONTROL filter (for ISO 15765). On ISO 15765, PCI bytes are transparently removed by the API. If the function is successful, a value of STATUS_NOERROR is returned.

Section 8.3 shows the formatting of messages and indications in the PASSTHRU_MSG structure. Section 8.7.2 shows the valid combinations of the RxStatus bits for the messages and indications to be returned to the application. For each protocol, this function receives the indications from the interface as shown in Figure 11.

Message/ Indication	Protocol ID						
	ISO 9141	ISO 14230	SAE J1850 PWM	SAE J1850 VPW	CAN	ISO 15765-4	SAE J2610 (SCI)
Normal Message	X	X	X	X	X	X	X
RxStart Indication	X	X				X	
RxBreak Indication				X			X
TxDone Indication						X	
Loopback Message (if enabled)	X	X	X	X	X	X	X

FIGURE 11—INDICATIONS

7.2.5.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruReadMsgs
(
        unsigned long ChannelID,
        PASSTHRU_MSG *pMsg,
        unsigned long *pNumMsgs,
        unsigned long Timeout
)
```

7.2.5.2 Parameters

ChannelID	The channel ID assigned by the PassThruConnect function.
pMsg	Pointer to message structure(s).
pNumMsgs	Pointer to location where number of messages to read is specified. On return from the function this location will contain the actual number of messages read.
Timeout	Read timeout (in milliseconds). If a value of 0 is specified the function retrieves up to pNumMsgs messages and returns immediately. Otherwise, the API will not return until the Timeout has expired, an error has occurred, or the desired number of messages have been read. If the number of messages requested have been read, the function shall not return ERR_TIMEOUT, even if the timeout value is zero.

7.2.5.3 Return Values – See Figure 12

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_TIMEOUT	Timeout. Device could not read the specified number of messages. The actual number of messages read is placed in <NumMsgs>. If a timeout occurs and there are no available messages, ERR_BUFFER_EMPTY must be returned.
ERR_BUFFER_EMPTY	Protocol message buffer empty, no messages available to read.
ERR_NO_FLOW_CONTROL	No flow control filter set or matched (for protocolID ISO15765 only).
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_BUFFER_OVERFLOW	Indicates a buffer overflow occurred and messages were lost. The actual number of messages read is placed in <NumMsgs>.

FIGURE 12—RETURN VALUES

7.2.6 PASSTHRUWRITEMSGS

This function is used to send messages. The messages are placed in the buffer and sent in the order they were received. Only one message per protocol can be in transmission at a time (with one exception-See PassThruStartPeriodicMsg). If the function is successful, a value of STATUS_NOERROR is returned. Specifying a non-zero Timeout performs a blocking write. When using blocking writes, this function does not return until all messages are successfully sent on the vehicle network, or the timeout has expired or an error occurs.

Messages must follow the format specified in Section 8.3. The interface shall not modify structures pointed to by pMsg. Note that some protocols will generate indications when transmitting (See PassThruReadMsg).

When using the ISO 15765-4 protocol, only SingleFrame messages can be transmitted without a matching flow control filter. Also, PCI bytes are transparently added by the API. See PassThruStartMsgFilter and Appendix A for a discussion of flow control filters.

7.2.6.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruWriteMsgs
(
        unsigned long ChannelID,
        PASSTHRU_MSG *pMsg,
        unsigned long *pNumMsgs,
        unsigned long Timeout
)
```

7.2.6.2 Parameters

ChannelID — The channel ID assigned by the PassThruConnect function.

pMsg — Pointer to message structure(s).

pNumMsgs — Pointer to the location where number of messages to write is specified. On return will contain the actual number of messages that were transmitted (when Timeout is non-zero) or placed in the transmit queue (when Timeout is zero).

Timeout — Write timeout (in milliseconds). When a value of 0 is specified, the function queues as many of the specified messages as possible and returns immediately. When a value greater than 0 is specified, the function will block until the Timeout has expired, an error has occurred, or the desired number of messages have been transmitted on the vehicle network. Even if the device can buffer only one packet at a time, this function shall be able to send an arbitrary number of packets if a Timeout value is supplied. Since the function returns early if all the messages have been sent, there is normally no penalty for having a large timeout (several seconds). If the number of messages requested have been written, the function shall not return ERR_TIMEOUT, even if the timeout value is zero.

When an ERR_TIMEOUT is returned, only the number of messages that were sent on the vehicle network is known. The number of messages queued is unknown. Application writers should avoid this ambiguity by using a Timeout value large enough to work on slow devices and networks with arbitration delays.

7.2.6.3 Return Values – See Figure 13

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_NOT_SUPPORTED	Device cannot support a particular (requested) flag on a protocol mandated by this document. Device is not fully SAE J2534 compliant. Example: Requesting SCI TX VOLTAGE on a device without the capability
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by pMsg (Reference Section 8 Message Structure).
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_TIMEOUT	Timeout. Device could not write the specified number of messages. The actual number of messages sent on the vehicle network is placed <NumMsgs>. Only applies when Timeout is non-zero.
ERR_MSG_PROTOCOL_ID	Protocol type in the message does not match the protocol associated with the ChannelID
ERR_NO_FLOW_CONTROL	Multi-segment transmission without matching flow control filter set (for protocolID ISO15765 only).
ERR_BUFFER_FULL	Protocol message buffer is full. Some messages could not be queued. Only applies when Timeout is zero.

FIGURE 13—RETURN VALUES

7.2.7 PASSTHRUSTARTPERIODICMSG

This function will immediately queue the specified message for transmission, and repeat at the specified interval. Periodic messages are limited in length to a single frame message of 12 bytes or less, including header or CAN ID. Periodic messages shall have priority over messages queued with PassThruWriteMsgs, but periodic messages must not violate bus idle timing parameters (e.g. P3_MIN). Periodic messages shall generate TxDone indications (ISO 15765) and loopback messages (on any protocol, if enabled). On ISO 15765, periodic messages can be sent during a multi-frame transmission or reception. If the function is successful, a value of STATUS_NOERROR is returned. The Pass-Thru device must support a minimum of ten periodic messages.

PassThruDisconnect shall delete all periodic messages on that channel. PassThruClose shall delete all periodic messages on all channels for the device. All periodic messages will be stopped on a PassThruDisconnect for the associated protocol or a PassThruClose for the device.

7.2.7.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruStartPeriodicMsg
(
        unsigned long ChannelID,
        PASSTHRU_MSG *pMsg,
        unsigned long *pMsgID,
        unsigned long TimeInterval
)
```

7.2.7.2 Parameters

ChannelID	The channel ID assigned by the PassThruConnect function.
pMsg	Pointer to message structure.
pMsgID	Pointer to location for the message ID that is assigned by the DLL.
TimeInterval	Time interval between the start of successive transmissions of this message, in milliseconds. The valid range is 5-65535 milliseconds.

7.2.7.3 Return Values – See Figure 14

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_NOT_SUPPORTED	Device cannot support requested functionality mandated by this document. Device is not fully SAE J2534 compliant Example: Requesting 20ms interval, but hardware only supports 50ms intervals
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by pMsg. (Reference Section 8 Message Structure)
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_INVALID_TIME_INTERVAL	Invalid TimeInterval value.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_MSG_PROTOCOL_ID	Protocol type in the message does not match the protocol associated with the ChannelID
ERR_EXCEEDED_LIMIT	Exceeded the maximum number of periodic message IDs or the maximum allocated space.

FIGURE 14—RETURN VALUES

7.2.8 PASSTHRUSTOPPERIODICMSG

This function stops the specified periodic message. If the function is successful, a value of STATUS_NOERROR is returned. After this call the MsgID will be invalid.

Note that periodic messages that have been queued for transmission or that are in the process of being transmitted might not be stopped by this function.

7.2.8.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruStopPeriodicMsg
(
        unsigned long ChannelID,
        unsigned long MsgID
)
```

7.2.8.2 Parameters

ChannelID The channel ID assigned by the PassThruConnect function.

MsgID Message ID that is assigned by the PassThruStartPeriodicMsg function.

7.2.8.3 Return Values – See Figure 15

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device.
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description.
ERR_INVALID_MSG_ID	Invalid MsgID value.

FIGURE 15—RETURN VALUES

7.2.9 PASSTHRUSTARTMSGFILTER

This function starts filtering of incoming messages. If the function is successful, a value of STATUS_NOERROR is returned. A minimum of ten message filters shall be supported by the interface for each supported protocol. PassThruDisconnect shall delete all message filters on that channel. PassThruClose shall delete all filters on all channels for the device. Pattern and Mask messages shall follow the protocol formats specified in Section 8. However, only the first twelve (12) bytes, including header or CAN ID, are used by the filter. ERR_INVALID_MSG shall be returned if the filter length exceeds 12. Note that this function does not clear any messages that may have been received and queued before the filter was set. Users are cautioned to consider performing a CLEAR_RX_BUFFER after starting a message filter to be sure that unwanted frames are purged from any receive buffers.

For all protocols except ISO 15765:

- PASS_FILTERs and BLOCK_FILTERs will be applied to all received messages. They shall not be applied to indications or loopback messages
- FLOW_CONTROL_FILTERs must not be used and shall cause the interface to return ERR_INVALID_FILTER_ID
- Both pMaskMsg and pPatternMsg must have the same DataSize and TxFlags. Otherwise, the interface shall return ERR_INVALID_MSG
- The default filter behavior after PassThruConnect is to block all messages, which means no messages will be placed in the receive queue until a PASS_FILTER has been set. Messages that match a PASS_FILTER can still be blocked by a BLOCK_FILTER
- Figure 16 and Figure 17 show how the message filtering mechanism operates

PASS Filter	BLOCK Filter	Message Matches PASS Filter	Message Matches BLOCK Filter	Action
No	No	N/A	N/A	Block
Yes	No	No	N/A	Block
Yes	No	Yes	N/A	Pass
No	Yes	N/A	No	Block
No	Yes	N/A	Yes	Block
Yes	Yes	No	No	Block
Yes	Yes	No	Yes	Block
Yes	Yes	Yes	No	Pass
Yes	Yes	Yes	Yes	Block

FIGURE 16—MESSAGE FILTER ACTIONS

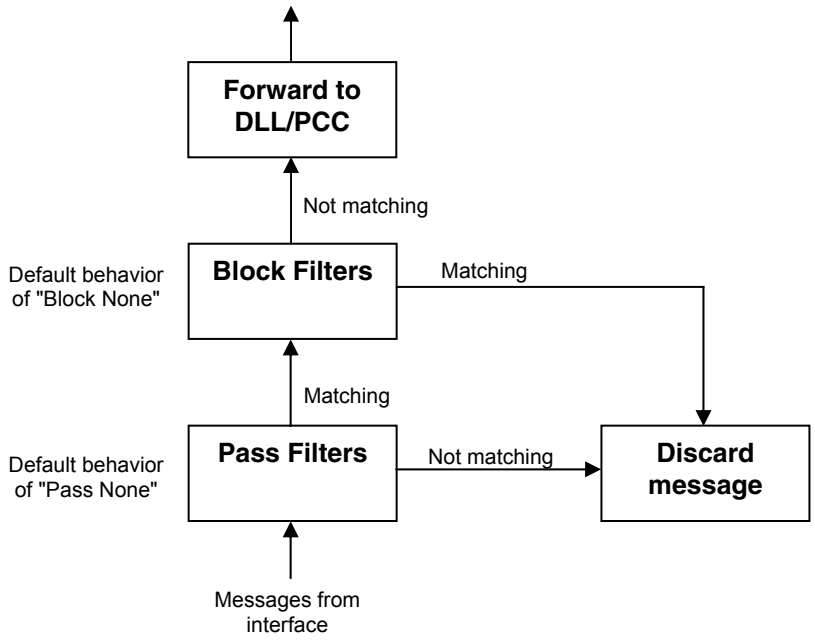

FIGURE 17—MESSAGE FILTER FLOW

For ISO 15765:

- PASS_FILTERs and BLOCK_FILTERs must not be used and shall cause the interface to return ERR_INVALID_FILTER_ID
- Filters shall not be applied to indications or loopback messages. When loopback is on, the original message shall be copied to the receive queue upon the last segment being transmitted on the bus
- Non-segmented messages do not need to match a FLOW_CONTROL_FILTER
- No segmented messages can be transmitted without matching an appropriate FLOW_CONTROL_FILTER. An appropriate filter is one in which the pFlowControlMsg CAN ID matches the messages to be transmitted. Also, the ISO 15765_ADDR_TYPE (reference TxFlags in Section 8.7.3) bits must match. If that bit is set, the first byte after the CAN IDs (the extended address) must match too
- No message (segmented or unsegmented) shall be received without matching an appropriate FLOW_CONTROL_FILTER. An appropriate filter is one in which the pPatternMsg CAN ID matches the incoming message ID. If the ISO 15765_ADDR_TYPE (reference TxFlags in Section 8.7.3) bit is set in the filter, the first byte after the CAN IDs (the extended address) must match too
- All 3 message pointers must have the same DataSize and TxFlags. Otherwise, the interface shall return ERR_INVALID_MSG
- Both the pFlowControlMsg ID and the pPatternMsg ID must be unique (not match any IDs in any other filters). The only exception is that pPatternMsg can equal pFlowControlMsg to allow for receiving functionally addressed messages. In this case, only non-segmented messages can be received
- See Appendix A for a detailed description of flow control filter usage.

7.2.9.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruStartMsgFilter
(
        unsigned long ChannelID,
        unsigned long FilterType,
        PASSTHRU_MSG *pMaskMsg,
        PASSTHRU_MSG *pPatternMsg,
        PASSTHRU_MSG *pFlowControlMsg,
        unsigned long *pFilterID
)
```

7.2.9.2 Parameters

ChannelID — The channel ID assigned by the PassThruConnect function.

FilterType — Designates:

PASS_FILTER – allows matching messages into the receive queue. This filter type is only valid on non-ISO 15765 channels

BLOCK_FILTER – keeps matching messages out of the receive queue. This filter type is only valid on non-ISO 15765 channels

FLOW_CONTROL_FILTER – allows matching messages into the receive queue and defines an outgoing flow control message to support the ISO 15765-2 flow control mechanism. This filter type is only valid on ISO 15765 channels.

pMaskMsg — For a PASS_FILTER or BLOCK_FILTER:

This designates a pointer to the mask message that will be applied to each incoming message (i.e., the mask message that will be ANDed to each incoming message) to mask any unimportant bits.

When using the CAN protocol, setting the first 4 bytes of pMaskMsg to $FF makes the filter specific to one CAN ID. Using other values allows for the reception or blocking of multiple CAN identifiers.

For a FLOW_CONTROL_FILTER:

The mask shall consist of 4 or 5 bytes of $FF, with a corresponding DataSize. Five bytes are only allowed when the extended address bit in TxFlags is set. Flow control filters are point-to-point, and shall not be allowed to match multiple CAN identifiers. The only exception is to allow for masking the priority field in a 29-bit CAN ID as specified in ISO 15765-2 Annex A. In this case Data[0] can be $E3.

pPatternMsg — For a PASS_FILTER or BLOCK_FILTER:

Designates a pointer to the pattern message that will be compared to the incoming message after the mask message has been applied. If the result matches this pattern message and the FilterType is PASS_FILTER, then the incoming message will be added to the receive queue (otherwise it will be discarded). If the result matches this pattern message and the FilterType is BLOCK_FILTER, then the incoming message will be discarded (otherwise it will be added to the receive queue). Message bytes in the received message that are beyond the DataSize of the pattern message will be treated as “don’t care”.

For a FLOW_CONTROL_FILTER:

Designates a pointer to the CAN ID (with optional extended address) at the other end of an ISO 15765-2 conversation. Any messages on the bus not matching a pPatternMsg must be discarded.

pFlowControlMsg — This pointer must be null when requesting a PASS_FILTER or a BLOCK_FILTER, otherwise ERR_INVALID_MSG shall be returned.

For a FLOW_CONTROL_FILTER:

Designates a pointer to the CAN ID used when sending CAN frames during an ISO 15765-2 segmented transmission or reception. This is the CAN ID to match against the CAN ID in a segmented PassThruWriteMsg. This message shall only contain the CAN ID (and extended address byte if the ISO15765_EXT_ADDR flag is set).

pFilterID — Pointer to location for the filter ID that is assigned by the DLL.

7.2.9.3 Filter Type Values – See Figure 18

Definition	**Value**
PASS_FILTER	0x00000001
BLOCK_FILTER	0x00000002
FLOW_CONTROL_FILTER	0x00000003
Reserved	0x00000004-0x00007FFF
Reserved for SAE J2534-2	0x00008000-0x0000FFFF
Tool manufacturer specific	0x00010000-0xFFFFFFFF

FIGURE 18—FILTER TYPE VALUES

7.2.9.4 Return Values – See Figure 19

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device.
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by message pointers or messages do not share common TxFlags and DataSize. (Reference Section 8 - Message Structure)
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description.
ERR_NOT_UNIQUE	A CAN ID in pPatternMsg or pFlowControlMsg matches either ID in an existing FLOW_CONTROL_FILTER.
ERR_EXCEEDED_LIMIT	Exceeded the maximum number of filter message IDs or the maximum allocated space.
ERR_MSG_PROTOCOL_ID	Protocol type in the message does not match the protocol associated with the ChannelID.

FIGURE 19—RETURN VALUES

7.2.10 PASSTHRUSTOPMSGFILTER

This function removes the specified filter. If the function is successful, a value of STATUS_NOERROR is returned. After this call the FilterID will be invalid.

7.2.10.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruStopMsgFilter
(
        unsigned long ChannelID,
        unsigned long FilterID
)
```

7.2.10.2 Parameters

ChannelID The channel ID assigned by the PassThruConnect function.

FilterID Filter ID that is assigned by the PassThruStartMsgFilter function.

7.2.10.3 Return Values – See Figure 20

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_FILTER_ID	Invalid FilterID value.

FIGURE 20—RETURN VALUES

7.2.11 PASSTHRUSETPROGRAMMINGVOLTAGE

This function sets a single programming voltage on a single specific pin. The programming voltage pins are mutually exclusive, and at any given time programming voltage can only be applied to a single pin. This does not apply to pin 15 (short to ground), which can be shorted to ground simultaneously with programming voltage on another pin. The programming voltage must be turned off before the programming voltage can be applied to a different pin. The default state of the pins shall be VOLTAGE_OFF.

If the function is successful, a value of STATUS_NOERROR is returned. It is up to the application programmer to insure that voltages are not applied to any pins incorrectly. This function cannot protect from incorrect usage (e.g., applying a voltage to pin 6 when it is being used for the CAN protocol). Note that for SCI protocol, the application would set the PinNumber, set the Voltage to VOLTAGE_OFF, and set SCI_TX_VOLTAGE in TxFlags of the message to pulse the programming voltage to 20 V DC.

This function can be called only after calling PassThruOpen.

7.2.11.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruSetProgrammingVoltage
(
        unsigned long DeviceID,
        unsigned long PinNumber,
        unsigned long Voltage
)
```

7.2.11.2 Parameters

DeviceID Device ID returned from PassThruOpen

PinNumber The pin on which the programming voltage will be set. Valid options are:
0 – Auxiliary output pin (for non-SAE J1962 connectors)
6 – Pin 6 on the SAE J1962 connector.
9 – Pin 9 on the SAE J1962 connector.
11 – Pin 11 on the SAE J1962 connector.
12 – Pin 12 on the SAE J1962 connector.
13 – Pin 13 on the SAE J1962 connector.
14 – Pin 14 on the SAE J1962 connector.
15 – Pin 15 on the SAE J1962 connector (short to ground only).

Voltage The voltage (in millivolts) to be set. Valid values are:
5000mV-20000mV (limited to 150mA with a resolution of 100 millivolts for pins 0, 6, 9, 11, 12, 13, and 14).
VOLTAGE_OFF – To turn output off (disconnect-high impedance ≥ 500KΩ).
SHORT_TO_GROUND – Short pin to ground (limited to 300mA on pin 15 only).

7.2.11.3 Voltage Values – See Figure 21

Definition	Value
Programming Voltage	0x00001388 (5000 mV) to 0x00004E20 (20000 mV)
SHORT_TO_GROUND	0xFFFFFFFE
VOLTAGE_OFF	0xFFFFFFFF

FIGURE 21—VOLTAGE VALUES

7.2.11.4 Return Values – See Figure 22

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device.
ERR_NOT_SUPPORTED	Function not supported.
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_FAILED	Undefined error, use PassThruGetLastError for text description.
ERR_PIN_INVALID	Invalid pin number, pin number already in use, or voltage already applied to a different pin.

FIGURE 22—RETURN VALUES

7.2.12 PASSTHRUREADVERSION

This function returns the version strings associated with the DLL. If the function is successful, a value of STATUS_NOERROR is returned. A buffer of at least eighty (80) characters must be allocated for each pointer by the application.

This function can be called only after calling PassThruOpen.

7.2.12.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruReadVersion
(
        unsigned long DeviceID
        char*pFirmwareVersion,
        char*pDllVersion,
        char*pApiVersion
)
```

7.2.12.2 Parameters

DeviceID	Device ID returned from PassThruOpen
pFirmwareVersion	Pointer to Firmware version string. This string is determined by the interface vendor that supplies the device.
pDllVersion	Pointer to DLL version string. This string is determined by the interface vendor that supplies the DLL.
pApiVersion	Pointer to API version string in YY. MM format. This string corresponds to the date of the approved document (may not be equivalent to SAE publication date). February 2002 Final = “02.02” November 2004 Final (this version) = “04.04”

7.2.12.3 Return Values – See Figure 23

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required

FIGURE 23—RETURN VALUES

7.2.13 PASSTHRUGETLASTERROR

This function returns the text string description for an error detected during the last function call (except PassThruGetLastError). The error string must be retrieved before calling any other function. The buffer pointed to by pErrorDescription is allocated by the application and must be at least eighty (80) characters.

This function can be called without first calling PassThruConnect or PassThruOpen. The last error returned is not specific to any particular Channel ID or Device ID and is related to the last function call. It would be expected that the application would call this function immediately after a function fails. This function is mainly for application developers.

7.2.13.1 C / C++ Prototype

```
extern “C” long WINAPI PassThruGetLastError
(
        char  *pErrorDescription
)
```

7.2.13.2 Parameters

pErrorDescription Pointer to error description string.

7.2.13.3 Return Values – See Figure 24

Definition	Description
STATUS_NOERROR	Function call successful
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required

FIGURE 24—RETURN VALUES

7.2.14 PASSTHRUIOCTL

This function is used to read and write all the protocol hardware and software configuration parameters. If the function is successful, a value of STATUS_NOERROR is returned. The structures pointed to by pInput and pOutput are determined by the IoctlID. See section on IOCTL structures for details.

PassThruOpen must be called prior to any Ioctl call. Some Ioctl functions do not require a ChannelID and therefore do not require that PassThruConnect be called prior to Ioctl. In this case, the DeviceID must be passed instead.

7.2.14.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruIoctl
(
        unsigned long ChannelID,
        unsigned long IoctlID,
        void *pInput,
        void *pOutput
)
```

7.2.14.2 Parameters

ChannelID	The channel ID assigned by the PassThruConnect function, except in designated Ioctls where the device ID is passed instead.
IoctlID	Ioctl ID (see the IOCTL Section).
pInput	Pointer to input structure (see the IOCTL Section).
pOutput	Pointer to output structure (see the IOCTL Section).

7.2.14.3 Ioctl ID Values – See Figure 25

Definition	Value
GET_CONFIG	0x01
SET_CONFIG	0x02
READ_VBATT	0x03
FIVE_BAUD_INIT	0x04
FAST_INIT	0x05
CLEAR_TX_BUFFER	0x07
CLEAR_RX_BUFFER	0x08
CLEAR_PERIODIC_MSGS	0x09
CLEAR_MSG_FILTERS	0x0A
CLEAR_FUNCT_MSG_LOOKUP_TABLE	0x0B
ADD_TO_FUNCT_MSG_LOOKUP_TABLE	0x0C
DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE	0x0D
READ_PROG_VOLTAGE	0x0E
Reserved for SAE	0x0F – 0x7FFF
Reserved for SAE J2534-2	0x8000 – 0xFFFF
Tool manufacturer specific	0x10000 – 0FFFFFFF

FIGURE 25—IOCTL ID VALUES

7.2.14.4 Return Values – See Figure 26

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Unable to communicate with device
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_IOCTL_ID	Invalid IoctlID value.
ERR_NULL_PARAMETER	NULL pointer supplied where a valid pointer is required
ERR_NOT_SUPPORTED	Invalid or unsupported parameter/value
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_MSG	Invalid message structure pointed to by pInput when using FAST_INT_Ioctl. (Reference Section 8 Message Structure)
ERR_INVALID_DEVICE_ID	Device ID invalid
ERR_INVALID_IOCTL_VALUE	Invalid value for Ioctl parameter

FIGURE 26—RETURN VALUES

7.3 IOCTL Section

Figure 27 provides the details on the IOCTLs available through PassThruIoctl function:

Value of IoctlID	InputPtr represents	OutputPtr represents	Purpose
GET_CONFIG	Pointer to SCONFIG_LIST	NULL pointer	To get the vehicle network configuration of the pass-thru device
SET_CONFIG	Pointer to SCONFIG_LIST	NULL pointer	To set the vehicle network configuration of the pass-thru device
READ_VBATT	NULL pointer	Pointer to unsigned long	To direct the pass-thru device to read the voltage on pin 16 of the J1962 connector
FIVE_BAUD_INIT	Pointer to SBYTE_ARRAY	Pointer to SBYTE_ARRAY	To direct the pass-thru device to initiate a 5 baud initialization sequence
FAST_INIT	NULL or Pointer to PASSTHRU_MSG	NULL or Pointer to PASSTHRU_MSG	To direct the pass-thru device to initiate a fast initialization sequence
CLEAR_TX_BUFFER	NULL pointer	NULL pointer	To direct the pass-thru device to clear all messages in its transmit queue
CLEAR_RX_BUFFER	NULL pointer	NULL pointer	To direct the pass-thru device to clear all messages in its receive queue
CLEAR_PERIODIC_MSGS	NULL pointer	NULL pointer	To direct the pass-thru device to clear all periodic messages on the channel, thus stopping all periodic message transmission
CLEAR_MSG_FILTERS	NULL pointer	NULL pointer	To direct the pass-thru device to clear all message filters on the channel
CLEAR_FUNCT_ MSG_LOOKUP_TABLE	NULL pointer	NULL pointer	To direct the pass-thru device to clear the Functional Message Look-up Table
ADD_TO_FUNCT_ MSG_LOOKUP_TABLE	Pointer to SBYTE_ARRAY	NULL pointer	To direct the pass-thru device to add a functional address to the Functional Message Look-up Table
DELETE_FROM_FUNCT_ MSG_LOOKUP_TABLE	Pointer to SBYTE_ARRAY	NULL pointer	To direct the pass-thru device to delete a functional address from the Functional Message Look-up Table
READ_PROG_VOLTAGE	NULL pointer	Pointer to unsigned long	To direct the pass-thru device to read the feedback of the programmable voltage set by PassThruSetProgrammingVoltage

FIGURE 27—IOCTL DETAILS

7.3.1 GET_CONFIG

The IoctlID value of GET_CONFIG is used to obtain the vehicle network configuration of the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 28. When the function is successfully completed, the corresponding parameter value(s) indicated in Figure 30 will be placed in each Value.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define GET_CONFIG.
InputPtr	Points to the structure SCONFIG_LIST, which is defined as follows: typedef struct { unsigned long NumOfParams; /* number of SCONFIG elements */ SCONFIG *ConfigPtr; /* array of SCONFIG */ } SCONFIG_LIST where: NumOfParms is an INPUT, which contains the number of SCONFIG elements in the array pointed to by ConfigPtr. ConfigPtr is a pointer to an array of SCONFIG structures. The structure SCONFIG is defined as follows: typedef struct { unsigned long Parameter; /* name of parameter */ unsigned long Value; /* value of the parameter */ } SCONFIG where: Parameter is an INPUT that represents the parameter to be obtained (See Figure 30 for a list of valid parameters). Value is an OUTPUT that represents the value of that parameter (See Figure 30 for a list of valid values).
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 28—GET_CONFIG DETAIL

7.3.2 SET_CONFIG

The IoctlID value of SET_CONFIG is used to set the vehicle network configuration of the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 29. When the function is successfully completed the corresponding parameter(s) and value(s) indicated in Figure 30 will be in effect.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define SET_CONFIG.
InputPtr	Points to the structure SCONFIG_LIST, which is defined as follows: typedef struct { unsigned long NumOfParams; /* number of SCONFIG elements */ SCONFIG *ConfigPtr; /* array of SCONFIG */ } SCONFIG_LIST where: NumOfParms is an INPUT, which contains the number of SCONFIG elements in the array pointed to by ConfigPtr. ConfigPtr is a pointer to an array of SCONFIG structures. The structure SCONFIG is defined as follows: typedef struct { unsigned long Parameter; /* name of parameter */ unsigned long Value; /* value of the parameter */ } SCONFIG where: Parameter is an INPUT that represents the parameter to be set (See Figure 30 for a list of valid parameters). Value is an INPUT that represents the value of that parameter (See Figure 30 for a list of valid values).
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 29—SET_CONFIG DETAILS

Parameter	ID Value	Valid values for Parameter	Default Value (decimal)	Description
DATA_RATE	0x01	5-500000	protocol specific	Represents the desired baud rate. An ERR_INVALID_IOCTL_VALUE will be returned if the desired baud rate cannot be achieved within the tolerance specified in Section 6.5. The interface will remain at the previous baud rate.
Unused	0x02			Reserved for SAE
LOOPBACK	0x03	0 (OFF) 1 (ON)	0	0 = Don't echo transmitted messages in the receive queue. 1 = Echo transmitted messages, including periodic messages, in the receive queue. Loopback messages must only be sent after successful transmission of a message. Loopback frames are not subject to message filtering.
NODE_ADDRESS	0x04	0x00-0xFF	N/A	For a protocol ID of J1850PWM, this sets the node address in the physical layer of the vehicle network.
NETWORK_LINE	0x05	0 (BUS_NORMAL) 1 (BUS_PLUS) 2 (BUS_MINUS)	0	For a protocol ID of J1850PWM, this sets the network line(s) that are active during communication (for cases where the physical layer allows this).
P1_MIN (not used by interface)	0x06	N/A	N/A	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum inter-byte time for ECU responses. Application shall not get or set this value. The interface will not check for P1_MIN violations. Interface must be capable of handling P1_MIN = 0.
P1_MAX	0x07	0x1-0xFFFF (.5 ms per bit)	40 (20 ms)	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum inter-byte time for ECU responses.
P2_MIN (not used by interface)	0x08	N/A	N/A	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum time between tester request and ECU responses or two ECU responses Application shall not get or set this value. The interface will not check for P2_MIN violations. After the request, the interface shall be capable of handling an immediate response (P2_MIN = 0). For subsequent responses, a byte received after P1_MAX shall be considered as the start of the subsequent response.
P2_MAX (not used by interface)	0x09	N/A	N/A	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum time between tester request and ECU responses or two ECU responses. Application shall not get or set this value. The interface will not check for P2_MAX violations. The interface will accept all responses up to P3_MIN.

Parameter	ID Value	Valid values for Parameter	Default Value (decimal)	Description
P3_MIN	0x0A	0x0-0xFFFF (.5 ms per bit)	110 (55 ms)	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum time between end of ECU response and start of new tester request.
P3_MAX (not used by interface)	0x0B	N/A	N/A	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum time between end of ECU response and start of new tester request. Application shall not get or set this value. Interface allows transmission of a request any time after P3_MIN.
P4_MIN	0x0C	0x0-0xFFFF (.5 ms per bit)	10 (5 ms)	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum inter-byte time for a tester request.
P4_MAX (not used by interface)	0x0D	N/A	N/A	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum inter-byte time for a tester request. Application shall not get or set this value. The interface shall transmit at P4_MIN.
W0	0x19	0x0-0xFFFF (1 ms per bit)	300	For protocol ID of ISO 9141, this sets the minimum bus idle time before the tester starts to transmit the address byte.
W1	0x0E	0x0-0xFFFF (1 ms per bit)	300	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum time from the end of the address byte to the start of the synchronization pattern.
W2	0x0F	0x0-0xFFFF (1 ms per bit)	20	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum time from the end of the synchronization pattern to the start of key byte 1.
W3	0x10	0x0-0xFFFF (1 ms per bit)	20	For protocol ID of ISO 9141 or ISO 14230, this sets the maximum time between key byte 1 and key byte 2.
W4	0x11	0x0-0xFFFF (1 ms per bit)	50	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum time between key byte 2 and its inversion from the tester.
W5	0x12	0x0-0xFFFF (1 ms per bit)	300	For protocol ID of ISO 14230, this sets the minimum bus idle time before the tester starts to transmit the address byte.
TIDLE	0x13	0x0-0xFFFF (1 ms per bit)	300	For protocol ID of ISO 9141 or ISO 14230, this sets the minimum amount of bus idle time that is needed before a fast initialization sequence will begin.
TINIL	0x14	0x0-0xFFFF (1 ms per bit)	25	For protocol ID of ISO 9141 or ISO 14230, this sets the duration for the low pulse in fast initialization.
TWUP	0x15	0x0-0xFFFF (1 ms per bit)	50	For protocol ID of ISO 9141 or ISO 14230, this sets the duration of the wake-up pulse in fast initialization.
PARITY	0x16	0 (NO_PARITY) 1 (ODD_PARITY) 2 (EVEN_PARITY)	0	For a protocol ID of ISO 9141 or ISO 14230 only.
BIT_SAMPLE_POINT	0x17	0-100 (1% per bit)	80	For a protocol ID of CAN, this sets the desired bit sample point as a percentage of the bit time.
SYNC_JUMP_WIDTH	0x18	0-100 (1% per bit)	15	For a protocol ID of CAN, this sets the desired synchronization jump width as a percentage of the bit time.

Parameter	ID Value	Valid values for Parameter	Default Value (decimal)	Description
T1_MAX	0x1A	0x0-0xFFFF (1 ms per bit)	20	For protocol ID of SCI_A_ENGINE, SCI_A_TRANS, SCI_B_ENGINE or SCI_B_TRANS, this sets the maximum inter-frame response delay.
T2_MAX	0x1B	0x0-0xFFFF (1 ms per bit)	100	For protocol ID of SCI_A_ENGINE, SCI_A_TRANS, SCI_B_ENGINE or SCI_B_TRANS, this sets the maximum inter-frame request delay.
T3_MAX	0x24	0x0-0xFFFF (1 ms per bit)	50	For protocol ID of SCI_A_ENGINE, SCI_A_TRANS, SCI_B_ENGINE or SCI_B_TRANS, this sets the maximum response delay from the ECU after processing a valid request message from the tester.
T4_MAX	0x1C	0x0-0xFFFF (1 ms per bit)	20	For protocol ID of SCI_A_ENGINE, SCI_A_TRANS, SCI_B_ENGINE or SCI_B_TRANS, this sets the maximum inter-message response delay.
T5_MAX	0x1D	0x0-0xFFFF (1 ms per bit)	100	For protocol ID of SCI_A_ENGINE, SCI_A_TRANS, SCI_B_ENGINE or SCI_B_TRANS, this sets the maximum inter-message request delay.
ISO15765_BS	0x1E	0x0-0xFF (see ISO 15765-2)	0	For protocol ID of ISO 15765, this sets the block size the interface should report to the vehicle for receiving segmented transfers.
ISO15765_STMIN	0x1F	0x0-0xFF (see ISO 15765-2)	0	For protocol ID of ISO 15765, this sets the separation time the interface should report to the vehicle for receiving segmented transfers.
BS_TX	0x22	0x0-0xFF, 0xFFFF (see ISO 15765-2)	0xFFFF	For protocol ID of ISO 15765, this sets the block size the interface should use to transmit segmented messages to the vehicle. The flow control value reported by the vehicle should be ignored. If value is 0xFFFF, use the value reported by the vehicle.
STMIN_TX	0x23	0x0-0xFF, 0xFFFF (see ISO 15765-2)	0xFFFF	For protocol ID of ISO 15765, this sets the separation time the interface should use to transmit segmented messages to the vehicle. The flow control value reported by the vehicle should be ignored. If value is 0xFFFF, use the value reported by the vehicle.
DATA_BITS	0x20	0 (8 data bits) 1 (7 data bits)	0	For protocol ID of ISO 9141 or ISO 14230 only.
FIVE_BAUD_MOD	0x21	0-Initialization as defined in ISO 9141-2 and ISO 14230-4 1-ISO 9141 initialization followed by interface sending inverted Key Byte 2 2- ISO 9141 initialization followed by ECU sending inverted address 3- Initialization as defined in ISO 9141	0	For a protocol ID of ISO 9141 or ISO 14230 only. Initialization for ISO 9141-2 and ISO 14230 include the initialization sequence as defined in ISO 9141 plus inverted key byte #2 sent from the tester to the ECU and inverted address sent from the ECU to the tester. This parameter allows either ISO 9141 initialization sequence, ISO 9141-2 / ISO 14230 initialization sequence, or hybrid versions which include only one of the extra bytes defined for ISO 9141-2 and ISO 14230.

Parameter	ID Value	Valid values for Parameter	Default Value (decimal)	Description
ISO15765_WFT_MAX	0x25	0x0-0xff	0	For protocol ID ISO 15765, the number of WAIT flow control frames allowed (N_WFTmax) during a multi-segment transfer.
Reserved	0x26- 0x7FFF			Reserved for SAE
Reserved	0x8000 – 0xFFFF			Reserved for SAE J2534-2
Tool manufacturer specific	0x10000 – 0xFFFFFFFF	Manufacturer Specific		Manufacturer Specific

FIGURE 30—IOCTL GET_CONFIG / SET_CONFIG PARAMETER DETAILS

7.3.3 READ_VBATT

The IoctlID value of READ_VBATT is used to obtain the voltage measured on pin 16 of the SAE J1962 connector from the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 31. When the function is successfully completed, battery voltage will be placed in the variable pointed to by OutputPtr. The units will be in milli-volts and will be rounded to the nearest tenth of a volt.

Parameter	Description
ChannelID	Device ID assigned by DLL during PassThruOpen
IoctlID	Is set to the define READ_VBATT.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a pointer to an unsigned long.

FIGURE 31—READ_VBATT DETAILS

7.3.4 READ_PROG_VOLTAGE

The IoctlID value of READ_PROG_VOLTAGE is used to obtain the programming voltage of the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 32. When the function is successfully completed, programming voltage will be placed in the variable pointed to by OutputPtr. The units will be in milli-volts and will be rounded to the nearest tenth of a volt.

Parameter	Description
ChannelID	Device ID assigned by DLL during PassThruOpen
IoctlID	Is set to the define READ_PROG_VOLTAGE.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a pointer to an unsigned long.

FIGURE 32—READ_PROG_VOLTAGE DETAILS

7.3.5 FIVE_BAUD_INIT

The IoctlID value of FIVE_BAUD_INIT is used to initiate a five-baud initialization sequence from the pass-thru device. The ISO 9141 five baud initialization sequence includes the five baud address sent from the tester to the ECU, followed by the synchronization byte and two key bytes sent from the ECU to the tester. The ISO 9141-2 and ISO 14230 five baud initialization includes the ISO 9141 initialization sequence, but adds the inverted key byte 2 value sent from the tester to the ECU, and the inverted initialization address sent from the ECU to the tester. The FIVE_BAUD_MOD parameter in SET_CONFIG allows the application to selectively include or exclude the two additional bytes defined in the ISO 14230 initialization sequence.

The calling application is responsible for allocating and initializing the associated parameters described in Figure 33. When the function is successfully completed, the key words will be placed in structure pointed to by OutputPtr. It should be noted that this only applies to Protocol ID of ISO9141 or ISO14230.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define FIVE_BAUD_INIT.
InputPtr	Points to the structure SBYTE_ARRAY, which is defined as follows: Typedef struct { unsigned long NumOfBytes; /* number of bytes in the array */ unsigned char *BytePtr; /* array of bytes */ } SBYTE_ARRAY where: NumOfBytes is an INPUT that must be set to “1” and indicates the number of bytes in the array BytePtr. BytePtr[0] is an INPUT that contains the target address. The remaining elements in BytePtr are not used.
OutputPtr	Points to the structure SBYTE_ARRAY defined above where: NumOfBytes is an INPUT which indicates the maximum size of the array BytePtr and an OUTPUT which indicates the number of bytes in the array BytePtr. Must be 2 or less. BytePtr[0] is an OUTPUT that contains key word 1 from the ECU. BytePtr[1] is an OUTPUT that contains key word 2 from the ECU. The remaining elements in BytesPtr are not used.

FIGURE 33—FIVE_BAUD_INIT DETAILS

7.3.6 FAST_INIT

The IoctlID value of FAST_INIT is used to initiate a fast initialization sequence from the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 34. When the function is successfully completed, the response message will be placed in structure pointed to by OutputPtr. It should be noted that this only applies to Protocol ID of ISO9141 or ISO14230. In case of successful initialization the OutputPtr will contain valid data.

For the ISO9141 Protocol, no verification of message format will be applied.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define FAST_INIT.
InputPtr	Points to the structure PASSTHRU_MSG (see the message definition section of this document) which the pass-thru device will send. This can be a Start Communication Request Message as defined by ISO 14230, or any other ISO 9141 or ISO 14230 message specified by the application. If NULL, no message is transmitted on the bus as part of the fast init sequence
OutputPtr	Points to the structure PASSTHRU_MSG (see the message definition section of this document) which will hold the response to the Start Communication Request Message. If a response is not received in the time allowed by ISO14230 the Fast Initialization is deemed to have failed and the contents of this structure will be indeterminate. If NULL, no response is expected.

FIGURE 34—FAST_INIT DETAILS

7.3.7 CLEAR_TX_BUFFER

The IoctlID value of CLEAR_TX_BUFFER is used to direct the pass-thru device to clear its transmit queue. The calling application is responsible for allocating and initializing the associated parameters described in Figure 35. When the function is successfully completed, the transmit queue will have been cleared.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define CLEAR_TX_BUFFER.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 35—CLEAR_TX_BUFFER DETAILS

7.3.8 CLEAR_RX_BUFFER

The IoctlID value of CLEAR_RX_BUFFER is used to direct the pass-thru device to clear its receive queue. The calling application is responsible for allocating and initializing the associated parameters described in Figure 36. When the function is successfully completed, the receive queue will have been cleared.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define CLEAR_RX_BUFFER.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 36—CLEAR_RX_BUFFER DETAILS

7.3.9 CLEAR_PERIODIC_MSGS

The IoctlID value of CLEAR_PERIODIC_MSGS is used to direct the pass-thru device to clear its periodic messages. The calling application is responsible for allocating and initializing the associated parameters described in Figure 37. When the function is successfully completed, the list will have been cleared and all periodic messages will have stopped transmitting.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define CLEAR_PERIODIC_MSGS.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 37—CLEAR_PERIODIC_MSGS DETAILS

7.3.10 CLEAR_MSG_FILTERS

The IoctlID value of CLEAR_MSG_FILTERS is used to direct the pass-thru device to clear its message filters on the specified channel. The calling application is responsible for allocating and initializing the associated parameters described in Figure 38. When the function is successfully completed, there will be no filters on the channel (the default state after a PassThruConnect). No more messages shall be queued until a PASS_FILTER or FLOW_CONTROL_FILTER is added.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define CLEAR_MSG_FILTERS.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 38—CLEAR_MSG_FILTERS DETAILS

7.3.11 CLEAR_FUNCT_MSG_LOOKUP_TABLE

The IoctlID value of CLEAR_FUNCT_MSG_LOOKUP_TABLE is used to direct the pass-thru device to clear its functional message look-up table. The calling application is responsible for allocating and initializing the associated parameters described in Figure 39. When the function is successfully completed, the table will have been cleared. It should be noted that this only applies to Protocol ID of SAE J1850PWM.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define CLEAR_FUNCT_MSG_LOOKUP_TABLE.
InputPtr	Is a NULL pointer, as this parameter is not used.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 39—CLEAR_FUNCT_MSG_LOOKUP_TABLE DETAILS

7.3.12 ADD_TO_FUNCT_MSG_LOOKUP_TABLE

The IoctlID value of ADD_TO_FUNCT_MSG_LOOKUP_TABLE is used to add functional address(es) to the functional message look-up table in the physical layer of the vehicle network on the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 40. When the function is successfully completed, the look-up table will have been altered. It should be noted that this only applies to Protocol ID of SAE J1850PWM.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define ADD_TO_FUNCT_MSG_LOOKUP_TABLE.
InputPtr	Points to the structure SBYTE_ARRAY, which is defined as follows: Typedef struct { unsigned long NumOfBytes; /* number of bytes in the array */ unsigned char *BytePtr; /* array of bytes */ } SBYTE_ARRAY where: NumOfBytes is an INPUT that indicates the number of bytes in the array BytePtr. BytePtr[0] is an INPUT that contains the first functional address to be added. . . . BytePtr[n] is an INPUT that contains the nth functional address to be added.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 40—ADD_TO_FUNCT_MSG_LOOKUP_TABLE DETAILS

7.3.13 DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE

The IoctlID value of DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE is used to delete functional address(es) from the functional message look-up table in the physical layer of the vehicle network on the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 41. When the function is successfully completed, the look-up table will have been altered. It should be noted that this only applies to Protocol ID of J1850PWM.

Parameter	Description
ChannelID	Channel ID assigned by DLL during PassThruConnect
IoctlID	Is set to the define DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE.
InputPtr	Points to the structure SBYTE_ARRAY, which is defined as follows: Typedef struct { unsigned long NumOfBytes; /* number of bytes in the array */ unsigned char *BytePtr; /* array of bytes */ } SBYTE_ARRAY where: NumOfBytes is an INPUT that indicates the number of bytes in the array BytePtr. BytePtr[0] is an INPUT that contains the first functional address to be deleted. . . . BytePtr[n] is an INPUT that contains the nth functional address to be deleted.
OutputPtr	Is a NULL pointer, as this parameter is not used.

FIGURE 41—DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE DETAILS

8. Message Structure

The following message structure will be used for all messages (Transmit, Receive, Filters, and Periodics) and indications. The total message size (in bytes) is the DataSize, and includes header bytes, ID bytes, and data bytes. For consistency, all interfaces should detect only the errors listed for each protocol in the following sections when returning ERR_INVALID_MSG.

8.1 C / C++ Definition

```
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;
```

8.2 Elements

ProtocolID	Protocol type
RxStatus	Receive message status – See RxStatus in “Message Flags and Status Definition” section
TxFlags	Transmit message flags – See TxFlags in “Message Flags and Status Definition” section
Timestamp	Received message timestamp (microseconds): For the START_OF_FRAME indication, the timestamp is for the start of the first bit of the message. For all other indications and transmit and receive messages, the timestamp is the end of the last bit of the message. For all other error indications, the timestamp is the time the error is detected.
DataSize	Data size in bytes, including header bytes, ID bytes, message data bytes, and extra data, if any.
ExtraDataIndex	Start position of extra data in received message (for example, IFR). The extra data bytes follow the body bytes in the Data array. The index is zero-based. When no extra data bytes are present in the message, ExtraDataIndex shall be set equal to DataSize. Therefore, if DataSize equals ExtraDataIndex, there are no extra data bytes. If ExtraDataIndex=0, then all bytes in the data array are extra bytes.
Data	Array of data bytes. Includes message headers, message body, and any extra data bytes. The application must fill all fields for each PASSTHRU_MSG structure passed to the API, except for RxStatus, Timestamp, and ExtraDataIndex. These three fields are only valid when reading a message or indication with PassThruReadMsg.

8.3 Message Data Formats

This section describes the bytes in the Data section of the PASSTHRU_MSG structure. Figure 42 shows the minimum and maximum transmit and receive message size for each protocol.

When using CAN or ISO 15765-4, the first 4 bytes of Data contain the CAN ID. Data [0] contains CAN ID bits 28-24 (the three most significant bits will be zero), Data[1] contains bits 23-16, Data [2] contains bits 15-8, and Data [3] contains bits 7-0. If ISO15765_ADDR_TYPE is set in TxFlags, then the next byte (Data[4]) will be the extended address.

Protocol	Min Tx	Max Tx	Min Rx	Max Rx	Notes
CAN	4	12	4	12	4 bytes of CANID, followed by up to 8 data bytes
ISO15765	4	4099	4	4099	4 bytes of CAN ID, followed by up to 4095 data bytes
ISO15765 (Extended Address)	5	4100	5	4100	4 bytes of CAN ID, 1 Extended Address byte, followed by up to 4095 data bytes
J1850PWM	3	10	3	11	3 header bytes followed by up to 7data bytes. On Rx, any number of IFR bytes as long as the total message size is less than 11 bytes.
J1850VPW	1	4128	1	4128	
ISO9141	1	4128	1	4128	
ISO14230	1	259	1	259	1-4 header bytes followed by up to 255 data bytes.
ISO14230 (Manual Checksum*)	1	260	1	260	Format not defined. Allows for 4 header bytes, 255 data bytes and an application-defined checksum byte.
SCI	1	4128	1	4128	

FIGURE 42—ALLOWED MESSAGE SIZES PER PROTOCOL

NOTE—Manual checksum is when the ISO9141_NO_CHECKSUM bit in the Connect Flag is set to 1.

8.4 Format Checks for Messages Passed to the API

The vendor DLL shall validate all PASSTHRU_MSG structures, and return an ERR_INVALID_MSG in the following cases:

- DataSize violates Min Tx or Max Tx columns in Figure 42
- Source address (Data[3]) is different from the Node ID (Ioctl SET_CONFIG, Parameter NODE_ADDRESS) on J1850PWM
- The header length field is incorrect for the number of bytes in the message on ISO14230
- The CAN_29_BIT flag of the message does not match the CAN_29_BIT flag passed to PassThruConnect, unless the CAN_ID_BOTH bit was set on connect

The vendor DLL shall return ERR_MSG_PROTOCOL_ID when the ProtocolID field in the message does not match the Protocol ID specified when opening the channel.

8.5 Conventions for Returning Messages from the API

When returning a message in PassThruReadMsg:

– DataSize shall tell the application how many bytes in the Data array are valid. ExtraDataIndex will be the (non-zero) index of the last byte of the message. If ExtraDataIndex is not equal to DataSize there are extra data bytes after the message. If loopback is on, RxStatus must be consulted to tell if the message came via loopback.
– DataSize will be in the range shown in the Min Rx and Max Rx columns of Figure 42. If the device receives a message from the vehicle bus that is too long or too short, the message shall be discarded with no error.
– For received messages, ExtraDataIndex shall be equal to DataSize, except when the interface is returning SAE J1850 PWM IFR bytes. In no case shall ExtraDataIndex be larger than DataSize.
– When receiving a message on an SAE J1850 PWM channel, the message shall have any IFR bytes appended. In this case, ExtraDataIndex shall be the index of the first IFR byte, and DataSize shall be the total length of the original message plus all IFR bytes. For example, if there are two IFR bytes, DataSize will be incremented by two, and ExtraDataIndex will be DataSize - 2. When loopback is on, the loopback message shall contain any IFR bytes.

8.6 Conventions for Retuning Indications from the API

When returning an indication in PassThruReadMsg:

– ExtraDataIndex must be zero
– DataSize shall tell the application how many bytes in the Data array are valid
– RxStatus must be consulted to determine the indication type (See Section 8.4).
– A TxDone indication (ISO 15765 only) is generated by the DLL after a SingleFrame message is sent, or the last frame of a multi-segment transmission is sent. DataSize shall be 4 (or 5 when the message was using Extended Addressing). Data shall contain the CAN ID (and possible Extended Address) of the message just sent. If loopback is on, the TxDone indication shall precede the loopback message in the receive queue.

– An RxBreak indication (SAE J2610/SCI and SAE J1850VPW only) is generated by the DLL if a break is received.
– An RxStart indication is generated by the DLL when starting to receive a message on ISO9141 or ISO14230, or when receiving the FirstFrame signal of a multi-segment ISO 15765 message.

8.7 Message Flag and Status Definitions

8.7.1 RxStatus

Definitions for RxStatus bits are shown in Figure 43. The application shall ignore any flags that do not apply to the current channel.

Definition	RxStatu Bit(s)	Description	Value
	31-24	Tool manufacturer specific	Shall be set to 0
Reserved	23-16	Reserved for SAE J2534-2	Shall be set to 0
Reserved	15-9	Reserved for SAE	Shall be set to 0
CAN_29BIT_ID	8	CAN ID Type for CAN and ISO 15765	0 = 11-bit Identifier 1 = 29-bit Identifier
ISO15765_ADDR_TYPE	7	ISO 15765-2 Addressing Method	0= no extended address, 1= extended address is first byte after the CAN ID
	6-5	Reserved for SAE	Shall be set to 0
ISO15765_ PADDING_ERROR	4	For ProtocolID ISO 15765 a CAN frame was received with less than 8 data bytes	0 = No Error 1 = Padding Error
TX_INDICATION	3	ISO 15765 TxDone indication-CANID and extended address, if present, shall be included in the message structure	0= No TxDone 1= TxDone
RX_BREAK	2	Break indication received – SAE J2610 and SAE J1850 VPW only	0 = No break received 1 = Break received
START_OF_MESSAGE	1	Indicates the reception of the first byte of an ISO9141 or ISO14230 message or first frame of an ISO15765 multi-frame message	0 = Not a start of message indication 1 = First byte or frame received
TX_MSG_TYPE	0	Receive Indication/Transmit Loopback	0 = received i.e. this message was transmitted on the bus by another node, 1 = transmitted i.e. this is the echo of the message transmitted by the PassThru device

FIGURE 43—RXSTATUS BIT DEFINITIONS

8.7.2 RxSTATUS BITS FOR MESSAGE STATUS AND ERROR INDICATIONS

Valid combinations of RxStatus bits for messages and indications are shown in Figure 44: Valid RxStatus Bit Combinations. When the DLL is returning both a TxDone indication and a Loopback message, the TxDone indication shall be queued first. The RxStatus bits CAN_29BIT_ID and ISO15765_ADDR_TYPE are not shown because they do not affect the message/indication type. See PassThruReadMsg for a listing of which indications are valid for each protocol.

		RxStatus Bit(s) / Definition				
		4	3	2	1	0
Status	**Description**	ISO15765_ PADDING _ERROR	TX_ DONE	RX_ BREAK	START_ OF_ MESSAGE	TX_ MSG_ TYPE
Normal Message	Response was received successfully	**0**	**0**	**0**	**0**	**0**
RxStart	First byte/frame of a message received	**0**	**0**	**0**	**1**	**0**
RxBreak	Break indication received	**0**	**0**	**1**	**0**	**0**
RxPadError	A CAN frame was received with less than 8 bytes	**1**	**0**	**0**	**0**	**0**
TxDone	Request was transmitted successfully	**0**	**1**	**0**	**0**	**1**
Loopback Message	Loopback of message transmitted	**0**	**0**	**0**	**0**	**1**

FIGURE 44—VALID RX STATUS BIT COMBINATIONS

8.7.3 TxFlags

Definitions for TxFlags bits are shown in Figure 45. The interface shall ignore any flags that do not apply to the current channel.

Definition	TxFlags Bit(s)	Description	Value
	31-24	Unused	Tool manufacturer specific
SCI_TX_VOLTAGE	23	SCI programming voltage	0 = no voltage after message transmit, 1 = apply 20V after message transmit
SCI_MODE	22	SCI transmit mode	0 = Transmit using SCI Full duplex mode. 1 = Transmit using SCI Half duplex mode.
	21 – 16	Unused	Reserved for SAE J2534-2 – shall be set to 0
	15 - 10	Unused	Reserved for SAE – shall be set to 0
WAIT_P3_MIN_ONLY	9	Modified message timing for ISO 14230-used to decrease programming time if application knows only one response will be received	0 = Interface message timing as specified in ISO 14230. 1 = After a response is received for a physical request, the wait time shall be reduced to P3_MIN. Does not affect timing on Responses to functional requests
CAN_29BIT_ID	8	CAN ID type for CAN And ISO 15765	0 = 11-bit, 1 = 29-bit
ISO15765_ADDR_TYPE	7	ISO 15765-2 Addressing Method	0 = no extended address, 1 = extended address is first byte after the CAN ID
ISO15765_FRAME_PAD	6	ISO 15765-2 Frame Padding	0 = no padding, 1 = pad all flow controlled messages to a full CAN frame using zeroes
	5-0	Unused	Reserved for SAE – shall be set to 0

FIGURE 45—TXFLAGS BIT DEFINITIONS

9. DLL Installation and Registry

9.1 Naming of Files

Each vendor will provide a different name implementation of the API DLL and a number of these implementations could simultaneously reside on the same PC. No vendor shall name its implementation "J2534.DLL". All implementations shall have the string "32" suffixed to end of the name of the API DLL to indicate 32-bit. For example, if the company name is "Vendor X" the name could be VENDRX32.DLL. For simplicity, an API DLL shall be named in accordance with the file allocation table (FAT) file system naming convention (which allows up to eight characters for the file name and three characters for the extension with no spaces anywhere). Note that, given this criteria, the major name of an API DLL can be no greater than six characters. The OEM application can determine the name of the appropriate vendor's DLL using the Win32 Registry mechanism described in this section.

9.2 Win32 Registry

This section describes the use of the Windows Registry for storing information about the various vendors supplying the device drivers conforming to this recommended practice, the various devices supported by each vendor, information about each device, etc. The Win32 registration is shown in Figure 46.

The installation program provided with the interface shall create the appropriate registry entries listed below (including the PassThruSupport.04.04 key if not yet present). Only one key shall be created per DLL installed. The uninstall program shall remove its device-specific registry information, but shall not affect the remaining registry entries. The programming application shall use the Windows function RegEnumKeyEx (or similar) to enumerate all devices.

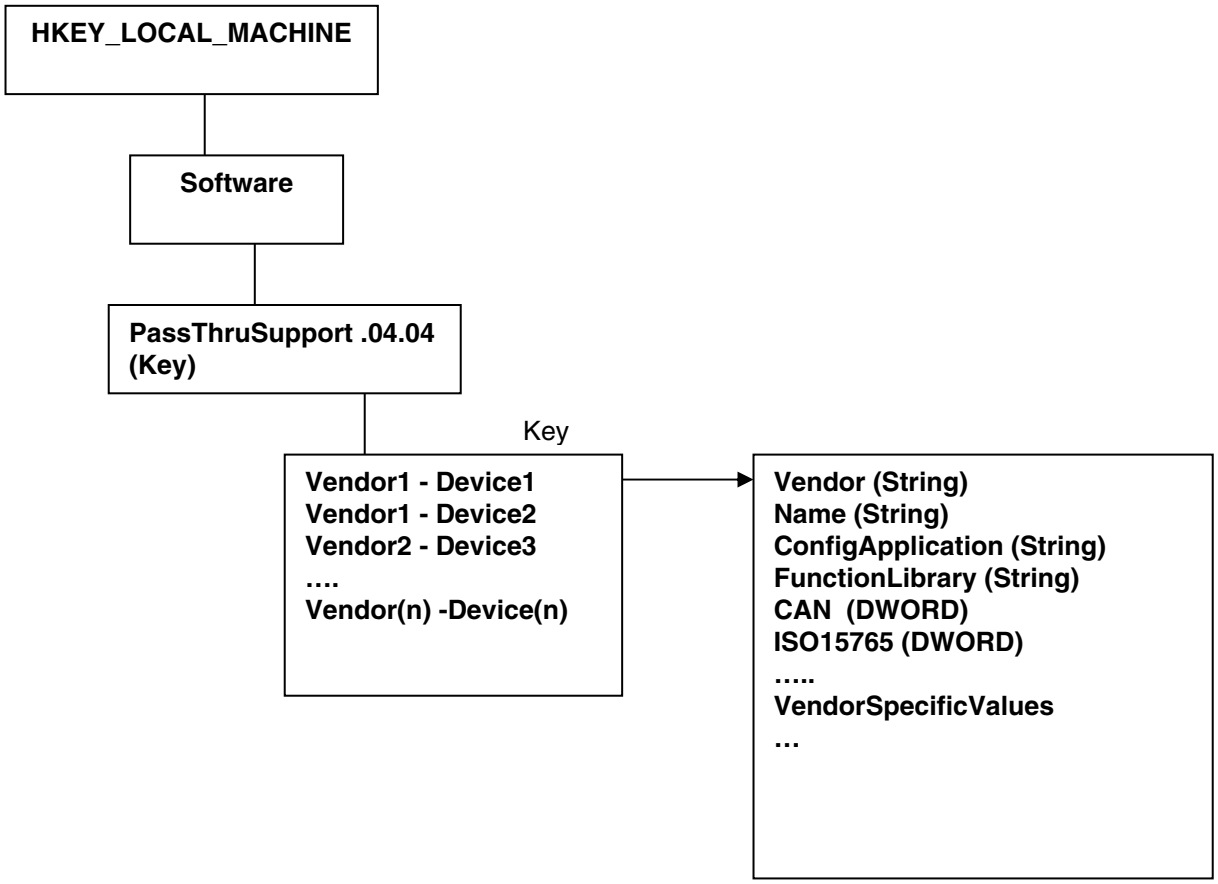

FIGURE 46—WIN32 REGISTRY

The registry will contain both:

a. General information used by the user applications for selection of hardware, user information, etc.
b. Vendor/Device specific information that the vendor uses in the implementation of the API. Considering that the object of this recommended practice is the need for interchangeability of hardware from various vendors, the user application using the this API will be required to use the registry to present to the users all the hardware devices that have been installed and display their capabilities. The user should be allowed to select any hardware having the required capabilities, in terms of protocols supported etc., for a particular reprogramming session.

Under the PassThruSupport.04.04 key will be a list of devices. Each device key will consist of the name of the vendor, a hyphen (with spaces), and the name of the device. For example, "Acme Corp.-Turtle Programmer". Under each device key the following entries (values) included in Figure 47 shall be present:

Vendor	String	The full name of the vendor. Ex: "ACME Corporation"
Name	String	The name of the device. Ex: " ACME CAN Device over Ethernet"
CAN ISO15765 J1850PWM J1850VPW ISO9141 ISO14230 SCI_A_ENGINE SCI_A_TRANS SCI_B_ENGINE SCI_B_TRANS	DWORD	For each of the supported protocols, the vendor can indicate how many simultaneous channels the hardware supports. The listing of a protocol here is only for the purpose of information and will not guarantee that the actual hardware will support the protocol, as it is possible that the hardware configuration may have changed. Ex: CAN has a value of 2, J1850PWM has a value of 1, and ISO 9141 has a value of 0. If protocol does not exist, its value is assumed to be zero.
ConfigApplication	String	The complete path of the configuration application for this device. Every device vendor is required to provide a configuration application where the user can set the different parameters required for successfully using the device, like COM port, Ethernet address etc. Ex: "c:\ACME\ACMESERCFG.exe" The user applications using the API will automatically launch this application when the user needs to configure the selected device.
FunctionLibrary	String	The complete path of the DLL supplied by the vendor to communicate with this device. The user applications using this device should automatically load the DLL specified here and map into the J2534 API functions. Ex: "C:\ACME\ACMESE32.dll"
<Vendor Specific Values>	-	The vendor will store all the vendor specific information here.

FIGURE 47—WIN32 REGISTRY VALUES

9.2.1 USER APPLICATION INTERACTION WITH THE REGISTRY

The user application should use the registry to present to the user the list of devices available for use from the application. Once the device has been selected by the user the Registry should be used to retrieve all the information regarding the device so that the appropriate DLL can be loaded for use etc. Figure 48 is a flow chart that shows a typical usage.

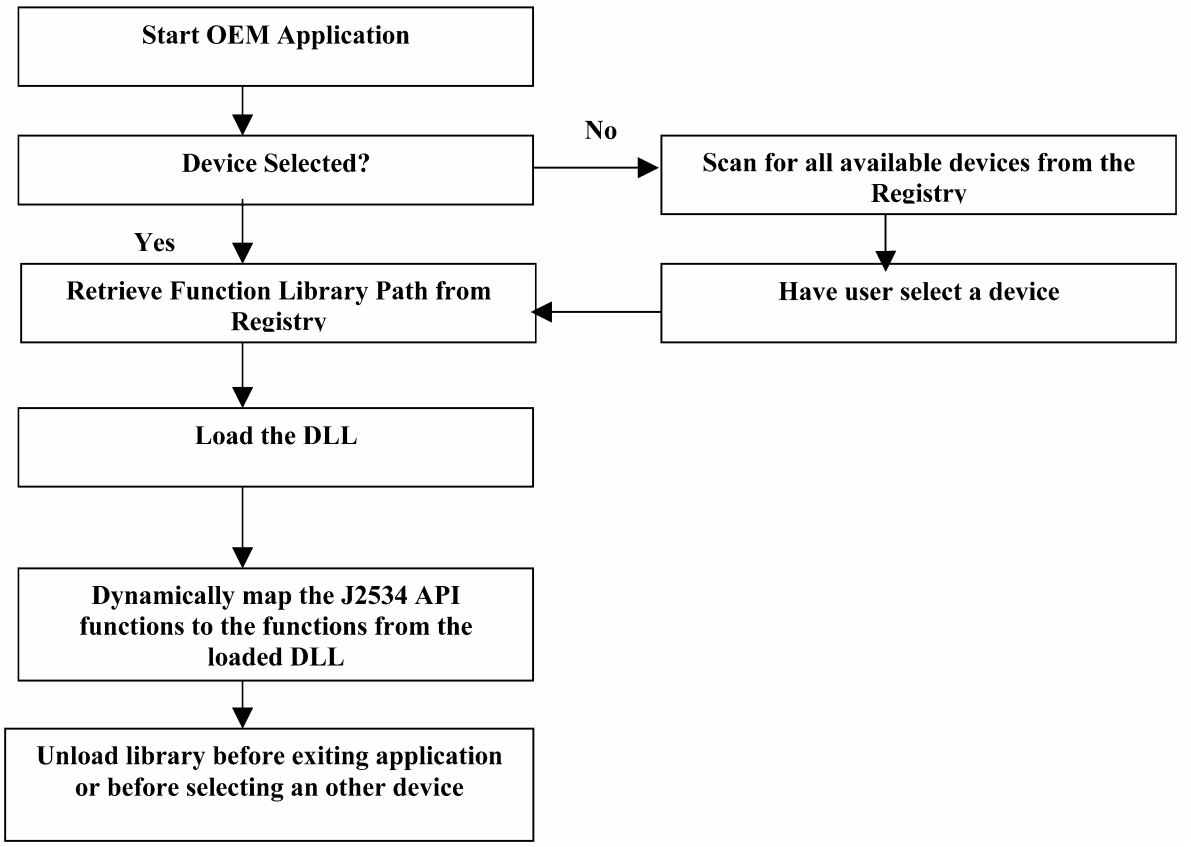

FIGURE 48—APPLICATION INTERACTION WITH REGISTRY

9.2.2 ATTACHING TO THE DLL FROM AN APPLICATION

This document requires OEM programming applications to explicitly load the appropriate DLL and resolve references to the DLL supplied functions. This is accomplished by using the native Win32 API functions, LoadLibrary, GetProcAddress and FreeLibrary (see the Win32 API SDK reference for the details of these functions).

When using GetProcAddress, the application must supply the name of the function whose address is being requested. The function names should be used with GetProcAddress in order to explicitly resolve DLL function addresses when using GetProcAddress.

To support this method, it is required that all tool vendors compile their DLL with the following export library definition file. This will help prevent name mangling and allow software developers to use the process defined in this section as well as calling by ordinal for compilers/languages that may not support that functionality.

All vendor DLLs and OEM applications shall be built with byte alignment (i.e., packing) set to one (1) byte. All DLLs and OEM applications shall set the "file version" metadata on their .EXE/.DLL.

9.2.2.1 Export Library Definition File

```
;VENDOR32.DEF:  Declares the module parameters.
LIBRARY "VENDOR32.DLL"
EXPORTS
        PassThruOpen                    @1PRIVATE
        PassThruClose                   @2PRIVATE
        PassThruConnect                 @3 PRIVATE
        PassThruDisconnect              @4 PRIVATE
        PassThruReadMsgs                @5PRIVATE
        PassThruWriteMsgs               @6 PRIVATE
        PassThruStartPeriodicMsg        @7 PRIVATE
        PassThruStopPeriodicMsg         @8 PRIVATE
        PassThruStartMsgFilter          @9 PRIVATE
        PassThruStopMsgFilter           @10 PRIVATE
        PassThruSetProgrammingVoltage   @11 PRIVATE
        PassThruReadVersion             @12 PRIVATE
        PassThruGetLastError            @13 PRIVATE
        PassThruIoctl                   @14 PRIVATE
```

10. Return Value Error Codes

Figure 49 lists the numerical equivalents and text description for the error or return codes identified in this document.

Definition	Value(s)	Description
STATUS_NOERROR	0x00	Function call successful
ERR_NOT_SUPPORTED	0x01	Device cannot support requested Functionality mandated in this Document. Device is not fully SAE J2534 compliant
ERR_INVALID_CHANNEL_ID	0x02	Invalid ChannelID value
ERR_INVALID_PROTOCOL_ID	0x03	Invalid ProtocolID value, unsupported ProtocolID, or there is a resource conflict (i.e. trying to connect to multiple protocols that are mutually exclusive such as J1850PWM and J1850VPW, or CAN and SCI A, etc.)
ERR_NULL_PARAMETER	0x04	NULL pointer supplied where a valid pointer is required
ERR_INVALID_IOCTL_VALUE	0x05	Invalid value for Ioctl parameter
ERR_INVALID_FLAGS	0x06	Invalid flag values
ERR_FAILED	0x07	Undefined error, use PassThruGetLastError for text description
ERR_DEVICE_NOT_CONNECTED	0x08	Device ID invalid

ERR_TIMEOUT	0x09	Timeout. PassThruReadMsg: No message available to read or could not read the specified number of messages. The actual number of messages read is placed in <NumMsgs> PassThruWriteMsg: Device could not write the specified number of messages. The actual number of messages sent on the vehicle network is placed in <NumMsgs>.
ERR_INVALID_MSG	0x0A	Invalid message structure pointed to by pMsg (Reference Section 8 – Message Structure)
ERR_INVALID_TIME_INTERVAL	0x0B	Invalid TimeInterval value.
ERR_EXCEEDED_LIMIT	0x0C	Exceeded maximum number of message IDs or allocated space.
ERR_INVALID_MSG_ID	0x0D	Invalid MsgID value.
ERR_DEVICE_IN_USE	0x0E	Device is currently open.
ERR_INVALID_IOCTL_ID	0x0F	Invalid IoctlID value.
ERR_BUFFER_EMPTY	0x10	Protocol message buffer empty, no messages available to read.
ERR_BUFFER_FULL	0x11	Protocol message buffer full. All the messages specified may not have been transmitted.
ERR_BUFFER_OVERFLOW	0x12	Indicates a buffer overflow occurred and messages were lost.
ERR_PIN_INVALID	0x13	Invalid pin number, pin number already in use, or voltage already applied to a different pin.
ERR_CHANNEL_IN_USE	0x14	Channel number is currently connected.
ERR_MSG_PROTOCOL_ID	0x15	Protocol type in the message does not match the protocol associated with the Channel ID
ERR_INVALID_FILTER_ID	0x16	Invalid Filter ID value
ERR_NO_FLOW_CONTROL	0x17	No flow control filter set or matched (for protocolID ISO15765 only).
ERR_NOT_UNIQUE	0x18	A CAN ID in pPatternMsg or pFlowControlMsg matches either ID in an existing FLOW_CONTROL_FILTER

ERR_INVALID_BAUDRATE	0x19	The desired baud rate cannot be achieved within the tolerance specified in Section 6.5
ERR_INVALID_DEVICE_ID	0x1A	Unable to communicate with device
Reserved	0x1B-0xFFFF	Reserved for SAE J2534-1
Reserved	0x10000-0xFFFFFFFF	Reserved for SAE J2534-2

FIGURE 49—ERROR VALUES

11. Notes

11.1 Marginal Indicia

The change bar (l) located in the left margin is for the convenience of the user in locating areas where technical revisions have been made to the previous issue of the report. An (R) symbol to the left of the document title indicates a complete revision of the report.

PREPARED BY THE SAE PASS-THRU PROGRAMMING SAE J2534 TASK FORCE
OF THE SAE VEHICLE E/E SYSTEMS DIAGNOSTICS STANDARD COMMITTEE

APPENDIX A
GENERAL ISO 15765-2 FLOW CONTROL EXAMPLE

A.1 Flow Control Overview

ISO 15765-2 was designed to send blocks of up to 4095 bytes on top of the limited 8-byte payload of raw CAN frames. If the data is small enough, it can fit in a single frame and be transmitted like a raw CAN message with additional headers. Otherwise, the block is broken up into segments and becomes a segmented transmission, generating CAN frames in both directions. For flexibility, the receiver of the segments can control the rate at which the segments are sent.

Each transmission is actually part of a conversation between two nodes. There is no discovery mechanism for conversation partners. Therefore, each desired conversation must be pre-defined on each side before the conversation can start. Conversations are symmetric, meaning that either side can send a block of data to the other. A conversation can only have one transfer (in one direction) in progress at a time. One transfer must complete before the next transfer (in the same or in a different direction) can start. The device must support multiple transfers at once, as long as each one is part of a different conversation. Raw CAN frames are not allowed when using ISO15765-2.

A key feature of a conversation is that each side has a unique CAN ID, and each side uses their unique CAN ID for all transmissions during the conversation. No other CAN IDs are part of the conversation. Even though the useful data is only flowing in one direction, both sides are transmitting. One side is sending the flow control message to pace the segments of data coming from the other side.

For example, during OBD communication, a pass-thru device and an ECU might have a conversation. The pass-thru device will use the "Tester1" physical CAN ID ($241), and the first ECU will use the "ECU1" physical CAN ID ($641). During a multi-segment transfer, both sides will be transmitting using only their respective IDs. It does not matter if the data is being sent by the ECU or by the Tester, the IDs remain the same.

It is important to understand the difference between OBD Requests/Responses and ISO 15765-2 transfers. The OBD Request is transmitted from the Tester to the ECU using functional addressing. Because segmented transfer is not possible on functional addresses, the message must fit in a single frame. The OBD Response is a message from the ECU to the Tester using physical addressing. Unlike other protocols, the responses are not sequential. In fact, the responses can overlap, as if each ECU were having a private conversation with the Tester. Some of the responses may fit in a single frame, while others will require a segmented transfer from the ECU to the tester.

A.1.1 Examples Overview

This appendix contains several examples of a transmission using ISO 15765-2. These examples assume that normal addressing is used (no extended address present), and that the CAN identifier assignments shown in Figure A1 apply.

CAN ID	CAN ID type	Usage
$241	Physical request CAN ID	For the transmission of a request message from the pass-thru interface to the ECU this CAN ID Is used by the interface for: • FirstFrame • ConsecutiveFrame(s) For the reception of a response message from the ECU this CAN ID is used by the pass-thru interface for: • FlowControl frame
$641	Response CAN ID	For the reception of a request message from the pass-thru interface this CAN ID is used by the ECU for: • FlowControl frame For the transmission of a response message from the ECU to the pass-thru interface this CAN ID Is used by the ECU for: • FirstFrame • ConsecutiveFrame(s)

FIGURE A1—CAN IDENTIFIER ASSIGNMENT EXAMPLE

In these examples, we assume that the application has called PassThruOpen and PassThruConnect to create an ISO 15765 channel. Before the conversation is setup, we cannot receive any messages, nor transmit any segmented messages. After setup, we can both transmit and receive. Setting up a conversation does not actually transmit anything. It just informs the pass-thru device which conversations to handle.

During the following conversations, the pass-thru device will always send messages with a CAN ID of $241, and the ECU will always send messages with $641.

A.2 Transmitting a Segmented Message

When PassThruWrite is called, the API will search the list of flow control filters, looking for a pFlowControlMsg that matches the CAN ID (and possible extended address) of the message being sent. Upon matching a filter, the pass-thru device will:

- Start the ISO 15765 transfer by sending a FirstFrame on the bus. The CAN ID of this segment was specified in both the message and the matching pFlowControlMsg. In our example, this is $241.
- Wait for a FlowControl frame from the conversation partner. The CAN ID to look for is specified in the corresponding pPatternMsg. In our example, this is $641.
- Transmit the message data in ConsecutiveFrames according to the FlowControl frame's instructions for BS (BlockSize) and STmin (SeparationTime minimum). Again, the pass-thru device transmits using CAN ID specified in pFlowControlMsg. In our example, this is $241.
- Repeat the previous two steps as required.
- When finished, the pass-thru device will place a TxDone indication in the API receive queue. The data will contain the CAN ID specified in pFlowControlMsg. In our example, this is $241.
- If loopback is on, the entire message sent will appear in the API receive queue with the TX_MSG_TYPE bit set to 1 in RxStatus. The loopback shall not precede the TxDone indication.

A.2.1 Conversation Setup

Before any multi-segment transfer can take place, the conversation must be set up on both sides. It's assumed that the ECU is already setup. The application is responsible for setting up the pass-thru device. This setup must be done once (and only once) per conversation. The setup involves a single call to PassThruStartMsgFilter, with the following parameters:

ChannelID: Contains the value retrieved previously via the PassThruConnect function for the ISO 15765 protocol.

FilterType: Must be FLOW_CONTROL_FILTER

pMaskMsg: Pointer to a PASSTHRU_MSG that contains the receive message mask. The structure members are set as follows (note that all bits are relevant to be filtered on for the given example):

ProtocolID:	Must be ISO15765
RxStatus:	Don't care / not used
TxFlags:	Should be zero except for CAN_29BIT_ID and ISO15765_ADDR_TYPE. In our examples, they will both be zero, so the resulting TxFlags value is 00000000 hex.
TimeStamp:	Don't care / not used
DataSize:	4 because we are filtering on CAN ID only. Could be 5 if we were using extended addressing.
ExtraDataIndex:	Don't care / not used
Data:	FF FF FF FF hex

pPatternMsg: Pointer to a PASSTHRU_MSG that contains the conversation partner's ID. The structure members are set as follows:

ProtocolID:	Must be ISO15765
RxStatus:	Don't care / not used
TxFlags:	Must be the same as pMaskMsg.
TimeStamp:	Don't care / not used
DataSize:	Must be the same as pMaskMsg.
ExtraDataIndex:	Don't care / not used
Data:	00 00 06 41 hex to indicate will be conversing with an ECU using the CAN ID of $641.

pFlowControlMsg: Pointer to a PASSTHRU_MSG that contains the our ID. The structure members are set as follows:

ProtocolID:	Must be ISO15765
RxStatus:	Don't care / not used
TxFlags:	Must be the same as pMaskMsg.
TimeStamp:	Don't care / not used
DataSize:	Must be the same as pMaskMsg.
ExtraDataIndex:	Don't care / not used
Data:	00 00 02 41 hex to indicate will be conversing using the Tester CAN ID of $241.

pMsgID: Pointer to storage location for filter reference identifier (later used to delete filter).

A.2.2 Data Transmission

Once the conversation is set up, any number of messages (to the conversation partner) can be transmitted using PassThruWriteMsg. The interface shall handle all aspects of the transfer, including pacing (slowing) the transmission to the requirements of the receiver.

When there are multiple conversations setup, the pass-thru device will search all of the flow control filters for a matching pFlowControlMsg. If there is no match, the message cannot be sent because the pass-thru device doesn't know which partner will be pacing the conversation.

When doing blocking writes, it is important to pick a timeout long enough to cover entire transfer, even if the ECU is pacing things slowly. Otherwise PassThruWriteMsg will return with a timeout, even though the transmission is proceeding normally.

In this example, the application calls PassThruWriteMsg with the following parameters:

ChannelID: Contains the value retrieved previously via the PassThruConnect function for the ISO 15765 protocol.

pMsg: Pointer to a PASSTHRU_MSG that contains the message we want to transmit. The structure members are set as follows:

ProtocolID:	Must be ISO15765
RxStatus:	Don't care / not used
TxFlags:	00000000 hex. Should be zero except for CAN_29BIT_ID and ISO15765_ADDR_TYPE. In our examples, they will both be zero.
TimeStamp:	Don't care / not used
DataSize:	14. The size of our example message, including the CAN ID.
ExtraDataIndex:	Don’t care / not used
Data:	00 00 02 41 01 02 03 04 05 06 07 08 09 0a hex. The first 4 bytes are the Tester CAN ID of $241, so the pass-thru device can find the correct flow control filter. This is followed by 10 data bytes.

pNumMsgs: Pointer to storage location for number of messages. On input, this must be 1 because we are only passing a single PASSTHRU_MSG. On output, this can be 0 if the message did not go out in the time allotted, or 1 if the message transmission did complete.

Timeout 0 ms. This indicates that the application will not wait for the transfer to take place. Instead, the application will do it’s waiting in PassThruReadMsg. An alternate strategy would be to use a large Timeout here (large enough for the slowest ECU to receive all the data), and a smaller timeout on PassThruReadMsg.

If the API returns ERR_NO_FLOW_CONTROL, then the flow control filter was not set up properly.

A.2.3 Verification

Because the message transmission is paced by the receiver, there is no way to know in advance how long it will take. To help applications, the pass-thru device will generate a TxDone indication when the last frame of the message goes out. Note that ISO 15765-2 is an unacknowledged transfer, so there are no guarantees that the receiver actually got the message correctly. Also, if the transfer fails, no TxDone will arrive.

After our sample message transmission, we should call PassThruReadMsg to look for the indication. The parameters are as follows:

ChannelID: Contains the value retrieved previously via the PassThruConnect function for the ISO 15765 protocol.

pMsg: Pointer to a PASSTHRU_MSG that will contain the next message or indication in the API receive queue. The structure does not need to be initialized. After the function call, the following fields will be filled in by the API:

ProtocolID:	Must be ISO15765
RxStatus:	A TxDone indication is composed of: TX_MSG_TYPE = 1, TX_DONE=1 and ISO15765_PADDING_ERROR = 0.
TxFlags:	Don't care / not used
TimeStamp:	The time the message transmission completed.
DataSize:	4 (or 5 with extended addressing). The TxDone indication contains only the CAN ID of our transmission.
ExtraDataIndex:	0. All indications have EDI of zero.
Data:	00 00 02 41 hex. This is the Tester CAN ID of $241 to remind us which message just finished.

pNumMsgs: Pointer to storage location for number of messages. On input, this must be 1 because we are only expecting a single PASSTHRU_MSG. On output, this can be 0 if there were no messages or indications in the time allotted, or 1 if the there was a message or indication.

Timeout: 1000ms. This contains the time to wait for a message or indication. In the PassThruWriteMsg call above we did not wait for the transfer to complete by using blocking writes. Therefore, when we make this call to PassThruReadMsg, the transfer will be just starting. This Timeout must be long enough for the transfer to complete, even if the ECU requests slow pacing.

NOTE—This example assumes that the receive buffer was empty before the PassThruWriteMsg call, and that no ECUs decided to send messages in the meantime. A real application should use defensive programming techniques to process or discard unexpected messages.

A.3 Transmitting an Unsegmented Message

As a special case, transfers that fit in a single frame can be transmitted without setting up a conversation. This is useful during an OBD Request, which is a functionally addressed message that is broadcast to all ECUs. This message must be small enough to fit into a single frame (including headers) because it is not possible to do one segmented transfer to multiple ECUs.

When using functional addressing for an OBD Request, it is important to remember that there can be no direct reply. Instead, each ECU will send their OBD Response using physical addressing to their conversation partner (e.g. ECU1 to Tester1, ECU2 to Tester2) as defined by ISO 15765-4. The OBD Response may be a segmented transfer, or it may be a single frame.

A.3.1 Data Transmission

In this case, no conversation setup is necessary. The call to PassThruWriteMsg is the same as above, except that the DataSize must be 7 bytes or less (6 bytes or less if extended addressing is turned on). The pass-thru device will automatically insert a PCI byte before transmission.

A.3.2 Verification

Verification of the TxDone is the same as above.

A.4 Receiving a Segmented Message

Message reception is asynchronous to the application. When a FirstFrame is seen on the bus, the pass-thru device will search the list of flow control filters, looking for a pPatternMsg message with the same CAN ID (and possible extended address) as the FirstFrame. Upon matching a filter, the pass-thru device will:

- Place an RxStart indication in the API receive queue. This indication has the START_OF_MESSAGE bit set in RxFlags. The message data will contain the CAN ID of the sender. In our example, this is $641. DataSize will be 4 bytes (5 with extended addressing), and ExtraDataIndex will be zero.
- Send a FlowControl frame to the conversation partner. The FlowStatus field shall be set to ContinueToSend. The CAN ID of this segment comes from the filter's corresponding pFlowControlMsg. In our example, this CAN ID is $241. The BS (BlockSize) and STmin (SeparationTime minimum) parameters default to zero, but can be changed with the SET_CONFIG Ioctl.
- Wait for the conversation partner to send ConsecutiveFrames containing the actual data. The partner's CAN ID is specified in pPatternMsg. In our example, this CAN ID is $641.
- Repeat as necessary until the entire block has been received. When finished, the pass-thru device will put the assembled message into the API receive queue. The CAN ID of the assembled message will be the CAN ID of the sender. In our example, this CAN ID is $641.

If the FirstFrame does not match any flow control filters, then the message must be ignored by the device.

A.4.1 Conversation Setup

No messages can be received until a conversation is setup. Each conversation setup will receive messages from exactly one CAN ID (and extended address if present). Because setup is bi-directional, the same PassThruStartMsgFilter call used for transmission will allow for message reception too.

A.4.2 Reception Notification

Segmented messages cause the API to generate an RxStart indication. This lets the application know that the device has started message reception. It may take a while before message reception is complete, especially if the application has increased BS and STmin.

To receive the indication, the application should call PassThruReadMsg with the following parameters:

ChannelID: Contains the value retrieved previously via the PassThruConnect function for the ISO 15765 protocol.

pMsg: Pointer to a PASSTHRU_MSG that will contain the next message or indication in the API receive queue. The structure does not need to be initialized. After the function call, the following fields will be filled in by the API:

ProtocolID:	Must be ISO15765
RxStatus:	An RxStart indication is composed of: START_OF_MESSAGE = 1.
TxFlags:	Don't care / not used
TimeStamp:	The time the message transmission started.
DataSize:	4 (or 5 with extended addressing). The RxStart indication contains only the CAN ID of the sender.
ExtraDataIndex:	0. All indications have EDI of zero.
Data:	00 00 06 41 hex. This is the ECU CAN ID of $641 to tell us they have started sending us a message.

pNumMsgs: Pointer to storage location for number of messages. On input, this must be 1 because we are only expecting a single PASSTHRU_MSG. On output, this can be 0 if there were no messages or indications in the time allotted, or 1 if the there was a message or indication.

Timeout: 1000ms. This contains the time to wait for a message or indication

NOTE—This example assumes that the receive buffer was empty before the PassThruReadMsg call, and that no ECUs decided to send messages in the meantime. A real application should use defensive programming techniques to process or discard unexpected messages.

A.4.3 Data Reception

Once the transfer is complete, the entire message can be read like on any other protocol. Usually, applications will call PassThruReadMsg again immediately after getting an RxStart indication. Application writers should not assume that the complete message will always follow the RxStart indication. If multiple conversations are setup, indications and messages from other conversations can be received in between the RxStart indication and the actual message. The parameters for PassThruReadMsg are exactly the same as in the previous section. The only difference is that the DataSize will be larger and ExtraDataIndex will be non-zero.

A.5 Receiving an Unsegmented Message

No messages can be received until a conversation is setup. Each conversation setup will receive messages from exactly one CAN ID (and extended address if present). Because setup is bi-directional, the same PassThruStartMsgFilter call used for transmission will allow for message reception.

When a SingleFrame is seen on the bus, the pass-thru device will search the list of flow control filters, looking for a pPatternMsg message with the same CAN ID (and possible extended address) as the SingleFrame. Upon matching a filter, the pass-thru device will strip the PCI byte and queue the packet for reception. If the SingleFrame does not match a flow control filter, it must be discarded.

The only difference between the previous cases is that single-frame messages do not generate an RxStart indication.

SAE J2534-1 Revised DEC2004

Rationale

The U.S. Environmental Protection Agency (EPA) and the California Air Resources Board (ARB) have been working with vehicle manufacturers to provide the aftermarket with increased capability to service emission-related ECU's for all vehicles with a minimal investment in hardware needed to communicate with the vehicles. Both agencies have issued regulations that will require standardized programming tools to be used for all vehicle manufacturers. The Society of Automotive Engineers (SAE) developed this Recommended Practice to satisfy the intent of the U.S. EPA and the California ARB.

The original SAE J2534 document published in February, 2002 was developed before vehicle manufacturers had created reprogramming applications using this capability and before interface manufacturers had developed hardware to be used for this capability. As this work progressed at both vehicle and interface manufacturers, additional needed functionality and areas subject to different interpretations were identified. This major revision includes all known changes and clarifications, and because of the extensive modifications required, this document is not always backwards compatible with the previous version of the document.

Relationship of SAE Standard to ISO Standard

Not applicable.

Application

This SAE Recommended Practice provides the framework to allow reprogramming software applications from all vehicle manufacturers the flexibility to work with multiple vehicle data link interface tools from multiple tool suppliers. This system enables each vehicle manufacturer to control the programming sequence for electronic control units (ECUs) in their vehicles, but allows a single set of programming hardware and vehicle interface to be used to program modules for all vehicle manufacturers.

This document does not limit the hardware possibilities for the connection between the PC used for the software application and the tool (e.g., RS-232, RS-485, USB, Ethernet…). Tool suppliers are free to choose the hardware interface appropriate for their tool. The goal of this document is to ensure that reprogramming software from any vehicle manufacturer is compatible with hardware supplied by any tool manufacturer.

U.S. Environmental Protection Agency (EPA) and the California Air Resources Board (ARB) "OBD service information" regulations include requirements for reprogramming emission-related control modules in vehicles for all manufacturers by the aftermarket repair industry. This document is intended to conform to those regulations for 2004 and later model year vehicles.For some vehicles, this interface can also be used to reprogram emission-related control modules in vehicles prior to the 2004 model year, and for non-emission related control modules. For other vehicles, this usage may require additional manufacturer specific capabilities to be added to a fully compliant interface. A second part to this document, SAE J2534-2, is planned to include expanded capabilities that tool suppliers can optionally include in an interface to allow programming of these additional non-mandated vehicle applications. In addition to reprogramming capability, this interface is planned for use in OBD compliance testing as defined in SAE J1699-3. SAE J2534-1 includes some capabilities that are not required for Pass-Thru Programming, but which enable use of this interface for those other purposes without placing a significant burden on the interface manufacturers.

Additional requirements for future model years may require revision of this document, most notably the inclusion of SAE J1939 for some heavy-duty vehicles. This document will be reviewed for possible revision after those regulations are finalized and requirements are better understood. Possible revisions include SAE J1939 specific software and an alternate vehicle connector, but the basic hardware of an SAE J2534 interface device is expected to remain unchanged.

Reference Section

SAE J1850—Class B Data Communications Network Interface

SAE J1939—Truck and Bus Control and Communications Network (Multiple Parts Apply)

SAE J1962—Diagnostic Connector

SAE J2610—DaimlerChrysler Information Report for Serial Data Communication Interface (SCI)

ISO 7637-1:1990—Road vehicles—Electrical disturbance by conduction and coupling—Part 1: Passenger cars and light commercial vehicles with nominal 12 V supply voltage

ISO 9141:1989—Road vehicles—Diagnostic systems—Requirements for interchange of digital information

ISO 9141-2:1994—Road vehicles—Diagnostic systems—CARB requirements for interchange of digital information

ISO 11898:1993—Road vehicles—Interchange of digital information—Controller area network (CAN) for high speed communication

ISO 14230-4:2000—Road vehicles—Diagnostic systems—Keyword protocol 2000—Part 4: Requirements for emission-related systems

ISO/FDIS 15765-2—Road vehicles—Diagnostics on controller area networks (CAN)—Network layer services

ISO/FDIS 15765-4—Road vehicles—Diagnostics on controller area networks (CAN)—Requirements for emission-related systems

Developed by the SAE Pass-Thru Programming SAE J2534 Task Force

Sponsored by the SAE Vehicle E/E Systems Diagnostics Standard Committee