TMC Recommended Practice

RP 1210A

VMRS 053

WINDOWS™ COMMUNICATION API

TABLE OF CONTENTS

*** These sections appeared in the original version of TMC RP 1210, but are omitted from RP 1210A.**

Issued 4/97
Revised 7/99

PREFACE

The following Recommended Practice is subject to the Disclaimer at the front of TMC's *Recommended Maintenance Practices Manual.* Users are urged to read the Disclaimer before considering adoption of any portion of this Recommended Practice.

1. INTRODUCTION

This document describes a standardized interface—TMC's RP1210 Windows™ Communication Application Program Interface (API)—for personal computer (PC) to on-vehicle datalink communications under the Microsoft Windows™ family of operating systems (Windows™ 3.1x, Windows™ 95, and Windows™ NT).

The Maintenance Council (TMC) established this Recommended Practice for vehicle ECU communication and control under the Microsoft Windows™ family of operating systems. Anyone is welcome to employ this RP in implementing software systems for ECU reprogramming and communication. All J1708 applications written for the previous version of RP 1210 will be supported by drivers following this RP. If any new features of this version of the RP are used, backward compatibility will not be guaranteed. Future versions of RP1210 will not be backwards compatible.

NOTE: Appendices I and II, which appeared in the initial version of RP 1210, are obsolete and have been deleted from RP 1210A. However, the first Appendix in RP 1210A has been intentionally entitled Appendix III to retain section-by-section compatibility with the initial version of RP 1210.

1.1 SCOPE AND OBJECTIVE

This Recommended Practice defines a communication API between the on-vehicle data link and PC application software programs running under the Windows™ family of operating systems. This Recommended Practice (RP) establishes a standard interface between the physical data link (J1708/J1587/J1922, CAN/J1939, or J1850) and Windows™ software applications for the personal computer. This RP is consistent with TMC RP1208, *PC Service Tool: Hardware Selection Guidelines*.

Current and future hardware requirements have been considered in developing this software API. This promotes the development of the software applications for fleet maintenance, ECU reprogramming, and vehicle diagnostics using a standard common software interface.

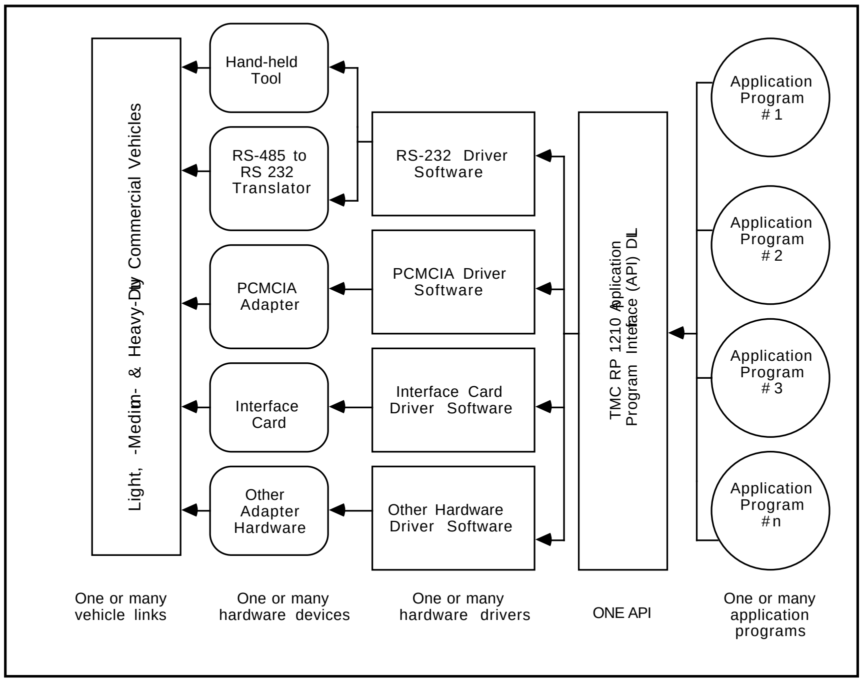
1.2 PERCEIVED NEEDS

TMC recognizes the following as perceived needs for the communication API:

1. Allow hardware adapters to be transparent to the application programmer. No portion of the interface should be tied to any specific method or hardware configuration.
2. Allow software product differentiation on a proprietary basis while still maintaining consistency between hardware vendors.
3. Support SAE J1708, SAE CAN/J1939, and SAE J1850 network protocols.

1.3 POTENTIAL COMMUNICATION SOFTWARE/ HARDWARE INTERFACES

As can be seen in **Fig. 1**, a wide variety of hardware devices can be used to connect the personal computer (PC) to the vehicle datalink (in theory, these interfaces could communicate simultaneously



Light-, Medium- & Heavy-Duty Commercial Vehicles
Hand-held Tool
RS-485 to RS 232 Translator
PCMCIA Adapter
Interface Card
Other Adapter Hardware
RS-232 Driver Software
PCMCIA Driver Software
Interface Card Driver Software
Other Hardware Driver Software
TMC RP 1210 Application Program Interface (API) DLL
Application Program # 1
Application Program # 2
Application Program # 3
Application Program # n
One or many vehicle links
One or many hardware devices
One or many hardware drivers
ONE API
One or many application programs

Fig. 1: Architectural Overview: Potential Communications Software/Hardware Interfaces

through the API). Some examples are:

- A hand-held tool connected to the RS-232 I/O port on the PC and the vehicle data link.
- A RS-485 to RS-232 translator connected to the RS-232 I/O port on the personal computer and the vehicle data link.
- A PCMCIA adapter connected to the personal computer and the vehicle data link.
- A PC interface card connected to the personal computer and the vehicle data link.
- A generic hardware device connected to the personal computer and the vehicle data link.

2. FUNCTIONAL SPECIFICATION

TMC has identified a set of functions which an API must support in order to conform to this Recommended Practice.

2.1 REQUIRED FUNCTIONALITY

***Multiple Client*—**The API will support a minimum of 16 clients and a maximum of 128 clients.

Send/Receive Message Buffering—Since Microsoft Windows™ is event-driven, an API satisfying this RP must support this feature. There must be implicit support for synchronous as well as asynchronous communication.

Initialization/Reset—The application software must be able to initialize and reset the hardware through API-supplied functions at any time. If the hardware does not support mid-stream initialization or resetting, the API-supplied function shall provide a return value to this effect.

Time--Stamping of Messages—Since messages received from the vehicle data link are buffered by the API, there must be a time-stamp associated with each message to resolve ambiguity and establish an order of precedence (for the application software) between successive messages. This function would ordinarily be implemented in the driver software interfacing with the API. The timestamp is four bytes in length and is in big endian or Motorola format. See the *RP1210_ReadMessage* function in **Appendix III** for the definition and layout of the timestamp. The resolution of the timestamp is implementation-dependent and is given in the INI file (see **Appendix V**).

Version Reporting—The API shall be able to report its software version information.

Message Filtering—The API must support message filtering, typically performed by the lower-level interfaces, as specified in the definition of the *RP1210_SendCommand* function in **Appendix III**. An implementation of the API shall provide each client with exactly those messages as are specified in its filter specification. Command numbers 4,5, and 7 have been reserved for setting these filters. Command numbers 3, 17, and 18 have been reserved for additional filtering capabilities. Since, with J1850, the parameter values desired must be explicitly solicited by the application, there is no need to specify a filter for the J1850 protocol. An application desiring the equivalent of message filtering will simply not request the parameters sought to be filtered.

3. HIGH-LEVEL DESIGN

This section provides a high-level design description of TMC's RP1210 Windows™ communication API, which will be implemented as a Windows™ Dynamic Link Library or DLL (hereafter referred to as the API DLL). The main purpose of this DLL, which will have several vendor-specific implementations, is to provide a generalized interface between any hardware-specific drivers and applications running under the Microsoft Windows™ family of operating systems.

Each vendor implementation must have built-in interface(s) with the underlying physical layers (hardware) supported by the particular vendor. Information about the vendor DLLs present as well as the protocols and devices supported by each implementation can be retrieved through INI files (see **Appen-**

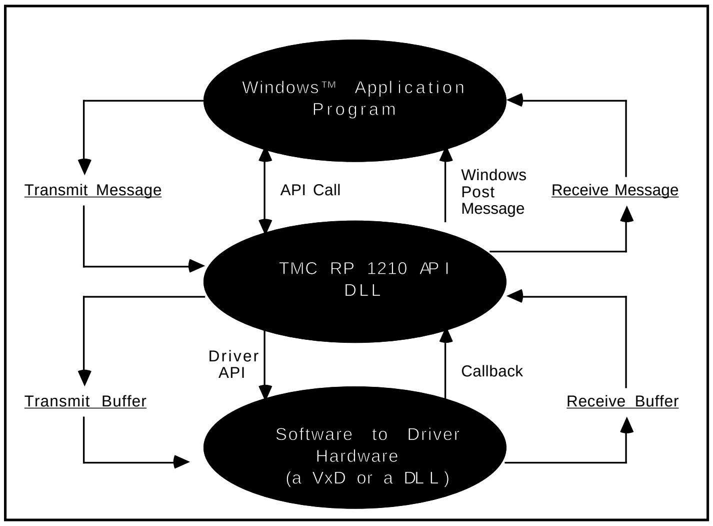


Fig. 2: Major Component Data Flow Diagram for Windows™

dix V). A specific DLL will, in general, be explicitly loaded by the client application through the use of the Windows™ LoadLibrary function, and specific functions therein called through the use of the Windows™ *GetProcAddress* function.

3.1 MAJOR COMPONENTS

See **Fig. 2**, "Major Component Data Flow Diagram for Windows™."

3.2 API INTERFACE

TMC's RP1210 API provides a set of functions and a client application messaging system that permits client applications to perform send and receive operations through the DLL, to and from the lower-level interfaces.

The client application performs a *RP1210_SendMessage* API function call to send a message to the RP1210 API DLL. Messages are placed in the API DLL transmit queue for processing. A message is rejected when the transmit queue becomes full or an error condition is detected. The API DLL notifies the client application through the return code of any failures (see **Appendix III**, Section 3.3).

Under Windows™ 3.1x, two separate mechanisms can be employed for retrieving messages from the datalink, depending upon the interface being used. The RP1210 API DLL notifies the client applications of a reception of a message from the device interfacing with the vehicle datalink by posting a message to the client application. Upon receiving the posted message, the client application retrieves the message packet by performing some variant of the *RP1210_ReadMessage* API call. The client must perform repeated *RP1210_ReadMessage* API calls to retrieve all received messages (see **Appendix III, Section 3.4**). The messaging scheme used to achieve this functionality is detailed in **Section 3.4**, "Notification Via Message Posting for Windows™."

In addition to notification through message posting, under Windows 95™ and Windows NT™, the client application repeatedly calls the *RP1210_ReadMessage* function (with blocking enabled). The *RP1210_ReadMessage* function call will not return until a message is available in the DLL's receive buffer or an error occurs. The messaging scheme used to achieve this functionality is detailed in **Section 3.5**, "Notification Scheme for Windows™ 95 and Windows NT™."

The API defines the structure for a vendor-specific INI file that relates the device IDs internally used by the vendor-specific DLL with the protocol strings associated with the device. **Appendix V** defines the layout of this INI file, while **Appendix III** lays out the definition of the required functionality of the API itself.

3.3 NAMING OF FILES

In general, each vendor will provide a different name implementation of the API and a number of these implementations could simultaneously reside on the same PC. No vendor shall name its implementation "RP1210.DLL". All 32-bit implementations shall have the string "32" suffixed to end of the name of the DLL. For example, if the name of the 16-bit implementation of this API by a particular vendor is VENDRX.DLL, then its 32-bit counterpart shall be named VENDRX32.DLL. This would allow the 16-bit and 32-bit implementations to reside on a given computer at the same time. The same policy shall apply to the naming of the INI files. Furthermore, although long file names are supported by Windows 95™ and Windows NT™, 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 a 16-bit version of an API DLL can be no greater than six characters.

3.4 NOTIFICATION VIA MESSAGE POSTING FOR WINDOWS™

Windows™ 3.1x is a message-based, event-driven architecture. Portions of application programs communicate with each other as well as with other applications through a sophisticated messaging system. In the proposed API, a DLL implementation under 16-bit Windows™ will notify the client application(s) of any data received from the vehicle link through the associated device interface and of the error status incurred during the sending of data packets to the vehicle through the device interface by posting an event notification message to its window (for which the API DLL saves a handle).

In the case of a received data packet, the client application will perform a *RP1210_ReadMessage* function call to read the data link packet received (which, in no relation to Windows™, is also commonly, and unfortunately, termed a "message").

A unique message identifier, shared by the application and the API DLL, shall be created by both the API

DLL and the application on their respective invocations through a Windows™ API call such as:

```
WM_RP1210_MESSAGE =
RegisterWindowMessage(RP1210_MESSAGE_STRING)
```

where RP1210_MESSAGE_STRING (or some other named variable) is read in as a character string from the vendor-provided INI file, and WM_RP1210_MESSAGE (or some other named variable) is declared as a WORD. The specific implementation of a DLL is free to choose the message character string for notification as long as this can be retrieved by the client application in conformance with the INI file specification (see **Appendix V, Section 5.3**).

It is inside the message processing loop of the client application that the *RP1210_ReadMessage* function is called whenever the application is notified of a data packet received from the vehicle link, through the application's device interface.

A client application may choose to bypass the messaging by polling for messages. It should be noted, however, that polling is not the norm in Windows™ for situations such as this.

The API DLL will also support other notifications to the application. This method is used to notify the application, if requested, of the status of a data packet transmission from application to data link using a call to *RP1210_SendMessage* (see **Appendix III, Section A3.3**), or of changes in interface device status and several protocol specific events. For this purpose, a unique message identifier, shared by the application and the API DLL, shall be created by both the API DLL and the application on their respective invocations through a Windows™ API call like the one below:

```
WM_RP1210_ERROR_MESSAGE =
RegisterWindowMessage(RP1210_ERROR_STRING)
```

where RP1210_ERROR_STRING (or some other named variable) is read in as a character string from the vendor-provided INI file, and WM_RP1210_ERROR_MESSAGE (or some other named variable) is declared as a WORD. The specific implementation of the DLL is free to choose the message character string for notification, as long as this can be retrieved by the client application in conformance with the INI file specification (see **Appendix V, Section 5.3**).

The following definitions will describe the various notifications sent to the Application through this Window Message:

RP1210_SendMessage **- Message Sent Notification**

wParam will have a value of ERR_TXMESSAGE_STATUS (145)

lParam will contain the Message Number as returned from the *RP1210_SendMessage* call and will be in the range of 1-127 if the message was successfully sent, or this message number plus 128 if the message was not sent for any reason.

RP1210_SendCommand **- Protect J1939 Address (if the status request byte is set to 1)**

wParam will have a value of 19 indicating this is a notification for a Protect J1939 Address request.

lParam will contain a zero if the Address Claim was successful or one of the following if the operation failed: (See Protect J1939 Address.)

```
ERR_COULD_NOT_TX_ADDRESS_CLAIMED
ERR_ADDRESS_CLAIM_FAILED
```

RP1210 Hardware Status Change

wParam will have a value of ERR_HARDWARE_STATUS_CHANGE

lParam will contain a zero

Upon fielding this Message, the Application should call *RP1210_GetHardwareStatus* to see the new status of the Hardware.

J1939 Address Lost

wParam will have a value of `ERR_ADDRESS_LOST`
lParam will contain the address that was lost

NOTE: The above Window Message Notifications are only available if the Application supplied a Window Handle on the *RP1210_ClientConnect* call.

This Window Message based Notification uses system capabilities which are supported across all existing Windows platforms. This strategy should be used by applications which will be built and distributed for multiple Windows platforms.

3.5 NOTIFICATION SCHEME FOR WINDOWS 95™ AND WINDOWS NT™

Windows 95™ and Windows NT™ support preemptive multitasking. This is accomplished through the use of threads. A thread describes a path of execution within a process or application. It could be thought of as a basic unit of CPU utilization. A process or an application can spawn other subprocesses as additional threads. Since, under 32-bit Windows™, each application has a primary thread automatically associated with it, the operating system divides CPU time between threads rather than applications. This provides for efficient CPU utilization and cleanly supported multitasking.

The strategy outlined in this document harnesses the power of multi-threading, which is supported by Windows 95™ and Windows NT™, while sparing the application developer specific details of its Win32 implementation.

Multi-threading shall be supported through the mechanisms of a blocking read and a blocking send. There is a flag in the *RP1210_ReadMessage* and the *RP1210_SendMessage* functions that allows the user to specify whether the functions should execute in a blocked or a non-blocked manner. For 16-bit Windows™, these flags shall be set to perform non-blocked reads and sends. Under Windows 95™ and Windows NT™, the reads and sends can be performed in a blocked manner.

Using the blocking mechanism, the application shall continuously call the *RP1210_ReadMessage* function (with the blocking flag set). The API DLL implementation of this function shall not return until a new message is received or an error is detected. The API DLL shall internally create a thread and use the *Win32 PulseEvent* and *WaitForSingleMessage* functions (or variants thereof) to efficiently wait for detection of a reception of a data link message. From the efficiency standpoint, it is preferable for the client application to create a new thread that performs the continuous blocking calls to the *RP1210_ReadMessage* function rather than do this in the main stream of the application code.

Under 32-bit Windows™, the *RP1210_SendMessage* shall also employ the blocking paradigm. An invocation of this function by the application shall not return till the API DLL can establish, as conclusively as possible , that the message has been delivered to the data link. The availability of this functionality makes the API implementation more concrete. This functionality may, however, require the institution of a transparent protocol between the API DLL and the device driver(s) by the API DLL developer. Again, as in *RP1210_ReadMessage*, it is best if the client application spawns a new thread when transmitting for maximum efficiency.

4.0 ATTACHING TO THE API DLL FROM AN APPLICATION

This RP requires Windows™ Applications to explicitly load the appropriate DLL and resolve references to the DLL supplied functions. This is accomplished by using the Window API functions, *LoadLibrary*, *GetProcAddress* and *FreeLibrary* (see the Windows 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*.

NOTE: If developing a 32-bit Visual Basic Application, the names defined should be as the Alias in the appropriate *Public Declare Function* statement.

APPENDIX III
RP1210 API REQUIRED FUNCTIONS

This section describes the required functions of the RP1210 API. Note that an implementation claiming to conform with this API shall minimally support these functions without deviating in name (case-sensitivity should be preserved), type, or argument list. As noted in **Section 3** of this RP, implementations may exceed the minimal functionality listed here, as long as the functionality provided by the given set of functions below is rigidly maintained. The required functionality is constituted by the following set of functions:

RP1210_ClientConnect	establishes connection between the client application and the API DLL
RP1210_ClientDisconnect	disconnects the client application from the API DLL
RP1210_SendMessage	sends a message to the API DLL
RP1210_ReadMessage	reads a message from the API DLL
RP1210_SendCommand	sends a command to the API DLL
RP1210_ReadVersion	reads version information from the API
RP1210_GetErrorMsg	translates an error code into a description of the error
RP1210_GetHardwareStatus	polls the interface hardware to determine a) if the interface hardware exists and b) what is the datalink status.

A3.1 RP1210_CLIENTCONNECT

This function is called by the client application seeking connection with a DLL that corresponds to the implementation of this API. Inside the API DLL, the function allocates and initializes any client data structures, and loads, initializes, or activates any device drivers (virtual or otherwise) to communicate with the hardware. If the connection is successful, the function shall return a unique identifier, corresponding to the ID of the client application, as assigned by the API DLL.

The default connection is in Converted Mode. In Converted Mode multiple clients (connections) can be established. However, if command ***Set J1708 Mode*** has been issued to change to Raw Mode, only one client can be connected. So, if a client has changed to Raw Mode, successive connects will fail. Likewise, if multiple clients are connected, trying to switch to Raw Mode will also fail.

The following illustrates the protocol names that should be used in Client Connects and each vendor's INI file:

Protocol String	**Protocol Description**
J1708	SAE J1708 network protocol
J1850	SAE J1850 network protocol
J1939	SAE J1939 network protocol
CAN	CAN network protocol

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_ClientConnect
(
             HWND hwndClient,
             short nDeviceID,
             char far* fpchProtocol,
             long lTxBufferSize,
             long lRcvBufferSize,
             short nIsAppPacketizingIncomingMsgs
);
```

Visual Basic Prototype

```
Declare Function RP1210_ClientConnect Lib “VENDRX.DLL”...
...(ByVal hwndClient As Integer, ByVal nDevice As Integer, ...
... ByVal fpchProtocol As String, ByVal lTxBufferSize As ...
... Long, ByVal lRcvBufferSize As Long, ByVal ...
... nIsAppPacketizingIncomingMsgs As Integer) As Integer
```

Parameters

`hwndClient`	A window handle identifying the application window that will receive posted event messages in Windows™. This handle is given by Form.hWnd in Visual Basic, where Form represents the name of the form (or window) in question. Under Windows™, when this parameter is passed as 0, the application will poll for its messages (and no events for the application shall, as such, be posted by the API DLL). Under Windows 95™ and Windows NT™, when this parameter is passed as 0, the application can poll for its messages (and no events for the application shall, as such, be posted by the API DLL) or use the blocking mechanism described in this RP.(See **Section 3.5**).
`nDeviceID`	Identifies the device with which the client application is requesting connection.
`fpchProtocol`	A pointer to a null-terminated string of the protocol name to be used by the device designated in the previous parameter. (See **Table 1** in **Appendix III** for protocol naming conventions.)
`lTxBufferSize`	A long integer for the requested size (in bytes) of the client transmit buffer to be allocated by the API for the queuing of messages sought to be transmitted by the client application. Should be passed as 0 if the application does not want to dictate the buffer size and the API DLL default of 8K is acceptable.
`lRcvBufferSize`	A long integer for the requested size (in bytes) of the client receive buffer to be allocated by the API for the queuing of messages meant to be received by the client application. Should be passed as 0 if the application does not want to dictate the buffer size and the API DLL default of 8K is acceptable.
`nIsAppPacketizingIncomingMsgs`	A flag that is relevant only for J1939. Should be set to zero if the application wants the lower-level components to perform the tasks of fragmenting the message into packets and return complete messages assembled from packets received. Should be set to 1 in the rare case where the application itself will perform the packetizing of message fragments.

Return Value

If the connection is successful, then the function returns a value between 0 and 127, corresponding to the client identifier that the application program is assigned by the API DLL. The application program must save this return value in order to conduct future transactions with the DLL. If the connection is unsuccessful, then an error code is returned that corresponds to a number greater than 127. The typical error codes are described below. More descriptive or proprietary codes may be returned by individual vendor implementations as long as the numerical equivalents of these return values are greater than 192, and the descriptions are clearly documented by the vendor.

Return Codes

`ERR_CLIENT_ALREADY_CONNECTED`	indicates that a client is already connected to the specified device.
`ERR_CLIENT_AREA_FULL`	indicates that the maximum number of connections has been reached.
`ERR_CONNECT_NOT_ALLOWED`	indicates that only one connection is allowed in the requested Mode.
`ERR_DEVICE_IN_USE`	indicates that the specified device is already in use and does not have the ability to maintain connections with multiple clients simultaneously.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_HARDWARE_NOT_RESPONDING`	indicates that the device hardware interface, through which the message is routed, is not responding.
`ERR_INVALID_DEVICE`	indicates that the specified device ID is invalid.
`ERR_INVALID_PROTOCOL`	indicates that the specified protocol is invalid or unsupported.
`ERR_NOT_ENOUGH_MEMORY`	indicates that the API DLL could not allocate enough memory to create a new client.

The numerical equivalents for these return codes are listed in **Appendix IV**.

Notes
For an application program to conduct future transactions with the API DLL, the application program must first successfully establish connection with it. In other words, the application must receive and recognize a successful return value from this function before calling any command. Under Windows™, this function would typically be called before the application enters its message processing loop.

If multiple J1939 applications are connected through the same device and any of them changes the *nIsAppPacketizingIncomingMsgs* flag, then the client attempting to change this flag shall receive the ERROR_CONNECT_NOT_ALLOWED return code. In other words, as long as there exist one or more J1939 connections to a given device that have set the *nIsAppPacketizingIncomingMsgs* flag, all applications issuing this command must also set this flag in a similar fashion, or they will receive the ERROR_CONNECTION_NOT_ALLOWED.

When a J1708, J1939, or CAN application establishes a connection, it is assumed that no filters are set and the drivers do not allow any data link messages to pass through. Only after filters have been set by using either command 3, 4, 5, or 7 of the *RP1210_SendCommand* functions, can the application start receiving messages.

Examples
For a C/C++ interface, assuming that *hwndMain* is the window handle for the main application window, and the client identifier returned by the API DLL is stored in a variable *nClientID*, the client application may have code somewhere that looks similar to this segment:

```
...
```

```
if (nRet = (RP1210_ClientConnect(hwndMain, 101, “J1708”,
             16000, 16000, 0) ) > 127)
{
       char pchBuf[ 256] ;
       fpchDescription[80];

       if (!RP1210_GetErrorMsg(nRet, fpchDescription))
              sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
       else
              sprintf(pchBuf, “Error #: %d. No description available.”, nRet);
       MessageBox(hWndMain, pchBuf, “Connect Error”, MB_ICONEXCLAMATION |
                     MB_OK);
}
else
{
       nClientID = nRet;

... /* client connected successfully, proceed as necessary */
}
...
```

In Visual Basic, a similar situation would be coded as follows:

```
...
nRet% = RP1210_ClientConnect(FormMain.hWnd, 101, ...
... “J1708”, 16000, 16000, 0)

If nRet > 127 Then
       If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
       Else
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                            “No description available.”
       End If
       nRet= MsgBox(Msg$, MB_ICONEXCLAMATION, “Connect Error”)
Else
       nClientID = nRet

...  ‘ client connected successfully, proceed as usual
End If
...
```

A3.2 RP1210_CLIENTDISCONNECT

This function is called by the client application seeking to terminate its connection with an API DLL that corresponds to the implementation of this API. Internal to the API DLL, the function de-allocates any client data structures and disables any active filters associated with the application. If the last client connected to the API calls this function, the DLL will also deactivate any device drivers (virtual or otherwise) that communicate with the hardware.

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_ClientDisconnect
(
       short nClientID
);
```

Visual Basic Prototype

```
Declare Function RP1210_ClientDisconnect Lib "VENDRX.DLL"...
...(ByVal nClientID As Integer) As Integer
```

Parameters

`nClientID`	The client identifier for the client that needs to be disconnected.

Return Value

If the disconnect is successful, then the function returns 0. Otherwise, a value greater than 127 is returned, corresponding to the code for the error that was registered in the processing of this function call. The typical error codes are described below. More descriptive or proprietary codes may be returned by individual vendor implementations as long as the numerical equivalents of these return values are greater than 192, and the descriptions are clearly documented by the vendor.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_FREE_MEMORY`	indicates that an error occurred in the process of memory de-allocation.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is being sought to be disconnected was invalid or unrecognized.

The numerical equivalents for these return codes are listed in **Appendix IV**.

Notes

This function can only be called only after a connection has been successfully established with the API DLL. The parameter passed to this function is the identifier for the connected client that is sought to be disconnected. Typically, a call to this function will be made in the clean-up code of the client application, when it is closing. It is the responsibility of the user application to terminate all of its threads that it has created to perform the IO.

If the API DLL calls *RP1210_ClientDisconnect* while blocking on *RP1210_ReadMessage* or *RP1210_SendMessage*, the DLL will signal the blocked function(s) to return with error value `ERR_CLIENT_DISCONNECTED`. This method will give a client application the ability to gracefully stop any threads waiting for returns from blocked functions. Any additional disconnect information concerning thread shut down is passed back in the call to *RP1210_GetErrorMsg*.

Examples

Assuming that the client identifier returned by the API DLL is stored in a variable *nClientID*, the client application may have code somewhere that looks similar to this segment:

```
...

if (nRet = (RP1210_ClientDisconnect(nClientID)) > 127)
{
       char pchBuf[256];
       fpchDescription[80];

       if (!RP1210_GetErrorMsg(nRet, fpchDescription))
              sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
       else
              sprintf(pchBuf, “Error #: %d. No description available.”, nRet);
       MessageBox(NULL, pchBuf, “Disconnect Error”, MB_ICONEXCLAMATION |
              MB_OK);
}
else
{
...    /* client disconnected successfully, proceed as necessary */
}

...
```

For Visual Basic, a similar situation would be coded as follows:

```
...

nRet = RP1210_ClientDisconnect(nClientID)

If nRet > 127 Then
       If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
       Else
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                            “No description available.”
       End If
       nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Disconnect Error”)
Else
...  ‘ client disconnected successfully, proceed as necessary

End If

...
```

A3.3 RP1210_SENDMESSAGE

This function is called by the client application to send a message to the device associated with it. The message is placed on a queue and processed in the order it was received.

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_SendMessage
(
       short nClientID,
       char far* fpchClientMessage,
       short nMessageSize,
       short nNotifyStatusOnTx,
       short nBlockOnSend,
)
```

Visual Basic Prototype

```
Declare Function RP1210_SendMessage Lib “VENDRX.DLL”...
...(ByVal nClientID As Integer, ByVal fpchClientMessage As...
...String, ByVal nMessageSize As Integer, ByVal...
...nNotifyStatusOnTx As Integer, ByVal nBlockOnSend As...
...Integer) As Integer
```

Parameters

`nClientID`	The identifier for the client that wants to transmit the message to its corresponding device.
`fpchClientMessage`	A pointer to the buffer (allocated by the client application) that contains the message to be sent. The protocol-specific message formats must be in conformance with those specified under the “Notes” section of this function.
`nMessageSize`	The size of the message in bytes (including any identifier or other qualifier bytes preceding the actual message data).
`nNotifyStatusOnTx`	When this flag is set, the API DLL will assign a message number between 1 and 127 to the message. In the event that the lower-level software either successfully transmits the message to the data link or is unable to do so (for whatever reasons), and this flag is set to 1, then an error notification will be sent to the client application. The error notification is accomplished through the WM_RP1210_ERROR_MESSAGE, with the error number (corresponding to ERR_TXMESSAGE_STATUS) returned in the *wParam* parameter, and message number and the transmit status returned in the *lParam* parameter, according to the scheme outlined below. If the message is successfully transmitted, then the message number is returned in *lParam*. Otherwise, in the case of failure, the message number added to 128 will be returned in *lParam*. If this flag is set to zero, then the application implicitly assumes successful message transmission at all times, and no backward notification mechanism is desired. For Windows 95™ and Windows NT™, the notification on transmit can be automatically provided through the blocking version of this function.
`nBlockOnSend`	A flag to indicate whether the function must block on message transmit or not. For Windows™ 3.1x, this flag is set to 0 (indicating blocking is **off**), while for Windows 95™ and Windows NT™, it can be set to 1 (indicating blocking is **on**). See Section 3.5 of this RP for more detail.

Return Value

If the *nNotifyStatusOnTx* flag described above is set to 0, then the function will always return a value of 0 for successful queuing of the message for transmission. Else, if the *nNotifyStatusOnTx* flag described above is 1, then the API DLL shall return a message number corresponding to a number between 1 and 127. If all available message numbers are assigned, the function shall return an error. In the event of an error in queuing the message for transmission, an error code, corresponding to a value of greater than 127 is returned (regardless of the*nNotifyStatusOnTx* flag setting).

The return value may also depend on the combination of parameters. Table A3.3.1, “*RP1210_SendMessage* Parameter Matrix” shows the possible parameter values and the corresponding action taken by the function.

Table A3.3.1: *RP1210_SendMessage* Parameter Matrix

nNotifyStatus	nBlockOnSend	OS	Window Handle	Actions
FALSE	FALSE	3.1	0	No notifications.
FALSE	FALSE	3.1	HWnd	No notifications.
FALSE	TRUE	3.1	0	Error. NBlockOnSend not supported
FALSE	TRUE	3.1	HWnd	Error. NBlockOnSend not supported
TRUE	FALSE	3.1	0	Error. Cannot notify w/o window handle.
TRUE	FALSE	3.1	HWnd	Notify the Application
TRUE	TRUE	3.1	0	Error. NBlockOnSend not supported.
TRUE	TRUE	3.1	Hwnd	Error. NBlockOnSend not supported.
FALSE	FALSE	95/NT	0	Ignore NotifyStatus.
FALSE	FALSE	95/NT	HWnd	Ignore NotifyStatus.
FALSE	TRUE	95/NT	0	Block until notification.
FALSE	TRUE	95/NT	HWnd	Ignore NotifyStatus.
TRUE	FALSE	95/NT	0	Error. Cannot notify w/o window handle.
TRUE	FALSE	95/NT	HWnd	Notify the Application.
TRUE	TRUE	95/NT	0	Error. Blocked notifications not supported.
TRUE	TRUE	95/NT	HWnd	Error. Blocked notifications not supported.

The typical error codes are described below. More descriptive or proprietary codes may be returned by individual vendor implementations as long as the numerical equivalents of these return values are greater than 192, and the descriptions are clearly documented by the vendor.

Return Codes

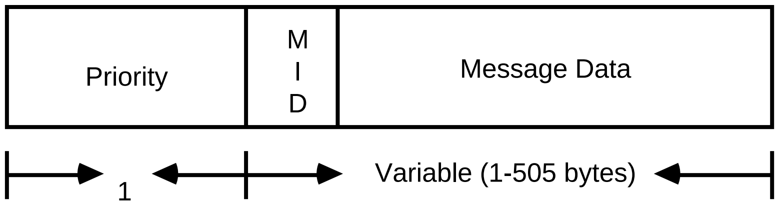
`ERR_ADDRESS_LOST`	indicates the API was forced to concede the address to another node on the J1939 network. This return code is for J1939 clients only.
`ERR_ADDRESS_NEVER_CLAIMED`	J1939 client never issued a protect address. This return code is for J1939 clients only.
`ERR_BLOCK_NOT_ALLOWED`	returned under Windows™ 3.1x if blocking was requested.
`ERR_CLIENT_DISCONNECTED`	indicates *RP1210_ClientDisconnect was called* while blocking on *RP1210_ReadMessage* or *RP1210_SendMessage.*
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_HARDWARE_NOT_RESPONDING`	indicates that the device hardware interface, through which the message is routed, is not responding.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking message transmission is invalid or unrecognized.
`ERR_MAX_NOTIFY_EXCEEDED`	returned if notification is requested and no more handles are available.
`ERR_MESSAGE_NOT_SENT`	returned if blocking is used and the message was not sent.
`ERR_MESSAGE_TOO_LONG`	indicates that the message being sought to be transmitted is too long.

`ERR_TX_QUEUE_CORRUPT`	indicates that the API DLL's transmit message queue is corrupt.
`ERR_TX_QUEUE_FULL`	indicates that the API DLL's transmit message queue is full.
`ERR_WINDOW_HANDLE_REQUIRED`	returned if a notification was requested and no Window Handle

The numerical equivalents for these return codes are listed in **Appendix IV**.

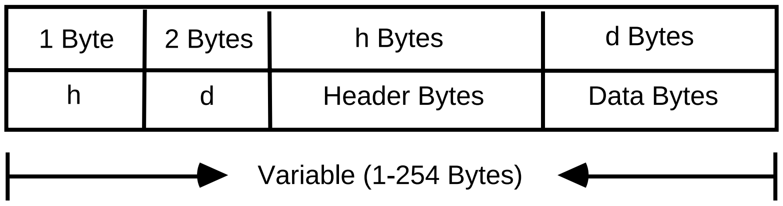
Notes
The message construction for the different protocols supported by this API must be in conformance with the specification detailed below.

J1708



With J1708, the first byte is the message priority. The remainder of the message beginning at the second byte is the message data which may be between 1 to 505 bytes. The *nMessageSize* parameter should reflect the size of the entire message, including the message priority byte. It is the responsibility of the API to add a checksum byte to the message before sending the J1708 message to the data link if the API DLL is in *Converted Mode*. (See **Appendix III**, **section A3.5.**)

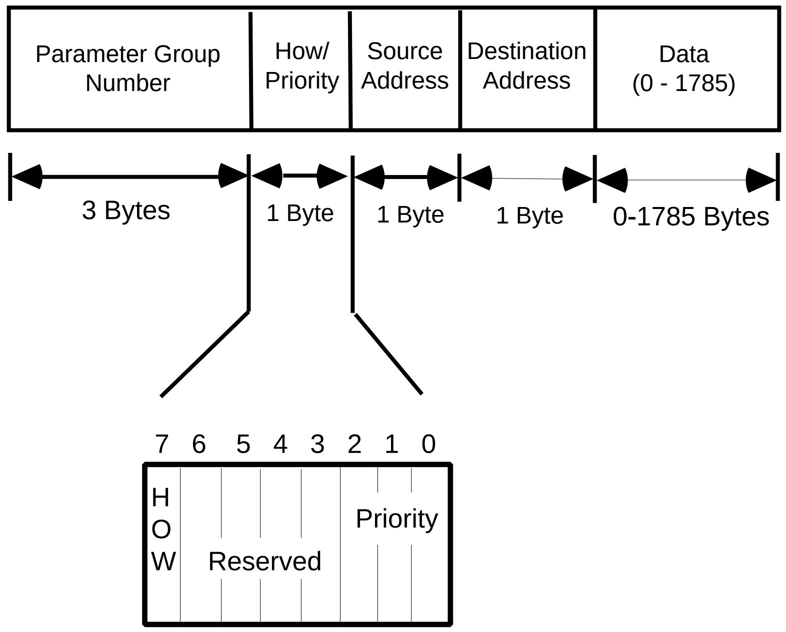
J1850



where	h = Number of header bytes
	d = Number of data bytes

The message data comprises the J1850 transmit messages and can be up to 255 bytes long.

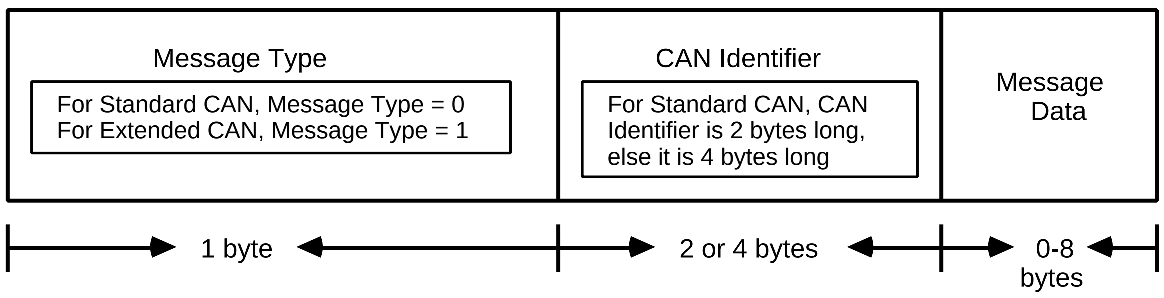
J1939



The J1939 SendMessage is constructed from five fields: *Parameter Group Number, How To Send/Priority, Destination Address, Source Address* and *Data* fields. The first field, *Parameter Group Number* (*PGN*), is three bytes in length and is in little endian/intel format. The second field, *How To Send/Priority*, is a bit field and is one byte (eight bits) in length. The higher order bit, bit 7,determines the J1939 transport type to use for data field lengths greater than eight bytes. A value of one in the high order bit will cause the lower level driver to use the Broadcast Announcement Message transport, a value of zero will enable the Connection Management transport facilities (reference SAE J1939/21 Table 4). The lower three bits (bits 0-2) determine the message priority. The other bits (bits 3-6) are reserved for future use and should be set to zero. The next field, *Source Address*, is one byte in length and contains the originating nodes address. If the *Source Address* field contains an address that has not been claimed, the API will still send the message (see Section A3.5, Protect J1939 Address). The fourth field, *Destination Address*, is one byte in length and contains the address of the node for which the packet is bound. The last field, *Data*, contains all the data elements of the message. The API DLL, or its associated drivers, is required to break down long messages and transmit multi-packet message fragments in accordance with the J1939 transport protocol detailed in Section 3.10 of SAE J1939/21.

The data field to send the J1939 EEC2 Message (PGN 61443) from address 6 to address 0 would be as follows: $03F000_{16}$, 03, 06, 00, FF FE 26 01 FF FF FF FF_{16}

CAN


Message Type
For Standard CAN, Message Type = 0
For Extended CAN, Message Type = 1
CAN Identifier
For Standard CAN, CAN Identifier is 2 bytes long, else it is 4 bytes long
Message Data
1 byte
2 or 4 bytes
0-8 bytes

The first byte of a CAN message identifies whether it is a Standard CAN message or an Extended CAN message. If this first byte equals 0, then the message is a Standard CAN message, otherwise it is an Extended CAN message. Based on whether the first byte is 0 or 1, the next two bytes either specify a two-byte Standard CAN Identifier or the next four bytes specify a four-byte Extended CAN Identifier. The final fragment (ranging in length from 0 to 8 bytes) identifies the message data. The *nMessageSize* parameter for a CAN transmit message should include the byte count for the message type as well as the CAN Identifier.

Examples

Assuming that the client identifier returned by the API DLL is stored in a variable *nClientID,* a client application may have code somewhere that looks similar to this segment:

```
...

char far* fpchMessage;

fpchMessage = _fmalloc(128);

/* build a message for transmission */
fpchMessage[0] = 0x80;
fpchMessage[1] = 0x92;
      ...
fpchMessage[23] = 0x62

if (nRet = (RP1210_SendMessage(nClientID, fpchMessage, 24,  0, 0) > 127)
{
char pchBuf[256];
char fpchDescription[80];

      if (!RP1210_GetErrorMsg(nRet, fpchDescription))
            sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
      else
            sprintf(pchBuf, “Error #: %d. No description available.”,
                  nRet);

      MessageBox(NULL, pchBuf, “Transmit Error”, MB_ICONEXCLAMATION |
                 MB_OK);
}
else
{
...   /* message sent successfully, proceed as necessary */
}
...
```

In Visual Basic, a similar situation would be coded as follows:

```
        ...

        fpchMessage$ = Space$(64)
        messageChar$ = Space$(64)

        ‘ build a message for transmission
        messageChar$ = Chr$(128)
        fpchMessage$ = fpchMessage$ + messageChar$
        messageChar$ = Chr$(146)
        fpchMessage$ = fpchMessage$ + messageChar$
        ...
        messageChar$ = Chr$(98)
        fpchMessage$ = fpchMessage$ + messageChar$

        nRet% = RP1210_SendMessage(nClientID, fpchMessage$, 24, 0, 0)

        If nRet > 127 Then
        If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
                Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
        Else
                Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                                “No description available.”
        End If
                nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Transmit Error”)
        Else
        ...     ‘ message sent successfully, proceed as necessary
End If

...
```

A3.4 RP1210_ReadMessage

This function is called by the client application to read a message from the device associated with it. The message is placed on a queue and processed in the order it was received.

C/C++ Prototype

```
--declpsec (dll export) WINAPI RP1210_ReadMessage
(
        short nClientID,
        char far* fpchAPIMessage,
        short nBufferSize,
        short nBlockOnRead
)
```

Visual Basic Prototype

```
Declare Function RP1210_SendMessage Lib “VENDRX.DLL”...
...(ByVal nClientID As Integer, ByVal fpchAPIMessage As...
...String, ByVal nBufferSize As Integer, ByVal nBlockOnRead...
...As Integer) As Integer
```

Parameters

`nClientID`	The client identifier for the client that wants to read the message received.

`fpchAPIMessage`	A pointer to the buffer (allocated by the client application) to where the transfer of the message is sought. If a message is correctly received, then the first four-bytes of the message represents the timestamp of the message (the resolution of this timestamp is implementation-specific, and can be retrieved from the vendor-supplied INI file), and the bytes following the timestamp shall be protocol-specific in accordance with the protocol detailed in the "Notes" section below.
`nBufferSize`	The maximum size of the message buffer.
`nBlockOnRead`	A flag to indicate whether the function must block on message reading or not. If this flag is set, the API will not return until a message has been received. For Windows™ 3.1x, this flag is set to 0. See **Section 3.5** for more detail.

Return Value
If the read is successful, then the function returns the number of bytes read including the four bytes for timestamp (for timestamp format, see Section 2.1). If no message is present, then the function returns 0. If an error occurred, then a value, corresponding to the additive inverse of the error code, is returned (for example, -185 is returned if the error corresponds to code 185). For Windows™ 3.1x, if the nBlockOnRead flag is set to 1, the function shall return an error indicating that this feature is not supported. The typical error codes are described below. More descriptive or proprietary codes may be returned by individual vendor implementations as long as the numerical equivalents of these return values are greater than 192, and the descriptions are clearly documented by the vendor.

Return Codes

`ERR_BLOCK_NOT_ALLOWED`	returned under Windows™ 3.1x if blocking was requested.
`ERR_CLIENT_DISCONNECTED`	indicates *RP1210_ClientDisconnect was called* while blocking on *RP1210_ReadMessage* or *RP1210_SendMessage*.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_HARDWARE_NOT_RESPONDING`	indicates that the device hardware interface, through which the message is routed, is not responding.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking the message read is invalid or unrecognized.
`ERR_MESSAGE_TOO_LONG`	indicates that the message being sought to be read is too long to fit in the client application-allocated message buffer.
`ERR_RX_QUEUE_CORRUPT`	indicates that the API DLL's message receive queue is corrupt.
`ERR_RX_QUEUE_FULL`	indicates that the API DLL's message receive queue is full.

The numerical equivalents for these return codes are listed in **Appendix IV**.

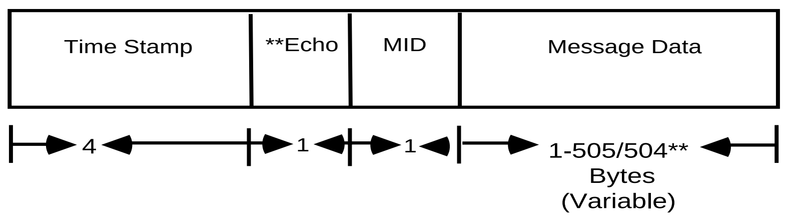
Notes

This function may be called in two ways. Under the Windows™ 3.1x messaging paradigm, whenever the incoming message queue goes from empty to non-empty, the API DLL posts an event for the appropriate application(s), to which the message is directed. Acting on that event in the message-processing loop of either its main window or some other window, designated to receive these messages, the client application calls the *RP1210_ReadMessage* function until this function returns zero. The client application may, alternatively, call this function on discrete intervals (every n milliseconds), to see if a message has been received—this polling paradigm is discouraged by the Windows™ 3.1x architecture.

Windows 95 ™and Windows NT™ may use the method described above for reading messages or the client application may continuously calls the *RP1210_ReadMessage* function with blocking enabled. The API DLL internally spawns a thread that waits for an event to be pulsed whenever a data link message is received to be passed on to the client application. From the client application perspective, the *RP1210_ReadMessage* is called and processed in a loop.

The message received for the different protocols supported by this API must be in conformance with the specification detailed below. In the case of J1708, the checksum bytes, if any, should be inspected by the lower-layer driver software and should not be passed up to the application through the API DLL unless the API DLL has been set to *Raw Mode*. (See **Appendix III**, **section A3.5.**)

J1708



** The echo byte is only there when Echo has been turned on.

The first four bytes comprise the time stamp. If the command ***Set Echo Transmitted Messages*** has configured the API to echo transmitted messages, this field is followed by a an echo byte which is set to one if the received message originated at the API's client application. If the received message did not originate at the client application, this byte is set to zero. (See Appendix III, A3.5 , command number 16.) If the API has not been configured to ***Set Echo Transmitted Messages,*** this byte is not present in the message. This is followed by the actual message which is in accordance with the J1708 Standard. If the message is correctly received, the return value will specify the size of the entire message, including the four timestamp bytes. The entire message, including time stamp and echo byte (if applicable), has a maximum size of 511 bytes.

J1850

4 Bytes	1 Byte	1 Byte	2 Bytes	1 Byte	h Bytes	d Bytes	i Bytes
Time Stamp	Echo	h	d	i	Header Bytes	Data Bytes	IFR Data

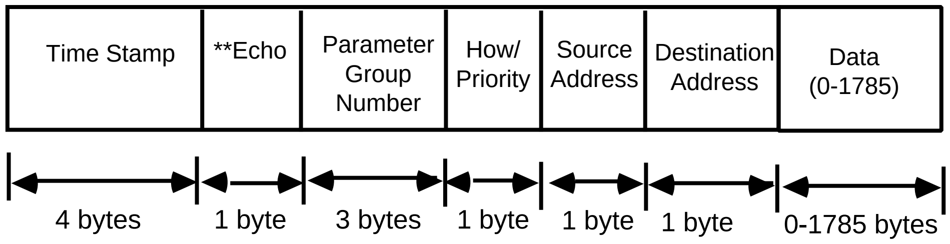
Variable (22-255 bytes)

where	h = Number of header bytes
	d = Number of data bytes excluding IFR Data
	i = Number of IFR data bytes

The minimum size for this buffer is 22 bytes.

The first four bytes comprise the time stamp followed by the actual message which is in accordance with the J1850 Standard. If the message is correctly received, the return value will specify the size of the entire message, including the four timestamp bytes. The echo byte will only be included when *Echo Transmitted Messages* enables Echo Mode. The message data has a CRC tagged at the end. In the case of IFR type 3, the IFR data also has a CRC tagged at the end of the IFR data. The assumption is that the CRC bytes will be stripped by default before the message is passed up from the RP1210 interface. Command number 15 can be used for J1850 also, to specify raw/converted mode. In raw mode the CRCs will be directly passed.

J1939



** The echo byte is only there when Echo has been turned on.

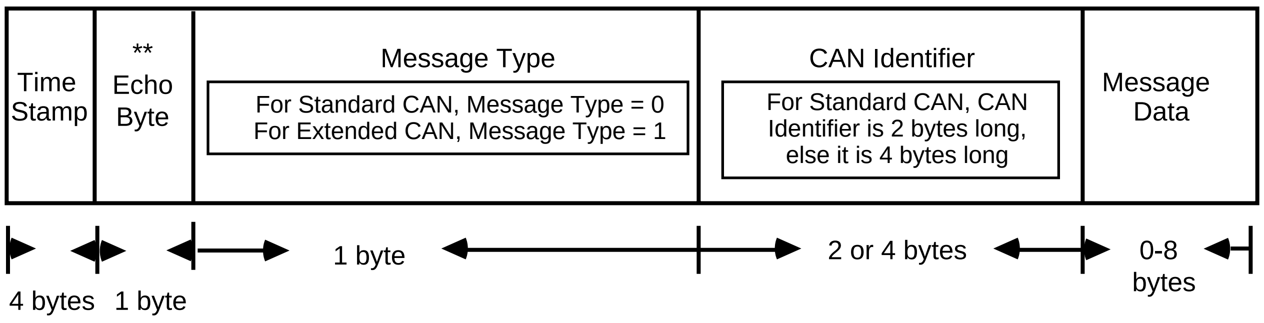
The J1939 receive message is constructed from seven fields: *Time Stamp*,*Echo*, *Parameter Group Number, How To Send/Priority, Destination Address, Source Address* and *Data* fields. The first field, *Time Stamp*, is four bytes in length and indicates the time the message arrived at the lower level interface. This field is placed on the header of the packet by the lower level software. For Messages greater that eight bytes, the *TimeStamp* field will contain the time that the last packet is received.

If the command ***Set Echo Transmitted Messages*** has configured the API to echo transmitted messages, the timestamp field is followed by a an echo byte which is set to one if the received message originated at the API's client application. If the received message did not originate at the client application, this byte is set to zero. (See **Appendix III, A3.5**, command number 16.) If the API has not been configured to ***Set Echo Transmitted Messages***, this byte is not present in the message. (See Appendix III, A3.5 , command number 16.)

The next field, *Parameter Group Number* (*PGN*), is three bytes in length and is in little endian or intel format. The next field, *How To Send/Priority*, is a bit field and is one byte (eight bits) in length. The higher order bit, bit 7,determines the J1939 transport type to use for data field lengths greater than eight bytes. A value of one in the high order bit, will cause the lower level driver to use the Broadcast Announcement Message transport. A value of zero will enable the Connection Management transport facilities. For proper usage of this bit refer to SAE J1939/21 Table 4. The lower three bits (bits 0-2) determine the message priority. The other bits (bits 3- 6) are unused and should be set to zero.

The *Source Address* field of a J1939 message is one byte in length and contains the originating nodes address. The next field, *Destination Address*, is one byte in length and contains the address of the node for which the packet is bound. The last field, *Data*, contains all the data elements of the message. If the message is correctly received, the return value will specify the size of the entire message, including the four bytes for the timestamp as well as the four bytes for the identifier.

CAN


Time Stamp
** Echo Byte
Message Type
For Standard CAN, Message Type = 0
For Extended CAN, Message Type = 1
CAN Identifier
For Standard CAN, CAN Identifier is 2 bytes long, else it is 4 bytes long
Message Data
4 bytes
1 byte
1 byte
2 or 4 bytes
0-8 bytes

** The echo byte is only there when Echo has been turned on.

The first four bytes of the message comprise the timestamp. If the command ***Set Echo Transmitted Messages*** has configured the API to echo transmitted messages, the next byte is an echo byte which is set to one if the received message originated at the API's client application. If the received message did not originate at the client application, this byte is set to zero. (See **Appendix III, A3.5**, command number 16.) If the API has not been configured to ***Set Echo Transmitted Messages*** this byte is not present in the message. (See Appendix A3.5 , command number 16.) The next byte identifies whether the message is a Standard CAN message or an Extended CAN message. If this byte equals 0, then the message is a Standard CAN message, otherwise it is an Extended CAN message. Based on whether this Message Type byte is 0 or 1, the next two bytes either specify a two-byte Standard CAN Identifier or the next four bytes specify a four-byte Extended CAN Identifier. The last field (ranging in length from 0 to 8 bytes) contains the message data. If the message is correctly received, the return value will specify the size of the entire message, including the four bytes for the timestamp as well as the message type and the CAN identifier.

Example

Assuming the ClientID returned by the API DLL is stored in a variable *usClientID*, the client application may have code somewhere that looks similar to this:

```
...
char far* fpchMessage;

fpchMessage = _fmalloc(512);

if (nRet = (RP1210_ReadMessage(usClientID, fpchMessage, 512, 0)))
{
      pchBuf[256];
      fpchDescription[80];

      if (!RP1210_GetErrorMsg(nRet, fpchDescription))
            sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
      else
            sprintf(pchBuf, “Error #: %d. No description available.”,
                        nRet);
      MessageBox(NULL, pchBuf, “Read Error”, MB_ICONEXCLAMATION |
                  MB_OK);
}
else if (nRet == 0)
{
 ...  /* no message, proceed as necessary */
      /* the first four bytes represent the timestamp */
}
```

```
        else
        {
        ...     /* fpchMessage buffer contains the message read */
                /* proceed as necessary */
        }
        ...
```

In Visual Basic, a similar situation would be coded as follows:

```
        ...

        fpchMessage$ = Space$(512)

        nRet% = RP1210_ReadMessage(usClientID, fpchMessage$, 512, 0)

        If nRet > 127 Then
                If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
                        Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
                Else
                        Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                                “No description available.”
                End If
                nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Read Error”)

        ElseIf nRet = 0
          ...           ‘ no message, proceed as necessary

        Else
         ...    ‘ fpchMessage buffer contains the message read
‘ proceed as necessary
        End If
        ...
```

A3.5 RP1210_SendCommand

This function is called by the client application to send a command to the device driver(s) associated with it. The command is typically intercepted by the driver and processed according to the command number. This method is notably used to set up filters for different protocols at the device level. The definition of the *fpchClientCommand* character buffer is dependent upon the n*CommandNumber* parameter.

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_SendCommand
(
        short nCommandNumber,
        short nClientID,
        char far* fpchClientCommand,
        short nMessageSize
)
```

Visual Basic Prototype

```
Declare Function RP1210_SendCommand Lib “VENDRX.DLL”...
...(ByVal nCommandNumber As Integer, ByVal nClientID As...
...Integer, ByVal fpchClientCommand As String, ByVal...
...nMessageSize As Integer) As Integer
```

Parameters

`nCommandNumber`	The command number, specified below for different functions defined as part of the TMC standard. Proprietary commands should be greater than 255, and new command numbers can be requested through TMC.

`nClientID`	The client identifier for the client that wants to transmit the command to its corresponding driver(s).
`fpchClientCommand`	A pointer to the buffer (allocated by the client application). This buffer is defined based on the command number.
`nMessageSize`	The total number of bytes in the *fpchClientCommand* buffer.

Reserved Values of `nCommandNumber` Parameter

TMC has reserved some values for the *nCommandNumber* parameter to support functionality considered required functionality. This section defines, in detail, the structure and constitution of these commands. The table below summarizes the send commands defined in this RP.

RP1210_SendCommand	Command Number
Reset Device	0
Set All Filter States to Pass	3
Set Message Filtering for J1939	4
Set Message Filtering for CAN	5
Set Message Filtering for J1708	7
Generic Driver Command	14
Set J1708 Mode	15
Set Echo Transmitted Messages	16
Set All Filter States to Discard	17
Set Message Receive	18
Protect J1939 Address	19

nCommandNumber = 0

Reset Device —This API command allows the user to attempt to reset the physical device. This command is allowed only when one client is actively connected. If more than one client is connected, this command will return an ERR_MULTIPLE_CLIENTS_CONNECTED error. Any requests pending by the user will be closed and the API will close the last connection prior to returning to the user. The user must reconnect after calling this API function by issuing a RP1210_ClientConnect call. This support is required for all implementations of the RP1210 API.

`nCommandNumber`	0
`nClientID`	The client identifier for the client that wants to control its filter states.
`fpchClientCommand`	An empty buffer (ignored).
`nMessageSize`	Always set to 0.

Return Value

If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_HARDWARE_NOT_RESPONDING`	indicates that the device is unable to perform a reset.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_MULTIPLE_CLIENTS_CONNECTED`	indicates that the API was not able to reset the device because more than one client is connected.

The numerical equivalents for these return codes are listed in **Appendix IV**.

<u>*nCommandNumber* = 3</u>

Set All Filter States To Pass —The API can set all message filter states to pass through this configuration of the *RP1210_SendCommand* function. By commanding "set all filter states to pass" it is implied that the API will allow all messages meant for the application to go through. This support is required for all implementations of the RP1210 API.

`nCommandNumber`	3
`nClientID`	The client identifier for the client that wants to control its filter states.
`fpchClientCommand`	An empty buffer (ignored).
`nMessageSize`	Always set to 0.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.

The numerical equivalents for these return codes are listed in **Appendix IV**.

<u>*nCommandNumber = 4*</u>

Set Message Filtering for J1939—The API can set up a message filter or enable the lower-level components to set up a message filter, through this particular configuration of the *RP1210_SendCommand* function. This support is required for all implementations of the RP1210 API that provide a J1939 interface.

The filter takes effect once the DLL's J1939 receive buffer is empty. Successive calls to the *RP1210_SendCommand* function with *nCommandNumber* equal to 4, would augment, rather than replace, the set of currently active J1939 filters. The lower-level layers will discard all incoming messages except those that match the associated filter parameters specified by this command.

`nCommandNumber`	4
`nClientId`	The client identifier
`fpchClientCommand`	A byte array specifying the filtering parameters. A filter parameter frame is shown below.
`nMessageSize`	The number of bytes in fpchClientCommand. Should be a multiple of seven.

Filter Parameter Frame

Byte 1	Bytes 2 thru 4	Byte 5	Byte 6	Byte 7
(1 Byte)	(3 bytes)	(1 Byte)	(1 Byte)	(1 Byte)
Filter Flag	PGN	Priority	Source Address	Destination Address

Filter Flag
The filter flag indicates which of the possible fields in this frame that should be used for filtering. Filtering is effected by matching the appropriate fields to the corresponding fields in the incoming message. The valid values are shown in the table below.

Name	**Value**	**Comments**
FILTER_PGN	1 (01_{16})	Use the specified PGN value for filtering
FILTER_SOURCE	4 (04_{16})	Use the specified source address for filtering
FILTER_DESTINATION	8 (08_{16})	Use the specified destination address for filtering
FILTER_PRIORITY	2 (02_{16})	Use the specified Priority value for filtering

To specify more than one option these values can be ORed together.

PGN
The PGN that needs to be filtered. This value is used only if the flag specifies FILTER_PGN option. This value is three bytes in length and is in little endian/intel format.
Valid Range : 0 – 1FFFFH

Priority
The priority of the messages that need to be filtered. This value is only used if the flag specifies the FILTER_PRIORITY option.
Valid Range : 0 – 7

Destination Address
The destination address of the messages that need to be filtered. This value is only used if the flag specifies the FILTER_DESTINATION option.
Valid Range : 0 – 255

Source Address
The source address of the messages that need to be filtered. This value is only used if the flag specifies the FILTER_SOURCE option.
Valid Range : 0 – 255

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

*nCommandNumber = 5*

Set Message Filtering for CAN - The API can set up a message filter or enable the lower-level components to set up a message filter, through this particular configuration of the *RP1210_SendCommand* function. This support is required for all implementations of the RP1210 API that provide a CAN interface.

The filter takes effect once the receive buffer is empty. Successive calls to the *RP1210_SendCommand* function with *nCommandNumber* equal to 5, would augment, rather than replace, the set of currently active CAN filters. The lower-level layers will discard all incoming messages except those that match the list specified by this command.

`nCommandNumber`	= 5
`nClientID`	The client ID for the client that wants to set the filter for the CAN messages.
`fpchClientCommand`	A byte array specifying the filtering parameters. A filter parameter frame is shown below.
`nMessageSize`	Number of bytes in fpchClientCommand buffer. It should be a multiple of nine bytes.

There are three parts to each filter. The first byte represents whether you are trying to set a filter for Standard or Extended CAN. A “0” represents Standard CAN and a “1” represents Extended CAN. The next 4 bytes represent a mask. The mask indicates which bits in the header need to be matched. A “1” means that the value of the bit is important; a “0” means that the value is unimportant. The last 4 bytes represent a header. The Header indicates what value is required for each bit of interest. To stay generic here, four bytes are used for both the mask and header regardless of whether Standard or Extended CAN is used. In the case of Standard CAN, only the low two bytes in both the mask and header would be used.

11 or 29 bit (1 byte)	Mask (4 bytes)				Header (4 bytes)			
0 or 1	High			Low	High			Low

The following is a Standard CAN (11 bit) example:
Header = 0481h
Mask = 0183h

then the bit pattern is as follows

Bit #	12 - 32	11	10	9	8	7	6	5	4	3	2	1
Header	N/A	1	0	0	1	0	0	0	0	0	0	1
Mask	N/A	0	0	1	1	0	0	0	0	0	1	1

This means that bits 1, 2, 8, and 9 are the only bits of interest in the header of the incoming message; the other bit values are ignored. Furthermore, the bit values of interest must match the mask exactly, to satisfy the filter condition:

Bit 1 = 1
Bit 2 = 0
Bit 8 = 1
Bit 9 = 0

(Notice that the “1” in bit 11 of Header has no effect)

Now with this setting if we receive 4 messages with message headers 681h, 689h, 609h and 9h respectively, only the first two messages will pass through.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 7

Set Message Filtering for J1708/J1587—The API can set up a message filter or enable the lower-level components to set up a message filter, through this particular configuration of the *RP1210_SendCommand* function. This support is required for all implementations of the RP1210 API that provide a J1708/J1587 interface.

The filter will be set up by supplying a list of module IDs (MIDs) in the *fpchClientCommand*. The lower-level layers will discard all incoming messages except for the ones that are associated with the MIDs supplied in this list.

Successive calls to the *RP1210_SendCommand* function, with *nCommandNumber* equal to 7, would not replace the set of currently active MID filters but would rather increment that set.

`nCommandNumber`	7
`nClientID`	The client identifier for the client that wants to transmit the command to its corresponding driver(s). This client must be associated with the J1708/J1587 protocol.
`fpchClientCommand`	A pointer to the buffer (allocated by the client application), containing a list of MIDs.
`nMessageSize`	The total number of bytes in the *fpchClientCommand* buffer. Note that the number of MIDs equals the number of bytes in the message.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 14

Generic Driver Command—This command allows the client application to send a generic message to its drivers. The API simply passes the message in the *fpchClientCommand* buffer down to the driver(s), if any, associated with the device hardware without intercepting or interpreting it. This support is optional for implementations of the RP1210 API.

`nCommandNumber`	14
`nClientID`	The client identifier for the client that wants to transmit the command to its corresponding driver(s). This client can be associated with the J1939 protocol.
`fpchClientCommand`	A free form buffer of bytes to be interpreted by the associated driver(s).
`nMessageSize`	The total number of bytes in the *fpchClientCommand* buffer.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_COMMAND_NOT_SUPPORTED`	indicates that the command number is not supported by the API DLL.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 15

Set J1708 Mode—The API can toggle the mode of the J1708 interface from *Converted Mode* to *Raw Mode*. When a J1708 link is established, the link, by default, is in *Converted Mode*.

Converted Mode is defined as follows: When a J1708 data packet is received, the lower level interface is verifying the checksum byte of the message, stripping off the checksum byte, and passing only validated messages to the client application. When the client application is sending a data packet, the lower level interface calculates the checksum byte and adds the byte to the trailer of the data packet.

Raw Mode is defined as follows: When a J1708 data packet is received, the lower level driver passes the message, intact, to the application. The checksum byte would neither be inspected nor removed. When the client application is sending a data packet, the application would be responsible for calculating and adding the checksum byte. This mode would expand the diagnostic capabilities of the API by giving a client application the ability to inspect the entire protocol and all J1708 bus traffic.

In addition, only one client can be connected if Raw Mode is turned on through this command. If only one client is connected and switches to Raw Mode, additional connects will fail. If one of multiple clients tries to switch to Raw Mode, this command will also fail.

Whenever this command is called to switch either to Raw or Converted Mode, the receive and send buffers allocated at *RP1210_ClientConnect* must be cleared.

This support is required for all implementations of the RP1210 API that provide a J1708 interface.

`nCommandNumber`	15
`nClientID`	The client identifier for the client that wants to transmit the command to its corresponding driver(s). This client must be associated with the J1708 protocol.
`fpchClientCommand`	A pointer to the buffer (allocated by the client application), containing '1 if *Converted Mode* is desired or '0' if *Raw Mode* is desired.
`nMessageSize`	The total number of bytes in the *fpchClientCommand* buffer.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_CHANGE_MODE_FAILED`	indicates that *Change_1708_Mode* was tried while multiple clients were connected.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 16

Set Echo Transmitted Messages—The API can toggle the lower level interface to echo transmitted messages back to the client application.

This support is required for all implementations of the RP1210 API.

When Echo Mode is turned on, the receive and send buffers allocated at *RP1210_ClientConnect* must be cleared.

Turning Echo on introduces a new byte into each incoming message that will indicate to the client whether the incoming message was the result of an echo or was an incoming message. This byte is only available when echo is turned on so the client must remember that all incoming messages will have an extra byte.

`nCommandNumber`	16
`nClientID`	The client identifier for the client that wants to transmit the command to its corresponding driver(s).
`fpchClientCommand`	A pointer to the buffer (allocated by the client application), containing '1' if *Echo On* is desired or '0' if *Echo Off* is desired.
`nMessageSize`	The total number of bytes in the *fpchClientCommand* buffer.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 17

Set All Filter States To Discard —The API can set all message filter states to discard through this configuration of the *RP1210_SendCommand* function. By "setting all filter states to discard," it is implied that the API will not allow any messages to go through. This support is required for all implementations of the RP1210 API.

`nCommandNumber`	17
`nClientID`	The client identifier for the client that wants to control its filter states.
`fpchClientCommand`	An empty buffer (ignored).
`nMessageSize`	Always set to 0.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

Return Codes

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 18

Set Message Receive - This command is used to enable or disable receive from the specified ClientID. By default a newly created connection has the receive turned on. The filter present at the time of enabling the receive will take effect. If the client is already receiving messages turning it on (again) has no effect. The client can stop receiving messages by calling this function and turning the receive off. If receiving has already stopped then the call will have no effect. When the receive has been turned off by this command, all messages will be lost.

`nCommandNumber`	18
`nClientID`	The client ID for the client that wants to turn receive On or Off.
`fpchClientCommand`	A pointer to a byte holding the command. A value of 1 will turn the receive on. A value of 0 will turn the receive off.
nMessageSize	Always set to 1.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned.

`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.

`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.

The numerical equivalents for these return codes are listed in **Appendix IV**.

nCommandNumber = 19

Protect J1939 Address—The API must have the ability to claim and protect a J1939 address on the vehicle network. When this command is issued, the API will attempt to claim the address that the application has requested. If the address is successfully claimed, the API will continue to protect the address. The API must support the claiming and protection of at least one address per client attached to the API.

If the API is forced to concede the address to another node on the network, the API will notify the application via the Windows Message, WM_RP1210_ERROR_MESSAGE (see 3.4) with wParam set to ERR_ADDRESS_LOST, lParam will contain the address that was lost, if a Window Handle was passed on the *RP1210_ClientConnect* call. If the application is in a polling mode, the next call to the RP1210_SendMessage() will return an error indicating that the address has been lost. In Windows 95™ and Windows NT™, any blocked calls to *RP1210_SendMessage()* will be released with an error. The API will also cease to handshake for RTS/CTS transport sessions to the lost address.

After the application has been notified of the lost address, the application can either attempt claim another address, or attempt to claim the global address to turn off network management. The application must perform one of these tasks before the API will allow it to send another message.

Once an address has been successfully claimed, the API may begin to handshake for J1939 RTS/CTS sessions to the protected address, depending on the parameters that were passed to the *RP1210_ClientConnect()* function. The API will ***not*** handshake for any messages that are not destined to an address that it is protecting. Upon the loss of a protected address, it is the duty of the API to abort current J1939 transport sessions, and perform necessary cleanup ***prior*** to the notification of the client application.

It is not required that the application send this command. If this command is never called, the application will be able to send and receive single frame messages to/from any address on the J1939 vehicle network.

Even if the client application is protecting an address, the API will allow the application to send from any address, determined by the identifier passed to the *RP1210_SendMessage()* function.

An address can be released by claiming a global address or claiming any second address.

`nCommandNumber`	19
`nClientID`	The client identifier for the client that wants to claim a J1939 address. This client must be associated with the J1939 protocol.
`fpchClientCommand`	A pointer to the buffer (allocated by the client application), containing in the first byte the 8-bit address that the client wishes to attempt to claim on the J1939 bus. This address is not limited to the service tool addresses. The next 8 bytes contain the 8-byte name of the client on the network. This name must be in compliance with the J1939 network management standard and is the responsibility of the client application to assemble. The last byte contains the requested

status byte for the protect address command. The status request byte is set by the application and can set to the following values:

BLOCK_UNTIL_DONE	0
POST_MESSAGE	1
RETURN_BEFORE_COMPLETION	2

`nMessageSize`	Always set to 10.

Return Value
If the command is successfully transmitted, then the function returns a value of 0 (unless otherwise specified for a specific command). Otherwise, an error code, corresponding to a value of greater than 127 is returned. If the requested status byte is set to BLOCK_UNTILL_DONE, this command will not return until the driver has completed the J1939 address claim procedure

Return Codes

`ERR_ADDRESS_CLAIM_FAILED`	indicates that the API was not able to claim the requested address.
`ERR_ADDRESS_LOST`	indicates the API was forced to concede the address to another node on the network.
`ERR_BUS_OFF`	indicates that the API was not able to transmit a CAN packet due to the hardware being BUS_OFF.
`ERR_COULD_NOT_TX_ADDRESS_CLAIMED`	indicates that the API was not able to request an address.
`ERR_COMMAND_NOT_SUPPORTED`	indicates that the command number is not supported by the API DLL.
`ERR_DLL_NOT_INITIALIZED`	indicates that the API DLL was not initialized.
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is seeking command transmission is invalid or unrecognized.
`ERR_INVALID_COMMAND`	indicates that the command number or the parameters associated with it are wrongly specified.
`ERR_WINDOW_HANDLE_REQUIRED`	indicates the user must supply a window handle to request the notify status.

The numerical equivalents for these return codes are listed in **Appendix IV**.

A3.5.1 Send_Command Examples

Assuming that the client identifier returned by the API DLL is stored in a variable *nClientID*, the client application may have code somewhere that looks similar to this segment:

```
...

char far* fpchMessage;

fpchMessage = _fmalloc(1);

/* build a message to filter out all */
/* but Engine #1 PIDs for J1708/ */
fpchMessage[0] = 0x80;

if (nRet = (RP1210_SendCommand(7, nClientID, fpch, 1)) > 127)
{
      pchBuf[256];
      fpchDescription[80];

      if (!RP1210_GetErrorMsg(nRet, fpchDescription))
            sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
      else
            sprintf(pchBuf, “Error #: %d. No description available.”,
                        nRet);
      MessageBox(NULL, pchBuf, “Command Error”, MB_ICONEXCLAMATION |
                  MB_OK);
}
else
{
...  /* command sent successfully, proceed as necessary */
}

...
```

In Visual Basic, a similar situation would be coded as follows:

```
...

fpchMessage$ = Space$(1);

‘ build a message to filter out all
‘ but Engine #1 PIDs for J1708/
fpchMessage$ = Chr$(128)

nRet% = RP1210_SendCommand(1, nClientID, fpchMessage$, 1)

If nRet > 127 Then
      If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
            Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
      Else
            Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                        “No description available.”
      End If
      nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Command Error”)

Else
...    ‘ command sent successfully, proceed as necessary

End If

...
```

A3.5.2 *RP1210_SendCommand* Return Codes

The table below shows the error codes returned by each of the send commands defined in this RP. The numerical equivalents for these return codes are listed in **Appendix IV**.

Error Codes	RP-1210 Send Command Number										
	0	3	4	5	7	14	15	16	17	18	19
ERR_ADDRESS_CLAIM_FAILED											✓
ERR_ADDRESS_LOST											✓
ERR_BUS_OFF											✓
ERR_CHANGE_MODE_FAILED							✓				
ERR_COMMAND_NOT_SUPPORTED						✓					
ERR_COULD_NOT_TX_ADDRESS_CLAIMED											✓
ERR_DLL_NOT_INITIALIZED	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ERR_HARDWARE_NOT_RESPONDING	✓										
ERR_INVALID_CLIENT_ID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ERR_INVALID_COMMAND			✓	✓	✓	✓	✓	✓		✓	✓
ERR_MULTIPLE_CLIENTS_CONNECTED	✓										
ERR_WINDOW_HANDLE_REQUIRED											✓

A3.6 *RP1210_ReadVersion*

This function is called by the client application to read the version information for the API DLL.

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_ReadVersion
(
       char far* fpchDLLMajorVersion,
       char far* fpchDLLMinorVersion,
       char far* fpchAPIMajorVersion,
       char far* fpchAPIMinorVersion
)
```

Visual Basic Prototype

```
Declare Sub RP1210_ReadVersion Lib “VENDRX.DLL” (ByVal...
...fpchDLLMajorVersion As String, ByVal fpchDLLMinorVersion As...
...String, ByVal fpchAPIMajorVersion As String, ByVal...
...fpchAPIMinorVersion As String)
```

Parameters

`fpchDLLMajorVersion`	A pointer to a character (a string in Visual Basic), which the API DLL fills with the major version of the DLL that corresponds to its implementation.
`fpchDLLMinorVersion`	A pointer to a character (a string in Visual Basic), which the API DLL fills with the minor version of the DLL that corresponds to its implementation.
`fpchAPIMajorVersion`	A pointer to a character (a string in Visual Basic), that corresponds to the major version of the TMC-produced API document with which the DLL conforms. This version’s ACSII character is “2.”
`fpchAPIMinorVersion`	A pointer to a character (a string in Visual Basic), that corresponds to the minor version of the TMC-produced API document with which the DLL conforms. This version‘s ACSII character is “0.”

Return Value

None. Values are returned in the passed parameters through pointers.

Notes
This function is for informative purposes, and may, very well, not be used by some applications that rely on only the most basic functionality of the API DLL.

Examples
This function, if used, may appear in the client application in a segment similar to the one exemplified below.

```
...

char* fpchDLLMajorVersion = _fmalloc(2);
char* fpchDLLMinorVersion = _fmalloc(2);
char* fpchAPIMajorVersion = _fmalloc(2);
char* fpchAPIMinorVersion = _fmalloc(2);
char pchBuf[256];

ReadVersion(fpchDLLMajorVersion,
             fpchDLLMinorVersion,
             fpchAPIMajorVersion,
             fpchAPIMinorVersion);

sprintf(pchBuf, “DLL Version: %c.%c\nAPI Version: %c.%c”,
         fpchDLLMajorVersion[0], fpchDLLMinorVersion[0],
         fpchAPIMajorVersion[0], fpchAPIMinorVersion[0]);

MessageBox(NULL, pchBuf, “Version Information”, MB_OK);
...
```

In Visual Basic, a similar situation would be coded as follows:

```
...

fpchDLLMajorVersion$ = Space$(1);
fpchDLLMinorVersion$ = Space$(1);
fpchAPIMajorVersion$ = Space$(1);
fpchAPIMinorVersion$ = Space$(1);

ReadVersion(fpchDLLMajorVersion$, fpchDLLMinorVersion$,...
...fpchAPIMajorVersion$, fpchAPIMinorVersion$)

Msg$ = “DLL Version: ” + fpchDLLMajorVersion$ + “.” +...
...fpchDLLMinorVersion$ + Chr$(13) + Chr$(10) + “API Version:...
... + fpchAPIMajorVersion$ + “.” + fpchAPIMinorVersion$

nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Version Information”)
...
```

A3.7 *RP1210_GetErrorMsg*

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_GetErrorMsg
(
       short ErrorCode,
       char far* fpchDescription

);
```

Visual Basic Prototype

```
Declare Function RP1210_GetErrorMsg Lib “VENDRX.DLL”...
...(ByVal nDevice As Integer, ByVal fpchDescription as String) As Integer
```

Parameters

`ErrorCode`	The numerical value for the last error which occurred.
`fpchDescription`	A pointer to the buffer (allocated by the client application) of 80 bytes, used to return the error message associated with the error code.

Return Value

If the function was able to successfully convert the given error code to a useful description, a value of 0 will be returned. If the function could not convert the error into a description an error code will be returned.

Return Codes

`ERR_CODE_NOT_FOUND`	indicates that there is no description available for the requested error code.

The numerical equivalents for these return codes are listed in **Appendix IV**.

Notes

The returned description is dependent on the order in which the function is called. This function should be invoked in response to the last error incurred by the API DLL. The returned string length should be no greater than 80 bytes in length including the NULL character (79 bytes of message).

Examples

This function, if used, may appear in the client application in a segment similar to the one exemplified below where *RP1210_xxxxxxx* represents an API DLL function that may return an error > 127.

```
if (nRet = (RP1210_xxxxxx(…,…,…,)
              ) > 127)
{
       pchBuf[256];
       fpchDescription[80];

       if (!RP1210_GetErrorMsg(nRet, fpchDescription))
              sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
       else
              sprintf(pchBuf, “Error #: %d. No description available.”,
                            nRet);
       MessageBox(NULL, pchBuf, “Error”, MB_ICONEXCLAMATION | MB_OK);
}
else
{
...  /* command sent successfully, proceed as necessary */
}

...
```

In Visual Basic, a similar situation would be coded as follows:

```
        ...

        nRet = RP1210_xxxxxx(…,…,…,)

        If nRet > 127 Then

               If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
                      Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
               Else
                      Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                                     “No description available.”
               End If
               nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Command Error”)
        Else
        ...    ‘ command sent successfully, proceed as necessary

        End If

        ...
```

A3.8 *RP1210_GetHardwareStatus*

This function is called by the client application to determine the hardware interface status and whether the device is physically connected or not.

C/C++ Prototype

```
--declspec (dll export) WINAPI RP1210_GetHardwareStatus
(
        short nClientID,
        char far* fpchClientInfo,
        short nInfoSize,
        short nBlockOnRequest
)
```

Visual Basic Prototype

```
Declare Function RP1210_GetHardwareStatus Lib “VENDRX.DLL”...
...(ByVal nClientID As Integer, ByVal fpchClientInfo As...
...String, ByVal nInfoSize As Integer) As Integer
```

Parameters

nClientID	The identifier for the client that wants to transmit the message to its corresponding device.
fpchClientInfo	A pointer to the buffer (allocated by the client application) where data link information is to be placed. The format of this buffer is defined as follows: 8 pairs of bytes. The low byte of each pair represents the status of the hardware/protocol. The high byte of each pair indicates the number of clients currently connected to the hardware/protocol. So, with the HWS or Hardware Status bytes, the low byte indicates whether the device is present. The high byte is the number of active connections. The rest indicate protocol information. The low byte indicates whether the protocol is connected and the high indicates the number of connected clients under that protocol.

0-1	2-3	4-5	6-7	8-9	10-11	12-13	14-15
HWS	J1939	J1708	CAN	J1850	Vendor 1	Vendor 2	Vendor 3

Hardware Status (Bytes 0-1)

Byte 0

Bit 0 is set if the hardware device has been located.
Bit 1 is set if the located hardware is an internal device.
Bit 2 is set if the located hardware is an external device.
Bit 3 reserved for future RP1210 use (set to 0).
Bits 4-7 are available for vendor specific use.

Byte 1

Indicates the current number of clients connected to this device.

J1939 (Bytes 2-3)

Byte 2

Bit 0 is set if the J1939 link has been activated.
Bit 1 is set if traffic has been detected on this link.
Bit 2 is set if CAN controller reports a BUS_OFF status.
Bit 3 reserved for future RP1210 use (set to 0).
Bits 4-7 are available for vendor specific use.

Byte 3

Indicates the current number of clients connected to this link.

J1708 (Bytes 4-5)

Byte 4

Bit 0 is set if the J1708 link has been activated.
Bit 1 is set if traffic has been detected on this link.
Bits 2-3 reserved for future RP1210 use (set to 0).
Bits 4-7 are available for vendor specific use.

Byte 5

Indicates the current number of clients connected to this link.

CAN (Bytes 6-7)

Byte 6

Bit 0 is set if the CAN link has been activated.
Bit 1 is set if traffic has been detected on this link.
Bit 2 is set if CAN controller reports a BUS_OFF status.
Bit 3 reserved for future RP1210 use (set to 0).
Bits 4-7 are available for vendor specific use.

Byte 7

Indicates the current number of clients connected to this link.

J1850 (Bytes 8-9)

Byte 8

Bit 0 is set if the J1850 link has been activated.
Bit 1 is set if traffic has been detected on this link.
Bits 2-3 reserved for future RP1210 use (set to 0).
Bits 4-7 are available for vendor specific use.

Byte 9

Indicates the current number of clients connected to this link.

Reserved for Vender (Bytes 10-15)

If these are not used, they should be set to zero.

`nInfoSize`	Always set to 16 bytes.
`nBlockOnRequest`	A flag to indicate whether the function must block on requesting the hardware status or not. For Windows™ 3.1x, this flag is set to 0 (indicating blocking is **off**), while for Windows 95™ and Windows NT™, it can be set to 1 (indicating blocking is **on**). See Section 3.5 of this RP for more detail. A call with blocking on will wait until the hardware status, byte 0 or the low byte from the protocol changes. At that time, the function will return with the new current status. For Windows™ 3.1x, Windows 95™ and Windows NT™, if a window handle was passed to *RP1210_ClientConnect*, a window message indicated by WM_RP1210_ERROR_MESSAGE will be posted to the application with the wParam parameter set to ERR_HARDWARE_STATUS_CHANGE. The application should then call *RP1210_GetHardwareStatus* to obtain the new status.

Return Value

If the function determines that the hardware interface is present and functioning, then the function returns a value of 0. Otherwise, an error code, corresponding to a value of greater than 127 is returned. If no error occurs, the function also places data link information in the client application provided buffer.

Return Codes

`ERR_BLOCK_NOT_ALLOWED`	returned under Windows^{TM} 3.1x if blocking was requested.
`ERR_CLIENT_DISCONNECTED`	indicates *RP1210_ClientDisconnect was called* while blocking on *RP1210_GetHardwareStatus.*
`ERR_INVALID_CLIENT_ID`	indicates that the identifier for the client that is requesting the hardware status is invalid.

Example

Assuming that the client identifier returned by the API DLL is stored in a variable *nClientID*, the client application may have code somewhere that looks similar to this:

```
...
char far* fpchInfo;

fpchInfo = _fmalloc(2);

if ((nRet = RP1210_GetHardwareStatus(nClientID,fpchInfo, 2, 0)) != 0)
{
      pchBuf[256];
      fpchDescription[80];

      if (!RP1210_GetErrorMsg(nRet, fpchDescription))
            sprintf(pchBuf, “Error #: %d. %s”, nRet, fpchDescription);
      else
            sprintf(pchBuf, “Error #: %d. No description available.”,
                        nRet);
      MessageBox(NULL, pchBuf, “Error”, MB_ICONEXCLAMATION | MB_OK);     }
else
```

```
{
...    /* fpchInfo contains data link information */
       /* proceed as necessary */
}
...
```

In Visual Basic, a similar situation would be coded as follows:

```
...

fpchMessage$ = Space$(2)

nRet% = RP1210_ReadMessage(nClientID, fpchInfo$, 2)

If (nRet > 127) Then
       If (!RP1210_GetErrorMsg(nRet, fpchDescription)) Then
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” + fpchDescription
       Else
              Msg$ = “Error #:  ” + Str$(nRet) + “. ” +
                            “No description available.”
       End If
       nRet = MsgBox(Msg$, MB_ICONEXCLAMATION, “Read Error”)

Else
 ...   ‘ fpchInfo buffer contains data link information

       ‘ proceed as necessary
End If
...
```

This section lists the numerical equivalents for the error or return codes identified in this document and used in **Section 3** of this RP.

APPENDIX IV
RETURN CODE EQUIVALENCIES

ERR_ADDRESS_CLAIM_FAILED	146
ERR_ADDRESS_LOST	153
ERR_ADDRESS_NEVER_CLAIMED	157
ERR_BLOCK_NOT_ALLOWED	155
ERR_BUS_OFF	151
ERR_CANNOT_SET_PRIORITY	147
ERR_CHANGE_MODE_FAILED	150
ERR_CLIENT_ALREADY_CONNECTED	130
ERR_CLIENT_AREA_FULL	131
ERR_CLIENT_DISCONNECTED	148
ERR_CODE_NOT_FOUND	154
ERR_COMMAND_NOT_SUPPORTED	143
ERR_CONNECT_NOT_ALLOWED	149
ERR_COULD_NOT_TX_ADDRESS_CLAIMED	152
ERR_DEVICE_IN_USE	135
ERR_DLL_NOT_INITIALIZED	128
ERR_FREE_MEMORY	132
ERR_HARDWARE_NOT_RESPONDING	142
ERR_HARDWARE_STATUS_CHANGE	162
ERR_INVALID_CLIENT_ID	129
ERR_INVALID_COMMAND	144
ERR_INVALID_DEVICE	134
ERR_INVALID_PROTOCOL	136
ERR_MAX_FILTERS_EXCEEDED	161
ERR_MAX_NOTIFY_EXCEEDED	160

ERR_MESSAGE_NOT_SENT	159
ERR_MESSAGE_TOO_LONG	141
ERR_MULTIPLE_CLIENTS_CONNECTED	156
ERR_NOT_ENOUGH_MEMORY	133
ERR_RX_QUEUE_CORRUPT	140
ERR_RX_QUEUE_FULL	139
ERR_TXMESSAGE_STATUS	145
ERR_TX_QUEUE_CORRUPT	138
ERR_TX_QUEUE_FULL	137
ERR_WINDOW_HANDLE_REQUIRED	158

APPENDIX V
INI FILE FORMAT

A5.1 File Format of the RP1210.INI/RP121032.INI File

Each vendor is likely to have its own API DLL implementation of the RP1210 API. So, any given PCcould have a number of such DLLs with names like VENDRX.DLL, VENDRX32.DLL, VENDRY.DLL, VENDRY32.DLL, Z_1210.DLL, Z_121032.DLL etc. Those DLLs, in this example, are all implementations of the RP1210 API standard. Of course, the client application may not know what, if any, DLLs are present. It should be able to go to a common source and determine which vendor DLLs are present. This common source will be the RP1210.INI file (for Windows™ 3.1x) or the RP121032.INI file (for Windows™ 95 and Windows™ NT), with the minimal format outlined below:

```
[RP1210Support]
APIImplementations=[string1],[string2],...,[stringn]
```

where each string$_i$ identifies the name of the DLL-INI couple associated with a particular vendor's implementation.

An example of the RP1210.INI file would, then, look like:

```
[RP1210Support]
APIImplementations=VENDRX,VENDRY,Z_1210
```

This implies that, in this particular case, the client application has access to the following DLLs and INI files: `VENDRX.DLL, VENDRX.INI, VENDRY.DLL, VENDRY.INI, Z_1210.DLL, and Z_1210.INI.`

An example of the RP121032.INI file would, then, look like:

```
[RP1210Support]
APIImplementations=VENDRX32,VENDRY32,Z_121032
```

This implies that, in this particular case, the client application has access to the following DLLs and INI files: `VENDRX32.DLL, VENDRX32.INI, VENDRY32.DLL, VENDRY32.INI, Z_121032.DLL, and Z_121032.INI.`

The installation program for each vendor-provided API DLL would go through the following steps (taking VENDRX as an example):

1. Check to see if the RP1210.INI file (under Windows™ 3.1x) or the RP121032.INI file (under Windows™ 95 and Windows™ NT) exists in the Windows™ directory (the directory where the Windows™ operating system is installed). We will assume a Windows™ 3.1x installation for this example.

2. If it does not exist, create it, and write it as:

   ```
   [ RP1210Support]
   APIImplementations=VENDRX
   ```

   and go to step 3.

   If, however, the RP1210.INI file does exist on the computer, then retrieve the string for keyname APIImplementations (use the GetPrivateProfileString function for this purpose). Scan the returned to string check if it contains the sub-string "VENDRX". If it does exist, go to Step 3; otherwise, append the string ",VENDRX" to the end of the string that was returned and overwrite the APIImplementations keyname value with the newly appended string, and proceed to the next step.

3. Check to see if the VENDRX.INI file exists on the computer's Windows™ directory (the directory where the Windows™ operating system is installed).

4. If VENDRX.INI does not exist, create one according to the vendor INI file specification detailed below. Otherwise, make the appropriate additions to it based on the new device support.

Since the RP1210.INI file is one that may, possibly, be written by a number of vendors, it is important that extreme care be taken in the modification of this file by any setup program. An incorrect modification introduced by the setup program of one vendor could very well jeopardize the function of the API DLL implementation of another.

A5.3 File Format of the Vendor-Provided INI File

Each vendor providing a API DLL implementation of the RP1210 API shall also provide an associated INI file. This file would contain information on the hardware devices and protocols supported by the vendor and would be updated every time the vendor's implementation of the RP1210 API is changed to accommodate a new device or protocol. Application programs will parse a vendor-provided INI file for the required connection parameters. This file may also (optionally) contain information to contact the vendor. The file format is described below:

`[VendorInformation]`	
`Name=[string]`	where `string` represents the name of the vendor enclosed
`Address1=[string]`	where `string` represents the first line in vendor's mailing address—optional
`Address2=[string]`	where `string` represents the second line of the vendor's address—optional
`City=[string]`	where `string` represents the city that the vendor is located in—optional
`State=[string]`	where `string` represents the state/province of vendor's location—optional
`Country=[string]`	where `string` represents the country where the vendor is located—optional
`Postal=[string]`	where `string` represents the vendor's postal or ZIP code—optional
`Telephone=[string]`	where `string` encapsulates a telephone number to reach support—optional
`Fax=[string]`	where `string` represents a fax number to reach the vendor—optional
`MessageString=[string]`	where `string` is a unique message string for the vendor DLL
`ErrorString=[string]`	where `string` is a unique error string for the vendor DLL
`TimestampWeight=[number]`	where `[number]` represents the weight, per bit, in microseconds of the timestamp
Devices=[d_1],[d_2],...,[d_n]	where d's represent the device IDs for the devices supported by this vendor; each d_i is a number between 0 and 255; the device IDs are delimited by commas with no spaces in between
Protocols=[p_1],[p_2],...,[p_m]	where p_i's represents the identifiers for the protocols supported by this vendor; each protocol identifier is a number between 0 and 255; the protocol identifiers are delimited by commas with no spaces in between; note that, unlike the device IDs, the protocol identifiers have no significance in the application software other than to concoct unique tokens for the different protocols supported by the vendor, and can be arbitrarily assigned

[DeviceInformationd$_1$]

DeviceID=[number]	where [number] represents a value ranging between 0 and 255, note that here: number = d_1
DeviceDescription=[string,p]	where [string] represents a description of the device. This description is normally the name of the product and is typically listed on the device. When the device can be connected to several ports, the value of [p] is also specified after a comma. If the device does not support several ports, [p] can be left blank. [p] represents the port the device is connected to, the table below lists several values for p.

P Value	Description
COM1	Computers first serial port
COM2	Computers second serial port
COMN	Computers Nth serial port
LPT1	Computers first parallel port
LPT2	Computers second parallel port
LPTN	Computers Nth parallel port
PCMCIA	PCMCIA Card Device

DeviceName=[string]	where [string] represents the a vendor-specific device name.
DeviceParams=[string]	where [string] represents vendor-specific parameters, if any—optional

[DeviceInformationd$_2$]

DeviceID=[number]	where [number] represents a value ranging between 0 and 255, note that here: number = d_2
DeviceDescription=[string,p]	where [string] represents a description of the device. This description is normally the name of the product and is typically listed on the device. When the device can be connected to several ports, the value of [p] is also specified after a comma. If the device does not support several ports, [p] can be left blank. [p] represents the port the device is connected to, the table below lists several values for p.

P Value	Description
COM1	Computers first serial port
COM2	Computers second serial port
COMN	Computers Nth serial port
LPT1	Computers first parallel port
LPT2	Computers second parallel port
LPTN	Computers Nth parallel port
PCMCIA	PCMCIA Card Device

DeviceName=[string]	where [string] represents the a vendor-specific device name.

`DeviceParams=[string]`	where `[string]` represents vendor-specific parameters, if any—optional
	. . .
`[DeviceInformationd`$_n$`]`	
`DeviceID=[number]`	where `[number]` represents a value ranging between 0 and 255, note that here: number = d_n
`DeviceDescription=[string,p]`	where `[string]` represents a description of the device. This description is normally the name of the product and is typically listed on the device. When the device can be connected to several ports, the value of [p] is also specified. If the device does not support several ports, [p] can be left blank. [p] represents the port the device is connected to, the table below lists several values for p.

P Value	Description
COM1	Computers first serial port
COM2	Computers second serial port
COMN	Computers Nth serial port
LPT1	Computers first parallel port
LPT2	Computers second parallel port
LPTN	Computers Nth parallel port
PCMCIA	PCMCIA Card Device

`DeviceName=[string]`	where `[string]` represents the a vendor-specific device name.
`DeviceParams=[string]`	where `[string]` represents vendor-specific parameters, if any—optional
`[ProtocolInformationp`$_1$`]`	
`ProtocolDescription=[string]`	where `[string]` represents a description of the protocol
`ProtocolString=[string]`	where `[string]` represents the protocol string expected by the *RP_1210ClientConnect* function
`ProtocolParams=[string]`	where `[string]` represents vendor-specific parameters, if any, associated with the protocol—optional
`Devices=[`$k_1,k_2,...,k_t$`]`	where `[`$k_1,k_2,...,k_t$`]` represents the list of devices associated with this protocol—each k_i is a DeviceID and the list is comma-delimited (without any spaces in between). Note that, mathematically: $\{[k_1,k_2,...,k_t]\}$ Õ$\{[d_1,d_2,...,d_n]\}$
`[ProtocolInformationp`$_2$`]`	
`ProtocolDescription=[string]`	where `[string]` represents a description of the protocol
`ProtocolString=[string]`	where `[string]` represents the protocol string expected by the

	RP_1210ClientConnect function
`ProtocolParams=[string]`	where `[string]` represents vendor-specific parameters, if any, associated with the protocol—optional
`Devices=[`$k_1,k_2,...,k_t$`]`	where `[`$k_1,k_2,...,k_t$`]` represents the list of devices associated with this protocol—each k_i is a DeviceID and the list is comma-delimited (without any spaces in between). Note that, mathematically: $\{[k_1,k_2,...,k_t]\}$ Õ $\{[d_1,d_2,...,d_n]\}$
`[ProtocolInformationp`$_m$`]`	
`ProtocolDescription=[string]`	where `[string]` represents a description of the protocol
`ProtocolString=[string]`	where `[string]` represents the protocol string expected by the *RP_1210ClientConnect* function
`ProtocolParams=[string]`	where `[string]` represents vendor-specific parameters, if any, associated with the protocol—optional
`Devices=[`$k_1,k_2,...,k_t$`]`	where `[`$k_1,k_2,...,k_t$`]` represents the list of devices associated with this protocol—each k_i is a DeviceID and the list is comma-delimited (without any spaces in between). Note that, mathematically: $\{[k_1,k_2,...,k_t]\}$ Õ $\{[d_1,d_2,...,d_n]\}$

For example a VENDRX.INI may look something like:

```
[VendorInformation]
Name=VendorX, Inc.
MessageString=VENDORX RP1210 INTERRUPT
ErrorString=VENDORX RP1210 ERROR
TimeStampWeight=500
Devices=0,100,101
Protocols=0,10,11,18

[DeviceInformation0]
DeviceID=0
DeviceDescription=ISA-BUS VENDORX Snoop Card
DeviceName=Generic Interface Card
DeviceParams=0x2000-0x2DDD

[DeviceInformation100]
DeviceID=100
DeviceDescription=Standard PC Port, COM1
DeviceName=COM1
DeviceParams=IRQ=5

[DeviceInformation101]
DeviceID=101
DeviceDescription=Standard PC Port, COM2
DeviceName=COM2

[ProtocolInformation0]
ProtocolDescription=J1708 Link Layer Protocol
ProtocolString=J1708
ProtocolParams=NODROP MODE
Devices=0

[ProtocolInformation10]
ProtocolDescription=J1939 Link Layer Protocol
ProtocolString=J1939
Devices=0
```

A5.4 Standardization of the Vendor INI file

In order to provide meaningful descriptions for the users of a vendor specific INI files, this RP defines several ProtocolDescriptions and DeviceNames that are to be used. In addition to these descriptions, any vendor that implements an RP1210 compliant API DLL shall obtain a unique vendor name.

The following is a list of protocols and protocol descriptions:

J1708	ProtocolDescription = J1708 Link Layer Protocol
J1939	ProtocolDescription = J1939 Link Layer Protocol
J1850_3H	ProtocolDescription = J1850 3H Link Layer Protocol
J1850_1H	ProtocolDescription = J1850 1H Link Layer Protocol
CAN	Protocol Description = Generic CAN

APPENDIX VI
TERMINOLOGY

The following table describes the terminology used in this specification:

Abbreviation	**Description**
API	Application Program Interface. A documented methodology for programming application programs such that they operate cooperatively with other programs. Software programmers write applications that conform to one or many API's.
DLL	Dynamic Link Library. A DLL can be an implementation of an API. A mechanism to link applications to libraries at run-time. The libraries reside in their own (special) executable files and are not copied into an application's executable files as with static linking. These libraries are called dynamic-link libraries (DLL) to emphasize that they are linked to an application when it is loaded and executed, rather than when it is linked.
Real-time Processing/Communication	Computation that requires operations to be performed at a high speed (often of the order of milliseconds) and, at times, in an unconditionally prioritized manner. This kind of processing is typical of PC to vehicle data link communications.
RP 1202 API	A recommended practice by The Maintenance Council (TMC) for vehicle communication under DOS. An implementation of this API may have the ability to support RP1202 type communications as an option.
RP1208	TMC Recommended Practice for PC Service Tool hardware selection.
Non-multitasking operating system	An operating system that allows only one process (or task) to execute at a given time. This allows the majority of system resources to be dedicated to the running process, making it eminently suited for real-time processing. DOS is an example of such an operating system. Compare with multitasking operating systems.
Multitasking operating system	An operating system that allows multiple processes to execute at a given time. Such an operating system divides its time (or time-slices) and resources between the different running processes. Typically, time-slicing makes it difficult to do dedicated real-time processing with such systems. Windows™ is an example of such an operating system. Compare with non-multitasking operating systems.
Event-driven Architecture	An operating system architecture that revolves around a messaging scheme governed by system events (a mouse click, a keyboard press are all examples of events). When an application receives an event directed to it, either from the operating system or another application or driver, it executes the code associated with that event. Windows™ architecture is modeled on this paradigm.

RS-232	A term usually used in reference to the communication ports available on most personal computers. Formally defined as the “Recommended Standard” 232 in the ANSI (American National Standard Institution) specification as “the interface between data terminal equipment and data circuit-terminating equipment employing serial binary data interchange.”
RS-485	A term used in reference to a communication (COM) port available on some personal computers. Formally, “Recommended Standard” 485 in the ANSI specification that defines a transmission scheme. This standard is widely utilized in the trucking industry for interfacing with the vehicle data link.
Network Protocol	A documented standard for network communications. Examples of network protocol standards in the automotive industry: SAE J1587, J1708, J1850, J1922, J1939; CAN (Controller Area Network)
Big Endian or Motorola Format	Also referred to in the document as the “most significant byte first format.” A format for representing integral bytes of data, conventionally used in Motorola micro-processors, where the bytes are numbered from left to right. The Intel format, called Little Endian, numbers the bytes from right to left. The terms “Big Endian” and “Little Endian” are coined after the politicians in Gulliver’s Travels who went to war over which end of an egg to break.
Interface box	A term usually used in the industry to refer to hardware components that translate between the RS-485 link interfacing with the vehicle data link, on one hand, and the RS-232 port of the personal computer on the other. An interface box may be a hand-held diagnostic tool that supports this translation or specially designed RS-485 to RS-232 translation hardware.
PC interface card	A piece of hardware that interfaces with the personal computer on one hand and the vehicle data link on the other. Such cards would, typically, plug into one of the expansion slots on the PC.
PCMCIA card	PCMCIA stands for Personal Computer Memory Card Interface Association. The term “memory card” is a misnomer as these cards, about the size of credit cards, now support many other functions like modems, Ethernet interfaces, and, conceivably, a vehicle data link interface.

APPENDIX VII
CLARIFICATION NOTES FOR APPLICATION DEVELOPERS

INTRODUCTION

TMC RP 1210A provides a series of API functions that abstract the vehicle interface adapter from the application. Just as a printer API enables a program to get the same hard copy results regardless of the printer type, RP 1210A allows a diagnostic application to communicate with the vehicle without regard to most of the unique characteristics of individual vehicle interface adapters. In either case, a generalized set of functions can be used, assuming, of course, that the vehicle interface adapter conforms to TMC RP 1210A.

Even with the elaborate details provided in RP 1210A's function definitions, there have been slight differences in the interpretation of those functions and how certain situations are handled. Two of these differences—unexpected disconnect and filter state ambiguities—are described below with recommendations on how to manage them.

Issue 1: Unexpected Vehicle Interface Adapter Disconnect From Vehicle or Computer.

There are differences in how an adapter disconnect is managed among vehicle interface adapter vendors. The RP indicates that the API should return error code 14210—"ERR_HARDWARE_NOT_RESPONDING"—to the calling application whenever a hardware error occurs. However, the reaction of Vehicle Interface Adapter vendor API's differ greatly when hardware is accidentally disconnected. Some require no intervention from the client application while others require full client application management of the disconnect. Because of these differences, it is recommended that all applications go through the following procedure:

The client application monitors for changes in hardware status by periodically calling the "RP1210_GetHardwareStatus()" function and/or processing the return code from "RP1210_SendMessage()" or "RP1210_ReadMessage()", to determine the status of the hardware translator. If an error condition exists, it notifies the operator and recommends that a check of the vehicle and computer connections be made before proceeding.

In severe cases the client application may be required to do the following:

a. Disconnect from the API by using RP1210_ClientDisconnect().
b. Wait for all other applications running on the host computer to disconnect from the API.
c. Reconnect running applications to the API using RP1210_ClientConnect().
d. Set up all filter states again.
e. Reclaim all J1939 addresses.

Vendor APIs that require no intervention retain all client identification and filter states and will verify ownership of all previously J1939-claimed addresses upon correction of the disconnect. This type of API does not require the recovery method described above.

Issue 2: Filter State Ambiguities

Differences exist among vehicle interface adapter manufacturers in their handling of "RP1210_SendCommand()" when using Command 4 - "Set Message Filtering for J1939," Command 5- "Set Message Filtering for CAN", and Command 7- "Set Message Filtering for J1708", after sending Command 3- "Set All Filter States To Pass". Some vendors interpret Command 3 as a temporary condition that is canceled automatically when issuing Commands 4, 5, and 7. Others interpret Command 3 as a permanent condition. The best way to deal with the differences is to issue Command 17- "Clear All Filter States" after the client application has completed looking at all messages (MIDs, PGNs, or CAN IDs) from the vehicle interface.

SUMMARY

If the deployment of a software application requires compatibility with all vehicle interface adapters, then the application must be designed based upon the lowest common denominator of adapter performance. If such compatibility is not required, then users should contact the selected vehicle interface adapter manufacturer to understand the performance and functional capabilities of their products.