CERTIFICATE

By Authority Of
THE UNITED STATES OF AMERICA
Legally Binding Document

By the Authority Vested By Part 5 of the United States Code § 552(a) and Part 1 of the Code of Regulations § 51 the attached document has been duly INCORPORATED BY REFERENCE and shall be considered legally binding upon all citizens and residents of the United States of America. ***HEED THIS NOTICE***: Criminal penalties may apply for noncompliance.

Document Name: SAE J2534: Recommended Practice for Pass-Thru Vehicle Programming

CFR Section(s): 40 CFR 86.096-38(g)(17)(iv)

Standards Body: Society of Automotive Engineers

Official Incorporator:

THE EXECUTIVE DIRECTOR
OFFICE OF THE FEDERAL REGISTER
WASHINGTON, D.C.

RECOMMENDED PRACTICE FOR PASS-THRU VEHICLE PROGRAMMING—SAE J2534 FEB2002

SAE Recommended Practice

Report of the SAE Pass-Thru Programming SAE J2534 Task Force of the SAE Vehicle E/E Systems Diagnostics Standard Committee approved February 2002. Rationale statement available.

Foreword—The use of reprogrammable memory technology in vehicle electronic control units (ECU's) 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 ECU's 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 ECU's 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 ECU's.

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 proposed 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.

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 (ECU's) 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.

The U.S. Environmental Protection Agency (EPA) and the California Air Resources Board (ARB) have proposed requirements for reprogramming vehicles for all manufacturers by the aftermarket repair industry. This document is intended to meet those proposed requirements for 2004 model year vehicles. Additional requirements for the 2005 model year 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/DIS 15765-2—Road vehicles—Diagnostics on controller area networks (CAN)—Network layer services
ISO/DIS 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 for Character 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.

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

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 manufacturer of an SAE J2534 pass-thru device must supply both the device driver software and the pass-thru device hardware that communicates directly with the vehicle. 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.

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:

Clause	Test Description	See ISO	Certification Initial	Certification Periodic[1]	If Required[2] Initial	If Required[2] Periodic[1]
5	Dimensions					
5.1	Outside Cable Diameter and Ovality	14572	x	x		
5.2	Thickness of the Sheath	14572	x	x		
5.3	Visual Appearance	14572				
6	Electrical Characteristics					
6.1	Continuity	14572	x	x		
6.2	Withstand Voltage	14572	3)	3)		
7	Mechanical Characteristics					
7.1	Adhesion of the Sheath	14572	x	x		
7.2	Cyclic Bending	14572			x	x
8	Low temperature Characteristics					
8.1	Winding	14572	x	x		
9	Resistance to Abrasion	14572	x	x		
10	Heat Aging					
10.1	Short Term Aging 240 h	14572	x	x		
10.2	Long Term Aging 3000 h	14572	x			
11	Resistance to Chemicals					
11.1	Fluid Compatibility of the Sheath	14572	x	x		
11.2	Resistance to Ozone	14572			x	
12	Resistance to Flame Propagation	14572	x	x		

NOTE 1: The frequency of periodic testing will be established by agreement between the customer and the supplier.
NOTE 2: The usage of "if required" tests will be established by agreement between customer and supplier.
NOTE 3: Some cables are rated at 60 V and others at 600 V. See clause 3 for details.

FIGURE 1—SAE J2534 OVERVIEW

6. Pass-Thru System Requirements

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 communicated on sequentially (serialized) from the single application thread. This will prevent the unnecessary complexity of determining what message responses belong to which application thread.

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. Vehicle manufacturers will need to supply information about necessary connections to any connector other than the SAE J1962 connector.

6.5 Communication Protocols—A fully compliant pass-thru interface shall support all communication protocols as specified in this section. Additionally, the pass-thru device must support simultaneous communication of an ISO 9141 OR ISO 14230-4 protocol AND an SAE J1850 protocol AND a CAN or SCI based protocol during a single programming event. Note that only one type of SAE J1850 is required per programming event, as the two types of SAE J1850 are mutually exclusive on any given vehicle. As well, CAN and SCI are mutually exclusive on some vehicles as the same pins are used.

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 minimum 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 rates with ±2% tolerance: 9600, 9615, 10000, 10870, 11905, 12500, 13158, 13889, 14706, and 15625.

f. 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.

6.5.2 ISO 14230-4 (KWP2000)—The ISO 14230-4 protocol is the same as the ISO 9141 protocol with the following additions:

a. The interface will handle the tester present message and 0x78 response code automatically (i.e., without intervention from the PC).

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 for 41.6 kbps PWM:

a. Capable of 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 for 10.4 kbps VPW:

a. High speed mode of 41.6 kbps

b. 4K block transfer

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

a. 250 and 500 kbps

b. 11 and 29 bit identifiers

c. Support for 80% ± 2% and 68.5% ± 2% bit sample point

d. Pass-thru message interface (i.e., raw CAN frames with no flow control in the pass-thru device)

6.5.6 ISO 15765-4 (CAN)—The following features of ISO 15765-4 must be supported by the pass-thru device:

a. 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.

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

6.6 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. 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

b. Maximum 20 V

c. Accuracy ±0.1 V

d. Maximum source current 200 mA

e. Maximum sink current 300mA (only for SHORT_TO_GROUND option).

f. Maximum 1 ms settling time (required for SCI protocol, reference SAE J2610 Information Report)

g. Pin assignment software selectable

6.7 Data Buffering—The interface shall be capable of buffering a 4K byte transmit message as well as a 4K byte receive message.

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 2.

Function	Description
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 2—SAE J2534 API FUNCTIONS

7.2 API Functions - Detailed Information

7.2.1 PASSTHRUCONNECT—This function is used to establish a logical connection with a protocol channel. After this function is called, the value pointed to by pChannelID is used as the logical identifier for the connection. The DLL can use this function to initialize data structures and device drivers. If the function operates successfully, 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 all filters for the given protocol will be cleared with this function.

7.2.1.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruConnect
(
    unsigned long ProtocolID,
    unsigned long Flags,
    unsigned long *pChannelID
)
```

7.2.1.2 Parameters

ProtocolID	Protocol ID.
Flags	Connection flags, normally set to zero.
pChannelID	Pointer to location for the channel ID that is assigned by the DLL.

7.2.1.3 Flag Values—See Figure 3.

Flags Bit(s)	Description	Value
31-9	Unused	Tool manufacturer specific
8	CAN ID type	0 = 11-bit, 1 = 29-bit
7	ISO15765-2 Addressing Method	0 = no extended address, 1 = extended address is first byte after the ID bytes
6-0	Unused	Reserved for SAE - shall be set to 0

FIGURE 3—FLAG VALUES

7.2.1.4 ProtocolID Values—See Figure 4.

Definition	Description	Value(s)
J1850VPW	GM / DaimlerChrysler CLASS2	0x01
J1850PWM	Ford SCP	0x02
ISO9141	ISO9141 and ISO9141-2	0x03
ISO14230	ISO14230-4 (Keyword Protocol 2000)	0x04
CAN	Raw CAN (flow control not handled automatically by interface)	0x05
ISO15765	ISO15765-2 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
Unused	Reserved for SAE use	0x0B – 0xFFFF
Unused	Tool manufacturer specific	0x10000 – 0xFFFFFFFF

FIGURE 4—PROTOCOL ID VALUES

7.2.1.5 Return Values—See Figure 5.

7.2.2 PASSTHRUDISCONNECT—This function is used to terminate a logical connection with a protocol channel. The DLL can use this function to de-allocate data structures and deactivate any device drivers. If the function operates successfully, a value of STATUS_NOERROR is returned. After this call the Channel ID will no longer be valid.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_PROTOCOL_ID	Invalid ProtocolID value 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_NULLPARAMETER	NULL pointer supplied where a valid pointer is required
ERR_INVALID_FLAGS	Invalid flag values.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_CHANNEL_IN_USE	Channel number is currently connected.

FIGURE 5—RETURN VALUES

7.2.2.1 *C / C++ Prototype*

```
extern "C" long WINAPI PassThruDisconnect
(
    unsigned long ChannelID
)
```

7.2.2.2 *Parameters*

ChannelID The channel ID assigned by the PassThruConnect function.

7.2.2.3 *Return Values*—See Figure 6.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.

FIGURE 6—RETURN VALUES

7.2.3 PASSTHRUREADMSGS—This function reads messages from the receive buffer in the order they were received. If the function operates successfully, a value of STATUS_NOERROR is returned. Note that the ISO 15765-2 FirstFrame and TxDone indications will be returned as messages when calling this function. Also note that all messages and indications shall be read in the order that they occurred on the bus.

7.2.3.1 *C / C++ Prototype*

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

7.2.3.2 *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 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.

7.2.3.3 *Return Values*—See Figure 7.

7.2.4 PASSTHRUWRITEMSGS—This function is used to send messages. The messages are placed in the buffer and sent in the order they were received. If the function operates successfully, a value of STATUS_NOERROR is returned. To perform blocking writes (i.e., the function does not return until message is successfully sent on the vehicle network or a timeout occurs), use the blocking flag in the TxFlags element of the message structure (Reference 8.4.2).

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_NULLPARAMETER	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 should be returned.
ERR_BUFFER_EMPTY	No messages available to read.
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 7—RETURN VALUES

7.2.4.1 *C / C++ Prototype*

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

7.2.4.2 *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 or placed in the transmit queue.
- Timeout — Write timeout (in milliseconds). If a value of 0 is specified the function 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 written. If the number of messages requested have been written, the function shall not return ERR_TIMEOUT, even if the timeout value is zero.

7.2.4.3 *Return Values*—See Figure 8.

7.2.5 PASSTHRUSTARTPERIODICMSG—This function starts sending a message at the specified interval. If the function operates successfully, a value of STATUS_NOERROR is returned. The maximum number of periodic messages is ten.

7.2.5.1 *C / C++ Prototype*

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

7.2.5.2 *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.

7.2.5.3 *Return Values*—See Figure 9.

7.2.6 PASSTHRUSTOPPERIODICMSG—This function stops the process of sending a periodic message. If the function operates successfully, a value of STATUS_NOERROR is returned. After this call the MsgID will be invalid.

7.2.6.1 C / C++ Prototype

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

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by pMsg (e.g. sending a 20 byte long J1850PWM message, sending a J1850PWM message where the third data byte is not the same as the node ID, etc.).
ERR_NULLPARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_TIMEOUT	Timeout.
ERR_MSG_PROTOCOL_ID	Protocol type in the message does not match protocol associated with the ChannelID
ERR_BUFFER_FULL	Protocol message buffer is full.

FIGURE 8—RETURN VALUES

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by pMsg.
ERR_NULLPARAMETER	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 protocol associated with the ChannelID
ERR_EXCEEDED_LIMIT	Exceeded the maximum number of periodic message IDs or the maximum allocate space.

FIGURE 9—RETURN VALUES

7.2.6.2 Parameters

ChannelID — The channel ID assigned by the PassThruConnect function.

MsgID — Message ID that is assigned by the PassThruStartPeriodicMsg function.

7.2.6.3 Return Values—See Figure 10.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_MSG_ID	Invalid MsgID value.

FIGURE 10—RETURN VALUES

7.2.7 PASSTHRUSTARTMSGFILTER—This function starts filtering incoming messages. If the function operates successfully, a value of STATUS_NOERROR is returned. The maximum number of message filters is ten. See Appendices A and B for a description of the use of these message filters for transmission and reception of multi-frame messages.

7.2.7.1 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 *pMsgID
)
```

7.2.7.2 Parameters

ChannelID — The channel ID assigned by the PassThruConnect function.

FilterType — Designates:

PASS_FILTER – allows matching messages into the receive queue.

BLOCK_FILTER - keeps matching messages out of the receive queue.

FLOW_CONTROL_FILTER – defines a filter and outgoing flow control messageto support the ISO 15765-2 flow control mechanism.

pMaskMsg — 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.

The usage of the pMaskMsg allows for configuring a filter that passes thru multiple CAN identifiers. In case the filter allows for the reception of multiple CAN identifiers then those messages are only allowed to be Single-Frame messages, because only a single FlowControl CAN identifier can be specified.

pPatternMsg — 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 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".

pFlowControlMsg — Designates a pointer to an ISO 15765-2 flow control message. This message will be sent out when the received message ANDed with the message pointed to by pMaskMsg matches the message pointed to by pPatternMsg and the interface is receiving a segmented message. This message shall only contain the message ID (and extended address byte if the ISO15765_EXT_ADDR flag is set). The interface will provide the PCI bytes when this message is transmitted. To modify the BS and STmin values that are used by the interface, reference the IOCTL section. This pointer only applies to the FLOW_CONTROL_FILTER type and must be set to NULL when the FilterType is PASS_FILTER or BLOCK_FILTER.

pMsgID — Pointer to location for the message ID that is assigned by the DLL.

7.2.7.3 Filter Type Values—See Figure 11.

Definition	Value
PASS_FILTER	0x00000001
BLOCK_FILTER	0x00000002
FLOW_CONTROL_FILTER	0x00000003

FIGURE 11—FILTER TYPE VALUES

7.2.7.4 Return Values—See Figure 12.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_MSG	Invalid message structure pointed to by pMsg.
ERR_NULLPARAMETER	NULL pointer supplied where a valid pointer is required.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_EXCEEDED_LIMIT	Exceeded the maximum number of filter message IDs or the maximum allocate space.
ERR_MSG_PROTOCOL_ID	Protocol type in the message does not match protocol associated with the ChannelID

FIGURE 12—RETURN VALUES

7.2.8 PASSTHRUSTOPMSGFILTER—This function stops the process of filtering messages. If the function operates successfully, a value of STATUS_NOERROR is returned. After this call the MsgID will be invalid.

7.2.8.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruStopMsgFilter
(
    unsigned long ChannelID,
    unsigned long MsgID
)
```

7.2.8.2 Prameters

ChannelID The channel ID assigned by the PassThruConnect function.

MsgID Message ID that is assigned by the PassThruStartMsgFilter function.

7.2.8.3 Return Values—See Figure 13.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_INVALID_MSG_ID	Invalid MsgID value.

FIGURE 13—RETURN VALUES

7.2.9 PASSTHRUSETPROGRAMMINGVOLTAGE—This function sets a programming voltage on a specific pin. If the function operates successfully, 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.

7.2.9.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruSetProgrammingVoltage
(
    unsigned long PinNumber,
    unsigned long Voltage
)
```

7.2.9.2 Parameters

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 200mA with a resolution of ±100 millivolts for pins 0, 6, 9, 11, 12, 13, and 14).

VOLTAGE_OFF – To turn output off (disconnect).

SHORT_TO_GROUND – Short pin to ground (pin 15 only).

7.2.9.3 Voltage Values—See Figure 14.

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

FIGURE 14—VOLTAGE VALUES

7.2.9.4 Return Values—See Figure 15.

Definition	Description
STATUS_NOERROR	Function call successful.
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC.
ERR_NOT_SUPPORTED	Function not supported.
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_PIN_INVALID	Invalid pin number specified.

FIGURE 15—RETURN VALUES

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

7.2.10.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruReadVersion
(
    char*pFirmwareVersion,
    char*pDllVersion,
    char*pApiVersion
)
```

7.2.10.2 Parameters

pFirmwareVersion Pointer to Firmware version string in XX.YY format (e.g., 01.01). This string is determined by the interface vendor that supplies the device.

pDllVersion Pointer to DLL version string in XX.YY format (e.g., 01.01). This string is determined by the interface vendor that supplies the DLL.

pApiVersion Pointer to API version string in XX.YY format. This string corresponds to the date of the balloted document.

October 2001 Ballot = "01.01"
December 2001 Ballot = "01.02"
February 2002 Final = "02.02"

7.2.10.3 Return Values—See Figure 16.

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC
ERR_FAILED	Undefined error, use PassThruGetLastError for text description
ERR_NULLPARAMETER	NULL pointer supplied where a valid pointer is required

FIGURE 16—RETURN VALUES

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

7.2.11.1 C / C++ Prototype

```
extern "C" long WINAPI PassThruGetLastError
(
    char *pErrorDescription
)
```

7.2.11.2 Parameters

pErrorDescription Pointer to error description string.

7.2.11.3 Return Values—See Figure 17.

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

FIGURE 17—RETURN VALUES

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

7.2.12.1 C / C++ Prototype

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

7.2.12.2 Parameters

ChannelID The channel ID assigned by the PassThruConnect function.
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).

7.2.12.3 Ioctl ID Values—See Figure 18.

7.2.12.4 Return Values—See Figure 19.

7.3 IOCTL Section—Figure 20 provides the details on the IOCTLs available through PassThruIoctl function:

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 – 0xFFFF
Tool manufacturer specific	0x10000 – 0xFFFFFFFF

FIGURE 18—IOCTL ID VALUES

Definition	Description
STATUS_NOERROR	Function call successful
ERR_DEVICE_NOT_CONNECTED	Device not connected to PC
ERR_INVALID_CHANNEL_ID	Invalid ChannelID value.
ERR_INVALID_IOCTL_ID	Invalid IoctlID value.
ERR_NULLPARAMETER	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

FIGURE 19—RETURN VALUES

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	Pointer to PASSTHRU_MSG	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, thus stopping all periodic message transmission
CLEAR_MSG_FILTERS	NULL pointer	NULL pointer	To direct the pass-thru device to clear all message filters, thus stopping all filtering
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 20—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 21. When the function is successfully completed, the corresponding parameter value(s) indicated in Figures 23A, 23B, and 23C will be placed in each Value.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>IoctlID</td><td>Is set to the define GET_CONFIG.</td></tr>
<tr><td>InputPtr</td><td>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 23 for a list of valid parameters).
Value is an OUTPUT that represents the value of that parameter (See Figure 23 for a list of valid values).</td></tr>
<tr><td>OutputPtr</td><td>Is a NULL pointer, as this parameter is not used.</td></tr>
</table>

FIGURE 21—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 22. When the function is successfully completed the corresponding parameter(s) and value(s) indicated in Figures 23A, 23B, and 23C will be in effect.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>IoctlID</td><td>Is set to the define SET_CONFIG.</td></tr>
<tr><td>InputPtr</td><td>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 23 for a list of valid parameters).
Value is an INPUT that represents the value of that parameter (See Figure 23 for a list of valid values).</td></tr>
<tr><td>OutputPtr</td><td>Is a NULL pointer, as this parameter is not used.</td></tr>
</table>

FIGURE 22—SET_CONFIG DETAILS

Valid values for Parameter	ID Value	Valid values for Value	Description
DATA_RATE	0x01	5-500000	Represents the desired baud rate. **There is no default value.**
Unused	0x02		Reserved for SAE
LOOPBACK	0x03	0 (OFF) 1 (ON)	0 = Don't echo transmitted messages in the receive queue. 1 = Echo transmitted messages in the receive queue. **The default value is OFF.**
NODE_ADDRESS	0x04	0x00-0xFF	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)	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). **The default value is BUS_NORMAL.**
P1_MIN	0x06	0x0-0xFFFF	For protocol ID of ISO9141, this sets the minimum inter-byte time (in milli-seconds) for ECU responses. **The default value is 0 milli-seconds.**
P1_MAX	0x07	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum inter-byte time (in milli-seconds) for ECU responses (in milli-seconds). **The default value is 20 milli-seconds.**
P2_MIN	0x08	0x0-0xFFFF	For protocol ID of ISO9141, this sets the minimum time (in milli-seconds) between tester request and ECU responses or two ECU responses. **The default value is 25 milli-seconds.**

FIGURE 23A—IOCTL GET_CONFIG / SET_CONFIG PARAMETER DETAILS

Valid values for Parameter	ID Value	Valid values for Value	Description
P2_MAX	0x09	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) between tester request and ECU responses or two ECU responses. **The default value is 50 milli-seconds.**
P3_MIN	0x0A	0x0-0xFFFF	For protocol ID of ISO9141, this sets the minimum time (in milli-seconds) between end of ECU response and start of new tester request. **The default value is 55 milli-seconds.**
P3_MAX	0x0B	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) between end of ECU response and start of new tester request. **The default value is 5000 milli-seconds.**
P4_MIN	0x0C	0x0-0xFFFF	For protocol ID of ISO9141, this sets the minimum inter-byte time (in milli-seconds) for a tester request. **The default value is 5 milli-seconds.**
P4_MAX	0x0D	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum inter-byte time (in milli-seconds) for a tester request. **The default value is 20 milli-seconds.**
W1	0x0E	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) from the end of the address byte to the start of the synchronization pattern. **The default value is 300 milli-seconds.**
W2	0x0F	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) from the end of the synchronization pattern to the start of key byte 1. **The default value is 20 milli-seconds.**
W3	0x10	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) between key byte 1 and key byte 2. **The default value is 20 milli-seconds.**
W4	0x11	0x0-0xFFFF	For protocol ID of ISO9141, this sets the maximum time (in milli-seconds) between key byte 2 and its inversion from the tester. **The default value is 50 milli-seconds.**
W5	0x12	0x0-0xFFFF	For protocol ID of ISO9141, this sets the minimum time (in milli-seconds) before the tester start to transmit the address byte. **The default value is 300 milli-seconds.**
TIDLE	0x13	0x0-0xFFFF	For protocol ID of ISO9141, this sets the amount of bus idle time that is needed before a fast initialization sequence will begin. **The default is the value of W5.**
TINIL	0x14	0x0-0xFFFF	For protocol ID of ISO9141, this sets the duration (in milli-seconds) for the low pulse in fast initialization. **The default value is 25 milli-seconds.**
TWUP	0x15	0x0-0xFFFF	For protocol ID of ISO9141, this sets the duration (in milli-seconds) of the wake-up pulse in fast initialization. **The default value is 50 milli-seconds.**
PARITY	0x16	0 (NO_PARITY) 1 (ODD_PARITY) 2 (EVEN_PARITY)	For a protocol ID of ISO9141 only. **The default value is NO_PARITY.**
BIT_SAMPLE_POINT	0x17	0-100	For a protocol ID of CAN, this sets the desired bit sample point as a percentage of the bit time. **The default is 80%.**
SYNC_JUMP_WIDTH	0x18	0-100	For a protocol ID of CAN, this sets the desired synchronization jump width as a percentage of the bit time. **The default is 15%.**

FIGURE 23B—IOCTL GET_CONFIG / SET_CONFIG PARAMETER DETAILS (CONTINUED)

Valid values for Parameter	ID Value	Valid values for Value	Description
Unused	0x19		Reserved for SAE
T1_MAX	0x1A	0x0-0xFFFF	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. **The default value is 20 milli-seconds.**
T2_MAX	0x1B	0x0-0xFFFF	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. **The default value is 100 milli-seconds.**
T4_MAX	0x1C	0x0-0xFFFF	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. **The default value is 20 milli-seconds.**
T5_MAX	0x1D	0x0-0xFFFF	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. **The default value is 100 milli-seconds.**
ISO15765_BS	0x1E	0x0-0xFF	For protocol ID of ISO15765, this sets the block size for segmented transfers. **The default value is 0. Default value or value set by the application may be overridden by interface to match the capabilities of the interface.**
ISO15765_STMIN	0x1F	0x0-0xFF	For protocol ID of ISO15765, this sets the separation time for segmented transfers. **The default value is 0. Default value or value set by the application may be overridden by interface to match the capabilities of the interface.**
Unused	0x20 - 0xFFFF		Reserved for SAE
Tool manufacturer specific	0x10000 – 0xFFFFFFFF	Manufacturer Specific	Manufacturer Specific

FIGURE 23C—IOCTL GET_CONFIG / SET_CONFIG PARAMETER DETAILS (CONTINUED)

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 24. 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
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 24—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 25. 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
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 25—READ_PROG_VOLTAGE DETAILS

7.3.5 FIVE_BAUD_INIT—The IoctlID value of FIVE_BAUD_INIT is used to initiate a 5-baud initialization sequence from the pass-thru device. The calling application is responsible for allocating and initializing the associated parameters described in Figure 26. 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 ISO 9141.

Parameter	Description
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. May be less than "2". 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 26—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 27. 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 ISO 9141.

Parameter	Description
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.
OutputPtr	Points to the structure PASSTHRU_MSG (see the message definition section of this document) which the pass-thru device will receive.

FIGURE 27—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 28. When the function is successfully completed, the transmit queue will have been cleared.

Parameter	Description
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 28—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 29. When the function is successfully completed, the receive queue will have been cleared.

Parameter	Description
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 29—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 30. When the function is successfully completed, the list will have been cleared and all periodic messages will have stopped transmitting.

Parameter	Description
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 30—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. The calling application is responsible for allocating and initializing the associated parameters described in Figure 31. When the function is successfully completed, the list will have been cleared and all message filtering will have stopped.

Parameter	Description
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 31—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 32. When the function is successfully completed, the table will have been cleared. It should be noted that this only applies Protocol ID of SAE J1850PWM.

Parameter	Description
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 32—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 33. When the function is successfully completed, the look-up table will have been altered. It should be noted that this only applies Protocol ID of J1850PWM.

Parameter	Description
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 33—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 34. When the function is successfully completed, the look-up table will have been altered. It should be noted that this only applies Protocol ID of J1850PWM.

Parameter	Description
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 34—DELETE_FROM_FUNCT_MSG_LOOKUP_TABLE DETAILS

8. *Message Structure*—The following message structure will be used for all messages. The total message size (in bytes) is the DataSize. The ExtraDataIndex points to the IFR or checksum/CRC byte(s) when applicable. 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)
DataSize	Data size in bytes
ExtraDataIndex	Start position of extra data in received message (e.g., IFR, CRC, checksum, ...). The extra data bytes follow the body bytes in the Data array. The index is zero-based.
Data	Array of data bytes.

8.3 Message Data Formats—The following sections describe the bytes in the Data section of the PASSTHRU_MSG structure. In cases where extra data is included, the ExtraDataIndex will give the byte index from the beginning of the PASSTHRU_MSG structure Data section to the first byte of extra data.

NOTE—Extra bytes are not supported for PASSTHRU_MSG structures used for transmitting messages.

8.3.1 CAN DATA FORMAT—The CAN protocol is used for raw CAN message interfacing to the vehicle. This protocol can be used to handle any custom CAN messaging protocol, including custom flow control mechanisms. The order of the bytes is shown in Figure 35.

Offset	Data
0	CAN ID (bits 24-29)
1	CAN ID (bits 16-23)
2	CAN ID (bits 8-15)
3	CAN ID (bits 0-7)
4	First data byte of message
...	...
DataSize - 1	Last data byte of message

FIGURE 35—CAN DATA FORMAT

NOTE—Extra bytes are not supported for PASSTHRU_MSG structures used for transmitted messages.

8.3.1.1 CAN Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for CAN data:

a. DataSize less than four (4) bytes or greater than twelve (12) bytes (4 ID bytes + 8 data bytes).

8.3.2 ISO 15765-4 DATA FORMAT—The ISO 15765-4 protocol implements the network layer (i.e., adding the PCI byte to the transmitted messages, performing flow control, and removing the PCI byte from received messages) in the device so the application just sends and receives the actual message data. The order of the bytes is shown in Figure 36.

Offset	Data
0	CAN ID (bits 24-29)
1	CAN ID (bits 16-23)
2	CAN ID (bits 8-15)
3	CAN ID (bits 0-7)
4	First data byte of message (or ISO15765-2 extended address byte when ISO15765_ADDR_TYPE is set)
...	...
DataSize - 1	Last data byte of message

FIGURE 36—ISO 15765-4 DATA FORMAT

NOTE—Extra bytes are not supported for PASSTHRU_MSG structures used for transmitted messages.

8.3.2.1 ISO 15765-4 Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for ISO 15765-4 data:

a. DataSize less than four (4) bytes (ID only) or greater than 4101 bytes (4 ID bytes + 1 possible extended address byte + 4096 data bytes).

8.3.3 SAE J1850PWM DATA FORMAT—The order of bytes for J1850PWM is shown in Figure 37.

Offset	Data
0	First byte of message
...	...
N	Last byte of message
ExtraDataIndex	IFR byte or CRC
...	...
DataSize – 1	CRC

FIGURE 37—SAE J1850PWM DATA FORMAT

NOTE—Extra bytes are not supported for PASSTHRU_MSG structures used for transmitted messages.

8.3.3.1 SAE J1850PWM Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for J1850PWM data:

a. DataSize less than three (3) bytes (3 header bytes) or greater than 10 bytes (3 header bytes + 7 data bytes).
b. Source address that is different than the node ID.

8.3.4 SAE J1850VPW DATA FORMAT—The order of bytes for SAE J1850VPW is shown in Figure 38.

Offset	Data
0	First byte of message
...	...
N	Last byte of message
ExtraDataIndex	IFR byte or CRC
...	...
DataSize - 1	CRC

FIGURE 38—SAE J1850VPW DATA FORMAT

NOTE—Extra bytes are not supported for PASSTHRU_MSG structures used for transmitted messages.

8.3.4.1 SAE J1850VPW Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for SAE J1850VPW data:

a. DataSize of zero or greater than 4128 bytes.

8.3.5 ISO 9141 DATA FORMAT—The order of bytes for ISO 9141 is shown in Figure 39.

Offset	Data
0	First byte of message
...	...
n	Last byte of message
ExtraDataIndex / DataSize - 1	Checksum

FIGURE 39—ISO 9141 DATA FORMAT

8.3.5.1 ISO 9141 Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for ISO 9141 data:

a. DataSize of zero or greater than 261 bytes.

8.3.6 ISO 14230-4 DATA FORMAT—The order of bytes for ISO 14230-4 is shown in Figure 40.

Offset	Data
0	First byte of message
...	...
n	Last byte of message
ExtraDataIndex / DataSize - 1	Checksum

FIGURE 40—ISO 14230-4 DATA FORMAT

8.3.6.1 ISO 14230-4 Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for ISO 14230-4 data:

a. DataSize of less than four (4 byte header) or greater than 261 bytes (4 byte header + 256 data bytes + 1 byte checksum).

8.3.7 SCI DATA FORMAT—The order of bytes for SCI is shown in Figure 41.

Offset	Data
0	First byte of message
...	...
N	Last byte of message

FIGURE 41—SCI DATA FORMAT

8.3.7.1 SCI Data Format Error Detection—The following data format errors should be detected when using the ERR_INVALID_MSG for SCI data:

a. DataSize of zero or greater than 256 bytes.

8.4 Message Flag and Status Definitions

8.4.1 RXSTATUS—Definitions for RxStatus bits are shown in Figure 42.

8.4.2 TXFLAGS—Definitions for TxFlags bits are shown in Figure 43.

Definition	RxStatus Bit(s)	Description	Value
	31-24	Unused	Tool manufacturer specific
	23-9	Unused	Reserved for SAE – shall be set to 0
CAN_29BIT_ID	8	CAN ID Type	0 = 11-bit, 1 = 29-bit
	7-3	Unused	Reserved for SAE – shall be set to 0
RX_BREAK	2	Break indication received	0 = no indication, 1 = break received
ISO15765_FIRST_FRAME	1	ISO15765-2 First Frame Indication	0 = no indication, 1 = First Frame Note: no data is reported with this message
TX_MSG_TYPE	0	Receive Indication/ Transmit Confirmation	0 = received, 1 = transmitted

FIGURE 42—RXSTATUS BIT DEFINITIONS

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
	22-17	Unused	Reserved for SAE - shall be set to 0
BLOCKING	16	Blocking flag	0 = non-blocking, 1 = blocking
	15-9	Unused	Reserved for SAE - shall be set to 0
CAN_29BIT_ID	8	CAN ID type	0 = 11-bit, 1 = 29-bit
ISO15765_ADDR_TYPE	7	ISO15765-2 Addressing Method	0 = no extended address, 1 = extended address is first byte after the ID bytes Note: if different, this will override Flags in the PassThruConnect for this message
ISO15765_FRAME_PAD	6	ISO15765-2 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 43—TXFLAGS BIT DEFINITIONS

9. *DLL Installation and Registration*

9.1 Naming of Files—In general, 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 Registration—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 44.

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.

The Devices key will contain a list of keys, one for each device supported by the vendor.

Ex: ACME Serial Device
ACME Ethernet Device
ACME Parallel Device etc.

Each Vendor Device Key will have the entries shown in Figure 45 associated with them:

Example for Key: ACME Ethernet Device

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 46 is a flow chart that shows a typical usage.

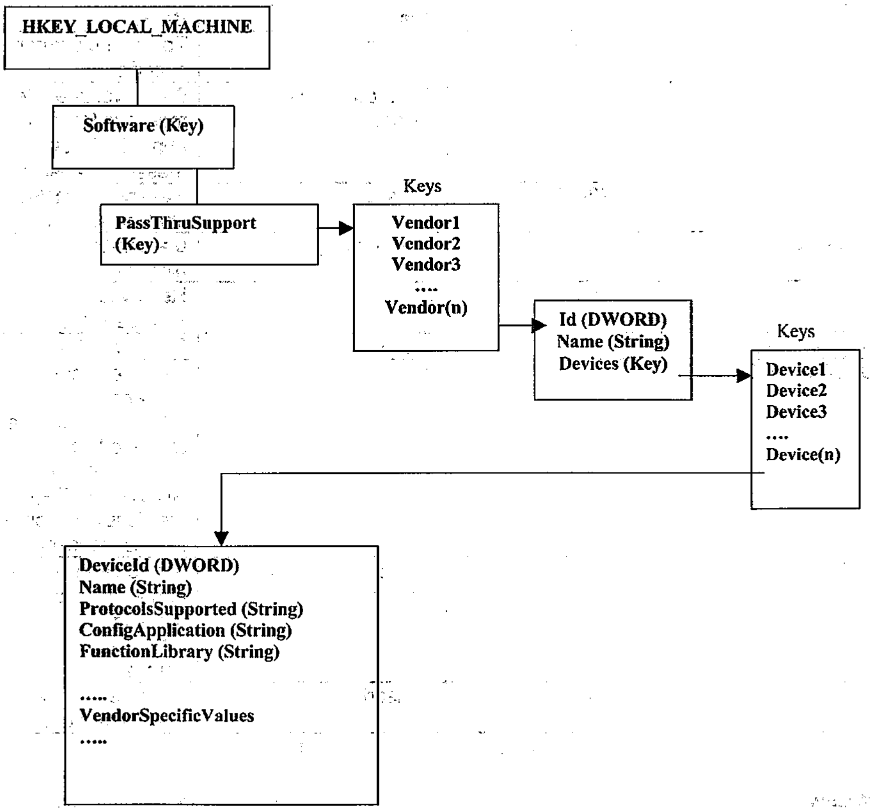

FIGURE 44—WIN32 REGISTRATION

DeviceId	DWORD	A unique ID for the device supplied by the vendor
Name	String	The name of the device. Ex: " ACME CAN Device over Ethernet"
ProtocolsSupported	String	The various protocols supported by the device are listed here separated by commas. The representations of protocols here will be same as the Protocol Id symbolic constants used in PassThruConnect function for the purpose of consistency. 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, ISO15765, J1850VPW, J1850PWM, ISO9141, ISO14230" A protocol appearing multiple times will indicate that more than one channel supporting the protocol exists on the hardware.
ConfigApplication	String	The complete path of the configuration application for this device. For every device listed in the section the 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 45—WIN32 REGISTRY VALUES

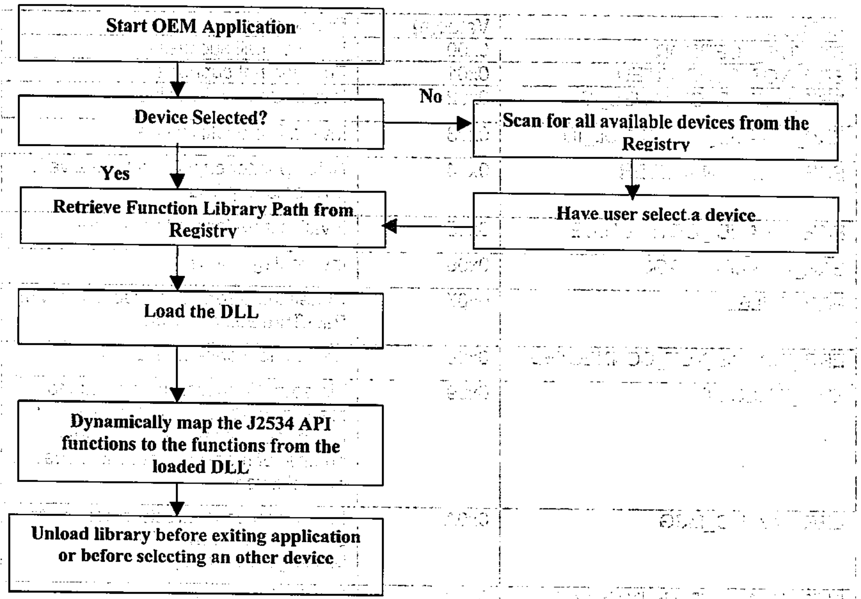

FIGURE 46—APPLICATION INTERACTION WITH REGISTRY

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 should be built with byte alignment (i.e., packing) set to one (1) byte.

9.2.2.1 Export Library Definition File

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

10. Return Value Error Codes—Figure 47 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	Function not supported
ERR_INVALID_CHANNEL_ID	0x02	Invalid ChannelID value
ERR_INVALID_PROTOCOL_ID	0x03	Invalid ProtocolID value
ERR_NULLPARAMETER	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. Get description with PassThruGetLastError.
ERR_DEVICE_NOT_CONNECTED	0x08	Device not connected to PC
ERR_TIMEOUT	0x09	Timeout. No message available to read or could not read the specified number of messages. The actual number of messages read 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_INVALID_ERROR_ID	0x0E	Invalid ErrorID value
ERR_INVALID_IOCTL_ID	0x0F	Invalid IoctlID value
ERR_BUFFER_EMPTY	0x10	Protocol message buffer empty
ERR_BUFFER_FULL	0x11	Protocol message buffer full
ERR_BUFFER_OVERFLOW	0x12	Protocol message buffer overflow
ERR_PIN_INVALID	0x13	Invalid pin number
ERR_CHANNEL_IN_USE	0x14	Channel already in use
ERR_MSG_PROTOCOL_ID	0x15	Protocol type does not match the protocol associated with Channel ID
Unused	0x16-0xFFFFFFFF	Reserved for SAE use

FIGURE 47—ERROR VALUES

APPENDIX A
GENERAL ISO 15765-2 FLOW CONTROL EXAMPLE

A.1 Normal Addressing Used—This section includes examples of multi-frame request and response messages using flow control as defined in ISO 15765-2. These examples assume that normal addressing is used for the request and the response messages (no extended address present), and that the CAN identifier assignments shown in Figure A1 apply.

CAN Id	CAN Id type	Usage
241 hex	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 hex	Response CAN ID	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) For the reception of a request message from the pass-thru interface this CAN ID is used by the ECU for: • FlowControl frame

FIGURE A1—CAN IDENTIFIER ASSIGNMENT EXAMPLE

A.2 General Request Message Flow Example—The general request message CAN frame flow example in Figure A2 shows the usage of the PassThru functions in the pass-thru interface to transmit a multi-frame request message to the ECU and how the CAN frames are transmitted onto the CAN bus by the interface and the ECU.

a. The application requests the transmission of a request message via the PassThruWriteMsgs API function. The pass-thru interface transmits the FirstFrame to the ECU using the physical request CAN Identifier.
b. The ECU confirms the reception of the FirstFrame and transmits its FlowControl frame (using the response CAN Identifier) with FlowStatus set to CTS (ClearToSend), BS equal to 3 and STmin set to the minimum time the pass-thru interface shall wait between the transmission of the ConsecutiveFrames.
c. After the reception of the FlowControl frame from the ECU the pass-thru interface starts to transmit the first block of ConsecutiveFrames of the request message, using the physical request CAN Identifier. After the transmission of 3 ConsecutiveFrames the interface stops transmitting, because it awaits that the ECU sends a FlowControl frame.
d. The ECU confirms the reception of the 3 ConsecutiveFrames and transmits its FlowControl frame (using the response CAN Identifier) with FlowStatus set to WAIT. This indicates to the pass-thru interface that the ECU is in progress of processing the ConsecutiveFrames and that a further FlowControl will be transmitted (which either indicates that the ECU needs further time to process the received data or that the interface can continue to send ConsecutiveFrames).
e. The ECU transmits its FlowControl frame with FlowStatus set to CTS (ClearToSend), BS equal to 3 and STmin set to the minimum time the pass-thru interface shall wait between the transmission of the further ConsecutiveFrames.
f. After the reception of the FlowControl frame from the ECU the pass-thru interface starts to transmit the remaining 2 ConsecutiveFrames of the request message, using the physical request CAN Identifier. After the transmission of the 2 ConsecutiveFrames the request message is completely transmitted to the ECU and the ECU can process the request. The completion of the transmission is confirmed to the application via the TX_MSG_TYPE bit in RxStatus retrieved through the PassThruReadMsgs API function.

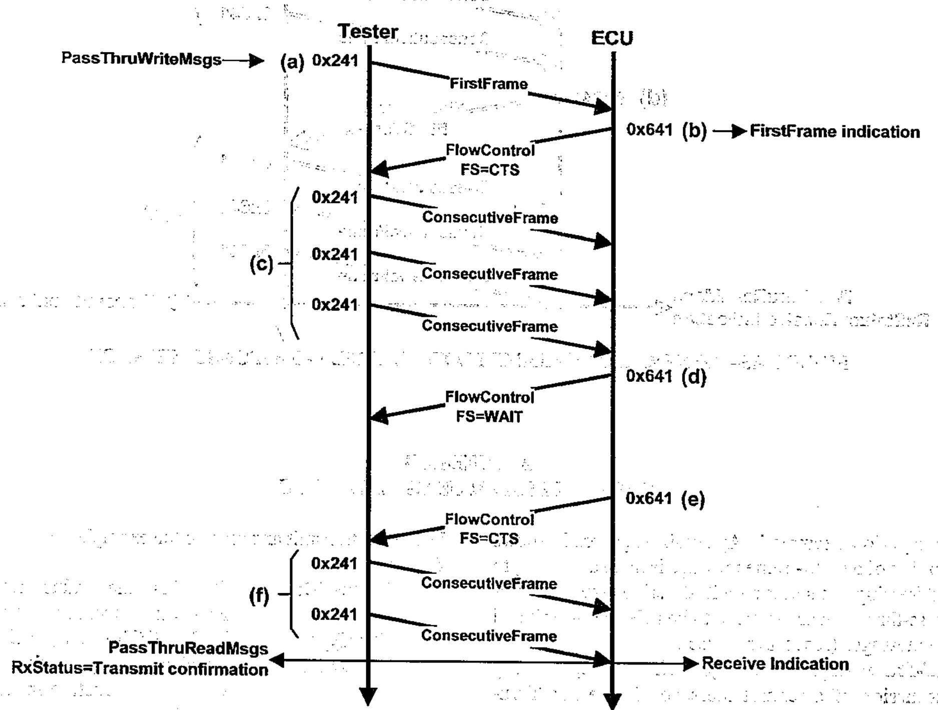

FIGURE A2—GENERAL CAN FRAME FLOW EXAMPLE - REQUEST MESSAGE

A.3 General Response Message Flow Example—The response message CAN frame flow example in Figure A3 shows the usage of the PassThru functions in the pass-thru interface during the reception of a multi-frame response message from the ECU and how the CAN frames are transmitted onto the CAN bus by the interface and the ECU.

a. The ECU application requests the transmission of a response message. The ECU transmits the FirstFrame to the pass-thru interface using the response CAN Identifier.
b. The pass-thru interface confirms the reception of the FirstFrame and transmits its FlowControl frame (using the physical request CAN Identifier) with FlowStatus set to CTS (ClearToSend), BS equal to 5 and STmin set to the minimum time the ECU shall wait between the transmission of the ConsecutiveFrames. The reception of the FirstFrame is indicated to the application via the ISO15765_FIRST_FRAME bit in RxStatus retrieved through the PassThruReadMsgs API function.
c. After the reception of the FlowControl frame from the pass-thru interface the ECU starts to transmit the first block of ConsecutiveFrames of the response message, using the response CAN Identifier. After the transmission of 5 ConsecutiveFrames the ECU stops transmitting, because it awaits that the interface sends a FlowControl frame.
d. The pass-thru interface confirms the reception of the 5 ConsecutiveFrames and transmits its FlowControl frame (using the physical request CAN Identifier) with FlowStatus set to CTS, BS equal to 5 and STmin set to the minimum time the ECU shall wait between the transmission of the further ConsecutiveFrames.
e. After the reception of the FlowControl frame from the pass-thru interface the ECU starts to transmit the remaining 3 ConsecutiveFrames of the response message, using the response CAN Identifier. After the transmission of the 3 ConsecutiveFrames the response message is completely transmitted to the interface. The completion of the reception is indicated to the application via the TX_MSG_TYPE bit in RxStatus retrieved through the PassThruReadMsgs API function (plus the received data).

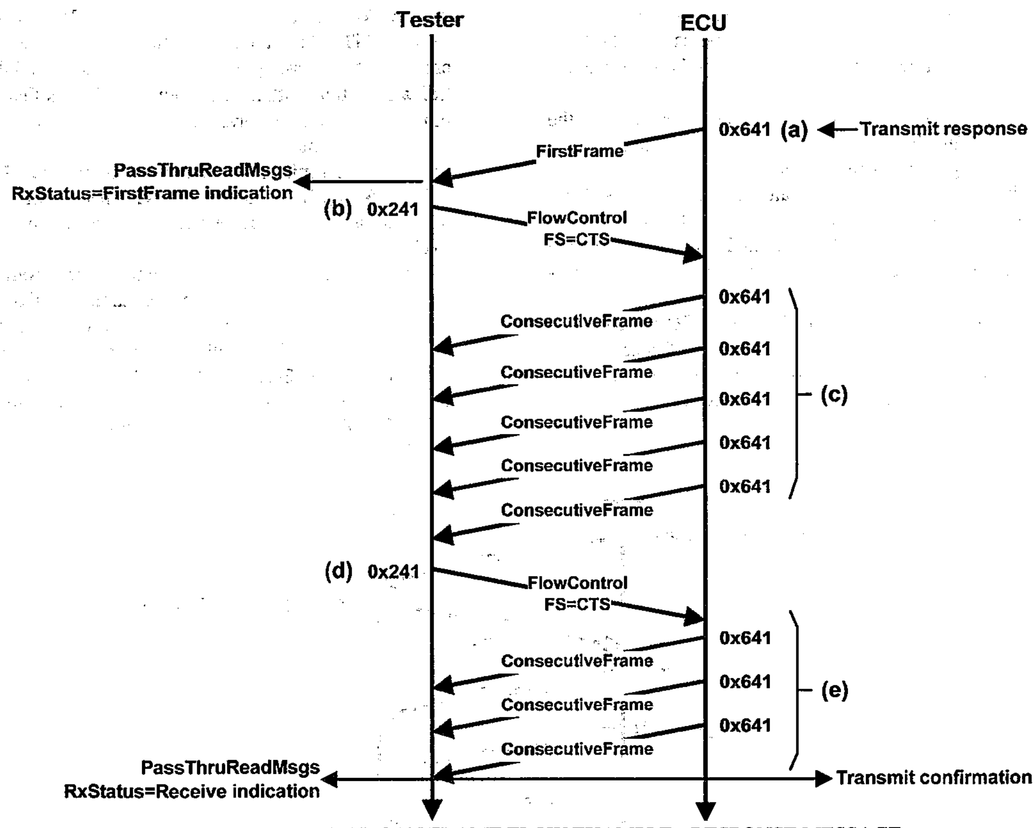

FIGURE A3—GENERAL CAN FRAME FLOW EXAMPLE - RESPONSE MESSAGE

APPENDIX B
MESSAGE FILTER USAGE EXAMPLE

B.1 Filter Usage—The message flow example in Appendix A generally shows how the transmission and reception of a multi-frame message is done according to ISO 15765-2, using normal addressing. This section will describe how the filters have to be configured in the pass-thru interface in order to be able to transmit and receive the shown multi-frame messages (request/response).

B.2 Transmission of a Multi-Frame Request Message—The programming application requests the transmission of a request message via the PassThruWriteMsgs API function. If the transmitted message is more than will fit into a single CAN frame then the pass-thru interface transmits the FirstFrame of the multi-frame message. The FirstFrame uses the CAN ID (241 hex plus optional extended address) as specified in the message passed via the PassThruWriteMsgs API function. The FlowControl sent by the ECU is received, masked, and matched (CAN Identifier 641 hex plus optional extended address) with the flow control filter that was setup with the PassThruStartMsgFilter API function. If there is a match, the message is then transmitted according to the BS and STmin values in the FlowControl message.

B.3 Reception of a Multi-Frame Response Message—The ECU starts to transmit its response message by sending the FirstFrame. The FirstFrame sent by the ECU is received, masked, and matched (CAN Identifier 641 hex plus optional extended address) with the flow control filter that was setup with the PassThruStartMsgFilter API function. If there is a match, a FirstFrame indication is given by a zero length message with the ISO15765_FIRST_FRAME bit set in the RxStatus. Next, FlowControl frame is sent to the ECU using either the default BS and STmin parameters, or the modified values set using the PassThruIoctl API function. If the interface is not capable of supporting those values, the interface may override them.

B.4 Filter Configuration—This section defines how the filter in the API shall be specified in order to be able to receive and transmit the multi-frame messages as given in the previous sections. It is assumed that the pass-thru interface is connected properly to the application (PassThruConnect already performed) and the ChannelID required to be passed to the PassThruStartMsgFilter API function is valid. The parameters passed to the PassThruStartMsgFilter function in order to be able to transmit and receive the example multi-frame messages are specified as follows:

ChannelID: Contains the value retrieved previously via the PassThruConnect function for the ISO15765 protocol.

FilterType: FLOW_CONTROL_FILTER

pMaskMsg: Receive message mask, points to a PASSTHRU_MSG, where the structure members are set as follows (note that all bits are relevant to be filtered on for the given example):

ProtocolID:	ISO15765
RxStatus:	00 hex (don't care for filter)
TxFlags:	SCI_TX_VOLTAGE = 0 BLOCKING = 0 CAN_29BIT_ID = 0 (11 bit CAN ID used) ISO15765_ADDR_TYPE =0 (normal addressing used) ISO15765_FRAME_PAD =0 (don't care for reception) resulting TxFlags value: 00000000 hex
TimeStamp:	00000000 hex (don't care)
DataSize:	4 (CAN ID only)
ExtraDataIndex:	0
Data:	00 00 07 FF hex

pPatternMsg: Receive message, points to a PASSTHRU_MSG, where the structure members are set as follows:

ProtocolID:	ISO15765
RxStatus:	00 hex (don't care)
TxFlags:	SCI_TX_VOLTAGE = 0 BLOCKING = 0

CAN_29BIT_ID = 0 (11 bit CAN ID used)
ISO15765_ADDR_TYPE =0 (normal addressing used)
ISO15765_FRAME_PAD =0 (don't care for reception)
resulting TxFlags value: 00000000 hex

TimeStamp: 00000000 hex (don't care)
DataSize: 4 (CAN ID only)
ExtraDataIndex: 0
Data: 00 00 06 41 hex

pFlowControlMsg: Transmit message, points to a PASSTHRU_MSG, where the structure members are set as follows:

ProtocolID: ISO15765
RxStatus: 00 hex (don't care)
TxFlags: SCI_TX_VOLTAGE = 0
BLOCKING = 0
CAN_29BIT_ID = 0 (11 bit CAN ID used)
ISO15765_ADDR_TYPE =0 (normal addressing used)
ISO15765_FRAME_PAD =0 (no padding in case of Flow Control transmission. In case of FirstFrame and ConsecutiveFrame transmission the padding flag given in the message to be transmitted is used - provided in PassThruWriteMsgs)
resulting TxFlags value: 00000000 hex

TimeStamp: 00000000 hex (don't care)
DataSize: 4 (CAN ID only)
ExtraDataIndex: 0
Data: 00 00 02 41 hex

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

With the filter configured as shown in this section, the interface is able to transmit and receive the multi-frame messages as given in the examples. The following figures provide details regarding the handling in the pass-thru interface, taking into account that this filter is set-up in the pass-thru interface.

B.4.1 Request Message Transmission—See Figure B1.

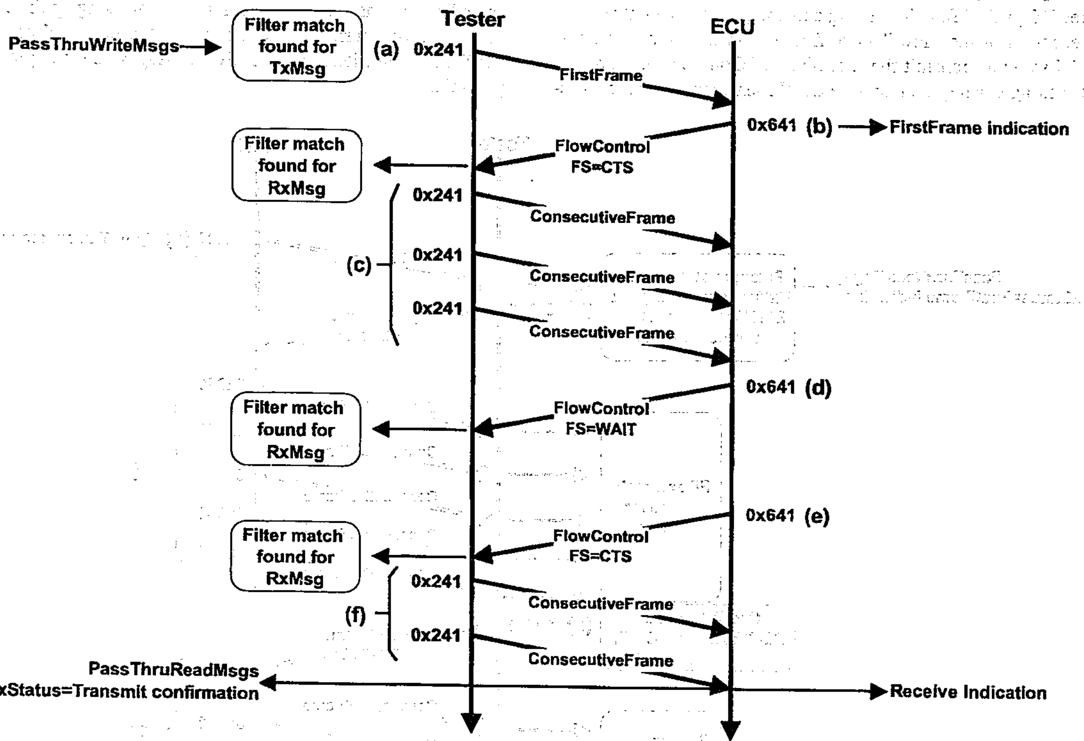

FIGURE B1—MESSAGE FLOW EXAMPLE WITH REFERENCES TO FILTER PARAMETERS - REQUEST MESSAGE

The application configures the flow control filter using the PassThruStartMsgFilter API function.

a. The application requests the transmission of a segmented (i.e., more than one CAN frame of data) message via the PassThruWriteMsgs API function. The interface transmits the FirstFrame to the ECU using the CAN Identifier as given in the message to be transmitted.
b. The ECU confirms the reception of the FirstFrame and transmits its FlowControl frame (using the response CAN Identifier) with FlowStatus set to CTS (ClearToSend), BS equal to 3 and STmin set to the minimum time the pass-thru interface shall wait between the transmission of the ConsecutiveFrames.
c. The pass-thru interface searches all configured flow control filters to see if a match with FlowControl message can be found. In case a match is found then the pass-thru interface starts transmitting ConsecutiveFrames according to the FlowControl parameters received, using the CAN Identifier as given in the message to be transmitted. After the transmission of 3 ConsecutiveFrames the pass-thru interface stops transmitting, because it awaits that the ECU sends a FlowControl frame.
d. The ECU confirms the reception of the 3 ConsecutiveFrames and transmits its FlowControl frame (using the response CAN Identifier) with FlowStatus set to WAIT. The pass-thru interface searches all configured filters for a match. In case a match is found then the pass-thru interface behaves as specified in the FlowControl frame (wait for further FlowControl).
e. The ECU transmits its FlowControl frame with FlowStatus set to CTS (ClearToSend), BS equal to 3 and STmin set to the minimum time the pass-thru interface shall wait between the transmission of the further ConsecutiveFrames.
f. The pass-thru interface searches all configured filters for a match. In case a match is found then the pass-thru interface behaves as specified in the FlowControl frame. The pass-thru interface starts to transmit the remaining 2 ConsecutiveFrames of the request message, using the CAN Identifier as given in the original message to be transmitted. After the

transmission of the 2 ConsecutiveFrames the request message is completely transmitted to the ECU and the ECU can process the request. The completion of the transmission is confirmed to the application via the TX_MSG_TYPE bit in RxStatus retrieved through the PassThruReadMsgs API function.

B.4.2 Response Message Reception—See Figure B2.

The application configures the flow control filter using the PassThruStartMsgFilter API function. The application configures the BS (5) and STmin (0) parameters for the interface using the PassThruIoctl API function, but the interface may override these values to match the capabilities of the interface.

a. The ECU application requests the transmission of a response message. The ECU transmits the FirstFrame to the pass-thru interface using the response CAN Identifier.
b. The pass-thru interface receives the FirstFrame and searches all configured filters for a match. In case a match is found then the pass-thru interface confirms the reception of the FirstFrame and transmits its FlowControl frame (using the CAN Identifier and the padding information as specified in the flow control filter message). The FlowStatus will be CTS (ClearToSend), BS (IOCTL parameter) will be equal to 5 and STmin (IOCTL parameter) will be set to the minimum time the ECU shall wait between the transmission of the ConsecutiveFrames. Furthermore the reception of the FirstFrame is indicated to the application via the ISO15765_FIRST_FRAME bit in RxStatus retrieved through the PassThruReadMsgs API function (using a message of zero length).
c. After the reception of the FlowControl frame from the pass-thru interface the ECU starts to transmit the first block of ConsecutiveFrames of the request message, using the response CAN Identifier. After the transmission of 5 ConsecutiveFrames the ECU stops transmitting, because it awaits that the pass-thru interface sends a FlowControl frame. For any received ConsecutiveFrame the pass-thru interface will search through the list of configured filters to find a match. In case a match is found then the data of the ConsecutiveFrame will be stored internally for the later message receive indication.
d. The pass-thru interface confirms the reception of the block of 5 ConsecutiveFrames and transmits its FlowControl frame using the message configured in the filter. The FlowStatus will be set to CTS, BS will be equal to 5 and STmin will be set to the minimum time the ECU shall wait between the transmission of the further ConsecutiveFrames.
e. After the reception of the FlowControl frame from the pass-thru interface the ECU starts to transmit the remaining 3 ConsecutiveFrames of the response message, using the response CAN Identifier. For any received ConsecutiveFrame the pass-thru interface will search through the list of configured filters to find a match. In case a match is found then the data of the ConsecutiveFrame will be stored internally for the later receive indication. After the transmission of the 3 ConsecutiveFrames the response message is completely transmitted to the pass-thru interface. The completion of the reception is indicated to the application via the TX_MSG_TYPE bit in RxStatus retrieved through the PassThruReadMsgs API function (plus the collected message data).

B.5 ISO 15765-2 Extended Addressing Notes—For extended addressing the same handling as described for normal addressing applies, except that the filter in the pass-thru interface is set-up to use the extended address in addition to the CAN ID when filtering on receive messages and verifying that a transmission is possible.

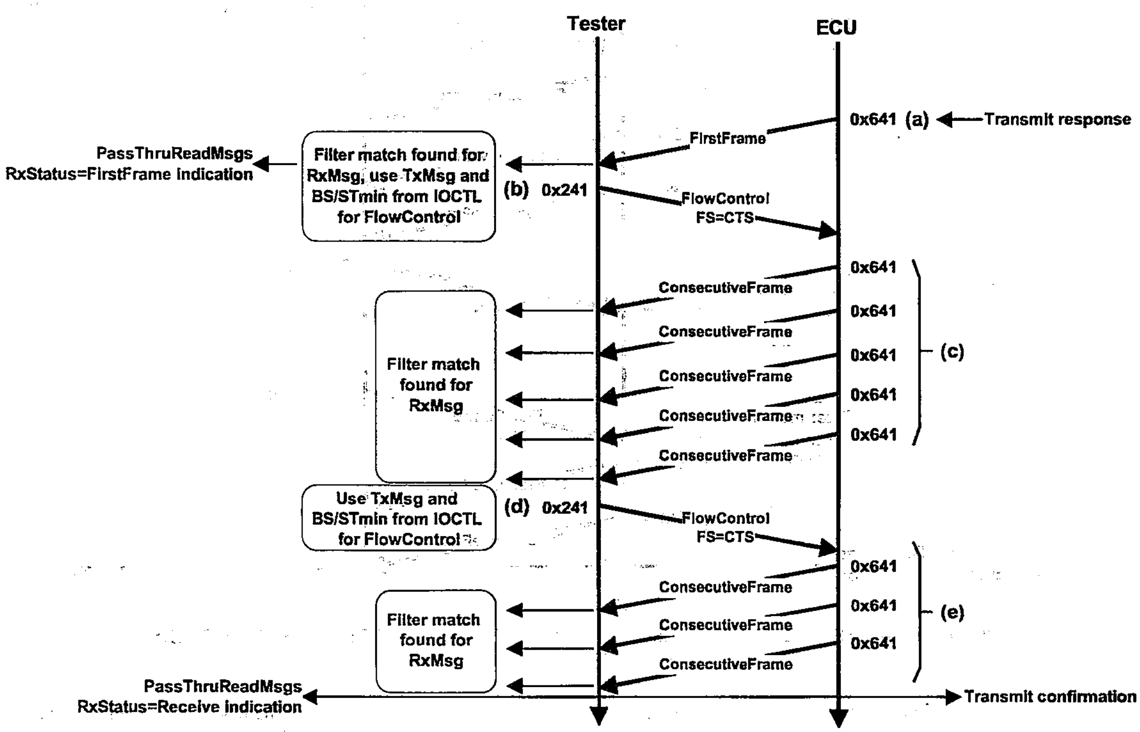

FIGURE B2—MESSAGE FLOW EXAMPLE WITH REFERENCES TO FILTER PARAMETERS - RESPONSE MESSAGE