PLX PCI Host SDK
SOFTWARE DEVELOPMENT KIT
(WINDOWS HOST)
Programmer's Reference Manual

PLX PCI Host SDK

SOFTWARE DEVELOPMENT KIT

(WINDOWS HOST)

Programmer's Reference Manual

Version 3.1

May 2000

Website: http://www.plxtech.com
Email: apps@plxtech.com
Phone: 408 774-9060
800 759-3735
Fax: 408 774-2169

Document number: PCI-SDK-PRM-P1-3.1

PLX SOFTWARE LICENSE AGREEMENT

THIS PLX SOFTWARE IS LICENSED TO YOU UNDER SPECIFIC TERMS AND CONDITIONS. CAREFULLY READ THE TERMS AND CONDITIONS PRIOR TO USING THIS SOFTWARE. OPENING THIS SOFTWARE PACKAGE OR INITIAL USE OF THIS SOFTWARE INDICATES YOUR ACCEPTANCE OF THE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD RETURN THE ENTIRE SOFTWARE PACKAGE TO PLX.

LICENSE

This PLX Software License agreement is a legal agreement between you and PLX Technology, Inc. for the PLX Software, which is provided on the enclosed PLX CD-ROM. PLX Technology owns this PLX Software. The PLX Software is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties, and is licensed, not sold. If you are a rightful possessor of the PLX Software, PLX grants you a license to use the PLX Software as part of or in conjunction with a PLX chip on a **per project basis**. PLX grants this permission provided that the above copyright notice appears in all copies and derivatives of the PLX Software. Use of any supplied runtime object modules or derivatives from the included source code in any product without a PLX Technology, Inc. chip is strictly prohibited. You obtain no rights other than those granted to you under this license. You may copy the PLX Software for backup or archival purposes. You are not authorized to use, merge, copy, display, adapt, modify, execute, distribute or transfer, reverse assemble, reverse compile, decode, or translate the PLX Software except to the extent permitted by law.

PLX Software License Agreement

GENERAL

If you do not agree to the terms and conditions of this PLX Software License Agreement, do not install or use the PLX Software and promptly return the entire unused PLX Software to PLX Technology, Inc. You may terminate your PLX Software license at any time. PLX Technology may terminate your PLX Software license if you fail to comply with the terms and conditions of this License Agreement. In either event, you must destroy all your copies of this PLX Software. Any attempt to sub-license, rent, lease, assign or to transfer the PLX Software except as expressly provided by this license, is hereby rendered null and void**.**

WARRANTY

PLX Technology, Inc. provides this PLX Software AS IS, WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, AND ANY WARRANTY OF MERCHANTIBILITY OR FITNESS FOR A PARTICULAR PURPOSE. PLX makes no guarantee or representations regarding the use of, or the results based on the use of the software and documentation in terms of correctness, or otherwise; and that you rely on the software, documentation, and results solely at your own risk. In no event shall PLX be liable for any loss of use, loss of business, loss of profits, incidental, special or, consequential damages of any kind. In no event shall PLX's total liability exceed the sum paid to PLX for the product licensed here under.

PLX Copyright Message Guidelines

The following copyright message along with the following text must appear in all software products generated and distributed, which use the PLX API libraries:

"Copyright © 2000 PLX Technology, Inc."

Requirements:

- Arial font
- Font size 12 (minimum)
- Bold type
- Must appear as shown above in the first section or the so called "Introduction Section" of all manuals
- Must also appear as shown above in the beginning of source code as a comment

TABLE OF CONTENTS

1 General Information 1-1
1.1 About This Manual 1-1
1.2 Where To Go From Here 1-1
1.3 Conventions 1-1
1.3.1 Programming Conventions 1-1
1.4 Terminology 1-2
1.5 Customer Support 1-2
2 Creating Applications Using The PLX API 2-1
2.1 Host (Windows) Applications 2-1
2.1.1 Sample Applications 2-1
2.1.2 Creating a Host Application 2-1
2.1.2.1 Creating A Microsoft Developer's Studio Project File 2-2
2.1.2.2 Building A Custom Application 2-3
2.2 IOP Applications 2-3
3 Host PCI API 3-1
3.1 PCI API Function Quick Reference List 3-1
3.2 PCI API Feature List 3-3
3.3 Sample Function Entry 3-4
Sample_Function 3-4
3.4 API Function Details 3-4
PlxBusIopRead ..... **Error! Bookmark not defined.**
PlxBusIopWrite ..... **Error! Bookmark not defined.**
PlxDmaBlockChannelClose ..... **Error! Bookmark not defined.**
PlxDmaBlockChannelOpen ..... **Error! Bookmark not defined.**
PlxDmaBlockTransfer ..... **Error! Bookmark not defined.**
PlxDmaBlockTransferRestart ..... **Error! Bookmark not defined.**
PlxDmaControl ..... **Error! Bookmark not defined.**
PlxDmaSglChannelClose ..... **Error! Bookmark not defined.**
PlxDmaSglChannelOpen ..... **Error! Bookmark not defined.**
PlxDmaSglTransfer ..... **Error! Bookmark not defined.**
PlxDmaShuttleChannelClose ..... **Error! Bookmark not defined.**
PlxDmaShuttleChannelOpen ..... **Error! Bookmark not defined.**
PlxDmaShuttleTransfer ..... **Error! Bookmark not defined.**
PlxDmaStatus ..... **Error! Bookmark not defined.**
PlxHotSwapIdRead ..... **Error! Bookmark not defined.**

PlxHotSwapNcpRead ..... **Error! Bookmark not defined.**
PlxHotSwapStatus ..... **Error! Bookmark not defined.**
PlxIntrAttach ..... **Error! Bookmark not defined.**
PlxIntrDisable ..... **Error! Bookmark not defined.**
PlxIntrEnable ..... **Error! Bookmark not defined.**
PlxIntrStatusGet ..... **Error! Bookmark not defined.**
PlxIoPortRead ..... **Error! Bookmark not defined.**
PlxIoPortWrite ..... **Error! Bookmark not defined.**
PlxMuHostOutboundIndexRead ..... **Error! Bookmark not defined.**
PlxMuHostOutboundIndexWrite ..... **Error! Bookmark not defined.**
PlxMuInboundPortRead ..... **Error! Bookmark not defined.**
PlxMuInboundPortWrite ..... **Error! Bookmark not defined.**
PlxMuOutboundPortRead ..... **Error! Bookmark not defined.**
PlxMuOutboundPortWrite ..... **Error! Bookmark not defined.**
PlxPciAbortAddrRead ..... **Error! Bookmark not defined.**
PlxPciBarRangeGet ..... **Error! Bookmark not defined.**
PlxPciBaseAddressesGet ..... **Error! Bookmark not defined.**
PlxPciBusSearch ..... **Error! Bookmark not defined.**
PlxPciCommonBufferGet ..... **Error! Bookmark not defined.**
PlxPciConfigRegisterRead ..... **Error! Bookmark not defined.**
PlxPciConfigRegisterReadAll ..... **Error! Bookmark not defined.**
PlxPciConfigRegisterWrite ..... **Error! Bookmark not defined.**
PlxPciDeviceClose ..... **Error! Bookmark not defined.**
PlxPciDeviceFind ..... **Error! Bookmark not defined.**
PlxPciDeviceOpen ..... **Error! Bookmark not defined.**
PlxPmIdRead ..... **Error! Bookmark not defined.**
PlxPmNcpRead ..... **Error! Bookmark not defined.**
PlxPowerLevelGet ..... **Error! Bookmark not defined.**
PlxPowerLevelSet ..... **Error! Bookmark not defined.**
PlxRegisterDoorbellRead ..... **Error! Bookmark not defined.**
PlxRegisterDoorbellSet ..... **Error! Bookmark not defined.**
PlxRegisterMailboxRead ..... **Error! Bookmark not defined.**
PlxRegisterMailboxWrite ..... **Error! Bookmark not defined.**
PlxRegisterRead ..... **Error! Bookmark not defined.**
PlxRegisterReadAll ..... **Error! Bookmark not defined.**
PlxRegisterWrite ..... **Error! Bookmark not defined.**
PlxSdkVersion ..... **Error! Bookmark not defined.**

PlxSerialEepromPresent ..... **Error! Bookmark not defined.**
PlxSerialEepromRead ..... **Error! Bookmark not defined.**
PlxSerialEepromWrite ..... **Error! Bookmark not defined.**
PlxUserRead ..... **Error! Bookmark not defined.**
PlxUserWrite ..... **Error! Bookmark not defined.**
PlxVpdIdRead ..... **Error! Bookmark not defined.**
PlxVpdNcpRead ..... **Error! Bookmark not defined.**
PlxVpdRead ..... **Error! Bookmark not defined.**
PlxVpdWrite ..... **Error! Bookmark not defined.**

4 IOP API 3-4

5 PCI SDK Data Structures Used by API 5-1

5.1 Sample Data Structure 5-1
SAMPLE structure 5-1
5.2 Details of data structures 5-2
ADDRESS, SDATA and UDATA Data Types 5-3
BOOLEAN Types 5-4
S8 and U8 Data Types 5-5
S16 and U16 Data Types 5-6
S32 and U32 Data Types 5-7
U64 Data Type 5-8
Access Type Enumerated Data Type 5-9
API Parameters Structure 5-10
BAR Space Enum Data Type 5-13
Bus Index Enum Data Type 5-14
Device Location Data Type 5-15
DMA Channel Descriptor Structure 5-16
DMA Channel Enum Data Type 5-21
DMA Channel Priority Enum Data Type 5-22
DMA Command Enum Data Type 5-23
DMA Direction Enum Data Type 5-24
DMA Resource Manager Parameters Structure 5-25
DMA Transfer Element Structure And SGL Address Structure 5-26
Echo State Enum Data Type 5-31
EEPROM Type Enum Data Type 5-32
Hot Swap Status Definition 5-33
IOP Arbitration Descriptor Structure 5-34

IOP Bus Properties Structure 5-36
IOP Endian Descriptor Structure 5-44
IOP Space Enum Data Type 5-48
Mailbox ID Enum Data Type 5-50
PCI Arbitration Descriptor Structure 5-51
PCI Bus Properties Structure 5-52
PCI Memory Data Type 5-55
PCI Space Enum Data Type 5-56
Pin State Enum Data Type 5-57
PLX Interrupt Structure 5-58
Power Level Enum Data Type 5-65
Power Management Properties Structure 5-66
Serial Port Descriptor Structure 5-68
SPU Status Structure 5-70
USER Pin Direction Enum Data Type 5-72
USER Pin Enum Data Type 5-73
Virtual Addresses Data Type 5-74
MANAGEMENTDATA structure 5-75
REGISTERDATA structure 5-76
BUSIOPDATA structure 5-77
DMADATA structure 5-78
MISCDATA structure 5-79
IOCTLDATA union 5-80

Appendix A. PLX SDK Revision Notes ..... A

1 General Information

The PCI SDK included in the development package is a powerful aid to software designers.

The PCI Host SDK (described by this document) includes a powerful Application Programmer's Interface (API) for Win32 Host platforms, sample Host applications, and device drivers for NT/98/2000.

Using the Host API (and its underlying use of the PLX drivers), frequently a user application can perform all necessary access and control of the PLX chip across the PCI bus, with no further driver development being necessary.

The Host SDK also includes PLXMon, a graphical application for controlling PLX chips. PLXMon uses the PLX API and drivers to access the PLX chips across the PCI bus.

The PCI Pro SDK includes the complete Host SDK, and in addition also includes a powerful Application Programmer's Interface (API) for the I/O Platform (IOP), sample IOP applications, Board Support Packages, and RTOS support. Please see the Pro SDK documentation for details.

We are confident that with the PCI SDK, your designs will be brought to market faster and more efficiently.

1.1 About This Manual

This manual provides design information on using the Windows host-side PLX APIs (often described in this document as PCI API).

PLX also offers, in the SDK Pro, equivalent IOP-side (or local) APIs for use from the I/O platform (RDK or user board, containing the PLX chip and processor functionality, either on chip or as separate CPU). This "IOP API" will be described in a separate document. As of the 3.1 release of the Host SDK, the IOP API from SDK 3.0 is current, and fully compatible with the 3.1 Host API and drivers.

Note: *This manual assumes that the user has read and is familiar with the PCI Host SDK User's Manual.*

1.2 Where To Go From Here

The following is a brief summary of the chapters to help guide your reading of this manual:

Chapter 2, Creating Applications Using The PLX API, is an explanation of how to create a Windows application using the PLX PCI API.

Chapter 3, Host PCI API, provides a description of all the (Host) PCI API functions.

Chapter 5, PCI SDK Data Structures Used by API provides a description of the data structures used in the PCI SDK.

1.3 Conventions

Please note that when software samples are provided the following notations are used:

- *italics* are used to represent variables, function names, and program names; and,
- `courier` is used to represent source code given as examples.

1.3.1 Programming Conventions

Some designers may not be familiar with our programming conventions. Therefore, a few conventions have been noted below:

- *PU32 data* is analogous to *U32 *data* or *unsigned long *data;* and,
- *IN* and *OUT* are used to distinguish between parameters that are being passed into API functions and parameters that are being returned by API functions.

1.4 Terminology

All references to Windows NT assume Windows NT 4.0 and may be denoted as WinNT.

All references to Windows 98 may be denoted as Win98.

References to Windows 2000 may be denoted as Win2K.

Win32 references are used throughout this manual to mean any application that is compatible with the Windows 32-bit environment.

All references to IOP (I/O Platform) throughout this manual denote the a Custom board with a PLX chip or a PLX RDK board and all references to IOP software denote the software running on a board.

1.5 Customer Support

Prior to contacting PLX customer support, please ensure that you are situated close to the computer that has the PCI SDK installed and have the following information:

1. Model number of the PLX PCI RDK (if any);
1. PLX PCI SDK version (if any);
2. Host Operating System and version;
3. Description of your intended design:
    - PLX chip used
    - Microprocessor
    - Local Operating System and version (if any)
    - I/O
4. Description of your problem; and
5. Steps to recreate the problem.

You may contact PLX customer support at:

Address: PLX Technology, Inc.
Attn. Technical Support
390 Potrero Avenue
Sunnyvale, CA 94086
Phone: 408-774-9060
Fax: 408-774-2169
Web: http://www.plxtech.com

You may send email to one of the following addresses:

- west-apps@plxtech.com
- mid-apps@plxtech.com
- east-apps@plxtech.com
- euro-apps@plxtech.com
- asia-apps@plxtech.com

2 Creating Applications Using The PLX API

2.1 Host (Windows) Applications

The PCI Host SDK (described by this document) includes a powerful Application Programmer's Interface (API) for Win32 Host platforms, sample Host applications, and device drivers for NT/98/2000.

Using the Host API (and its underlying use of the PLX drivers), frequently a user application can perform all necessary access and control of the PLX chip across the PCI bus, with no further driver development being necessary.

To use the PLX Host API, a user application is linked to the import library, PLXAPI.lib. At runtime the executable will invoke functions in PLXAPI.dll, which should have been installed in the Windows executable path by the installer.

2.1.1 Sample Applications

Several sample applications are included with the Host SDK. They are located in the "<INSTALLPATH>\win32\samples" directories. These demonstrate how an application can use the included API (with its underlying access to the PLX drivers) to control PLX PCI devices across the PCI bus. Project files are included for Microsoft Visual C++.

2.1.2 Creating a Host Application

The procedure for creating a Windows application using the PCI API will be presented along with some design considerations. The steps for creating a Windows application will be described into two sections being:

- **2.1.2.1 Creating A Microsoft Developer's Studio Project File**, which describes the steps involved in creating a workspace file and the environment setup.
- **2.1.2.2 Building A Custom Application**, which describes the steps for building a custom Win32 application for use with the PCI API.

2.1.2.1 Creating A Microsoft Developer's Studio Project File

The main steps to creating a new workspace file are as follows:

1. On the File menu tab, choose New… item.
2. Create a new project file by choosing the Projects tab.
3. Choose a Win32 Application project file. Ensure that the Create New Workspace button is set.
4. Choose a project name and destination location for the project.
5. Set up the project environment.
   - Under the Project menu of Developer Studio, choose Settings... This brings up the project settings dialog box.
   - Choose the C/C++ tab.
   - Change the Category type to Preprocessor.
   - In the Preprocessor definitions window add the following entries:
     - `PCI_CODE`
     - `LITTLE_ENDIAN`
   - In the Additional include directories add a path to the PLX SDK include directory. ***Note:*** *It is recommended that a relative path from the workspace project directory to the PCI SDK include directory be given if both the include directory and the workspace project are located on the same logical Windows Disk Drive. This makes the workspace project more compatible between various Windows Systems and Windows Disk Drive Mappings.*
6. Create the application that will use the PCI API.
7. Include all the source files into the project. Copy the `PlxApi.lib` file in the PCI SDK directory into the workspace project directory. Include the `PlxApi.lib` file into the workspace project files.
8. Build the application.

2.1.2.2 Building A Custom Application

To start building a custom application with the PCI API, create a new project for the application (see section 2.1.2.1 for more information). To start accessing a PCI reference design board using the PCI API follow these steps:

1. Include the following headers:

   - **Plx.h** contains some constants that are common for all development platforms;
   - **PciApi.h** contains the PCI API function prototypes

2. Connect to an IOP (get a device driver handle). Handles to IOPs are always obtained via the *PlxPciDeviceOpen()* function. Sometimes, the *PlxPciDeviceFind()* can help to find device driver handle for a specific installed IOP. There are several methods for selecting a board (using the device location structure):

   - **Method 1:** Using the device's serial number. The serial number is always in this format: `<devicedrivername>-<indexnumber>` where `<device driver name>` is the name of the respective device driver for the reference design board and `<index number>` is its unique number based on the order in which the device driver found the reference design board, for example,'pci9080-1'.
     Then a call to *PlxPciDeviceOpen()* can be performed, using the serial number as the `SerialNumber` member of the device location argument.

   - **Method 2:** Using a vendor ID, device ID, bus number and/or slot number combination. In this case the `SerialNumber` member of the device location structure has to be set to an empty string (""). Then a call to *PlxPciDeviceOpen()* can be performed. *PlxPciDeviceOpen()* will return a device driver handle to the first reference design board matching the search criteria. See the Host API *PlxPciDeviceOpen()* function for more details.

   - **Method 3:** When more than one IOP matches the search criteria, *PlxPciDeviceFind()* may be useful. Here is a procedure capable of listing all the matching reference design boards:

   a. Call *PlxPciDeviceFind()* with the device location structure set with the search criteria (with the `SerialNumber` structure set to a NULL string) and the second argument set to FIND_AMOUNT_MATCHED. This will give you the number of matching reference design boards.

   b. Call *PlxPciDeviceFind()* for each reference design board found. The device location structure returned will give you completed information about the reference design board.

   c. Call *PlxPciDeviceOpen()* with the desired device location as argument.

3. Make connections to other reference design boards if necessary. Repeat Step 2.

4. Start using the PCI API function calls with the appropriate device driver handles.

5. When the application completes close connections to all reference design boards in use with *PlxPciDeviceClose()*.

2.2 IOP Applications

For details on building applications on the I/O Platform using the PLX IOP API, see the PCI Pro SDK documentation.

3 Host PCI API

The PCI API is designed around the features of the PLX chips.

3.1 PCI API Function Quick Reference List

The following table lists all the PCI API functions available. Designers should consult Section 0 for detailed description of the API functions.

Legends for the "Chip Support" Column:

8: PCI 9080

5: PCI 9054

4: IOP 480

3: PCI 9030

API Function Name	Chip Support	Purpose	Page
PlxBusIopRead	8, 5, 4, 3	Read data from the IOP bus.	3-4
PlxBusIopWrite	8, 5, 4, 3	Write data to the IOP bus.	3-6
PlxDmaBlockChannelClose	8, 5, 4	Close a DMA channel for Block DMA.	3-8
PlxDmaBlockChannelOpen	8, 5, 4	Open a DMA channel for Block DMA.	3-10
PlxDmaBlockTransfer	8, 5, 4	Start a DMA Block transfer.	3-12
PlxDmaBlockTransferRestart	8, 5, 4	Restart a previous DMA Block transfer.	3-15
PlxDmaControl	8, 5, 4	Perform a supplied command to the DMA channel given.	3-17
PlxDmaSglChannelClose	8, 5, 4	Close a DMA channel for SGL DMA.	3-19
PlxDmaSglChannelOpen	8, 5, 4	Open a DMA channel for Scatter-Gather DMA.	3-21
PlxDmaSglTransfer	8, 5, 4	Start a SGL transfer.	3-23
PlxDmaShuttleChannelClose	8, 5, 4	Close a DMA channel for Shuttle DMA.	3-25
PlxDmaShuttleChannelOpen	8, 5, 4	Open a DMA channel for Shuttle DMA.	3-27
PlxDmaShuttleTransfer	8, 5, 4	Start a Shuttle DMA transfer.	3-29
PlxDmaStatus	8, 5, 4	Return the status of a DMA channel.	3-31
PlxHotSwapIdRead	5, 4, 3	Read the capability's ID Byte.	3-33
PlxHotSwapNcpRead	5, 4, 3	Read the next capability pointer location	3-34
PlxHotSwapStatus	5, 4, 3	Return the status of the Hot Swap Registers.	3-35
PlxIntrAttach	8, 5, 4, 3	Attach a wait event to a PLX interrupt.	3-36
PlxIntrDisable	8, 5, 4, 3	Disable specific PLX interrupts.	3-38
PlxIntrEnable	8, 5, 4, 3	Enable specific PLX interrupts.	3-40
PlxIntrStatusGet	8, 5, 4, 3	Get the status of the PLX interrupts.	3-42
PlxIoPortRead	8, 5, 4, 3	Read data from an I/O port.	3-44
PlxIoPortWrite	8, 5, 4, 3	Write data to an I/O port.	3-46
PlxMuHostOutboundIndexRead	4	Read the Host Outbound Index register.	3-48
PlxMuHostOutboundIndexWrite	4	Write to the Host Outbound Index register.	3-49
PlxMuInboundPortRead	8, 5, 4	Read the Inbound Port.	3-50
PlxMuInboundPortWrite	8, 5, 4	Write to the Inbound Port.	3-51
PlxMuOutboundPortRead	8, 5, 4	Read the Outbound Port.	3-52

API Function Name	Chip Support	Purpose	Page
PlxMuOutboundPortWrite	8, 5, 4	Write to the Outbound Port.	3-53
PlxPciAbortAddrRead	4	Read the PCI Abort Address Location.	3-54
PlxPciBarRangeGet	8, 5, 4, 3	Get the memory range for a base address register.	3-55
PlxPciBaseAddressesGet	8, 5, 4, 3	Get the virtual addresses for the base address registers.	3-57
PlxPciBusSearch	8, 5, 4, 3	Search for a PLX device on the PCI bus.	3-59
PlxPciCommonBufferGet	8, 5, 4	Get the information for the device driver's common buffer.	3-61
PlxPciConfigRegisterRead	8, 5, 4, 3	Read a configuration register of a PCI device.	3-63
PlxPciConfigRegisterReadAll	8, 5, 4, 3	Read all the Configuration registers of a PCI device.	3-65
PlxPciConfigRegisterWrite	8, 5, 4, 3	Write to a Configuration register of a PCI device.	3-67
PlxPciDeviceClose	8, 5, 4, 3	Close a PLX device channel.	3-69
PlxPciDeviceFind	8, 5, 4, 3	Find a PLX device on the PCI bus.	3-71
PlxPciDeviceOpen	8, 5, 4, 3	Open a PLX device channel.	3-73
PlxPmIdRead	5, 4, 3	Read the Power Management ID Byte.	3-75
PlxPmIdNcpRead	5, 4, 3	Read the next capability pointer location.	3-76
PlxPowerLevelGet	8, 5, 4, 3	Get the power level.	3-77
PlxPowerLevelSet	8, 5, 4, 3	Set the power level.	3-79
PlxRegisterDoorbellRead	8, 5, 4	Read from then clear a Doorbell register.	3-81
PlxRegisterDoorbellSet	8, 5, 4	Set a Doorbell register.	3-82
PlxRegisterMailboxRead	8, 5, 4	Read a Mailbox register.	3-84
PlxRegisterMailboxWrite	8, 5, 4	Write to a Mailbox register.	3-86
PlxRegisterRead	8, 5, 4, 3	Read a register.	3-88
PlxRegisterReadAll	8, 5, 4, 3	Read a register group.	3-90
PlxRegisterWrite	8, 5, 4, 3	Write to a register.	3-92
PlxSdkVersion	8, 5, 4, 3	Get the SDK Version numbers.	3-94
PlxSerialEepromPresent	8, 5, 4, 3	Determine if a Serial EEPROM is on a device.	3-95
PlxSerialEepromRead	8, 5, 4, 3	Read the serial EEPROM.	3-96
PlxSerialEepromWrite	8, 5, 4, 3	Write to the serial EEPROM.	3-98
PlxUserRead	8, 5, 4	Read a USER pin value.	3-100
PlxUserWrite	8, 5, 4	Write a value to a USER pin.	3-102
PlvVpdIdRead	5, 4, 3	Read the capability's ID byte.	3-104
PlxVpdNcpRead	5, 4, 3	Read the next capability pointer location.	3-105
PlxVpdRead	5, 4, 3	Read a U32 value from the Vital Product Data Register Port.	3-106
PlxVpdWrite	5, 4, 3	Write a U32 value to the Vital Product Data Register Port.	3-107

3.2 PCI API Feature List

Feature	Related API Calls	Page	Related Data Structure (Page number)
Interrupt Control	PlxIntrAttach	3-36	PLX Interrupt Structure (5-58)
	PlxIntrDisable	3-38	
	PlxIntrEnable	3-40	
	PlxIntrStatusGet	3-42	
Flyby DMA	PlxDmaBlockChannelClose	3-8	DMA Channel Descriptor Structure (5-16) DMA Channel Enum Data type (5-21) DMA Command Enum Data Type (5-23) DMA Resource Manager Parameters Structure (5-25) DMA Transfer Element Structure (5-26)
	PlxDmaBlockChannelOpen	3-10	
	PlxDmaBlockTransfer	3-12	
Local to Local DMA	PlxDmaBlockChannelClose	3-8	DMA Channel Descriptor Structure (5-16) DMA Channel Enum Data type (5-21) DMA Command Enum Data Type (5-23) DMA Resource Manager Parameters Structure (5-25) DMA Transfer Element Structure (5-26)
	PlxDmaBlockChannelOpen	3-10	
	PlxDmaBlockTransfer	3-10	
Outbound Option	PlxMuHostOutboundRead	3-48	
	PlxMuHostOutboundWrite	3-49	
Hot Swap	PlxHotSwapIdRead	3-33	Hot Swap Status Definition (5-33)
	PlxHotSwapNcpRead	3-34	
	PlxHotSwapStatus	3-35	
Power Management	PlxPmIdRead	3-75	
	PlxPmNcpRead	3-76	
Vital Product Data	PlxVpdIdRead	3-104	
	PlxVpdNcpRead	3-105	
	PlxVpdRead	3-106	
	PlxVpdWrite	3-107	
Serial EEPROM	PlxSerialEepromPresent	3-95	EEPROM Type Enum Data Type (5-32)
	PlxSerialEepromRead	3-96	
	PlxSerialEepromWrite	3-98	

3.3 Sample Function Entry

The following sample entry lists each entry section and describes the information therein.

Sample_Function

Syntax:

`function(modifier` *`parameter`*`[,...]);`

This gives the declaration syntax for each function. Each parameter is *italicized*.

PLX Chip Support:

A list of PLX chips that support this function.

Description:

Summary of the function's purpose followed by the parameters it takes. Also includes any relevant information pertaining to the function.

Return Value:

The possible values returned by the function.

Notes:

Provides any relevant information pertaining to the function.

Usage:

A sample code is provided to demonstrate how the function is used. The sample code may need minor change in order to be incorporated into your code.

Cross Reference:

Provide page numbers to various associated data, structures.

3.4 API Function Details

This section contains a detailed description of each function in the API. The functions are listed in alphabetical order.

Note: Devices supported by PCI SDK Version 3.1: PCI 9080, PCI 9054, IOP 480, and PCI 9030.

PlxBusIopRead

Syntax:

```
RETURN_CODE
PlxBusIopRead(
    IN HANDLE      drvHandle,
    IN IOP_SPACE   iopSpace,
    IN U32         address,
    IN BOOLEAN     remapAddress,
    OUT PU32       destination,
    IN U32         transferSize,
    IN ACCESS_TYPE accessType
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads a range of values from the local bus of a PCI device containing a PLX chip (Direct Slave Read).

- *drvHandle* is the handle of the PCI device;
- *iopSpace* defines which Local Address Space register is used;
- *address* is the starting offset from the IOP space PCI remap address (*remapAddress*=TRUE) or the actual IOP bus address to start reading from (*remapAddress*=FALSE);
- *remapAddress* states how to treat the IOP address given. FALSE means that the IOP address is an offset that must be within the local space's addressable space (which is the inverse of its range value). If the *remp* value is set, any address within the 4 GB 32-bit address range is valid.;
- *destination* is a pointer to the storage buffer for the data read;
- *transferSize* defines the number of bytes to read from the IOP bus; and,
- *accessType* defines the access type size.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *destination* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInsufficientResources	There is no memory available for the PCI API.
ApiInvalidAccessType	The *accessType* size is not supported for this device.
ApiInvalidAddress	The *address* parameter is not aligned based on the *accessType* provided.
ApiInvalidSize	The *transferSize* parameter is 0 or is not aligned based on the *accessType* provided.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

If *remapAddress* is set to TRUE, this API function will adjust the IOP space window according to the *address* given.

This function is not recommended when performance is a concern. Internal error checking and local window adjustment code keeps performance less than optimal. For increased performance, use the *PlxPciBaseAddressGet()* function and access local memory directly through virtual addresses. For ultimate performance, use DMA functionality, if available, to alleviate the host processor.

Please make sure that local space descriptors are setup properly before using this function. Incorrect settings may result in incorrect data.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 length = 0x100;
U32 buf[0x40], localStartOffset = 0x20000;

rc = PlxBusIopRead(
         drvHandle,
         IopSpace0,
         localStartOffset,
         TRUE,       /* remap */
         (PU32)buf,
         length,
         BitSize32
         );
if (rc != ApiSuccess)
{
  printf("Error: Unable to read data.\n");
  return -1;
}
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	5-48
ACCESS_TYPE	5-9

PlxBusIopWrite

Syntax:

```
RETURN_CODE
PlxBusIopWrite(
    IN HANDLE      drvHandle,
    IN IOP_SPACE   iopSpace,
    IN U32         address,
    IN BOOLEAN     remapAddress,
    IN PU32        source,
    IN U32         transferSize,
    IN ACCESS_TYPE accessType
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Writes a range of values to the local bus of a PCI device containing a PLX chip (Direct Slave Write).

- *drvHandle* is the handle of the PCI device;
- *iopSpace* defines which Local Address Space register to used;
- *address* is the starting offset from the IOP space PCI remap address or the actual IOP bus address to start writing to;
- *remapAddress* states how to treat the IOP address given. FALSE means that the IOP address is an offset that must be within the local space's addressable space (which is the inverse of its range value). If the *remp* value is set, any address within the 4 GB 32-bit address range is valid.;
- *source* is a pointer to the data buffer;
- *transferSize* defines the number of bytes to write to the IOP bus; and,
- *accessType* defines the access type size.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* or the *source* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInsufficientResources	There is no memory available for the PCI API.
ApiInvalidAccessType	The *accessType* size is not supported for this device.
ApiInvalidAddress	The *address* parameter is not aligned based on the *accessType* provided.
ApiInvalidSize	The *transferSize* parameter is 0 or is not aligned based on the *accessType* provided.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

If *remapAddress* is set to TRUE, this API function will adjust the IOP space window according to the *address* given.

This function is not recommended when performance is a concern. Internal error checking and local window adjustment code keeps performance less than optimal. For increased performance, use the *PlxPciBaseAddressGet()* function and access local memory directly through virtual addresses. For ultimate performance, use DMA functionality, if available, to alleviate the host processor.

Please make sure that local space descriptors are setup properly before using this function. Incorrect settings may result in incorrect data.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 length = 0x100;
U32 buf[0x40], localStartOffset = 0x20000;

/* Clear buffer */
memset(buf, 0, length);

/*
  Now clear an IOP buffer starting at localStartOffset
  over a length of 0x100.
*/

rc = PlxBusIopWrite(
         drvHandle,
         IopSpace0,
         localStartOffset,
         TRUE,
         (PU32)buf,
         length,
         BitSize32
         );
if (rc != ApiSuccess)
{
  printf("Error: Unable to write data.\n");
  return -1;
}
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	5-48
ACCESS_TYPE	5-9

PlxDmaBlockChannelClose

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelClose(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Closes the Block DMA channel.

- *drvHandle* is the handle of the PCI device; and,
- *dmaChannel* is the DMA channel number previously opened.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiDmaChannelUnavailable	The DMA channel was not opened for Block DMA.
ApiDmaInProgress	A DMA transfer is in progress.
ApiDmaPaused	The DMA channel is paused.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaBlockChannelOpen()*.

Usage:

```
HANDLE      drvHandle;
RETURN_CODE rc;

rc = PlxDmaBlockChannelClose(
         drvHandle,
         PrimaryPciChannel0
         );
if (rc != ApiSuccess)
{
  printf("Unable to close DMA channel, error code = %d\n”, rc);
  return;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21

PlxDmaBlockChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelOpen(
    IN HANDLE            drvHandle,
    IN DMA_CHANNEL       dmaChannel,
    IN PDMA_CHANNEL_DESC dmaChannelDesc
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Opens and initializes a DMA channel for Block DMA transfers. If *IopChannel2* is selected then this function can be used to initialize Flyby DMA or Local2Local DMA transfers on IOP 480.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiDmaChannelInvalid	This *dmaChannel* parameter is not supported by this PLX chip.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel is not closed.
ApiDmaInvalidChannelPriority	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

If *dmaChannelDesc* parameter is NULL, the function uses the current setting for the channel.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
DMA_CHANNEL_DESC desc;

memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));
/* Set up DMA configuration structure */
desc.EnableReadyInput    = 1;
```

```
desc.DmaStopTransferMode = AssertBLAST;
desc.DmaChannelPriority  = Rotational;   /* rotational priority */
desc.IopBusWidth         = 3;            /* 32 bit bus */

rc = PlxDmaBlockChannelOpen(
         drvHandle,
         PrimaryPciChannel0,
         &desc
         );
if (rc != ApiSuccess)
{
  printf ("Unable to open DMA channel, error code = %d\n", rc);
  return;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
DMA_CHANNEL_DESC	5-16

PlxDmaBlockTransfer

Syntax:

```
RETURN_CODE
PlxDmaBlockTransfer(
    IN HANDLE                drvHandle,
    IN DMA_CHANNEL           dmaChannel,
    IN DMA_TRANSFER_ELEMENT *dmaData,
    IN BOOLEAN               returnImmediate
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Starts the Block DMA transfer for a given DMA channel.

- *drvHandle* is the handle of the PCI device
- *dmaChannel* is the DMA channel number previously opened
- *dmaData* is a pointer to the data for the DMA transfer (refer to the table below)
- *returnImmediate* determines if this function returns immediately after the DMA command is completed.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiFailed	Internal API synchronization resources could not be allocated.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiNullParam	The *drvHandle* or *dmaData* (if *dmaCommand* is not `DmaStart`) parameters are NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel was not opened for Block DMA.
ApiDmaInProgress	The DMA channel is in progress.
ApiPciTimeout	No interrupt was received to signal the end of a synchronous transfer.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaBlockChannelOpen()*.

Block DMA transfers are useful with contiguous host buffers described by a *PCI address*. DMA channels require valid PCI physical addresses, not user or virtual addresses. Virtual addresses are those returned by *malloc()*, for example, or a static buffer in the application. The physical address of the Common buffer provided by PLX drivers (*refer to PciCommonBufferGet()*) is a valid PCI address.

The DMA done interrupt is automatically enabled when this function is called. This allows the PLX driver to perform cleanup tasks after the DMA transfer has completed.

The DMA_TRANSFER_ELEMENT structure contains members whose meanings may differ or even be ignored depending on the DMA transfer type selected by the calling function.

DMA_TRANSFER_ELEMENT structure:

Structure Element	Signification
UserAddr2	Ignored.
LowPciAddr	Low PCI address of the PCI buffer.
SourceAddr	Local2Local Channel2 source address -not used in Flyby mode.
HighPciAddr	(Pci9054, IOP480 only) High PCI address of the PCI buffer.
IopAddr	The IOP address for the transfer.
DestAddr	Local2Local Channel2 destination address -This register is read for Flyby writes to I/O devices, and is read for Flyby reads from I/O devices.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Indicates if an interrupt is to be generated when the DMA transfer is done. A 1 means yes.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

```
HANDLE plxHandle;
PCI_MEMORY PciMemory;
DMA_TRANSFER_ELEMENT dmaData;

rc = PlxPciCommonBufferGet(
         plxHandle,
         &PciMemory
         );
if (rc != ApiSuccess)
{
   printf("ERROR: Unable to get Common Buffer information\n");
   return;
}

dmaData.Pci9080Dma.LowPciAddr        = PciMemory.PhysicalAddr;
dmaData.Pci9080Dma.IopAddr           = 0x10010000;
dmaData.Pci9080Dma.TransferCount     = 0x1000;
dmaData.Pci9080Dma.IopToPciDma       = 0;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
rc = PlxDmaBlockTransfer(
         plxHandle,
         PrimaryPciChannel0,
         &dmaData,
         FALSE       /* return upon completion */
         );
```

```
if (rc != ApiSuccess)
{
    printf("ERROR: Unable to perform DMA transfer, code = %d\n", rc);
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
PDMA_TRANSFER_ELEMENT	5-26

PlxDmaBlockTransferRestart

Syntax:

```
RETURN_CODE
PlxDmaBlockTransferRestart(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel,
    IN U32         transferSize,
    IN BOOLEAN     returnImmediate
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Restarts the Block DMA transfer for a pre-programmed DMA channel.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number previously opened and programmed;
- *transferSize* is the DMA transfer size; and,
- *returnImmediate* determines if this function waits for the DMA command to complete before returning.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiFailed	Internal API synchronization resources could not be allocated.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiDmaChannelUnavailable	The DMA channel was not opened for Block DMA.
ApiDmaInProgress	The DMA channel is in progress.
ApiPciTimeout	No interrupt was received to signal the end of a synchronous transfer.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaBlockChannelOpen()*.

Before calling this function the appropriate DMA channel must be successfully programmed using *PlxDmaBlockTransfer()*.

Usage:

```
RETURN_CODE rc;
```

```
HANDLE plxHandle;
U32 totalSize = 0x100;


rc = PlxDmaBlockTransferRestart(
         plxHandle,
         PrimaryPciChannel0,
         totalSize,
         TRUE
         );
if (rc != ApiSuccess)
{
    PlxPrintf("Restart failed\n");
    return -1;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21

PlxDmaControl

Syntax:

```
RETURN_CODE PlxDmaControl(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel,
    IN DMA_COMMAND dmaCommand
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Controls the DMA transfer for a given DMA channel.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number previously opened;
- *dmaCommand* is the action to perform on this DMA channel;

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this chip.
ApiDmaChannelUnavailable	The DMA channel was not opened by the selected device.
ApiDmaNotPaused	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused). Also will occur when attempting to pause a Flyby DMA transfer.
ApiDmaCommandInvalid	The *dmaCommand* parameter is invalid.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaBlockChannelOpen()*.

Usage:

```
HANDLE plxHandle;
PCI_MEMORY PciMemory;
DMA_TRANSFER_ELEMENT dmaData;

/* PciMemory contains a valid PCI address */
```

```
/* zero out the structure */
memset(&dmaData, 0, sizeof(dmaData));

dmaData.Pci9080Dma.LowPciAddr        = PciMemory.PhysicalAddr;
dmaData.Pci9080Dma.IopAddr           = 0x10010000;
dmaData.Pci9080Dma.TransferCount     = 0x1000;
dmaData.Pci9080Dma.IopToPciDma       = 0;
dmaData.Pci9080Dma.TerminalCountIntr = 0;

rc = PlxDmaBlockTransfer(
         plxHandle,
         PrimaryPciChannel0,
         &dmaData,
         FALSE          /* return upon completion */
         );
If (rc != ApiSuccess)
{
  printf ("\n\nErrors in Block Dma.\n");
  printf ("Returned code %d\n",rc);
  return -1;
}

rc = PlxDmaControl(
         plxHandle,
         PrimaryPciChannel0,
         DmaPause
         );
If (rc = != ApiSuccess)
{
  printf ("\n\nCould not pause DMA transfer.\n");
  printf ("Returned code %d\n",rc);
  return -1;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
DMA_COMMAND	5-23

PlxDmaSglChannelClose

Syntax:

```
RETURN_CODE
PlxDmaSglChannelClose(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Closes the Scatter-Gather DMA channel.

- *drvHandle* is the handle of the PCI device; and,
- *dmaChannel* is the DMA channel number previously opened.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiDmaChannelUnavailable	The DMA channel was not opened for Scatter-Gather DMA.
ApiDmaInProgress	A DMA transfer is in progress.
ApiDmaPaused	The DMA channel is paused.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaSglChannelOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

rc = PlxDmaSglChannelClose(
         drvHandle,
         PrimaryPciChannel0
         );
if (rc != ApiSuccess)
{
  printf ("ERROR: Unable to close DMA channel, code = %d\n", rc);
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21

PlxDmaSglChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaSglChannelOpen(
    IN HANDLE            drvHandle,
    IN DMA_CHANNEL       dmaChannel,
    IN PDMA_CHANNEL_DESC dmaChannelDesc
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Opens and initializes a DMA channel for Scatter-Gather DMA transfers.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiDmaChannelInvalid	This *dmaChannel* parameter is not supported by this PLX chip.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel is not closed.
ApiDmaInvalidChannelPriority	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

If *dmaChannelDesc* parameter is NULL, the function uses the current setting for the channel.

The DMA done interrupt is automatically enabled when this function is called. This allows the PLX driver to perform cleanup tasks after the DMA transfer has completed.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
DMA_CHANNEL_DESC desc;

memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));
```

```
/* Set up DMA configuration structure */
desc.EnableReadyInput    = 1;
desc.DmaStopTransferMode = AssertBLAST;
desc.DmaChannelPriority  = Rotational;   /* rotational priority */
desc.IopBusWidth         = 3;            /* 32 bit bus */

rc = PlxDmaSglChannelOpen(
         drvHandle,
         PrimaryPciChannel0,
         &desc
         );
if (rc != ApiSuccess)
{
  printf ("Unable to open DMA channel, error code = %d\n", rc);
  return;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
PDMA_CHANNEL_DESC	5-26

PlxDmaSglTransfer

Syntax:

```
RETURN_CODE
PlxDmaSglTransfer(
    IN HANDLE                 drvHandle,
    IN DMA_CHANNEL            dmaChannel,
    IN PDMA_TRANSFER_ELEMENT  dmaData,
    IN BOOLEAN                returnImmediate
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Starts a Scatter-Gather DMA transfer for a given DMA channel.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number previously opened;
- *dmaData* is the data for the DMA transfer (see the table in the comments section); and,
- *returnImmediate* determines if this function returns immediately after the DMA command is completed.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel was not opened for SGL DMA.
ApiDmaInProgress	The DMA channel is in progress.
ApiPciTimeout	No interrupt was received to signal the end of a synchronous transfer.
ApiFailed	Failed to start the DMA transfer. ***Note:*** *Ensure that there is enough memory to build the Scatter-Gather List. Memory for the Scatter-Gather List can be added by increasing the* `MaxSglTransferSize` *registry entry.*

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaSglChannelOpen()*.

When this function is called, the PLX driver takes the provided user-mode buffer and locks it into memory. The driver then determines the physical address of each memory block fragment that makes up the buffer

and sets up a descriptor for each. The descriptor and other cleanup tasks are performed after the SGL interrupt, noting SGL transfer completion.

The DMA_TRANSFER_ELEMENT structure contains members whose meanings may differ or even be ignored depending on the DMA transfer type selected by the calling function.

DMA_TRANSFER_ELEMENT structure:

Structure Element	Signification
UserAddr	User address of the PCI buffer.
LowPciAddr	Ignored
SourceAddr	Ignored
HighPciAddr	Ignored
IopAddr	The IOP address for the transfer.
DestAddr	Ignored
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Ignored.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

```
HANDLE plxHandle;
U8 buffer[0x10000];
DMA_TRANSFER_ELEMENT dmaData;

dmaData.Pci9080Dma.UserAddr      = buffer;
dmaData.Pci9080Dma.IopAddr       = 0x10040000;
dmaData.Pci9080Dma.TransferCount = 0x10000;
dmaData.Pci9080Dma.IopToPciDma   = 1;

rc = PlxDmaSglTransfer(
         plxHandle,
         PrimaryPciChannel0,
         &dmaData,
         FALSE      /* return upon completion */
         );
if (rc != ApiSuccess)
    printf("ERROR: Unable to perform SGL transfer, code = %d\n", rc);
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
PDMA_TRANSFER_ELEMENT	5-26

PlxDmaShuttleChannelClose

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelClose(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Closes the Shuttle DMA channel.

- *drvHandle* is the handle of the PCI device; and,
- *dmaChannel* is the DMA channel number previously opened.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiDmaChannelUnavailable	The DMA channel was not opened for Shuttle DMA.
ApiDmaInProgress	A DMA transfer is in progress.
ApiDmaPaused	The DMA channel is paused.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaShuttleChannelOpen()*.

Usage:

```
HANDLE      drvHandle;
RETURN_CODE rc;


rc = PlxDmaShuttleChannelClose(
         drvHandle,
         PrimaryPciChannel0
         );
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to close DMA channel, code = %d\n", rc);
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21

PlxDmaShuttleChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelOpen(
    IN HANDLE            drvHandle,
    IN DMA_CHANNEL       dmaChannel,
    IN PDMA_CHANNEL_DESC dmaChannelDesc
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Opens and initializes a DMA channel for Shuttle DMA transfers.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiDmaChannelInvalid	This *dmaChannel* parameter is not supported by this PLX chip.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel is not closed.
ApiDmaInvalidChannelPriority	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

If *dmaChannelDesc* parameter is NULL, the function uses the current setting for the channel.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
DMA_CHANNEL_DESC desc;

memset(&desc, 0, sizeof(DMA_CHANNEL_DESC));
```

```
/* Set up DMA configuration structure */
desc.EnableReadyInput    = 1;
desc.DmaStopTransferMode = AssertBLAST;
desc.DmaChannelPriority  = Rotational;  /* rotational priority */
desc.IopBusWidth         = 3;           /* 32 bit bus */

rc = PlxDmaShuttleChannelOpen(
         drvHandle,
         PrimaryPciChannel0,
         &desc
         );
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to open DMA channel, code = %d\n", rc);
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
DMA_CHANNEL_DESC	5-26

PlxDmaShuttleTransfer

Syntax:

```
RETURN_CODE
PlxDmaShuttleTransfer(
    IN HANDLE                drvHandle,
    IN DMA_CHANNEL           dmaChannel,
    IN PDMA_TRANSFER_ELEMENT dmaData
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Starts a Shuttle DMA transfer for a given DMA channel.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number previously opened;
- *dmaData* is the data for the DMA transfer (see the table in the comments section).

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this PLX chip.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* or *dmaData* (if *dmaCommand* is not `DmaStart`) parameters are NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiDmaChannelUnavailable	The DMA channel was not opened for Shuttle DMA.
ApiDmaInProgress	The DMA channel is in progress.
ApiFailed	Failed to start the DMA transfer. ***Note:*** *Ensure that there is enough memory to build the Scatter-Gather List. Memory for the Scatter-Gather List can be added by increasing the* `MaxSglTransferSize` *registry entry.*

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using *PlxDmaShuttleChannelOpen()*.

Shuttle transfers are similar to SGL transfers, except that the last SGL descriptor “points” back to the first descriptor. The PLX driver sets up Shuttle transfers similar to SGL transfers, except interrupts are not triggered since shuttle transfers are continuous. Refer to *PlxDmaSglTransfer()* notes for additional information.

The DMA_TRANSFER_ELEMENT structure contains members whose meanings may differ or even be ignored depending on the DMA transfer type selected by the calling function.

DMA_TRANSFER_ELEMENT structure:

Structure Element	Signification
UserAddr	User address of the PCI buffer.
LowPciAddr	Ignored
SourceAddr	Ignored
HighPciAddr	Ignored
IopAddr	The IOP address for the transfer.
DestAddr	Ignored
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Ignored.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

```
U8                  buffer[0x10000];
HANDLE              plxHandle;
DMA_TRANSFER_ELEMENT dmaData;

dmaData.Pci9080Dma.UserAddr      = buffer;
dmaData.Pci9080Dma.IopAddr       = 0x10040000;
dmaData.Pci9080Dma.TransferCount = 0x10000;
dmaData.Pci9080Dma.IopToPciDma   = 1;

rc = PlxDmaShuttleTransfer(
         plxHandle,
         PrimaryPciChannel0,
         &dmaData
         );
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to initiate Shuttle transfer, code = %d\n", rc);
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21
PDMA_TRANSFER_ELEMENT	5-26

PlxDmaStatus

Syntax:

```
RETURN_CODE
PlxDmaStatus(
    IN HANDLE      drvHandle,
    IN DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Returns the status for the DMA transfer of a given DMA channel.

- *drvHandle* is the handle of the PCI device; and,
- *dmaChannel* is the DMA channel number previously opened.

Return Value:

Return Value	Description
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiDmaChannelInvalid	The *dmaChannel* parameter is not supported by this chip.
ApiDmaChannelUnavailable	The DMA channel was not opened by the selected device.
ApiDmaDone	The DMA channel is done.
ApiDmaPaused	The DMA channel is paused.
ApiDmaInProgress	The DMA channel is in progress.
ApiDmaNotPaused	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused). Also will occur when attempting to pause a Flyby DMA transfer.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Before calling this function the appropriate DMA channel must be successfully opened using one of the *PlxDmaXxxChannelOpen()* functions.

This function can be used with any type of DMA transfer.

Usage:

```
HANDLE      plxHandle;
RETURN_CODE rc;

rc = PlxDmaBlockTransfer(
         plxHandle,
```

```
          PrimaryPciChannel0,
          &dmaData,
          TRUE     /* return upon completion */
          );
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to start DMA transfer, code = %d\n", rc);
  return -1;
}

rc = PlxDmaStatus(
          plxHandle,
          PrimaryPciChannel0
          );
switch (rc)
{
    case ApiDmaInProgress:
        printf("DMA transfer is in progress!\n");
        break;

    case ApiDmaDone
        printf("DMA transfer is complete!\n");
        break;

    case ApiDmaPaused
        printf("DMA transfer is paused!\n");
        break;
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	5-21

PlxHotSwapIdRead

Syntax:

```
U8
PlxHotSwapIdRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Reads the ID for the Hot Swap capability registers.

- *drvHandle* is the handle to the device selected; and,
- *pReturnCode* is a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiConfigAccessFailed	The driver was unable to read the configuration registers.

Usage:

```
U8          hotSwapId;
HANDLE      drvHandle;
RETURN_CODE rc;

hotSwapId = PlxHotSwapIdRead(
                drvHandle,
                &rc
                );

if (rc != ApiSuccess)
    printf("ERROR: Unable to read Hot Swap ID, code = %d\n", rc);
else
    printf("Hot Swap ID = 0x%x\n", hotSwapId);
```

PlxHotSwapNcpRead

Syntax:

```
U8
PlxHotSwapNcpRead(
    IN HANDLE       drvHandle,
    IN PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Reads the location of the next element in the linked list of capabilities. If this is the last item in the list, the value will return 0.

- *drvHandle* is the handle of the selected device; and,
- *pReturnCode* is a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.

Usage:

```
U8          nextElement;
HANDLE      drvHandle;
RETURN_CODE rc;

nextElement = PlxHotSwapNcpRead(
                  drvHandle,
                  &rc
                  );

if (rc != ApiSuccess)
    printf("ERROR: Unable to read Hot Swap NCP, code = %d\n", rc);
else
    printf("Hot Swap Next Capability Pointer = 0x%x\n", nextElement);
```

PlxHotSwapStatus

Syntax:

```
U8
PlxHotSwapStatus(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Returns the Hot-Swap status of the PLX chip.

- *drvHandle* is handle of the device selected.

Return Value:

If there is no error, the return value is an ORed combination of these bits:

Return Value	Description
HS_LED_ON	The Hot Swap LED is on.
HS_BOARD_REMOVED	The board is in process of being removed.
HS_BOARD_INSERTED	The board was inserted and is being initialized.
0xFF	Hot Swap is not supported or not present on the board

Usage:

```
U8     value;
HANDLE drvHandle;

value = PlxHotSwapStatus(drvHandle);
if (value == 0xFF)
  PlxPrintf("Function failed.\n");
else
{
    if (value & HS_LED_ON)
        PlxPrintf("The LED is on\n");
    if (value & HS_BOARD_REMOVED)
        PlxPrintf("The board is about to be removed\n");
    if (value & HS_BOARD_INSERTED)
        PlxPrintf("The board was just inserted\n");
}
```

PlxIntrAttach

Syntax:

```
RETURN_CODE
PlxIntrAttach(
    IN HANDLE   drvHandle,
    IN PLX_INTR intrTypes,
    OUT PHANDLE pEventHdl
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

This function is used by applications when they need to wait for an interrupt to occur. It will send an event handle to the device driver and wait until the event is set (the PCI API creates the event). The driver will set the event when the interrupt occurs. After an interrupt happens and signals the event this function would have to be called again in order to re-create another interrupt event.

- *drvHandle* is the handle of the PCI device;
- *intrTypes* is a structure containing the sources of interrupts associated to the event (contained within the overlapped structure); and,
- *pEventHdl* is a pointer to an event created by the PCI API.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pEventHdl* parameter is NULL.
ApiUnsupportedFunction	The PCI API function call is not supported by the current device.
ApiInsufficientResources	The number of attached events has been exceeded.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

When this function is used to notify an application of an interrupt event, an exception fault may arise after the call to *WaitForSingleObject()* returns. The application should handle the exception by enclosing the code within a try or __try{} handler. See the code sample below for more details.

Usage:

```
HANDLE      myPlxDevice;
HANDLE      eventHandle;
PLX_INTR    intfield;
```

```
RETURN_CODE rc;

/* Clear structure first */
memset(&intfield, 0, sizeof(PLX_INTR));

/* Attach to interrupt */
intfield.PciDmaChannel1   = 1;
rc = PlxIntrAttach(
         myPlxDevice,
         intfield,
         &eventHandle
         );
if (rc != ApiSuccess)
  printf("ERROR: Unable to attach to interrupt, code = %d\n", rc);

/* Enable DMA channel 1 interrupt */
rc = PlxIntrEnable(
         myPlxDevice,
         &intfield
         );
if (rc != ApiSuccess)
  printf("ERROR: Unable to enable interrupt\n, code = %d\n", rc);

/* Wait for signal at end of transfer */
__try
{
    val = WaitForSingleObject(
              eventHandle,
              5000     /* 5 seconds */
              );
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    /* An exception ocurred, do nothing */
}

if (val == WAIT_TIMEOUT || val == WAIT_FAILED )
    printf("ERROR: Timeout waiting for interrupt\n");
else
    printf("Interrupt received!\n");
```

Cross Reference:

Referenced Item	Page
PLX_INTR	5-58

PlxIntrDisable

Syntax:

```
RETURN_CODE
PlxIntrDisable(
    IN HANDLE    drvHandle,
    IN PLX_INTR *plxIntr
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Disables specific interrupts of a PCI device containing a PLX chip.

- *drvHandle* is the handle of the PCI device; and,
- *plxIntr* is a pointer to the interrupt structure that describes which interrupts will be disabled.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *plxIntr* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
HANDLE      plxHandle;
PLX_INTR    plxIntrStatus;
RETURN_CODE rc;


/* zero out the structure */
memset(&plxIntrStatus, 0, sizeof(plxIntrStatus));

plxIntrStatus.InboundPost    = 1;
plxIntrStatus.OutboundPost   = 1;
plxIntrStatus.IopDmaChannel0 = 1;
plxIntrStatus.PciDmaChannel0 = 1;
plxIntrStatus.IopDmaChannel1 = 1;
plxIntrStatus.PciDmaChannel1 = 1;

rc = PlxIntrDisable(
         plxHandle,
```

```
          &plxIntrStatus
          );
if (rc != ApiSuccess)
    printf("ERROR: Unable to disable interrupt, code = %d\n", rc);
```

Reference:

Referenced Item	Page
PLX_INTR	5-58

PlxIntrEnable

Syntax:

```
RETURN_CODE
PlxIntrEnable(
    IN HANDLE    drvHandle,
    IN PLX_INTR *plxIntr
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Enables specific interrupts of a PCI device containing a PLX chip.

- *drvHandle* is the handle of the PCI device; and,
- *plxIntr* is a pointer to the interrupt structure that describes which interrupts will be enabled.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *plxIntr* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE plxHandle;
PLX_INTR plxIntrStatus;

memset(&plxIntrStatus, 0, sizeof(PLX_INTR));
plxIntrStatus.InboundPost = 1;
plxIntrStatus.OutboundPost = 1;
plxIntrStatus.PciDmaChannel0 = 1;
plxIntrStatus.PciDmaChannel1 = 1;

rc = PlxIntrEnable(plxHandle, &plxIntrStatus);
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to enable interrupts, code = %d\n", rc);
}
```

Reference:

Referenced Item	Page
PLX_INTR	5-58

PlxIntrStatusGet

Syntax:

```
RETURN_CODE
PlxIntrStatusGet(
    IN HANDLE     drvHandle,
    OUT PLX_INTR *plxIntr
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Returns the interrupts of the PCI device containing a PLX chip that were last active. The interrupts are cleared once they are read.

- *drvHandle* is the handle of the PCI device; and,
- *plxIntr* is a pointer to the interrupt structure that contains information detailing which interrupts are active.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *plxIntr* parameter is NULL.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE plxHandle;
PLX_INTR plxIntrStatus;

/* zero out the structure */
memset(&plxIntrStatus, 0, sizeof(PLX_INTR));

rc = PlxIntrStatusGet(
         plxHandle,
         &plxIntrStatus
         );
if (rc != ApiSuccess)
```

```
{
  printf("\a\nPlxIntrStatusGet failed with error code %08lx.", rc);
}
else
{
  printf("\nInterrupted with following interrupt status.");
  printf("\nOutboundOverflow %s",
         (plxIntrStatus.OutboundOverflow)?"Yes":"NO");
  printf("\nIopDmaChannel0   %s",
         (plxIntrStatus.IopDmaChannel0)?"Yes":"NO");
  printf("\nPciDmaChannel0   %s",
         (plxIntrStatus.PciDmaChannel0)?"Yes":"NO");
  printf("\nIopDmaChannel1   %s",
         (plxIntrStatus.IopDmaChannel1)?"Yes":"NO");
  printf("\nPciDmaChannel1   %s",
         (plxIntrStatus.PciDmaChannel1)?"Yes":"NO");
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	5-58

PlxIoPortRead

Syntax:

```
RETURN_CODE
PlxIoPortRead(
    IN HANDLE      drvHandle,
    IN U32         address,
    IN ACCESS_TYPE bits,
    OUT PVOID      pOutData
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads a value from an I/O port.

- *drvHandle* is the handle of the PCI device;
- *address* is the I/O port address to read from;
- *accessType* defines the access type size; and,
- *pOutData* is the data read from the I/O port.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pOutData* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidAccessType	The *accessType* size is not supported for this device.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 value;
rc = PlxIoPortRead(drvHandle, 0x6F00, BitSize32, &value);
if (rc != ApiSuccess)
{
  printf("Error: Unable to read data.\n");
  return -1;
}
rc = PlxIoPortWrite(drvHandle, 0x6F04, BitSize32, &value);
if (rc != ApiSuccess)
```

```
{
  printf("Error: Unable to write data.\n");
  return -1;
}
```

Cross Reference:

Referenced Item	Page
ACCESS_TYPE	5-9

PlxIoPortWrite

Syntax:

```
RETURN_CODE
PlxIoPortWrite(
    IN HANDLE      drvHandle,
    IN U32         address,
    IN ACCESS_TYPE bits,
    IN PVOID       pValue
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Writes a value to an I/O port.

- *drvHandle* is the handle of the PCI device;
- *address* is the I/O port address to write to;
- *accessType* defines the access type size; and,
- *pOutData* is the data to write to the I/O port.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidAccessType	The *accessType* size is not supported for this device.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 value;

rc = PlxIoPortRead(drvHandle, 0x6F00, BitSize32, &value);
if (rc != ApiSuccess)
{
```

```
  printf("Error: Unable to read data.\n");
  return -1;
}
rc = PlxIoPortWrite(drvHandle, 0x6F04, BitSize32, &value);
if (rc != ApiSuccess)
{
  printf("Error: Unable to write data.\n");
  return -1;
}
```

Cross Reference:

Referenced Item	Page
ACCESS_TYPE	5-9

PlxMuHostOutboundIndexRead

Syntax:

```
U32
PlxMuHostOutboundIndexRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

IOP 480

Description:

Reads the Host Outbound Index register to determine the number of IOP frames that were processed (Outbound Option).

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* is a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidHandle	The function was passed a handle which was not previously opened.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 indexValue;

/* Read the Outbound Index value */
indexValue = PlxMuHostOutboundIndexRead(drvHandle, &rc);
if (rc != ApiSuccess)
  /* error handler code goes here */;

return (indexValue);
```

PlxMuHostOutboundIndexWrite

Syntax:

```
RETURN_CODE
PlxMuHostOutboundIndexWrite(
    IN HANDLE drvHandle,
    IN PU32   pRegisterData
    );
```

PLX Chip Support:

IOP 480

Description:

Writes to the Host Outbound Index register to signal to the IOP the number of frames that were processed (Outbound Option).

- *drvHandle* is the handle of the PCI device; and
- *pRegisterData* is a pointer to the index value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 indexValue;

/* Update the Outbound Index Value, */
/* This is an absolute index each time I call it! */
rc = PlxMuHostOutboundIndexWrite(drvHandle, &indexValue);
if (rc != ApiSuccess)
  /* error handler code goes here */;
```

PlxMuInboundPortRead

Syntax:

```
RETURN_CODE
PlxMuInboundPortRead(
    IN HANDLE drvHandle,
    OUT PU32  framePointer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Reads the Inbound Port and gets a Free Message Frame.

- *drvHandle* is the handle of the PCI device; and
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

/* Read inbound port */
rc = PlxMuInboundPortRead(drvHandle, &framePointer);
if (rc != ApiSuccess)
  /* error handler goes here */
```

PlxMuInboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuInboundPortWrite(
    IN HANDLE drvHandle,
    IN PU32   framePointer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Writes to the Inbound Port with a posted message frame.

- *drvHandle* is the handle of the PCI device;
- *framePointer* is the address of the Message Frame (MFA) to write.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 framePointer;

/* Initialize message frame */
InitializeMessageFrame(framePointer);

/* Write to the inbound port */
rc = PlxMuInboundPortWrite(drvHandle, &framePointer);
if (rc != ApiSuccess)
  /* error handler goes here */
```

PlxMuOutboundPortRead

Syntax:

```
RETURN_CODE
PlxMuOutboundPortRead(
    IN HANDLE drvHandle,
    OUT PU32  framePointer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Reads the Outbound Port and gets a posted message frame.

- *drvHandle* is the handle of the PCI device; and
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

/* Read outbound port */
rc = PlxMuOutboundPortRead(drvHandle, &framePointer);
if (rc != ApiSuccess)
  /* error handler */
```

PlxMuOutboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuOutboundPortWrite(
    IN HANDLE drvHandle,
    IN PU32   framePointer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Writes to the Outbound Port with a free message frame.

- *drvHandle* is the handle of the PCI device; and
- *framePointer* is the address of the Message Frame (MFA) to write.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *pValue* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The Messaging Unit must be properly initialized (typically by the IOP CPU).

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 framePointer;

/* Write to the outbound port */
rc = PlxMuOutboundPortWrite(drvHandle, &framePointer);
if (rc != ApiSuccess)
  /* error handler */
```

PlxPciAbortAddrRead

Syntax:

```
U32
PlxPciAbortAddrRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

IOP 480

Description:

Returns the starting address to the operation that caused the Abort.

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* or *framePointer* parameters are NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 abortAddr;

/* Read the Pci Abort Address */
abortAddr = PlxPciAbortAddrRead(drvHandle, &rc);

if (rc != ApiSuccess)
  /* error handler code goes here */;

return (abortAddr);
```

PlxPciBarRangeGet

Syntax:

```
RETURN_CODE
PlxPciBarRangeGet(
    IN HANDLE drvHandle,
    IN U32    barRegisterNumber,
    OUT PU32  data
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Retrieves the range of any PCI base address register.

- *drvHandle* is the handle of the PLX device;
- *barRegisterNumber* is the base address register number; and,
- *data* is a pointer to a buffer that stores the range.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or *data* parameter is NULL.
ApiInvalidRegister	The *registerNumber* parameter is out of range or does not contain a range.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
#define GETBARRANGE_MAX             6

RETURN_CODE rc;
HANDLE plxHandle;
U32 BarRanges[GETBARRANGE_MAX];

for (i = 0; i < GETBARRANGE_MAX; i ++)
{
  rc = PlxPciBarRangeGet(
           plxHandle,
           i,
           &BarRanges[i]
           );
  if (rc != ApiSuccess)
  {
```

```
     printf("\a\nFailed to get bar range for %08lx", i);
  }
  else
    printf("\nBar range for %d is %08lX", i, BarRanges[i]);
}
```

PlxPciBaseAddressesGet

Syntax:

```
RETURN_CODE
PlxPciBaseAddressesGet(
    IN HANDLE              drvHandle,
    OUT PVIRTUAL_ADDRESSES virtAddr
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Gets the user virtual addresses for the PCI base address register values of the PLX device.

- *drvHandle* is the handle of the PLX device; and,
- *virtAddr* is a pointer to the virtual address information.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or *virtAddr* parameter is NULL.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

The virtual address structure contains all the user virtual addresses for the various PCI base addresses available to access the PCI device.

Usage:

```
RETURN_CODE rc;
HANDLE plxHandle;
VIRTUAL_ADDRESSES virtualAddresses;

rc = PlxPciBaseAddressesGet(
         plxHandle,
         &virtualAddresses
         );
if (rc != ApiSuccess)
{
  printf("ERROR: Unable to get virtual address info, code = %d\n", rc);
  exit(0);
}
```

```
else
{
   printf("\n    U32 Va1    is %08lX", virtualAddresses.Va1);
   printf("\n    U32 Va2    is %08lX", virtualAddresses.Va2);
   printf("\n    U32 Va3    is %08lX", virtualAddresses.Va3);
   printf("\n    U32 Va4    is %08lX", virtualAddresses.Va4);
   printf("\n    U32 Va5    is %08lX", virtualAddresses.Va5);
}
```

Cross Reference:

Referenced Item	Page
PVIRTUAL_ADDRESSES	5-74

PlxPciBusSearch

Syntax:

```
RETURN_CODE
PlxPciBusSearch(
    IN OUT PDEVICE_LOCATION pDevData
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Searches for a PLX device on the PCI bus. When the function returns, *device* will contain the information for the specified device. *PlxPciBusSearch()* scans a list of supported devices for the first device matching the search data. If the device is found, the search data is completed. Otherwise `ApiInvalidDeviceInfo` is returned.

- *pDevData* contains the search data for the desired device.

Return Value:

On success, *pDevData* contains valid device information.

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *pDevData* parameter is NULL.
ApiInvalidDeviceInfo	The information in *pDevData* is not valid for any device in the system.
ApiNoActiveDriver	There is no device driver installed into the system.
ApiInsufficientResources	There is no memory available for the PCI API.

Notes:

This function is provided only for existing applications. There is no guarantee that this function will exist in future SDK versions. Please use *PlxPciDeviceFind()* with the *requestLimit* parameter set to 0 instead.

Usage:

```
RETURN_CODE rc;
DEVICE_LOCATION device;      /* Device Location Information */

device.VendorId        = PLX_VENDOR_ID;
device.DeviceId        = PLX_9080RDK_860_DEVICE_ID;
device.BusNumber       = 0xFFFFFFFF;
device.SlotNumber      = 0xFFFFFFFF;
device.SerialNumber[0] = '\0';

rc = PlxPciBusSearch(
```

```
        &device
        );

if (rc != ApiSuccess)
  printf("\a\nPlxPciBusSearch failed.");
else
{
    printf("\nU32 DeviceId   is %08lX",    device.DeviceId);
    printf("\nU32 VendorId   is %08lX",    device.VendorId);
    printf("\nU32 BusNumber  is %08lX",    device.BusNumber);
    printf("\nU32 SlotNumber is %08lX",    device.SlotNumber);
    printf("\nU8 SerialNumber [16] is %s", device.SerialNumber);
}
```

Cross Reference:

Referenced Item	Page
PDEVICE_LOCATION	5-15

PlxPciCommonBufferGet

Syntax:

```
RETURN_CODE
PlxPciCommonBufferGet(
    IN HANDLE       drvHandle,
    OUT PPCI_MEMORY pMemoryInfo
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Provides the memory information on the physical memory buffer that can be shared between the application, device driver or the PCI device.

- *drvHandle* is the handle of the PLX device;
- *pMemoryInfo* is a structure containing the information for the physical memory buffer.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or *pMemoryInfo* parameter is NULL.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

PLX device drivers allocate a physical buffer in PCI space for use by Win32 and IOP applications. Typically, it is utilized for DMA transfers. *PlxPciCommonBufferGet()* provides the physical address, virtual address, and size of this physical memory buffer.

The PLX chip's DMA engine requires physical addresses when transferring to/from PCI. Conversely, Windows applications may only use virtual addresses when accessing the PCI buffer.

To change the default size of the buffer, please refer to the *CommonBufferSize* entry in the registry section. Note that Windows OS does not guarantee allocation of large size buffers.

Usage:

```
HANDLE plxHandle;
PCI_MEMORY PciMemory;

rc = PlxPciCommonBufferGet(plxHandle, &PciMemory);
if (rc != ApiSuccess)
{
    printf("\a\nPlxPciCommonBufferGet tested: FAILED.");
```

```
}
else
{
    printf("\nCommon Buffer User address is %08lX",
           PciMemory.UserAddr);
    printf("\nCommon Buffer PCI physical address is %08lX",
           PciMemory.PhysicalAddr);
    printf("\nCommon Buffer Size is %08lX", PciMemory.Size);
}
```

Cross Reference:

Referenced Item	Page
PPCI_MEMORY	5-55

PlxPciConfigRegisterRead

Syntax:

```
U32
PlxPciConfigRegisterRead(
    IN U32            bus,
    IN U32            slot,
    IN U32            registerNumber,
    OUT PPRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads a configuration register from a PCI device.

- *bus* is the PCI bus number of the device to read;
- *slot* is the PCI slot number of the device to read;
- *registerNumber* is the configuration register to read; and,
- *pReturnCode* is a pointer to the return code of the function.

Return Value:

This function returns the value read from the register. The status of the function call is returned via the *pReturnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidRegister	The *registerNumber* parameter is out of range or not on a 4-byte boundary.
ApiConfigAccessFailed	The device described by the *bus* and *slot* parameter is not present.

Notes:

This function can access any supported device present on the PCI bus.

Usage:

```
RETURN_CODE rc;
U32 range, address;

/* Save a copy of the address */
address =  PlxPciConfigRegisterRead(0x0,
                                    0x12,
                                    PCI9080_LOCAL_BASE0,
                                    &rc);
```

```
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  exit(2);
}

/* Get the BAR range */
range = 0xFFFFFFFF;
rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &range);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  exit(2);
}

range =  PlxPciConfigRegisterRead(0x0,
                                  0x12,
                                  PCI9080_LOCAL_BASE0,
                                  &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  exit(2);
}

rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &address);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  exit(2);
}
```

PlxPciConfigRegisterReadAll

Syntax:

```
RETURN_CODE
PlxPciConfigRegisterReadAll(
    IN U32   bus,
    IN U32   slot,
    OUT PU32 buffer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads all PCI Configuration registers on a PLX PCI device.

- *bus* is the PCI bus number of the device to read;
- *slot* is the PCI slot number of the device to read; and,
- *buffer* is the storage location for the configuration register values.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *buffer* parameter is NULL.
ApiConfigAccessFailed	The device described by the *bus* and *slot* parameter is not present.
ApiInsufficientResources	There is no memory available for the PCI API.

Notes:

buffer MUST be already allocated and must hold enough room for all PCI Configuration registers. See the table below for the required sizes. This function can access any supported device present on the PCI bus.

PLX Chip	Minimum Buffer size in bytes
PCI 9080	64
PCI 9054	84
PCI 9030	84
IOP 480	96

Usage:

```
RETURN_CODE rc;
U32 pciRegsBuffer[0x20];

rc = PlxPciConfigRegisterReadAll(
        0x0,
        0x12,
        pciRegsBuffer
```

```
          );

if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterReadAll failed.");
  /* error handler code goes here */
  exit(1);
}
```

PlxPciConfigRegisterWrite

Syntax:

```
RETURN_CODE
PlxPciConfigRegisterWrite(
    IN U32  bus,
    IN U32  slot,
    IN U32  registerNumber,
    IN PU32 data
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Writes data to a configuration register on a PCI device.

- *bus* is the PCI bus number of the device to write;
- *slot* is the PCI slot number of the device to write;
- *registerNumber* is the configuration register to write to; and,
- *data* is a pointer to the buffer that contains the data to write.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidRegister	The *registerNumber* parameter is out of range or not on a 4-byte boundary.
ApiNullParam	The *data* parameter is NULL.
ApiConfigAccessFailed	The device described by the *bus* and *slot* parameter is not present.

Notes:

This function can access any supported device present on the PCI bus.

Usage:

```
RETURN_CODE rc;
U32 range, address;

/* Save a copy of the address */
address =  PlxPciConfigRegisterRead(0x0,
                                    0x12,
                                    PCI9080_LOCAL_BASE0,
                                    &rc);
```

```
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  exit(2);
}

/* Get the BAR range */
range = 0xFFFFFFFF;
rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &range);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  exit(2);
}

range =  PlxPciConfigRegisterRead(0x0,
                                  0x12,
                                  PCI9080_LOCAL_BASE0,
                                  &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  exit(2);
}

rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &address);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  exit(2);
}
```

PlxPciDeviceClose

Syntax:

```
RETURN_CODE
PlxPciDeviceClose(
    IN HANDLE drvHandle
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Closes a PLX device channel.

- *drvHandle* is the handle of the device driver.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidDeviceInfo	The device was not closed properly.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

This function should be used to close a PLX device handle before the application terminates.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
DEVICE_LOCATION device;

/* Open any supported device */
device.SerialNumber[0] = 0;
device.BusNumber       = MINUS_ONE_LONG;
device.SlotNumber      = MINUS_ONE_LONG;
device.DeviceId        = MINUS_ONE_LONG;
device.VendorId       = MINUS_ONE_LONG;
rc = PlxPciDeviceOpen(
         &device,
         &myPlxHandle
         );
if (rc != ApiSuccess)
{
    printf("Could not find any supported device.\n");
    printf ("Press any key to exit.....");
```

```
    getch();
    return;
}

/* Close the handle to the PLX device */
rc = PlxPciDeviceClose(myPlxDevice);

if (rc != ApiSuccess)
{
  printf ("\n\nErrors in closing device.\n");
  printf ("Returned code is %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxPciDeviceFind

Syntax:

```
RETURN_CODE
PlxPciDeviceFind(
    IN OUT PDEVICE_LOCATION device,
    IN OUT PU32             requestLimit
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Finds PLX devices on the PCI bus given a combination of bus number, slot number; vendor ID, and/or device ID, or by giving the `SerialNumber`. This function does one of two things, depending of the value of *requestLimit*:

> If *requestLimit* contains the value FIND_AMOUNT_MATCHED then the function looks for all devices that matches the search criteria (given in *device*) and returns the total number of matches in *requestLimit*.
>
> If *requestLimit* does not contain FIND_AMOUNT_MATCHED, *PlxPciDeviceFind()* assumes *requestLimit* contains the desired device number (numbering starts at zero). Then when the function returns *device* will contain completed information for the specified device, given the search criteria.

- *device* is a pointer to the device information to search for; and,
- *requestLimit* is a pointer to a U32 value.

Members of *device* may be selected/unselected for the search criteria. To select a member, just give it an appropriate value. To unselect it, just place MINUS_ONE_LONG in it (equivalent of (U32(-1))). See the Usage section below for a good example.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *device* parameter is NULL.
ApiInvalidDeviceInfo	The information in *device* is not valid for any device in the system.
ApiNoActiveDriver	There is no device driver installed into the system.
ApiInsufficientResources	There is no memory available for the PCI API.

Notes:

If the *`SerialNumber`* element within the DEVICE_LOCATION structure is not being used as a search criterion the first character should be set to '\0' to denote an empty string.

Usage:

```
RETURN_CODE rc;
DEVICE_LOCATION device;      /* Device Location Information */
U32 reqLimit;

/* Set API to find out the number of devices */
reqLimit = FIND_AMOUNT_MATCHED;

/* Set to find all PCI9054 RDK boards */
device.BusNumber = MINUS_ONE_LONG;
device.SlotNumber = MINUS_ONE_LONG;
device.VendorId = 0x10b5;
device.DeviceId = 0x9054;
device.SerialNumber[0] = '\0';

rc = PlxPciDeviceFind(
         &device,
         &reqLimit
         );
if ((rc != ApiSuccess) || (reqLimit < 1))
{
  printf("\a\nCould not find the 0x%04x:0x%04x board in system. ",
         device.VendorID, device.DeviceId);
  exit(2);
}
```

Cross Reference:

Referenced Item	Page
PDEVICE_LOCATION	5-15

PlxPciDeviceOpen

Syntax:

```
RETURN_CODE
PlxPciDeviceOpen(
    IN PDEVICE_LOCATION pDevice,
    OUT PHANDLE         pDrvHandle
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Gets a handle to a PLX device on a PCI bus given a combination of bus number, slot number; vendor ID, and/or device ID, or by giving a `SerialNumber`. When the function returns, *device* will contain completed information for the specified device. This function does one of two things, depending on the value of the `SerialNumber` member of *pDevice* parameter.

If no serial number if specified (pDevice->SerialNumber[0] is '\0'), the *PlxPciDeviceOpen()* calls *PlxPciDeviceFind()* and tries to get a handle to the first device matching the combination of bus number, slot number, vendor ID and/or device ID.

If a serial number is specified, the other members of the *pDevice* structure are ignored and *PlxPciDeviceOpen()* will try to get a handle to the PLX device described by the unique serial number. Then, the other members of the structure are filled with the appropriate values and the structure is returned.

- *pDevice* is the structure that contains device information necessary to open a unique device; and,
- *pDrvHandle* is the handle of the device driver that the PCI API is using.

The structure *pDevice* should contain information returned by *PlxPciDeviceFind()*. However, a user may fill the structure with his own values and pass it to *PlxPciDeviceOpen()*. Then, if no serial number is specified, *PlxPciDeviceOpen()* will call *PlxPciDeviceFind()* for a complete description of the first device matching the search criteria and will return a handle to that device.

A user may decide to fill the *pDevice* structure. In this case, some members of the structure may be selected for the search and others ignored. Those to be ignored must be filled with MINUS_ONE_LONG (-1). The members that are to participate in the search have to be filled with an appropriate value. If the `SerialNumber` member is to be ignored, its first character has to be set to '\0'.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *pDevice* or *pDrvHandle* parameter is NULL.
ApiInvalidDeviceInfo	The information in *pDevice* is not valid for any device in the system.
ApiNoActiveDriver	There is no device driver installed into the system.
ApiInsufficientResources	There is no memory available for the PCI API.

Notes:

This function should be used to open a channel to a PLX device and get a handle to the device driver before any PCI API functions that require a handle can be used.

The handle returned by this API function should be closed using the *PlxPciDeviceClose()* function before the user application exits.

Usage:

```
RETURN_CODE rc;
DEVICE_LOCATION device;
HANDLE myPlxDevice;

/* Open a handle to the first PLX device found */
device.BusNumber  = MINUS_ONE_LONG;
device.SlotNumber = MINUS_ONE_LONG;
device.DeviceId   = MINUS_ONE_LONG;
device.VendorId   = MINUS_ONE_LONG;
strcpy (device.SerialNumber, "");
rc = PlxPciDeviceOpen(
         &device,
         &myPlxDevice
         );
if (rc != ApiSuccess)
{
  printf ("\n\nErrors in opening device.\n");
  printf ("Returned code is %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

Cross Reference:

Referenced Item	Page
DEVICE_LOCATION	5-15

PlxPmIdRead

Syntax:

```
U8
PlxPmIdRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Returns the Power Management new capabilities ID.

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
HANDLE drvHandle;
RETURN_CODE rc;
U8 powerManId;

/* this function assumes we have a valid drvHandle */

powerManId = PlxPmIdRead(drvHandle, &rc);

if (rc != ApiSuccess)
{
    printf("PlxPmIdRead() failed, returned rc= %d\n",rc);
    return (-1);
}

return (powerManId);
```

PlxPmNcpRead

Syntax:

```
U8
PlxPmNcpRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Reads the location of the next element in the linked list of capabilities. If this is the last item in the list, the value will return 0.

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* a pointer to the return code;

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
HANDLE drvHandle;
RETURN_CODE rc;
U8 nextElement;

/* this function assumes we have a valid drvHandle */

nextElement = PlxPmNcpRead(drvHandle, &rc);

if (rc != ApiSuccess)
{
    printf("PlxPmNcpRead() failed, returned rc= %d\n",rc);
    return (-1);
}

return (nextElement);
```

PlxPowerLevelGet

Syntax:

```
PLX_POWER_LEVEL
PlxPowerLevelGet(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE returnCode
    );
```

PLX Chip Support:

PCI 9080*, PCI 9054, IOP 480, PCI 9030

*The PCI 9080 does not support power management. Therefore the only possible valid return value is: D0.

Description:

Gets the current power level of a PCI device.

- *drvHandle* is the handle of the PCI device; and,
- *returnCode* is the return code of the function.

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen().*

Return Value:

This function returns the current power level of the PLX chip. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle which was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
PLX_POWER_LEVEL powerLevel;
powerLevel = PlxPowerLevelGet(myPlxDevice, &rc);
if (rc != ApiSuccess)
{
  printf("\n Error Getting Power Level. RC = %x", rc);
  getch();
}
else
{
  if (powerLevel == D0)
```

```
    printf("\n OK: Power Level received as D0");
  else
  {
    printf("\n ERROR: Power Level received as %d", powerLevel);
    getch();
  }
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	5-65

PlxPowerLevelSet

Syntax:

```
RETURN_CODE
PlxPowerLevelSet(
    IN HANDLE          drvHandle,
    IN PLX_POWER_LEVEL plxPowerLevel
    );
```

PLX Chip Support:

PCI 9080*, PCI 9054, IOP 480, PCI 9030

*The PCI 9080 does not support power management. Therefore, the only valid power level for the PCI 9080 is: D0.

Description:

Sets the power level of a PCI device.

- *drvHandle* is the handle of the PCI device; and,
- *plxPowerLevel* is the new power level.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidPowerState	The *plxPowerLevel* parameter is invalid for this PLX chip.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

rc = PlxPowerLevelSet(drvHandle, D0);
if (rc != ApiSuccess)
{
  printf("\n Error Setting a valid Power Level (D0). RC = %x", rc);
  getch();
}
else
  printf("\n OK: Valid Power Level Set");
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	5-65

PlxRegisterDoorbellRead

Syntax:

```
U32
PlxRegisterDoorbellRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE returnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Clears and reads value to the IOP-to-PCI doorbell register on a PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

This function returns the value read from the IOP-to-PCI doorbell register. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 value;

value = PlxRegisterDoorbellRead(myPlxDevice, &rc);
if (rc != ApiSuccess)
  printf("\n ERROR: Reading Doorbell did not work");
else
  printf("\n OK: Doorbell Read succeeded. Value is %x", value);
```

PlxRegisterDoorbellSet

Syntax:

```
RETURN_CODE
PlxRegisterDoorbellSet(
    IN HANDLE drvHandle,
    IN U32    data
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Writes a value to the PCI-to-IOP doorbell register on a PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *data* is a U32 value to store in the register.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 value;

rc = PlxRegisterDoorbellSet(myPlxDevice, 0x1234567);
if (rc != ApiSuccess)
{
  printf("\n ERROR: Setting Doorbell did not work");
  getch();
}
else
{
```

```
  if (PlxRegisterRead(myPlxDevice, 0x60, &rc) == 0x1234567)
    printf("\n OK: DoorbellSet succeeded");
  else
  {
    printf("\n ERROR: After setting doorbell and reading the "
           "register the return data was wrong!");
    printf("\n ERROR: This is normal since the BSP will clear "
           "the doorbell immediately!");
    getch();
  }
}
```

PlxRegisterMailboxRead

Syntax:

```
U32
PlxRegisterMailboxRead(
    IN HANDLE        drvHandle,
    IN MAILBOX_ID    mailboxId,
    OUT PRETURN_CODE returnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Reads a mailbox register on the currently selected PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *mailboxId* is the mailbox register ID; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

This function returns the value read from the mailbox register. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidRegister	The *mailboxId* parameter is not a valid mailbox ID.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 value;

value = PlxRegisterMailboxRead(myPlxDevice, MailBox0, &rc);
if (rc != ApiSuccess)
{
  printf("\n Error Reading a valid Mailbox");
```

```
  getch();
}
else
  printf("\n OK: Valid Mailbox Read succeeded with a value of %x",
         value);
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	5-50

PlxRegisterMailboxWrite

Syntax:

```
RETURN_CODE
PlxRegisterMailboxWrite(
    IN HANDLE     drvHandle,
    IN MAILBOX_ID mailboxId,
    IN U32        data
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Writes a value to a mailbox register on a PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *mailboxId* is the mailbox register ID; and,
- *data* is a U32 value to store in the register.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidRegister	The *mailboxId* parameter is not a valid mailbox ID.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;

rc = PlxRegisterMailboxWrite(myPlxDevice, MailBox0, 0x3DC87A00);
if (rc != ApiSuccess)
{
  printf("\n Error Writing a valid Mailbox");
  getch();
}
else
```

```
printf("\n OK: Valid Mailbox Write succeeded");
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	5-50

PlxRegisterRead

Syntax:

```
U32
PlxRegisterRead(
    IN HANDLE        drvHandle,
    IN U32           registerOffset,
    OUT PRETURN_CODE returnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads a register on the currently selected PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *registerOffset* is the register number offset; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

This function returns the value read from the register. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidRegister	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 value;

value = PlxRegisterRead(myPlxDevice, 0x18, &rc);
if (rc != ApiSuccess)
{
```

```
    printf("\n Error Reading a valid Register");
    getch();
  }
  else
    printf("\n OK: Valid register Read succeeded with a value of %x",
           value);
```

PlxRegisterReadAll

Syntax:

```
RETURN_CODE
PlxRegisterReadAll(
    IN HANDLE drvHandle,
    IN U32    startOffset,
    IN U32    registerCount,
    OUT PU32  buffer
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads multiple registers on a PLX PCI device.

- *drvHandle* is the handle of the PCI device;
- *startOffset* is the register offset to start reading at;
- *registerCount* is the number of bytes to read starting at *startOffset*; and,
- *buffer* is the storage location for the register values.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or the *buffer* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInsufficientResources	There is no memory available for the PCI API.
ApiInvalidRegister	The *startOffset* and the *registerCount* parameters combined exceed the valid range of registers or *startOffset* is not on a 4-byte boundary.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

This function reads only local configuration registers, not PCI configuration registers. Use *PlxPciConfigRegisterReadAll()* for PCI registers.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 buffer[0x40];
```

```
rc = PlxRegisterReadAll(
         myPlxDevice,
         0x0,
         0x100,
         buffer
         );
if (rc != ApiSuccess)
{
  printf("\n ERROR: Reading All registers failed API call");
}
```

PlxRegisterWrite

Syntax:

```
RETURN_CODE
PlxRegisterWrite(
    IN HANDLE drvHandle,
    IN U32 registerOffset,
    IN U32 data
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Writes a value to a register on a PLX PCI device.

- *drvHandle* is the handle of the PCI device
- *registerNumber* is the register number
- *data* is a U32 value to store in the register

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidRegister	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;

rc = PlxRegisterWrite(myPlxDevice, 0x18, 0x234980C4);
if (rc != ApiSuccess)
{
  printf("\n Error Writing a valid Register");
  getch();
```

```
        }
        else
          printf("\n OK: Valid register Write succeeded");
```

PlxSdkVersion

Syntax:

```
RETURN_CODE
PlxSdkVersion(
    U8 *pMajor,
    U8 *pMinor,
    U8 *pRevision
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Returns the version numbers in decimal of the PCI SDK PCI API.

- *pMajor* is a pointer to the U8 containing the Major version number;
- *pMinor* is a pointer to the U8 containing the Minor version number; and,
- *pRevision* is a pointer to the U8 containing the Revision version number.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	One or more of the parameters were passed in as NULL.

Usage:

```
RETURN_CODE rc;
U8 SdkMajor, SdkMinor, SdkRevision;

rc = PlxSdkVersion(
         &SdkMajor,
         &SdkMinor,
         &SdkRevision
         );
if (rc != ApiSuccess)
{
  printf("\n ERROR: Getting SDK Version Number. RC = %x", rc);
}
else
{
  printf("\nSDK Major is %d, Minor is %d, Revision Number is %d",
         SdkMajor, SdkMinor, SdkRevision);
}
```

PlxSerialEepromPresent

Syntax:

```
BOOLEAN
PlxSerialEepromPresent(
    IN HANDLE       drvHandle,
    IN PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Determines whether a Serial EEPROM device is found on the PLX PCI device selected.

- *drvHandle* is the handle of the PCI device
- *pReturnCode* a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function completed successfully.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

For some PLX chips, this function may return FALSE if an EEPROM is present, but it's blank.

Usage:

```
HANDLE      drvHandle;
RETURN_CODE rc;
BOOLEAN     isPresent;

isPresent = PlxSerialEepromPresent(drvHandle, &rc)
if (rc != ApiSuccess)
    printf("PlxSerialEepromPresent() failed\n");
else if (isPresent)
    printf ("Eeprom Detected!\n");
```

PlxSerialEepromRead

Syntax:

```
RETURN_CODE
PlxSerialEepromRead(
    IN HANDLE      drvHandle,
    IN EEPROM_TYPE eepromType,
    OUT PU32       buffer,
    IN U32         size
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Reads values from the configuration EEPROM connected to the PLX PCI device selected.

- *drvHandle* is the handle of the PCI device;
- *eepromType* is the type of EEPROM on the PCI device;
- *buffer* is a pointer a buffer to store the data read; and,
- *size* defines the number of bytes you want to read from the EEPROM.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or *buffer* parameters are NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidSize	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.
ApiInsufficientResources	There is no memory available for the PCI API.
ApiEepromTypeNotSupported	The *eepromType* parameter is not supported for this PLX chip.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U16 eepromData[0x16];          /* Holding data read from EEPROM */

/* check if serial EEPROM is present or not */
```

```
if (PlxSerialEepromPresent(plxHandle, &rc))
{
  /* Reading EEPROM into eepromData buffer */
  rc = PlxSerialEepromRead(plxHandle,
                           Eeprom93CS56,
                           (U32 *)eepromData,
                           0x16 * sizeof(U16));  /* size in bytes */
  if (rc != ApiSuccess)
    /* error handler code goes here */
}
else
{
    /* serial EEEPROM is not present, error handler code goes here */
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	5-32

PlxSerialEepromWrite

Syntax:

```
RETURN_CODE
PlxSerialEepromWrite(
    IN HANDLE      drvHandle,
    IN EEPROM_TYPE eepromType,
    IN PU32        buffer,
    IN U32         size
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Writes values to the configuration EEPROM connected to the PLX PCI device selected.

- *drvHandle* is the handle of the PCI device;
- *eepromType* is the type of EEPROM on the PCI device;
- *buffer* is a pointer a buffer that contains the data; and,
- *size* defines the number of bytes you want to write to the EEPROM.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* or *buffer* parameters are NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidSize	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.
ApiInsufficientResources	There is no memory available for the PCI API.
ApiEepromTypeNotSupported	The *eepromType* parameter is not supported for this PLX chip.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U16 eepromData[0x16];          /* Contains valid data for EEPROM */

if ( PlxSerialEepromPresent(plxHandle, &rc))
```

```
{
    /* Write eepromData buffer to the serial EEPROM */
    rc = PlxSerialEepromWrite(plxHandle,
                              Eeprom93CS56,
                              (U32 *)eepromData,
                              0x16 * sizeof(U16));/* size in bytes */
    if (rc != ApiSuccess)
          /* error handler */
}
else
{
    /* serial EEPROM is not present, error handler code goes here */
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	5-32

PlxUserRead

Syntax:

```
PIN_STATE
PlxUserRead(
    IN HANDLE        drvHandle,
    IN USER_PIN      userPin,
    OUT PRETURN_CODE returnCode
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Reads the PCI device's USERI pins.

- *drvHandle* is the handle of the PCI device.
- *userPin* is the USERI pin number to be read; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

This function returns the state of the USER pin, being either `Active` or `Inactive`. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidUserPin	This *userPin* is not present on this PLX chip.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
PIN_STATE pinState;

pinState = PlxUserRead(drvHandle, USER0, &rc);
if (rc != ApiSuccess)
{
```

```
    printf("\n ERROR: Cannot Read USER In . RC = %x", rc);
    getch();
  }
  else
    printf("\n OK: USER0 pin was read as %x", pinState);
```

Cross Reference:

Referenced Item	Page
USER_PIN	5-73

PlxUserWrite

Syntax:

```
RETURN_CODE
PlxUserWrite(
    IN HANDLE    drvHandle,
    IN USER_PIN  userPin,
    IN PIN_STATE pinState
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480

Description:

Writes to the PCI device's USERO pins, setting them either `ACTIVE` or `INACTIVE`.

- *drvHandle* is the handle of the PCI device;
- *userPin* is the USERO pin number to be written; and,
- *pinState* is the new state to set the USERO pin.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiInvalidHandle	The function was passed a handle that was not previously opened.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiPowerDown	The PLX device is in a power state that is lower than required for this function.
ApiInvalidUserPin	This *userPin* is not present on this PLX chip.

Notes:

Before this function can be used, a PCI device must be selected using *PlxPciDeviceOpen()*.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

rc = PlxUserWrite(drvHandle, USER0, Active);
if (rc != ApiSuccess)
{
  printf("\n ERROR: Cannot Write to USER Out. RC = %x", rc);
  getch();
}
else
```

```
printf("\n OK: USER0 pin was written");
```

Cross Reference:

Referenced Item	Page
USER_PIN	5-73

PlxVpdIdRead

Syntax:

```
U8
PlxVpdIdRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Returns the Power Management new capabilities ID.

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
HANDLE drvHandle;
RETURN_CODE rc;
U8 vpdId;

/* this function assumes we have a valid drvHandle */

vpdId = PlxVpdIdRead(drvHandle, &rc);

if (rc != ApiSuccess)
{
    printf("PlxVpdIdRead() failed, returned rc= %d\n",rc);
    return (-1);
}

return (vpdId);
```

PlxVpdNcpRead

Syntax:

```
U8
PlxVpdNcpRead(
    IN HANDLE        drvHandle,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Reads the location of the next element in the linked list of capabilities. If this is the last item in the list, the value will return 0.

- *drvHandle* is the handle of the PCI device; and
- *pReturnCode* a pointer to the return code;

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
HANDLE drvHandle;
RETURN_CODE rc;
U8 nextElement;

/* this function assumes we have a valid drvHandle */
nextElement = PlxVpdNcpRead(drvHandle, &rc);

if (rc != ApiSuccess)
{
    printf("PlxVpdNcpRead() failed, returned rc= %d\n",rc);
    return (-1);
}

return (nextElement);
```

PlxVpdRead

Syntax:

```
U32
PlxVpdRead(
    IN HANDLE        drvHandle,
    IN U32           offset,
    OUT PRETURN_CODE pReturnCode
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Reads the Vital Product Data from the specified byte offset.

- *drvHandle* is the handle of the PCI device;
- *offset* is the byte offset to read or write, it must be U32 aligned; and,
- *pReturnCode* a pointer to the return value.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 vpdData;

/* Read the Vital Product Data region */
for (offset = 0; offset <= MAX_VPD_OFFSET; offset +=4)
{
    vpdData = PlxVpdRead(drvHandle, offset, &rc);
    if (rc != ApiSuccess)
        break;
    printf("Vpd offset %d = 0x%08x\n",offset, vpdData);
}
if (rc != ApiSuccess)
    /* error handler */;
```

PlxVpdWrite

Syntax:

```
RETURN_CODE
PlxVpdWrite(
    IN HANDLE drvHandle,
    IN U32    offset,
    IN U32    VpdData
    );
```

PLX Chip Support:

PCI 9054, IOP 480, PCI 9030

Description:

Writes a value to the VPD at the the specified offset.

- *drvHandle* is the handle of the PCI device;
- *offset* is the byte offset to read or write, it must be U32 aligned; and,
- *VpdData* is the 32-bit data to be written.

Return Value:

Return Value	Description
ApiSuccess	The function returned successfully.
ApiNullParam	The *drvHandle* parameter is NULL.
ApiInvalidHandle	The function was passes a handle that was improperly opened.

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
U32 vpdValue;
U32 offset = 0x0;

/* assume vpdValue contains the writeable data */
rc = PlxVpdWrite(drvHandle, offset, &vpdValue);
if (rc != ApiSuccess)
  /* error handler */
```

Syntax:

RETURN_CODE

```
PlxBusIopRead(
    IN HANDLE      drvHandle,
    IN IOP_SPACE   iopSpace,
    IN U32         address,
    IN BOOLEAN     remapAddress,
    OUT PU32       destination,
    IN U32         transferSize,
    IN ACCESS_TYPE accessType
    );
```

PLX Chip Support:

PCI 9080, PCI 9054, IOP 480, PCI 9030

Description:

Reads a range of values from the local bus of a PCI device containing a PLX chip (Direct Slave Read).

- *drvHandle* is the handle of the PCI device;
- *iopSpace* defines which Local Address Space register is used;
- *address* is the starting offset from the IOP space PCI remap address (remapAddress=TRUE) or the actual IOP bus address to start reading from (remapAddress=FALSE);
- *remapAddress* states how to treat the IOP address given. FALSE means that the IOP address is an offset that must be within the local space's addressable space (which is the inverse of its range value). If the *remp* value is set, any address within the 4 GB 32-bit address range is valid.;

4 IOP API

PLX also offers, in the SDK Pro, equivalent IOP-side (or local) APIs for use from the I/O platform (RDK or user board, containing a PLX chip and processor functionality, either on chip or as separate CPU). This "IOP API", designed around the features of the PLX chip, will be described in SDK Pro Programmer's Reference. As of the 3.1 release of the Host SDK, the IOP API from SDK 3.0 is current, and fully compatible with the 3.1 Host API and drivers.

5 PCI SDK Data Structures Used by API

5.1 Sample Data Structure

The following is an example of a data structure or data type definition.

SAMPLE structure

```
typedef struct _SAMPLE
{
  U32 someRegister;
  U32 someNumber;
  U32 someSize;
  U32 someBuffer[SOME_BUFFER_SIZE];
} SAMPLE, *PSAMPLE;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480	PCI 9030
SomeRegister	xxx, yy	xxx, yy	xxx, yy	xxx, yy

The registers that are affected by changing the values in the structure for each PLX device. All register offsets are from the IOP side unless otherwise stated. “xxx” is the register offset and “yy” is the bits in the register that are affected.

Purpose

The reasons for using this structure.

Members

An explanation of the members contained within the structure. Possible values are given when applicable.

Comments

Extra comments on how and when this structure is used.

5.2 Details of data structures

ADDRESS, SDATA and UDATA Data Types

```
#if defined(PLX_A64_D64)    /* Both 64-bit Data and Address buses */
    typedef U64         ADDRESS, *PADDRESS;
    typedef S64         SDATA, *PSDATA;
    typedef U64         UDATA, *PUDATA;
#elif defined(PLX_A64_D32)  /* 64-bit Address, 32-bit Data buses */
    typedef U64         ADDRESS, *PADDRESS;
    typedef S32         SDATA, *PSDATA;
    typedef U32         UDATA, *PUDATA;
#elif defined(PLX_A32_D64)  /* 32-bit Address, 64-bit Data buses */
    typedef U32         ADDRESS, *PADDRESS;
    typedef S64         SDATA, *PSDATA;
    typedef U64         UDATA, *PUDATA;
#else                       /* The defaults are both U32 buses */
    typedef U32         ADDRESS, *PADDRESS;
    typedef S32         SDATA, *PSDATA;
    typedef U32         UDATA, *PUDATA;
#endif
```

Affected Register Location

Not applicable.

Purpose

This data type is used for IOP API function compatibility among 32 bit Data Bus, 32 bit Address Bus; 32-bit Data Bus, 64-bit Address Bus; 64-bit Data Bus, 32-bit Address Bus and 64-bit Data Bus, 64-bit Address Bus.

Comments

Currently, the defaults are 32-bit data and address buses in the IOP API code.

BOOLEAN Types

```
#if defined(BOOLEAN)
    typedef BOOLEAN BOOL;
    typedef BOOLEAN *PBOOLEAN, *PBOOL;
#elif defined(BOOL)
    typedef BOOL BOOLEAN;
    typedef BOOL *PBOOLEAN, *PBOOL;
#else
    typedef S8       BOOLEAN, *PBOOLEAN;
    typedef BOOLEAN  BOOL;
#endif /* BOOLEAN or BOOL */
```

Affected Register Location

Not applicable.

Purpose

This data type defines the BOOLEAN or BOOL data type.

S8 and U8 Data Types

```
typedef char           S8, *PS8;
typedef unsigned char  U8, *PU8;
```

Affected Register Location

Not applicable.

Purpose

These data types are used for 8 bit values.

Comments

This data type allows compatibility of all the PCI API functions with all compilers.

S16 and U16 Data Types

```
typedef short           S16, *PS16;
typedef unsigned short  U16, *PU16;
```

Affected Register Location

Not applicable.

Purpose

These data types are used for 16 bit values.

Comments

This data type allows compatibility of the PCI API functions with all compilers.

S32 and U32 Data Types

```
typedef long           S32, *PS32;
typedef unsigned long  U32, *PU32;
```

Affected Register Location

Not applicable.

Purpose

These data types are used for 32 bit values.

Comments

This data type allows compatibility of the PCI API functions with all compilers.

U64 Data Type

```
#if defined(IOP_CODE)
    #if defined(longlong)
        #define BITS_64                 1
        typedef signed long long int   S64, *PS64;
        typedef unsigned long long int U64, *PU64;
    #else
        #define BITS_64                 0
        typedef struct _S64 {
           U32                          LowPart;
           S32                          HighPart;
        } S64;

        typedef struct _U64 {
           U32                          LowPart;
           U32                          HighPart;
        } U64;
    #endif
#endif

#if defined(PCI_CODE)
  #ifdef BITS_64
    typedef LARGE_INTEGER U64, *PU64;
  #else
    typedef U32 U64, *PU64;
  #endif /* BITS_64 */
#endif /* IOP_CODE or PCI_CODE */
```

Affected Register Location

Not applicable.

Purpose

This structure supports 64 bit systems and PLX chips with 64 bit registers

Comments

This data type allows compatibility with the PCI API functions with 32 bit and 64 bit systems and compilers.

Access Type Enumerated Data Type

```
typedef enum _ACCESS_TYPE
{
  BitSize8,
  BitSize16,
  BitSize32,
  BitSize64
} ACCESS_TYPE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for determining the access type size.

Members

BitSize8
Use 8 bits (char) for the access type size.

BitSize16
Use 16 bits (short) for the access type size.

BitSize32
Use 32 bits (long) for the access type size.

BitSize64
Use 64 bits (longlong) for the access type size.

Comments

The access type enumerated type is used to state the access type size for a data transfer.

API Parameters Structure

```
typedef struct _API_PARMS
{
  VOID *PlxIcIopBaseAddr;
  VOID *SpuMemBaseAddr;
  PPCI_BUS_PROP PtrPciBusProp;
  PPCI_ARBIT_DESC PtrPciArbitDesc;
  PIOP_BUS_PROP PtrIopBus0Prop;
  PIOP_BUS_PROP PtrIopBus1Prop;
  PIOP_BUS_PROP PtrIopBus2Prop;
  PIOP_BUS_PROP PtrIopBus3Prop;
  PIOP_BUS_PROP PtrLcs0Prop;
  PIOP_BUS_PROP PtrLcs1Prop;
  PIOP_BUS_PROP PtrLcs2Prop;
  PIOP_BUS_PROP PtrLcs3Prop;
  PIOP_BUS_PROP PtrDramProp;
  PIOP_BUS_PROP PtrDefaultProp;
  PIOP_BUS_PROP PtrExpRomBusProp;
  PIOP_ARBIT_DESC PtrIopArbitDesc;
  PIOP_ENDIAN_DESC PtrIopEndianDesc;
  PPM_PROP PtrPMProp;
  PU32 PtrVPDBaseAddress;
}API_PARMS, *PAPI_PARMS;
```

Affected Register Location

Not applicable.

Purpose

This data type is used to pass data from the BSP to the IOP API and to initialize certain data structures to the PLX chip's default values.

Basically, *PlxIcIopBaseAddr* and *SpuMemBaseAddr* are used to pass data to the IOP API. The rest data structures are to be initialized by the IOP API (PlxInitApi) to the PLX chip's default values.

Members

PlxIcIopBaseAddr

The base IOP address for the PLX chip, user should supply this value when call PlxInitApi() to pass data to the IOP API.

SpuMemBaseAddr

The memory mapped base address of Serial Port Unit, user should supply this value when call PlxInitApi() to pass data to the IOP API.

PtrPciBusProp

A pointer to the PCI_BUS_PROP structure that will be used to initialize the PCI Bus properties of the PLX chip.

PtrPciArbitDesc

A pointer to the PCI_ARBIT_DESC structure that will be used to initialize the PCI Bus arbiter of the PLX chip.

PtrIopBus0Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 0 register accesses to the IOP Bus.

PtrIopBus1Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 1 register accesses to the IOP Bus.

PtrIopBus2Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 2 register accesses to the IOP Bus.

PtrIopBus3Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 3 register accesses to the IOP Bus.

PtrLcs0Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Memory Region LCS0 register accesses to the IOP Bus.

PtrLcs1Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Memory Region LCS1 register accesses to the IOP Bus.

PtrLcs2Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Memory Region LCS2 register accesses to the IOP Bus.

PtrLcs3Prop

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Memory Region LCS3 register accesses to the IOP Bus.

PtrDramProp

A pointer to the IOP_BUS_PROP structure that will be used to initialize the DRAM register accesses to the IOP Bus.

PtrDefaultProp

A pointer to the IOP_BUS_PROP structure that will be used to initialize the Default register accesses to the IOP Bus.

PtrExpRomBusProp

A pointer to the IOP_BUS_PROP structure that will be used to initialize the accesses to the Expansion ROM.

PtrIopArbitDesc

A pointer to the IOP_ARBIT_DESC structure that will be used to initialize the IOP Bus arbiter of the PLX chip.

PtrIopEndianDesc

A pointer to the IOP_ENDIAN_DESC structure that will be used to initialize the IOP Bus endianness.

PtrPMProp

A pointer to the PM_PROP structure that will be used to initialize the IOP Power Management properties.

PtrVPDBaseAddress

A pointer to a U32 that will be initialized by PlxInitApi to the Vital Product Data Write-Protected Address Boundary. (The returned value specify the top of the protected area in the serial EEPROM, those bits are in units of 32-bit words)

Comments

This data type provides information about the board to the IOP API and provides data structures to be initialized to the PLX chip's default values.

BAR Space Enum Data Type

```
typedef enum _BAR_SPACE
{
  Bar0,
  Bar1,
  Bar2,
  Bar3,
  Bar4,
  Bar5,
  IopExpansionRom
} IOP_SPACE, *PIOP_SPACE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired Base Address Register when accessing the IOP memory space.

Members

Bar0
Use Base Address Register 0.

Bar1
Use Base Address Register 1.

Bar2
Use Base Address Register 2.

Bar3
Use Base Address Register 3.

Bar4
Use Base Address Register 4.

Bar5
Use Base Address Register 5.

IopExpansionRom
Use Expansion ROM base address register.

Comments

The BAR space enumerated type is used to choose the Base Address Register when accessing the IOP Bus.

Bus Index Enum Data Type

```
typedef enum _BUS_INDEX
{
  PrimaryPciBus,
  SecondaryPciBus
} BUS_INDEX;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired PCI bus.

Members

PrimaryPciBus
Use the primary PCI bus.

SecondaryPciBus
Use the secondary PCI bus.

Comments

The bus index enumerated type is used to select the PCI bus.

Device Location Data Type

```
typedef struct _DEVICE_LOCATION
{
  U32 DeviceId;
  U32 VendorId;
  U32 BusNumber;
  U32 SlotNumber;
  U8  SerialNumber [16];
} DEVICE_LOCATION, *PDEVICE_LOCATION;
```

Purpose

This data type provides information on PCI devices. It is used with the *PlxPciDeviceOpen()* and the *PlxPciDeviceFind()* PCI API functions.

Members

DeviceId

The Device ID for the PCI device. Set to MINUS_ONE_LONG if you want this member to be ignored by the PCI API function.

VendorId

The Vendor ID for the PCI device. Set to MINUS_ONE_LONG if you want this member to be ignored by the PCI API function.

BusNumber

The Bus Number where the PCI device is located. Set to MINUS_ONE_LONG if you want this member to be ignored by the PCI API function.

SlotNumber

The Slot Number where the PCI device is located. Set to MINUS_ONE_LONG if you want this member to be ignored by the PCI API function.

SerialNumber

A unique identifier for the PCI device. Set SerialNumber[0] to '\0' if you want this member to be ignored by the PCI API function. This member is usually ignored by users. The format of the serial number is: "<device name>-<index number>". The currently supported devices are: "pci9080" and "pci9054". The index number starts with 0 for each device type. An example of a serial number would be: "pci9080-0". Note that functions using serial numbers are case sensitive.

Comments

This data type contains information on each PLX PCI device and provides the appropriate driver name (used when connecting to a device, see the *PlxPciDeviceOpen()* and *PlxPciDeviceFind()* functions).

DMA Channel Descriptor Structure

```
typedef struct _DMA_CHANNEL_DESC
{
    U32 EnableReadyInput         :1;
    U32 EnableBTERMInput         :1;
    U32 EnableIopBurst           :1;
    U32 EnableWriteInvalidMode   :1;
    U32 EnableDmaEOTPin          :1;
    U32 DmaStopTransferMode      :1;
    U32 HoldIopAddrConst         :1;
    U32 HoldIopSourceAddrConst   :1;
    U32 HoldIopDestAddrConst     :1;
    U32 DemandMode               :1;
    U32 SrcDemandMode            :1;
    U32 DestDemandMode           :1;
    U32 EnableTransferCountClear :1;
    U32 WaitStates               :4;
    U32 IopBusWidth              :2;
    U32 EOTEndLink               :1;
    U32 ValidStopControl         :1;
    U32 ValidModeEnable          :1;
    U32 EnableDualAddressCycles  :1;
    U32 Reserved1                :9;
    U32 TholdForIopWrites        :4;
    U32 TholdForIopReads         :4;
    U32 TholdForPciWrites        :4;
    U32 TholdForPciReads         :4;
    U32 EnableFlybyMode          :1;
    U32 FlybyDirection           :1;
    U32 EnableDoneInt            :1;
    U32 Reserved2               :13;
    DMA_CHANNEL_PRIORITY DmaChannelPriority;
} DMA_CHANNEL_DESC, *PDMA_CHANNEL_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableReadyInput	0x100, 6 0x114, 6	0x100, 6 0x114, 6	N/A ❖
EnableBTERMInput	0x100, 7 0x114, 7	0x100, 7 0x114, 7	N/A ❖
EnableIopBurst	0x100, 8 0x114, 8	0x100, 8 0x114, 8	N/A ❖
EnableWriteInvalidMode	0x04, 4 0x100, 13	0x04, 4 0x100, 13	0x304, 4 0x200, 13

Structure Element	PCI 9080	PCI 9054	IOP 480
	0x114, 13	0x114, 13	0x220, 13 ♣
EnableDmaEOTPin	0x100, 14 0x114, 14	0x100, 14 0x114, 14	0x200, 14 0x220, 14 0x240, 14
DmaStopTransferMode	0x100, 15 0x114, 15	0x100, 15 0x114, 15	0x200, 15 0x220, 15 0x240, 15
HoldIopAddrConst	0x100, 11 0x114, 11	0x100, 11 0x114, 11	0x200, 11 0x220, 11 ♣
HoldIopSourceAddrConst	*N/A*	*N/A*	0x240, 6 ⊕
HoldIopDestAddrConst	*N/A*	*N/A*	0x240, 7 ⊕
DemandMode	0x100, 12 0x114, 12	0x100, 12	0x200, 12 0x220, 12 ♣
SrcDemandMode	N/A	N/A	0x240, 12 ⊕
DestDemandMode			0x240, 11 ⊕
EnableTransferCountClear ****Note:** Can clear transfer count only if DMA chain is located on the IOP bus. Cannot clear transfer count on DMA chains located on the PCI bus.* *This bit is always set to 1 in shuttle DMA.* *This bit is always set by the program for Shuttle DMA transfer because in PlxDmaIsr, the terminal count interrupt is cleared after the transfer count is 0*	0x100, 16* 0x114, 16*	0x100, 16 * 0x114, 16 *	0x200, 16 * 0x220, 16 * ♣
WaitStates	0x100, 2-5 0x114, 2-5	0x100, 2-5 0x114, 2-5	N/A
IopBusWidth	0x100, 0-1 0x114, 0-1	0x100, 0-1 0x114, 0-1	N/A ❖
EOTEndLink	N/A	N/A	0x200, 20 0x220, 20 ♣
ValidStopControl	N/A	N/A	0x200, 19 0x220, 19 ♣
ValidModeEnable	N/A	N/A	0x200, 18 0x220, 18 ♣
EnableDualAddressCycles	N/A	0x100, 18 0x114, 18	0x200, 17 0x220, 17 ♣
TholdForIopWrites	0x130, 0-3 0x130, 16-19	0x130, 0-3 0x130, 16-19	0x21C, 0-3 0x23C, 0-3 ♣
TholdForIopReads	0x130, 4-7 0x130, 20-23	0x130, 4-7 0x130, 20-23	0x21C, 4-7 0x23C, 4-7 ♣
TholdForPciWrites	0x130, 8-11 0x130, 24-27	0x130, 8-11 0x130, 24-27	0x21C, 8-11 0x23C, 8-11 ♣

Structure Element	PCI 9080	PCI 9054	IOP 480
TholdForPciReads	0x130, 12-15 0x130, 28-30	0x130, 12-15 0x130, 28-30	0x21C, 12-15 0x23C, 12-15 ♣
EnableFlybyMode	N/A	N/A	0x240, 10 ⊕
FlybyDirection	N/A	N/A	0x240, 9 ⊕
DmaChannelPriority	0x88, 19-20	0x88, 19-20	0x90, 4-5
EnableDoneInt ***Note:*** *This bit is always set by the program for SGL and Block DMA in order for DMA Interrupt Service Routine to clean up the SGL pool.* *This bit is always cleared by the program for Shuttle DMA so the DMA is able to work on the same element over and over again.*	0x100, 10 0x114, 10	0x100, 10 0x114, 10	0x200, 10 0x220, 10 0x240, 13

❖ This bit should be initialized in *PlxInitIopBusProperties()* for each memory region.

♣ This bit is used by Channel 0 or Channel 1.

⊕ This bit is used by Channel 2;

Purpose

Structure used to configure the DMA channel.

Members

EnableReadyInput
The Ready Input is enabled if this value is set.

EnableBTERMInput
The BTERM# Input is enabled if this value is set.

EnableIopBurst
Bursting is enabled on the IOP's local bus if this value is set.

EnableWriteInvalidMode
The write and invalidate cycles will be performed on the PCI bus for DMA transfers when this value is set.

EnableDmaEOTPin
The EOT input pin is enabled if this value is set.

DmaStopTransferMode

For Channel 0 and Channel 1:
This value states which type of DMA termination is implemented. There are two options, one is to send a BLAST to terminate the DMA transfer (for CBUS and JBUS) or negate BDIP at the nearest 16-byte boundary (for MBUS). Set the value to `AssertBLAST` for this option.
The other option is that the EOT will be asserted or DREQ# will be negated to indicate a DMA termination. Set the value to `EOTAsserted` for this option.

For Channel 2 of IOP 480:

Since the action of asserting EOT# and BLAST# pins are different than the above. We use 0 to indicate when EOT# is asserted or DREQ2# is de-asserted, the DMA controller completes the current word's transfer and completes the next word's transfer with BLAST# asserted. When set to 1, the DMA controller stops transferring data after the current word is transferred.

HoldIopAddrConst

During a DMA transfer the IOP address will stay constant (not increment) if this value is set.

HoldIopSourceAddrConst

During a DMA transfer the source IOP address will stay constant (not increment) if this value is set.

HoldIopDestAddrConst

During a DMA transfer the destination IOP address will stay constant (not increment) if this value is set.

DemandMode

When channel 0 or channel 1 is the active DMA channel, the DMA controller will work in demand mode if this value is set.

SrcDemandMode

When channel 2 is the active DMA channel, the DMA controller will work in demand mode while reading source data.

DestDemandMode

When channel 2 is the active DMA channel, the DMA controller will work in demand mode while writing destination data.

EnableTransferCountClear

When DMA chaining is enabled and this value is set the DMA controller will clear the transfer count value of a DMA descriptor block when the DMA transfer described by that DMA descriptor block is terminated.

WaitStates

The wait states inserted after the address strobe and before the data is ready on the bus is defined with this value.

IopBusWidth

The width of the IOP's local bus for the DMA channel is defined by this value.

EOTEndLink

Used only for DMA Scatter/Gather transfers. When EOT0#/EOT1# is asserted, value of 1 indicates the DMA transfer completes the current Scatter/Gather transfer and continues with the remaining Scatter/Gather transfers. When EOT0#/EOT1# is asserted, value 0 indicates the DMA transfer completes the current Scatter/Gather transfer, but does not continue with the remaining Scatter/Gather transfer.

ValidStopControl

Value of 0 indicates the DMA chaining controller continuously polls a descriptor with the Valid bit (*DMADescriptorValid)* set to 0 if the *ValidModeEnable* bit is set. Value of 1

indicates the Chaining controller stops polling when the *DMADescrptorValid* with a value of 0 is detected.

ValidModeEnable

Value of 0 indicates the *DMADescriptorValid* bit is ignored. Value of 1 indicates that DMA descriptors are processed only when the *DMADescriptorValid* bit is set. If *DMADescriptorValid* bit is set, the transfer count is 0, and the descriptor is not the last descriptor in the chain. The DMA controller then moves to the next descriptor in the chain.

EnableDualAddressCycles

When DMA chaining is enabled and this value is set the DMA controller will load the high 32-bits PCI address into the DMA dual address register. When this value is cleared, the DMA controller will not load the high PCI address.

Reserved1

This value is reserved for future definitions.

TholdForIopWrites

The number of pairs of full entries (minus 1) in the FIFO before requesting the IOP's local bus for writes.

TholdForIopReads

The number of pairs of empty entries (minus 1) in the FIFO before requesting the IOP's local bus for reads.

TholdForPciWrites

The number of pairs of full entries (minus 1) in the FIFO before requesting the PCI bus for writes.

TholdForPciReads

The number of pairs of empty entries (minus 1) in the FIFO before requesting the PCI bus for reads.

EnableFlybyMode

Enable Flyby mode if this bit is set, otherwise normal DMA addressing is used.

FlybyDirection

When Flyby mode is enabled, a value of 0 indicates the Destination address is written. Value of 1 indicates the Destination address is read.

EnableDoneInt

Enable Done interrupt if this bit is set. This bit is always set by the program for SGL and Shuttle DMA transfers in order for DMA Interrupt Service Routine to clean up the SGL pool. The user is able to set or unset this bit for Block DMA transfers. However, if the user set TerminalCountIntr (from DMA_TRANSFER_ELEMENT) to 1 in Block DMA transfer, the program will set *EnableDoneInt* to 1. (Please refer to page 5-29 for a complete explanation)

Reserved2

This value is reserved for future definitions.

DmaChannelPriority

The DMA channel priority scheme is set with this value.

DMA Channel Enum Data Type

```
typedef enum _DMA_CHANNEL
{
  IopChannel0,
  IopChannel1,
  IopChannel2,
  PrimaryPciChannel0,
  PrimaryPciChannel1,
  SecondaryPciChannel0,
  SecondaryPciChannel1
} DMA_CHANNEL, *PDMA_CHANNEL;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for requesting a DMA channel.

Members

IopChannel0
Request the IOP-IOP DMA channel 0.

IopChannel1
Request the IOP-IOP DMA channel 1.

IopChannel2
Request the IOP-IOP DMA channel2.

PrimaryPciChannel0
Request the Primary PCI-IOP DMA channel 0.

PrimaryPciChannel1
Request the Primary PCI-IOP DMA channel 1.

SecondaryPciChannel0
Request the Secondary PCI-IOP DMA channel 0.

SecondaryPciChannel1
Request the Secondary PCI-IOP DMA channel 1.

Comments

The DMA channel enumerated type is used to request a DMA channel.

DMA Channel Priority Enum Data Type

```
typedef enum _DMA_CHANNEL_PRIORITY
{
  Channel0Highest,
  Channel1Highest,
  Channel2Highest,
  Channel3Highest,
  Rotational
} DMA_CHANNEL_PRIORITY;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired DMA channel priority scheme.

Members

Channel0Highest
DMA channel 0 has the highest priority.

Channel1Highest
DMA channel 1 has the highest priority.

Channel2Highest
DMA channel 2 has the highest priority.

Channel3Highest
DMA channel 3 has the highest priority.

Rotational
Rotate the channel priority.

Comments

The DMA channel priority enumerated type is used to select the DMA channel priority scheme.

DMA Command Enum Data Type

```
typedef enum _DMA_COMMAND
{
  DmaStart,
  DmaPause,
  DmaResume,
  DmaAbort,
  DmaStatus
} DMA_COMMAND, *PDMA_COMMAND;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to control a DMA transfer.

Members

DmaStart
Start a DMA transfer.

DmaPause
Suspend a DMA transfer.

DmaResume
Resume a DMA transfer.

DmaAbort
Abort a DMA transfer.

DmaStatus
Determine the status of a DMA transfer.

Comments

The DMA command enumerated type is used to control a DMA transfer. The commands DmaStart and DmaStatus are no loger used and are provided for enum compatiblitiy only.

DMA Direction Enum Data Type

```
typedef enum _DMA_DIRECTION
{
  IopToIop,
  IopToPrimaryPci,
  PrimaryPciToIop,
  IopToSecondaryPci,
  SecondaryPciToIop,
  PrimaryPciToSecondaryPci,
  SecondaryPciToPrimaryPci
} DMA_DIRECTION, *PDMA_DIRECTION;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for providing the DMA transfer direction.

Members

IopToIop
Transfer between two IOP bus addresses.

IopToPrimaryPci
Transfer from an IOP bus address to a Primary PCI bus address.

PrimaryPciToIop
Transfer from a Primary PCI bus address to an IOP bus address.

IopToSecondaryPci
Transfer from an IOP bus address to a Secondary PCI bus address.

SecondaryPciToIop
Transfer from a Secondary PCI bus address to an IOP bus address.

PrimaryPciToSecondaryPci
Transfer from a Primary PCI bus address to a Secondary PCI bus address.

SecondaryPciToPrimaryPci
Transfer from a Secondary PCI bus address to a Primary PCI bus address.

Comments

The DMA direction enumerated type is used to state the DMA transfer direction.

DMA Resource Manager Parameters Structure

```
typedef struct _DMA_PARMS
{
    DMA_CHANNEL DmaChannel;
    PDMA_TRANSFER_ELEMENT FirstSglElement;
    PU32 WaitQueueBase;
    U32 NumberOfElements;
}DMA_PARMS, *PDMA_PARMS;
```

Affected Register Location

Not applicable.

Purpose

This data type is used for passing information from the BSP module into the DMA Resource Manager when it is initialized.

Members

DmaChannel

The DMA channel number for the given information. See the DMA Channel Enum Data Type for the possible values.

FirstSglElement

The address of the memory block that will be used for all the SGLs. This is a pointer to an array of DMA_TRANSFER_ELEMENT. The size of this array must be at least equal to the *NumberOfElements* member. Note that for some PLX chips, this array have to be located on a 16-bytes boundary.

WaitQueueBase

The address of the memory block allocated for the Wait Queue. This is a pointer to an array of U32. The size of this array must be at least equal to the *NumberOfElements* member.

NumberOfElements

The total number of SGL elements available in the SGL memory block provided. This is also the size of the waiting queue.

Comments

This data type provides memory block addresses needed by the DMA Resource Manager for each DMA channel. The DMA Resource Manager determines the end of the DMA_PARMS array by one of the follow criteria:

- The *DmaChannel* member is invalid;
- The *FirstSglElement* member is NULL;
- The *WaitQueueBase* member is NULL; or,
- The *NumberOfElements* is 0.

DMA Transfer Element Structure And SGL Address Structure

```
typedef union _DMA_TRANSFER_ELEMENT
{
    struct
    {
    #if defined(PCI_CODE)
        union
        {
            U32 LowPciAddr;
            U32 UserAddr;
        };
    #else
        U32 LowPciAddr;
    #endif
        U32 IopAddr;
        U32 TransferCount;
    #if defined(LITTLE_ENDIAN)
        U32 PciSglLoc         :1;
        U32 LastSglElement    :1;
        U32 TerminalCountIntr :1;
        U32 IopToPciDma       :1;
        U32 NextSglPtr        :28;
    #elif defined(BIG_ENDIAN)
        U32 NextSglPtr        :28;
        U32 IopToPciDma       :1;
        U32 TerminalCountIntr :1;
        U32 LastSglElement    :1;
        U32 PciSglLoc         :1;
    #else
        #error Need To Define Endian Type Used.
    #endif /* LITTLE_ENDIAN or BIG_ENDIAN */
    } Pci9080Dma;

    struct
    {
    #if defined(PCI_CODE)
        union
        {
            U32 LowPciAddr;
            U32 UserAddr;
        };
    #else
        U32 LowPciAddr;
    #endif
```

```
    U32 IopAddr;
    U32 TransferCount;
#if defined(LITTLE_ENDIAN)
    U32 PciSglLoc         :1;
    U32 LastSglElement    :1;
    U32 TerminalCountIntr :1;
    U32 IopToPciDma       :1;
    U32 NextSglPtr        :28;
#elif defined(BIG_ENDIAN)
    U32 NextSglPtr        :28;
    U32 IopToPciDma       :1;
    U32 TerminalCountIntr :1;
    U32 LastSglElement    :1;
    U32 PciSglLoc         :1;
#endif /* LITTLE_ENDIAN or BIG_ENDIAN */
    U32 HighPciAddr;
} Pci9054Dma;

struct
{
    U32 TransferCount;
#if defined(PCI_CODE)
    union
    {
        U32 LowPciAddr;
        U32 UserAddr;
        U32 SourceAddr;
    };
#else
    union loc1
    {
        U32 LowPciAddr; /* DMA channel 0, 1 */
        U32 SourceAddr; /* DMA channel 2 */
    } Loc1;
#endif

#if defined(PCI_CODE)
    union loc2
    {
        U32 IopAddr;    /* DMA channel 0, 1 */
        U32 DestAddr;   /* DMA channel 2 */
    };
#else
    union loc2
    {
```

```
            U32 IopAddr;    /* DMA channel 0, 1 */
            U32 DestAddr;   /* DMA channel 2 */
        }Loc2;
    #endif

    #if defined(LITTLE_ENDIAN)
        U32 PciSglLoc         :1;
        U32 LastSglElement    :1;
        U32 TerminalCountIntr :1;
        U32 IopToPciDma       :1;
        U32 NextSglPtr        :28;
    #elif defined(BIG_ENDIAN)
        U32 NextSglPtr        :28;
        U32 IopToPciDma       :1;
        U32 TerminalCountIntr :1;
        U32 LastSglElement    :1;
        U32 PciSglLoc         :1;
    #endif /* LITTLE_ENDIAN or BIG_ENDIAN */
        U32 HighPciAddr;
    } Iop480Dma;

    /*
       The DMA Transfer Element must always start on a 16 byte
       boundary so the following reserve field ensures this. Total size =
0x30.
     */
    U32 Reserved[12];
} DMA_TRANSFER_ELEMENT, *PDMA_TRANSFER_ELEMENT;

typedef PDMA_TRANSFER_ELEMENT SGL_ADDR, *PSGL_ADDR;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
LowPciAddr / UserAddr	0x104 0x118	0x104 0x118	0x20C 0x22C
HighPciAddr	N/A	0x134 0x138	0x218 0x238
IopAddr	0x108 0x11C	0x108 0x11C	0x210 0x230
DestAddr	N/A	N/A	0x250
SourceAddr	N/A	N/A	0x24C
TransferCount	0x10C 0x120	0x10C 0x120	0x208, 0-22 0x228, 0-22 0x248, 2-31

Structure Element	PCI 9080	PCI 9054	IOP 480
PciSglLoc	0x110, 0 0x124, 0	0x110, 0 0x124, 0	0x214, 0 0x234, 0
LastSglElement	0x110, 1 0x124, 1	0x110, 1 0x124, 1	0x214, 1 0x234, 1
TerminalCountIntr	0x110, 2 0x124, 2	0x110, 2 0x124, 2	0x214, 2 0x234, 2
IopToPciDma	0x110, 3 0x124, 3	0x110, 3 0x124, 3	0x214, 3 0x234, 3
NextSglPtr	0x110, 4-31 0x124, 4-31	0x110, 4-31 0x124, 4-31	0x214, 4-31 0x234, 4-31

Register bit that will be enabled or disabled by related functions*:

Done Interrupt Enable bit	0x100, 10 0x114, 10	0x100, 10 0x114, 10	0x200, 10 0x220, 10 0x240, 13

Note:

* On the PCI9080, PCI9054, and IOP 480, the Terminal Count Interrupt bit does not affect the behavior of the PLX chip for block DMA. Terminal Count Interrupts only apply for chaining DMA when an SGL element has completed.

The PlxDmaBlockTransfer() function offers, through the DMA_TRANSFER_ELEMENT structure, the possibility of raising an interrupt after it is done. Since this function cannot use the Terminal Count Interrupt bit, it must use the Done Interrupt Enable bit of the DMA channel mode register (Register 0x100 or 0x114, bit 10 on PCI9080 and PCI9054; Register 0x200 or 0x220, bit 10 or Register 0x240, bit 13 on IOP 480)

Purpose

Structure used to program the DMA registers.

Members

Pci9080Dma
The structure containing the DMA data specific for the PCI 9080.

Pci9054Dma
The structure containing the DMA data specific for the PCI 9054.

Iop480Dma
The structure containing the DMA data specific for the IOP 480.

Pci9080Dma.UserAddr and Pci9054Dma.UserAddr
This member should be used by host applications for SGL and Shuttle DMA only. It represents the user address (virtual) of the PCI buffer for the DMA transfer. This value is used to program the PCI Lower Address Register for a given DMA channel.

Pci9080Dma.LowPciAddr, Pci9054Dma.LowPciAddr and Iop480Dma.Loc1.LowPciAddr
The PCI buffer lower address for the DMA transfer. This value is used to program the PCI Lower Address Register for a given DMA channel.

Pci9054Dma.HighPciAddr and Iop480Dma.HighPciAddr
The PCI buffer upper address for the DMA transfer. This value is used to program the Dual Address Cycle Address Register for a given DMA channel.

Iop480Dma.Loc2.DestAddr
Destination Address Register for DMA Channel 2 of IOP 480. Indicates the Local Memory space location in which the DMA transfers start.

Iop480Dma.Loc1.SourceAddr
Source Address Register for DMA Channel 2 of IOP 480. Indicates the Local Memory space location in which the DMA transfers start.

Pci9080Dma.IopAddr, Pci9054Dma.IopAddr and Iop480Dma.Loc2.IopAddr
The IOP buffer address for the DMA transfer. This value is used to program the Local Address Register for a given DMA channel.

Pci9080Dma.TransferCount, Pci9054Dma.TransferCount and Iop480Dma.TransferCount
The number of bytes to be transferred. This value is used to program the Transfer Count Register for a given DMA channel.

Pci9080Dma.PciSglLoc, Pci9054Dma.PciSglLoc and Iop480Dma.PciSglLoc
The next SGL element is located in PCI memory if this value is set. Otherwise, the next SGL element is located in IOP memory.

Pci9080Dma.LastSglElement, Pci9054Dma.LastSglElement and Iop480Dma.LastSglElement
This is the last SGL element in the SGL if this value is set. Otherwise, the Descriptor Pointer Register points to the next SGL element in the SGL.

Pci9080Dma.TerminalCountIntr, Pci9054Dma.TerminalCountIntr and Iop480Dma.TerminalCountIntr
A DMA interrupt will be generated after completing this SGL element's DMA transfer if this value is set.

Pci9080Dma.IopToPciDma, Pci9054Dma.IopToPciDma and Iop480Dma.IopToPciDma
The DMA transfer will transfer data from IOP memory space to PCI memory space if this value is set. Otherwise, the data will be transferred from PCI memory space to IOP memory space.

Pci9080Dma.NextSglPtr, Pci9054Dma.NextSglPtr, and Iop480Dma.NextSglPtr
The pointer that points to the next SGL element in the SGL. This value is used to program the Descriptor Pointer Register for a given DMA channel.

Echo State Enum Data Type

```
typedef enum _ECHO_STATE
{
  Echo,
  NoEcho
} ECHO_STATE, *PECHO_STATE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to set the echo state of the UART Services functions.

Members

Echo
All characters received through the serial port are echoed back to the originator.

NoEcho
No characters received through the serial port are echoed back.

Comments

The echo state enumerated type is used to set the echo state of UART Services functions.

EEPROM Type Enum Data Type

```
typedef enum _EEPROM_TYPE
{
  Eeprom93CS46,
  Eeprom93CS56,
  Eeprom93CS66,
  EepromX24012,
  EepromX24022,
  EepromX24042,
  EepromX24162,
  EEPROM_UNSUPPORTED
} EEPROM_TYPE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to state the EEPROM type used to program the configuration registers at power-up.

Members

Eeprom93CS46
Use a compatible EEPROM to the National NM93CS46 EEPROM.

Eeprom93CS56
Use a compatible EEPROM to the National NM93CS56 EEPROM.

Eeprom93CS66
Use a compatible EEPROM to the National NM93CS66 EEPROM.

EepromX24012
Use a compatible EEPROM to the Xicor X24012 EEPROM.

EepromX24022
Use a compatible EEPROM to the Xicor X24022 EEPROM.

EepromX24042
Use a compatible EEPROM to the Xicor X24042 EEPROM.

EepromX24162
Use a compatible EEPROM to the Xicor X24162 EEPROM.

Comments

The EEPROM type enumerated type is used to state the type of EEPROM used to program the configuration registers at power-up.

Hot Swap Status Definition

```
#define HS_LED_ON            0x08
#define HS_BOARD_REMOVED     0x40
#define HS_BOARD_INSERTED    0x80
```

Related Register Location

Structure Element	PCI 9080	PCI 9054	PCI 9030	IOP 480
HS_LED_ON	N/A	0x18A, 19	0x48, 19	0x356, 19
HS_BOARD_REMOVED	N/A	0x18A, 21	0x48, 21	0x356, 21
HS_BOARD_INSERTED	N/A	0x18A, 22	0x48, 22	0x356, 22

Purpose

Hot Swap Status bits.

Members

HS_LED_ON
Indicate the external LED is turned on.

HS_BOARD_REMOVED
Indicate that a board is in process of being removed.

HS_BOARD_INSERTED

Indicate that a board was inserted and is being initialized.

Comments

The Hot Swap Status bits can be ORed combination

IOP Arbitration Descriptor Structure

```
typedef struct _IOP_ARBIT_DESC
{
  U32 IopBusDSGiveUpBusMode     :1;
  U32 EnableDSLockedSequence    :1;
  U32 GateIopLatencyTimerBREQo  :1;
  U32 EnableWAITInput           :1;
  U32 EnableBOFF                :1;
  U32 BOFFTimerResolution       :1;
  U32 EnableIopBusLatencyTimer  :1;
  U32 EnableIopBusPauseTimer    :1;
  U32 EnableIopArbiter          :1;
  U32 IopArbitrationPriority    :3;
  U32 BOFFDelayClocks           :4;
  U32 IopBusLatencyTimer        :8;
  U32 IopBusPauseTimer          :8;
  U32 EnableIopBusTimeOut       :1;
  U32 IopBusTimeout             :15;
  U32 Reserved                  :16;
} IOP_ARBIT_DESC, *PIOP_ARBIT_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
IopBusDSGiveUpBusMode	0x88, 21	0x88, 21	0x90, 8
EnableDSLockedSequence	0x88, 22	0x88, 22	0x90, 9
GateIopLatencyTimerBREQo	0x88, 27	0x88, 27	0x90, 10
EnableWAITInput	N/A	0x88, 31	N/A
EnableBOFF	N/A	N/A	0x90, 16
BOFFTimerResolution	N/A	N/A	0x90, 21
EnableIopBusLatencyTimer	0x88, 16	0x88, 16	0x8C, 8
EnableIopBusPauseTimer	0x88, 17	0x88, 17	0x8C, 24
EnableIopArbiter	N/A	N/A	0x90, 0
IopArbitrationPriority	N/A	N/A	0x90, 1-3
BOFFDelayClocks	N/A	N/A	0x90, 17-20
IopBusLatencyTimer	0x88, 0-7	0x88, 0-7	0x8C, 0-7
IopBusPauseTimer	0x88, 8-15	0x88, 8-15	0x8C, 16-23
EnableIopBusTimeout	N/A	N/A	0x88, 15
IopBusTimeout	N/A	N/A	0x88, 0-14

Purpose

Structure used to describe the IOP bus arbitration.

Members

IopBusDSGiveUpBusMode
The Direct Slave access releases the IOP bus when the Direct Slave write FIFO becomes empty or the Direct Slave read FIFO becomes full when this value is set.

EnableDSLockedSequence
The Direct Slave locked sequences mode is enabled when this value is set.

GateIopLatencyTimerBREQo
The IOP bus latency timer is gated with BREQo when this value is set.

EnableWAITInput
The WAIT# input is enabled when this bit is set.

EnableBOFF
the IOP 480 can assert the BOFF# pin when this value is set.

BOFFTimerResolution
When this value is set the LSB of the IOP 480's BOFF timer is set to be 64 clocks. Otherwise, the LSB is set to 8 clocks.

EnableIopBusLatencyTimer
The IOP bus latency timer is enabled when this value is set.

EnableIopBusPauseTimer
The IOP bus pause timer is enabled when this value is set.

EnableIopArbiter
The IOP bus arbiter is enabled when this value is set.

IopArbitrationPriority
The IOP bus arbitration priority is set with this value.

BOFFDelayClocks
This value contains the number of delay clocks in which a Direct Slave bus request is pending and a Local Direct Master access is in progress and not being granted the bus before asserting BOFF#.

IopBusLatencyTimer
This value contains the number of IOP bus clocks cycles that the PCI device will hold the IOP bus before releasing it to another requester.

IopBusPauseTimer
This value contains the number of IOP bus clocks cycles before requesting the IOP bus after releasing the IOP bus for internal masters.

EnableIopBusTimeout
Value of 1 indicates the Local Bus Timeout Timer is enabled. If this value is set, *IopBusTimeout* must be specified.

IopBusTimeout
Local Bus Timeout Value. Value loaded into a timer at the beginning of the Local Bus transfer.

IOP Bus Properties Structure

```
typedef struct _IOP_BUS_PROP
{
    U32 EnableReadyRecover          :1;
    U32 EnableReadyInput            :1;
    U32 EnableBTERMInput            :1;
    U32 DisableReadPrefetch         :1;
    U32 EnableReadPrefetchCount     :1;
    U32 ReadPrefetchCounter         :4;
    U32 EnableBursting              :1;
    U32 EnableIopBusTimeoutTimer    :1;
    U32 BREQoTimerResolution        :1;
    U32 EnableIopBREQo              :1;
    U32 BREQoDelayClockCount        :4;
    U32 MapInMemorySpace            :1;
    U32 OddParitySelect             :1;
    U32 EnableParityCheck           :1;
    U32 MemoryWriteProtect          :1;
    U32 InternalWaitStates          :4;
    U32 PciRev2_1Mode               :1;
    U32 IopBusWidth                 :2;
    U32 Reserved1                   :4;
    U32 Iop480WADWaitStates         :4;
    U32 Iop480WDDWaitStates         :4;
    U32 Iop480WDLYDelayStates       :3;
    U32 Iop480WHLDHoldStates        :3;
    U32 Iop480WRCVRecoverStates     :3;
    U32 Reserved2                   :15;
    U32 Iop480RADWaitStates         :4;
    U32 Iop480RDDWaitStates         :4;
    U32 Iop480RDLYADelayStates      :3;
    U32 Iop480RDLYDDelayStates      :3;
    U32 Iop480RRCVRecoverStates     :3;
    U32 Reserved3                   :15;
    U32 DramRefreshEnable           :1
    U32 DramRefreshInterval         :11;
    U32 Iop480TWRdelay              :2;
    U32 Iop480W2Wdelay              :2;
    U32 Iop480A2Cdelay              :2;
    U32 Iop480RRCVdelay             :2;
    U32 Iop480PRCGdelay             :2;
    U32 Iop480WCWdelay              :2;
    U32 Iop480RCWdelay              :2;
    U32 Iop480C2Cdelay              :2;
```

```
    U32 Iop480R2Cdelay              :2;
    U32 Iop480R2Rdelay              :2;
} IOP_BUS_PROP, *PIOP_BUS_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableReadyRecover	N/A	N/A	0x100, 19 0x114, 19 0x128, 19 0x13C, 19 0x150, 19
EnableReadyInput	0x98, 6 0x98, 22 0x178, 6	0x98, 6 0x98, 22 0x178, 6	0x100, 9 0x114, 9 0x128, 9 0x13C, 9
EnableBTERMInput	0x98, 7 0x98, 23 0x178, 7	0x98, 7 0x98, 23 0x178, 7	0x100, 10 0x114, 10 0x128, 10 0x13C, 10 0x150, 10 0x168, 10
DisableReadPrefetch ***Note:*** *By setting the Read Prefetch Count to 00 disables Read Prefetching.*	0x98, 8 0x98, 9 0x178, 9	0x98, 8 0x98, 9 0x178, 9	0x100, 17-18 0x114, 17-18 0x128, 17-18 0x13C, 17-18 0x150, 17-18 0x168, 17-18
EnableReadPrefetchCount	0x98, 10 0x178, 10	0x98, 10 0x178, 10	0x100, 16 0x114, 16 0x128, 16 0x13C, 16 0x150, 16 0x168, 16
ReadPrefetchCounter	0x98, 11-14 0x178, 11-14	0x98, 11-14 0x178, 11-14	0x100, 17-18 0x114, 17-18 0x128, 17-18 0x13C, 17-18 0x150, 17-18 0x168, 17-18
EnableBursting	0x98, 24 0x98, 26 0x178, 8	0x98, 24 0x98, 26 0x178, 8	0x100, 8 0x114, 8 0x128, 8 0x13C, 8 0x150, 8

Structure Element	PCI 9080	PCI 9054	IOP 480
			0x168, 8
EnableIopBusTimeoutTimer	N/A	N/A	0x100, 14 0x114, 14 0x128, 14 0x168, 14
BREQoTimerResolution	0x94, 5	0x94, 5	N/A
EnableIopBREQo	0x94, 4	0x94, 4	N/A
BREQoDelayClockCount	0x94, 0-3	0x94, 0-3	N/A
MapInMemorySpace	0x80, 0 0x170,0	0x80, 0 0x170,0	0xA8, 0 0xB0, 0
OddParitySelect	N/A	N/A	0x100, 13 0x114, 13 0x128, 13 0x13C, 13 0x150, 13 0x168, 13
EnableParityCheck	N/A	N/A	0x100, 12 0x114, 12 0x128, 12 0x13C, 12 0x150, 12 0x168, 12
MemoryWriteProtect	N/A	N/A	0x100, 11 0x114, 11 0x128, 11 0x13C, 11 0x150, 11 0x168, 11
PciRev2_1Mode	0x88, 24	0x88, 24	0x98, 25
IopBusWidth	0x98, 0-1 0x98, 16-17 0x178, 0-1	0x98, 0-1 0x98, 16-17 0x178, 0-1	0x100, 0-1 0x114, 0-1 0x128, 0-1 0x13C, 0-1 0x150, 0-1 0x168, 0-1
InternalWaitStates	0x98, 2-5 0x98, 18-21 0x178, 2-5	0x98, 2-5 0x98, 18-21 0x178, 2-5	N/A
Iop480WADWaitStates	N/A	N/A	0x104, 0-3 0x118, 0-3 0x12C, 0-3 0x140, 0-3

Structure Element	PCI 9080	PCI 9054	IOP 480
Iop480WDDWaitStates	N/A	N/A	0x104, 4-7 0x118, 4-7 0x12C, 4-7 0x140, 4-7
Iop480WDLYDelayStates	N/A	N/A	0x104, 8-10 0x118, 8-10 0x12C, 8-10 0x140, 8-10
Iop480WHLDHoldStates	N/A	N/A	0x104, 11-13 0x118, 11-13 0x12C, 11-13 0x140, 11-13
Iop480WRCVRecoverStates	N/A	N/A	0x104, 16-18 0x118, 16-18 0x12C, 16-18 0x140, 16-18
Iop480RADWaitStates	N/A	N/A	0x108, 0-3 0x11C, 0-3 0x130, 0-3 0x144, 0-3
Iop480RDDWaitStates	N/A	N/A	0x108, 4-7 0x11C, 4-7 0x130, 4-7 0x144, 4-7
Iop480RDLYDelayStates	N/A	N/A	0x108, 8-10 0x11C, 8-10 0x130, 8-10 0x144, 8-10
Iop480RDLYDDelayStates	N/A	N/A	0x108, 11-13 0x11C, 11-13 0x130, 11-13 0x144, 11-13
Iop480RRCVRecoverStates	N/A	N/A	0x108, 16-18 0x11C, 16-18 0x130, 16-18 0x144, 16-18
DramRefreshEnable	N/A	N/A	0x154, 23
DramRefreshInterval	N/A	N/A	0x154, 12-22
Iop480TWRdelay	N/A	N/A	0x15C, 20-21
Iop480W2Wdelay	N/A	N/A	0x15C, 18-19
Iop480A2Cdelay	N/A	N/A	0x15C, 16-17
Iop480RRCVdelay	N/A	N/A	0x15C, 12-13

Structure Element	PCI 9080	PCI 9054	IOP 480
Iop480PRCGdelay	N/A	N/A	0x15C, 10-11
Iop480WCWdelay	N/A	N/A	0x15C, 8-9
Iop480RCWdelay	N/A	N/A	0x15C, 6-7
Iop480C2Cdelay	N/A	N/A	0x15C, 4-5
Iop480R2Cdelay	N/A	N/A	0x15C, 2-3
Iop480R2Rdelay	N/A	N/A	0x15CC, 0-1

Purpose

Structure used to describe the local bus characteristics.

Members

EnableReadyRecover
Value of 0 indicates the READY# pin is driven only with the IOP 480 internal ready status. Value of 1 indicates the READY# pin is also driven active during recovery states. Can be used to prevent external Local Bus Masters from starting a new cycle until the recovery period expires.

EnableReadyInput
The Ready input is enabled when this value is set.

EnableBTERMInput
The BTERM input is enabled when this value is set.

DisableReadPrefetch
Read prefetching is disabled when this value is set.

EnableReadPrefetchCount
The read prefetch counter is enabled when this bit is set. If enabled the PCI device reads up to the number of U32s specified in the prefetch counter. If disabled the PCI device ignores the prefetch counter and reads continuously until terminated by the PCI bus.

ReadPrefetchCounter
Stores the number of values that can be prefetched.

EnableBursting
Bursting is enabled if this value is set. If bursting is disabled then the PCI device performs continuous single cycle accesses for burst PCI read/write cycles.

EnableIopBusTimeoutTimer
The IOP bus timeout timer is enabled if an external master controls the IOP bus when this value is set.

BREQoTimerResolution
When this value is set the LSB of the BREQo timer changes from 8 to 64 clocks.

EnableIopBREQo
The PCI device can assert the BREQo output to the IOP bus when this value is set.

BREQoDelayClockCount

The value represents the number of IOP bus clocks in which a Direct Slave HOLD request is pending and a Direct Master access is in progress and not being granted the bus before asserting BREQo.

MapInMemorySpace

The local space region is mapped into PCI memory space when this value is set.

OddParitySelect

Select odd parity checking on the IOP bus when this value is set. Otherwise, select even parity checking.

EnableParityCheck

The parity of the IOP bus data will be checked when this value is set.

MemoryWriteProtect

When this value is set, MDQM[3:0]# and WR# signals are not asserted during write cycles (write-protected). Otherwise, these signals are asserted.

PciRev2_1Mode

The PCI device operates in Delayed Transaction mode for Direct Slave reads when this value is set. Otherwise, the PCI device does not return a TRDY# signal to the PCI host until the read data are available.

IopBusWidth

The width of the IOP bus.

InternalWaitStates

The number of wait states inserted after the address is presented on the IOP bus until the data is ready. The value must be between 0-15.

Reserved1

This value is reserved for future definitions.

Iop480WADWaitStates

This value contains the number of Write Address-to-Data wait states to insert for the IOP 480. The value must be between 0-15.

Iop480WDDWaitStates

This value contains the number of Write Data-to-Data wait states (used when bursting) to insert for the IOP 480. The value must be between 0-15.

Iop480WDLYWaitStates

This value contains the number of Write Enable Delay states to insert for the IOP 480. The value must be between 0-15.

Iop480WHLDWaitStates

This value contains the number of Write Hold states to insert for the IOP 480. The value must be between 0-15.

Iop480WRCVRecoverStates

This value contains the number of Write Recovery states to insert for the IOP 480. The value must be between 0-15.

Reserved2
This value is reserved for future definitions.

Iop480RADWaitStates
This value contains the number of Read Address-to-Data wait states to insert for the IOP 480. The value must be between 0-15.

Iop480RDDWaitStates
This value contains the number of Read Data-to-Data wait states (used when bursting) to insert for the IOP 480. The value must be between 0-15.

Iop480RDLYADelayStates
SRAM timing parameter, this value contains the number of Read Enable Delay states to insert for the IOP 480. The value must be between 0-15.

Iop480RDLYDDelayStates
SRAM timing parameter, this value contains the number of Read Enable Delay states to insert for the IOP 480. The value must be between 0-15.

Iop480RRCVRecoverStates
This value contains the number of Read Recovery states to insert for the IOP 480. The value must be between 0-15.

Reserved3
This value is reserved for future definitions.

DramRefreshEnable
Value of 0 indicates that Refresh cycles are disabled. Value of 1 indicates that Refresh cycles are enabled.

DramRefreshInterval
The interval between Refresh cycles in termis of LCLK Clock cycles. The value can be calculated as follows: DramRefreshInterval = refresh rate x LCLK clock frequency.

Iop480TWRdelay
DRAM timing parameter, this value is the delay between the last word written during a Single or Burst Write cycle and a Precharge command.

Iop480W2Wdelay
DRAM timing parameter, this value is the delay between words of a Burst Write for an SDRAM Write cycle.

Iop480A2Cdelay
DRAM timing parameter, this value is the Active to Read/Write command delay for an SDRAM Read or Write cycle.

Iop480RRCVdelay
DRAM timing parameter, this value is the number of recovery states after the end of a Single or Burst EDO or an SDRAM Read cycle.

Iop480PRCGdelay
DRAM timing parameter, this value is the RAS precharge delay for EDO or SDRAMs.

Iop480WCWdelay
DRAM timing parameter, this value is the CAS pulse width for an EDO Write cycle.

Iop480RCWdelay

DRAM timing parameter, this value is the CAS pulse width for EDO Read cycle.

Iop480C2Cdelay

DRAM timing parameter, this value is the delay from column address to CAS for EDO cycle.

Iop480R2Cdelay

DRAM timing parameter, this value is the delay from RAS to column address for EDO cycle.

Iop480R2Rdelay

DRAM timing parameter, this value is the delay from row address to RAS for EDO cycle.

IOP Endian Descriptor Structure

```
typedef struct _IOP_ENDIAN_DESC
{
    U32 BigEIopSpace0              :1;
    U32 BigEIopSpace1              :1;
    U32 BigEExpansionRom           :1;
    U32 BigEMaster480LCS0          :1;
    U32 BigEMaster480LCS1          :1;
    U32 BigEMaster480LCS2          :1;
    U32 BigEMaster480LCS3          :1;
    U32 BigEMaster480Dram          :1;
    U32 BigEMaster480Default       :1;
    U32 BigEIopBusRegion0          :1;
    U32 BigEIopBusRegion1          :1;
    U32 BigEIopBusRegion2          :1;
    U32 BigEIopBusRegion3          :1;
    U32 BigEDramBusRegion          :1;
    U32 BigEDefaultBusRegion       :1;
    U32 BigEDmaChannel0            :1;
    U32 BigEDmaChannel1            :1;
    U32 BigEIopConfigRegAccess     :1;
    U32 BigEDirectMasterAccess     :1;
    U32 BigEIopConfigRegInternal   :1;
    U32 BigEDirectMasterInternal   :1;
    U32 BigEByteLaneMode           :1;
    U32 BigEByteLaneModeLBR0       :1;
    U32 BigEByteLaneModeLBR1       :1;
    U32 BigEByteLaneModeLBR2       :1;
    U32 BigEByteLaneModeLBR3       :1;
    U32 BigEByteLaneModeDRAMBR     :1;
    U32 BigEByteLaneModeDefBR      :1;
    U32 Reserved1                  :4;
    U32 ReservedForFutureUse;
} IOP_ENDIAN_DESC, *PIOP_ENDIAN_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
BigEIopSpace0	0x8C, 2	0x8C, 2	N/A
BigEIopSpace1	0x8C, 5	0x8C, 5	N/A
BigEExpansionRom	0x8C, 3	0x8C, 3	N/A
BigEMaster480LCS0	N/A	N/A	0x100, 4
BigEMaster480LCS1	N/A	N/A	0x114, 4
BigEMaster480LCS2	N/A	N/A	0x128, 4

Structure Element	PCI 9080	PCI 9054	IOP 480
BigEMaster480LCS3	N/A	N/A	0x13C, 4
BigEMaster480Dram	N/A	N/A	0x150, 4
BigEMaster480Default	N/A	N/A	0x168, 4
BigEIopBusRegion0	N/A	N/A	0x100, 2
BigEIopBusRegion1	N/A	N/A	0x114, 2
BigEIopBusRegion2	N/A	N/A	0x128, 2
BigEIopBusRegion3	N/A	N/A	0x13C, 2
BigEDramBusRegion	N/A	N/A	0x150, 2
BigEDefaultBusRegion	N/A	N/A	0x168, 2
BigEIopConfigRegAccess	0x8C, 0	0x8C, 0	0x94, 0
BigEDirectMasterAccess	0x8C, 1	0x8C, 1	0x94, 1
BigEIopConfigRegInternal	N/A	N/A	0x94, 2
BigEDirectMasterInternal	N/A	N/A	0x94, 3
BigEDmaChannel0	0x8C, 7	0x8C, 7	N/A
BigEDmaChannel1	0x8C, 6	0x8C, 6	N/A
BigEByteLaneMode	0x8C, 4	0x8C, 4	N/A
BigEByteLaneModeLBR0	N/A	N/A	0x100, 3
BigEByteLaneModeLBR1	N/A	N/A	0x114, 3
BigEByteLaneModeLBR2	N/A	N/A	0x128, 3
BigEByteLaneModeLBR3	N/A	N/A	0x13C, 3
BigEByteLaneModeDRAMBR	N/A	N/A	0x150, 3
BigEByteLaneModeDefBR	N/A	N/A	0x168, 3

Purpose

Structure used to describe the IOP bus endian data ordering.

Members

BigEIopSpace0
The local space 0 is configured with big endian data ordering if this value is set.

BigEIopSpace1
The local space 1 is configured with big endian data ordering if this value is set.

BigEExpansionRom
The Expansion ROM is configured with big endian data ordering if this value is set.

BigEMaster480LCS0
When IOP 480 CPU is the local Bus Master, the LCS0 is configured with big endian data ordering if this value is set.

BigEMaster480LCS1
When IOP 480 CPU is the local Bus Master, the LCS1 is configured with big endian data ordering if this value is set.

BigEMaster480LCS2
When IOP 480 CPU is the local Bus Master, the LCS2 is configured with big endian data ordering if this value is set.

BigEMaster480LCS3
When IOP 480 CPU is the local Bus Master, the LCS3 is configured with big endian data ordering if this value is set.

BigEMaster480Dram
When IOP 480 CPU is the local Bus Master, the DRAM bus region is configured with big endian data ordering if this value is set.

BigEMaster480Default
When IOP 480 CPU is the local Bus Master, the default bus region is configured with big endian data ordering if this value is set.

BigEIopBusRegion0
The local bus region 0 is configured with big endian data ordering if this value is set.

BigEIopBusRegion1
The local bus region 1 is configured with big endian data ordering if this value is set.

BigEIopBusRegion2
The local bus region 2 is configured with big endian data ordering if this value is set.

BigEIopBusRegion3
The local bus region 3 is configured with big endian data ordering if this value is set.

BigEDramBusRegion
The DRAM bus region is configured with big endian data ordering if this value is set.

BigEDefaultBusRegion
The default bus region is configured with big endian data ordering if this value is set.

BigEDmaChannel0
The DMA channel 0 is configured with big endian data ordering if this value is set.

BigEDmaChannel1
The DMA channel 1 is configured with big endian data ordering if this value is set.

BigEIopConfigRegAccess
The IOP configuration register access is configured with big endian data ordering if this value is set.

BigEDirectMasterAccess
The direct master access is configured with big endian data ordering if this value is set.

BigEIopConfigRegInternal
For internal IOP 480 CPU, the IOP configuration register is configured with big endian data ordering if this value is set.

BigEDirectMasterInternal
For internal IOP 480 CPU, the direct master access is configured with big endian data ordering if this value is set.

BigEByteLaneMode

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

BigEByteLaneModeLBR0

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for Local bus region 0. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

BigEByteLaneModeLBR1

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for local bus region 1. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

BigEByteLaneModeLBR2

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for local bus region 2. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

BigEByteLaneModeLBR3

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for local bus region 3. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

BigEByteLaneModeDRAMBR

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for DRAM bus region. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeDefBR

When this value is set use D31:16 for a 16-bit bus and D31:24 for an 8-bit bus for Default bus region. Otherwise, use D15:0 for a 16-bit bus and D7:0 for an 8-bit bus.

Reserved1

This value is reserved for future definitions.

ReservedForFutureUse

This value is reserved for future definitions.

IOP Space Enum Data Type

```
typedef enum _IOP_SPACE
{
  IopSpace0,
  IopSpace1,
  IopSpace2,
  IopSpace3,
  MsLcs0,
  MsLcs1,
  MsLcs2,
  MsLcs3,
  MsDram,
  ExpansionRom
} IOP_SPACE, *PIOP_SPACE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to select the desired Local Space register when accessing the IOP memory space.

Members

IopSpace0
Use Local Space 0 base address register.

IopSpace1
Use Local Space 1 base address register.

IopSpace2
Use Local Space 2 base address register.

IopSpace3
Use Local Space 3 base address register.

MsLcs0
Use LCS0 Base Address registers.

MsLcs1
Use LCS1 Base Address registers.

MsLcs2
Use LCS2 Base Address registers.

MsLcs3
Use LCS3 Base Address registers.

MsDram4
Use DRAM Base Address registers

ExpansionRom
Use Expansion ROM base address register.

Mailbox ID Enum Data Type

```
typedef enum _MAILBOX_ID
{
  MailBox0,
  MailBox1,
  MailBox2,
  MailBox3,
  MailBox4,
  MailBox5,
  MailBox6,
  MailBox7
} MAILBOX_ID;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
MailBox0	0xC0	0xC0	0x180
MailBox1	0xC4	0xC4	0x184
MailBox2	0xC8	0xC8	0x188
MailBox3	0xCC	0xCC	0x18C
MailBox4	0xD0	0xD0	0x190
MailBox5	0xD4	0xD4	0x194
MailBox6	0xD8	0xD8	0x198
MailBox7	0xDC	0xDC	0x19C

Purpose

Enumerated type used for choosing the desired mailbox register.

Members

MailBox0 Use mailbox 0 register.

MailBox1 Use mailbox 1 register.

MailBox2 Use mailbox 2 register.

MailBox3 Use mailbox 3 register.

MailBox4 Use mailbox 4 register.

MailBox5 Use mailbox 5 register.

MailBox6 Use mailbox 6 register.

MailBox7 Use mailbox 7 register.

PCI Arbitration Descriptor Structure

```
typedef struct _PCI_ARBIT_DESC
{
  U32 PciHighPriority;
} PCI_ARBIT_DESC, *PPCI_ARBIT_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
PciHighPriority	N/A	N/A	0x98, 17

Purpose

Structure used to describe the PCI bus arbitration.

Note: *This structure will be determined at a future date.*

Members

PciHighPriority

Value of 0 indicates the IOP 480 participates in a round-robin arbitration with the other PCI Masters. Value of 1 indicates that the other PCI Bus Masters participate in their own round-robin arbitration. The winner of this arbitration then arbitrates for the PCI Bus with the IOP 480.

PCI Bus Properties Structure

```
typedef struct _PCI_BUS_PROP
{
  U32 PciRequestMode           :1;
  U32 DmPciReadMode            :1;
  U32 EnablePciArbiter         :1;
  U32 EnableWriteInvalidMode   :1;
  U32 DmPrefetchLimit          :1;
  U32 PciReadNoWriteMode       :1;
  U32 PciReadWriteFlushMode    :1;
  U32 PciReadNoFlushMode       :1;
  U32 EnableRetryAbort         :1;
  U32 WFifoAlmostFullFlagCount :5;
  U32 DmWriteDelay             :2;
  U32 ReadPrefetchMode         :2;
  U32 IoRemapSelect            :1;
  U32 EnablePciBusMastering    :1;
  U32 EnableMemorySpaceAccess  :1;
  U32 EnableIoSpaceAccess      :1;
  U32 Reserved1                :10;
  U32 ReservedForFutureUse;
} PCI_BUS_PROP, *PPCI_BUS_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
PciRequestMode	0x88, 23	0x88, 23	0x98, 26
DmPciReadMode	0xA8, 4	0xA8, 4	0xD0, 6
EnablePciArbiter	N/A	N/A	0x98, 16
EnableWriteInvalidMode	0x04, 4 0xA8, 9	0x04, 4 0xA8, 9	0x304, 4 0xD0, 7
DmPrefetchLimit	0xA8, 11	0xA8, 11	0xD0, 5
PciAddressSpaceBusWidth	N/A	N/A	N/A
PciReadNoWriteMode	0x88, 25	0x88, 25	0x98, 24
PciReadWriteFlushMode	0x88, 26	0x88, 26	0x98, 23
PciReadNoFlushMode	0x88, 28	0x88, 28	0x98, 22
EnableRetryAbort	N/A	N/A	0x98, 21
WfifoAlmostFullFlagCount	0xA8, 5-8 0xA8, 10	0xA8, 5-8 0xA8, 10	0xD0, 8-12
DmWriteDelay	0xA8, 14-15	0xA8, 14-15	0xD0, 13-14
ReadPrefetchMode	0xA8, 3 0xA8, 12	0xA8, 3 0xA8, 12	0xD0, 3-4
IoRemapSelect	0xA8, 13	0xA8, 13	0xD0, 15
EnablePciBusMastering	N/A	0x04, 2	0x304, 2

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableMemorySpaceAccess	N/A	N/A	0x304, 1
EnableIoSpaceAccess	N/A	N/A	0x304, 0

Purpose

Structure used to describe the PCI bus characteristics.

Members

PciRequestMode

When this value is set the PCI device negates REQ0# when it asserts FRAME# during a bus master cycle. Otherwise, the PCI device leaves REQ0# asserted for the entire bus master cycle.

DmPciReadMode

When this value is set the PCI device should keep the PCI bus and de-assert IRDY# when the read FIFO becomes full. Otherwise, the PCI device should release the PCI bus when the read FIFO becomes full.

EnablePciArbiter

The PCI arbiter is enabled when this value is set. Otherwise, the PCI arbiter is disabled and the PCI device uses REQ0# and GNT0# to acquire the PCI bus.

EnableWriteInvalidMode

The write and invalidate cycles will be performed on the PCI bus for Direct Master accesses when this value is set.

DmPrefetchLimit

The prefetching is terminated at 4K boundaries when this value is set.

PciAddressSpaceBusWidth

When this value is set the PCI bus width for PCI space is 64 bits. Otherwise, the PCI bus width is 32 bits.

PciReadNoWriteMode

When this value is set PCI device forces a retry on writes if read is pending. Otherwise, the PCI device allows writes while a read is pending.

PciReadWriteFlushMode

When this value is set PCI device submits a request to flush a pending read cycle if a write cycle is detected. Otherwise, the PCI device submits a request to not effect pending reads when a write cycle occurs.

PciReadNoFlushMode

When this value is set PCI device submits a request to not flush the read FIFO if the PCI read cycle completes. Otherwise, the PCI device submits a request to flush the read FIFO if the PCI read cycle completes.

EnableRetryAbort

The IOP 480 treats 256 Master consecutive retries to a Target as a Target Abort when this value is set. Otherwise, the IOP 480 attempts Master Retries indefinitely.

WFifoAlmostFullFlagCount
This value sets the retry limit before asserting an IOP NMI signal.

DmWriteDelay
This value sets the delay clocks placed between the PCI bus request and the start of direct master burst write cycle.

ReadPrefetchMode
This value sets the amount of data that can be prefetched from the PCI bus and enables or disables read prefetching.

IoRemapSelect
When this value is set the PCI address bits 31:16 are forced to zero. Otherwise, the address bits 31:16 in the Direct Master Remap register will be used on the PCI bus.

EnablePciBusMastering
Set this value to 1 allows the device to behave as a bus master. Clear this bit disables the device from generating Bus Master accesses.

EnableMemorySpaceAccess
Set this value to 1 allows the device to respond to Memory Space accesses.

EnableIoSpaceAccess
Set this value to 1 allows the device to respond to I/O Space accesses.

Reserved1
This value is reserved for future definitions.

ReservedForFutureUse: This value is reserved for future definitions.

PCI Memory Data Type

```
typedef struct _PCI_MEMORY
{
    U32 UserAddr;
    PHYSICAL_ADDRESS PhysicalAddr;
    U32 Size;
}PCI_MEMORY, *PPCI_MEMORY;
```

Purpose

This data type provides information on a Physical Memory Buffer (PMB) located in the device driver code.

Members

UserAddr
User Virtual Address for the PMB.

PhysicalAddr
Physical Address for the PMB.

Size
The size for the PMB.

PCI Space Enum Data Type

```
typedef enum _PCI_SPACE
{
  PciMemSpace,
  PciIoSpace
} PCI_SPACE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired PCI Address Space access.

Members

PciMemSpace
Use PCI memory cycles when accessing the PCI bus.

PciIoSpace
Use PCI I/O cycles when accessing the PCI bus.

Pin State Enum Data Type

```
typedef enum _PIN_STATE
{
  Inactive,
  Active
} PIN_STATE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to set the state of a USER pin.

Members

Inactive

Set the USER pin to inactive state. If the USER pin is active low then setting the USER pin inactive would cause it to go high.

Active

Set the USER pin to active state. If the USER pin is active low then setting the USER pin active would cause it to go low.

PLX Interrupt Structure

```
typedef struct _PLX_INTR
{
  U32 InboundPost       :1;
  U32 OutboundPost      :1;
  U32 OutboundOverflow  :1;
  U32 OutboundOption    :1;
  U32 IopDmaChannel0    :1;
  U32 PciDmaChannel0    :1;
  U32 IopDmaChannel1    :1;
  U32 PciDmaChannel1    :1;
  U32 IopDmaChannel2    :1;
  U32 PciDmaChannel2    :1;
  U32 Mailbox0          :1;
  U32 Mailbox1          :1;
  U32 Mailbox2          :1;
  U32 Mailbox3          :1;
  U32 Mailbox4          :1;
  U32 Mailbox5          :1;
  U32 Mailbox6          :1;
  U32 Mailbox7          :1;
  U32 IopDoorbell       :1;
  U32 PciDoorbell       :1;
  U32 SerialPort1       :1;
  U32 SerialPort2       :1;
  U32 BIST              :1;
  U32 PowerManagement   :1;
  U32 PciMainInt        :1;
  U32 IopToPciInt       :1;
  U32 IopMainInt        :1;
  U32 PciAbort          :1;
  U32 PciReset          :1;
  U32 PciPME            :1;
  U32 Enum              :1;
  U32 PciENUM           :1;
  U32 IopBusTimeout     :1;
  U32 AbortLSERR        :1;
  U32 ParityLSERR       :1;
  U32 RetryAbort        :1;
  U32 LocalParityLSERR  :1;
  U32 PciSERR           :1;
  U32 IopRefresh        :1;
  U32 PciINTApin        :1;
  U32 IopINTIpin        :1;
```

```
  U32 TargetAbort       :1;
  U32 Ch1Abort          :1;
  U32 Ch0Abort          :1;
  U32 DMAbort           :1;
  U32 IopToPciInt_2     :1;
  U32 Reserved          :18;
} PLX_INTR, *PPLX_INTR;
```

Affected Register Location

E: Enable interrupt bit location

A: Active interrupt bit location

C: Write to this bit to clear the interrupt

Structure Element	PCI 9080	PCI 9054	IOP 480	PCI 9030
InboundPost	E 0x168, 4 A 0x168, 5	E 0x168, 4 A 0x168, 5	E 0x28, 4 A 0x28, 5 C 0x28, 4	N/A
OutboundPost	E 0xB4, 3 A 0xB0, 3	E 0xB4, 3 A 0xB0, 3	E 0x34, 3 A 0x30, 3 C 0x34, 3	N/A
OutboundOverflow	E 0x168, 6 A 0x168, 7	E 0x168, 6 A 0x168, 7	E 0x28, 6 A 0x28, 7 C 0x28, 7	N/A
OutboundOption	N/A	N/A	E 0x28, 8 A 0x30, 3 C 0x28, 8	N/A
PciDmaChannel0 ***Note:** DMA Interrupt is routed to PCI interrupt.*	E 0x100*, 17 A 0xE8, 21	E 0x100*, 17 A 0xE8, 21	E 0x1B4, 8 A 0x1B0, 8 C 0x204, 3	N/A
IopDmaChannel0	E 0xE8, 18 A 0xE8, 21	E 0xE8, 18 A 0xE8, 21	E 0x1BC, 8 A 0x1B8, 8 C 0x1B8, 8	N/A
PciDmaChannel1 ***Note:** DMA Interrupt is routed to PCI interrupt.*	E 0x114*, 17 A 0xE8, 22	E 0x114*, 17 A 0xE8, 22	E 0x1B4, 9 A 0x1B0, 9 C 0x224, 3	N/A
IopDmaChannel1	E 0xE8, 19 A 0xE8, 22	E 0xE8, 19 A 0xE8, 22	E 0x1BC, 9 A 0x1B8, 9 C 0x1B8, 9	N/A
PciDmaChannel2	N/A	N/A	E 0x1B4, 10 A 0x1B0, 10 C 0x244, 3	N/A
IopDmaChannel2	N/A	N/A	E 0x1BC, 10 A 0x1B8, 10	N/A
Mailbox0 * ***Note:** Only one bit enables*	E 0xE8, 3*	E 0xE8, 3*	E 0x1BC, 24	N/A

Structure Element	PCI 9080	PCI 9054	IOP 480	PCI 9030
Mailbox 0-3 Interrupts. No individual enabling of interrupts.	A 0xE8, 28	A 0xE8, 28	A 0x1B8, 24	
Mailbox1 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 25 A 0x1B8, 25	N/A
Mailbox2 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 26 A 0x1B8, 26	N/A
Mailbox3 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 27 A 0x1B8, 27	N/A
Mailbox4	N/A	N/A	E 0x1BC, 28 A 0x1B8, 28	N/A
Mailbox5	N/A	N/A	E 0x1BC, 29 A 0x1B8, 29	N/A
Mailbox6	N/A	N/A	E 0x1BC, 30 A 0x1B8, 30	N/A
Mailbox7	N/A	N/A	E 0x1BC, 31 A 0x1B8, 31	N/A
IopDoorbell	E 0xE8, 17 A 0xE8, 20	E 0xE8, 17 A 0xE8, 20	E 0x1BC, 11 A 0x1B8, 11 C 0x1A0	N/A
PciDoorbell	E 0xE8, 9 A 0xE8, 13	E 0xE8, 9 A 0xE8, 13	E 0x1B4, 11 A 0x1B0, 11 C 0x1A4	N/A
SerialPort1	N/A	N/A	E 0x1BC, 4 A 0x1B8, 4	N/A
SerialPort2	N/A	N/A	E 0x1BC, 5 A 0x1B8, 5	N/A
BIST ***Note:*** *There is no enable or disable funtionality for BIST for PCI9054 and PCI9080*	A 0xE8, 23 *	A 0xE8, 23 *	E 0x1BC, 12 A 0x1B8, 12 C 0x30F, 6	N/A
PowerManagement	N/A	E 0xE8, 4 A 0xE8, 5 C 0xE8, 5	E 0x1BC, 13 A 0x1B8, 13 C 0x1B8, 13	N/A
PciMainInt	E 0xE8, 8	E 0xE8, 8	E 0x1B4, 0	E 0x4C, 6
IopToPciInt	E 0xE8, 11 A 0xE8, 15	E 0xE8, 11 A 0xE8, 15	E 0x1B4, 12 A 0x1B0, 12	E 0x4C, 0 + 8 A 0x4C, 2 C 0x4c, 10
IopToPciInt_2	N/A	N/A	N/A	E 0x4C, 3 + 9

Structure Element	PCI 9080	PCI 9054	IOP 480	PCI 9030
				A 0x4C, 5 C 0x4C, 11
IopMainInt ***Note:*** *There is no master bit that shows that the IOP interrupt is active. Each interrupt trigger's active bit must be checked to determine the interrupt source.*	E 0xE8, 16	E 0xE8, 11	E 0x1BC, 0	N/A
PciAbort	E 0xE8, 10 A 0xE8, 14	E 0xE8, 10 A 0xE8, 14	E 0x1B4, 13 A 0x1B0, 13	N/A
PciReset	N/A	N/A	N/A	N/A
PciPME	N/A	E 0xE8, 4 A 0xE8, 5	E 0x1BC, 15 A 0x1B8, 15 C 0x1B8, 15	N/A
Enum	N/A	N/A	E 0x356, 1 A 0x356, 1	E 0x4A, 1 A 0x4A, 6-7
PciENUM	N/A	N/A	E 0x1BC, 16 A 0x1B8, 16 C 0x1B8, 16	N/A
IopBusTimout	N/A	N/A	E 0x1BC, 3 A 0x1B8, 3 C 0x1B8, 3	N/A
AbortLSERR	E 0xE8, 0 A 0x06, 12-13	E 0xE8, 0 A 0x06, 12-13	E 0x1BC, 1 A 0x1B8, 1 C 0x306, 12-13	N/A
ParityLSERR	E 0xE8, 1 A 0x06, 15	E 0xE8, 1 A 0x06, 15	E 0x1BC, 2 A 0x1B8, 2 C 0x306, 15	N/A
RetryAbort	E 0xE8, 12 A 0x06, 12-13	E 0xE8, 12 A 0x06, 12-13	N/A	N/A
LocalParityLSERR	N/A	E 0xE8, 6 A 0xE8, 7	E 0x1BC, 6 A 0x1B8, 6 C 0x1B8, 6	N/A
PciSERR	N/A	N/A	E 0x1BC, 20 A 0x1B8, 20 C 0x1B8, 20	N/A
IopRefresh	N/A	N/A	E 0x1BC, 21 A 0x1B8, 21 C 0x1B8, 21	N/A
PciINTApin	N/A	N/A	E 0x1BC, 14 A 0x1B8, 14 C 0x1B8, 14	N/A
IopINTIpin	N/A	N/A	E 0x1BC, 7	N/A

Structure Element	PCI 9080	PCI 9054	IOP 480	PCI 9030
			A 0x1BC. 7 C 0x1BC, 7	
TargetAbort	N/A	N/A	A 0x1B0, 19 C 0x204, 3	N/A
Ch1Abort	N/A	N/A	A 0x1B0, 18 C 0x224, 3	N/A
Ch0Abort	N/A	N/A	A 0x1B0, 17 C 0x204, 3	N/A
DMAbort	N/A	N/A	A 0x1B0, 16	N/A

Purpose

Structure containing the various PLX device interrupts that are used to return active interrupts or to enable or select certain interrupts.

Members

InboundPost
The value represents the messaging unit's inbound post FIFO interrupt.

OutboundPost
The value represents the messaging unit's outbound post FIFO interrupt.

OutboundOverflow
The value represents the messaging unit's outbound FIFO overflow interrupt.

OutboundOption
The value represents the Outbound Option to set the Outbound Post Queue interrupt.

IopDmaChannel0, IopDmaChannel1 and IopDmaChannel2
The value represents DMA channel interrupt on the IOP side.

PciDmaChannel0, PciDmaChannel1 and PciDmaChannel2
The value represents DMA channel interrupt on the PCI side.

Mailbox0, Mailbox1, Mailbox2, Mailbox3, Mailbox4, Mailbox5, Mailbox6, and Mailbox7
The value represents mailbox interrupt.

IopDoorbell
The value represents the PCI to IOP doorbell interrupt.

PciDoorbell
The value represents the IOP to PCI doorbell interrupt.

SerialPort1
The value represents the serial port 1 interrupt.

SerialPort2
The value represents the serial port 2 interrupt.

BIST
The value represents the BIST interrupt.

PowerManagement
The value represents the power management interrupt.

PciMainInt
The value represents the INTA interrupt line, which is the master interrupt for all PCI interrupts.

IopToPciInt
The value represents the INTI interrupt line, which is an input line for the IOP to trigger PCI interrupts.

IopToPciInt_2
The value represents the INT2 interrupt line, which is an input line for the IOP to trigger PCI interrupts.

IopMainInt
The value represents the INTo interrupt line which is the master interrupt for all IOP interrupts.

PciAbort
The value represents the PCI abort interrupt.

PciReset
The value represents the PCI reset interrupt.

PciPME
The value represents the PCI PME interrupt.

Enum
The value represents the ENUM# interrupt Mask.

PciENUM
The value represents the PCI ENUM interrupt. Enable this member allows local interrupt to generate when the PCI ENUM# pin is asserted.

IopBusTimeout
The value represents the IOP bus timeout interrupt.

AbortLSERR
The value represents the IOP LSERR interrupt caused by PCI bus Target Aborts or by Master Aborts.

ParityLSERR
The value represents the IOP LSERR interrupt caused by PCI bus Target Aborts or by Master Aborts.

RetryAbort
The value enables the PLX chip to generate a Target Abort after 256 Master consecutive retries to the target.

LocalParityLSERR
The value represents the IOP LSERR interrupt caused by Direct Master Local Data Parity Check Errors.

PciSERR
The value represents a local interrupt caused by PCI SERR# pin.

PciINTApin
Value of 1 enables a Local interrupt to generate when the PCI INTA# pin is asserted.

IopINTIpin
Value of 1 enables an interrupt to generate if the INTI pin is asserted.

TargetAbort
Value of 1 indicates a Target Abort was generated by the IOP 480 after 256 consecutive Master Retries to a Target.

Ch0Abort
DMA Channel 0 Master or Target Abort Detected. Value of 1 indicates DMA Channel 0 was the Bus Master during a Master or Target abort.

Ch1Abort
DMA Channel 1 – Master or Target Abort Detected. Value of 1 indicates DMA Channel 1 was the Bus Master during a Master or Target abort.

DMAbort
Direct Master – Master or Target Abort Detected. Value of 1 indicates a Direct Master was the Bus Master during a Master or Target abort.

Reserved
This value is reserved for future definitions.

Power Level Enum Data Type

```
typedef enum _PLX_POWER_LEVEL
{
  D0Uninitialized,
  D0,
  D1,
  D2,
  D3Hot,
  D3Cold
} PLX_POWER_LEVEL, *PPLX_POWER_LEVEL;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to state the power level.

Members

D0Uninitialized

Change to uninitialized state. This state is the full power state and is the state for normal operation.

D0

Change to active state. This state is the full power state and is the state for normal operation.

D1

Change to light sleep state. This state will allow the IOP to only PCI configuration accesses. Responses to other accesses are disabled. Other background tasks on the IOP may still be running such as monitoring a network. Function context should be maintained to support transitions back to the *D0* state.

D2

Change to deeper sleep state. This state will allow the IOP to only PCI configuration accesses. Responses to other accesses are disabled. This state should consume less power than *D1* state. Function context should be maintained to support transitions back to the *D0* state.

D3Hot

Change to power down state. This state prepares the IOP for a hot swap. Configuration space accesses must provide responses as long as the power and clock are supplied so that the software can change the power level state back to *D0* state. This state should consume very little power (while still connected to the PCI bus).

D3Cold

Change to no power state. This states only supports bus reset. All context is lost in this state.

Power Management Properties Structure

```
typedef struct _PM_PROP
{
  U32 Version                  :3;
  U32 PMEClockNeeded           :1;
  U32 DeviceSpecialInit        :1;
  U32 D1Supported              :1;
  U32 D2Supported              :1;
  U32 AssertPMEfromD0          :1;
  U32 AssertPMEfromD1          :1;
  U32 AssertPMEfromD2          :1;
  U32 AssertPMEfromD3Hot       :1;
  U32 Read_Set_State           :2;
  U32 PME_Enable               :1;
  U32 PME_Status               :1;
  U32 PowerDataSelect          :3;
  U32 PowerDataScale           :2;
  U32 PowerDataValue           :8;
  U32 Reserved                 :4;
} PM_PROP, *PPM_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
Version	N/A	0x182, 0-2	0x342, 0-2
PMEClockNeeded	N/A	0x182, 3	0x342, 3
DeviceSpecialInit	N/A	0x182, 5	0x342, 5
D1Supported	N/A	0x182, 9	0x342, 9
D2Supported	N/A	0x182, 10	0x342, 10
AssertPMEfromD0	N/A	0x182, 11	0x342, 11
AssertPMEfromD1	N/A	0x182, 12	0x342, 12
AssertPMEfromD2	N/A	0x182, 13	0x342, 13
AssertPMEfromD3Hot	N/A	0x182, 14	0x342, 14
Read_Set_State	N/A	0x184, 0-1	0x344, 0-1
PME_Enable	N/A	0x184, 8	0x344, 8
PME_Status	N/A	0x184, 15	0x344, 15
PowerDataSelect	N/A	0x184, 9-12	0x344, 9-12
PowerDataScale	N/A	0x184, 13-14	0x344, 13-14
PowerDataValue	N/A	0x187, 0-7	0x347, 0-7

Purpose

Structure used to describe the Power Management characteristics.

Members

Version
This value represents the version of the *PCI Power Management Interface Specification*.

PMEClockNeeded
When this value is set to 1, the PLX chip indicates that the device relies on the presence of PCI clock for PME# operation.

DeviceSpecialInit
When set to 1, the PLX chip requires special initializatiion following a transition to D_0 uninitialized state.

D1Supported
When set to 1, the PLX chip supports the D_1 power state.

D2Supported
When set to 1, the PLX chip supports the D_2 power state.

AssertPMEfromD0
When set to 1, PME# can be asserted from power state D_0.

AssertPMEfromD1
When set to 1, PME# can be asserted from power state D_1.

AssertPMEfromD2
When set to 1, PME# can be asserted from power state D_2.

AssertPMEfromD3Hot
When set to 1, PME# can be asserted from power state D_{3hot}.

Read_Set_State
Power state of the PLX chip.

PME_Enable
When set to 1, PME# can be asserted.

PME_Status
Indicate the status of PME#. When set to 1 and *PME_Enable* is 1, PME# is asserted.

PowerDataSelect
This value selects which data the PLX chip will report through the Power Management Data Register.

PowerDataScale
This value sets the scaling factor to use when interpreting the value of the Power Management Data Register.

PowerDataValue
This value represents the power consumed or dissipated in various power states. The exact meaning of this value is selected by the *PowerDataSelect* field and is scaled by the *PowerDataScale* field.

Reserved
This value is reserved for future definitions.

Serial Port Descriptor Structure

```
typedef struct _SPU_DESC
{
  U32 BaudRate;
  U32 LclkFreq;
  U32 LM                :2;
  U32 DTR               :1;
  U32 RTS               :1;
  U32 DB                :1;
  U32 PE                :1;
  U32 PTY               :1;
  U32 SB                :1;
  U32 Reserved          :24
}SPU_DESC, *PSPU_DESC;
```

Affected Register Location (offset from SPU base address)

Structure Element	IOP 480
BaudRate	0x10, 0x14
LM	0x18, 0-1
DTR	0x18, 2
RTS	0x18, 3
DB	0x18, 4
PE	0x18, 5
PTY	0x18, 6
SB	0x18, 7

Purpose

Data type used to initialize the Serial Port Control.

Members

BaudRate

Specify the baud rate at which the Serial Port Unit (SPU) operates. They should be one of the following value, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200.

LclkFreq

Specify the external system clock (LCLK) frequency in Hz. The LCLK is the Baud Rate Generator Clock input.

LM

Loopback Modes.

00 - Normal mode

01 - Internal Loopback mode

10 - Automatic Echo mode

11 - Reserved

DTR

Data Terminal

0 - DTR signal is inactive

1 - DTR signal is active

RTS

Request to Send.

0 - RTS signal is inactive

1 - RTS signal is active

DB

Data Bits.

0 - 7 Data bits

1 - 8 Data bits

PE

Parity Enable

PTY

Parity.

0 - Even parity

1 - Odd parity

SB

Stop bits.

0 - One stop bit

1 - Two stop bits

SPU Status Structure

```
typedef struct _SPU_STATUS
{
  U32 DSRInputInactive           :1;
  U32 CTSInputInactive           :1;
  U32 ReceiveBufferReady         :1;
  U32 FramingError               :1;
  U32 OverrunError               :1;
  U32 ParityError                :1;
  U32 LineBreak                  :1;
  U32 TransmitBufferReady        :1;
  U32 TransmitterShiftRegReady   :1;
  U32 Reserve                    :23
}SPU_STATUS, *PSPU_STATUS;
```

Affected Register Location (offset from SPU base address)

Structure Element	IOP 480
DSRInputInactive	0x08, 0
CTSInputInactive	0x08, 1
ReceiveBufferReady	0x00, 0
FramingError	0x00, 1
OverrunError	0x00, 2
ParityError	0x00, 3
LineBreak	0x00, 4
TransmitBufferReady	0x00, 5
TransmitterShiftRegReady	0x00, 6

Purpose

Data type used to describe the SPU status.

Members

DSRInputInactive

0 - DSR input is active

1 - DSR input has gone inactive

CTSInputInactive

0 - CTS input is active

1 - CTS input has gone inactive

ReceiveBufferReady

0 - Receive buffer is not full

1 - Receive Buffer is full

FramingError

0 - No framing error detected

1 - Framing error detected

ParityError

0 - No parity error detected

1 - Parity error detected

LineBreak

0 - No line break detected

1 - Line break detected

TransmitBufferReady

0 - Transmit Buffer is full (not ready)

1 - Transmit buffer is empty and ready

TransmitterShiftRegReady

0 - Transmitter Shift Register is full

1 - Transmitter Shift Register is empty

Comments

This data type is used to set or get Serial Port Handshake Status and Line Status.

USER Pin Direction Enum Data Type

```
typedef enum _PIN_DIRECTION
{
  INPUT,
  OUTPUT
} PIN_DIRECTION;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
INPUT	N/A	N/A	0x84, 2, 5, 9
OUTPUT	N/A	N/A	0x84, 2, 5, 9

Purpose

Enumerated type used to select a USER pin direction to be input or output.

Members

INPUT
Select direction to be input

OUTPUT
Select direction to be output

USER Pin Enum Data Type

```
typedef enum _USER_PIN
{
  USER0,
  USER1,
  USER2,
  USER3,
  USER4,
  USER5
} USER_PIN;
```

Affected Register Location

I represents input pins.

O represents output pins.

Structure Element	PCI 9080	PCI 9054	IOP 480
USER0	I 0xEC, 17 O 0xEC, 16	I 0xEC, 17 O 0xEC, 16	0x84, 0-3
USER1	N/A	N/A	0x84, 1 0x84, 4-6
USER2	N/A	N/A	0x84, 4 0x84, 8-10
USER3	N/A	N/A	0x84, 16
USER4	N/A	N/A	0x84, 17
USER5	N/A	N/A	0x84, 18

Purpose

Enumerated type used to select a USER pin.

Members

USER0 Select USER0 pin.

USER1 Select USER1 pin.

USER2 Select USER2 pin.

USER3 Select USER3 pin.

USER4 Select USER4 pin.

USER5 Select USER5 pin.

Comments

The USER pin enumerated type is used to select a USER pin.

Virtual Addresses Data Type

```
typedef struct _VIRTUAL_ADDRESSES
{
  U32 Va0;
  U32 Va1;
  U32 Va2;
  U32 Va3;
  U32 Va4;
  U32 Va5;
  U32 VaRom;
} VIRTUAL_ADDRESSES, *PVIRTUAL_ADDRESSES;
```

Purpose

This data type provides a list of User Virtual Addresses (UVA) for a PCI device. The UVAs correspond to the device's Base Address Registers (BAR)

Members

Va0
UVA for BAR 0.

Va1
UVA for BAR 1.

Va2
UVA for BAR 2.

Va3
UVA for BAR 3.

Va4
UVA for BAR 4.

Va5
UVA for BAR 5.

VaRom
UVA for the Expansion ROM.

Comments

This data type contains the UVAs for all the BARs of a PCI device.

MANAGEMENTDATA structure

```
typedef struct _MANAGEMENTDATA
{
    U32                 CfgRegNumber;
    U32                 Value;
    DEVICE_LOCATION     Device;
    VIRTUAL_ADDRESSES   VirtAddr;
    HANDLE              UserProcessId;
    PCI_MEMORY          PciMemory;
}MANAGEMENTDATA, *PMANAGEMENTDATA;
```

Affected Register Location

Not applicable.

Purpose

Used to group common data in a buffer to the driver.

Members

CfgRegNumber
The register offset in bytes of the Configuration Register;

Value
generic data variable;

Device
Contains PCI device information;

VirtAddr
User-mapped virtual addresses;

UserProcessId
to be removed (verify); and,

PciMemory
common buffer information;

Comments

Allows the management class of PCI API functions to communicate with the driver.

REGISTERDATA structure

```
typedef struct _REGISTERDATA
{
    RETURN_CODE  ReturnCode;
    U32          RegOffset;
    U32          Count;
    U32          Data;
    MAILBOX_ID   MailboxId;
} REGISTERDATA, *PREGISTERDATA;
```

Affected Register Location

Not applicable.

Purpose

Used to group common data in a buffer to the driver.

Members

ReturnCode
Copies back the driver's function status;

RegOffset
Register's offset;

Count
Used to specify number of registers to read or write;

Data
Stores register values;

MAILBOX_ID
Specifies the specific mailbox to access;

Comments

Allows the register class of PCI API functions to communicate with the driver.

BUSIOPDATA structure

```
typedef struct _BUSIOPDATA
{
    RETURN_CODE  ReturnCode;
    IOP_SPACE    IopSpace;
    U32          Address;
    BOOLEAN      Remap;
    U32          Data;
    U32          TransferSize;
    ACCESS_TYPE  AccessType;
    U8           ByteBuffer;
    U16          ShortBuffer;
    U32          LongBuffer;
}BUSIOPDATA, *PBUSIOPDATA;
```

Purpose

Used to group common data in a buffer to the driver.

Members

ReturnCode
Copies back the driver's function status;

IopSpace
Contains the type of Local Space to access, if applicable;

Address
Either the source or destination address;

Remap
Signifies whether the transfer was remapped or not;

Data
Stores full register values, or generic flags;

AccessType
The transfer size in bits;

ByteBuffer
Stores a byte value;

ShortBuffer
Stores a 16 bit value;

LongBuffer
Stores a 32 bit value;

Comments

Allows the Direct Slave and I/O access class of PCI API functions to communicate with the driver.

DMADATA structure

```
typedef struct _DMADATA
{
    RETURN_CODE             ReturnCode;
    U32                     TransferSize;
    DMA_TRANSFER_ELEMENT    TransBlock1;
    DMA_CHANNEL_DESC        DmaChannelDesc;
    DMA_CHANNEL             DmaChannel;
    DMA_COMMAND             DmaCommand;
    BOOLEAN                 ReturnImmediate;
}DMADATA, *PDMADATA;
```

Affected Register Location

Not applicable.

Purpose

Used to group common data in a buffer to the driver.

Members

ReturnCode
Copies back the driver's function status;

TransferSize
The number of bytes to transfer;

TransBlock1
DmaStructure that gives DMA information;

DmaChannelDesc
DmaChannelDesc provides all information needed to open a DMA channel;

DmaChannel
The DMA channel on which to perform the DMA operation;

DmaCommand
Contain one of various DMA commands;

ReturnImmediate
Determines whether this DMA call is synchronous or asynchronous;

Comments

Allows the DMA class of PCI API functions to communicate with the driver.

MISCDATA structure

```
typedef struct _MISCDATA
{
    PLX_INTR        IntrInfo;
    U32             Value1;
    U32             Value2;
    U32             Value;
    PLX_POWER_LEVEL PlxPowerLevel;
    EEPROM_TYPE     EepromType;
    USER_PIN        UserPinNum;
    PIN_STATE       PinState;
    RETURN_CODE     ReturnCode;
    HANDLE          EventHandle;
}MISCDATA, *PMISCDATA;
```

Purpose

Used to group common data in a buffer to the driver.

Members

IntrInfo
Contains interrupt information;

Value1
Generic 32 bit value;

Value2
Generic 32 bit value;

Value
Generic 64 bit value;

PlxPowerLevel
The power state which is read or to set;

EepromType
Enumerated type, which refers to the type of EEPROM present;

UserPinNum
The User Pin type to read or write;

ReturnCode
Copies back the driver's function status;

EventHandle
A handle to a Win32 event, used for asynchronous message passing;

Comments

Allows the miscellaneous class of PCI API functions to communicate with the driver.

IOCTLDATA union

```
typedef union _IOCTLDATA
{
    MANAGEMENTDATA MgmtData;
    REGISTERDATA   RegData;
    BUSIOPDATA     BusIopData;
    DMADATA        DmaData;
    MISCDATA       MiscData;
}IOCTLDATA, *PIOCTLDATA;
```

Affected Register Location

Not applicable.

Purpose

Used to union exclusive data in a buffer.

Members

MgmtData
Holds all the API management data;

RegData
Holds all the API register access data;

BusIopData
Holds all the API Direct Master and I/O access data;

DmaData
Holds all the API DMA data;

MiscData
Holds all the API Miscellaneous data;

Comments.

Keeps the user buffer small (not maximally so) and groups data into easy to manage sections.

Appendix A. PLX SDK Revision Notes

A.1 API Name Modification from PLX SDK Version 2.1

API Names at SDK version 3.0	API Names at SDK version 2.1	Page
PlxHotSwapIdRead()	PlxHotSwapId()	3-33
PlxHotSwapNcpRead()	PlxHotSwapNextItemPointer()	3-34
PlxVpdIdRead()	PlxVpdId()	3-104
PlxVpdNcpRead()	PlxVpdNextItemPointer()	3-105

A.2 API Parameter Changes from PLX SDK Version 2.1

Parameters used in SDK v3.x	Parameters used in SDK v 2.x	Page
PlxDmaBlockTransfer(DMA_CHANNEL, PDMA_TRANSFER_ELEMENT, BOOLEAN)	PlxDmaBlockTransfer(DMA_CHANNEL, DMA_COMMAND, PDMA_TRANSFER_ELEMENT, BOOLEAN)	3-12
PlxDmaSglTransfer(DMA_CHANNEL, SGL_ADDR, BOOLEAN)	PlxDmaSglTransfer(DMA_CHANNEL, DMA_COMMAND, SGL_ADDR, BOOLEAN)	3-23
PlxDmaShuttleTransfer(DMA_CHANNEL, U32, PDMA_TRANSFER_ELEMENT, BOOLEAN)	PlxDmaShuttleTransfer(DMA_CHANNEL, U32, DMA_COMMAND, PDMA_TRANSFER_ELEMENT)	3-29
PlxSdkVersion(U8*, U8*, U8*)	PlxSdkVersion(S8 **, S8 **)	3-94