DG Dearborn Group Technology

DEARBORN PROTOCOL ADAPTER FAMILY USER'S MANUAL

DPA II Plus / DPA III Plus (serial versions),

DPA III – ISA, DPA III – PC/104

Version 1.21

27007 Hills Tech Court
Farmington Hills, MI 48331
Phone (248) 488-2080 • Fax (248) 488-2082
http://www.dgtech.com

IMPORTANT NOTICE

The DPA is intended to be used as an evaluation tool only. Damage to the tool, if caused by misuse, is not covered under the seller's product warranty.

When using this manual, please remember the following:

- This manual may be changed, in whole or in part, without notice. Current updates to this manual may be found on Dearborn Group's web site at *http://www.dgtech.com*.

- Dearborn Group Inc., does not assume responsibility for any damage resulting from any accident—or for any other reason—while the DPA is in use.

- Examples of circuitry described herein are for illustration purposes only and do not necessarily represent the latest revisions of hardware or software. Dearborn Group Inc. assumes no responsibility for any intellectual property claims that may result from use of this material.

- No license is granted—by implication or otherwise–for any patents or any other rights of Dearborn Group Inc., or of any third party.

Table of Contents

Table of Figures

CHAPTER

1

1 INTRODUCTION

The Dearborn Protocol Adapter (DPA) product family is a group of tools used to interconnect serial communication networks and PCs (or other hosts). It is provided with a software library that is common to all DPA family products, to provide flexibility across all hardware platforms and networks. The DPA family software includes an RP1210A interface and a C library API for Windows. DPA hardware is available in the following formats.

Serial (RS-232) versions: The *DPA II Plus* and *DPA III Plus* are small packages providing a CAN (J1939) and J1708 interface to a serial connection. The RS-232 hardware supports a special "pass-through" mode which allows the DPA to emulate current interface products on the market. *The DPA III Plus* provides additional support for the J1850 protocol. For information specific to the DPA II Plus and DPA III Plus, consult Chapter 2.

ISA card and **PC/104 card**: The *DPA III* ISA card and PC/104 card support the CAN (J1939), J1708, and J1850 protocols. For information specific to the ISA and PC/104 cards, consult Chapter 3 and Chapter 4, respectively.

The DPA supports the following features:

- Platform independence for software development
- RS-232 interface port (serial versions)
- Support for CAN (11- and 29-bit Identifiers) and J1708 (modified RS-485) protocols; DPA III and III Plus versions also support J1850
- DPA API - DLL/VxD, RP1210A driver
- J1939/11-, J1939/15- and ISO-11898-compatible physical layer
- J1939/21 transport layer support

The DPA is not a stand-alone device. In normal operation, it is a slave to the PC, being told when and where to send and receive messages, as well as setting and resetting the timer and other functions. However, it is an asynchronous device that can inform the PC of datalink messages without being polled.

The DPA's hardware and PC layers and functions are described at length in the chapters that follow.

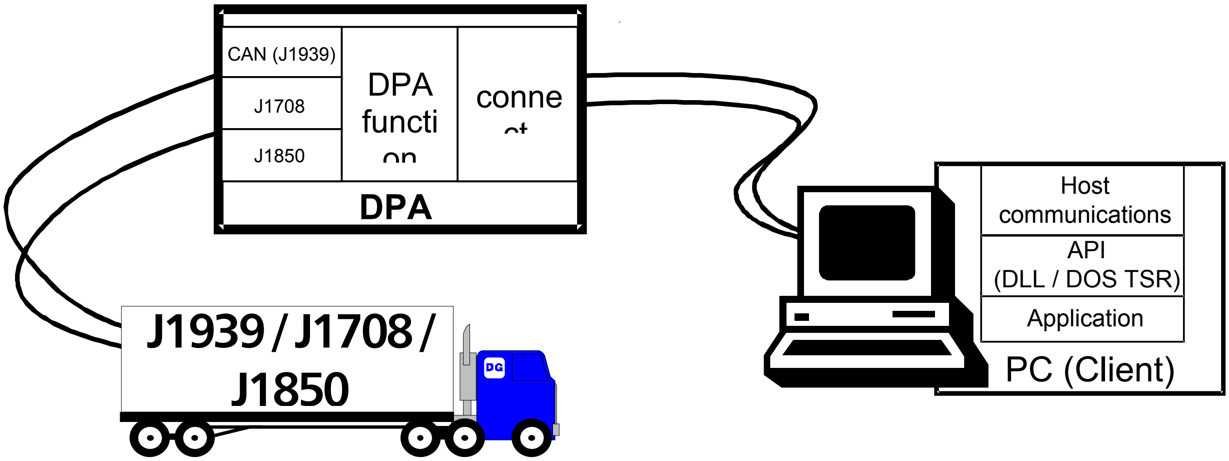


Figure 1: Functional overview

1.1 Documentation organization

This manual contains several chapters, an appendix, and an index. This chapter, **Chapter 1**, provides an overview of the manual and summarizes the contents of the remaining chapters and appendices. The remainder of this chapter provides reference to related documentation and technical support. The chapters that follow address these subjects:

Chapter 2 – RS-232 Hardware: Getting Started - Instructions for proper installation and setup of the DPA II Plus and DPA III Plus.

Chapter 3 – ISA Hardware: Getting Started - Instructions for proper installation and setup of the DPA III ISA card.

Chapter 4 – PC/104 Hardware: Getting Started - Instructions for proper installation and setup of the DPA III PC/104 card.

Chapter 5 – Functional Overview - Overview of main DPA functions.

Chapter 6 – API Overview – Introduction to the Dynamic Link Library (DLL) used in creating programs to interface with the DPA.

Appendix A – Defines and Structures

Appendix B – Filters (Masks) and CAN Bit-Timing Registers

Appendix C – Driver Summary

1.2 Technical support

☎ Call us for technical support

In the U.S., technical support representatives are available to answer your questions between 9 a.m. and 5 p.m. EST. You may also fax or e-mail your questions to us. Please include your voice telephone number, for prompt assistance. Non-U.S. users may want to contact their local representatives.

Phone:	(248) 488-2080
Fax:	(248) 488-2082
E-mail:	techsupp@dgtech.com
Web site:	http://www.dgtech.com

1.3 Related documents

The following product publications can be accessed through their respective authors, as indicated below.

Intel (800) 628-8686

87C196CA/87C196CB 20 MHz Advanced 16-Bit CHMOS Microcontroller with Integrated CAN 2.0 (data sheet)	Document #272405
82527 Serial Communications Controller Architectural Overview (data sheet)	Document #272410

Philips (800) 234-7381

PCA82C250 CAN Controller Interface data sheet	Data Book IC20

Society of Automotive Engineers (412) 776-4841

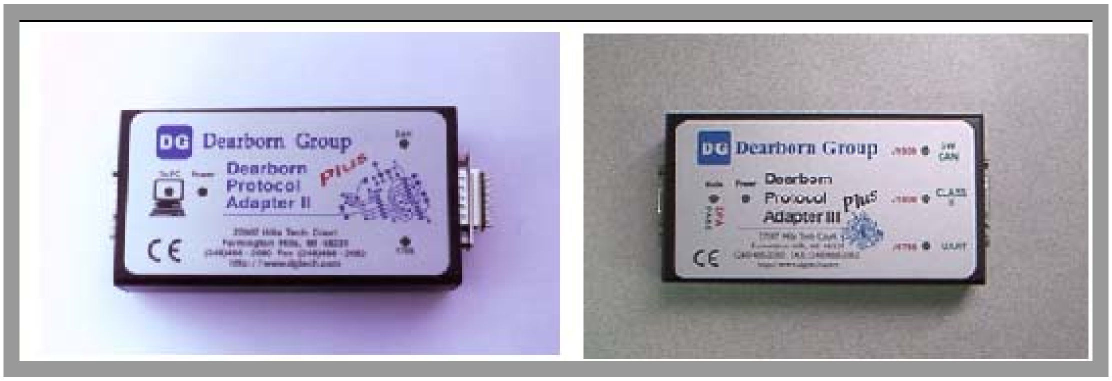
Recommended Practice for a Serial Control and Communication Vehicle Network	J1939
Recommended Practice	J1708

CHAPTER
2

2 RS-232 HARDWARE: GETTING STARTED

Before using your DPA II Plus or DPA III Plus serial (RS-232) hardware, please read this chapter. It describes the software, hardware, and settings necessary for successful installation and operation of your DPA unit It will also direct you to the appropriate appendices for setup information specific to your hardware unit.

IMPORTANT: please read this section before using the DPA!



2.1 Checking package contents

Your DPA package should include the following items.

- DPA hardware unit (RS-232)
- *DPA Family User's Manual* (this document)
- RS-232 straight-through cable
- 15-pin network connector
- Protocol Adapter Library API disk
- RP1210A driver disk

2.2 Software

The API library consists of one or more DLLs for Windows 3.1, 95, 98, NT, and Windows 2000.. Copy the required files to the appropriate directory on your hard drive, (typically the project directory where the application resides). The files should include a DLL, a library file, and a header file.

2.3 RS-232 hardware installation and setup

The DPA requires the following hardware for operation:

- AT-compatible computer (or higher)
- 9 - 32 volt @ 250mA power supply

The DPA serial unit uses an 80C196CA processor with a 16 MHz crystal and requires 9v - 32v at 250 ma of power. The operating temperature range is from 0 - 85° C.

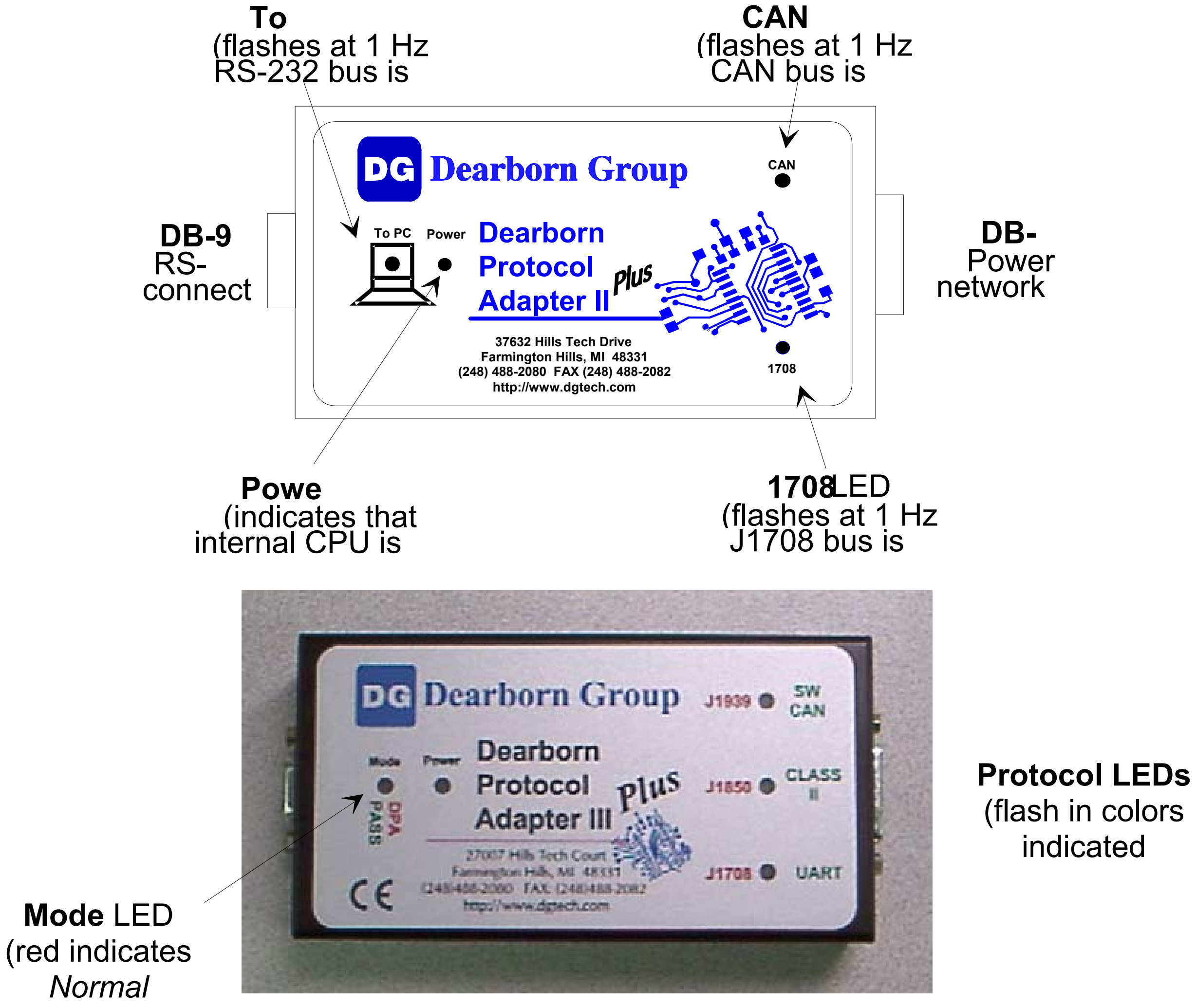


Figure 2 : DPA II Plus and DPA III Plus RS-232 hardware

7

To install the DPA II Plus or DPA III Plus, follow the following steps.

2.3.1 Power / network connection to the DPA II Plus

CAN media attachment

CAN connections (CANH and CANL) are created through the on-board Philips 82C250 transceiver. The CAN shield is also provided with an RC filter. A termination resistor of 120 ohms may be added. (See pinouts below.)

J1708 attachment

The J1708 interface is connected as specified by the SAE J1708 document. The connections are referenced as **ATA+** and **ATA-** on the pin configuration drawings below.

Power and network connections for the DPA are made through the female DB15 connector. There are two connector styles: **Original** (with no marking on the end plate) and **/T** (as labeled on the end plate).

2.3.1.1 Original connector

A termination resistor of 120 ohms may be added to the CAN link through the placement of a jumper between pins 7 and 8 on the connector, as follows:

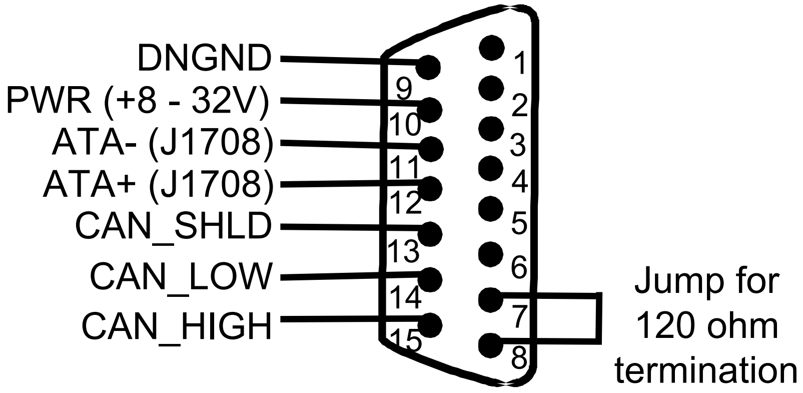


Figure 3: *Original* power and network connector

2.3.1.2 /T connector

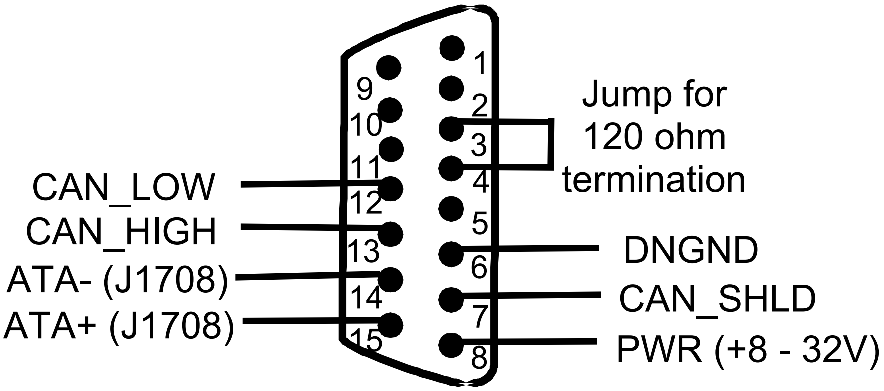


Figure 4: */T* power and network connector

2.3.2 Power / network connection to the DPA III Plus

CAN media attachment
CAN connections (CANH and CANL) are created through the on-board Philips 82C250 transceiver. The CAN shield is also provided with an RC filter. A termination resistor of 120 ohms may be added. (See pinouts below.)

J1708 attachment
The J1708 interface is connected as specified by the SAE J1708 document. The connections are referenced as **ATA+** and **ATA-** on the pin configuration drawings below.

J1850 and GM UART media attachment
The J1850 connections is made via the Harris HIP7010 J1850 controller and HIP7020 J1850 transceiver. The GM UART connections follow the GM ALDL standard.

2.3.2.1 DPA III /T connector

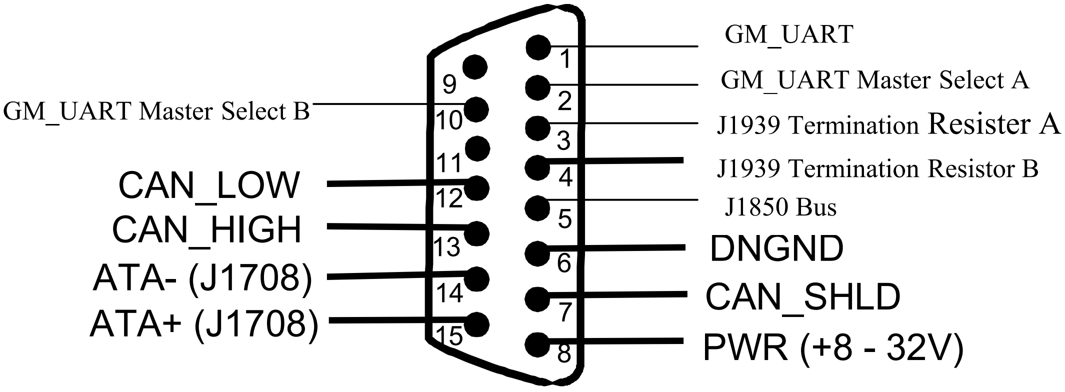


Figure 5: */IIIT* power and network connector

2.3.3 Connection to the PC

Once power is supplied to the DPA unit, the RS-232 DB-9 female connector (RS-232) needs to be joined to a PC serial port with the supplied straight-through DB9 cable. It is interfaced, pin-to-pin, with a standard nine-pin AT®-type serial connector:

Pin number	Host RS-232 signal name
1	No connection
2	RxD (Receive data)
3	TxD (Transmit data)
4	DTR (Data Terminal Ready)
5	SG (Signal Ground)
6	DSR (Data Set Ready)
7	RTS (Request to Send)
8	CTS (Clear to Send)
9	No connection

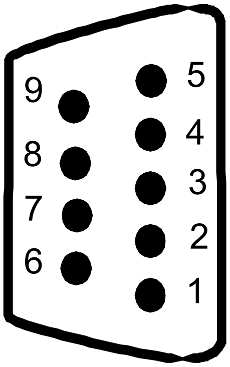


Figure 6: RS-232 DB9 connector pinout

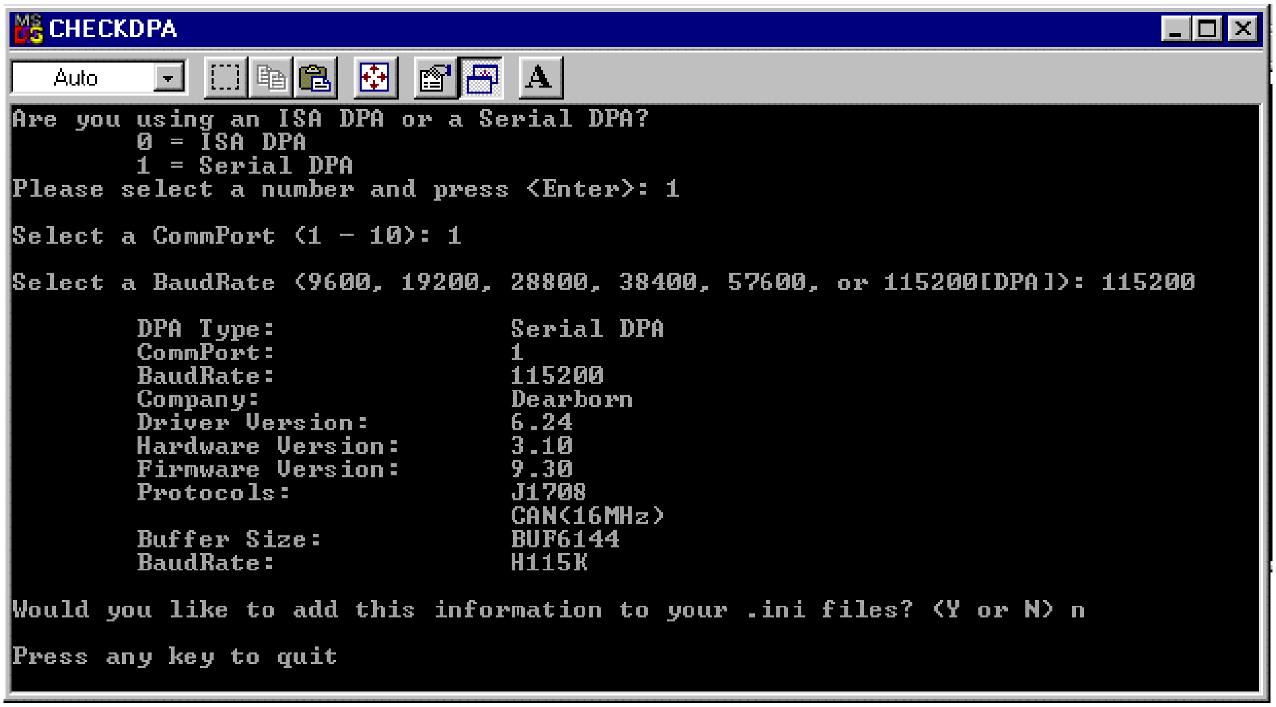
2.3.4 Driver installation

Install the DPA drivers provided on the disk labeled "Dearborn Group DPA Driver Installation," by inserting the disk into your PC's floppy drive and selecting Start | Run | A:\DPAINST.EXE. (If your floppy drive is not designated "A," then replace the "A" with the appropriate drive designation.) Follow the instructions on the screen.

This setup program will install driver files to the specified directory. There are currently six driver interfaces delivered with the DPA; consult the guidelines in Appendix C to determine which driver should be used.

2.3.5 Checking communication

Check to see that the DPA is communicating with the PC, by running the CHECKDPA.EXE utility once the DPA drivers have been installed (as described in section 2.3.4). This utility is installed along with the drivers and will prompt you for information about your DPA. Answer the questions presented, and if the DPA is working properly, the application will report information about the DPA.



```
Are you using an ISA DPA or a Serial DPA?
        0 = ISA DPA
        1 = Serial DPA
Please select a number and press <Enter>: 1

Select a CommPort (1 - 10): 1

Select a BaudRate (9600, 19200, 28800, 38400, 57600, or 115200[DPA]): 115200

        DPA Type:               Serial DPA
        CommPort:               1
        BaudRate:               115200
        Company:                Dearborn
        Driver Version:         6.24
        Hardware Version:       3.10
        Firmware Version:       9.30
        Protocols:              J1708
                                CAN(16MHz)
        Buffer Size:            BUF6144
        BaudRate:               H115K

Would you like to add this information to your .ini files? (Y or N) n

Press any key to quit
```

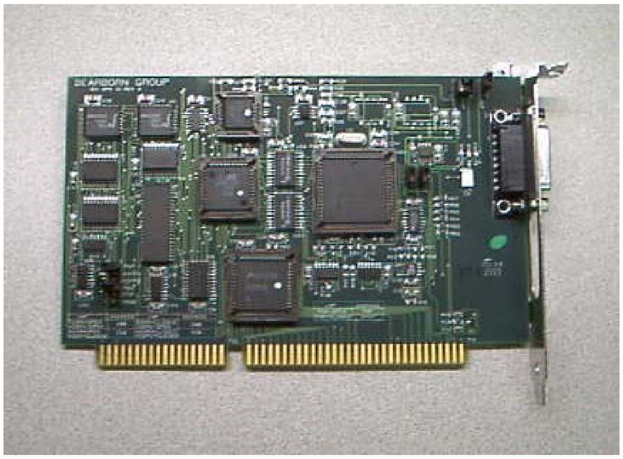
Example of successful communication using CHECKDPA

CHAPTER

3

3 ISA HARDWARE: GETTING STARTED

Before using your DPA III ISA hardware, please read this chapter. It describes the software, hardware, and settings necessary for successful installation and operation of your DPA unit It will also direct you to the appropriate appendices for setup information specific to your hardware unit.



3.1 Checking package contents

Your DPA package should include the following items.

- DPA hardware unit (ISA card)
- *DPA Family User's Manual* (this document)
- Installation and driver disks and manual
- Protocol Adapter Library API disk
- RP1210A driver disk

3.2 Software

The API library consists of one ore more DLLs for Windows 3.1, 95. 98, NT, and 2000. Copy the required files to the appropriate directory on your hard drive, (typically the project directory where the application resides). The files should include a DLL, a library file, and a header file. (RP1210A drivers are explained in an accompanying document.)

3.3 ISA card hardware installation and setup

The DPA requires the following hardware for operation:

- AT-compatible computer (or higher)
- 9 - 32 volt @ 250mA power supply

Before installing the DPA ISA card, it is important to check your PC's current configuration to identify an unused IRQ and address region. In Windows 95 or Windows 98, you may identify these regions using the device manager; in Windows NT, you may check the current configuration using the WINMSD.EXE program.

3.3.1 Setting the jumpers

The DPA III ISA card uses jumpers to set the resources (IRQ and Base Address) that the adapter will use. The DPA supports four base address choices and three IRQ selections, as defined in the following charts. (This information is also printed on the adapter itself, for simple configuration).

JP5	IRQ
	5 (default)
	7
	10

ADDR1	ADDR2	Base Address
O	O	0x200 (default)
O	C	0x220
C	O	0x300
C	C	0x320

Plug-and-play support is not yet available; leave the PNP jumper intact.

3.3.2 Power / network connection to the DPA III - ISA

The pinout for the ISA card's DB15 connector is as follows:

PIN	DESCRIPTION
1	GM_UART
2	GM_UART Master Select A[1]
3	J1939 Termination Resister A[2]
4	J1939 Termination Resister B[2]
5	J1850_BUS
6	Ground
7	CAN_SHIELD
8	Datalink Power (optional)
9	Datalink Ground (optional)
10	GM_UART Master Select B[1]
11	Reserved
12	CAN_LO
13	CAN_HI
14	J1708_ATA-
15	J1708_ATA+

[1] *To make this GM_UART node a master, jumper pin 2 to pin 10 in the connector.*
[2] *To terminate the CAN link with a 120 ohm resister, jumper pin 3 to pin 4 in the connector.*

3.3.3 Installing the ISA card

Once you have configured the resources and settings for your DPA, power down the PC. Locate an available ISA slot, and insert the adapter. Once the adapter is installed in the PC, you must install the appropriate drivers, as described in the following section.

3.3.4 Driver installation

Install the DPA drivers provided on the disk labeled "Dearborn Group DPA Driver Installation," by inserting the disk into your PC's floppy drive and selecting Start | Run | A:\DPAINST.EXE. (If your floppy drive is not designated "A," then replace the "A" with the appropriate drive designation.) Follow the instructions on the screen.

This setup program will install the virtual device driver for the DPA – ISA; it will also copy driver files to the specified directory. There are currently six driver interfaces delivered with the DPA; consult the guidelines in Appendix C to determine which driver should be used.

3.3.5 Checking communication

Check to see that the DPA is communicating with the PC, by running the CHECKDPA.EXE utility once the DPA drivers have been installed (as described in section 3.3.4). This utility is installed along with the drivers and will prompt you for information about your DPA. Answer the questions presented, and if the DPA is working properly, the application will report information about the DPA.

CHAPTER

4

4 PC/104 HARDWARE: GETTING STARTED

Before using your DPA III PC/104 hardware, please read this chapter. It describes the software, hardware, and settings necessary for successful installation and operation of your DPA unit It will also direct you to the appropriate appendices for setup information specific to your hardware unit.

4.1 Checking package contents

Your DPA package should include the following items.

- DPA hardware unit (PC/104 card)
- *DPA Family User's Manual* (this document)
- Installation and driver disks and manual
- Protocol Adapter Library API disk
- RP1210A driver disk

4.2 Software

The API library consists of one or more DLLs for Windows 3.1, 95, 98, NT, and 2000. Copy the required files to the appropriate directory on your hard drive, (typically the project directory where the application resides). The files should include a DLL, a library file, and a header file. (RP1210A drivers are explained in an accompanying document.)

4.3 ISA card hardware installation and setup

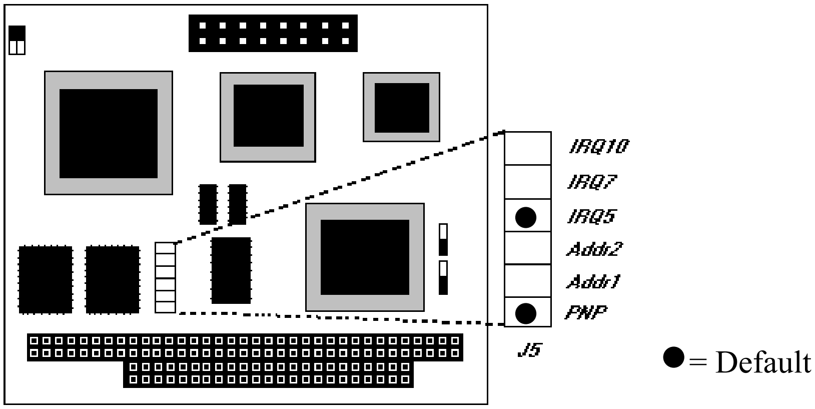
The DPA requires the following hardware for operation:

- AT-compatible computer (or higher)
- 9 - 32 volt @ 250mA power supply

Before installing the DPA PC/104 card, it is important to check your PC's current configuration to identify an unused IRQ and address region. In Windows 95 or Windows 98, you may identify these regions using the device manager; in Windows NT, you may check the current configuration using the WINMSD.EXE program.

4.3.1 Setting the jumpers

The DPA III PC/104 card uses jumpers to set the resources (IRQ and Base Address) that the adapter will use. The DPA supports four base address choices and three IRQ selections, as defined in the following charts.



JP5	IRQ
	5 (default)
	7
	10

ADDR1	ADDR2	Base Address
O	O	0x200 (default)
O	C	0x220
C	O	0x300
C	C	0x320

Plug-and-play support is not yet available; leave the PNP jumper intact.

4.3.2 Power / network connection to the DPA III – PC/104

The pinout for the PC/104 card's DB15 connector is as follows.

PIN	DESCRIPTION
1	GM_UART
2	GM_UART Master Select A[1]
3	J1939 Termination Resister A[2]
4	J1939 Termination Resister B[2]
5	J1850_BUS
6	Ground
7	CAN_SHIELD
8	Datalink Power (optional)
9	Datalink Ground (optional)
10	GM_UART Master Select B[1]
11	Reserved
12	CAN_LO
13	CAN_HI
14	J1708_ATA-
15	J1708_ATA+

[1] *To make this GM_UART node a master, jumper pin 2 to pin 10 in the connector.*
[2] *To terminate the CAN link with a 120 ohm resister, jumper pin 3 to pin 4 in the connector.*

4.3.3 Installing the PC/104 card

Once you have configured the resources and settings for your DPA, power down the PC. Locate an available PC/104 slot, and insert the adapter. Once the adapter is installed in the PC, you must install the appropriate drivers, as described in the following section.

4.3.4 Driver installation

Install the DPA drivers provided on the disk labeled "Dearborn Group DPA Driver Installation," by inserting the disk into your PC's floppy drive and selecting Start | Run | A:\DPAINST.EXE. (If your floppy drive is not designated "A," then replace the "A" with the appropriate drive designation.) Follow the instructions on the screen.

This setup program will install the virtual device driver for the DPA – PC/104; it will also copy driver files to the specified directory. There are currently six driver interfaces delivered with the DPA; consult the guidelines in Appendix C to determine which driver should be used.

4.3.5 Checking communication

Check to see that the DPA is communicating with the PC, by running the CHECKDPA.EXE utility once the DPA drivers have been installed (as described in section 4.3.4). This utility is installed along with the drivers and will prompt you for information about your DPA. Answer the questions presented, and if the DPA is working properly, the application will report information about the DPA.

CHAPTER
5

5 FUNCTIONAL OVERVIEW

The DPA embedded architecture is comprised of four main parts that manage the PC and network interfaces: a **Timer** (or *clock*), the **Tx Mailbox** (CAN, J1708, J1850), the **Rx Mailbox** (CAN, J1708, J1850), and an **IO Buffer** (or *Host Scratch Pad*). The following diagram shows the basic architecture.

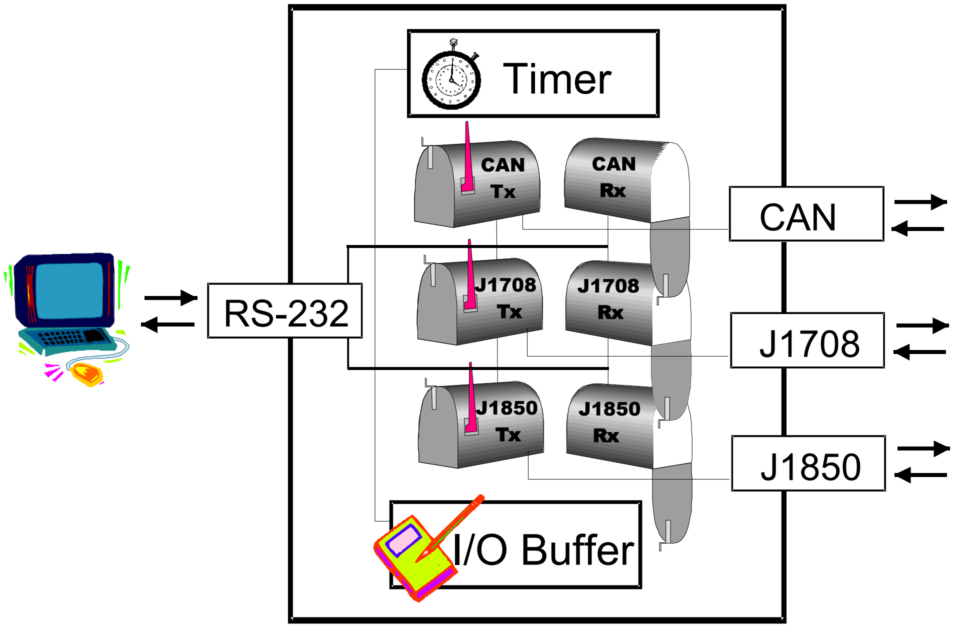


Figure 7: DPA architecture "map"

5.1 Timer

The timer is a free-running millisecond clock that runs 49.5 days before rolling over to zero. It is used to determine when a message is to be sent by transmit, to timestamp incoming messages with their respective times received, and to set timed interrupts to the PC.

The timer may be set to control the timing of outgoing broadcast messages and to set a base time for the timestamping of incoming messages. The PC commands used to control the timer allow the user to reset the timer, request synchronization with the DPA's internal timer, pause and resume timer function, and suspend timer interrupts.

Interrupt functions can be used to help process messages. The *EnableTimerInterrupts* function allows the operator to specify time intervals for interrupts from the DPA, while *DisableTimerInterrupts* suspends the timed interrupts. The suspension of timer interrupts allows transmits and callbacks to continue without interrupts to the PC; while the pausing of the timer suspends interrupts, transmits, and callbacks from the DPA hardware.

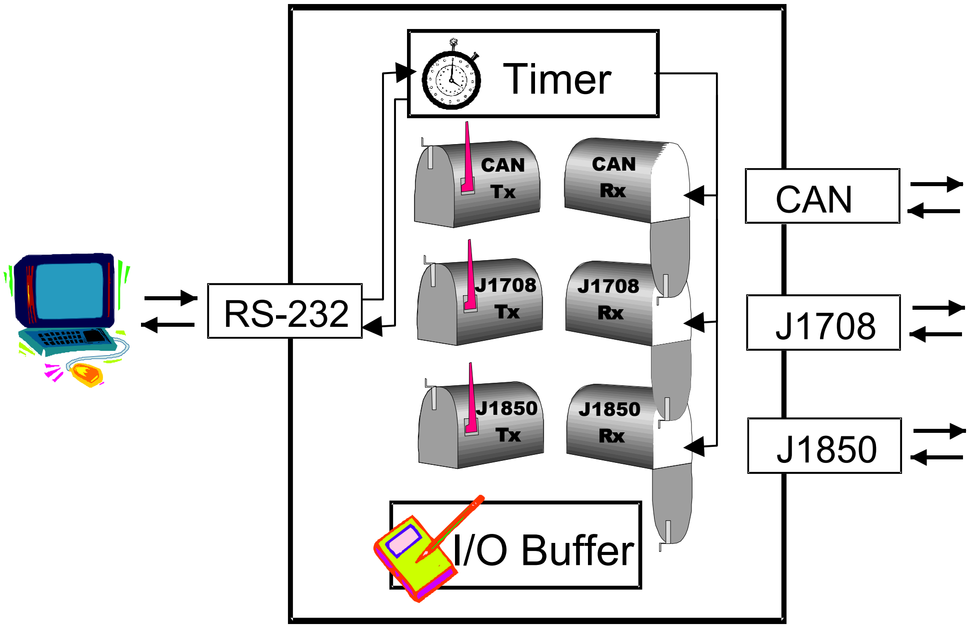


Figure 8: Timer "map"

5.2 Transmit Mailbox

A Transmit Mailbox, whether CAN, J1708, or J1850, is typically used to send messages over a network. The DPA allows the user to customize each message by specifying the following:

- When the CAN, J1708, or J1850 network message is to be sent
- When, relative to the DPA timer, message transmission is to begin
- The number of times the message is to be sent
- The desired time interval between transmissions
- The ID and data to be sent
- The conditions for a callback announcing a successful transmission
- The number of times the message should be sent before auto-deletion occurs
- Whether to enable a callback announcing the time of a message deletion

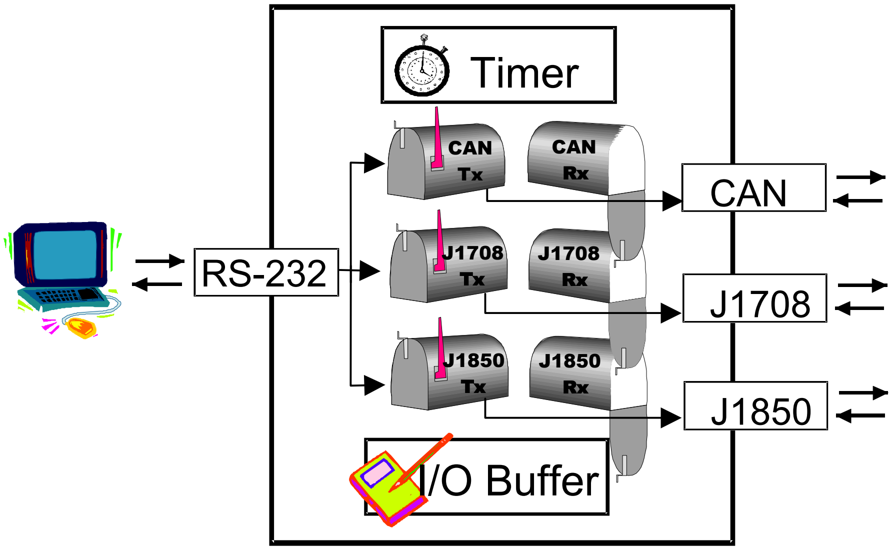


Figure 9: Transmit mailbox "map"

5.3 Receive Mailbox

A Receive (Rx) Mailbox (CAN, J1708, or J1850) is typically used to receive messages from a network. The options for receiving allow the user to specify the following:

- Which protocol to scan
- Which bits should be masked, and which ones should be matched, in hardware-level filtering
- What information (e.g., mailbox number, timestamp, identifier, length of data, and/or data) should be sent to the host immediately upon message receipt
- How the application will be notified when a message is received (Transparent Update, Receive Callbacks, Polling)

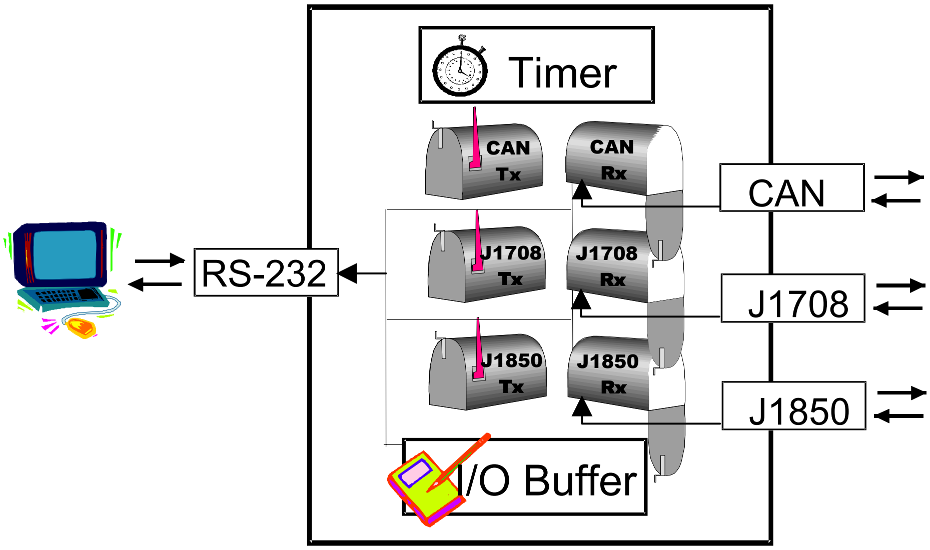


Figure 10: Receive mailbox "map"

5.4 I/O Buffer (host scratch pad)

The host scratch pad, or IO Buffer, is a space reserved in the DPA's memory; it is used for temporary storage of data for transmit or receive mailboxes. It adds flexibility to the transmitting and receiving of messages, regardless of network type (CAN, J1708, or J1850), by providing the following

- a temporary message storage location
- redirection of mailbox data
- storage for oversized messages (such as J1939 Transport Protocol messages)
- concatenation of small messages

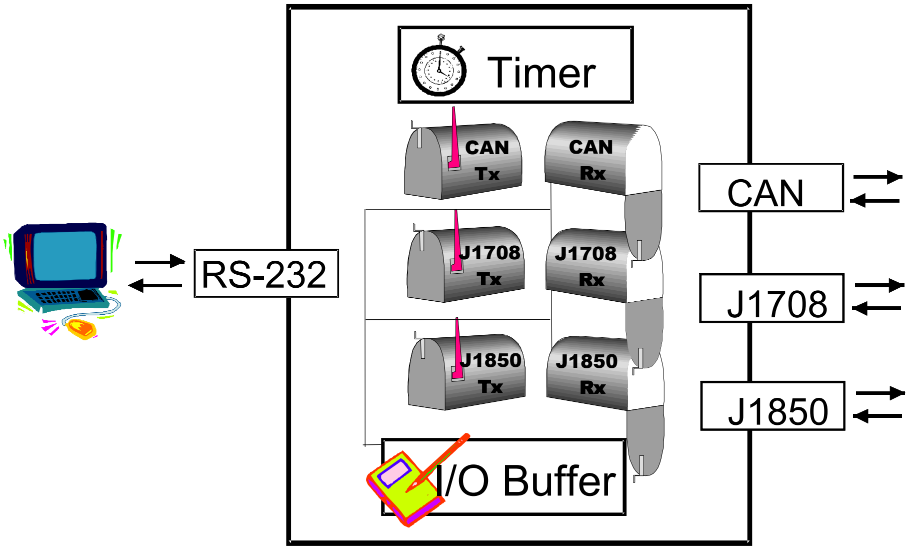


Figure 11: I/O buffer "map"

Oversized messages

J1708 and CAN networks sometimes transmit oversized messages. A normal J1708 message, for example, may be up to 21 bytes long; however, special modes may utilize longer messages. The DPA accommodates these oversized messages by putting the J1708 mailbox into extended mode and "attaching" it to a location in the I/O buffer (scratch pad).

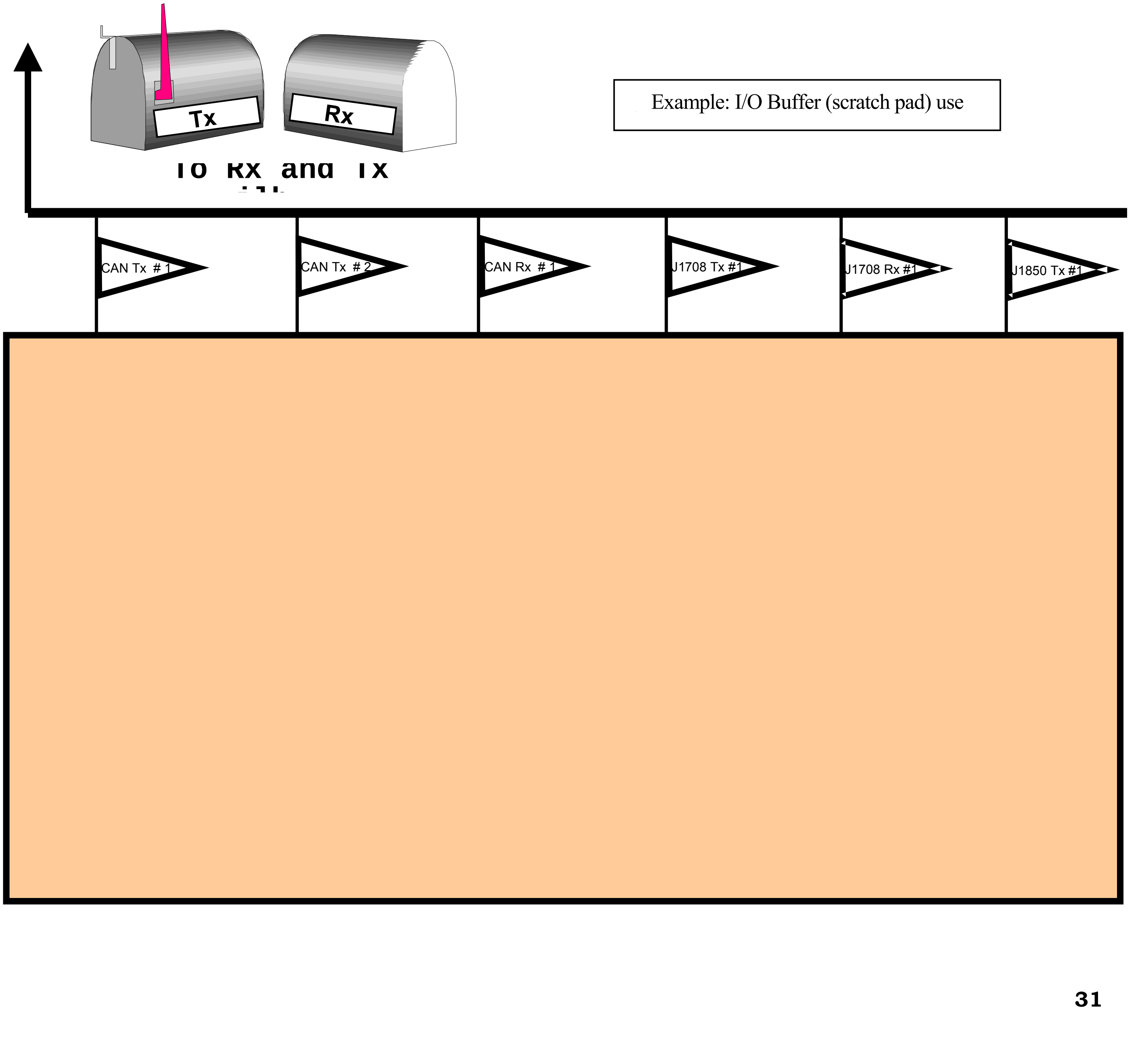
The J1939 transport layer also makes use of this buffer (scratch pad), to ensure that transport timing requirements are met. (Reference the *ConfigureTransportProtocol* function for further details.)

The MailBoxType structure must be set to extended mode in order to make use of the extended (or oversized) messages. This is accomplished through the setting of the following parameters:

```
bExtendedPtrMode = True
wExtendedOffset = Scratch pad address
```

Concatenated messages

The storing of multiple messages in the DPA's I/O buffer (scratch pad) reduces multiple reads and writes to the DPA hardware. The concatenation of these short messages, in turn, reduces the overhead on the serial port. The *LoadDPABuffer* function is used to re-assign mailboxes so that their data is stored in the scratch pad for concatenation:



CHAPTER

6

6 API OVERVIEW

The PC (client) software can be broken down in to two parts: the host communication level and the API (DLL). The host communication level is the lowest level of communication. The API structures and usage are described in Appendix A, the functions for the API are described in Chapter 6. If you need futher assistance with this level of programming please contact Dearborn Group's Technical Support staff for further assistance.

The Protocol Adapter Library API (Application Program Interface) was developed to provide a programming interface to the Protocol Adapter. The API has been developed as a linkable library and as a DLL/VxD (Dynamic Link Library/Virtual Device Driver) for Windows.

6.1 Compilers

Borland

The API was compiled using the Microsoft C++ compiler. For use with Borland, simply include the Borland import library included in the Borland directory, and call the functions as labeled in this manual. For use with older Borland compilers, you may be required to explicitly load the DLL and map the functions.

Microsoft

For use with Microsoft products, include the library in the directory with each DLL.

NOTE: You must change the project settings so that the *struct member alignment* value is one byte. To set this value, select **Project | Settings**, click the **C/C++** tab, select **Code Generation**, and specify *one byte* in the **Struct member alignment** box.

6.2 The API Choices

The DPA is delivered with two interface choices. There is a single DPA interface, and a multiple DPA interface.

The single DPA interface (DPA16.DLL, DPA32.DLL) allows communication to one DPA at a time in a given application. This interface is recommended for applications that have code that has been written for previous version of the DPA API. The single DPA interface is the only interface that is provided for DOS drivers.

The multiple DPA interface (DPAM16.DLL, DPAM32.DLL) allows an application to communicate to more than one DPA simultaneously. The Multiple DPA interface is very similar to the single DPA interface. The difference is that when an application opens a DPA, a DPA Handle is returned. This handle is passed to all future DPA calls to identify the DPA that the call is for. The Multiple DPA interface allows an application to communicate to multiple DPA's simultaneously. This interface is recommended for all new software development.

For additional help choosing the correct driver, refer to Appendix C - Choosing the Correct Driver for a Given Application.

6.3 The DPA API functions list

The Protocol Adapter Library API includes five function types: **system** functions**, data-link configuration** functions, **message handling** functions, **timer** functions, and **buffer** (host scratch pad) functions. These function names and functionality are the same for both the single DPA and the multiple DPA interfaces, but the parameter lists will differ slightly. A brief description of each function appears below, along with a reference to its corresponding detailed description in section 4.4 (where the functions appear alphabetically).

6.3.1 System functions

InitDPA
Specifies and initializes communication between the PC and the DPA. (See section 4.4.6 or 4.5.6)

InitCommLink
Specifies and initializes communication between the PC and the serial DPA. (See section 4.4.4. or 4.5.4)

InitPCCard
Specifies and initializes communication between the PC and the ISA/PC104 DPA. (See section 4.4.7 or 4.5.7)

RestoreDPA
Restores the communication port between the PC and the DPA II to its previous (pre-*InitDPA*) state. (See section 4.4.17 or 4.5.17)

RestoreCommLink
Restores the communication port between the PC and the DPA II (serial verion only) to its previous (pre-*InitCommLink*) state. (See section 4.4.16 or 4.5.16)

RestorePCCard
Restores the communication port between the PC and the DPA (ISA and PC104 versions only) to its previous (pre-*InitPCCard*) state. (See section 4.4.18 or 4.5.18)

CheckDataLink
Returns an identifier specifying the manufacturer name, the DLL version, the version of firmware installed on the DPA, and installed hardware capabilities. (See section 4.4.1. or 4.5.1)

ReadDPAChecksum
Verifies the checksum of the DPA's Flash memory. (See section 4.4.13 or 4.5.13)

ResetDPA
Performs a low-level reset of the DPA or its communications. (See section 4.4.15 or 4.5.15)

SetBaudRate (Serial only)
Allows the calling application to command the DPA to run at a different baud rate. (See section 4.4.21 or 4.5.21)

6.3.2 Data-link configuration functions

InitDataLink
Initializes communications with the Protocol Adapter and specifies which protocol is being implemented. (See section 4.4.5 or 4.5.5)

ConfigureTransportProtocol
Allows the user to configure J1939 Transport Protocol characteristics. (See section 4.4.2 or 4.5.2)

6.3.3 Message handling functions

LoadMailBox
Opens (creates) mailboxes for the receiving and transmitting of messages. (See section 4.4.9 or 4.5.9)

TransmitMailBox
Sends messages, using previously opened mailboxes. (See section 4.4.23 or 4.5.23)

TransmitMailBoxAsync
Sends messages asynchronously, using previously opened mailboxes from a function within a callback (ISR). (See section 4.4.24 or 4.5.24)

UpdateTransMailBoxData
Updates information in a previously opened broadcast mailbox. (See section 4.4.28 or 4.5.28)

UpdateTransMailBoxDataAsync
Updates information, from within a callback, in a broadcast mailbox. (See section 4.4.31 or 4.5.31)

UpdateTransmitMailBox
Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) from a previously opened transmit mailbox. (See section 4.4.28 or 4.5.28)

UpdateTransmitMailBoxAsync
(For use inside a callback routine.) Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) from a previously opened transmit mailbox. (See section 4.4.31 or 4.5.31)

ReceiveMailBox
Retrieves the latest message from a specific mailbox. (See section 4.4.14 or 4.5.14)

UpdateReceiveMailBox
Updates the data count, data location, identifier, and identifier mask from a previously opened receive mailbox. (See section 4.4.26 or 4.5.26)

UpdateReceiveMailBoxAsync
(For use inside a callback routine.) Updates the data count, data location, identifier, and identifier mask from a previously opened receive mailbox. (See section 4.4.27 or 4.5.27)

UnloadMailBox
Closes a previously opened mailbox. (See section 4.4.23 or 4.5.23)

6.3.4 Timer functions

LoadTimer
Sets the timer, for the timestamping of received and transmitted messages. (See section 4.4.10 or 4.5.10)

RequestTimerValue
Returns the current DPA timer value. (See section 4.4.19 or 4.5.19)

EnableTimerInterrupt
Enables a timer interrupt from the DPA, to call a user-supplied callback function. (See section 4.4.3 or 4.5.3)

SuspendTimerInterrupt
Disables a DPA timer interrupt. (See section 4.4.16.)

PauseTimer
Pauses the timer and suspends transmits and timer callbacks. (See section 4.4.11 or 4.5.11)

ResumeTimer
Resumes a previously paused timer function and re-starts all transmits and callback interrupts. (See section 4.4.20 or 4.5.20)

6.3.5 Buffer (host scratch pad) functions

LoadDPABuffer
Loads data into the DPA's I/O buffer (internal scratch memory). (See section 4.4.8 or 4.5.8)

ReadDPABuffer
Reads data from the DPA's I/O buffer (internal scratch memory). (See section 4.4.12 or 4.5.12)

6.4 Single DPA API function descriptions (alphabetical order)

6.4.1 CheckDataLink

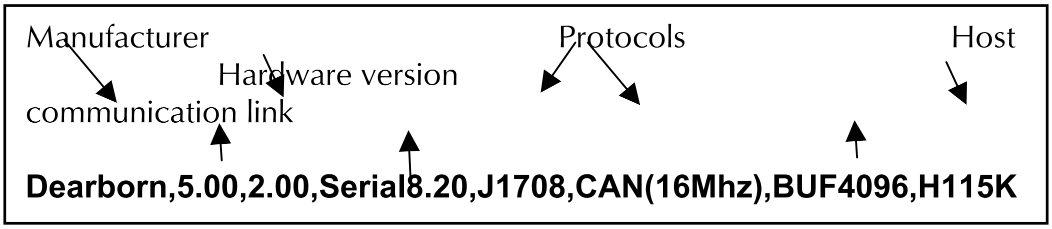
Function Returns an identifier specifying the manufacturer name, the DLL version, the version of firmware installed on the DPA, and installed hardware capabilities.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType CheckDataLink (char *szVersion) ;
```

Prototype In dpa16.h or dpa32.h

Remarks CheckDataLink returns an ASCII character string to **szVersion**. The response is a NULL terminated ASCII string using **,** as a delimiter and identifying the manufacturer, the software version, the hardware version, the firmware version, the protocol(s) available, the buffer size, and the host communication link.



Dearborn,5.00,2.00,Serial8.20,J1708,CAN(16Mhz),BUF4096,H115K

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

<u>Example</u>:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"

void DisplayCheckDataLink(void);

void main(void)
{
      CommLinkType            CommLinkData;
      ReturnStatusType  InitCommStatus;
      ReturnStatusType  RestoreCommStatus;

      CommLinkData.bCommPort        =eComm2;
      CommLinkData.bBaudRate        =eB115200;
      InitCommStatus = InitCommLink(&CommLinkData);

   DisplayCheckDataLink();

      RestoreCommStatus =
RestoreCommLink();
}
void DisplayCheckDataLink(void)
{
      char  szVersion[81];
      ReturnStatusType  CheckDataStatus;

      CheckDataStatus = CheckDataLink(szVersion);

      if( InitCommStatus == eNoError )
      {
            MessageBox( NULL, szVersion, "CheckDataLink", MB_OK );
      }
      else
      {
            MessageBox(NULL,
                        "DataLink is not responding.",
                        "CheckDataLink",
                        MB_OK);
      }
}
```

← If CommLink was successful, then check DataLink, i.e., if(

6.4.2 ConfigureTransportProtocol

Function Sets the timing and size parameters necessary for a J1939 Transport Protocol session.

Syntax `ReturnStatusTypeConfigureTransportProtocol (`**`ConfigureTransportType`**`*)`

Prototype In dpa16.h or dpa32.h

Remarks This routine allows configuration of the Transport Protocol either as a Broadcast Announce Message (BAM) or a Request-To-Send / Clear-To-Send (RTS/CTS) session. For BAM, the transmit time and receive timeouts must be set. For CTS/RTS, the timeouts, data times, and CTS size must be set. The following *LoadMailBoxType* parameters must be configured for each MailBox:

🕮 Reference the J1939 Specification for more information regarding BAM and RTS/CTS.

bTransportType	denotes session type: RTS/CTS or BAM
CTSSource	destination address for the RTS transport

The following *InitDataLink* parameters must also be enabled for the active Transport Protocol:

bProtocol = J1939
bParam2 = 3

All undefined values will default to the J1939 recommended value.

```
typedef struct{
  byte      bProtocol;
  word      iBamTimeOut;
  word      iBam_BAMTXTime;
  word      iBam_DataTXTime;
  word      iRTS_Retry;
  word      iRTS_RetryTransmitTime;
  word      iRTS_TX_Timeout;
  word      iRTS_TX_TransmitTime;
  word      iRTS_RX_TimeoutData;
  word      iRTS_RX_TimeoutCMD;
  word      iRTS_RX_CTS_Count;
  word      iRTS_TX_CTS_Count;
} ConfigureTransportType;
```

bProtocol The protocol selected for the mailbox. Must be eJ1939 - J1939

iBamTimeOut The BAM timeout, (1 bit = 10 mS): the maximum time allowed between messages before a timeout occurs and the connection is aborted.

iBam_BAMTXTime The maximum time allowed between the BAM message and the first data message transmitted, before a timeout occurs and the connection is closed. (1 bit = 1 mS)

iBam_DataTXTime The maximum time allowed between data messages in a BAM session. (1 bit = 10 mS)

i**RTS_Retry** The maximum number of times the DPA will send request-to-send packets without receiving a CTS

iRTS_RetryTransmitTime The time delay between RTS request messages. (1 bit = 1 mS)

iRTS_TX_Timeout The DPA timeout value for a unit waiting for a CTS after starting a data transmission. (1 bit = 10 mS)

iRTS_TX_TransmitTime The interval between data message transmissions.

iRTS_RX_TimeoutData The maximum amount of time the DPA will wait for a message before aborting a connection.

iRTS_RX_TimeoutCMD The timeout value for the interval between a CTS and first data packet

iRTS_RX_CTS_Count The number of packets to CTS when receiving messages from a sender. (J1939 Reference: PGN 60416, Control byte = 17, byte 2)

iRTS_TX_CTS_Count The message sender's number of messages to CTS. (J1939 Reference: PGN 60416, Control byte = 16, byte 5)

Return Value *ConfigureTransportProtocol* returns **eNoError** on success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError Success

eDeviceTimeout Unable to communicate with the DPA

eCommLinkNotInitialized Serial port not opened

Example:

This function can be used when it is necessary to implement the J1939 Transport Protocol. The Transport Protocol is only used by J1939. Shown below is a sample function calling the ConfigureTransportProtocol function.

```
void TestConfigureTransportProtocol(void)
{
      ConfigureTransportType  cth;

      cth.bProtocol                      =eJ1939;
      cth.iBamTimeout                    =1000L;
      cth.iBAM_BAMTXTime                  =1000L;
      cth.iBAM_DataTXTime                 =1000L;
      cth.iRTS_Retry                      =3;
      cth.iRTS_RetryTransmitTime          =10L;
      cth.iRTS_TX_Timeout                 =1000L;
      cth.iRTS_TX_TransmitTime            =100L;
      cth.iRTS_RX_TimeoutData             =5000L;
      cth.iRTS_RX_TimeoutCMD              =1000L;
      cth.iRTS_RX_CTS_Count               =2;
      cth.iRTS_TX_CTS_Count               =5;

      ConfigureTransportProtocol(&cth);
}
```

6.4.3 EnableTimerInterrupt

Function Enables a timer interrupt from the DPA. The interrupt initiates a user-supplied callback function.

Syntax

```
#include “dpa16.h” or “dpa32.h”
ReturnStatusType EnableTimerInterrupt(EnableTimerInterruptType
                              *pEnableTimerInterruptData) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *EnableTimerInterrupt* specifies an interrupt time interval and callback function. The current DPA timer value is passed to the callback function as a parameter.

```
typedef struct
{
unsigned long     dwTimeOut;
void (CALLBACK *pTimerFunction)(unsigned long);
} EnableTimerInterruptType;
```

dwTimeOut Specifies the period of the interrupt, in milliseconds.

(CALLBACK *pTimerFunction)(unsigned long) Address of callback routine for timer. (NULL disables this function.)

Return Value *EnableTimerInterrupt* returns ***eNoError*** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eTimeoutValueOutOfRange	A period greater than 1 minute (60,000 ms) was specified
eCommLinkNotInitialized	Serial port not opened

eSyncCommandNotAllowed Cannot call from within callback (ISR)

Example:

```
#include <windows.h>
#include “dpa32.h”

for long unsigned intdwTimeValue;

void CALLBACK TimerFunction(long unsigned int dwLocalTimerValue)
{
  dwTimeValue = dwLocalTimerValue;
}

void TestEnableTimerInterrupt  (void)
{
  ReturnStatusType EnableTimerStatus;
  long unsigned int      dwRequestedTime;

  /* enable 1 sec heart beat timer */
  EnableTimerInterruptData.dwTimeOut        = 1000L;
  EnableTimerInterruptData.pTimerFunction   = TimerFunction;
  (void)EnableTimerInterrupt (&EnableTimerInterruptData);

  /* turn off timer interrupt */
  SuspendTimerInterrupt();
}
```

6.4.4 InitCommLink

Function Specifies and initializes communication between the PC and the Protocol Adapter.

Syntax #include”dpa16.h” or “dpa32.h”
ReturnStatusType InitCommLink (CommLinkType *CommLinkData);

Prototype In dpa16.h or dpa32.h

Remarks *InitCommLink* is used to initialize communications between the PC and the DPA, by identifying the COM ports and baud rate. It also sets the flags so that all other calls will know which COM port to use. This function must be called before any other library functions are called. When *InitCommLink()* is called, the appropriate interrupt vector pointers are saved so that they can be restored using *RestoreCommLink ()*.

The *CommLinkType* structure is as follows:

```
typedef struct
{
unsigned char  bCommPort;
unsigned char  bBaudRate;
} CommLinkType;
```

bCommPort The serial communication port on the PC:
eComm1 – COM1
eComm2 – COM2
eComm3 – COM3
eComm4 – COM4
eComm5 – COM5
eComm6 – COM6
eComm7 – COM7
eComm8 – COM8
eComm9 – COM9

bBaudRate The serial communication baud rate. The DPA only supports 115K baud upon initialization. If you have a comm port that is capable of faster communication, the DPA can be set to communicate at 230K baud using the SetBaudRate() command. (32-bit only)
eB115200 = 115200 baud

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eSerialIncorrectPort	Incorrect Com port
eSerialIncorrectBaud	Incorrect Baudrate
eSerialPortNotFound	Com port not found
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes COM2 to communicate with the DPA at 115K baud.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
     CommLin kType       CommLinkData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;

     CommLinkData.bCommPort        =eComm2;
     CommLinkData.bBaudRate        =eB115200;
     InitCommStatus = InitCommLink(&CommLinkData);

     RestoreCommStatus = RestoreCommLink();
}
```

6.4.5 InitDataLink

Function Initializes the specified vehicle data link with data-link information; empties all mailboxes associated with the specified data link and identifies the protocol being implemented.

Syntax

```
#include “dpa16.h”or “dpa32.h”
ReturnStatusType InitDataLink (InitDataLinkType*pInitDataLinkData) ;
```

Prototype In dpa16.h or dpa32.h

Remarks This function initializes the hardware for specified protocols. The DPA II hardware currently supports J1939 and J1708 protocols. And the DPA III hardware currently support J1939, J1850 and J1708. It allows for setting of the baud rate (for CAN and J1708), and of *Extended* (29-bit) or *Standard* (11-bit) CAN identifier priority. The hardware supports both 11- and 29- bit identifiers for CAN; however, the hardware can only double-buffer one CAN message type, so selection is left to the application.

Three callback functions (for *DataLinkError*, *TransmitVector* and *ReceiveVector*) are provided, (see below).

```
typedef struct
{
  unsigned char         bProtocol;
  unsigned char         bParam0;
  unsigned char         bParam1;
  unsigned char         bParam2;
  unsigned char         bParam3;
  unsigned char         bParam4;
  void (CALLBACK *pfDataLinkError)(MailBoxType *,
                   DataLinkErrorType *);
  void (CALLBACK *pfTransmitVector)(MailBoxType *);
  void (CALLBACK *pfReceiveVector)(void);
} InitDataLinkType;
```

bProtocol - the network protocol types
eISO9141 - ISO9141 protocol
eJ1708 - J1708 protocol
eJ1939 - J1939 protocol
eCAN - CAN protocol

bParam0-bParam4 - specific parameters for the various protocol types, described in the following section.

For CAN:

bParam0 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 0. (Reference Appendix B.1.)
bParam1 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 1. (Reference Appendix B.1.)
bParam2 is used to give preferred status to 29-bit or 11-bit CAN identifiers.
0x00 = 29 bit identifier has preferred status
0x01 = 11 bit identifier has preferred status
0x03 = 29 bit with J1939 transport protocol
bParam3 is not used for J1939
bParam4 is not used for J1939

In order for the DPA hardware to utilize the Single-Wire CAN physical layer, you will need to set bit 4 of Param2 when initializing the Data Link. This setting switches the transceiver being used by the hardware. All other CAN functions are unchanged. Please note that the transceiver for the Single-Wire CAN physical layer has a maximum baud rate of 100 Kbaud.

Example

This example initializes the DPA to Single-Wire CAN, 29 bit identifiers, at 33.33 Kbaud.
InitCAN (0x4E, 0x58, 0x10)

This example initializes the DPA to Single-Wire CAN. 11 bit identifiers, at 33.33 Kbaud.
InitCAN (0x4E, 0x58, 0x11)

For J1708:

bParam0 is used to set the baudrate used for the J1708 link
0x00 - 9600 baud
0x01 - 19.2K baud
0x02 – 10.4K baud
bParam1 is not used for J1708
bParam2 sets the use of automatic checksum creation in J1708:
0 = Full automatic checksum for transmit and receive
1 = Automatic checksum for receive only
2 = Automatic checksum for transmit only
3 = no automatic checksum

bParam3 is not used for J1708
bParam4 is not used for J1708

For pass-through mode (DPA II *Plus* only):

bProtocol = eJ1708
bParam0 = 0x80
bParam1 is used to set CD (Carrier Detect)
bParam2 is used to set DSR (Data Terminal Ready)
bParam3 is used to set RTS (Request to Send) value
bParam4 is used to set RI (Ring Indicator) value

The above parameters are configured by the following bit configuration (**0** = *Off*, **1** = *On*) for the byte:

Priority 0000 = 0 (End of Message)
0001 = Priority 1
0010 = Priority 2
0011 = Priority 3
0100 = Priority 4
0101 = Priority 5
0110 = Priority 6
0111 = Priority 7
1000 = Priority 8

(CALLBACK *pfDataLinkError)(MailBoxType *, **DataLinkErrorType** *) Calls a routine from the address pointed by *pfDataLinkError* if a DPA error occurs. (NULL disables this function.) *MailBoxType* definition is located in Appendix A.3.

DataLinkErrorType Structure returns any errors during data link initialization:

```
typedef struct
{
unsigned char  bProtocol;
unsigned char  bErrorCode;
} DataLinkErrorType;
```

bProtocol Network protocol type (same as previous values)
bErrorCode Error code.

CALLBACK *pfTransmitVector)(**MailBoxType** *) Calls a routine pointed at by *pfTransmitVector* when a messages is transmitted. (NULL disables this callback.) *MailBoxType* definition is located in Appendix A.3. Transmit callbacks are also avalable for each individual mailbox.

(**CALLBACK *pfReceiveVector**) Calls a routine pointed at by *pfReceiveVector* when a message is received. (NULL disables this callback.) Receive callbacks are also avalable for each individual mailbox.

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eProtocolNotSupported	Invalid protocol specified
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example initializes the CAN Datalink to 250K bps 29-bit identifier preferred.

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"

void DisplayInitDataLink(void);

void main(void)
{
       CommLinkType  CommLinkData;
       ReturnStatusType    InitCommStatus;
       ReturnStatusType    RestoreCommStatus;

       CommLinkData.bCommPort     =eComm2;
       CommLinkData.bBaudRate     =eB115200;
       InitCommStatus = InitCommLink(&CommLinkData);

       DisplayInitDataLink();

       printf("This example works!!! \n");
       RestoreCommStatus = RestoreCommLink();
}
```

```
← If CommLink was
successful, then check
DataLink, i.e.,
 if(
InitCommStatus==eNoError)
```

```
void DisplayInitDataLink(void)
{
       InitDataLinkType    InitDataLinkData;
       ReturnStatusType    InitDataLinkStatus;

       InitDataLinkData.bProtocol        = eJ1939;    // Select protocol
       InitDataLinkData.pfDataLinkError  = NULL;      // No Error Callback
       InitDataLinkData.pfTransmitVector = NULL;      // No CAN TX Callback
       InitDataLinkData.pfReceiveVector  = NULL;      // No CAN Receive Callback
       InitDataLinkData.bParam0          = 0x41;      // Set Baud
       InitDataLinkData.bParam1         = 0x58;
       InitDataLinkData.bParam2         = 0x00;             //  29-bit preferred
       InitDataLinkStatus               = InitDataLink( &InitDataLinkData);

       /* process the returned status */
       switch(InitDataLinkStatus)
       {
             case eNoError:
             {
                    MessageBox(NULL,
                    "DataLink successfully initialized.",
                    InitDataLink,MB_OK);
                    break;
             }

             case eDeviceTimeout:
             {
                    MessageBox(NULL,
                    "DataLink not responding.",
                    InitDataLink,MB_OK);
                    break;
             }

             case eProtocolNotSupported:
             {
                    MessageBox(NULL,
                       "Requested protocol is not supported",
                    InitDataLink,MB_OK);
                    break;
             }

             case eSyncCommandNotAllowed:
             {
                    MessageBox(NULL,
                    "Cannot call from within a callback (ISR) routine.",
                    InitDataLink,MB_OK);
                    break;
             }
       }
}
```

Example #2:

Initializing the J1708 Datalink to 9600 baud.

```
#include <windows.h>
#include “dpa32.h”

void DisplayInitDataLink  (void)
{
   InitDataLinkType            InitDataLinkData;
   ReturnStatusType            InitDataLinkStatus;

  /*  send Init DataLink command  */
  memset(&InitDataLinkData, 0, sizeof(InitDataLinkType));
  InitDataLinkData.bProtocol         = eJ1708;
  InitDataLinkData.bParam0           = 0x00;
  InitDataLinkData.pfDataLinkError   = NULL;
  InitDataLinkData.pfTransmitVector  = NULL;
  InitDataLinkData.pfReceiveVector   = NULL;
  InitDataLinkStatus                 = InitDataLink(&InitDataLinkData);

  /* process returned status */
  switch  (InitDataLinkStatus)
  {
    case eNoError:
    {
       MessageBox  (NULL,
            “DataLink successfully initialized”,
            “InitDataLink”, MB_OK);
       break;
    }
    case eDeviceTimeout:
    {
       MessageBox  (NULL,
            “DataLink not responding”,
            “InitDataLink”, MB_OK);
       break;
    }
    case eProtocolNotSupported:
    {
       MessageBox  (NULL,
            “Requested protocol not supported”,
            “InitDataLink”, MB_OK);
       break;
    }
    case eSyncCommandNotAllowed:
    {
       MessageBox  (NULL,
            “Cannot call from within a callback (ISR) routine”,
            “InitDataLink”, MB_OK);
       break;
    }
  }
}
```

6.4.6 InitDPA (Windows Only)

Function Specifies and initializes communication between the PC and the DPA.

Syntax #include "dpa16.h" or "dpa32.h"
ReturnStatusType InitDPA (short DPANumber);

Prototype In dpa16.h or dpa32.h

Remarks *InitDPA* is used to initialize communications between the PC and the DPA, by identifying the number of the dpa. This function will read the parameters of the DPA from the DG1210.INI or DG121032.INI. The entry for the DPA number 1 in the INI file is listed below.

```
[DeviceInformation1]
DeviceID=1
DeviceDescription=DPAII,COM1
DeviceName=Dearborn Protocol Adapter II
DeviceParams= COM1,0x3F8,4,B115200,INTERRUPT
```

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eSerialIncorrectPort	Incorrect Com port
eSerialIncorrectBaud	Incorrect Baudrate
eSerialPortNotFound	Com port not found
eInvalidINI	The INI record is invalid
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes DPA number 1 for communications.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
      ReturnStatusType InitCommStatus;
      ReturnStatusType      RestoreCommStatus;

      Status = InitDPA(1);

      RestoreCommStatus = RestoreDPA();
}
```

6.4.7 InitPCCard

Function Specifies and initializes communication between the PC and the ISA/PC104 DPA.

Syntax #include"dpa16" or "dpa32.h"
ReturnStatusType InitPCCard (PCCardType *PCCardData);

Prototype In datalink.h

Remarks *InitPCCard* is used to initialize communications between the PC and the DPA, by identifying the base address and the IRQ. It also sets the flags so that all other calls will know which DPA to use. This function must be called before any other library functions are called. When *InitPCCard()* is called, the appropriate interrupt vector pointers are saved so that they can be restored using *RestoreCommLink ()*.

The *CommLinkType* structure is as follows:

```
typedef struct
{
unsigned short bBaseAddress;
unsigned char  bIrq;
} PCCardType;
```

bBaseAddress The base address of the PC Card.
0x200
0x220
0x300
0x320

bIrq The IRQ of the PC Card
5
7
10

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eIrqConflict	Irq in use
eIncorrectDriver	The driver is the wrong driver
eInvalidDriverSetup	The driver is set up incorrectly
eInvalidBaseAddress	The base address is invalid
eInvalidDll	There is a bad or missing DLL
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes port 200 to communicate with the DPA on IRQ 7.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
     PCCardType          PCCardData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;

     PCCardData.bBaseAddress  =0x200;
     PCCardData.bIrq          =7;
     InitCommStatus = InitPCCard(&PCCardData);

     RestoreCommStatus = RestorePCCard();
}
```

6.4.8 LoadDPABuffer

Function Loads data into the DPA's internal buffer (scratch pad).

Syntax

```
ReturnStatusType LoadDPABuffer (unsigned char *bData,
unsigned int wLength, unsigned int wOffset);
```

Prototype In dpa16.h or dpa32.h

Remarks *LoadDPABuffer* points to the current data location, assigns a length value for the data, and indicates an offset for data placement. There are limits on the amount of data that can be passed across the serial link in a single call. These limits will differ between 16-bit and 32-bit applications.

bData	Pointer to the data to be loaded into the buffer
wLength	Number of bytes to transfer
wOffset	Buffer address to which writing should start

Return Value *LoadDPABuffer* returns **eNoError** on successful transmission.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened

Example:

This example loads data into the IO buffer at address 21.

```
void WriteAppID( unsigned char *ucAppID,  int iLength )
           {
   (void)LoadDPABuffer ( ucAppID, iLength, 21 );
           }
```

6.4.9 LoadMailbox

Function Opens or creates mailboxes for receiving and transmitting messages.

This function is used to create mailboxes for one of the following functions:

- Transmitting messages
- Broadcasting messages
- Receiving messages

When a **Transmit** mailbox is opened, the host application determines:

- When the message is to be sent.
- How many times it is to be sent (Broadcast Count).
- The time intervals between consecutive transmits (Broadcast Time).
- Whether to automatically delete the mailbox after all messages are sent.
- Whether the host is to be notified when the message is sent (RX CallBack).
- Which ID (CAN) or MID/PID (J1708) is to be sent.
- What data should be sent.

When a **Receive** mailbox is opened, the host application determines the following:

- Which ID (CAN) or MID/PID (J1708) bits should be masked, and which ones should be matched, in hardware-level filtering.
- What information is needed when the message is received.
- Whether the host is to be notified when the message is sent (TX CallBack).

Syntax

```
#include “dpa16.h” or “dpa32.h”
ReturnStatusTypeLoadMailBox(LoadMailBoxType*pLoadMailBoxData);
```

Prototype In dpa16.h or dpa32.h

Remarks *LoadMailBox* opens (creates) mailboxes for the receiving and transmitting of messages, according to the structures presented in section A.3. There are four types of mailboxes:

- Transmit (*Broadcast*, used to continuously transmit the same message)
- Transmit (*Release,* used to broadcast messages or send one message then unload the mailbox)
- Transmit (*Resident*, used to send messages on command or a given number of times while leaving the mailbox to be used again or updated)
- Receive

You should initialize all structure variables to zero before using the structure shown in the following example.

Example (in C):

```
LoadMailBox MyMBox;
memset(&MyMBox,0,sizeof(LoadMailBox));
```

(The *LoadMailBoxType* structure is defined in Appendix A.3.)

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eProtocolNotSupported	Invalid protocol specified
eInvalidBitIdentSize	Invalid identifier size (11 or 29 are the only valid numbers for the CAN protocol)
eInvalidDataCount	J1939 protocol only supports 8 bytes
eMailBoxNotAvailable	All available mailboxes (16 for receiving, 16 for transmitting) are in use
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example illustrates the creation of a Receive mailbox using transparent updating.

```
#include <windows.h>
#include "dpa32.h"

unsigned char     TransUpdateDataBuffer[8];

void TestLoadMailBox  (void)
{
  LoadMailBoxType        LoadMailBoxData;
  ReturnStatusType       LoadMailBoxStatus,
                                     UnloadMailBoxStatus;
  MailBoxType            *TransUpdateHandle  = NULL;

/* load data for mailbox */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                      = eJ1939;
   LoadMailBoxData.bRemote_Data                   = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize                  = 29;
   LoadMailBoxData.bTransportType                 = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent                 = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask             = 0x00000000L;
   LoadMailBoxData.bTransparentUpdateEnable       = TRUE;
   LoadMailBoxData.bTimeStampInhibit              = FALSE;
   LoadMailBoxData.bIDInhibit                     = FALSE;
   LoadMailBoxData.bDataCount                     = 8;
   LoadMailBoxData.pfApplicationRoutine           = NULL;
   LoadMailBoxData.vpData                         = TransUpdateDataBuffer;
   LoadMailBoxStatus                              =  LoadMailBox
   if (LoadMailBoxStatus == eNoError)                  (&LoadMailBoxData);

   {
     TransUpdateHandle  = LoadMailBoxData.pMailBoxHandle;     //save the point-
   }                                                          //er  to  the  new
                                                              //mailbox for fu-
   /* unload mailbox now that we are done */                  //ture  loading
   UnloadMailBoxStatus  = UnloadMailBox (TransUpdateHandle);
   TransUpdateHandle  = NULL;
  }
```

Example #2:

This example illustrates the creation of a Receive mailbox using a callback routine.

```
#include <windows.h>
#include "dpa32.h"

unsigned char           bCallBackBuffer[8];

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
/* Copy data */
memcpy (Handle->vpData, Handle->bData, Handle->bDataCount);
}

void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         UnloadMailBoxStatus;
   MailBoxType           *CallBackHandle = NULL;

/* load Receive MailBox for callback */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask       = 0x00L;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.pfApplicationRoutine     = CallBackRoutine;
   LoadMailBoxData.vpData                   = bCallBackBuffer;
   LoadMailBoxStatus            = LoadMailBox(&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     CallBackHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (CallBackHandle);
   CallBackHandle = NULL;
}
```

Example #3:

*This example illustrates the creation of a mailbox used to receive messages on request only, using the **ReceiveMailBox** function.*

```
#include <windows.h>
#include "dpa32.h"

void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         ReceiveMailBoxStatus,
                         UnloadMailBoxStatus;
   unsigned char         bRequestBuffer[8];
   MailBoxType           *RequestHandle = NULL;

/* load Receive MailBox for request */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask       = 0x0000ffffL;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.pfApplicationRoutine     = NULL;
   LoadMailBoxData.vpData                   = bRequestBuffer;
   LoadMailBoxStatus         = LoadMailBox(&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     RequestHandle = LoadMailBoxData.pMailBoxHandle;
   }

   if (RequestHandle != NULL)
   {
     ReceiveMailBoxStatus = ReceiveMailBox (RequestHandle);
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (RequestHandle);
   RequestHandle = NULL;
 }
```

Example #4:

*This example illustrates the creating of a mailbox used to transmit on command using only the **TransmitMailBox** function.*

```
#include <windows.h>
#include "dpa32.h"

void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         TransmitStatus,
                         UnloadMailBoxStatus;
   unsigned char               bTransmitData[8],
                               index;
   MailBoxType                 *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease       = eResident;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.bTimeAbsolute            = FALSE;
   LoadMailBoxData.dwTimeStamp              = 0x00L;
   LoadMailBoxData.dwBroadcastTime          = 0x00L;
   LoadMailBoxData.iBroadcastCount          = 0;
   LoadMailBoxData.pfApplicationRoutine     = NULL;
   LoadMailBoxData.vpData                   = bTransmitData;
   LoadMailBoxStatus                        = LoadMailBox (&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* if mailbox was succesfully created then transmit data 4 times */
   if (TransmitHandle != NULL)
   {
     for (index=0; index<4; index++)
     {
       TransmitStatus = TransmitMailBox (TransmitHandle);
     }
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (TransmitHandle);
   TransmitHandle = NULL;
 }
```

Example #5:

*This example illustrates the creation of a mailbox used to transmit once when created, and then on command, using the **TransmitMailBox** function.*

(*TransmitMailBox* can be used an unlimited number of times.)

```
#include <windows.h>
#include "dpa32.h"

void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         TransmitStatus,
                         UnloadMailBoxStatus;
   unsigned char         bTransmitData[8],
                         index;
   MailBoxType           *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol            = eJ1939;
   LoadMailBoxData.bRemote_Data         = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease   = eResident;
   LoadMailBoxData.bBitIdentSize        = 29;
   LoadMailBoxData.bTransportType       = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent       = 0x00001234L;
   LoadMailBoxData.bDataCount           = 8;
   LoadMailBoxData.bTimeAbsolute        = FALSE;
   LoadMailBoxData.dwTimeStamp          = 0x00L;
   LoadMailBoxData.dwBroadcastTime      = 0x00L;
   LoadMailBoxData.iBroadcastCount      = 1
   LoadMailBoxData.pfApplicationRoutine = NULL;
   LoadMailBoxData.vpData               = bTransmitData;
   LoadMailBoxStatus                    = LoadMailBox (&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* if mailbox was succesfully created then transmit data 4 times */
   if (TransmitHandle != NULL)
   {
     for (index=0; index<4; index++)
     {
       TransmitStatus = TransmitMailBox (TransmitHandle);
     }
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (TransmitHandle);
   TransmitHandle = NULL;
```

Example #6:

This example illustrates the creation of a mailbox used to broadcast a message 100 times every 100 ms. This example also illustrates the use of ***UpdateTransMailBoxData*** *for the updating of data being broadcast.*

```
#include <windows.h>
#include "dpa32.h"

void TestLoadMailBox  (void)
{
   LoadMailBoxType        LoadMailBoxData;
   ReturnStatusType       LoadMailBoxStatus,
                          UpdateStatus,
                          UnloadMailBoxStatus;
   unsigned char          bTransmitData[8],
                          bNewData[8],
                          index;
   MailBoxType            *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease       = eRelease;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.bTimeAbsolute            = FALSE;
   LoadMailBoxData.dwTimeStamp              = 0x00L;
   LoadMailBoxData.dwBroadcastTime          = 100L;
   LoadMailBoxData.iBroadcastCount          = 1000L;
   LoadMailBoxData.pfApplicationRoutine     = NULL;
   LoadMailBoxData.vpData                   = bTransmitData;
   LoadMailBoxStatus                        = LoadMailBox (&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     /* get handle of mailbox if successful */
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;

     /* update data */
     TransmitHandle->vpData = bNewData;
     UpdateStatus = UpdateTransMailBoxData (TransmitHandle);
   }
   /* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (TransmitHandle);
   TransmitHandle = NULL;
 }
```

Example #7:

This example illustrates the creation of a mailbox used to transmit a transport message. This example may differ depending on wether you are using the 32-bit of 16-bit drivers. The InitDataLink() call must have been called with the transport layer enabled.

```
#include <windows.h>
#include "dpa32.h"

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
    /* Message Transmitted – Perform required processing here */

}
void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus;
   unsigned char         bTransmitData[1000],
                         index;
   MailBoxType           *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol              = eJ1939;
   LoadMailBoxData.bRemote_Data           = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease     = eRelease;
   LoadMailBoxData.bBitIdentSize          = 29;
   LoadMailBoxData.dwMailBoxIdent         = 0x00001234L;
   LoadMailBoxData.bDataCount             = 1000; //1 to 1785 bytes
   LoadMailBoxData.bTimeAbsolute          = FALSE;
   LoadMailBoxData.dwTimeStamp            = 0x00L;
   LoadMailBoxData.dwBroadcastTime        = 0L;
   LoadMailBoxData.iBroadcastCount        = 1L;
   LoadMailBoxData.pfApplicationRoutine   = CallBackRoutine;
   LoadMailBoxData.vpData                 = bTransmitData;

   //The following elements are for transport.  Extended pointer mode must
   //be used anytime that the data is longer than 8 bytes.  The extended
   //offset is the offset into the dpa buffer where the data will be
   //stored.  In 16-bit, it may be necessary to use the LoadDPABuffer
   //command to load the data into the buffer before you call the load
   //mailbox command.  If you use this method, set the bDataInhibit flag
   //to TRUE, telling the driver that the data is already in the buffer.
   //When using transport protocol, a transmit callback should be used to
   //inform the application when the transport protocol session has
   //completed.

   LoadMailBoxData.bDataInhibit           = TRUE;
   LoadMailBoxData.bTransportType         = eTransportRTS;
   LoadMailBoxData.bExtendedPtrMode       = TRUE;
   LoadMailBoxData.wExtendedOffset        = 0; //Offset into the DPA Buffer
   LoadMailBoxData.bCTSSource             = DestinationAddress;
   LoadMailBoxStatus = LoadMailBox (&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     /* get handle of mailbox if successful */
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }
}
```

Example #8:

This example illustrates the creation of a Receive mailbox for transport protocol using a callback routine.

```
#include <windows.h>
#include "dpam32.h"

unsigned char           bCallBackBuffer[8];

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
   /* Copy data */
   memcpy (Handle->vpData, Handle->bData, Handle->bDataCount);
}

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         UnloadMailBoxStatus;
   MailBoxType           *CallBackHandle = NULL;

/* load Receive MailBox for callback */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask       = 0x00L;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 1785; //largest message size
   LoadMailBoxData.pfApplicationRoutine     = CallBackRoutine;
   LoadMailBoxData.vpData                   = bCallBackBuffer;

   //The following elements are for transport.  Extended pointer mode must
   //be used anytime that the data is longer than 8 bytes.  The extended
   //offset is the offset into the dpa buffer where the data will be
   //stored.  In 16-bit, it may be necessary to use the ReadDPABuffer
   //command to read the data from the buffer after a receive callback is
   //received.  If you use this method, set the bDataInhibit flag to TRUE,
   //telling the driver that the data will be read later from the buffer.
   //The CTSSource element is the address of the node that will be sending
   //the CTS messages of the transport session.  For receive, this is the
   //address of the DPA on the link.

   LoadMailBoxData.bDataInhibit           = FALSE;
   LoadMailBoxData.bTransportType         = eTransportRTS;
   LoadMailBoxData.bExtendedPtrMode       = TRUE;
   LoadMailBoxData.wExtendedOffset        = 1785; //Offset in the Buffer
   LoadMailBoxData.bCTSSource             = OurLocalAddress;
   LoadMailBoxStatus            = LoadMailBox(&LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     CallBackHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (CallBackHandle);
   CallBackHandle = NULL;
}
```

6.4.10 LoadTimer

Function Sets the timer, for the timestamping of received and transmitted messages.

Syntax

```
#include "dpa16.h" or "dpa32.h"
ReturnStatusType  LoadTimer  (long unsigned int dwTime) ;
```

Prototype dpa16.h or dpa32.h

Remarks *LoadTimer* sets the timer used for the timestamping of received and transmitted messages. It takes the parameter passed in **dwTime** and loads it in the DPA II's timer. The timer has a 1 mS resolution, and count wrapping occurs about every 49 days.

Return Value *LoadTimer* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

*To use this function, you can create a function such as the one below, or just place the **LoadTimer** call in directly into your code.*

```
void GoLoadTimer(void)
{
     ReturnStatusType LoadTimerStatus;

     // Initialize the timer with a value of 1000ms
     LoadTimerStatus = LoadTimer(1000L);
}
```

6.4.11 PauseTimer

Function Pauses the DPA II's internal timer, which suspends all transmits and callbacks (interrupts).

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType  PauseTimer;
```

Prototype dpa16.h or dpa32.h

Remarks *PauseTimer* stops the timer, along with all transmits and callback interrupts.

Return Value *LoadTimer* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use the Pause Timer function, simply make the following call in your program:

```
     PauseTimer();
```

6.4.12 ReadDPABuffer

Function Reads data from the DPA's I/O buffer (internal scratch memory).

Syntax

```
ReturnStatusType ReadDPABuffer (unsigned char *bData,
unsigned int wLength, unsigned int wOffset);
```

Prototype In dpa16.h or dpa32.h

Remarks This routine allows the host to read data out of the I/O Buffer. That data may be loaded into the DPA by another application or via any attached mailbox. Due to linitations in 16-bit operating systems, tt may be necessary to break the reading of large buffers into multiple calls to ReadDPABuffer.

bData	pointer to the location for the data
wLength	number of data bytes to be read
wOffset	address in the buffer where reading should begin

Return Value *ReadDPABuffer* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened

Example:

*This example demonstrates the reading of data stored at the buffer's **Address 21**.*

```
void ReadAppIDIOBuffer( unsigned char *AppID, int iLength)
{
   (void)ReadDPABuffer (ucAppID, iLength, 21);
}
```

6.4.13 ReadDPAChecksum

Function Returns the value of the DPA checksum, for software verification of changes or corruption.

Syntax

```
#include "dpa16.h" or "dpa32.h"
ReturnStatusType ReadDPAChecksum (unsigned int *varname) ;
```

Prototype In dpa16.h or dpa32.h

Remarks This function checks the DPA checksum value, for any corruption of the DPA Flash memory. (The memory can become corrupted when the Flash memory fades, is improperly programmed, or is physically damaged.) *ReadDPAChecksum* returns the checksum and puts it in **varname**, a user-defined variable.

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA

Example:

*The following code demonstrates one example of how the **ReadDPAChecksum** function may be used. The variable **uiVer707Checksum** can represent anything you wish.*

```
void TestFlash(void)
{
unsigned int uiChecksum;

status = DPAReadDPAChecksum(&uiChecksum);

if (status == eNoError)
    {
    if uiChecksum == uiVer707Checksum)
        {
        // do whatever here
        }
    }
}
```

6.4.14 ReceiveMailBox

Function Retrieves the latest message from a specific mailbox.

Syntax

```
#include "dpa16.h" or "dpa32.h"
ReturnStatusType ReceiveMailBox (MailBoxType *pMailBoxHandle) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *ReceiveMailBox* retrieves the latest message from a specific mailbox. Previous messages are overwritten.

The **MailBoxType** structure is found in Appendix A.3.

pMailBoxHandle is a pointer to the Mailbox's unique name or handle.

Return Value *ReceiveMailBox* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eInvalidMailBox	Attempting to receive with unopened mailbox
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

<u>Example #1</u>:

*Retrieves data last received in a CAN mailbox specified by the mailbox handle **J1939ReceiveHandle**.*

```
if ((J1939ReceiveHandle != NULL) &&
                        (J1939ReceiveHandle->bActive != 0))
          {
             (void)ReceiveMailBox (J1939ReceiveHandle);
          }
```

<u>Example #2</u>:

*Retrieves data last received in a J1708 mailbox specified by the mailbox handle **J1708ReceiveHandle**.*

```
         if ((J1708ReceiveHandle != NULL) &&
          (J1708ReceiveHandle->bActive != 0))

     {

       (void)ReceiveMailBox (J1708ReceiveHandle);

}
```

6.4.15 ResetDPA

Function Resets the DPA based upon the parameter passed.

Syntax

```
#include"dpa16.h"or "dpa32.h"
ReturnStatusType ResetDPA (ResetType  *ResetData);
```

Prototype In dpa16.h or dpa32.h

Remarks *ResetDPA is* used to command a low-level reset of the DPA. It can be used to perform a full reset of the hardware, or a communications-only reset.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
     CommLinkType        CommLinkData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;
     ReturnStatusType    ResetStatus;
     ResetType           ResetData;

     CommLinkData.bCommPort        =eComm2;
     CommLinkData.bBaudRate        =eB115200;
     InitCommStatus = InitCommLink(&CommLinkData);
     ResetData.bResetType = eFullReset;
     ResetStatus = ResetDPA (ResetData);
     RestoreCommStatus = RestoreCommLink();
}
```

6.4.16 RestoreCommLink

Function Restores the communication port between the PC and the DPA II to its previous (pre-*InitCommLink*) state.

Syntax

```
#include"dpa16.h"or "dpa32.h"
ReturnStatusType RestoreCommLink (void);
```

Prototype In dpa16.h or dpa32.h

Remarks *RestoreCommLink* is used to restore any and all interrupt vectors that were set up during *InitCommLink()*. Once *RestoreCommLink ()* is called, no other library functions (except *InitCommLink ()*) can be called.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
     CommLinkType        CommLinkData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;

     CommLinkData.bCommPort        =eComm2;
     CommLinkData.bBaudRate        =eB115200;
     InitCommStatus = InitCommLink(&CommLinkData);

     RestoreCommStatus = RestoreCommLink();
}
```

6.4.17 RestoreDPA (Windows only)

Function Restores the communication port between the PC and the DPA II to its previous (pre-InitDPA) state.

Syntax

```
#include"dpa16.h" or "dpa32.h"
ReturnStatusType RestoreDPA (void);
```

Prototype In dpa16.h or dpa32.h

Remarks *RestoreDPA* is used to restore any and all interrupt vectors that were set up during *InitDPA().* Once *RestoreDPA()* is called, no other library functions (except *InitDPA ()*) can be called.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"



void main(void)
{
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;

     InitCommStatus = InitDPA(1);

     RestoreCommStatus = RestoreDPA();
}
```

6.4.18 RestorePCCard

Function Restores the communication port between the PC and the DPA II to its previous (pre-*InitPCCard*) state.

Syntax

```
#include"dpa16.h"or "dpa32.h"
ReturnStatusType RestorePCCard (void);
```

Prototype In dpa16.h or dpa32.h

Remarks *RestorePCCard* is used to restore any and all interrupt vectors that were set up during *InitPCCard().* Once *RestorePCCard ()* is called, no other library functions (except *InitPCCard ()*) can be called.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpa32.h"

void main(void)
{
     PCCardType          PCCardData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;

     PCCardData.bBaseAddress       =0x200;
     PCCardData.bIrq               =7;
     InitCommStatus = InitPCCard(&PCCardData);

     ReturnStatusType = RestorePCCard();
}
```

6.4.19 RequestTimerValue

Function Returns the current DPA timer value.

Syntax

```
#include "dpa16.h" or "dpa32.h"
ReturnStatusType   RequestTimerValue     (long   unsigned   int
*dwTimerValue) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *RequestTimerValue* returns the current DPA timer value. The timer value is placed into **dwTimerValue**.

Return Value *RequestTimerValue* returns **eNoError** upon success.
In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include <windows.h>
#include "dpa32.h"

void TestRequestTimerValue  (void)
{
  ReturnStatusType LoadTimerStatus,
             RequestTimerValueStatus;
  long unsigned int      dwRequestedTime;
  char             szBuffer [81];

  /* load timer with 1000, 1 second */
  LoadTimerStatus = LoadTimer (1000L);

  /* request timer value and display */
  RequestTimerValueStatus = RequestTimerValue (&dwRequestedTime);
  vsprintf  (szBuffer,
       "Current DataLink timer value is: %ld",
       dwRequestedTime);
  MessageBox  (NULL, szBuffer, "RequestTimerValue", MB_OK);
}
```

6.4.20 ResumeTimer

Function Resumes a previously paused timer function and re-starts all transmits and callback interrupts.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType  ResumeTimer;
```

Prototype dpa16.h or dpa32.h

Remarks *ResumeTimer* restarts the DPA II's internal timer previously stopped by the *PauseTimer* function. Transmit and callback interrupt functions are also resumed.

Return Value *LoadTimer* returns **eNoError** on success. In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use the Resume TImer funtion, simply make the following call in your program:

```
ResumeTimer();
```

6.4.21 SetBaudRate (32-bit Windows only)

Function Send a command to change to baud rate on the serial link between the DPA and the PC.

Syntax

```
ReturnStatusType  SetBaudRate (BaudRateType *baudRate);
```

Prototype dpa16.h or dpa32.h

Remarks SetBaudRate performs 3 steps. Sets the DPA baud to the requested value, sets the PC baud to the requested value, and verifies that the DPA and the PC can still communicate. Baud rates higher than 115200 are only available on serial ports that will support these baud rates.

dpaHandle	denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.
Return Value	*SetBaudRate* returns **eNoError** on success. In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eBaudRateNotSupported	Invalid baud rate passed in
eDeviceTimeout	Could not command DPA to change baud
eCommVerificationFailed	Could not communicate after change
eSerialOuputError	Could not set PC baud

Example:

To use the SetBaudRate funtion, simply make the following call in your program:

```
SetBaudType    baudRates;
BaudRates.bPCBaud = eb230400;
BaudRates.bDPABaud = eb230400;
SetBaudrate(baudRates);
```

6.4.22 SuspendTimerInterrupt

Function Disables a DPA timer interrupt.

Syntax

```
#include “dpa16.h” or “dpa32.h”
ReturnStatusType SuspendTimerInterrupt  (void) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *SuspendTimerInterrupt* disables a DPA timer interrupt.

Return Value *SuspendTimerInterrupt* returns **eNoError** upon success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use this function, simply make the following call in your program.

```
     SuspendTimerInterrupt();
```

6.4.23 TransmitMailBox

Function Sends messages, using previously opened mailboxes.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType   TransmitMailBox   (MailBoxType
*pMailBoxHandle) ;
```

Prototype In dpa16.h or dpa32.h

Remarks This function is used to transmit already loaded mailboxes. Once established, data can only be updated with the *UpdateTransMailBoxData* function before the *TransmitMailBox* function is re-called.

MailBoxType (Structure defined in Appendix A.3.)

***pMailBoxHandle** A pointer to the Mailbox's unique name or handle.

Return Value *TransmitMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eInvalidMailBox	Attempting to transmit an unopened mailbox
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example sends the specified mailbox once, with no modification to any of the mailbox parameters.

```
(void)TransmitMailBox (TransmitCANHandle);
```

6.4.24 TransmitMailBoxAsync

Function Sends messages asynchronously, using previously opened mailboxes, from a function within a callback (ISR).

Syntax

```
#include“dpa16.h”or “dpa32.h”
ReturnStatusType TransmitMailBoxAsync (MailBoxType *pMailBoxHandle);
```

Prototype In dpa16.h or dpa32.h

Remarks This function is identical to *TransmitMailBox*, except for its use of the DPA's *Non-Verbose* mode for turning off the DPA response inside the application interrupt.

TransmitMailBoxAsync is used to update Broadcast mailbox callback information, or to send messages using resident mailboxes.

The *MailBoxType* structure can be found in Appendix A.

Return Value *TransmitMailBoxAsync* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eInvalidMailBox	Attempting to transmit an unopened mailbox
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example illustrates the transmission (in response to a receive callback) of a previously loaded transmit mailbox.

```
void ReceiveCallBack ()
          {
TransmitMailBoxAsync(TransmitCANHandle);
           }
```

6.4.25 UnloadMailbox

Function Closes a previously opened mailbox.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType UnloadMailBox (MailBoxType *MailBoxHandle) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *UnloadMailBox* closes a mailbox and disables all related callback functions.

The *MailBoxType* structure can be found in Appendix A.

Return Value *UnloadMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example illustrates the unloading of the mailbox specified by the mailbox handle ***TransmitCANHandle***.

```
   (void)UnloadMailBox (TransmitCANHandle);
```

6.4.26 UpdateReceiveMailBox

Function Updates the data count, data location, identifier, and identifier mask of an open receive mailbox.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType    UpdateReceiveMailBox    (MailBoxType
*pMailBoxHandle, unsigned char bUpdateFlag) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *ReceiveMailBox* retrieves the latest message for a specific mailbox. Previous messages are overwritten.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *ReceiveMailBox* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:
*To use the **UpdateReceiveMailBox** function, you can simply add the following lines to your code or even create a function of your own. The **bUpdateFlag** variable is not mentioned in the description but is a necessary parameter.*

```
MailBoxType       *J1708ReceiveHandle;
ReturnStatusType  UpdateReceiveStatus;
unsigned char     bUpdateFlag;

UpdateReceiveStatus = UpdateReceiveMailBox(&J1708ReceiveHandle,
                      bUpdateFlag);
```

6.4.27 UpdateReceiveMailBoxAsync

Function Updates the data count, data location, identifier, and identifier mask of an open receive mailbox.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType      ReceiveMailBox      (MailBoxType
*pMailBoxHandle, unsigned char bUpdateFlag) ;
```

Prototype In dpa16.h or dpa32.h

Remarks *ReceiveMailBox* retrieves only the information which flags were set, (in the *MailBoxType* structure), to receive. There is no return value for *ReceiveMailBox*.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *ReceiveMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

*To use the **UpdateReceiveMailBoxAsync** function, you can simply add the following code to your program (in this case, for receiving J1708 messages):*

```
MailBoxType    *J1708ReceiveAsyncHandle;
ReturnStatusType UpdateReceiveAsyncStatus;
unsigned char bUpdateFlagAsync;

UpdateReceiveAsyncStatus = UpdateReceiveMailBoxAsync(

     &J1708ReceiveAsyncHandle,

     bUpdateFlagAsync);
```

6.4.28 UpdateTransMailBoxData

Function Updates information being sent from a previously opened transmit mailbox.

Syntax

```
#include "dpa16.h"or "dpa32.h"
ReturnStatusType UpdateTransMailBoxData (MailBoxType *pMailBoxHandle);
```

Prototype In dpa16.h or dpa32.h

Remarks Updates only the data being sent from a previously opened mailbox.

MailBoxType (Structure found in section A.3)

pMailBoxHandle Pointer to the Mailbox's unique name (handle)

Return Value *UpdateTransMailBoxData* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

You should include the following defines in your code:

```
MailBoxType *TransmitJ1708Handle;         // Declare the Mail Box Handle
unsigned char  TransmitMailBoxBroadcastData[9];
```

6.4.29 UpdateTransMailBoxDataAsync

Function Updates information being sent from a previously opened transmit mailbox., for use inside a callback routine.

Syntax

```
#include"dpa16.h"or "dpa32.h"
ReturnStatusTypeUpdate TransMailBoxDataAsync (MailBoxType
                                                    *pMailBoxHandle);
```

Prototype In dpa16.h or dpa32.h

Remarks *UpdateTransMailBoxDataAsync* is used to update only the data being sent from a previously opened mailbox, for use inside a callback routine.

MailBoxType (Structure found in section A.3)

pMailBoxHandle Pointer to the Mailbox's unique name (handle)

Return Value *UpdateTransMailBoxDataAsync* returns **eNoError** on success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This is an example of how to update and retransmit data, in a transmit callback routine.

```
void TransmitCallBack ()
          {
            strcpy ((char *)TransmitMailBoxBroadcastData, "12345678");
            (void)UpdateTransMailBoxData(TransmitCANHandle);
          }
```

6.4.30 UpdateTransmitMailBox

Function Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) from a previously opened transmit mailbox.

Syntax `ReturnStatusType UpdateTransmitMailBox (MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);`

Prototype In dpa16.h or dpa32**.**h

Remarks *UpdateTransmitMailBox* allows the application to change any of the parameters included inside a transmit mailbox.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *UpdateTransMailBoxData* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example illustrates an update of the data inside a transmit mailbox.

```
{
            unsigned char ucUpdateData[9];
            unsigned char bUpdateFlag;

              strcpy ((char *)UpdateData, "12345678");
   TransmitCANHandle->vpData                   = UpdateData;
            bUpdateFlag |= UPDATE_DATA;

            (void)UpdateTransmitMailBox (TransmitCANHandle, bUpdateFlag);

           }
```

Example #2:

This example shows the updating of the Broadcast count to zero, in essence terminating broadcast.

```
void  ShutOffTransmit( MailBoxType    *TransmitCANHandle)

{
            unsigned char ucUpdateData[9];
            unsigned char bUpdateFlag;

   TransmitCANHandle->iBroadcastCount         = 0;
   bUpdateFlag |= UPDATE_BROADCAST_COUNT;

   (void)UpdateTransmitMailBox (TransmitCANHandle, bUpdateFlag);
}
```

Example #3:

This example updates both the broadcast time and the broadcast count, causing the specified mailbox to be broadcast the specified number of times, at 100-millisecond intervals.

```
ReturnStatusType BroadCastCount100ms( MailBoxType    *TransmitCANHandle,
                                             signed int iBroadCastCount )
```

```
{

              // Update Broadcast Time
   TransmitCANHandle->dwBroadcastTime         = 100;
    bUpdateFlag |= UPDATE_BROADCAST_TIME;

   // Load Broad Cast Count
   TransmitCANHandle->iBroadcastCount         = iBroadCastCount;
    bUpdateFlag |= UPDATE_BROADCAST_COUNT;

   // Send Command to DPA and return response
   return (UpdateTransmitMailBox (TransmitCANHandle, bUpdateFlag) );

}
```

6.4.31 UpdateTransmitMailBoxAsync

Function (For use inside a Callback routine.) Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) in a previously opened transmit mailbox.

Syntax `ReturnStatusTypeUpdateTransmitMailBoxAsync(MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);`

Prototype In dpa16.h or dpa32**.**h

Remarks *UpdateTransmitMailBox* allows the application to change any of the parameters included inside a transmit mailbox.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *UpdateTransMailBoxData* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example updates the Broadcast count to zero, in essence terminating the broadcast:

```
void CALLBACK  ShutOffTransmit( )

{
               unsigned char ucUpdateData[9];
               unsigned char bUpdateFlag;

      TransmitCANHandle->iBroadcastCount         = 0 ;
      bUpdateFlag | = UPDATE_BROADCAST_COUNT;
      (void)UpdateTransmitMailBoxAsync (TransmitCANHandle, bUpdateFlag);
}
```

6.5 Multiple DPA API function descriptions (alphabetical order)

6.5.1 CheckDataLink

Function Returns an identifier specifying the manufacturer name, the DLL version, the version of firmware installed on the DPA, and installed hardware capabilities.

Syntax

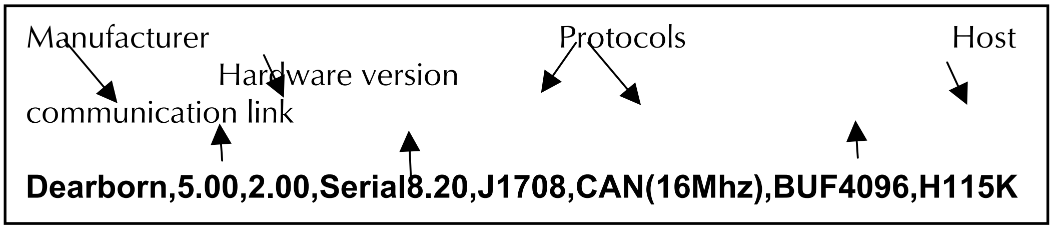
```
#include "dpam16.h"or "dpam32.h"
ReturnStatusType CheckDataLink

(short dpaHandle, char *szVersion) ;
```

Prototype In dpam16.h or dpam32.h

Remarks CheckDataLink returns an ASCII character string to **szVersion**. The response is a NULL terminated ASCII string using **,** as a delimiter and identifying the manufacturer, the software version, the hardware version, the firmware version, the protocol(s) available, the buffer size, and the host communication link.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.



Dearborn,5.00,2.00,Serial8.20,J1708,CAN(16Mhz),BUF4096,H115K

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void DisplayCheckDataLink(void);

void main(void)
{
      CommLinkType            CommLinkData;
      ReturnStatusType  InitCommStatus;
      ReturnStatusType  RestoreCommStatus;
      short             dpaHandle;

      CommLinkData.bCommPort        =eComm2;
      CommLinkData.bBaudRate        =eB115200;
      InitCommStatus = InitCommLink(&dpaHandle, &CommLinkData);

      DisplayCheckDataLink(dpaHandle);

      RestoreCommStatus =
      RestoreCommLink(dpaHandle);
}
void DisplayCheckDataLink(short
dpaHandle)
{
      char  szVersion[81];
      ReturnStatusType  CheckDataStatus;

      CheckDataStatus = CheckDataLink(dpaHandle, szVersion);

      if(CheckDataStatus == eNoError )
      {
            MessageBox( NULL, szVersion, "CheckDataLink", MB_OK );
      }
      else
      {
            MessageBox(NULL,
                        "DataLink is not responding.",
                        "CheckDataLink",
                        MB_OK);
      }
}
```

```
← If CommLink was
successful, then check
DataLink, i.e.,
 if(
```

6.5.2 ConfigureTransportProtocol

Function Sets the timing and size parameters necessary for a J1939 Transport Protocol session.

Syntax

```
ReturnStatusType   ConfigureTransportProtocol (short dpaHandle,
ConfigureTransportType*)
```

Prototype In dpam16.h or dpam32

🕮 Reference the J1939 Specification for more information regarding BAM and RTS/CTS.

Remarks This routine allows configuration of the Transport Protocol either as a Broadcast Announce Message (BAM) or a Request-To-Send / Clear-To-Send (RTS/CTS) session. For BAM, the transmit time and receive timeouts must be set. For CTS/RTS, the timeouts, data times, and CTS size must be set. The following *LoadMailBoxType* parameters must be configured for each MailBox:

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

bTransportType denotes session type: RTS/CTS or BAM
CTSSource destination address for the RTS transport

The following *InitDataLink* parameters for the mailbox must also be enabled for the active Transport Protocol:

bProtocol = J1939
bParam2 = 3

All undefined values will default to the J1939 recommended value.

```
typedef struct{
  byte      bProtocol;
  word      iBamTimeOut;
  word      iBam_BAMTXTime;
  word      iBam_DataTXTime;
  word      iRTS_Retry;
  word      iRTS_RetryTransmitTime;
  word      iRTS_TX_Timeout;
  word      iRTS_TX_TransmitTime;
  word      iRTS_RX_TimeoutData;
  word      iRTS_RX_TimeoutCMD;
  word      iRTS_RX_CTS_Count;
  word      iRTS_TX_CTS_Count;
} ConfigureTransportType;
```

bProtocol The protocol selected for the mailbox. Must be eJ1939 - J1939

iBamTimeOut The BAM timeout, (1 bit = 10 mS): the maximum time allowed between messages before a timeout occurs and the connection is aborted.

iBam_BAMTXTime The maximum time allowed between the BAM message and the first data message transmitted, before a timeout occurs and the connection is closed. (1 bit = 1 mS)

iBam_DataTXTime The maximum time allowed between data messages in a BAM session. (1 bit = 10 mS)

i**RTS_Retry** The maximum number of times the DPA will send request-to-send packets without receiving a CTS

iRTS_RetryTransmitTime The time delay between RTS request messages. (1 bit = 1 mS)

iRTS_TX_Timeout The DPA timeout value for a unit waiting for a CTS after starting a data transmission. (1 bit = 10 mS)

iRTS_TX_TransmitTime The interval between data message transmissions.

iRTS_RX_TimeoutData The maximum amount of time the DPA will wait for a message before aborting a connection.

iRTS_RX_TimeoutCMD The timeout value for the interval between a CTS and first data packet

iRTS_RX_CTS_Count The number of packets to CTS when receiving messages from a sender. (J1939 Reference: PGN 60416, Control byte = 17, byte 2)

iRTS_TX_CTS_Count The message sender's number of messages to CTS. (J1939 Reference: PGN 60416, Control byte = 16, byte 5)

Return Value *ConfigureTransportProtocol* returns **eNoError** on success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened

<u>Example</u>:

This function can be used when it is necessary to implement the J1939 Transport Protocol. The Transport Protocol is only used by J1939. Shown below is a sample function calling the <u>ConfigureTransportProtocol</u> function.

```
void TestConfigureTransportProtocol(short dpaHandle)
{
      ConfigureTransportType  cth;

      cth.bProtocol                       =eJ1939;
      cth.iBamTimeout                     =1000L;
      cth.iBAM_BAMTXTime                  =1000L;
      cth.iBAM_DataTXTime                 =1000L;
      cth.iRTS_Retry                      =3;
      cth.iRTS_RetryTransmitTime    =10L;
      cth.iRTS_TX_Timeout                 =1000L;
      cth.iRTS_TX_TransmitTime            =100L;
      cth.iRTS_RX_TimeoutData       =5000L;
      cth.iRTS_RX_TimeoutCMD        =1000L;
      cth.iRTS_RX_CTS_Count               =2;
      cth.iRTS_TX_CTS_Count               =5;

      ConfigureTransportProtocol(dpaHandle, &cth);
}
```

6.5.3 EnableTimerInterrupt

Function Enables a timer interrupt from the DPA. The interrupt initiates a user-supplied callback function.

Syntax

```
ReturnStatusType    EnableTimerInterrupt(short    dpaHandle,
EnableTimerInterruptType  *pEnableTimerInterruptData) ;
```

Prototype In dpam16.h or dpam32.h

Remarks *EnableTimerInterrupt* specifies an interrupt time interval and callback function. The current DPA timer value is passed to the callback function as a parameter.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

```
typedef struct
{
unsigned long     dwTimeOut;
void (CALLBACK *pTimerFunction)(unsigned long);
} EnableTimerInterruptType;
```

dwTimeOut Specifies the period of the interrupt, in milliseconds.

(CALLBACK *pTimerFunction)(unsigned long) Address of callback routine for timer. (NULL disables this function.)

Return Value *EnableTimerInterrupt* returns ***eNoError*** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eTimeoutValueOutOfRange	A period greater than 1 minute (60,000 ms) was specified
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include <windows.h>
#include “dpam32.h”

far long unsigned intdwTimeValue;

void CALLBACK TimerFunction(long unsigned int dwLocalTimerValue)
{
  dwTimeValue = dwLocalTimerValue;
}

void TestEnableTimerInterrupt  (short dpaHandle)
{
  ReturnStatusType EnableTimerStatus;
  long unsigned int      dwRequestedTime;

  /* enable 1 sec heart beat timer */
  EnableTimerInterruptData.dwTimeOut        = 1000L;
  EnableTimerInterruptData.pTimerFunction   = TimerFunction;
  (void)EnableTimerInterrupt (dpahandle, &EnableTimerInterruptData);

  /* turn off timer interrupt */
  SuspendTimerInterrupt(dpaHandle);
}
```

6.5.4 InitCommLink

Function Specifies and initializes communication between the PC and the Protocol Adapter.

Syntax ReturnStatusType InitCommLink
(short *dpaHandle, CommLinkType *CommLinkData);

Prototype In dpam16.h or dpam32.h

Remarks *InitCommLink* is used to initialize communications between the PC and the DPA, by identifying the COM ports and baud rate. It also sets the flags so that all other calls will know which COM port to use. This function must be called before any other library functions are called. When *InitCommLink()* is called, the appropriate interrupt vector pointers are saved so that they can be restored using *RestoreCommLink ().*

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

The *CommLinkType* structure is as follows:

```
typedef struct
{
unsigned char  bCommPort;
unsigned char  bBaudRate;
} CommLinkType;
```

bCommPort The serial communication port on the PC:
eComm1 – COM1
eComm2 – COM2
eComm3 – COM3
eComm4 – COM4
eComm5 – COM5
eComm6 – COM6
eComm7 – COM7
eComm8 – COM8
eComm9 – COM9

bBaudRate The serial communication baud rate. The DPA only supports 115K baud upon initialization. If you have a comm port that is capable of faster communication, the DPA can be set to communicate at 230K baud using the SetBaudRate() command. (32-bit only)
(eB115200 = 115200 baud)

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eSerialIncorrectPort	Incorrect Com port
eSerialIncorrectBaud	Incorrect Baudrate
eSerialPortNotFound	Com port not found
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes COM2 to communicate with the DPA at 115K baud.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
      CommLinkType             CommLinkData;
      ReturnStatusType  InitCommStatus;
      ReturnStatusType  RestoreCommStatus;
      short             dpaHandle;

      CommLinkData.bCommPort        =eComm2;
      CommLinkData.bBaudRate        =eB115200;
      InitCommStatus = InitCommLink(&dpaHandle, &CommLinkData);

      RestoreCommStatus = RestoreCommLink(dpaHandle);
}
```

6.5.5 InitDataLink

Function Initializes the specified vehicle data link with data-link information; empties all mailboxes associated with the specified data link and identifies the protocol being implemented.

Syntax

```
ReturnStatusType InitDataLink (short dpaHandle,
InitDataLinkType*pInitDataLinkData) ;
```

Prototype In dpam16.h or dpam32.h

Remarks This function initializes the hardware for specified protocols. The DPA II hardware currently supports J1939 and J1708 protocols. It allows for setting of the baud rate (for CAN and J1708), and of *Extended* (29-bit) or *Standard* (11-bit) CAN identifier priority. The hardware supports both 11- and 29- bit identifiers for CAN; however, the hardware can only double-buffer one CAN message type, so selection is left to the application.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Three callback functions (for *DataLinkError*, *TransmitVector* and *ReceiveVector*) are provided, (see below).

```
typedef struct
{
  unsigned char         bProtocol;
  unsigned char         bParam0;
  unsigned char         bParam1;
  unsigned char         bParam2;
  unsigned char         bParam3;
  unsigned char         bParam4;
  void (CALLBACK *pfDataLinkError)(MailBoxType *,
                        DataLinkErrorType *);
  void (CALLBACK *pfTransmitVector)(MailBoxType *);
  void (CALLBACK *pfReceiveVector)(void);
} InitDataLinkType;
```

bProtocol - the network protocol types

eISO9141 - ISO9141 protocol

eJ1708 - J1708 protocol
eJ1939 - J1939 protocol
eCAN - CAN protocol

bParam0-bParam4 - specific parameters for the various protocol types, described in the following section.

For CAN:

bParam0 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 0. (Reference Appendix B.1.)
bParam1 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 1. (Reference Appendix B.1.)
bParam2 is used to give preferred status to 29-bit or 11-bit CAN identifiers.
0x00 = 29 bit identifier has preferred status
0x01 = 11 bit identifier has preferred status
0x03 = 29 bit with J1939 protocol
bParam3 is not used for J1939
bParam4 is not used for J1939

For J1708:

bParam0 is used to set the baudrate used for the J1708 link
0x00 - 9600 baud
0x01 - 19.2K baud
0x02 – 10.4K baud
bParam1 is not used for J1708
bParam2 sets the use of automatic checksum creation in J1708:
0 = Full automatic checksum for transmit and receive
1 = Automatic checksum for receive only
2 = Automatic checksum for transmit only
3 = no automatic checksum
bParam3 is not used for J1708
bParam4 is not used for J1708

For pass-through mode (DPA II *Plus* only):

bProtocol = eJ1708
bParam0 = 0x80
bParam1 is used to set CD (Carrier Detect)
bParam2 is used to set DSR (Data Terminal Ready)
bParam3 is used to set RTS (Request to Send) value
bParam4 is used to set RI (Ring Indicator) value

The above parameters are configured by the following bit configuration (**0** = *Off*, **1** = *On*) for the byte:

Priority 0000 = 0 (End of Message)
0001 = Priority 1
0010 = Priority 2
0011 = Priority 3
0100 = Priority 4
0101 = Priority 5
0110 = Priority 6
0111 = Priority 7
1000 = Priority 8

(CALLBACK *pfDataLinkError)(MailBoxType *, **DataLinkErrorType** *)
Calls a routine from the address pointed by *pfDataLinkError* if a DPA error occurs. (NULL disables this function.) *MailBoxType* definition is located in Appendix A.3.

DataLinkErrorType Structure returns any errors during data link initialization:

```
typedef struct
{
unsigned char  bProtocol;
unsigned char  bErrorCode;
} DataLinkErrorType;
```

bProtocol Network protocol type (same as previous values)
bErrorCode Error code.

CALLBACK *pfTransmitVector)(**MailBoxType** *) Calls a routine pointed at by *pfTransmitVector* when a messages is transmitted. (NULL disables this callback.) *MailBoxType* definition is located in

Appendix A.3. Transmit callbacks may also be enabled on each mailbox.

(**CALLBACK *pfReceiveVector**) Calls a routine pointed at by *pfReceiveVector* when a message is received. (NULL disables this callback.) Receive callbacks may also be enabled on each mailbox.

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eProtocolNotSupported	Invalid protocol specified
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example initializes the CAN Datalink to 250K bps 29-bit identifier preferred.

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void DisplayInitDataLink(short dpaHandle);

void main(void)
{
       CommLinkType CommLinkData;
       ReturnStatusType    InitCommStatus;
       ReturnStatusType    RestoreCommStatus;
       short               dpaHandle;

       CommLinkData.bCommPort     =eComm2;
       CommLinkData.bBaudRate     =eB115200;
       InitCommStatus = InitCommLink(&dpaHandle,
       &CommLinkData);

       DisplayInitDataLink(dpaHandle);

       RestoreCommStatus = RestoreCommLink(dpaHandle);
}
```

```
← If CommLink was
successful, then check
DataLink, i.e.,
 if(
InitCommStatus==eNoError)
```

```
void DisplayInitDataLink(short dpaHandle)
{
       InitDataLinkType    InitDataLinkData;
       ReturnStatusType    InitDataLinkStatus;

       InitDataLinkData.bProtocol        = eJ1939;   // Select protocol
       InitDataLinkData.pfDataLinkError  = NULL;     // No Error Callback
       InitDataLinkData.pfTransmitVector = NULL;     // No CAN TX Callback
       InitDataLinkData.pfReceiveVector  = NULL;     // No CAN Receive Callback
       InitDataLinkData.bParam0          = 0x41;     // Set Baud
       InitDataLinkData.bParam1         = 0x58;
       InitDataLinkData.bParam2         = 0x00;             //  29-bit preferred
       InitDataLinkStatus               = InitDataLink( dpaHandle,
&InitDataLinkData);

       /* process the returned status */
       switch(InitDataLinkStatus)
       {
             case eNoError:
             {
                    MessageBox(NULL,
                    "DataLink successfully initialized.",
                    InitDataLink,MB_OK);
                    break;
             }

             case eDeviceTimeout:
             {
                    MessageBox(NULL,
                    "DataLink not responding.",
                    InitDataLink,MB_OK);
                    break;
             }

             case eProtocolNotSupported:
             {
                    MessageBox(NULL,
                       "Requested protocol is not supported",
                    InitDataLink,MB_OK);
                    break;
             }

             case eSyncCommandNotAllowed:
             {
                    MessageBox(NULL,
                    "Cannot call from within a callback (ISR) routine.",
                    InitDataLink,MB_OK);
                    break;
             }
       }
}
```

Example #2:

Initializing the J1708 Datalink to 9600 baud.

```
#include <windows.h>
#include “dpam32.h”

void DisplayInitDataLink  (short dpaHandle)
{
   InitDataLinkType            InitDataLinkData;
   ReturnStatusType            InitDataLinkStatus;

  /*  send Init DataLink command  */
  memset(&InitDataLinkData, 0, sizeof(InitDataLinkType));
  InitDataLinkData.bProtocol          = eJ1708;
  InitDataLinkData.bParam0            = 0x00;
  InitDataLinkData.pfDataLinkError    = NULL;
  InitDataLinkData.pfTransmitVector   = NULL;
  InitDataLinkData.pfReceiveVector    = NULL;
  InitDataLinkStatus     = InitDataLink(dpaHandle, &InitDataLinkData);

  /* process returned status */
  switch  (InitDataLinkStatus)
  {
    case eNoError:
    {
       MessageBox  (NULL,
           “DataLink successfully initialized”,
           “InitDataLink”, MB_OK);
       break;
    }
    case eDeviceTimeout:
    {
       MessageBox  (NULL,
           “DataLink not responding”,
           “InitDataLink”, MB_OK);
       break;
    }
    case eProtocolNotSupported:
    {
       MessageBox  (NULL,
           “Requested protocol not supported”,
           “InitDataLink”, MB_OK);
       break;
    }
    case eSyncCommandNotAllowed:
    {
       MessageBox  (NULL,
           “Cannot call from within a callback (ISR) routine”,
           “InitDataLink”, MB_OK);
       break;
    }
  }
}
```

6.5.6 InitDPA (Windows Only)

Function Specifies and initializes communication between the PC and the DPA.

Syntax #include "dpam16.h" or "dpam32.h"
ReturnStatusType InitDPA (short *dpaHandle, short DPANumber);

Prototype In dpam16.h or dpam32.h

Remarks *InitDPA* is used to initialize communications between the PC and the DPA, by identifying the number of the dpa. This function will read the parameters of the DPA from the DG1210.INI or DG121032.INI. The entry for the DPA number 1 in the INI file is listed below.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

```
[DeviceInformation1]
DeviceID=1
DeviceDescription=DPAII,COM1
DeviceName=Dearborn Protocol Adapter II
DeviceParams= COM1,0x3F8,4,B115200,INTERRUPT
```

Return Value On Success a handle to the DPA is opened is placed in the dpaHandle variable. In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eSerialIncorrectPort	Incorrect Com port
eSerialIncorrectBaud	Incorrect Baudrate
eSerialPortNotFound	Com port not found
eIrqConflict	Irq in use
eIncorrectDriver	The driver is the wrong driver
eInvalidDriverSetup	The driver is set up incorrectly
eInvalidBaseAddress	The base address is invalid
eInvalidDll	There is a bad or missing DLL
eInvalidINI	The INI record is invalid
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes DPA number 1 for communications.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;
     short               dpaHandle;

     Status = InitDPA(&dpaHandle, 1);

     RestoreCommStatus = RestoreDPA(dpaHandle);
}
```

6.5.7 InitPCCard

Function Specifies and initializes communication between the PC and the DPA.

Syntax #include"dpa16" or "dpa32.h"
ReturnStatusType InitPCCard (short *dpaHandle, PCCardType *PCCardData);

Prototype In datalink.h

Remarks *InitPCCard* is used to initialize communications between the PC and the DPA, by identifying the base address and the IRQ. It also sets the flags so that all other calls will know which DPA to use. This function must be called before any other library functions are called. When *InitPCCard()* is called, the appropriate interrupt vector pointers are saved so that they can be restored using *RestoreCommLink ().*

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

The *CommLinkType* structure is as follows:

```
typedef struct
{
unsigned short bBaseAddress;
unsigned char  bIrq;
} PCCardType;
```

bBaseAddress The base address of the PC Card.
0x200
0x220
0x300
0x320

bIrq The IRQ of the PC Card
5
7
10

Return Value On Success a handle to the DPA is opened is placed in the dpaHandle variable. In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eIrqConflict	Irq in use
eIncorrectDriver	The driver is the wrong driver
eInvalidDriverSetup	The driver is set up incorrectly
eInvalidBaseAddress	The base address is invalid
eInvalidDll	There is a bad or missing DLL
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example initializes port 200 to communicate with the DPA on IRQ 7.

```
#include  <windows.h>

#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
     PCCardType          PCCardData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;
     short               dpaHandle;

     PCCardData.bBaseAddress  =0x200;
     PCCardData.bIrq          =7;
     InitCommStatus = InitPCCard(&dpaHandle, &PCCardData);

     RestoreCommStatus = RestorePCCard(dpaHandle);
}
```

6.5.8 LoadDPABuffer

Function Loads data into the DPA's internal buffer (scratch pad).

Syntax `ReturnStatusType LoadDPABuffer (short dpaHandle, unsigned char *bData, unsigned int wLength, unsigned int wOffset);`

Prototype In dpam16.h or dpam32.h

Remarks *LoadDPABuffer* points to the current data location, assigns a length value for the data, and indicates an offset for data placement. . There are limits on the amount of data that can be passed across the serial link in a single call. These limits will differ between 16-bit and 32-bit applications.

bData Pointer to the data to be loaded into the buffer

wLength Number of bytes to transfer

wOffset Buffer address to which writing should start

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *LoadDPABuffer* returns **eNoError** on successful transmission.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened

Example:

This example loads data into the IO buffer at address 21.

```
   void WriteAppID( unsigned char *ucAppID,  int iLength )
               {
     (void)LoadDPABuffer ( dpaHandle, ucAppID, iLength, 21 );
               }
```

6.5.9 LoadMailbox

Function

Opens or creates mailboxes for receiving and transmitting messages.

This function is used to create mailboxes for one of the following functions:

- Transmitting messages
- Broadcasting messages
- Receiving messages

When a **Transmit** mailbox is opened, the host application determines:

- When the message is to be sent.
- How many times it is to be sent (Broadcast Count).
- The time intervals between consecutive transmits (Broadcast Time).
- Whether to automatically delete the mailbox after all messages are sent.
- Whether the host is to be notified when the message is sent (TX CallBack).
- Which ID (CAN) or MID/PID (J1708) is to be sent.
- What data should be sent.

When a **Receive** mailbox is opened, the host application determines the following:

- Which ID (CAN) or MID/PID (J1708) bits should be masked, and which ones should be matched, in hardware-level filtering.
- What information is needed when the message is received.
- Whether the host is to be notified when the message is sent (TX CallBack).

Syntax

```
#include “dpam16.h” or “dpam32.h”
ReturnStatusTypeLoadMailBox

(short dpaHandle, LoadMailBoxType*pLoadMailBoxData);
```

Prototype In dpam16.h or dpam32.h

Remarks

LoadMailBox opens (creates) mailboxes for the receiving and transmitting of messages, according to the structures presented in section A.3. There are three types of mailboxes:

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

- Transmit (*Broadcast* and *Release*, used to send messages out a given number of times)
- Transmit (*Resident*, used to send a message on command)
- Receive

You should initialize all structure variables to zero before using the structure shown in the following example.

Example (in C):

```
LoadMailBox MyMBox;
memset(&MyMBox,0,sizeof(LoadMailBox));
```

(The *LoadMailBoxType* structure is defined in Appendix A.3.)

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eProtocolNotSupported	Invalid protocol specified
eInvalidBitIdentSize	Invalid identifier size (11 or 29 are the only valid numbers for the CAN protocol)
eInvalidDataCount	J1939 protocol only supports 8 bytes
eMailBoxNotAvailable	All available mailboxes (16 for receiving, 16 for transmitting) are in use
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example illustrates the creation of a Receive mailbox using transparent updating.

```
#include <windows.h>
#include "dpam32.h"

unsigned char     TransUpdateDataBuffer[8];

void TestLoadMailBox  (short dpaHandle)
{
  LoadMailBoxType        LoadMailBoxData;
  ReturnStatusType       LoadMailBoxStatus,
                                     UnloadMailBoxStatus;
  MailBoxType            *TransUpdateHandle  = NULL;

/* load data for mailbox */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                        = eJ1939;
   LoadMailBoxData.bRemote_Data                     = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize                    = 29;
   LoadMailBoxData.bTransportType                   = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent                   = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask               = 0x00000000L;
   LoadMailBoxData.bTransparentUpdateEnable         = TRUE;
   LoadMailBoxData.bTimeStampInhibit                = FALSE;
   LoadMailBoxData.bIDInhibit                       = FALSE;
   LoadMailBoxData.bDataCount                       = 8;
   LoadMailBoxData.pfApplicationRoutine             = NULL;
   LoadMailBoxData.vpData                           = TransUpdateDataBuffer;
   LoadMailBoxStatus =  LoadMailBox(dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)

   {
     TransUpdateHandle  = LoadMailBoxData.pMailBoxHandle;      //save the point-
   }                                                           //er  to  the  new
                                                               //mailbox for fu-
   /* unload mailbox now that we are done */                   //ture  loading
   UnloadMailBoxStatus  = UnloadMailBox (dpaHandle, TransUpdateHandle);
   TransUpdateHandle  = NULL;
  }
```

Example #2:

This example illustrates the creation of a Receive mailbox using a callback routine.

```
#include <windows.h>
#include "dpam32.h"

unsigned char           bCallBackBuffer[8];

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
/* Copy data */
memcpy (Handle->vpData, Handle->bData, Handle->bDataCount);
}

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         UnloadMailBoxStatus;
   MailBoxType           *CallBackHandle = NULL;

/* load Receive MailBox for callback */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask       = 0x00L;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.pfApplicationRoutine     = CallBackRoutine;
   LoadMailBoxData.vpData                   = bCallBackBuffer;
   LoadMailBoxStatus            = LoadMailBox(dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     CallBackHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (dpaHandle, CallBackHandle);
   CallBackHandle = NULL;
}
```

Example #3:

This example illustrates the creation of a mailbox used to receive messages on request only, using the ***ReceiveMailBox*** *function.*

```
#include <windows.h>
#include "dpam32.h"

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         ReceiveMailBoxStatus,
                         UnloadMailBoxStatus;
   unsigned char         bRequestBuffer[8];
   MailBoxType           *RequestHandle = NULL;

/* load Receive MailBox for request */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask       = 0x0000ffffL;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.pfApplicationRoutine     = NULL;
   LoadMailBoxData.vpData                   = bRequestBuffer;
   LoadMailBoxStatus         = LoadMailBox(dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     RequestHandle = LoadMailBoxData.pMailBoxHandle;
   }

   if (RequestHandle != NULL)
   {
     ReceiveMailBoxStatus = ReceiveMailBox (dpaHandle, RequestHandle);
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (RequestHandle);
   RequestHandle = NULL;
 }
```

Example #4:

*This example illustrates the creating of a mailbox used to transmit on command using only the **TransmitMailBox** function.*

```
#include <windows.h>
#include "dpam32.h"

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         TransmitStatus,
                         UnloadMailBoxStatus;
   unsigned char               bTransmitData[8],
                               index;
   MailBoxType                 *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease       = eResident;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.bTransportType           = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.bDataCount               = 8;
   LoadMailBoxData.bTimeAbsolute            = FALSE;
   LoadMailBoxData.dwTimeStamp              = 0x00L;
   LoadMailBoxData.dwBroadcastTime          = 0x00L;
   LoadMailBoxData.iBroadcastCount          = 0;
   LoadMailBoxData.pfApplicationRoutine     = NULL;
   LoadMailBoxData.vpData                   = bTransmitData;
   LoadMailBoxStatus           = LoadMailBox (dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* if mailbox was succesfully created then transmit data 4 times */
   if (TransmitHandle != NULL)
   {
     for (index=0; index<4; index++)
     {
       TransmitStatus = TransmitMailBox (dpaHandle, TransmitHandle);
     }
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (dpaHandle, TransmitHandle);
   TransmitHandle = NULL;
 }
```

Example #5:

This example illustrates the creation of a mailbox used to transmit once when created, and then on command, using the ***TransmitMailBox*** *function.*

(*TransmitMailBox* can be used an unlimited number of times.)

```
#include <windows.h>
#include "dpam32.h"

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         TransmitStatus,
                         UnloadMailBoxStatus;
   unsigned char         bTransmitData[8],
                         index;
   MailBoxType           *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol            = eJ1939;
   LoadMailBoxData.bRemote_Data         = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease   = eResident;
   LoadMailBoxData.bBitIdentSize        = 29;
   LoadMailBoxData.bTransportType       = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent       = 0x00001234L;
   LoadMailBoxData.bDataCount           = 8;
   LoadMailBoxData.bTimeAbsolute        = FALSE;
   LoadMailBoxData.dwTimeStamp          = 0x00L;
   LoadMailBoxData.dwBroadcastTime      = 0x00L;
   LoadMailBoxData.iBroadcastCount      = 1
   LoadMailBoxData.pfApplicationRoutine = NULL;
   LoadMailBoxData.vpData               = bTransmitData;
   LoadMailBoxStatus          = LoadMailBox (dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* if mailbox was succesfully created then transmit data 4 times */
   if (TransmitHandle != NULL)
   {
     for (index=0; index<4; index++)
     {
       TransmitStatus = TransmitMailBox (dpahandle, TransmitHandle);
     }
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (dpaHandle, TransmitHandle);
   TransmitHandle = NULL;
 }
```

Example #6:

This example illustrates the creation of a mailbox used to broadcast a message 100 times every 100 ms. This example also illustrates the use of ***UpdateTransMailBoxData*** *for the updating of data being broadcast.*

```
#include <windows.h>
#include "dpam32.h"

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         UpdateStatus,
                         UnloadMailBoxStatus;
   unsigned char         bTransmitData[8],
                         bNewData[8],
                         index;
   MailBoxType           *TransmitHandle = NULL;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol             = eJ1939;
   LoadMailBoxData.bRemote_Data          = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease    = eRelease;
   LoadMailBoxData.bBitIdentSize         = 29;
   LoadMailBoxData.bTransportType        = eTransportNone;
   LoadMailBoxData.dwMailBoxIdent        = 0x00001234L;
   LoadMailBoxData.bDataCount            = 8;
   LoadMailBoxData.bTimeAbsolute         = FALSE;
   LoadMailBoxData.dwTimeStamp           = 0x00L;
   LoadMailBoxData.dwBroadcastTime       = 100L;
   LoadMailBoxData.iBroadcastCount       = 1000L;
   LoadMailBoxData.pfApplicationRoutine  = NULL;
   LoadMailBoxData.vpData                = bTransmitData;
   LoadMailBoxStatus    = LoadMailBox (dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     /* get handle of mailbox if successful */
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;

     /* update data */
     TransmitHandle->vpData = bNewData;
     UpdateStatus = UpdateTransMailBoxData (dpaHandle, TransmitHandle);
   }
   /* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (dpaHandle, TransmitHandle);
   TransmitHandle = NULL;
 }
```

Example #7:

This example illustrates the creation of a mailbox used to transmit a transport message. This example may differ depending on wether you are using the 32-bit of 16-bit drivers. The InitDataLink() call must have been called with the transport layer enabled.

```
#include <windows.h>
#include "dpa32.h"

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
    /* Message Transmitted – Perform required processing here */

}
void TestLoadMailBox  (void)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus;
   unsigned char         bTransmitData[1000],
                         index;
   MailBoxType           *TransmitHandle = NULL;
   extern  short         dpaHandle;

/* load structure with data */
   memset (&LoadMailBoxData,0,sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eDataMailBox;
   LoadMailBoxData.bResidentOrRelease       = eRelease;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.bDataCount               = 1000; //1 to 1785 bytes
   LoadMailBoxData.bTimeAbsolute            = FALSE;
   LoadMailBoxData.dwTimeStamp              = 0x00L;
   LoadMailBoxData.dwBroadcastTime          = 0L;
   LoadMailBoxData.iBroadcastCount          = 1L;
   LoadMailBoxData.pfApplicationRoutine     = CallBackRoutine;
   LoadMailBoxData.vpData                   = bTransmitData;

   //The following elements are for transport.  Extended pointer mode must
   //be used anytime that the data is longer than 8 bytes.  The extended
   //offset is the offset into the dpa buffer where the data will be
   //stored.  In 16-bit, it may be necessary to use the LoadDPABuffer
   //command to load the data into the buffer before you call the load
   //mailbox command.  If you use this method, set the bDataInhibit flag
   //to TRUE, telling the driver that the data is already in the buffer.
   //When using transport protocol, a transmit callback should be used to
   //inform the application when the transport protocol session has
   //completed.

   LoadMailBoxData.bDataInhibit             = FALSE;
   LoadMailBoxData.bTransportType           = eTransportRTS;
   LoadMailBoxData.bExtendedPtrMode         = TRUE;
   LoadMailBoxData.wExtendedOffset          = 0; //Offset into the DPA Buffer
   LoadMailBoxData.bCTSSource               = DestinationAddress;
   LoadMailBoxStatus = LoadMailBox (dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     /* get handle of mailbox if successful */
     TransmitHandle = LoadMailBoxData.pMailBoxHandle;
   }
}
```

Example #8:

This example illustrates the creation of a Receive mailbox for transport protocol using a callback routine.

```
#include <windows.h>
#include "dpam32.h"

unsigned char           bCallBackBuffer[8];

void CALLBACK CallBackRoutine (MailBoxType *Handle)
{
   /* Copy data */
   memcpy (Handle->vpData, Handle->bData, Handle->bDataCount);
}

void TestLoadMailBox  (short dpaHandle)
{
   LoadMailBoxType       LoadMailBoxData;
   ReturnStatusType      LoadMailBoxStatus,
                         UnloadMailBoxStatus;
   MailBoxType           *CallBackHandle = NULL;

/* load Receive MailBox for callback */
   memset (&LoadMailBoxData, 0, sizeof(LoadMailBoxType));
   LoadMailBoxData.bProtocol                = eJ1939;
   LoadMailBoxData.bRemote_Data             = eRemoteMailBox;
   LoadMailBoxData.bBitIdentSize            = 29;
   LoadMailBoxData.dwMailBoxIdent           = 0x00001234L;
   LoadMailBoxData.dwMailBoxIdentMask      = 0x00L;
   LoadMailBoxData.bTransparentUpdateEnable = FALSE;
   LoadMailBoxData.bTimeStampInhibit        = FALSE;
   LoadMailBoxData.bIDInhibit               = FALSE;
   LoadMailBoxData.bDataCount               = 1785; //largest message size
   LoadMailBoxData.pfApplicationRoutine     = CallBackRoutine;
   LoadMailBoxData.vpData                   = bCallBackBuffer;

   //The following elements are for transport.  Extended pointer mode must
   //be used anytime that the data is longer than 8 bytes.  The extended
   //offset is the offset into the dpa buffer where the data will be
   //stored.  In 16-bit, it may be necessary to use the ReadDPABuffer
   //command to read the data from the buffer after a receive callback is
   //received.  If you use this method, set the bDataInhibit flag to TRUE,
   //telling the driver that the data will be read later from the buffer.
   //The CTSSource element is the address of the node that will be sending
   //the CTS messages of the transport session.  For receive, this is the
   //address of the DPA on the link.

   LoadMailBoxData.bDataInhibit           = FALSE;
   LoadMailBoxData.bTransportType         = eTransportRTS;
   LoadMailBoxData.bExtendedPtrMode       = TRUE;
   LoadMailBoxData.wExtendedOffset        = 1785; //Offset in the Buffer
   LoadMailBoxData.bCTSSource             = OurLocalAddress;

   LoadMailBoxStatus           = LoadMailBox(dpaHandle, &LoadMailBoxData);
   if (LoadMailBoxStatus == eNoError)
   {
     CallBackHandle = LoadMailBoxData.pMailBoxHandle;
   }

/* unload mailbox now that we are done */
   UnloadMailBoxStatus = UnloadMailBox (dpaHandle, CallBackHandle);
   CallBackHandle = NULL;
}
```

6.5.10 LoadTimer

Function Sets the timer, for the timestamping of received and transmitted messages.

Syntax

```
#include "dpam16.h"or "dpam32.h"
ReturnStatusType   LoadTimer   (short dpaHandle,long unsigned
int dwTime);
```

Prototype datalink.h

Remarks *LoadTimer* sets the timer used for the timestamping of received and transmitted messages. It takes the parameter passed in **dwTime** and loads it in the DPA II's timer. The timer has a 1 mS resolution, and count wrapping occurs about every 49 days.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *LoadTimer* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use this function, you can create a function such as the one below, or just place the ***LoadTimer*** *call in directly into your code.*

```
void GoLoadTimer(short dpaHandle)
{
     ReturnStatusType LoadTimerStatus;

     // Initialize the timer with a value of 1000ms
     LoadTimerStatus = LoadTimer(dpaHandle, 1000L);
}
```

6.5.11 PauseTimer

Function Pauses the DPA II's internal timer, which suspends all transmits and callbacks (interrupts).

Syntax `ReturnStatusType PauseTimer; (short dpaHandle)`

Prototype dpam16.h or dpam32.h

Remarks *PauseTimer* stops the timer, along with all transmits and callback interrupts.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *LoadTimer* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use the Pause Timer function, simply make the following call in your program:

```
     PauseTimer(dpaHandle);
```

6.5.12 ReadDPABuffer

Function Reads data from the DPA's I/O buffer (internal scratch memory).

Syntax

```
ReturnStatusType ReadDPABuffer (short dpaHandle,unsigned
char *bData, unsigned int wLength, unsigned int
wOffset);
```

Prototype In dpam16.h or dpam32.h

Remarks This routine allows the host to read data out of the I/O Buffer. That data may be loaded into the DPA by another application or via any attached mailbox. Due to linitations in 16-bit operating systems, tt may be necessary to break the reading of large buffers into multiple calls to ReadDPABuffer.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

bData	pointer to the location for the data
wLength	number of data bytes to be read
wOffset	address in the buffer where reading should begin

Return Value *ReadDPABuffer* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened

<u>Example</u>:

*This example demonstrates the reading of data stored at the buffer's **Address 21**.*

```
   void ReadAppIDIOBuffer( short dpaHandle, unsigned char *AppID,
int iLength)
               {

       (void)ReadDPABuffer (dpaHandle, ucAppID, iLength, 21);

   }
```

6.5.13 ReadDPAChecksum

Function Returns the value of the DPA checksum, for software verification of changes or corruption.

Syntax

```
ReturnStatusType ReadDPAChecksum (short dpaHandle,unsigned int
*varname) ;
```

Prototype In dpam16 or dpam32.h

Remarks This function checks the DPA checksum value, for any corruption of the DPA Flash memory. (The memory can become corrupted when the Flash memory fades, is improperly programmed, or is physically damaged.) *ReadDPAChecksum* returns the checksum and puts it in **varname**, a user-defined variable.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA

Example:

*The following code demonstrates one example of how the **ReadDPAChecksum** function may be used. The variable **uiVer707Checksum** can represent anything you wish.*

```
void TestFlash(short dpaHandle)
{
unsigned int uiChecksum;

status = DPAReadDPAChecksum(dpaHandle, &uiChecksum);

if (status == eNoError)
    {
    if uiChecksum == uiVer707Checksum)
        {
        // do whatever here
        }
    }
```

6.5.14 ReceiveMailBox

Function Retrieves the latest message from a specific mailbox.

Syntax `ReturnStatusType ReceiveMailBox (short dpaHandle,MailBoxType *pMailBoxHandle) ;`

Prototype In dpam16.h or dpam32.h

Remarks *ReceiveMailBox* retrieves the latest message from a specific mailbox. Previous messages are overwritten.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

The **MailBoxType** structure is found in Appendix A.3.

pMailBoxHandle is a pointer to the Mailbox's unique name or handle.

Return Value *ReceiveMailBox* returns **eNoError** upon success, or any of the following messages to report the conditions listed to their right:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eInvalidMailBox	Attempting to receive with unopened mailbox
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

<u>Example #1</u>:

*Retrieves data last received in a CAN mailbox specified by the mailbox handle **J1939ReceiveHandle**.*

```
if ((J1939ReceiveHandle != NULL) &&
                      (J1939ReceiveHandle->bActive != 0))
         {
            (void)ReceiveMailBox    (dpaHandle,    J1939ReceiveHandle);
}
```

Example #2:

*Retrieves data last received in a J1708 mailbox specified by the mailbox handle **J1708ReceiveHandle**.*

```
              if ((J1708ReceiveHandle != NULL) &&
                  (J1708ReceiveHandle->bActive != 0))
             {
      (void)ReceiveMailBox(dpaHandle,J1708ReceiveHandle);
      }
```

6.4.15 ResetDPA

Function Resets the DPA based upon the parameter passed.

Syntax
```
#include"dpam16.h"or "dpam32.h"
ReturnStatusType   ResetDPA   (short   DpaHandle,
ResetType  *ResetData);
```

Prototype In dpam16.h or dpam32.h

Remarks *ResetDPA is* used to command a low-level reset of the DPA. It can be used to perform a full reset of the hardware, or a communications-only reset.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
      CommLinkType            CommLinkData;
      ReturnStatusType  InitCommStatus;
      ReturnStatusType  RestoreCommStatus;
      ReturnStatusType  ResetStatus;
      ResetType               ResetData;
      Short             DpaHandle;

      CommLinkData.bCommPort        =eComm2;
      CommLinkData.bBaudRate        =eB115200;
     InitCommStatus =
                  InitCommLink(&DpaHandle,&CommLinkData);
      ResetData.bResetType = eFullReset;
      ResetStatus = ResetDPA (DpaHandle, ResetData);
      RestoreCommStatus = RestoreCommLink(DpaHandle);
}
```

6.5.16 RestoreCommLink

Function Restores the communication port between the PC and the DPA II to its previous (pre-*InitCommLink*) state.

Syntax `ReturnStatusType RestoreCommLink (short dpaHandle)`

Prototype In dpam16.h or dpam32.h

Remarks *RestoreCommLink* is used to restore any and all interrupt vectors that were set up during *InitCommLink().* Once *RestoreCommLink ()* is called, no other library functions (except *InitCommLink ()*) can be called.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

<u>Example</u>:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
     CommLinkType        CommLinkData;
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;
     short               dpaHandle;

     CommLinkData.bCommPort        =eComm2;
     CommLinkData.bBaudRate        =eB115200;
    InitCommStatus = InitCommLink(&dpaHandle,&CommLinkData);

     RestoreCommStatus = RestoreCommLink(dpaHandle);
}
```

6.5.17 RestoreDPA (Windows Only)

Function Restores the communication port between the PC and the DPA II to its previous (pre-InitDPA) state.

Syntax

```
#include"dpam16.h" or "dpam32.h"
ReturnStatusType RestoreDPA (short dpaHandle)
```

Prototype In dpam16.h or dpam32.h

Remarks *RestoreDPA* is used to restore any and all interrupt vectors that were set up during *InitDPA()*. Once *RestoreDPA()* is called, no other library functions (except *InitDPA ()*) can be called.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
     ReturnStatusType    InitCommStatus;
     ReturnStatusType    RestoreCommStatus;
     short               dpaHandle;

     InitCommStatus = InitDPA(&dpaHandle,1);

     RestoreCommStatus = RestoreDPA(dpaHandle);
}
```

6.5.18 RestorePCCard

Function Restores the communication port between the PC and the DPA II to its previous (pre-*InitPCCard*) state.

Syntax

```
#include"dpam16.h"or "dpam32.h"
ReturnStatusType RestorePCCard (short dpaHandle)
```

Prototype In dpam16.h or dpam32.h

Remarks *RestorePCCard* is used to restore any and all interrupt vectors that were set up during *InitPCCard().* Once *RestorePCCard ()* is called, no other library functions (except *InitPCCard ()*) can be called.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value The following *ReturnStatusType* messages may appear:

eNoError	Success
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"

void main(void)
{
     PCCardType           PCCardData;
     ReturnStatusType     InitCommStatus;
     ReturnStatusType     RestoreCommStatus;
     short                dpaHandle;

     PCCardData.bBaseAddress       =0x200;
     PCCardData.bIrq               =7;
     InitCommStatus = InitPCCard(&dpaHandle, &PCCardData);
     ReturnStatusType = RestorePCCard(dpaHandle);
}
```

6.5.19 RequestTimerValue

Function Returns the current DPA timer value.

Syntax `ReturnStatusType RequestTimerValue (short dpaHandle,`

`long unsigned int *dwTimerValue) ;`

Prototype In dpam16.h or dpam32.h

Remarks *RequestTimerValue* returns the current DPA timer value. The timer value is placed into **dwTimerValue**.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *RequestTimerValue* returns **eNoError** upon success.
In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial prot not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

```
#include <windows.h>
#include “dpam32.h”

void TestRequestTimerValue  (short dpaHandle)
{
  ReturnStatusType LoadTimerStatus,
             RequestTimerValueStatus;
  long unsigned int      dwRequestedTime;
  char             szBuffer [81];

  /* load timer with 1000, 1 second */
  LoadTimerStatus = LoadTimer (dpaHandle, 1000L);

  /* request timer value and display */
  RequestTimerValueStatus=RequestTimerValue (dpaHandle, &dwRequestedTime);
  vsprintf  (szBuffer,
       “Current DataLink timer value is: %ld”,
       dwRequestedTime);
  MessageBox  (NULL, szBuffer, “RequestTimerValue”, MB_OK);
}
```

6.5.20 ResumeTimer

Function Resumes a previously paused timer function and re-starts all transmits and callback interrupts.

Syntax `ReturnStatusType  ResumeTimer (short dpaHandle);`

Prototype dpam16.h or dpam32.h

Remarks *ResumeTimer* restarts the DPA II's internal timer previously stopped by the *PauseTimer* function. Transmit and callback interrupt functions are also resumed.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *LoadTimer* returns **eNoError** on success. In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use the Resume TImer funtion, simply make the following call in your program:

```
ResumeTimer(dpaHandle);
```

6.5.21 SetBaudRate (32-bit Windows Only)

Function Send a command to change to baud rate on the serial link between the DPA and the PC.

Syntax `ReturnStatusType SetBaudRate (short dpaHandle, BaudRateType *baudRate);`

Prototype dpam16.h or dpam32.h

Remarks SetBaudRate performs 3 steps. Sets the DPA baud to the requested value, sets the PC baud to the requested value, and verifies that the DPA and the PC can still communicate. Baud rates higher than 115200 are only available on serial ports that will support these baud rates.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *SetBaudRate* returns **eNoError** on success. In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eBaudRateNotSupported	Invalid baud rate passed in
eDeviceTimeout	Could not command DPA to change baud
eCommVerificationFailed	Could not communicate after change
eSerialOuputError	Could not set PC baud

Example:

To use the SetBaudRate funtion, simply make the following call in your program:

```
SetBaudType    baudRates;
BaudRates.bPCBaud = eb230400;
BaudRates.bDPABaud = eb230400;
SetBaudrate(dpaHandle, baudRates);
```

6.5.22 SuspendTimerInterrupt

Function Disables a DPA timer interrupt.

Syntax `ReturnStatusType SuspendTimerInterrupt`
`(short dpaHandle)`

Prototype In dpam16 or dpam32.h

Remarks **SuspendTimerInterrupt** disables a DPA timer interrupt.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *SuspendTimerInterrupt* returns **eNoError** upon success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use this function, simply make the following call in your program.

```
SuspendTimerInterrupt(dpaHandle);
```

6.5.23 TransmitMailBox

Function Sends messages, using previously opened mailboxes.

Syntax

```
ReturnStatusType TransmitMailBox
(short dpaHandle,MailBoxType *pMailBoxHandle) ;
```

Prototype In dpam16.h or dpam32.h

Remarks This function is used to transmit already loaded mailboxes. Once established, data can only be updated with the *UpdateTransMailBoxData* function before the *TransmitMailBox* function is re-called.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure defined in Appendix A.3.)
***pMailBoxHandle**	A pointer to the Mailbox's unique name or handle.

Return Value *TransmitMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eInvalidMailBox	Attempting to transmit an unopened mailbox
eCommLinkNotInitialized	**Serial port not opened**
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example sends the specified mailbox once, with no modification to any of the mailbox parameters.

```
(void)TransmitMailBox (dpaHandle, TransmitCANHandle);
```

6.5.24 TransmitMailBoxAsync

Function Sends messages asynchronously, using previously opened mailboxes, from a function within a callback (ISR).

Syntax

```
ReturnStatusType TransmitMailBoxAsync (short dpaHandle,MailBoxType
*pMailBoxHandle);
```

Prototype In dpam16.h or dpam32.h

Remarks This function is identical to *TransmitMailBox*, except for its use of the DPA's *Non-Verbose* mode for turning off the DPA response inside the application interrupt.

TransmitMailBoxAsync is used to update Broadcast mailbox callback information, or to send messages using resident mailboxes.

The *MailBoxType* structure can be found in Appendix A.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *TransmitMailBoxAsync* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eInvalidMailBox	Attempting to transmit an unopened mailbox
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This example illustrates the transmission (in response to a receive callback) of a previously loaded transmit mailbox.

```
     void ReceiveCallBack (MailBoxType *)
          {
     TransmitMailBoxAsync(dpaHandle, TransmitCANHandle)
}
```

6.5.25 UnloadMailbox

Function Closes a previously opened mailbox.

Syntax `ReturnStatusType UnloadMailBox (short dpaHandle,MailBoxType *MailBoxHandle) ;`

Prototype In dpam16.h or dpam32.h

Remarks *UnloadMailBox* closes a mailbox and disables all related callback functions.

The *MailBoxType* structure can be found in Appendix A

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

Return Value *UnloadMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

*This example illustrates the unloading of the mailbox specified by the mailbox handle **TransmitCANHandle**.*

```
   (void)UnloadMailBox (dpaHandle, TransmitCANHandle);
```

6.5.26 UpdateReceiveMailBox

Function Updates the data count, data location, identifier, and identifier mask of an open receive mailbox.

Syntax

```
ReturnStatusType UpdateReceiveMailBox
(short  dpaHandle,MailBoxType  *pMailBoxHandle,  unsigned
char bUpdateFlag) ;
```

Prototype In dpam16.h or dpam32.h

Remarks *ReceiveMailBox* retrieves the latest message for a specific mailbox. Previous messages are overwritten.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *ReceiveMailBox* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

*To use the **UpdateReceiveMailBox** function, you can simply add the following lines to your code or even create a function of your own. The **bUpdateFlag** variable is not mentioned in the description but is a necessary parameter.*

```
MailBoxType        *J1708ReceiveHandle;
ReturnStatusType  UpdateReceiveStatus;
unsigned char      bUpdateFlag;

      UpdateReceiveStatus =
      UpdateReceiveMailBox(dpaHandle,&J1708ReceiveHandle, bUpdateFlag);
```

6.5.27 UpdateReceiveMailBoxAsync

Function Updates the data count, data location, identifier, and identifier mask of an open receive mailbox.

Syntax

```
ReturnStatusType ReceiveMailBox
(short  dpaHandle,MailBoxType  *pMailBoxHandle,  unsigned
char bUpdateFlag) ;
```

Prototype In dpam16.h or dpam32.h

Remarks *ReceiveMailBox* retrieves only the information which flags were set, (in the *MailBoxType* structure), to receive. There is no return value for *ReceiveMailBox*.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *ReceiveMailBox* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

To use the ***UpdateReceiveMailBoxAsync*** *function, you can simply add the following code to your program (in this case, for receiving J1708 messages):*

```
MailBoxType    *J1708ReceiveAsyncHandle;
ReturnStatusType UpdateReceiveAsyncStatus;
unsigned char bUpdateFlagAsync;

UpdateReceiveAsyncStatus = UpdateReceiveMailBoxAsync(
dpaHandle, &J1708ReceiveAsyncHandle, bUpdateFlagAsync);
```

6.5.28 UpdateTransMailBoxData

Function Updates information being sent from a previously opened transmit mailbox.

Syntax

```
ReturnStatusType UpdateTransMailBoxData
(short dpaHandle,MailBoxType *pMailBoxHandle);
```

Prototype In dpam16.h or dpam32.h

Remarks Updates only the data being sent from a previously opened mailbox. **dpaHandle** denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)

Return Value *UpdateTransMailBoxData* returns **eNoError** on success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

You should include the following defines in your code:

```
MailBoxType *TransmitJ1708Handle;         // Declare the Mail Box Handle
unsigned char  TransmitMailBoxBroadcastData[9];
```

6.5.29 UpdateTransMailBoxDataAsync

Function Updates information being sent from a previously opened transmit mailbox., for use inside a callback routine.

Syntax

```
ReturnStatusTypeUpdate TransMailBoxDataAsync
(short dpaHandle,MailBoxType *pMailBoxHandle);
```

Prototype In dpam16.h or dpam32.h

Remarks *UpdateTransMailBoxDataAsync* is used to update only the data being sent from a previously opened mailbox, for use inside a callback routine.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)

Return Value *UpdateTransMailBoxDataAsync* returns **eNoError** on success.

In the event of an error return, the following ReturnStatusType messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example:

This is an example of how to update and retransmit data, in a transmit callback routine.

```
void TransmitCallBack ()
{
    strcpy ((char *)TransmitMailBoxBroadcastData, "12345678");
   (void)UpdateTransMailBoxDataAsync(dpaHandle, TransmitCANHandle);
        }
```

6.5.30 UpdateTransmitMailBox

Function Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) from a previously opened transmit mailbox.

Syntax `ReturnStatusType UpdateTransmitMailBox`

`(short dpaHandle,MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);`

Prototype In dpam16**.**h or dpaM32.h

Remarks *UpdateTransmitMailBox* allows the application to change any of the parameters included inside a transmit mailbox.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *UpdateTransMailBoxData* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened
eSyncCommandNotAllowed	Cannot call from within callback (ISR)

Example #1:

This example illustrates an update of the data inside a transmit mailbox.

```
   {
       unsigned char ucUpdateData[9];
       unsigned char bUpdateFlag;

       strcpy ((char *)UpdateData, "12345678");
       TransmitCANHandle->vpData  = UpdateData;
       bUpdateFlag |= UPDATE_DATA;

       (void)UpdateTransmitMailBox (dpahandle,TransmitCANHandle, bUpdateFlag);

   }
```

Example #2:

This example shows the updating of the Broadcast count to zero, in essence terminating broadcast.

```
   void  ShutOffTransmit( MailBoxType    *TransmitCANHandle)

   {
      unsigned char ucUpdateData[9];
      unsigned char bUpdateFlag;

      TransmitCANHandle->iBroadcastCount         = 0;
      bUpdateFlag |= UPDATE_BROADCAST_COUNT;

      (void)UpdateTransmitMailBox      (dpaHandle,      TransmitCANHandle,
bUpdateFlag);
   }
```

Example #3:

This example updates both the broadcast time and the broadcast count, causing the specified mailbox to be broadcast the specified number of times, at 100-millisecond intervals.

```
   ReturnStatusType BroadCastCount100ms( MailBoxType    *TransmitCANHandle,
                                                signed int iBroadCastCount )
   {

                  // Update Broadcast Time
      TransmitCANHandle->dwBroadcastTime         = 100;
       bUpdateFlag |= UPDATE_BROADCAST_TIME;

      // Load Broad Cast Count
      TransmitCANHandle->iBroadcastCount         = iBroadCastCount;
       bUpdateFlag |= UPDATE_BROADCAST_COUNT;

      // Send Command to DPA and return response
      return (UpdateTransmitMailBox(dpaHandle, TransmitCANHandle, bUpdateFlag));

   }
```

6.5.31 UpdateTransmitMailBoxAsync

Function (For use inside a Callback routine.) Updates any information (e.g., ID, transmit time, broadcast time, broadcast count, or data) in a previously opened transmit mailbox.

Syntax

```
ReturnStatusTypeUpdateTransmitMailBoxAsync

(short dpaHandle,MailBoxType  *pMailBoxHandle, unsigned
char bUpdateFlag);
```

Prototype In dpam16**.**h or dpam32.h

Remarks *UpdateTransmitMailBox* allows the application to change any of the parameters included inside a transmit mailbox.

dpaHandle denotes the DPA to send this command to. This is the handle that it returned from InitCommLink, InitPCCard or InitDPA.

MailBoxType	(Structure found in section A.3)
pMailBoxHandle	Pointer to the Mailbox's unique name (handle)
bUpdateFlag	Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

Return Value *UpdateTransMailBoxData* returns **eNoError** upon success.

In the event of an error return, the following *ReturnStatusType* messages may appear:

eNoError	Success
eDeviceTimeout	Unable to communicate with the DPA
eMailBoxNotActive	Mailbox was not open
eCommLinkNotInitialized	Serial port not opened

Example:

This example updates the Broadcast count to zero, in essence terminating the broadcast:

```
void CALLBACK  ShutOffTransmit( )

{
               unsigned char ucUpdateData[9];
               unsigned char bUpdateFlag;

      TransmitCANHandle->iBroadcastCount         = 0 ;
      bUpdateFlag | = UPDATE_BROADCAST_COUNT;
      (void)UpdateTransmitMailBoxAsync    (dpaHandle,    TransmitCANHandle,
bUpdateFlag);
}
```

6.6 ISO-9141 DPA API function descriptions

The following API functions are not available for a standard DPA. They are only available for DPAs which have ISO-9141 support installed.

6.6.1 INIT_KWP200_DPA

API Example:

INT RET=INIT_KWP2000_DPA(INT DPAiniEntry)

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

Action:

This API command shall be the first command called by the application software at the time that the application software is initialized. This will allow the network adapter to perform initializations for KWP communications.

This command shall not result in a KWP message to the ECU.

6.6.2 RELEASE_KWP2000

API Example:

INT RET=RELEASE_KWP2000()

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

Action:

This API command shall be the last command called by the application software before the application shuts down. This will allow the network adapter to release resources used for KWP communications.

This command shall not result in a KWP message to the ECU.

6.6.3 SET_TIMING

API Example:

```
INT RET=SET_TIMING (    INT P1,
                        INT P2,
                        INT P3,
                        INT P4... )
```

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

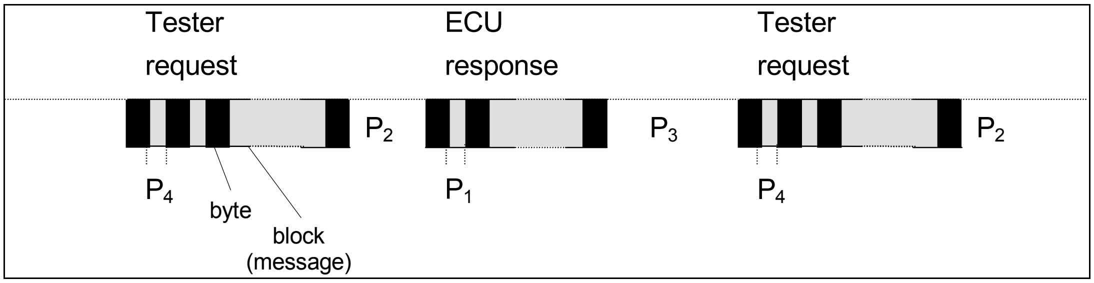
Action:

This command shall set the protocol timing parameters within the network adapter. Calling of this command does not result in a KWP message to the ECU. Parameters shall be expressed in units of milliseconds. Prior to this command being used the network adapter shall use the default values defined below.

SET_TIMING PARAMETERS TABLE

VALUE	DESCRIPTION
P_1	Inter-byte-time in the ECU response message
P_2	Time between end of tester request and start of ECU response (inter-block-time)
P_3	Time between end of ECU response and start of new tester request (inter-block-time)
P_4	Inter-byte-time in the tester request message

MESSAGE FLOW TIMING



	MAX	*MIN*	*DEFAULT*
P1	*20ms*	*2ms*	*5ms*
P2	*2000ms*	*25ms*	*50ms*
P3	*4000ms*	*25ms*	*50ms*
P4	*20ms*	*2ms*	*5ms*

6.6.4 SET_COM_PARAMETER_1

API Example:

```
RET=SET_COM_PARAMETER_1(   INT TimeInit_Low,
                           INT TimeInit_High,
                           BYTE DataBit,
                           BYTE ParityBit)
```

TimeInit_Low , TimeInit_High: expressed in milliseconds.
DataBit,ParityBit : Communication parameters.

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

Action:

This command shall set communication parameters in the network adapter. The application software must call this command successfully before the START_COMMUNICATION command is called. The network adapter shall only process this command BEFORE communication has been initialized with the ECU. This command does not result in a KWP message.

(Parameters in this command are only included for flexibility of design. Currently ISO 14230-2 does not allow any change to these parameters.)

If this command is not used by the application, the network adapter shall use the default parameters defined below.

The parameters TimeInit_Low and TimeInit_High (expressed in milliseconds) shall set the timing of the fast initialization of the ECU as defined in ISO14230-2.

The Parameter DataBit sets the number of data bits that the network adapter uses in a data byte. Valid values for this parameter are limited to the values listed below.

Valid DataBit Values

DataBit Value	Description
7	Seven data bits per byte
8	Eight data bits per byte

The Parameter ParityBit sets the use of parity in the communication of the network adapter. Valid Values for this parameter are limited to the values listed below.

Valid ParityBit Values

PARITYBIT VALUE	Description
0	NO PARITY USED IN COMMUNICATION
1	Odd Parity used in communication
2	Even Parity used in communication

Default Values:

Parameter	Default Value
TimeInit_Low	25ms
TimeInit_High	50ms
DataBit	8
ParityBit	0 (none)

6.6.5 SET_COM_PARAMETER_2

API Example:

```
        RET=SET_COM_PARAMETER_2(INT TargetAdd,
                                INT SourceAdd,
                                BYTE AddrPresenceByte)
```

TargetAdd, SourceAdd : (Target and Source address) see comments below.

AddrPresenceByte = 0/1/2/3 see comments below

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

Action:

This command shall set communication parameters in the network adapter. This command does not result in a KWP message.

The AddrPresenceByte parameter shall configure the Header field of KWP messages sent from network adapter via the SEND_MESSAGE and STOP_COMMUNICATION command. The first two most significant bits of the Format byte and the presence of the address bytes in the Header field shall be configured according to ISO 14230-2; definition is also repeated in the details section below. Valid Values for this parameter are limited to the values listed below.

Note: The Key Bytes returned from the ECU in the StartCommunication Positive Response Message (see section0) shall override this setting UNLESS the returned key byte value is 0x8FD0, according to ISO14230-2 section 5.2.4.1. (0x8FD0 indicates the ECU is not driving the header field values.)

This parameter setting shall not affect the StartCommunication KWP message (sent via the START_COMMUNICATION API command), as the address bytes are always required for this message according to ISO 14230-2.

Valid AddrPresenceByte Values

ADDRPRESENCEBYTE	MODE
0	No address information
2	with address information, physical addressing
3	with address information, functional addressing

The TargetAddress and SourceAddress parameters shall be used by the network adapter in Header fields as appropriate.

Default Values:

Parameter	Default Value
TargetAddress	0x00*
SourceAddress	0x01
AddrPresenceByte	0 (no addr info)

* It could be set by our application with SET_COM_PARAMETER_2 API.

Detail:

This note describes the structure of a message. This information is taken from and complies with ISO 14230-2. The message structure consists of three parts:

- Header
- Data bytes
- Checksum

Header				Data bytes		Checksum
Fmt	Tgt[1]	Src[1]	Len[1]	Sid[2]	. . . Data . . .	CS
Max. 4 byte				max. 255 bytes		1 byte

[1] Bytes are optional, depending on the format byte
[2] Service identification, part of data bytes

Header

The header consists of maximum 4 bytes. A format byte includes information about the form of the message. Target and source address bytes are optional for use with multi node connections. An optional separate length byte allows message lengths up to 255 bytes. The different ways of using header bytes is shown in the figure below.

Format Byte

The format byte contains 6 bit length information and 2 bit address mode information. The tester is informed about use of header bytes by the key bytes in the StartCommunication positive response message.

A_1	A_0	L_5	L_4	L_3	L_2	L_1	L_0

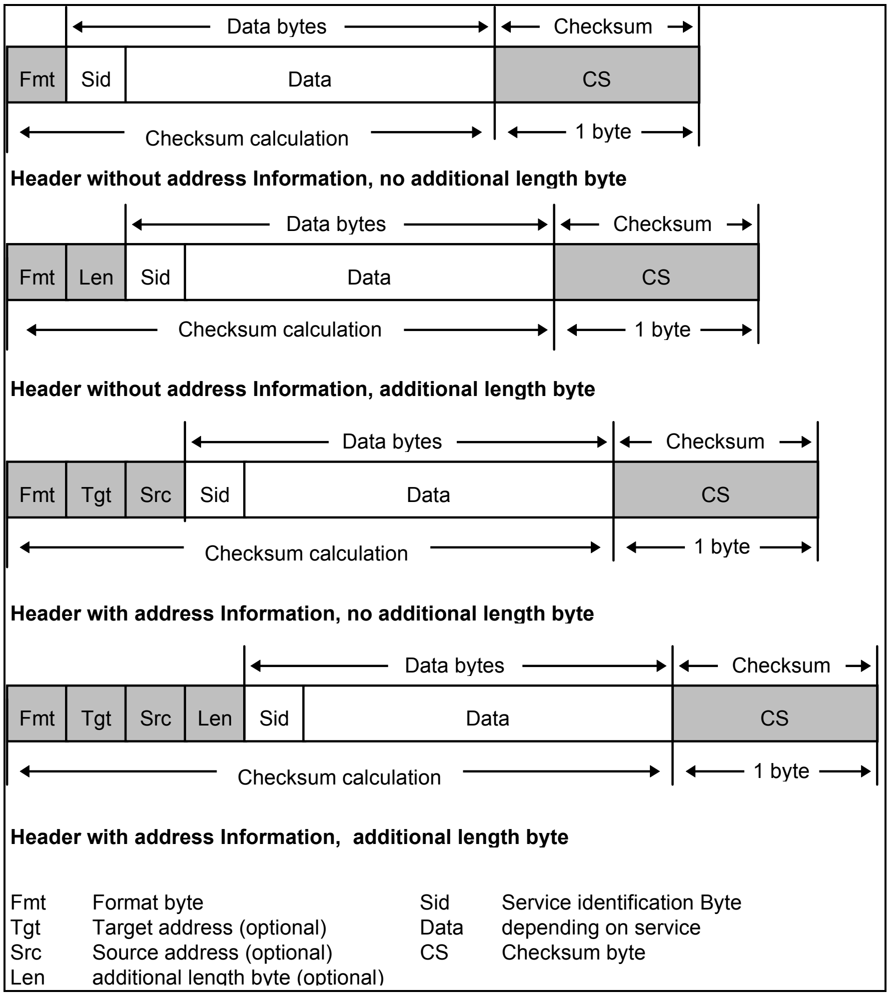
A_1, A_0: define the form of the header, which will be used by the message (see below).

$L_5..L_0$:define the length of a message from the beginning of the data field (service identification byte included) to the checksum byte (not included). A message length of 1 to 63 bytes is possible. If L_0 to L_5 = 0, then the additional length byte is included.

ADDRPRESENCEBYTE TABLE

A_1	A_0	ADDRPRESENCEBYTE	MODE
0	0	0	No address information
1	0	2	with address information, physical addressing
1	1	3	with address information, functional addressing

Use of Header bytes



Header without address Information, no additional length byte

Header without address Information, additional length byte

Header with address Information, no additional length byte

Header with address Information, additional length byte

Fmt	Format byte	Sid	Service identification Byte
Tgt	Target address (optional)	Data	depending on service
Src	Source address (optional)	CS	Checksum byte
Len	additional length byte (optional)		

6.6.6 SET_BAUDRATE

API Example:

RET=SET_BAUDRATE(LONG BaudRate)

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

This command shall set the baud rate at which the network adapter communicates to the ECU. (This value is NOT the baud rate at which the PC communicates with the network adapter.)

The baud rate set for this communication shall be used for KWP messages sent via the SEND_MESSAGE, START_COMMUNICATION, and STOP_COMMUNICATION command.

It is the responsibility of the application software to synchronize the baud rate of the network adapter to that of the ECU. If this command is not used the network adapter shall use the default value listed below.

Valid BaudRate Values:

BaudRate Value
8.192 Kbaud
9.6 Kbaud
19.2 Kbaud
10.4 Kbaud
38.4 Kbaud
115.2 Kbaud

Default BaudRate Value:

Parameter	Default Value
BaudRate	10.4Kbaud

6.6.7 START_COMMUNICATION

API Example:

RET=START_COMMUNICATION(BYTE* KEY_BYTES)

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

This command shall start the communication with the ECU. The sequence of events shall be 1) Fast Initialization and 2) a StartCommunication Request message.

The network adapter shall use the TimeInit_Low and TimeInit_High parameter values to perform the Fast Initialization. After Fast Initialization the baud rate in the network adapter shall be set to the value of the BaudRate parameter.

NOTE: 5-baud and CARB initialization is not supported via this API.

The StartCommunication Request message is sent with a three byte Header field; Format byte, Source Address byte, and Target address byte. The address bytes shall be included regardless of the setting of the AddrPresenceByte Parameter.

StartCommunication Request Message Example

Byte	Parameter Name	Hex Value	Comment
1	Format byte physical addressing functional addressing	xx=[0x81 0xC1]	Value determined by the AddrPresenceByte parameter.
2	Target address byte	xx	Value determined by TargetAdd parameter
3	Source address byte	xx	Value determined by SourceAdd parameter
4	startCommunication Request Service Id	0x81	Constant; as defined by ISO 14230-2.
5	Checksum	xx	As defined by ISO 14230-2.

The network adapter shall be prepared to receive a StartCommunication Positive Response. The ntework adapter shall use the Key Bytes returned here to define the header fields of the KWP messages sent to the ECU via the SEND_COMMUNICATION. The Key Bytes are defined in ISO 14230-2 section 5.2.4.1. The Key Bytes shall override the settings determind by the AddrPresenceByte unless the Key Byte value (together) is 0X8FD0 according to ISO 14230-2 section 5.2.4.1

Byte	Parameter Name	Hex Value	Comment
1	Format byte	xx	
2	Target address byte	xx	Conditional - Note 1
3	Source address byte	xx	Conditional - Note 1
4	Additional length byte	xx	Conditional - Note 2
5	startCommunication Positive Response Service Id	0xC1	Constant; as defined by ISO 14230-2.
6	Key byte 1 Key byte 2	Xx Xx	Network adapter shall use these key bytes as defined in ISO14230-2 sec 5.2.4.1
7	Checksum	Xx	As defined by ISO 14230-2.

Note 1: Format byte is 10xx xxxx or 11xx xxxx
Note 2: Format byte is xx00 0000.

The network adapter shall also store the Key Bytes in the memory address indicated by the KeyBytes parameter.

API command shall return '0' (OK) only after successful StartCommunication Positive Response has been received by the network adapter and the Key Bytes have been processed.

6.6.8 STOP_COMMUNICATION

API Example:

RET=STOP_COMMUNICATION()

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

This command shall stop the communication with the ECU. This command shall result in a StopCummunication Request message from the network adapter to the ECU.

The command shall return a 0 if the StopCommunication Positive Response is received from the ECU. The command shall return –1 if the StopCommunication Negative Response is return or if the command times out.

6.6.9 GET_STATUS

API Example:

INT STATUS=GET_STATUS()

STATUS = 0 → OK
STATUS =-1 → TIME_OUT COMMAND

This command shall return the status of the KWP communication between the network adapter and the ECU.

The network adapter shall determine the status of communication between itself and the ECU by the presence of a TesterPresence Positive Response. This command shall result in a TesterPresent Request Message. The network adapter shall return a '0' if it successfully receives a TesterPresent Positive Response. It shall return a '-1' if it receives a TesterPresent Negative Response or if it times out without a response.

Note: A solution may be implemented in which the network adapter does not need to send out a separate TesterPresent message if the network adapter is already sending out TesterPresent Request Messages according to section 0.

6.6.10 SEND_MESSAGE

API Example:

```
RET=SEND_MESSAGE    ( BYTE*     MSG_LEN,
                      BYTE* DATA,
                      BYTE*     MSG_LENRX,
                      BYTE*  DATARX).
```

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

This API can be used for transmitting and receiving data from the ECU's.

It requires the MSG_LEN byte and the DATA array bytes (it contains the application data bytes). The header bytes and the checksum byte are handled by the protocol layer application running in the network adapter.

This command shall return a '0' after the ECU response is received and the data and message length have been stored in the memory space indicated by DATARX and MSG_LENRX. The MSG_LEN and DATA are pointer variables so the receiving data can be stored in the same memory space. The network adapter shall allocate enough memory space to receive the maximum sized KWP message (255 data bytes, 260 bytes total with header and checksum).

SEND_MESSAGE	**KW2000 MESSAGE**		
ID=4	**Header Bytes**	**Format Byte** **Target Byte** **Source Byte** **Length Byte**	**-** **OPTIONAL** **OPTIONAL** **-**
MsgLen=n			
<DATA1> : <DATAn>	**<ServiceId>**	**<Service Name> Request Service Identifier**	
	<Parameter1> : <Parametern>	<List of parameters> = [<Parameter Name> : <Parameter Name>]	
	CS	**Checksum Byte**	

MsgLen is the number of the data byte in the KWP2000 message request.

6.6.11 Initialization

This API shall not support 5-baud or CARB initialization.

6.6.12 Tester Present

After a successful START_COMMUNICATION API call and in lack of any other KWP message being called from the application software, the network adaptor shall continuously send a TesterPresent message out before (90%*P3) milliseconds has expired. This is to keep the communication link from expiring according to ISO14230-3 Section 6.4.

Chapter
7

7 VSI Emulation

The DPA III Plus/V features a VSI (Vehicle Serial Interface) emulator and has the ability to act like a VSI box, and can be used to run VSI legacy software both old and new.

This tool features a "pass-through" mode that allows the DPA to emulate the VSI. box.. In addition to VSI support, the DPA III+/V also can support CAN and J1850 channels, and can be operated on Windows 95, 98, 00, NT, and API. For further instruction on the VSI programs and capablilities, please consult the Vehicle Serial Interface for Class 2 manual.

Appendix A

A DEFINES AND STRUCTURES

```
/**************************************/
/*                                       */
/*  Dearborn Group, Copyright (c) 1999        */
/*  Dearborn Protocol Adapter                 */
/*                                  */
/*                                      */
/********************************** ***/
```

A.1 Defines

Number of mailboxes

```
/#define NumberOfCANMailBoxs            16
#define NumberOfJ1708MailBoxs           32

/* max size of data buffer for mailbox */
#define MAILBOX_BUFFER_SIZE             2048
#define MaxBufferSize                   2048
```

Maximum size of check datalink string

```
#define MAX_CHECKDATALINK_SIZE   80
#ifndef CALLBACK
#define CALLBACK    _huge _pascal
#endif
```

Update transmit mailbox flag parameter definitions

```
#define UPDATE_DATA_LOCATION          0x80      // update data location
#define TRANSMIT_IMMEDIATE      0x40   // transmit immediately
#define UPDATE_DATA_COUNT       0x20   // update data count
#define UPDATE_BROADCAST_TIME         0x10      // update broadcast time
#define UPDATE_BROADCAST_COUNT        0x08      // update broadcast count
#define UPDATE_TIME_STAMP       0x04   // update time
#define UPDATE_ID               0x02   // update ID / (MID-PID-Priority)
```

```
#define UPDATE_DATA                  0x01    // update data
```

Inihibit flags

```
#define TimeStamp_Inhibit                     0x20
#define ID_Inhibit                   0x40
#define Data_Inhibit                 0x80
```

A.2 Typedefs

Enumerations for *CommPortType*

```
typedef enum CommPortType
{
  eComm1,
  eComm2,
  eComm3,
  eComm4
} CommPortType;
```

Enumerations for *BaudRateType*

```
typedef enum BaudRateType
{
  eB9600,
  eB19200,
  eB38400,
  eB57600,
  eB115200
} BaudRateType;
```

Enumerations for *ProtocolType*

```
typedef enum ProtocolType
{
  eISO9141,
  eJ1708,
  eJ1850,
  eJ1939,
  eCAN = 3
} ProtocolType;
```

Enumerations for *ReturnStatusType*

```
typedef enum ReturnStatusType
{
  eNoError,
  eDeviceTimeout,
  eProtocolNotSupported,
  eBaudRateNotSupported,
  eInvalidBitIdentSize,
  eInvalidDataCount,
  eInvalidMailBox,
  eNoDataAvailable,
  eMailBoxInUse,
  eMailBoxNotActive,
  eMailBoxNotAvailable,
  eTimerNotSupported,
  eTimeoutValueOutOfRange,
  eInvalidTimerValue,
  eInvalidMailBoxDirection,
  eSerialIncorrectPort,
  eSerialIncorrectBaud,
  eSerialPortNotFound,
  eSerialPortTimeout,
  eCommLinkNotInitialized,
  eAsyncCommBusy,
  eSyncCommInCallBack,
  eAsyncCommandNotAllowed,
  eSyncCommandNotAllowed,
  eLinkedMailBox,
  eInvalidExtendedFlags,
  eInvalidCommand
} ReturnStatusType;
```

Enumerations for MailBoxDirectionType

```
typedef enum MailBoxDirectionType
{
  eRemoteMailBox,      /* used for receiving data from link */
  eDataMailBox         /* used for sending data across link */
} MailBoxDirectionType;
```

Enumerations for TransmitMailBoxType

```
typedef enum TransmitMailBoxType
{
  eResident,          /* MailBox remain active and can be used again  */
  eRelease            /* MailBox is automatically unloaded after used          */
} TransmitMailBoxType;
```

Enumerations for DPA errors

```
typedef enum DataLinkCANErrorCodeType
{
  eBusOff = 1,
  eCanOverRun,
  eErrorSendingAsync
} DataLinkCANErrorCodeType;

typedef enum LinkType
{
  eNoLink,
  eLinkHead,
  eLinkBody,
  eLinkTail
} LinkType;
```

A.3 Structures

Structure for InitCommLink

```
typedef struct
{
  unsigned char   bCommPort;
  unsigned char   bBaudRate;
} CommLinkType;
```

Structure for DPA error

```
typedef struct
{
  unsigned char   bProtocol;
  unsigned char   bErrorCode;
} DataLinkErrorType;
```

Structure for PC copy of MailBox

```
typedef struct
{
  unsigned char   bProtocol;
  unsigned char   bActive;
  unsigned char   bBitIdentSize;
  unsigned char   bMailBoxNumber;
  int             iVBBufferNumber;
  unsigned long                           dwMailBoxIdent;
  unsigned char                           bMailBoxDirectionData;
  unsigned char                           bTimeAbsolute
  void (CALLBACK *pfApplicationRoutine)(void *);
  void (CALLBACK *pfMailBoxReleased)(void *);
  unsigned long   dwTimeStamp;
  unsigned long   dwTransmitTimeStamp;
  unsigned long   dwBroadcastTime;
  int             iBroadcastCount;
  unsigned int    wDataCount;
  unsigned char   bData[MAILBOX_BUFFER_SIZE];
  unsigned char   bTransparentUpdateEnable;
  unsigned char                         bTimeStampInhibit;
  unsigned char   bIDInhibit;
  unsigned char   bDataCountInhibit;
  unsigned char   bDataInhibit;
  unsigned char   bLinkType;
  unsigned char  bLink;
  unsigned char  bPriority;
  unsigned char  bMID;
  unsigned char  bPID;
  unsigned char  bJ1708ExtendedDataMode;
  unsigned char  bJ1708ExtendedPtrMode;
  unsigned int   wJ1708ExtendedOffset;
  unsigned int   wJ1708ExtendedLength;
  unsigned char  bDataRequested;
  unsigned char  bReceiveFlags;
  unsigned char  bDataUpdated;
  void           *vpData;
  void           *vpUserPointer;
} MailBoxType;
```

bProtocol - the protocol selected for the mailbox
eISO9141 - ISO9141
eJ1708 - J1708
eJ1850 - J1850
eJ1939 - J1939
eCAN - CAN

bActive - Active or *in-use* flag

bBitIdentSize - Specifies the length of the MailBox identifier, (max 32). For CAN, that may be **11** or **29**.

bMailBoxNumber - Handle used for communication between the PC and DPA.

iVBBufferNumber - Buffer number used for Visual Basic programming.

dwMailBoxIdent - MailBox identifier, (32bits maximum).

bMailBoxDirectionData - Used to identify MailBox direction, (either *transmit* or *receive*).

bTimeAbsolute - Flag for setting the timestamp function to absolute or relative time.
true = absolute
false = relative

pfApplicationRoutine - Address of callback routine. (NULL disables.)

pfMailBoxReleased - Address of callback routine. (NULL disables.)

dwTimeStamp - Time a message is received.

dwTransmitTimeStamp - Time a message is transmitted.

dwBroadcastTime - Specifies the time interval between broadcast messages.

iBroadcastCount - Specifies the number of times a broadcast message is to be sent.

wDataCount - Number of bytes per message (maximum of MAILBOX_BUFFER_SIZE bytes).

bData[MAILBOX_BUFFER_SIZE] - Temporary holding buffer for message data.

bTransparentUpdateEnable - Flag for enabling a transparent update.

bTimeStampInhibit - Flag for removing timestamp from a receive data message.

bIDInhibit - Flag to remove the MailBox identifier in a received data message.

bDataCountInhibit - Flag for removing data in a received data message.

bDataInhibit - Flag for removing data in receive data message.

bLinkType - J1708 link type for multiple PIDs.

bLink - J1708 link for multiple PIDs.

bPriority - J1708 priority.

bMID - J1708 MID

bPID - J1708 PID

bDataRequested - Flag indicating that data has been requested from the DPA.

bReceiveFlags - Flag indicating that requested data has arrived.

bDataUpdated - Flag indicating that data has been updated.

***vpData** - Address of user's copy of message data.

***vpUserPointer** - A user-defined pointer (typical use: for a "this" pointer).

Structure for initializing the DPA II

```
typedef struct
{
 unsigned char              bProtocol;
 unsigned char              bParam0;
 unsigned char              bParam1;
 unsigned char              bParam2;
 unsigned char              bParam3;
 unsigned char              bParam4;
 void (CALLBACK *pfDataLinkError)(MailBoxType *,
DataLinkErrorType *);
 void (CALLBACK *pfTransmitVector)(MailBoxType *);
 void (CALLBACK *pfReceiveVector)(void);
}InitDataLinkType;
```

Structure for *LoadMailBox*

Key elements for setting up specific mailbox types are as follows:

	J1708 RCV	J1708 XMIT	CAN RECV	CAN XMIT
typedef struct {				
unsigned char bProtocol;	X	X	X	X
MailBoxType *pMailBoxHandle;	X	X	X	X
unsigned char bRemote_Data;	X	X	X	X
unsigned char bTransportType;			X	X
unsigned char bResidentOrRelease;		X		X
unsigned char bBitIdentSize;			X	X
unsigned long dwMailBoxIdent;			X	X
byte bCTSSource;			X	X
unsigned long dwMailBoxIdentMask;			X	
byte bFilterType;	X		X	
unsigned char bTransparentUpdateEnable;	X		X	
unsigned char	X	X	X	X

bTimeStampInhibit;				
unsigned char bIDInhibit;	X		X	
unsigned char bDataCountInhibit;	X		X	
unsigned char bDataInhibit;	X		X	
unsigned char bTimeAbsolute;		X		X
unsigned long dwTimeStamp;		X		X
unsigned long dwBroadcastTime;		X		X
int iBroadcastCount;		X		X
unsigned char bLinkType;	unused			
unsigned char bLink;	unused			
unsigned char bPriority;		X		
unsigned char bMID;	X	X		
unsigned char bPID;	X	X		
unsigned char bMIDMask;	X			
unsigned char bPIDMask;	X			
byte bExtendedPrtMode;	X	X	X	X
word wExtendedOffset;	X	X	X	X
void (CALLBACK *pfApplicationRoutine)(MailBoxType *);	X	X	X	X
unsigned char bMailBoxReleased;	unused			
void (CALLBACK *pfMailBoxReleased)(MailBoxType *);		X		X
void *vpUserPointer;	X	X	X	X
unsigned int wDataCount;	X	X	X	X
void *vpData;	X	X	X	X
} LoadMailBoxType;				

bProtocol - The protocol selected for a particular mailbox.

eISO9141 - ISO9141
eJ1708 - J1708
eJ1850 - J1850
eJ1939 - J1939
eCAN - CAN

***pMailBoxHandle** - Address of MailBox handle returned if a load was successful.

bRemote_Data - Used to identify the MailBox direction:
0 - Receive
1 - Transmit

bTransportType - Transport type to be used:
0 - RTS/CTS (Point-to-point)
1 - BAM (Broadcast)

bResidentOrRelease - Specifies transmit mailbox: *Resident* or *Release*.
0 - Resident (the mailbox is permanent)
1 - Release (the mailbox is used to transmit once and is then deleted)

bBitIdentSize - Specifies the length of the MailBox identifier (32 bits maximum)

dwMailBoxIdent - MailBox Identifier (32 bits maximum)

bCTSSource - Destination address for RTS Transport

dwMailBoxIdentMask - MailBox Identifier mask (*1* = match, *0* = don't care)

bFilterType - Identifies a filter a block or pass type.
0 - Pass (pass only messages that match the filter)
1 - Block (do not pass messages that match the filter)

bTransparentUpdateEnable - Flag for enabling a transparent update of data.
TRUE = ON
FALSE = OFF

bTimeStampInhibit - Flag for removing the timestamp from a receive data message.
TRUE = ON
FALSE = OFF

bIDInhibit - Flag for removing the MailBox identifier from a receive data message.
TRUE = ON
FALSE = OFF

bDataCountInhibit - Flag for removing the data count from a receive data message.
TRUE = ON
FALSE = OFF

bDataInhibit - Flag for removing the data from a receive data message.
TRUE = ON
FALSE = OFF
bTimeAbsolute - Flag for setting the timestamp format to absolute or relative time.
TRUE = absolute
FALSE = relative

dwTimeStamp - Specifies a time (or delay) for the first message to be transmitted from the DPA.

dwBroadcastTime – Identifies the time interval between broadcast messages.

iBroadcastCount - Specifies the number of times broadcast message should be sent.

bLinkType - (Unused.)

bLink - (Unused.)

bPriority - J1708 priority.

bMID - J1708 MID.

bPID - J1708 PID.

bMIDMask - J1708 MID mask.

bPIDMask - J1708 PID mask.

bExtendedPrtMod - Specifies whether the scratch pad should be used for data.
0 - no
1 - yes

wExtendedOffset - The location of data in the buffer (scratch pad).

(CALLBACK *pfApplicationRoutine)(MailBoxType *) - a Mailbox Release flag to keep the active flag current.

bMailBoxReleased - (Unused.)

(CALLBACK *pfMailBoxReleased)(MailBoxType *) - Address of a Mailbox release callback routine.

***vpUserPointer** - A user-defined pointer.

wDataCount - Number of bytes per message (a maximum of MAILBOX_BUFFER_SIZE bytes).

***vpData** - The address of message data.

Structure for timer interrupts

```
typedef struct
{
  unsigned long         dwTimeOut;
  void (CALLBACK *pTimerFunction)(unsigned long);
} EnableTimerInterruptType;
```

Structure for Transport Protocol

```
typedef struct{
  byte    bProtocol;
  word    iBamTimeOut;
  word    iBAM_BAMTXTime;
  word    iBAM_DataTXTime;
  word    iRTS_Retry;
  word    iRTS_RetryTransmitTime;
  word    iRTS_TX_Timeout;
  word    iRTS_TX_TransmitTime;
  word    iRTS_RX_TimeoutData;
  word    iRTS_RX_TimeoutCMD;
  word    iRTS_RX_CTS_Count;
  word    iRTS_TX_CTS_Count;
} ConfigureTransportType;
```

A.4 Function Prototypes

ifdef __cplusplus

```
extern "C" {
        ReturnStatusType far InitCommLink
                (CommLinkType *CommLinkData);
        ReturnStatusType far RestoreCommLink
                (void);
        ReturnStatusType far InitDataLink
                (InitDataLinkType *InitDataLinkData);
        ReturnStatusType far CheckLock
                (char *szSearchString, unsigned char *bFound);
        ReturnStatusType far CheckDataLink
                (char *cVersion);
        ReturnStatusType far LoadDPABuffer
                (unsigned char *bData, unsigned int wLength, unsigned int wOffset);
        ReturnStatusType far ReadDPABuffer
                (unsigned char *bData, unsigned int wLength, unsigned int wOffset);
        ReturnStatusType far LoadMailBox
                (LoadMailBoxType *pLoadMailBoxData);
        ReturnStatusType far TransmitMailBox
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far TransmitMailBoxAsync
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far UpdateTransMailBoxData
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far UpdateTransMailBoxDataAsync
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far UpdateTransmitMailBox
                (MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);
        ReturnStatusType far UpdateTransmitMailBoxAsync
                (MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);
        ReturnStatusType far ReceiveMailBox
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far UnloadMailBox
                (MailBoxType *pMailBoxHandle);
        ReturnStatusType far LoadTimer
```

```
                (unsigned long dwTime);
        ReturnStatusType far EnableTimerInterrupt
                (EnableTimerInterruptType *pEnableTimerInterruptData);
        ReturnStatusType far SuspendTimerInterrupt
                (void);
        ReturnStatusType far RequestTimerValue
                (unsigned long *dwTimerValue);
        }
#else
```

APPENDIX
B

B FILTERS (MASKS) AND CAN BIT-TIMING REGISTERS

B.1 Filters (Masks) for CAN

The DPA uses filters (or masks) to eliminate messages at the hardware level, to help restrict the load on your application. This is accomplished using parameters in the *LoadMailBoxType* structure:

dwMailBoxIdent	The **identifier** for the mailbox
dwMailBoxIdentMask	The **mask** for the mailbox
bFilterType	The **filter type** for the mailbox.

The filters work like those used in CAN filtering: bit-by-bit, across the identifier and mask fields. The identifier (ID) field determines whether the bit value is *1* or *0*. The Mask field determines whether the bit is "care" (*1*) or "don't care" (*0*). Thus, all ID bits matched with Mask bits equal to 1 are set to their respective values (0 or 1). All ID bits matched with Mask bits equal to *0* are set as "don't care"; a "don't care" bit is represented with an **X**.

The filter type determines whether all the messages that *satisfy* the result are to be saved (pass) or whether all the messages that *do not* match the result are to be saved (block).

The following examples each use only one data byte, but the same filtering principles would be applied to additional bytes in the dialog boxes.

Example A:

	Hex Value	Byte							
ID	28H	0	0	1	0	1	0	0	0
Mask	E0H	1	1	1	0	0	0	0	0
Result	20H - 3FH	0	0	1	X	X	X	X	X

The result is that all messages from 20H to 3FH will satisfy the filter or trigger condition.

Example B:

To create a single ID filter or trigger, a Mask of *FFH* should be used.

	Hex Value	Byte							
ID	28H	0	0	1	0	1	0	0	0
Mask	FFH	1	1	1	1	1	1	1	1
Result	28H	0	0	1	0	1	0	0	0

The result is that only the 28H value will satisfy the filter or trigger condition.

B.2 Filters (masks) for J1708

The DPA uses filters (or masks) to eliminate messages at the hardware level, to help restrict the load on your application. This is accomplished using parameters in the *LoadMailBoxType* structure:

dwMID	The **identifier** for the MID mailbox
dwPID	The **identifier** for the PID mailbox
dwMIDMask	The **mask** for the MID in mailbox
dwPIDMask	The **mask** for the PID mailbox
bFilterType	The **filter type** for the mailbox.

The filters work like those used in CAN filtering: bit-by-bit, across the identifier and mask fields. The identifier (ID) field determines whether the bit value is *1* or *0*. The Mask field determines whether the bit is "care" (*1*) or "don't care" (*0*). Thus, all ID bits matched with Mask bits equal to *1* are set to their respective values (0 or 1). All ID bits matched with Mask bits equal to *0* are set as "don't care"; a "don't care" bit is represented with an **X**.

The filter type determines whether all the messages that *satisfy* the result are to be saved (pass) or whether all the messages that *do not* match the result are to be saved (block).

The following examples each use only one data byte, but the same filtering principles would be applied to additional bytes in the dialog boxes.

Example A:

	Hex Value	Byte							
ID	28H	0	0	1	0	1	0	0	0
Mask	E0H	1	1	1	0	0	0	0	0
Result	20H - 3FH	0	0	1	X	X	X	X	X

The result is that all messages from *20H* to *3FH* will satisfy the filter or trigger condition.

Example B:

To create a single ID filter or trigger, a Mask of *FFH* should be used.

	Hex Value	Byte							
ID	28H	0	0	1	0	1	0	0	0
Mask	FFH	1	1	1	1	1	1	1	1
Result	28H	0	0	1	0	1	0	0	0

The result is that only the *28H* value will satisfy the filter or trigger condition.

B.3 CAN Bit-timing registers

The CAN Bit Timing Registers (BTRs) are the registers that determine bus speed. They also set up sampling and re-synchronization of the CAN controller. Different networks use different parameters for these values. In fact, there are several combinations that can all generate the same bus speed.

The DPA uses an 80C196CA microcontroller with an 82527 CAN controller on-board. The crystal is 16 MHz. The following is a brief overview of the bit timing registers. Please reference the documents listed in section 1.3 for more information.

The CAN controller has two registers (BTR0 and BTR1) used to determine bus speed. Common values for networks are as follows::

Network	Bus Speed	BTR0	BTR1
J1939	250 Kbps	0x41	0x58
DeviceNet	125 kbps 250 Kbps 500 Kbps	0x03 0x01 0x00	0x1C 0x1C 0x1C
SDS	1 M bps	0x00	0x14

Register BTR0:

7	6	5	4	3	2	1	0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0

SJW1:0 **Synchronization Jump Width**
Defines the maximum number of time quanta by which re-synchronization can modify tseg1 and tseg2.

BRP0:5 **Baud-rate Prescaler**
Defines the length of one time quantum (tq).

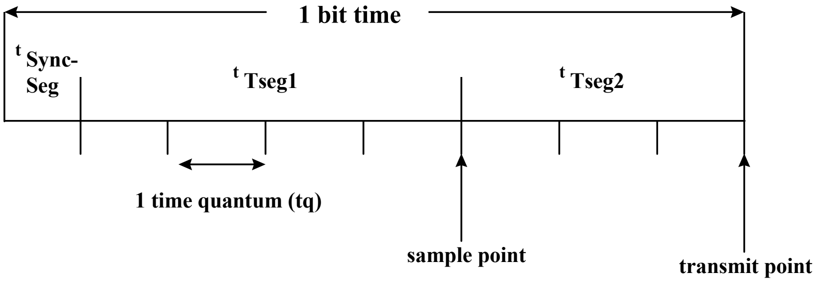
Register BTR1:

7	6	5	4	3	2	1	0
SPL	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

SPL **Sampling Mode**
Specifies the number of samples when determining a valid bit value:
1 = 3 samples, using majority logic
0 = 1 sample

TSEG2 **Time Segment 2**
Determines the length of time that follows the sample point within a bit time. (Valid values = 1-7.)

TSEG1 **Time Segment 1**
Defines the length of time that precedes the sample point within a bit time. (Valid values = 2-15.)



The bus speed can be calculated as follows:

$$\text{Bus Frequency} = \frac{F_{OSC}}{2 \times (BRP + 1) \times (3 + TSEG1 + TSEG2)}$$

F_{OSC} = Input Clock Frequency

If:

BTR0 = 01	SJW = 0; BRP = 1
BTR1 = 1C	SPL = 0; TSEG2 = 1; TSEG1 = 12
F_{OSC} = 16MHz	

$$\text{Bus Frequency} = \frac{16MHz}{2 \times (1 + 1) \times (3 + 1 + 12)} = 250{,}000$$

APPENDIX

C

C DRIVER SUMMARY

DPAM16

Description: The DPAM16 interface should be used to develop 16-bit Windows applications. This interface supports one ISA and one serial DPA simultaneously. This interface is recommended for all new code development.

Operating system(s): Windows 3.1, Windows 95, Windows 98
Interface: Multiple DPA
DPA(s) supported: Serial, ISA
DLL name: DPAM16.DLL
Borland Import LIB: DPAM16B.LIB
Microsoft Import LIB: DPAM16.LIB
'C' header files: DPAM16.H, DPA6XT.H

DPA16

Description: The DPA16 interface should be used to develop 16-bit Windows applications. This interface supports one DPA at a time. It will support either the ISA or the serial DPA.

Operating system(s): Windows 3.1, Windows 95, Windows 98
Interface: Single DPA
DPA(s) supported: Serial, ISA
DLL name: DPA16.DLL
Borland Import LIB: DPA16B.LIB
Microsoft Import LIB: DPA16.LIB
'C' header files: DPA16.H, DPA6XT.H

DPAM32

Description: The DPAM32 interface should be used to develop 32-bit Windows applications. This interface supports multiple ISA and multiple serial DPA's simultaneously. This interface is recommended for all new code development.

Operating system(s): Windows 95, Windows 98, Windows NT
Interface: Multiple DPA

DPA(s) supported: Serial, ISA
DLL name: DPAM32.DLL
Borland Import LIB: DPAM32B.LIB
Microsoft Import LIB: DPAM32.LIB
'C' header files: DPAM32.H, DPA6XT.H

DPA32

Description: The DPA32 interface should be used to develop 32-bit Windows applications. This interface supports one DPA at a time. This interface is recommended only to support code that was developed for previous versions of the DPA drivers.
Operating system(s): Windows 95, Windows 98, Windows NT
Interface: Single DPA
DPA(s) supported: Serial, ISA
DLL name: DPA32.DLL
Borland Import LIB: DPA32B.LIB
Microsoft Import LIB: DPA32.LIB
'C' header files: DPA32.H, DPA6XT.H

DPAS16 (Static Library)

Description: The DPAS16 interface should be used to develop DOS applications. This interface supports one serial DPA at a time.
Operating system(s): DOS
Interface: Single DPA
DPA(s) supported: Serial
DLL name: N/A
Borland LIB: DPAS16B.LIB
Microsoft LIB: DPAS16.LIB
'C' header files: DPA6X.H, DPA6XT.H

DPAI16 (Static Library)

Description: The DPAI16 interface should be used to develop DOS applications. This interface supports one ISA DPA at a time.
Operating system(s): DOS
Interface: Single DPA
DPA(s) supported: ISA
DLL name: N/A
Borland LIB: DPAI16B.LIB
Microsoft LIB: DPAI16.LIB
'C' header files: DPA6X.H, DPA6XT.