Dearborn Group
Technology

DEARBORN PROTOCOL ADAPTER FAMILY USER'S MANUAL

Version 1.26.1

Includes the following products:

DPA II *Plus*
DPA III (ISA and PC/104 versions)
DPA III *Plus* (serial versions)
DPA 4 (USB version)
DPA RF (wireless version)

© 2003 Dearborn Group Inc.

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

This document is copyrighted by the Dearborn Group Inc. Permission is granted to copy any or all portions of this manual, provided that such copies are for use with the product provided by the Dearborn Group, and that the name "Dearborn Group Inc." remain on all copies as on the original.

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.

DPA is a trademark of Dearborn Group Inc. CAN is a trademark of Bosch Inc. Other products are trademarks of their respective manufacturers.

Recent manual revision history:

Aug. 18, 2004	(added 'redundant filtering' information)
Jul. 7, 2004	(revise RF hardware section for clarity and accuracy)
Apr. 20, 2004	(revise StoreDataLink information for stand-alone program release)
Feb. 2, 2004	(corrections to driver installation instructions)
Nov. 20,2003	(miscellaneous edit corrections)
Sep. 15, 2003	(add DPA RF installation instructions)
Aug. 12, 2003	(add USB installation instructions, new DPA 4 functions, Index)

Table of Contents

1.0 INTRODUCTION 1
1.1 Documentation organization 2
1.2 Technical support 2
1.3 Related documents 3
2.0 HARDWARE: GETTING STARTED 5
2.1 Checking package contents 5
2.2 Software 5
2.3 Power / network connections 6
2.4 RS-232 Hardware 7
2.4.1 Hardware installation and setup 7
2.4.2 Connection to the PC 8
2.4.3 Checking communication 9
2.5 ISA HARDWARE 10
2.5.1 ISA card hardware installation and setup 10
2.5.2 Setting the jumpers 10
2.5.3 Power / network connection to the DPA III - ISA 10
2.5.4 Installing the ISA card 11
2.5.5 Checking communication 11
2.6 PC/104 hardware 11
2.6.1 Setting the jumpers 12
2.6.2 Power / network connection to the DPA III – PC/104 12
2.6.3 Installing the PC/104 card 13
2.6.4 Checking communication 13
2.7 USB Hardware 13
2.7.1 Connection to the PC 13
2.7.2 Hardware installation and setup 14
2.7.3 Power / network connection to the DPA 4 USB 14
2.7.4 Checking communication 15
2.8 RF Hardware 16
2.8.1 Connect the Base Station 16
2.8.2 Connecting the Remote Station 16
2.8.3 Installation and setup 16
2.8.4 Configuring the OEM software tool 17
2.8.5 Checking communication 17
3.0 FUNCTIONAL OVERVIEW 18
3.1 Timer 18
3.2 Transmit Mailbox 20
3.3 Receive Mailbox 21
3.4 I/O Buffer (host scratch pad) 22
4.0 API OVERVIEW 24

4.1 Compilers 24
4.2 The API Choices 25
4.3 The DPA API functions list 25
4.3.1 System functions 26
4.3.2 Data-link configuration functions 27
4.3.3 Message handling functions 27
4.3.4 Timer functions 28
4.3.5 Buffer (host scratch pad) functions 28
4.4 DPA API function descriptions (alphabetical order) 29
4.4.1 CheckDataLink 29
4.4.2 ConfigureTransportProtocol 31
4.4.3 DisableDataLink (DPA 4 Only) 34
4.4.4 EnableTimerInterrupt 35
4.4.5 InitCommLink 37
4.4.6 InitDataLink 39
4.4.7 InitDPA (Windows Only) 47
4.4.8 InitPCCard 50
4.4.9 LoadDPABuffer 52
4.4.10 LoadMailBox 53
4.4.11 LoadTimer 65
4.4.12 PauseTimer 66
4.4.13 ReadDataLink (DPA 4 Only) 67
4.4.14 ReadDPABuffer 69
4.4.15 ReadDPAChecksum 70
4.4.16 ReceiveMailBox 71
4.4.17 ResetDPA 73
4.4.18 RestoreCommLink 75
4.4.19 RestoreDataLink (DPA 4 Only) 76
4.4.20 RestoreDPA (Windows Only) 77
4.4.21 RestorePCCard 78
4.4.22 RequestTimerValue 79
4.4.23 ResumeTimer 80
4.4.24 SetBaudRate (32-bit Windows Only) 81
4.4.25 StoreDataLink (DPA 4 Only) 82
4.4.26 SuspendTimerInterrupt 84
4.4.27 TransmitMailBox 85
4.4.28 TransmitMailBoxAsync 86
4.4.29 UnloadMailBox 87
4.4.30 UpdateReceiveMailBox 88
4.4.31 UpdateReceiveMailBoxAsync 89
4.4.32 UpdateTransMailBoxData 90
4.4.33 UpdateTransMailBoxDataAsync 91
4.4.34 UpdateTransmitMailBox 92
4.4.35 UpdateTransmitMailBoxAsync 94
4.5 KWP2000 DPA API function descriptions 95
4.5.1 INIT_KWP2000_DPA 95
4.5.2 RELEASE_KWP2000 95
4.5.3 SET_TIMING 96
4.5.4 SET_COM_PARAMETER_1 97
4.5.5 SET_COM_PARAMETER_2 98
4.5.6 SET_BAUDRATE 101
4.5.7 START_COMMUNICATION 102

4.5.8 STOP_COMMUNICATION *104*
4.5.9 GET_STATUS *105*
4.5.10 SEND_MESSAGE *105*
4.5.11 Initialization *106*
4.5.12 Tester Present *106*
4.6 USB DPA API function descriptions 107
4.6.1 RestoreUSBLink *107*
4.6.2 InitUSBLink *108*
5.0 VSI EMULATION 110
A DEFINES AND STRUCTURES 111
A.1 Defines 111
Number of mailboxes *111*
Maximum size of check data link string *111*
Update transmit mailbox flag parameter definitions *111*
Inhibit flags *112*
Update mailbox flag definitions *112*
A.2 Typedefs 112
Enumerations for CommPortType *112*
Enumerations for BaudRateType *113*
Enumerations for ProtocolType *113*
Enumerations for ResetType *113*
Enumerations for ReturnStatusType *114*
Enumerations for MailBoxDirectionType *117*
Enumerations for TransmitMailBoxType *117*
Enumerations for DPA errors *117*
A.3 Structures 118
Structure for InitCommLink *118*
Structure for DPA error *118*
Structure for USBLinkType *118*
Structure for PC copy of MailBox *119*
Structure for initializing the DPA Data link *121*
Structure for Timer Interrupts *125*
Structure for Transport Protocol *125*
A.4 Function Prototypes 126
ifdef __cplusplus *126*
B FILTERS (MASKS) AND CAN BIT-TIMING REGISTERS 127
B.1 Filters (Masks) for CAN 127
B.2 Filters (masks) for J1708 128
B.3 CAN Bit-timing registers 129
C DRIVER SUMMARY 133
D CONNECTOR PINOUTS 135
I INDEX 137

Table of Figures

Figure 1: Functional overview 2
Figure 2: DPA II Plus and DPA III Plus RS-232 hardware 8
Figure 3: RS-232 DB9 connector pinout 9
Figure 4: Successful communication using CHECKDPA.EXE 9
Figure 5: DPA architecture "map" 18
Figure 6: Timer "map" 19
Figure 7: Transmit mailbox "map" 20
Figure 8: Receive mailbox "map" 21
Figure 9: I/O buffer "map" 22

1.0 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.

ISA card, **PC/104 card** and **USB Interface**: The *DPA III* ISA card, PC/104 card and USB interface support the CAN (J1939), J1708, and J1850 protocols.

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 data link messages without being polled.

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

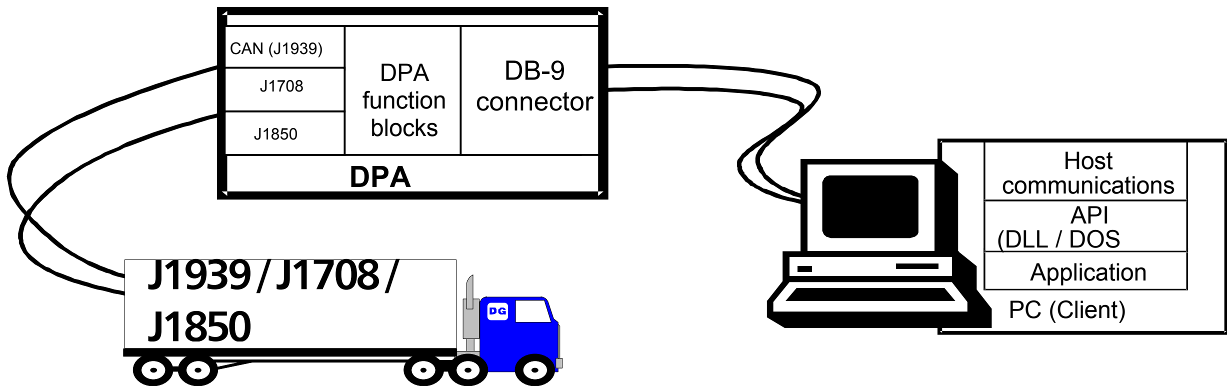


Figure 1: Functional overview

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 – **Hardware: Getting Started -** Instructions for proper installation and setup of the DPA.

Chapter 3 – **Functional Overview -** Overview of main DPA functions.

Chapter 4 – API Overview – Introduction to the Dynamic Link Library (DLL) used in creating programs to interface with the DPA.

Chapter 5 – VSI Emulation – Overview of VSI Emulation

Appendix A – Defines and Structures

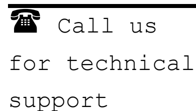
Appendix B – Filters (Masks) and CAN Bit-Timing Registers

Appendix C – Driver Summary

Appendix D – Connector Pinouts

1.2 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

PCA82C250 CAN Controller Interface data sheet	Data Book IC20

Society of Automotive Engineers (877)606-7323

Recommended Practice for a Serial Control and Communication Vehicle Network	J1939
Recommended Practice	J1708
Recommended Practice	J1587
Recommended Practice	J1850

International Standards Organization 41 22 749 01 11

Description	Document
Road Vehicles - Interchange of Digital Information – Controller Area Network for high speed communications Amendment 1	ISO 11898

Technology and Maintenance Council (703) 838-1763

Description	Document
TMC's RP1210 Windows™ Communication Application Program Interface (API)	RP1210A

Dearborn Group (248)-488-2080

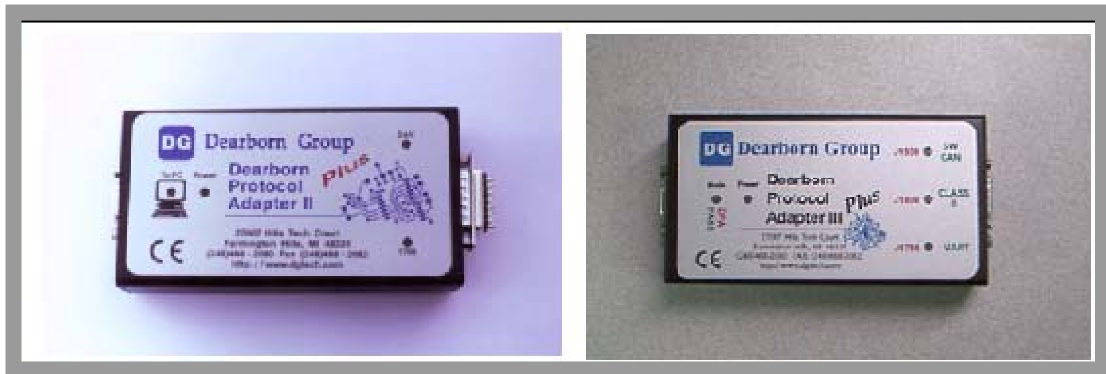
Description	Document
Description of how to use the various Dearborn Group INI files.	DG INI useage.doc

CHAPTER 2

2.0 HARDWARE: GETTING STARTED

IMPORTANT: please read this section before using the DPA!

Before using your DPA 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.



2.1 Checking package contents

Your DPA package should include the following items.

- DPA hardware unit
- *DPA Family User's Manual* (this document)
- RS-232 straight-through cable (RS-232 option only)
- USB cable (USB option only)
- 15-pin network connector
- Dearborn Group DPA Driver Installation CD (DPA Drivers, DPA RP1210a Drivers, DPA Manual, Sample RP1210 application and Adobe Acrobat Reader)

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.

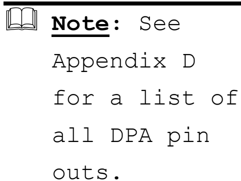
Driver installation
The DPA drivers are provided on the Installation CD. The DPA drivers are installed by inserting the disk into your PC's CD-rom drive. Using Windows Explorer, navigate to the *DPA8.xx Drivers* folder and select and open (run) the file named `dgdpa.exe`. Follow the instructions on the screen.

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 Power / network connections

Power and network connections are made through a female DP15 connector. There are several connector styles, please consult your DPA endplate and reference Appendix D for a list of all connector pinouts. The following is a list protocols supported.

Power and Ground Attachment (all DPAs)
A power supply of 9 – 32 V capable of supplying a minimum of 250 mA must be connected to the DPA. The ground on the DPA should be tied to the ground on both the power supply and the network. Often these connections are provided on the same connector as the network. (See Appendix D for pinouts.)



CAN / J1939 media attachment (all DPAs)
CAN connections (CANH and CANL) are connected through the on-board Philips 82C251 transceiver. The CAN shield is also provided with an RC filter. A termination resistor of 120 ohms may be added to the CAN link through the placement of a jumper between pins on the DB-15 connector. (See pinouts in Appendix D – *CAN Term 1* and *CAN Term 2*.)

J1708 media attachment (all DPAs)
The J1708 interface is connected as specified by the SAE J1708 document. The connections are referenced as **ATA+** and **ATA-.**

ALDL / GM UART media attachment (select DPAs)
The GM UART connections follow the GM ALDL standard. The DPA normally operates in slave mode, but Master mode can be selected by connecting a jumper between Master / Slave 1 and Master / Slave 2 pins on the connector. (See Appendix D for pinouts.)

J1850 VPW (Class 2, DCX) media attachment (select DPAs)
The J1850 VPW connection is made with a connection for J1850+. The ground on the DPA should be tied to the ground on both the power supply and the network. For J1850 VPW a Motorola SBCC controller is used with connection provided via J1850+ and J1850-. (See Appendix D for pinouts.)

J1850 PWM (SCP) media attachment (select DPAs)
For J1850 PWM, a Motorola SBCC controller is used with connection provided via J1850+ and J1850-. No termination is provided on the DPA, but an optional endplate with 360 ohm termination resistors is available from Dearborn Group. (See Appendix D for pinouts.)

ISO 9141 / KWP2000 media attachment (select DPAs)
K-line is connected on all DPAs, with L-line available on select versions. (See Appendix D for pinouts.)

ATEC media attachment (select DPAs)
ATEC connection follow the standard with ATEC Diag and ATEC Data provided on the connector. (See Appendix D for pinouts.)

2.4 RS-232 Hardware

2.4.1 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.

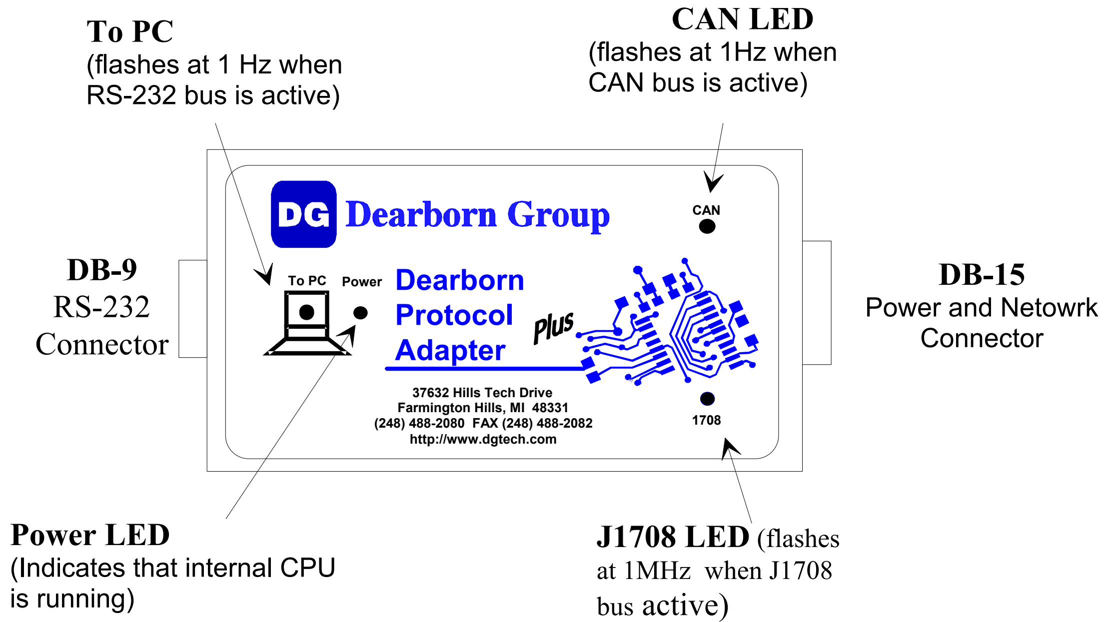


Figure 2: DPA II Plus and DPA III Plus RS-232 hardware

2.4.2 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

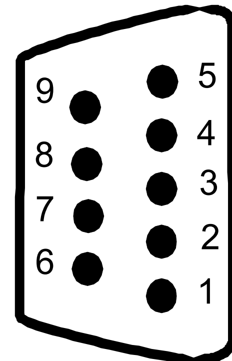


Figure 3: RS-232 DB9 connector pinout

2.4.3 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 2.2). 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.

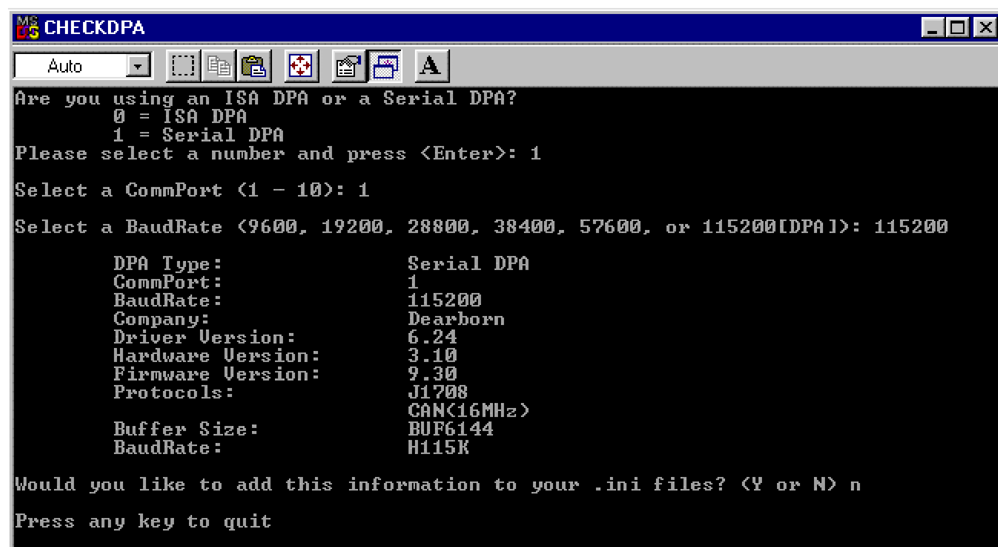


Figure 4: Successful communication using CHECKDPA.EXE

2.5 ISA HARDWARE

2.5.1 ISA card hardware installation and setup

The DPA requires the following hardware for operation:

- AT-compatible computer (or higher)
- Free ISA Bus Slot
- Interrupt 5, 7, or 10 available
- I/O Address Range 200, 220, 300, or 320 available

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.

2.5.2 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.

2.5.3 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	Data link Power (optional)
9	Data link 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.

See section 2.3 for more information about these protocols and their physical connections.

2.5.4 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.

2.5.5 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 2.2). 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.

2.6 PC/104 hardware

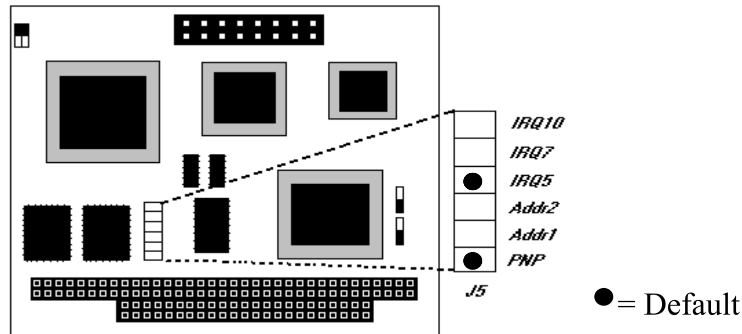
The DPA PC/104 requires the following hardware for operation:

- AT-compatible computer (or higher)
- Free PC/104 Bus Slot
- Interrupt 5, 7, or 10 available
- I/O Address Range 200, 220, 300, or 320 available

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.

2.6.1 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.

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

The pinout for the PC/104 card's 16-pin connector is as follows.

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

[1] To make this GM_UART node a master, jumper pin 3 to pin 4 in the connector.
[2] To terminate the CAN link with a 120 ohm resister, jumper pin 5 to pin 7 in the connector.

2.6.3 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.

2.6.4 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 2.2). 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.

2.7 USB Hardware

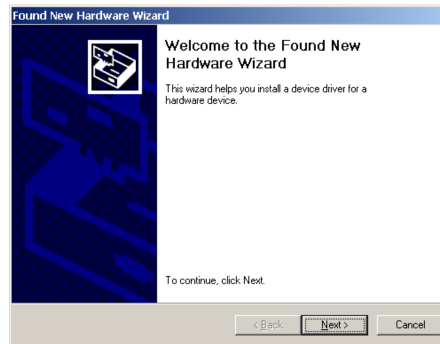
The DPA USB hardware has two connectors: a standard USB interface and a network / power connector. The network / power connector types are described Appendix D. See section 2.3 for more information about these protocols and their physical connections.

2.7.1 Connection to the PC

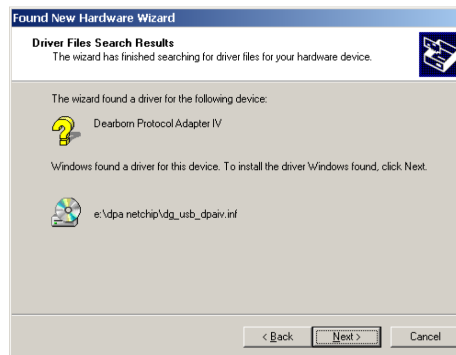
Once power is supplied to the DPA unit, the USB interface needs to be joined to a PC via a USB 2.0 A to B cable.

2.7.2 Hardware installation and setup

The first time that you have a powered DPA connected to the computer's USB port the operation system will prompted you for device drivers via a "Found New Hardware Wizard" box.



Click **Next** to locate the driver on the CDROM. If the file is auto-located, the box will appear below. Otherwise search the CDROM for the dg_usb_dpaiv.inf file. Once located, click **Next**.



The driver will be installed. Click **Finish** to close the box.

2.7.3 Power / network connection to the DPA 4 USB

The default power and network connection for the DPA 4 USB is the /T connector. It is defined Appendix D.

Note: Your DPA 4 requires driver Version 8.03 or higher.

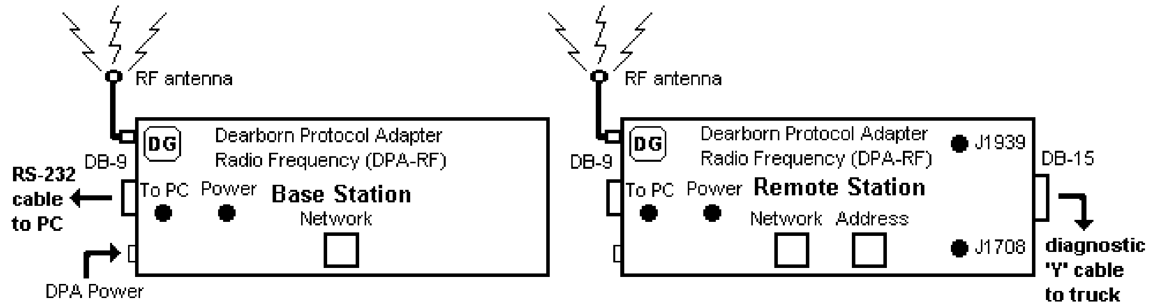
© 2004

2.7.4 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 2.2). 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.

2.8 RF Hardware

The DPA RF hardware consists of two hardware adapter units: the *Base Station* and the *Remote Station*. This section describes installation and setup of these units.



2.8.1 Connect the Base Station

Connect one end of the RS-232 serial cable to the Base Station. The other end of the cable is connected to a serial port on your PC. Connect the barrel connector of the AC power supply to the Base Station and plug the AC power supply into a standard electrical outlet. Once installed, the Base Station does not need to be moved. It uses an RF signal to communication with the Remote Station and will easily do so through brick or metal walls.

2.8.2 Connecting the Remote Station

Connect the DB-15 end of the "diagnostic cable" to the Remote Station. Connect the other end of the diagnostic cable to the vehicle diagnostic connector.

Note: The *Remote Station* receives its power from the vehicle through this cable.

2.8.3 Installation and setup

Your DPA RF package includes a *DG Installation CD-ROM* with the program files needed to operate and use this product. To install the program files, insert the *DG Installation CD-ROM* into your CD-ROM drive. If the computer using this CD-ROM is unable to automatically start the Installation program, use Windows Explorer to display the content of your CD-ROM drive. Double-click the `DPAInstall.EXE` file to run the Installation program.

2.8.4 Configuring the OEM software tool

The procedure for configuring your OEM software tool will vary, depending on the manufacturer. Refer to the manufacturer of the OEM tool's documentation for additional information.

2.8.5 Checking communication

Each DPA Base and Remote Station pair are configured at the factory with the same Network ID address so they can communicate exclusively with each other.

After using the DG DPA Installation CD to install the drivers (see sections 2.2 and 2.8.3), check to see that the DPA Remote Station is communicating with the DPA Base Station and the PC computer by running the **`CheckDPA.exe`** program which can be found in: **`C:\Program Files\Dearborn Group\8xDrivers\WIN32\CHECKDPA`** along with several code examples.

When you run the **`CheckDPA.exe`** program, it 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. If not, the program will report the problem it encountered.

The *Quick Start Card* that is included with your DPA product also contains common troubleshooting information. Copies of this card in PDF format are on our website as well as on your *DG Installation CD-ROM disc* in the *Manual* directory/folder.

CHAPTER

3

3.0 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.

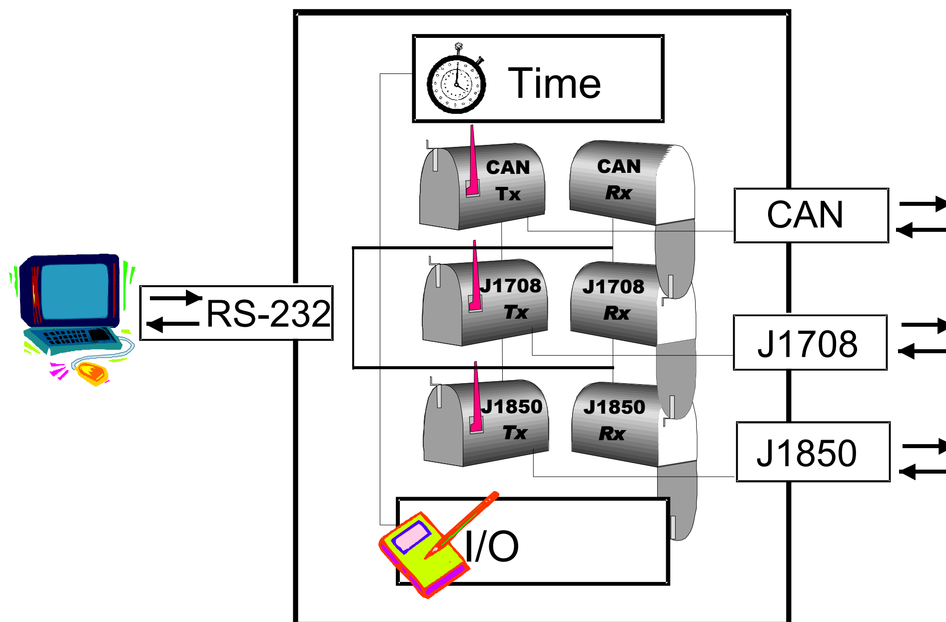


Figure 5: DPA architecture "map"

3.1 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 time stamping 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.

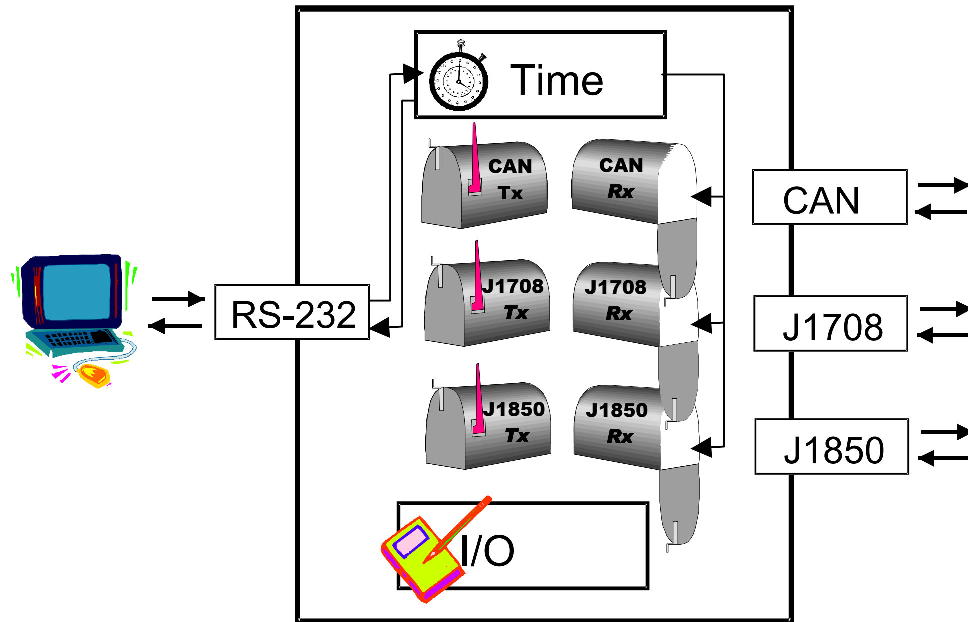


Figure 6: Timer "map"

3.2 Transmit Mailbox

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

- When the 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

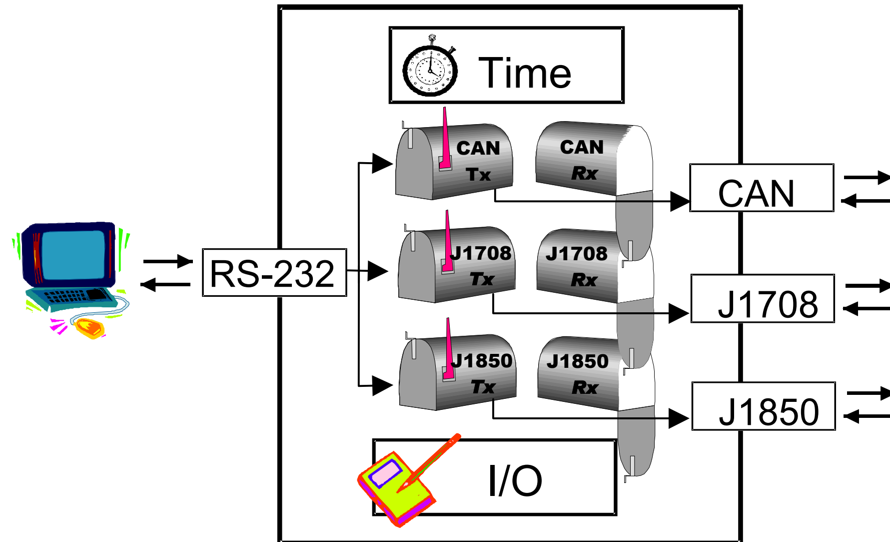


Figure 7: Transmit mailbox "map"

3.3 Receive Mailbox

A Receive Mailbox 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. (See Appendix B – *Filter Masks*)
- 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)

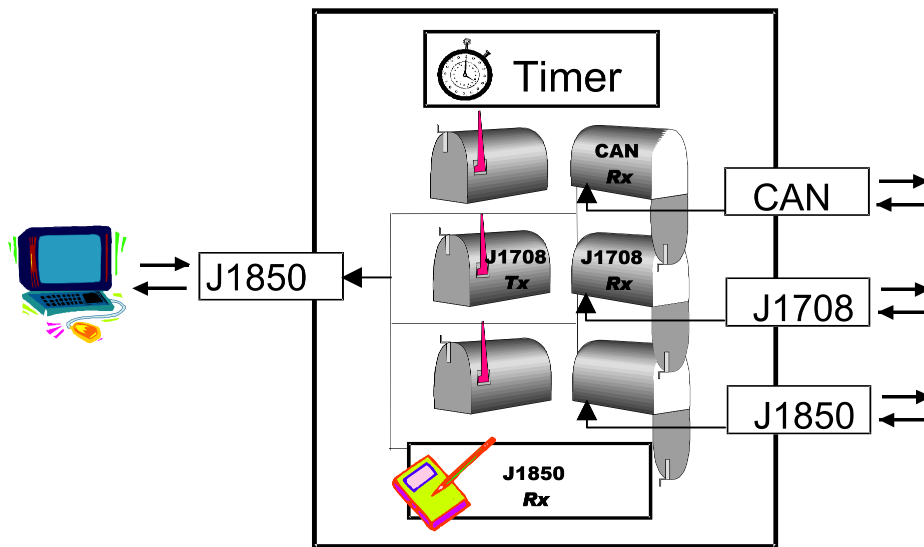


Figure 8: Receive mailbox "map"

3.4 I/O Buffer (host scratch pad)

The host scratch pad, or I/O 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 resources.

Note: The scratch pad in both the DPA 3 and DPA 4 is 6,144 bytes.

- a temporary message storage location
- redirection of mailbox data

storage for oversized messages (such as J1939 Transport Protocol messages)
concatenation of small messages

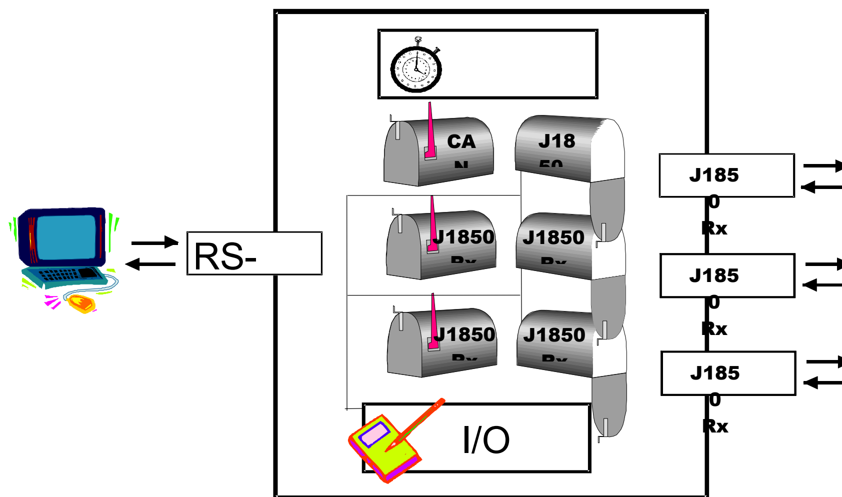


Figure 9: I/O buffer "map"

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).

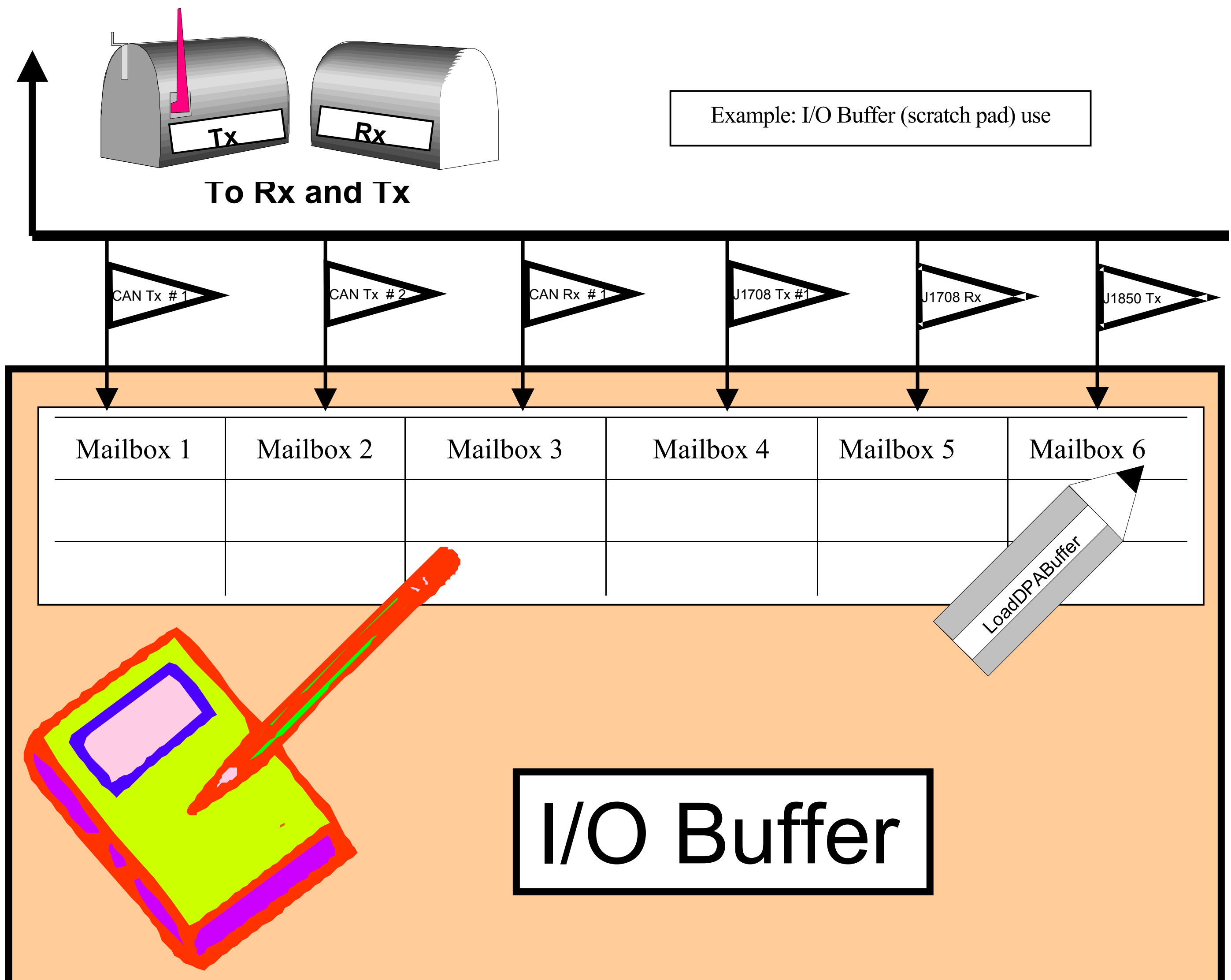
The J1939 transport layer also makes use of this buffer (scratch pad), to ensure that transport timing requirements are met. (Reference the *ConfigureTransportProtocol* function for further details.)

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:



4.0 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 this chapter. If you need further 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.

4.1 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.

4.2 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. The Multiple DPA interface is very similar to the single DPA interface. From a programmers' perspective, the only difference is that the multi-interface function calls take a "dpaHandle" parameter as the first parameter passed in. When an application opens a DPA for communication, 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 with multiple DPA's simultaneously. This interface is recommended for all new software development.

Example: Single DPA Driver

Status = CheckDataLink (VersionString);

Example: Multiple DPA Driver

Status = CheckDataLink (dpaHandle, VersionString);

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

4.3 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).

4.3.1 System functions

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

InitCommLink
Specifies and initializes communication between the PC and the serial DPA. For future development. Use of this command is discouraged. Use InitDPA instead.

InitPCCard
Specifies and initializes communication between the PC and the ISA/PC104 DPA. For future development. Use of this command is discouraged. Use InitDPA instead.

InitUSBLink
Specifies and initializes communiation between the PC and the USB DPA. For future development. Use of this command is discouraged. Use InitDPA instead.

RestoreDPA
Restores the communication port between the PC and the DPA to its previous (pre-*InitDPA*) state

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

RestorePCCard
Restores the communication port between the PC and the ISA/PC104 DPA to its previous (pre-*InitPCCard*) state.

RestoreUSBLink
Restores the USB port between the PC and the DPA to its previous (pre-*InitUSBLink*) state.

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

ReadDPAChecksum
Verifies the checksum of the DPA's Flash memory.

ResetDPA
Performs a low-level reset of the DPA or its communications.

SetBaudRate (Serial only)
Allows the calling application to command the DPA to run at a different baud rate.

4.3.2 Data-link configuration functions

InitDataLink
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.

ConfigureTransportProtocol
Allows the user to configure J1939 Transport Protocol characteristics.

DisableDataLink
Disables a datalink on power-up.

ReadDataLink
Reads the currently initialized datalink information for a given protocol

RestoreDataLink
Restores a datalink to its power-up parameters.

StoreDataLink
Stores the currently initialized datalink information to the power-up state for a given protocol.

4.3.3 Message handling functions

LoadMailBox
Opens (creates) mailboxes for the receiving and transmitting of messages.

TransmitMailBox
Sends messages, using previously opened mailboxes.

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

UpdateTransMailBoxData
Updates information in a previously opened broadcast mailbox.

UpdateTransMailBoxDataAsync
Updates information, from within a callback, in a broadcast mailbox.

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

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.

ReceiveMailBox
Retrieves the latest message from a specific mailbox.

UpdateReceiveMailBox
Updates the data count, data location, identifier, and identifier mask from a previously opened receive mailbox.

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

UnloadMailBox
Closes a previously opened mailbox.

4.3.4 Timer functions

LoadTimer
Sets the timer, for the time stamping of received and transmitted messages.

RequestTimerValue
Returns the current DPA timer value.

EnableTimerInterrupt
Enables a timer interrupt from the DPA, to call a user-supplied callback function.

SuspendTimerInterrupt
Disables a DPA timer interrupt.

PauseTimer
Pauses the timer and suspends transmits and timer callbacks.

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

4.3.5 Buffer (host scratch pad) functions

LoadDPABuffer
Loads data into the DPA's I/O buffer (internal scratch memory).

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

4.4 DPA API function descriptions (alphabetical order)

4.4.1 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

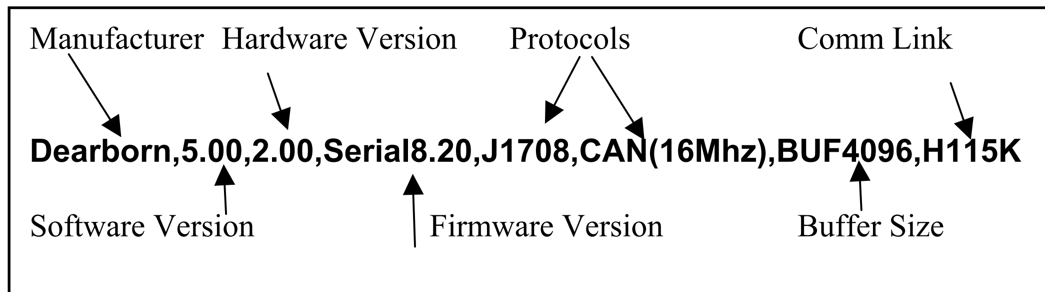
```
ReturnStatusType CheckDataLink (short dpaHandle,
     char *szVersion);
```

Prototype 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.



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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

void DisplayCheckDataLink(void);

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

4.4.2 ConfigureTransportProtocol

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

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

Prototype 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:

bTransportType Identifies session type: RTS/CTS or BAM
CTSSource Destination address for the RTS transport

Note: The J1939 data link must be initialized using the InitCommLink() function call with the following parameters passed.

bProtocol = J1939
bParam2 = 3
All undefined values will default to the J1939 recommended value.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

```
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 millisecond): 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 millisecond)

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

iRTS_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 millisecond)

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

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 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, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)

eDeviceTimeout
eCommLinkNotInitialized

<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 ConfigureTransportProtocol 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);
}
```

4.4.3 DisableDataLink (DPA 4 Only)

Function Disables a datalink on power-up.

Syntax
```
ReturnStatusType DisableDataLink (short dpaHandle,
     BYTE bProtocol);
```

Prototype dpam32.h or dpa32.h

Remarks *DisableDataLink* sets a datalinks power-up state to disabled. This command is only available on the DPA 4.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

bProtocol Identified the protocol that you would like disabled on power-up.

Return Value *DisableDataLink* returns **eNoError** on success.
In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eCommLinkNotInitialized

Example:

This example disables the J1850 datalink on power-up.

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


void DisableJ1850Settings(void)
{
      ReturnStatusType          InitStatus,DisableStatus,RestoreStatus;
      short                     dpaHandle;


      InitStatus = InitDPA(&dpaHandle, 601);

      if(InitStatus == eNoError)
      {
            DisableStatus =  DisableDataLink(dpaHandle, eJ1850);

            RestoreStatus = RestoreDPA(dpaHandle);
      }
}
```

4.4.4 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 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink 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** on success.

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eTimeoutValueOutOfRange
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

```
#include <windows.h>
#include <stdio.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);
}
```

4.4.5 InitCommLink

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

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

Prototype dpam16.h or dpam32.h

Remarks *InitCommLink* is used to initialize communications between the PC and the serial DPA II, II+, or III by identifying the COM port 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 serial DPA 4 uses a different driver and to initialize it, you have to use *InitDPA*.

Function Parameters

dpaHandle Identifies the DPA handle use when calling other commands.

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
See Appendix A.2 for complete list of CommPortType.

bBaudRate The serial communication baud rate. The DPA only supports 115K baud upon initialization. If you have a communication 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
See Appendix A.2 for complete list of BaudRateType.

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

In the event of an error, one of the following error codes may be returned:(see Appendix A.2 for error descriptions)
eSerialIncorrectPort

© 2004

eSerialIncorrectBaud
eSerialPortNotFound
eSyncCommandNotAllowed

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, RestoreCommStatus;
      short             dpaHandle;

      CommLinkData.bCommPort        =eComm2;
      CommLinkData.bBaudRate        =eB115200;
      InitCommStatus = InitCommLink(&dpaHandle, &CommLinkData);
      RestoreCommStatus = RestoreCommLink(dpaHandle);
}
```

4.4.6 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 dpam16.h or dpam32.h

Remarks *InitDataLink* initializes the hardware for specified protocols. The DPA II hardware currently supports J1939 and J1708 protocols. The DPA III and 4 hardware currently supports J1939, J1850 and J1708.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

The **InitDataLinkType** structure is as follows:

```
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.

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

For CAN:

bParam0 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 0. (Reference Appendix B.3.)
bParam1 is an unsigned character (hex) that will be loaded into the Intel 82527's Bit Timing Register 1. (Reference Appendix B.3.)
bParam2 CAN Datalink Control parameters:
0x00 = DPA Hardware prioritizes 29 bit CAN Identifiers
0x01 = DPA Hardware prioritizes 11 bit CAN Identifiers
0x03 = Enable J1939 Transport Protocol Layer
DPA Hardware prioritizes 29 bit CAN Identifiers
Note: LoadMailbox Settings for the J1939 transport protocol have no effect unless this datalink parameter is used.
bParam4 is used to set redundant filtering (0 = disabled, 1 = enabled)

Single-Wire CAN Physical Layer (DPA II/DPA II Plus do support)
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.

For J1708:

bParam0 is used to set the baud rate used for the J1708 link
0x00 - 9600 baud
0x01 - 19.2K baud
0x02 – 10.4K baud
0x03 – 38.4K 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
4 = No PID communication
bParam3 is not used for J1708
bParam4 is used to set redundant filtering (0 = disabled, 1 = enabled)

For GMJ1850:

bParam0 is used to set the baud rate
0x00 – 10.4 kbps
0x01 - 41.6 kbps
bParam4 is used to set redundant filtering (0 = disabled, 1 = enabled)

For Ford J1850:

bParam0: baud rate
0 = 10.4kbps
1 = 20.8kbps
2 = 41.6kbps
3 = 83.3kbps
bParam1: node address
Sets the physical address of the DPA on the network.
(Used as the Source Address in transmitted messages.)
bParam4 is used to set redundant filtering (0 = disabled, 1 = enabled)

All other bParams should be set to zero.

For Ford ISO-9141:

Supported on K-Line only.

BProtocol = eJ1708
bParam0 = 0x12 (b xxx1 xxxx signifies 9141
and b xxxx 0002 is baud rate)
bParam1 = interbyte timing. Time between bytes of a message
bParam2 = 0x04
xxxx xxx1 is No RX checksum
xxxx xxx2 is No TX checksum
xxxx xxx4 is No PID used - init timeout values
bParam3 = 0x00 not used
bParam4 = is used to set redundant filtering (0 = disabled, 1 = enabled)

For pass-through mode (DPA II does not support):

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 *InitDataLink* returns **eNoError** on success.
In the event of an error, one of the following error codes may be returned:
(see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example #1:

This example initializes the CAN Data link 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, 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;
            }
      }
}
```

<u>Example #2</u>:

Initializing the J1708 Data link to 9600 baud.

```
#include <windows.h>
#include <stdio.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;
          }
        }
}
```

Example #3:

J1850 InitDataLink - Set the protocol equal to eJ1850 and all parameters equal to zero.

```
ReturnStatusType Status;
InitDataLinkType intiStruct;

memset( &initStruct, 0, sizeof ( InitDataLinkType ) );
initStruct.bProtocol = eJ1850;

Status = InitDataLink(dpaHandle, &initStruct);
```

Redundant Filtering

Redundant filtering for all protocols is set up in the InitDataLink(). To enable redundant filtering, set Param4 to 1.

```
Param4 = 0x01 Redundant Filtering Enabled
Param4 = 0x00 Redundant Filtering Disabled
```

Turning redundant filtering on for one protocol does not turn it on for the others. If redundant filtering is required for all protocols, InitDataLink() must be called with Param4 set for each of the protocols.

When redundant filtering is enabled, the DPA checks *all* receive mailboxes to see if the incoming data matches that mailbox. Non-redundant filtering will stop checking for mailbox matches once a matching mailbox is found.

This extra checking of the mailboxes can cause a significant delay and for this reason redundant filtering is not enabled by default in the DPA II, II+, and IIIs. Because there is the possiblity of receiving multiple callbacks for one message (see example 2) the bandwidth between the DPA and host PC can also be reduced when redundant filtering is enabled. The default receive methodology for these DPAs is to stop checking the receive mailboxes once a match has been found. Because you have to store the default initialization parameters for the various protocols in the DPA 4 and DPA RF, redundant filtering may or may not be enabled by default.

Example 1: Redundant Filtering Disabled
You turn on redundant filtering and load one non-filtering mailbox with callbacks and one filtering mailbox with callbacks. A message matches the non-filtering and filtering mailbox. The DPA will send two callbacks to the PC, one for the non-filtering mailbox and one for the filtering mailbox.

Example 2: Redundant Filtering Enabled
You turn off redundant filtering and load one non-filtering mailbox with callbacks and one filtering mailbox with callbacks. A message matches the non-filtering and filtering mailbox. The DPA will send only one callback to the PC. Once the data is matched to the first mailbox, the DPA will stop checking for matches and will *not* send a callback to the second mailbox.

4.4.7 InitDPA (Windows Only)

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

Syntax
```
ReturnStatusType InitDPA (short *dpaHandle,
     short      DPANumber);
```

Prototype 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 (16-bit) or DG_DPA32.ini (32-bit). The entry for the DPANumber 101 in the INI file is listed below.

Function Parameters

dpaHandle Identifies the DPA handle use when calling other commands.

DPANumber The device number in the DG1210.ini (16-bit) or DG_DPA32.ini (32-bit) file that the DPA is initializing.

```
[DPAInformation101]

DPAParams=DPAII,COM1
```

Return Value On success a handle to the DPA is opened is placed in the dpaHandle variable and eNoError is returned.

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eSerialIncorrectPort
eSerialIncorrectBaud
eSerialPortNotFound
eIrqConflict
eIncorrectDriver
eInvalidDriverSetup
eInvalidBaseAddress
eInvalidDll
eInvalidINI
eSyncCommandNotAllowed

Example:

This example initializes DPA number 101 for communications.

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

void main(void)
{
      ReturnStatusType  InitCommStatus, RestoreCommStatus;
      short             dpaHandle;
      InitCommStatus = InitDPA(&dpaHandle, 101);
```

```
        RestoreCommStatus = RestoreDPA(dpaHandle);
    }
```

32-bit Native Driver INI File Changes

In the 32-bit version of 7.22 driver the native driver started looking in DG_DPA32.ini file for information to be used when InitDPA() was called. The "key" words that it looked for remained the same as the ones used in DG121032.ini. In 7.24 (and all the following releases) if no device exists with the specified device id in the DG_DPA32.ini the driver will try and get the information from the DG121032.ini file. If there is no device with that id then it return an error indicating that the ini file was invalid.

The 16-bit native driver still uses DG1210.ini for InitDPA().

The DG_DPA32.ini file format is as follows:
[DPAInformation(XXX)]
DPAParams=(ParameterInformation)

Examples:
[DPAInformation101]
DPAParams= DPAII,COM1

[DPAInformation103]
DPAParams= DPAII,COM1,POLLED

[DPAInformation201]
DPAParams= DPAIV/H,COM1

[DPAInformation301]
DPAParams= DPAIII,ISA,0x512,5

[DPAInformation501]
DPAParams=TCP/IP,IP=192.168.0.4,PORT=19485

Unlike the RP1210a you can use any DeviceID (RP1210a says to use 100-999).

7.x Key Words:

- COM – it's a serial device, load DPAS32.dll or DPAH32.dll depending on entry
    - /H or IV – use the DPAH32 driver (DPA IV, RF handshaking)
    - if /H or IV is **NOT** in the string use the DPAS driver (DPA II/II+,III handshaking)
    - COM(x) – communications port to connect to, COM1 – COM9

  - B(xxxx) – serial baud rate to connect at, if not specified the default is B115200
  - POLL – polled read thread – not used very often, default is non-polled
- ISA or PC_CARD – it's an ISA or PC_CARD, load DPAI32.dll
  - 0x(xx),(x) – Base Address, IRQ to connect at
- TCP/IP – it's a TCP/IP connection, load DPAT32.dll
  - IP=(xxx.xxx.xxx.xxx),PORT=(x) - IP Address, Port

8.x Key Words:

- Same as above with one added entry
- USB – USB connection, load DPAU(xx).dll (xx – depends on operating system)

4.4.8 InitPCCard

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

Syntax
```
ReturnStatusType InitPCCard (short *dpaHandle,
     PCCardType *PCCardData);
```

Prototype dpam16.h or dpam32.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 ()*.

Function Parameters

dpaHandle Identifies the DPA handle use when calling other commands.

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 and eNoError is returned.

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)

eIrqConflict
eIncorrectDriver
eInvalidDriverSetup
eInvalidBaseAddress
eInvalidDll
eSyncCommandNotAllowed

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, RestoreCommStatus;
      short              dpaHandle;

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

      RestoreCommStatus = RestorePCCard(dpaHandle);
}
```

4.4.9 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 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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 success.

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized

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 );
}
```

4.4.10 LoadMailBox

Function Opens or creates mailboxes for receiving and transmitting messages.

Syntax
```
ReturnStatusType LoadMailBox(short dpaHandle,
     LoadMailBoxType* pLoadMailBoxData);
```

Prototype 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:

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

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 (Resident or Release).
- 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 bits should be masked, and which ones should be matched, in hardware-level filtering. (See Appendix B – *Filter Masks*)
- What information is needed when the message is received.
- Whether the host is to be notified when the message is sent (TX Callback).

Yours truly, should initialize all structure variables to zero before using the structure as shown in the following example.

Example:

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

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure found in section A.3)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eInvalidBitIdentSize
eInvalidDataCount
eMailBoxNotAvailable
eCommLinkNotInitialized
eSyncCommandNotAllowed

<u>Example #1</u>:

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.wDataCount                         = 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->wDataCount);
}

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.wDataCount                = 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;
}
```

<u>Example #3</u>:

*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.wDataCount                  = 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.wDataCount                   = 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.wDataCount           = 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.wDataCount             = 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 or 16-bit drivers. ***The InitDataLink() call must have been called with the transport layer enabled. (see CAN InitDataLink parameters under section 4.4.6 for more detail)***

```
#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.wDataCount              = 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->wDataCount);
}

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.wDataCount                 = 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;
}
```

<u>Example #9</u>:

This example illustrates the creation of a J1850 Load Transmit Mailbox by loading a transmit resident mailbox with transmitCallback as the callback without transmitting.

```
LoadMailBoxType       loadMailboxStruct;
ReturnStatusType      loadMboxReturn;
unsigned char    dummyData[9];

memset(&loadMailboxStruct, 0, sizeof (LoadMailBoxType) );

loadMailboxStruct.bProtocol = eJ1850;
  loadMailboxStruct.bBitIdentSize = 24;
  loadMailboxStruct.dwMailBoxIdent = 0x00000000ul;
  loadMailboxStruct.wDataCount = 0;
  loadMailboxStruct.vpData = dummyData;
  loadMailboxStruct.bResidentOrRelease = eResident;  /* retain mbox */
  loadMailboxStruct.bTransportType = eTransportNone;
  loadMailboxStruct.bExtendedPtrMode = FALSE;
  loadMailboxStruct.wExtendedOffset = 0;
  loadMailboxStruct.bDataInhibit = 0;
  loadMailboxStruct.dwBroadcastTime = 0;
  loadMailboxStruct.bRemote_Data = eDataMailBox;
  loadMailboxStruct.bTimeAbsolute = FALSE;
  loadMailboxStruct.dwTimeStamp = 0;
  loadMailboxStruct.iBroadcastCount = 0;
  loadMailboxStruct.vpUserPointer = NULL;
  loadMailboxStruct.pfApplicationRoutine = transmitCallback;

  loadMboxReturn = LoadMailBox ( dpaHandle, &loadMailboxStruct );
```

Example #10:

This example loads a J1850 receive resident mailbox with receiveCallback as the callback. No message filtering and max data count of 9.

```
LoadMailBoxType  loadMailbox;
 ReturnStatusType dgStatus;
MailBoxType   *rxMailbox;

memset( &loadMailbox, 0, sizeof ( LoadMailBoxType) );

loadMailbox.bRemote_Data = eRemoteMailBox;
loadMailbox.bTransparentUpdateEnable = TRUE;
loadMailbox.pfApplicationRoutine = receiveCallback;
loadMailbox.vpUserPointer = NULL;
loadMailbox.bProtocol =  eJ1850;
loadMailbox.bBitIdentSize = 24;
loadMailbox.vpData = receiveData;
loadMailbox.bTransportType = eTransportNone;
loadMailbox.wDataCount = 9;
loadMailbox.dwMailBoxIdent = 0x00000000ul;
loadMailbox.dwMailBoxIdentMask = 0x00000000ul;

dgStatus =LoadMailBox( dpaHandle, &loadMailbox );
```

4.4.11 LoadTimer

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

Syntax
```
ReturnStatusType LoadTimer (short dpaHandle,
     long unsigned int dwTime);
```

Prototype dpam16.h or dpam32.h

Remarks *LoadTimer* sets the timer used for the time stamping of received and transmitted messages. It takes the parameter passed in **dwTime** and loads it in the DPA's timer. The timer has a 1 millisecond (mS) resolution, and count wrapping occurs every 49.5 days.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

dwTime Time that to set the DPA timer to.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

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);
}
```

4.4.12 PauseTimer

Function Pauses the DPA'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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
PauseTimer(dpaHandle);
```

4.4.13 ReadDataLink (DPA 4 Only)

Function Reads the currently initialized datalink information for a given protocol.

Syntax

```
ReturnStatusType StoreDataLink (short dpaHandle,
     BYTE bProtocol, ReadDataLinkType
     *ReadDataLinkData);
```

Prototype dpam32.h or dpa32.h

Remarks *ReadDataLink* gets the latest datalink parameters from the DPA. If power-up parameters are availabe and InitDataLink has not been called the power-up parameters are put into the ReadDataLinkData variable. This command is only available on the DPA 4.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

bProtocol Identified the protocol that you would like information about.

ReadDataLinkType Structure that contains datalink information.

DataSize	Variable Name	Description
BYTE	bProtocol	Specific protocol that you would like to read info from.
BYTE	bParamCount	Number of parameters used for a given protocol
BYTE	bParam0	Parameter zero
BYTE	bParam1	Parameter one
BYTE	bParam2	Parameter two
BYTE	bParam3	Parameter three
BYTE	bParam4	Parameter four

Return Value *ReadDataLink* returns **eNoError** on success.
In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eCommLinkNotInitialized

Example:

This example reads the current datalink settings for CAN.

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

```
void DisplayInitDataLink(short dpaHandle);

void ReadCurrentCANSettings(void)
{
      ReturnStatusType          InitStatus,ReadStatus,RestoreStatus;
      short                     dpaHandle;
      ReadDataLinkType          ReadDataLinkData


      InitStatus = InitDPA(&dpaHandle, 601);

      if(InitStatus == eNoError)
      {
            ReadStatus =  ReadDataLink(dpaHandle, eCAN,
                  &ReadDataLinkData);

            RestoreStatus = RestoreDPA(dpaHandle);
      }
}
```

4.4.14 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 dpam16.h or dpam32.h

Remarks *ReadDPABuffer* 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 limitations in 16-bit operating systems, it may be necessary to break the reading of large buffers into multiple calls to ReadDPABuffer.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink 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, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)

eDeviceTimeout
eCommLinkNotInitialized

Example:

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);
}
```

4.4.15 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 dpam16 or dpam32.h

Remarks *ReadDPAChecksum* 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

varname User defined variable where checksum is copied to.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout

Example:

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

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

      status = DPAReadDPAChecksum(dpaHandle, &uiChecksum);
      if (status == eNoError)
      {
            if uiChecksum == uiVer707Checksum)
            {
              // do whatever here
            }
      }
}
```

4.4.16 ReceiveMailBox

Function Retrieves the latest message from a specific mailbox.

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

Prototype dpam16.h or dpam32.h

Remarks *ReceiveMailBox* retrieves the latest message from a specific mailbox. Previous messages are overwritten. If called multiple times and no new message has come in, then the latest message is returned multiple times. Be sure to check the message timestamp for this situation.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure found in section A.3)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eInvalidMailBox
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example #1:

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

```
if ((J1939ReceiveHandle != NULL) &&(J1939ReceiveHandle->bActive !=
     0))
{
     (void)ReceiveMailBox (dpaHandle, 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(dpaHandle,J1708ReceiveHandle);
      }
```

4.4.17 ResetDPA

Function Resets the DPA based upon the parameter passed.

Syntax
```
ReturnStatusType ResetDPA (short dpaHandle,
     ResetType *ResetData);
```

Prototype 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

The **ResetType** are as follows:

```
typedef enum
{
        eFullReset,
        eCommReset
}eResetType;
```

eFullReset	Reset the entire DPA, this has the same effect as a power cycle.
eCommReset	Reset only the communication port.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout

Example

```
#include  <windows.h>
#include  <stdio.h>
#include "dpam32.h"
void main(void)
{
       CommLinkType  CommLinkData;
       ReturnStatusType    InitCommStatus, 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);
}
```

© 2004

4.4.18 RestoreCommLink

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

Syntax `ReturnStatusType RestoreCommLink (short dpaHandle);`

Prototype 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 for the restored device can be called until communication to it is initialized again.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eSyncCommandNotAllowed

Example:

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

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

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

      RestoreCommStatus = RestoreCommLink(dpaHandle);
}
```

4.4.19 RestoreDataLink (DPA 4 Only)

Function Restores a datalink to is power-up parameters.

Syntax `ReturnStatusType RestoreDataLink (short dpaHandle, BYTE bProtocol);`

Prototype dpam32.h or dpa32.h

Remarks *RestoreDataLink* restores a given datalink to its power-up state. If a datalink has no power-up parameters then it is disabled. This command is only available on the DPA 4.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

bProtocol Identified the protocol that you would like to restore to it power-up state.

Return Value *RestoreDataLink* returns **eNoError** on success.
In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eCommLinkNotInitialized

Example:

This example restores the J1708 datalink to its power-up state.

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


void RestoreJ1708Settings(void)
{
      ReturnStatusType           InitStatus,RestoreDLStatus,RestoreStatus;
      short                      dpaHandle;


      InitStatus = InitDPA(&dpaHandle, 601);

      if(InitStatus == eNoError)
      {
            RestoreDLStatus =  RestoreDataLink(dpaHandle, eJ1708);

            RestoreStatus = RestoreDPA(dpaHandle);
      }
}
```

© 2004

4.4.20 RestoreDPA (Windows Only)

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

Syntax `ReturnStatusType RestoreDPA (short dpaHandle);`

Prototype 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 for the restored device can be called until communication to it is initialized again.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eSyncCommandNotAllowed

Example:

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

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

     InitCommStatus = InitDPA(&dpaHandle,101);

     RestoreCommStatus = RestoreDPA(dpaHandle);
}
```

4.4.21 RestorePCCard

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

Syntax `ReturnStatusType RestorePCCard (short dpaHandle);`

Prototype 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 for the restored device can be called until communication to it is initialized again.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eSyncCommandNotAllowed

Example:

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

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

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

      ReturnStatusType = RestorePCCard(dpaHandle);
}
```

4.4.22 RequestTimerValue

Function Returns the current DPA timer value.

Syntax
```
ReturnStatusType RequestTimerValue(short
     dpaHandle, long unsigned int *dwTimerValue) ;
```

Prototype dpam16.h or dpam32.h

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

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned:(see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

```
#include <windows.h>
#include <stdio.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);
   sprintf  (szBuffer, "Current DataLink timer value is: %ld",
      dwRequestedTime);
   MessageBox  (NULL, szBuffer, "RequestTimerValue", MB_OK);
}
```

4.4.23 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's internal timer previously stopped by the *PauseTimer* function. Transmit and callback interrupt functions are also resumed.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
ResumeTimer(dpaHandle);
```

4.4.24 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eBaudRateNotSupported
eDeviceTimeout
eCommVerificationFailed
eSerialOuputError

Example:

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

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

4.4.25 StoreDataLink (DPA 4 Only)

Function Stores the currently initialized datalink information to the power-up state for a given protocol.

Syntax
```
ReturnStatusType StoreDataLink (short dpaHandle,
     BYTE bProtocol);
```

Prototype dpam32.h or dpa32.h

Remarks *StoreDataLink* takes the current protocol parameters from the latest call to InitDataLink and saves them as the power-up parameters for that protocol. This allows the DPA 4 to power-up with data links already initialized. This command is only available on the DPA 4.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

bProtocol Identified the protocol that you would like to store power-up commands for.

Return Value *StoreDataLink* returns **eNoError** on success.
In the event of an error, one of the following error codes may be returned:
(see Appendix A.2 for error descriptions)
eDeviceTimeout
eProtocolNotSupported
eCommLinkNotInitialized

Example #1:

This example initializes the CAN Data link to 250K bps 29-bit identifier preferred and then stores the parameters so the DPA powers up with these datalink setting already enabled.

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

void DisplayInitDataLink(short dpaHandle);

void main(void)
{
      ReturnStatusType           InitStatus,StoreStatus,RestoreStatus;
      short                      dpaHandle;
```

```
        InitStatus = InitDPA(&dpaHandle, 601);

        DisplayInitDataLink(dpaHandle);

        RestoreStatus = RestoreDPA(dpaHandle);
}

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 */
        if(InitDataLinkStatus == eNoError)
        {
                MessageBox(NULL,
                        "DataLink successfully initialized.",
                        InitDataLink,MB_OK);
                StoreStatus = StoreDataLink(dpaHandle, eJ1939);
                if(StoreStatus == eNoError)
                {
                        MessageBox(NULL,
                        "DataLink successfully stored.",
                        StoreDataLink,MB_OK);
                }
        }
}
```

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

Note: StoreDataLink is now a stand-alone program that is included with, and works only with, the DPA-4 (including the DPA-4 Serial and USB versions, and also the DPA-RF Base Station). The program is used to customize the DPA-4 power-up parameters so it can be used either in the Passenger Car or Truck & Bus market. The StoreDataLink program is used to define and set the data link defaults used by the DPA-4 during power-up. The StoreDataLink program writes the power-up parameters to the DPA-4 Flash Memory and only needs be done once.

The DPA-4 needs to be connected to the computer that is running the StoreDataLink program.

The StoreDataLink program requires the DPA-4 Firmware v33.12 or later.

4.4.26 SuspendTimerInterrupt

Function Disables a DPA timer interrupt.

Syntax `ReturnStatusType SuspendTimerInterrupt(short dpaHandle);`

Prototype dpam16 or dpam32.h

Remarks *SuspendTimerInterrupt* disables the DPA timer interrupt.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
SuspendTimerInterrupt(dpaHandle);
```

4.4.27 TransmitMailBox

Function Sends messages, using previously opened mailboxes.

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

Prototype dpam16.h or dpam32.h

Remarks *TransmitMailBox* 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink 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, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
DeviceTimeout
eInvalidMailBox
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
(void)TransmitMailBox (dpaHandle, TransmitCANHandle);
```

4.4.28 TransmitMailBoxAsync

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

Syntax

```
ReturnStatusType TransmitMailBoxAsync (short dpaHandle,
     MailBoxType *pMailBoxHandle);
```

Prototype dpam16.h or dpam32.h

Remarks *TransmitMailBoxAsync* 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eInvalidMailBox
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
        void ReceiveCallBack (MailBoxType *)
    {
        TransmitMailBoxAsync(dpaHandle, TransmitCANHandle)
        }
```

4.4.29 UnloadMailBox

Function Closes a previously opened mailbox.

Syntax

```
ReturnStatusType UnloadMailBox (short dpaHandle,
     MailBoxType *MailBoxHandle);
```

Prototype dpam16.h or dpam32.h

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

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
(void)UnloadMailBox (dpaHandle, TransmitCANHandle);
```

4.4.30 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 dpam16.h or dpam32.h

Remarks Updates the receive mailbox parameters (data count, identifier, identifier filter, etc.) fo ra specific mailbox. Any message that has already been placed inside the DPA's internal buffer will not be overwritten. After this command is received by the DPA, the filtering of messages for that mailbox will be modified. If you update the data in the *pMailBoxHandle* but do not set the *bUpdateFlag* for that data to be updated, then the data will no be updated.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3)

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

bUpdateFlag Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A.1)

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

*Example of how to update a previously opened receive J1708 mailbox (J1708ReceiveHandle). This example will update the ID, and data count using the **UpdateReceiveMailBox** function.*

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

J1708ReceiveHandle ->wDataCount     = 6;
J1708ReceiveHandle ->bMID           = 0x80;
J1708ReceiveHandle ->bPID           = 0x34;
bUpdateFlag = UPDATE_ID | UPDATE_DATA_COUNT;

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

4.4.31 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 dpam16.h or dpam32.h

Remarks *UpdateReceiveMailBoxAsync* updates the receive mailbox parameters (data count, identifier, identifier filter, etc.) for a specific mailbox. Any message that has already been placed inside the DPA'a internal buffer will not be overwritten. After this command is received by the DPA, the filtering of messages for that mailbox will be modified. This function is for use inside a callback routine. If you update the data in the *pMailBoxHandle* but do not set the *bUpdateFlag* for that data to be updated, then the data will no be updated.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

bUpdateFlag Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A1.)

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
ReturnStatusType CALLBACK  ReceiveCallBack (void)
{
     MailBoxType          *J1708ReceiveAsyncHandle;
     ReturnStatusType     UpdateReceiveAsyncStatus;
     unsigned char        bUpdateFlagAsync;
```

```
        bUpdateFlagAsync =
        UpdateReceiveAsyncStatus =
                UpdateReceiveMailBoxAsync( dpaHandle,
                &J1708ReceiveAsyncHandle,
                bUpdateFlagAsync);
    }
```

4.4.32 UpdateTransMailBoxData

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

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

Prototype dpam16.h or dpam32.h

Remarks *UpdateTransMailBoxData* updates only the data being sent from a previously opened mailbox.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

```
unsigned char        TransmitMailBoxBroadcastData;
ReturnStatusType     UpdateTransMailBoxData;

strcpy ((char *)TransmitMailBoxBroadcastData,
     "12345678");
UpdateTransMailBoxData = UpdateTransMailBoxData
     (dpaHandle, TransmitCANHandle);
```

4.4.33 UpdateTransMailBoxDataAsync

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

Syntax

```
ReturnStatusType UpdateTransMailBoxDataAsync(
     short dpaHandle, MailBoxType
     *pMailBoxHandle);
```

Prototype 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.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example:

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

```
ReturnStatusType CALLBACK  TransmitCallBack (void)
{
   unsigned char TransmitMailBoxBroadcastData;

   strcpy ((char *)TransmitMailBoxBroadcastData, "12345678");
   (void)UpdateTransMailBoxDataAsync(dpaHandle, TransmitCANHandle);
}
```

4.4.34 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 dpam16.h or dpaM32.h

Remarks *UpdateTransmitMailBox* allows the application to change any of the parameters included inside a transmit mailbox. If you update the data in the *pMailBoxHandle* but do not set the *bUpdateFlag* for that data to be updated, then the data will no be updated.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

bUpdateFlag Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A.1.)

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized
eSyncCommandNotAllowed

Example #1:

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

```
void UpdateCANData( MailBoxType    *TransmitCANHandle)
{
 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));

}
```

4.4.35 UpdateTransmitMailBoxAsync

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

Syntax

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

Prototype dpam16.h or dpam32.h

Remarks *UpdateTransmitMailBoxAsync* allows the application to change any of the parameters included inside a transmit mailbox, for use inside a callback routine. If you update the data in the *pMailBoxHandle* but do not set the *bUpdateFlag* for that data to be updated, then the data will no be updated.

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitCommLink, InitPCCard, InitUSBLink or InitDPA.

MailBoxType (Structure defined in Appendix A.3.)

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

bUpdateFlag Identifies which fields of the mailbox should be updated. (Valid values for *bUpdateFlag* are defined in section A.1.)

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout
eMailBoxNotActive
eCommLinkNotInitialized

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);
}
```

4.5 KWP2000 DPA API function descriptions

The following API functions are not available for a standard DPA. They are only available for DPA's which have ISO-9141 K and L line support installed. These functions are implemented in the *DGKWP2K.dll*, not the single or multi DPA interface (*DPAxx.dll or DPAMxx.dll*).

4.5.1 INIT_KWP2000_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.

4.5.2 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.

4.5.3 SET_TIMING

API Example:

```
INT RET=SET_TIMING (    INT P1,
                        INT P2,
                        INT P3,
                        INT P4…)
```

RET = 0 → OK
RET =-1 → TIME_OUT COMMAND

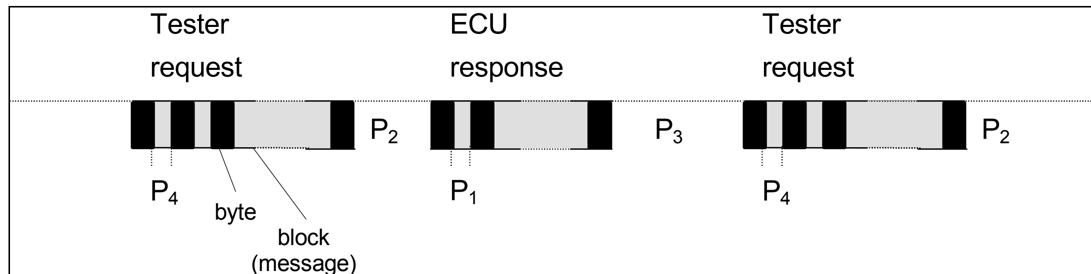
Action:

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*

4.5.4 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)

4.5.5 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 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

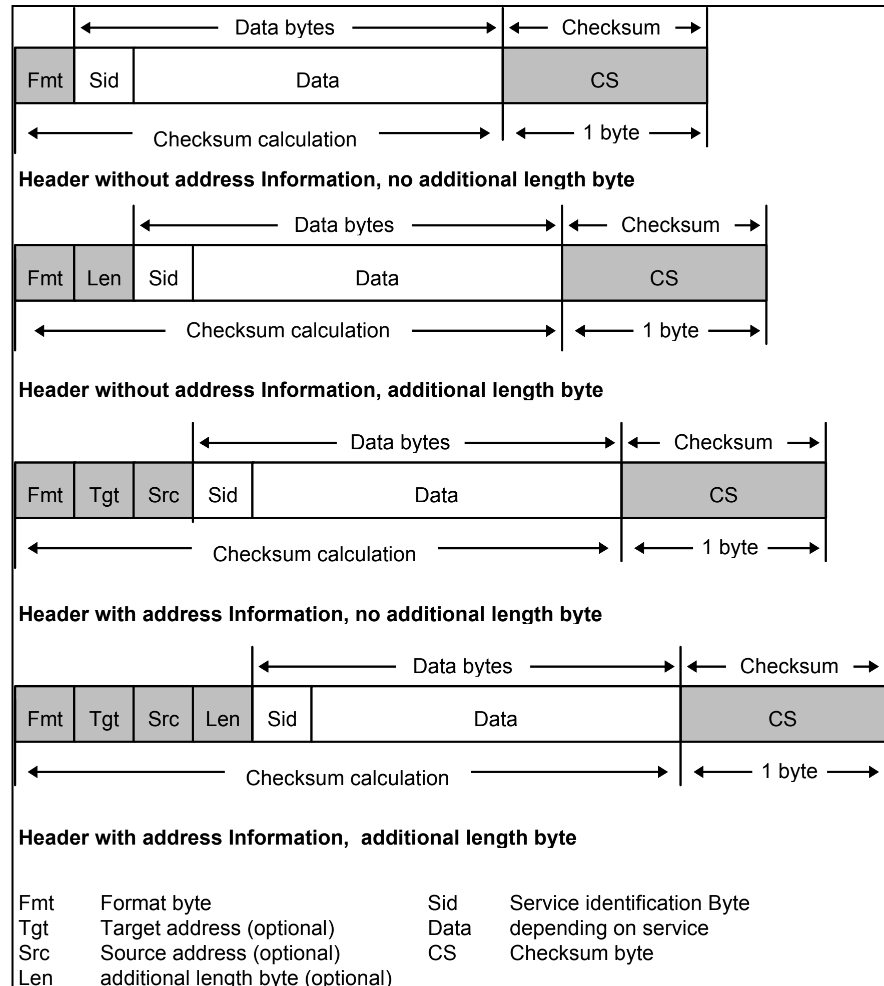
A_1, A_0: define the form of the header, which will be used by the message (see below).

$L_5..L_0$: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



4.5.6 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:

Baud Rate 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

4.5.7 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 NETWORK 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 DETERMINED 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.

4.5.8 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.

4.5.9 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.

4.5.10 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.

4.5.11 Initialization

This API shall not support 5-baud or CARB initialization.

4.5.12 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.

4.6 USB DPA API function descriptions

For USB support there are two function calls added to the native DPA 8.X drivers: InitUSBLink and RestoreUSBLink.

RestoreUSBLink is just like RestoreCommLink but it restores the DPA attached to the USB port. InitUSBLink described below because it utilizes a new USBLinkType structure.

4.6.1 RestoreUSBLink

Function Restores the USB port between the PC and the DPA to its previous (pre-*InitUSBLink*) state.

Syntax `ReturnStatusType RestoreUSBLink (short dpaHandle);`

Prototype dpam32.h or dpa32.h

Remarks *RestoreUSBLink* is used to restore any and all interrupt vectors that were set up during *InitUSBLink().* Once *RestoreUSBLink ()* is called, no other library functions for the restored device can be called until communication to it is initialized again.

The entries in the DG_DPA32.ini file are as follows:

```
DPAParams=DPAIV,USB,Interface=0,[HS or FS]
```

Function Parameters

dpaHandle Identifies the DPA to send this command to. The handle is returned from InitUSBLink, InitPCCard or InitDPA.

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

In the event of an error, one of the following error codes may be returned: (see Appendix A.2 for error descriptions)
eDeviceTimeout

Example:

RestoreUSBLink(dpaHandle);

4.6.2 InitUSBLink

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

Syntax ReturnStatusType InitUSBLink (short *dpaHandle, USBLinkType *USBLinkData);

Prototype In dpam32.h or dpa32.h

Remarks *InitUSBLink* is used to initialize communications between the PC and the DPA, by identifying the USB speed. This function must be called before any other library functions are called. When *InitUSBLink()* is called, the appropriate interrupt vector pointers are saved so that they can be restored using *RestoreUSBLink ()*.

The *USBLinkType* structure is as follows:

```
typedef struct
{
WORD        wInterface;
BYTE        bHS_FS;
} USBLinkType;
```

wInterface: The USB interface to use. Set to 0.

bHS_FS: High speed or Full speed USB.
eFullSpeed = 0 //default
eHighSpeed =1

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

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

eDeviceTimeout

<u>Example</u>:

This example initializes a USB DPA at full speed.

```
#include  <windows.h>

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

void main(void)
{
      short              dpaHandle;
      USBLinkType        USBLinkData;
      ReturnStatusType   InitUSBStatus;
```

```
        ReturnStatusType  RestoreUSBStatus;

        USBLinkData.wInterface         =0;
        USBLinkData.bHS_FS             =eFullSpeed;
        InitUSBStatus = InitUSBLink(&dpaHandle, &USBLinkData);
}
```

5.0 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 Plus/V also can support CAN and J1708 channels, and can be operated on MS DOS, Windows 95, 98, ME, 2000, and NT . For further instruction on the VSI programs and capabilities, please consult the Vehicle Serial Interface for Class 2 manual.

Note: When in the VSI mode, earlier releases of DPA/VSI allowed the use of a subset of DPA commands. With current and future releases, no DPA functions are possible when in the VSI mode.

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 MAILBOX_J1939_BUFFER_SIZE 1785
#define MAX_TRANSFER_SIZE                 2000
#define MaxBufferSize                     2048
```

Maximum size of check data link 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
```

Inhibit flags

```
#define TimeStamp_Inhibit              0x20
#define ID_Inhibit                     0x40
#define Data_Inhibit                   0x80
```

Update mailbox flag definitions

```
#define UPDATE_DATA_LOCATION       0x80 // update data location
#define TRANSMIT_IMMEDIATE         0x40 // transmit immediatly
#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_ID_MASK         0x10 // update ID MASK / (MID-PID-Priority)

// ONLY valid for receive
#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,
  eComm5,
  eComm6,
  eComm7,
  eComm8,
```

```
  eComm9
} CommPortType;
```

Enumerations for *BaudRateType*

```
typedef enum BaudRateType
{
  eB9600,
  eB19200,
  eB28800,
  eB38400,
  eB57600,
  eB115200,
  eB230400,
  eB460800
} BaudRateType;
```

Enumerations for *ProtocolType*

```
typedef enum ProtocolType
{
  eISO9141,
  eJ1708,
  eJ1850,
  eJ1939,
  ePassThru,
  eUSB,
  eRS232,
  eCAN = 3
} ProtocolType;
```

Enumerations for *ResetType*

```
typedef enum
{
  eFullReset,
  eCommReset
}eResetType;
```

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
  eInvalidTransportType,
  eSerialOutputError,
  eInvalidBufferOffset,
  eInvalidBufferLocation,
  eOutOfMemory,
  eInvalidDpa,
  eInvalidDpaHandle,
  eInvalidPointer,
  eBaudRateConflict,
  eIrqConflict,
  eIncorrectDriver,
  eInvalidDriverSetup,
```

```
  eInvalidBaseAddress,
  eInvalidINI,
  eInvalidDll,
  eCommVerificationFailed,
  eInvalidLock,
  eServerDisconnect,
  eInvalidSocket,
  eWinSockError,
  eInvalidDisplayType,
  eModemError,
  eInvalidResetType,
  eProtocolNotInitialized,
  eOperatingSystemNotSupported,
} ReturnStatusType;
```

eNoError – Success.

eDeviceTimeout – Unable to communicate with the DPA before the timeout period expired.

eProtocolNotSupported – Invalid protocol passed in.

eBaudRateNotSupported – Invalid baud rate passed in.

eInvalidBitIdentSize – Invalid identifier size. CAN supports only 11 or 29 bit identifiers.

eInvalidDataCount – Invalid data count. CAN supports only 8 bytes of data.

eInvalidMailBox – Attempting to transmit or receive with an unopened mailbox.

eNoDataAvailable – No data in mailbox.

eMailBoxInUse – Not used by 8.x drivers.

eMailBoxNotActive – Mailbox was not open.

eMailBoxNotAvailable – All available mailboxes (16) are in use.

eTimerNotSupported – Timer not supported.

eTimeoutValueOutOfRange – A period greater than 1 minute (60,000ms) was specified.

eInvalidTimerValue – Timer value out of range.

eInvalidMailBoxDirection – Not used by 8.x drivers.

eSerialIncorrectPort – Incorrect com port.

eSerialIncorrectBaud – Incorrect baud rate.

eSerialPortNotFound – Com port not found.

eSerialPortTimeout – Not used by 8.x drivers.

eCommLinkNotInitialized – Serial port not opened.

eAsyncCommBusy – Still waiting for a previous asynchronous command to finish processing.

eSyncCommInCallBack – Not used by 8.x drivers.

eAsyncCommandNotAllowed – Not used by 8.x drivers.

eSyncCommandNotAllowed – Cannot call from within callback (ISR).

eLinkedMailBox – Mailbox is linked.

eInvalidExtendedFlags – Not used by 8.x drivers.

eInvalidCommand – Not used by 8.x drivers.

eInvalidTransportType – Invalid transport type.

eSerialOuputError – Could not set PC baud.

eInvalidBufferOffset – Invalid buffer offset.

eInvalidBufferLocation – Invalid buffer location.

eOutOfMemory – Driver needs more memory on computer.

eInvalidDpa – Invalid dpa type.

eInvalidDPAHandle – Invalid dpa handle being passed to a command.

eInvalidPointer – Invalid pointer.

eBaudRateConflict – Trying to initialize a comport that is already initialized at a different baud rate.

eIrqConflict – Irq in use.

eIncorrectDriver – The driver is the wrong driver.

eInvalidDriverSetup – The driver is set up incorrectly.

eInvalidBaseAddress – The base adderss is invalid.

eInvalidINI – The INI record is invalid.

eInvalidDll – There is a bad or missing DLL.

eCommVerificationFailed – Could not communicate after change.

eInvalidLock – Lock not found.

eServerDisconnect – The TCP/IP server disconnected.

eInvalidSocket – Invalid socket.

eWinSockError – Win socket error occurred.

eInvalidDisplayType – Invalid display type.

eModemError – A modem error occurred.

eInvalidResetType – Invalid reset type.

eProtocolNotInitialized – Protocol not initialized.

eOperatingSystemNotSupported – OS not supported.

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 USBLinkType

```
Typedef struct
{
  WORD            wInterface;
  BYTE            bHS_FS;
}   USBLinkType;
```

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. Loaded when mailbox is active, cleared when mailbox inactive.

bFilterType - Pass or Block filtering.
ePass - Send data up
eBlock - Block data

bMailBoxDirectionData - Used to identify mailbox direction, either *transmit* or *receive*.

bTransportType - Type of transport to use.
eTransportNone - No Transport Layer
eTransportBAM - Use Broadcast Announcement Message
eTransportRTS - Use Request to Send

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

bMIDMask - J1708 MID mask

bPIDMask - J1708 PID mask

bDataRequested - Flag indicating that data has been requested from the DPA.

bReceiveFlags - Flag used internally indicating that requested data has arrived. Can not be changed or cleared by the user.

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 Data link

```
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;
```

Key Elements for *LoadMailBox Structure*

		J1708 RCV	J1708 XMIT	CAN RCV	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	bTimeStampInhibit;	X	X	X	X
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:
eRemoteMailBox - Receive
eDataMailBox - Transmit
bTransportType - Transport type to be used:
eTransportNone – No Transport Layer
eTransportBAM – Broadcast Announcement Message
eTransportRTS – RTS/CTS
bResidentOrRelease - Specifies transmit mailbox: *Resident* or *Release*.
eResident - The mailbox is permanent, it will remain active and can be used again.
eRelease - The mailbox is automatically unloaded after it is used.

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.
ePass - Pass (pass only messages that match the filter)
eBlock - 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.
TRUE = ON
FALSE = OFF

wExtendedOffset - The location of data in the buffer (scratch pad).

(CALLBACK *pfApplicationRoutine)(MailBoxType *) - 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.

© 2004

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;        /* Inter message timeout for BAM receive*/
  WORD  iBAM_BAMTXTime;     /* Time between BAM and first data packet*/
  WORD  iBAM_DataTXTime;    /* Time between BAM Data packets*/
  WORD  iRTS_Retry;         /*Number of times to send RTS without CTS*/
  WORD  iRTS_RetryTransmitTime;/* Time between retries of RTS TX*/
  WORD  iRTS_TX_Timeout;    /* Time to wait for a CTS after RTS TX*/
  WORD  iRTS_TX_TransmitTime;/* Time between data packets on RTS TX*/
  WORD  iRTS_RX_TimeoutData;/* Timeout between data packets on RTS RX*/
  WORD  iRTS_RX_TimeoutCMD; /* Timeout between CTS and 1st data packet*/
  BYTE  iRTS_RX_CTS_Count;  /* Number of packets to CTS for*/
  BYTE  iRTS_TX_CTS_Count;  /* Number of packets to CTS for*/
} ConfigureTransportType;
```

A.4 Function Prototypes

ifdef __cplusplus

```
extern "C" {
ReturnStatusType far InitCommLink (short dpaHandle, CommLinkType
        *CommLinkData);
ReturnStatusType far RestoreCommLink (short dpaHandle,);
ReturnStatusType far InitDataLink (short dpaHandle, InitDataLinkType
        *InitDataLinkData);
ReturnStatusType far CheckLock (short dpaHandle, char *szSearchString,
        unsigned char *bFound);
ReturnStatusType far CheckDataLink (short dpaHandle, char *cVersion);
ReturnStatusType far LoadDPABuffer (short dpaHandle, unsigned char *bData,
        unsigned int wLength, unsigned int wOffset);
ReturnStatusType far ReadDPABuffer(short dpaHandle, unsigned char *bData,
        unsigned int wLength, unsigned int wOffset);
ReturnStatusType far LoadMailBox (short dpaHandle, LoadMailBoxType
        *pLoadMailBoxData);
ReturnStatusType far TransmitMailBox  (short dpaHandle, MailBoxType
        *pMailBoxHandle);
ReturnStatusType far TransmitMailBoxAsync (short dpaHandle, MailBoxType
        *pMailBoxHandle);
ReturnStatusType far UpdateTransMailBoxData (short dpaHandle, MailBoxType
        *pMailBoxHandle);
ReturnStatusType far UpdateTransMailBoxDataAsync (short dpaHandle,
        MailBoxType *pMailBoxHandle);
ReturnStatusType far UpdateTransmitMailBox (short dpaHandle, MailBoxType
        *pMailBoxHandle, unsigned char bUpdateFlag);
ReturnStatusType far UpdateTransmitMailBoxAsync (short dpaHandle,
        MailBoxType *pMailBoxHandle, unsigned char bUpdateFlag);
ReturnStatusType far ReceiveMailBox (short dpaHandle, MailBoxType
        *pMailBoxHandle);
ReturnStatusType far UnloadMailBox (short dpaHandle, MailBoxType
        *pMailBoxHandle);
ReturnStatusType far LoadTimer (short dpaHandle, unsigned long dwTime);
ReturnStatusType far EnableTimerInterrupt (short dpaHandle,
        EnableTimerInterruptType *pEnableTimerInterruptData);
ReturnStatusType far SuspendTimerInterrupt (short dpaHandle,);
ReturnStatusType far RequestTimerValue (short dpaHandle, unsigned long
        *dwTimerValue);
 }
#else
```

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 micro controller 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).

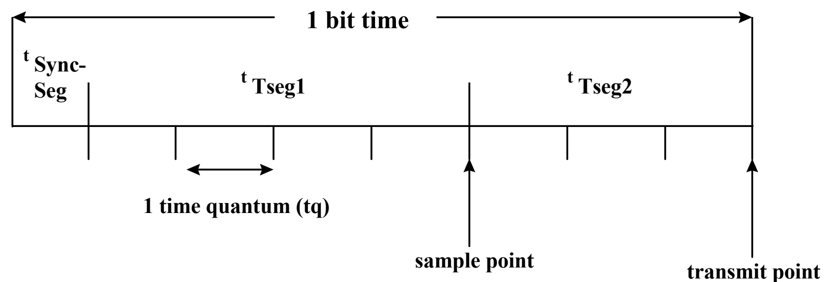
Register BTR1:

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.)



© 2004

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

$$\text{Bus Frequency} = \frac{16\text{MHz}}{2 \times (1 + 1) \times (3 + 1 + 12)} = 250{,}000$$

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, PC104, USB
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, PC104, USB
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.H

D Connector Pinouts

	DPA II	DPA II /T	DPA III	DPA III+ /M	DPA III+ /T DPA 4 /T (or RF)	DPA II DDE	DPA III+ /MH
Ground	9	6	9	9	6	6	9
Power (9-32 vdc)	10	8	10	10	8	8	10
J1708-	11	14	11	11	14	14	11
J1708+	12	15	12	12	15	15	12
CAN Shield	13	7	13	13	7	7	13
CAN Lo	14	12	14	14	12	12	14
CAN Hi	15	13	15	15	13	13	15
CAN Term 1	7	3	7	3	3	3	3
CAN Term 2	8	4	8	4	4	4	4
CAN TX	NC	NC	3	NC	NC	NC	NC
CAN RX	NC	NC	4	NC	NC	NC	NC
SW CAN	NC	NC	NC	NC	10*	NC	NC
ALDL	NC	NC	1	1	1	NC	1
ALDL RX	NC	NC	NC	NC	NC	NC	2
Master/Slave 1	NC	NC	2	2	2	NC	NC
Master/Slave 2	NC	NC	6	6	10*	NC	6
9141 K Line	NC	NC	NC	NC	NC	NC	NC
9141 L Line	NC	NC	NC	NC	NC	NC	NC
J1850+	NC	NC	5	5	5	NC	5
J1850-	NC	NC	NC	NC	NC	NC	NC
External Power	NC	NC	NC	NC	NC	NC	NC
External Ground	NC	NC	NC	NC	NC	NC	NC
ATEC Data	NC	NC	NC	7	NC	NC	7
ATEC Diag	NC	NC	NC	8	NC	NC	8
Discrete In	NC	NC	NC	NC	9	NC	NC
Discrete Out	NC	NC	NC	NC	11	NC	NC
UNUSED	1,2,3,4,5,6,7	1,2,5,9,10,11	NONE	NONE	NONE	1,2,5,9,10,11	NONE

* depends on endplate

	DPA III+ /C	VSI	DPA III+ /TSW	DPA III+ /MHSW	DPA III+ /SCP	DPA ISA	DPA PC/104
Ground	9	23, 24	6	9	6	6	11
Power (9-32 vdc)	10	3, 4	8	10	8	NC	NC
J1708-	11	NC	14	11	14	14	12
J1708+	12	NC	15	12	15	15	14
CAN Shield	13	NC	7	13	7	7	13
CAN Lo	14	NC	12	14	12	12	8
CAN Hi	15	NC	13	15	13	13	10
CAN Term 1	7	NC	3	3	3	3	5
CAN Term 2	8	NC	4	4	4	4	7
CAN TX	NC	NC	NC	NC	NC	NC	NC
CAN RX	NC	NC	NC	NC	NC	NC	NC
SW CAN	NC	NC	10	6	10		
ALDL	1	1	1	1	1	1	1
ALDL RX	NC	NC	NC	2	NC		
Master/Slave 1	2	NC	2	NC	2	2	3
Master/Slave 2	6	NC	NC	NC	NC	10	4
9141 K Line	1	NC	NC	NC	NC	NC	NC
9141 L Line	3	NC	NC	NC	NC	NC	NC
J1850+	5	14	5	5	5	5	9
J1850-	NC	NC	NC	NC	9		
External Power	NC	NC	NC	NC	NC	8	15
External Ground	NC	NC	NC	NC	NC	9	2
ATEC Data	NC	NC	NC	7	NC	NC	NC
ATEC Diag	NC	NC	NC	8	NC	NC	NC
Discrete In	4	NC	9	NC	NC	NC	NC
Discrete Out	NC	NC	11	NC	11	NC	NC
UNUSED	NONE	SEVERAL	NONE	NONE	NONE	11	6,16

I Index

87C196CB, 3
API functions, 25
API library, 5
API overview, 24
Application Program Interface. *See* API
Bit Timing Registers, 129
Buffer functions, 28
CAN, ii, 3
  Bit Timing Registers, 129
CAN (J1939), 1
CHECKDPA.EXE, 17
compiler information, 24
connections
  network, 6
  power, 6
Connector Pinouts, 135
Data-link configuration functions, 27
Dearborn Protocol Adapter. *See* DPA
Defines
  Inhibit flags, 112
  Maximum size of check data link string, 111
  Number of mailboxes, 111
  Update mailbox flag definitions, 112
  Update transmit mailbox flag parameter definitions, 111
DEFINES AND STRUCTURES, 111
DLL, 24
DLLs, 5
DPA
  checking communication, 9
  connecting to the PC, 8
  functional overview, 18
  introduction to, 1
DPA II Plus, 1
DPA III ISA card, 1
  setting jumpers, 10
DPA III Plus, 1
DPA ISA card
  install and setup, 10
DPA PC/104 card, 11
DPA RF, 16
driver
  RP1210A, 1
driver installation, 6
Drivers
  DPA16, 133
  DPA32, 134
  DPAM16, 133
Dynamic Link Library. *See* DLL
e-mail, 3
filtering
  redundant, 45, 46
filters, 127
  J1708, 128
format byte, **100**
function descriptions, 29
Function Prototypes
  ifdef __cplusplus, 126
functions
  Buffer, 28
  ConfigureTransportProtocol, 31
  Data-link configuration, 27
  DisableDataLink (DPA 4 Only), 34
  EnableTimerInterrupt, 35
  InitCommLink, 37
  InitDataLink, 39
  InitDPA (Windows Only), 47
  InitPCCard, 50
  LoadDPABuffer, 52
  LoadMailBox, 53
  LoadTimer, 65
  Message handling, 27
  PauseTimer, 66
  ReadDataLink (DPA 4 Only), 67
  ReadDPABuffer, 69
  ReadDPAChecksum, 70
  ReceiveMailBox, 71
  RequestTimerValue, 79
  ResetDPA, 73
  RestoreCommLink, 75

RestoreDataLink (DPA 4 Only), 76
RestorePDA (Windows Only), 77
RestorePCCard, 78
ResumeTimer, 80
SetBaudRate (32-bit Windows Only), 81
StoreDataLink (DPA 4 Only), 82
SuspendTimerInterrupt, 84
system, 26
Timer, 28
TransmitMailBox, 85
TransmitMailBoxAsync, 86
UnloadMailBox, 87
UpdateReceiveMailBox, 88
UpdateReceiveMailBoxAsync, 89
UpdateTransMailBoxData, 90
UpdateTransMailBoxDataAsync, 91
UpdateTransmitMailBox, 92
UpdateTransmitMailBoxAsync, 94
header, **100**
I/O Buffer, 22
iECM96
PC, 3
Intel, 3
inter-block-time, 96
inter-byte-time, 96
ISA card
installing, 11
network connection, 10
power connection, 10
ISO-11898 physical layer, 1
J1708 filters, 128
J1708 protocol, 1
J1850 protocol, 1
J1939 protocol, 1
J1939/11 physical layer, 1
J1939/15 physical layer, 1
J1939/21 transport layer, 1
KWP 2000 functions
GET_STATUS, 105
INIT_KWP2000_DPA, 95
RELEASE_KWP2000, 95
SEND_MESSAGE, 105
SET_BAUDRATE, 101
SET_COM_PARAMETER_1, 97
SET_COM_PARAMETER_2, 98
SET_TIMING, 96
START_COMMUNICATION, 102
STOP_COMMUNICATION, 104
masks. *See* filters
Message handling functions, 27
message structure, **100**
network connections, 6
oversized messages, 22
PC/104 card, 1
installation, 13
network connection, 13
power connection, 13
setting jumpers, 12
Philips
82C250, 3
PCA82C250, 3
physical layer
ISO-11898, 1
J1939/11, 1
J1939/15, 1
power connections, 6
protocols
CAN (J1939), 1
J1708, 1
J1850, 1
Receive Mailbox, 21
redundant filtering, 45, 46
related publications, 3
Remote Station
connecting the DPA, 16
RP1210A driver, 1
RS-232, 9
RS-232 installation and setup, 7
startCommunication, 100
Structures
Structure for DPA error, 118
Structure for InitCommLink, 118
Structure for initializing the DPA Data link, 121
Structure for PC copy of MailBox, 119
Structure for Timer Interrupts, 125
Structure for Transport Protocol, 125
Structure for USBLinkType, 118
System functions, 26
technical support
contacting, 2
timer, 18
Timer functions, 28
Transmit Mailbox, 20
transport layer
J1939/21, 1
Typedefs
Enumerations for BaudRateType, 113
Enumerations for CommPortType, 112
Enumerations for DPA errors, 117
Enumerations for MailBoxDirectionType, 117
Enumerations for ProtocolType, 113
Enumerations for ResetType, 113
Enumerations for ReturnStatusType, 114
Enumerations for TransmitMailBoxType, 117

USB API functions
  InitUSBLink, 108
  RestoreUSBLink, 107
USB hardware, 13
  installation and setup, 13
  network connection, 15
  power connection, 15
USB interface, 1
Vehicle Serial Interface. *See* VSI
VSI, 110
VSI Emulation, 110
web site, 3