PLX SDK

SOFTWARE DEVELOPMENT KIT

Programmer's Reference Manual

Version 3.2

March 2001

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-P1-3.2

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 © 2001 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

The PLX SDK included in the development package is a powerful aid to software designers. The PLX SDK is designed to provide customers with an easy path for developing custom applications.

Using the PLX Application Programming Interface (API), a user application can perform all necessary access and control of the PLX chips without the burden of register manipulation.

The SDK also includes PLXMon, a graphical debugging tool for accessing PLX chips and hardware devices.

The PLX SDK is available in two different packages. The first is SDK-LITE, which includes PLXMon, PLX Host API in a Windows environment, Windows device drivers, and sample host applications. The second offering is SDK PRO, which includes additional features not found in the SDK-LITE. These are an API for the I/O Platform (IOP) or Local environment, sample Local-side applications, Board Support Packages, and RTOS support.

1.1 About This Manual

This manual provides a reference for the Local and host side PLX APIs, and descriptions of API functionality and usage.

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, Local API, provides a detailed description of all Local-side API functions

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

Chapter 4, PLX SDK Data Structures Used by the API provides a description of the data structures used in the PLX SDK.

1.3 Terminology

- References to Windows NT assume Windows NT 4.0 and may be denoted as WinNT.
- References to Windows 98 may be denoted as Win98.
- References to Windows 2000 may be denoted as Win2k.
- References to Windows ME may be denoted as WinME.
- References to Visual C/C++ or Visual C++ refer to Microsoft Visual C/C++ 6.0.
- 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) or Local-side throughout this manual denote a Custom board with a PLX chip or a PLX RDK board. All references to IOP or Local software denote the software running on the board. References to the IOP 480 denote the PLX IOP 480 chip, which includes “IOP” as part of its name. This is a special case for use of IOP.

1.4 Customer Support

Prior to contacting PLX customer support, please be prepared to provide the following information:

1. PLX chip used
2. PLX SDK version (if applicable)
3. Host Operating System and version
4. Model number of the PLX RDK (if any)
5. Description of your intended design, including:
   - Local Microprocessor (if any)
   - Local Operating System and version (if any)
6. Detailed description of your problem
7. Steps to recreate the problem.

If you have comments, corrections, or suggestions, you may contact PLX Customer Support at:

Address: PLX Technology, Inc.
Attn. Technical Support
870 Maude Avenue
Sunnyvale, CA 94085

Phone: 408-774-9060

Fax: 408-774-2169

Web: http://www.plxtech.com

Email: PLX-Helpdesk@plxtech.com

2 Local API

The PLX Local API is designed for a local environment, in which a local CPU accesses the PLX chip directly through the local bus. The API is designed for a single-tasking environment with no locking mechanisms in place.

Note: The PLX SDK Version 3.2 Local API supports the following PLX Devices: 9054 and IOP 480.

2.1 Notes about the Local API and Multi-tasking Operating Systems

With the advent of Real-Time Operating Systems (RTOS), multi-tasking application are becoming more mainstream. As a consequence, the complexity of writing software is significantly increased. For those customers who intend to use the PLX software with an RTOS and use it with multiple simultaneous applications, some additional work and caution may be required.

At this time, the PLX API libraries are designed for a single-tasking environment. In other words, many functions are not re-entrant and there is no use of locking mechanisms for hardware access and access of internal global variables.

With SDK 3.2, complete source code is provided for the API libraries. With this in mind, customers are free to modify the code to insert locking mechanisms, provided by the OS, in the functions they plan to use. If performance is not a concern, the simplest solution is to use a single global Mutual Exclusion object (mutex) for the entire PLX API.

The issue of multi-tasking, however, is not so simple in the case of the PLX chip. This is because, for certain operations, the PLX chip must remain in a specific state until the operation has completed. The DMA engine is one example of this. The DMA registers cannot be changed while a DMA is in-progress or erratic behavior will occur. If a large transfer is in progress, say 2 MB, a second application must wait for a relatively long period of time before gaining access to the DMA channel. If real-time requirements are an issue, this situation must be resolved. For example, applications may be limited to small size transfers.

A future release of the PLX SDK may incorporate a generic set of locking mechanisms into the API functions. This will solve any simultaneous access issues, but customers may still need to customize some features if their requirements call for it.

2.2 Local API Function Quick Reference List

2.3 Local API Functions Details

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

The following is a sample entry demonstrating the information provided.

Sample Function Entry

Syntax:

```
function(
    Parameters,
    ...
    );
```

This gives the declaration syntax for each function.

PLX Chip Support:

A list of PLX chips that support this function.

Description:

Summary of the function's purpose

Parameters:

The function parameters

Return Codes:

The possible codes returned by the function.

Notes:

Provides any relevant information pertaining to the function.

Usage:

Sample source code is provided to demonstrate how the function is used. Note that the sample code has not been compiled or tested. It is provided for reference purposes only.

Cross Reference:

Provides page numbers to any relevant data structures or types.

PlxBusPciRead

Syntax:

```
RETURN_CODE
PlxBusPciRead(
    BUS_INDEX    busIndex,
    PCI_SPACE    PciSpace,
    ADDRESS      PciAddress,
    UDATA       *pDestination,
    U32          TransferSize,
    ACCESS_TYPE  AccessType
    );
```

PLX Chip Support:

9054, 480

Description:

Reads data from a specified PCI bus address into a user-supplied buffer (sometimes referred to as Direct Master Read).

Parameters:

busIndex
The bus index

PciSpace
Determines which space to access, memory or I/O

PciAddress
The PCI address to start reading from

pDestination
A pointer to a buffer, which will contain the data retrieved

TransferSize
The number of bytes to read

AccessType
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiInvalidPciSpace	The PCI_SPACE parameter is invalid
ApiInvalidAccessType	The ACCESS_TYPE parameter is invalid or unsupported
ApiInvalidAddress	The PCI address parameter is not aligned based on the ACCESS_TYPE
ApiInvalidSize	The transfer size parameter is 0 or not aligned based on the ACCESS_TYPE

Notes:

This function will adjust the Direct Master-to-PCI window, if necessary. The selected Direct Master space should be enabled and initialized before calling this function.

The destination buffer must be large enough to contain all of the data read.

Usage:

```
U8          buffer[0x100]
U32         PciAddress;
RETURN_CODE rc;


// Get the PLX driver reserved buffer PCI address, stored in Mailbox 3
PciAddress =
    PlxRegisterMailboxRead(
        PrimaryPciBus,
        MailBox3
        &rc
        );

// Read data from the PCI buffer
rc = PlxBusPciRead(
         PrimaryPciBus,
         PciMemSpace,
         PciAddress,
         (UDATA*)buffer,
         0x100,
         BitSize8
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read from PCI bus
}
```

Cross Reference

Referenced Item	Page
PCI_SPACE	4-52
ACCESS_TYPE	4-9
ADDRESS, UDATA	4-8

PlxBusPciWrite

Syntax:

```
RETURN_CODE
PlxBusPciWrite(
    BUS_INDEX    busIndex,
    PCI_SPACE    PciSpace,
    ADDRESS      PciAddress,
    UDATA       *pSource,
    U32          TransferSize,
    ACCESS_TYPE  AccessType
    );
```

PLX Chip Support:

9054, 480

Description:

Writes data from a user-supplied buffer to a specified PCI bus address (sometimes referred to as Direct Master Write).

Parameters:

busIndex
The bus index

PciSpace
Determines which space to access, memory or I/O

PciAddress
The PCI address to start reading from

pSource
A pointer to the buffer, which contains the data to write

TransferSize
The number of bytes to read

AccessType
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiInvalidPciSpace	The PCI_SPACE parameter is invalid
ApiInvalidAccessType	The ACCESS_TYPE parameter is invalid or unsupported
ApiInvalidAddress	The PCI address parameter is not aligned based on the ACCESS_TYPE
ApiInvalidSize	The transfer size parameter is 0 or not aligned based on the ACCESS_TYPE

Usage:

```
U8          buffer[0x100]
U32         PciAddress;
RETURN_CODE rc;


// Get the PLX driver reserved buffer PCI address, stored in Mailbox 3
PciAddress =
    PlxRegisterMailboxRead(
        PrimaryPciBus,
        MailBox3
        &rc
        );

// Write data to the PCI buffer
rc = PlxBusPciWrite(
         PrimaryPciBus,
         PciMemSpace,
         PciAddress,
         (UDATA*)buffer,
         0x100,
         BitSize8
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to the PCI bus
}
```

Cross Reference

Referenced Item	Page
PCI_SPACE	4-52
ACCESS_TYPE	4-9
ADDRESS, UDATA	4-8

PlxChipBaseAddressGet

Syntax:

```
ADDRESS
PlxChipBaseAddressGet(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

All

Description:

Queries the API for the Local base address of the PLX chip, which was initially provided by the BSP.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer to store the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid

Usage:

```
ADDRESS     PlxChipBase;
RETURN_CODE rc;


// Get the PLX chip address
PlxChipBase =
    PlxChipBaseAddressGet(
        PrimaryPciBus,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get PLX chip base address
}
```

Cross Reference:

Referenced Item	Page
ADDRESS	4-8

PlxChipTypeGet

Syntax:

```
RETURN_CODE
PlxChipTypeGet(
    BUS_INDEX  busIndex,
    U32       *pChipType,
    U8        *pRevision
    );
```

PLX Chip Support:

All

Description:

Returns the PLX chip type and its revision number

Parameters:

busIndex
The bus index

pChipType
A pointer to a 32-bit buffer to contain the PLX chip type

pRevision
A pointer to an 8-bit buffer to contain the PLX chip revision number

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiNullParam	One or more parameters is NULL

Notes:

The chip type is returned as a hex number matching the chip number. For examples, 0x9054 = 9054 and 0x0480 = IOP 480.

The Revision numbering begins at 1. For example, a 9054 AB returns a revision number of 2.

Usage:

```
U8          Revision;
U32         ChipType;
RETURN_CODE rc;


rc = PlxChipTypeGet(
         PrimaryPciBus
         &ChipType,
         &Revision
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get PLX chip type
}
else
{
    switch (ChipType)
    {
        case 0x9054:
            // Chip is a 9054
            if (Revision = 0x1)
                // Chip is an AA version
            if (Revision = 0x2)
                // Chip is an AB version
            break;

        case 0x480:
            // Chip is an IOP 480
            break
    }
}
```

PlxDmaBlockChannelClose

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelClose(
    DMA_CHANNEL channel
    );
```

PLX Chip Support:

9054, 480

Description:

Closes the Block DMA channel.

Parameters:

channel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for Block DMA.
ApiDmaInProgress	A DMA transfer is in progress
ApiDmaPaused	The DMA channel is paused

Notes:

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

Usage:

```
RETURN_CODE rc;


rc = PlxDmaBlockChannelClose(
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to close DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaBlockChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelOpen(
    DMA_CHANNEL       channel,
    DMA_CHANNEL_DESC *pDmaChannelDesc
    );
```

PLX Chip Support:

9054, 480

Description:

Opens and initializes a DMA channel for Block DMA transfers.

Parameters:

channel
The DMA channel to open

pDmaChannelDesc
A pointer to the structure containing the DMA channel descriptors

Notes:

The DMA Resource Manager should be initialized, with *PlxDmaResourceManagerInit()*, before using this function.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or parameters is NULL
ApiDmaChannelInvalid	The DMA channel parameter is not supported by this PLX chip
ApiDmaChannelUnavailable	The DMA channel is not closed
ApiDmaInvalidChannelPriority	The *DmaChannelPriority* member of DMA_CHANNEL_DESC is invalid
ApiFlybyNotSupported	The DMA channel does not support FlyBy

Usage:

```
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear the DMA descriptor
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

// Set up DMA configuration structure
DmaDesc.EnableReadyInput    = 1;
DmaDesc.DmaStopTransferMode = AssertBLAST;
DmaDesc.DmaChannelPriority  = Rotational;
DmaDesc.IopBusWidth         = 3;

rc = PlxDmaBlockChannelOpen(
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to open the DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-14

PlxDmaBlockTransfer

Syntax:

```
RETURN_CODE
PlxDmaBlockTransfer(
    DMA_CHANNEL           channel,
    DMA_TRANSFER_ELEMENT *pDmaData,
    BOOLEAN               ReturnImmediate
    );
```

PLX Chip Support:

9054, 480

Description:

Starts a Block DMA transfer for a given DMA channel.

Parameters:

channel
A previously opened and initialized DMA channel, which will be used for the transfer

pDmaDatal
A pointer to the data for the DMA transfer

ReturnImmediate
If R*eturnImmediate* is FALSE, the function waits until the DMA transfer has completed.

If R*eturnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for Block DMA
ApiDmaInProgress	A DMA transfer is currently in-progress

Notes:

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

Some members of the DMA_TRANSFER_ELEMENT structure differ in meanings depending upon the PLX chip and the mode of the DMA transfer.

DMA_TRANSFER_ELEMENT for DMA Channels 0 & 1:

Structure Element	Signification
LowPciAddr (or Loc1.LowPciAddr)	The lower 32-bits of the PCI address
HighPciAddr	The upper 32-bits of the PCI address for Dual-address cycles
IopAddr (or Loc2.IopAddr)	The Local 32-bit address for the transfer.
TransferCount	The number of bytes to transfer
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Enable/Disable the DMA done interrupt
IopToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored.

DMA_TRANSFER_ELEMENT for DMA Channel 2 (IOP 480 Local-to-Local):

Structure Element	Signification
Loc1.SourceAddr	The Local address to read from (source)
Loc2.DestAddr	The Local address to write to (destination)
TransferCount	The number of bytes to transfer
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Enable/Disable the DMA done interrupt
IopToPciDma	Ignored. Channel 2 is always a Local to Local transfer
NextSglPtr	Ignored

Usage:

```
U8                  buffer[0x100];
U32                 PciAddress;
RETURN_CODE         rc;
DMA_TRANSFER_ELEMENT DmaData;


// Get the PLX driver reserved buffer PCI address, stored in Mailbox 3
PciAddress =
    PlxRegisterMailboxRead(
        PrimaryPciBus,
        MailBox3
        &rc
        );
```

```
// Fill in DMA transfer parameters
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = buffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 1;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = buffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 1;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif

// Transfer the data using DMA
rc = PlxDmaBlockTransfer(
         PrimaryPciChannel0,
         &DmaData,
         FALSE          // Wait for DMA completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to perform DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_TRANSFER_ELEMENT	4-24

PlxDmaBlockTransferRestart

Syntax:

```
RETURN_CODE
PlxDmaBlockTransferRestart(
    DMA_CHANNEL channel,
    U32         TransferSize,
    BOOLEAN     ReturnImmediate
    );
```

PLX Chip Support:

9054, 480

Description:

Restarts a Block DMA transfer for a pre-programmed DMA channel. This function initiates a block DMA transfer just like *PlxDmaBlockTransfer()*, except it assumes the DMA transfer parameters are already setup, to provide a slight performance improvement.

Parameters:

channel
A previously opened and initialized DMA channel, which will be used for the transfer

TransferSize
Number of bytes to transfer

ReturnImmediate
If R*eturnImmediate* is FALSE, the function waits until the DMA transfer has completed.

If R*eturnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaChannelInvalid	The DMA channel parameter is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel has not been opened for Block DMA
ApiDmaInProgress	A DMA transfer is currently in-progress

Notes:

Before calling this function the appropriate DMA channel must have been successfully opened using *PlxDmaBlockChannelOpen()* and *PlxDmaBlockTransfer()* must have been successfully called to perform a DMA transfer using the same DMA channel.

Usage:

```
RETURN_CODE rc;


// A previous call to PlxDmaBlockTransfer() is assumed

rc = PlxDmaBlockTransferRestart(
         PrimaryPciChannel0,
         0x100,               // Provide transfer count
         TRUE                 // Don't wait for completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to restart Block DMA
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaControl

Syntax:

```
RETURN_CODE
PlxDmaControl(
    DMA_CHANNEL channel,
    U32         ElementIndex,
    DMA_COMMAND DmaCommand
    );
```

PLX Chip Support:

9054, 480

Description:

Controls the DMA engine for a given DMA channel.

Parameters:

channel
A previously opened DMA channel

ElementIndex
For Shuttle DMA transfers only, the index value of the desired DMA Transfer Element in a circular shuttle linked list. *(Ignored for SGL and Block DMA modes.)*

DmaCommand
The action to perform on the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaCommandInvalid	The DMA Command parameter is invalid
ApiDmaChannelClosed	The DMA channel is not currently open for DMA
ApiDmaChannelInvalid	The DMA channel parameter is not supported by this PLX chip
ApiDmaManCorrupted	The DMA manager is corrupted
ApiDmaNotPaused	If the DMA command is *Resume* and either a DMA transfer is in-progress or the DMA channel is idle (DMA channel complete/ready)
ApiDmaInProgress	If the DMA command is *Status* and a DMA transfer is currently in-progress
ApiDmaInvalidElementIndex	If the DMA command is *Abort* and the DMA Manager is unable to locate DMA element, by index, within the SGL
ApiDmaDone	If the DMA command is *Abort* or *Pause* and the DMA channel is idle (DMA channel complete/ready)
ApiDmaChannelError	If DMA channel 2 is selected and the DMA channel information is corrupted.

Notes:

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

The following describes the DMA commands:

DMA Command	Description
DmaPause	Pause a DMA transfer
DmaResume	Resume a paused DMA transfer
DmaAbort	Abort the current DMA transfer. For Shuttle DMA, this command will abort only the DMA transfer element with localed at the provided Element Index in the Shuttle linked list.

Usage:

```
RETURN_CODE          rc;
DMA_TRANSFER_ELEMENT DmaData;

// Start a DMA transfer
rc = PlxDmaBlockTransfer(
       PrimaryPciChannel0,
       &DmaData,
       FALSE             // Wait for DMA completion
       );

// Pause the DMA channel
rc = PlxDmaControl(
       PrimaryPciChannel0,
       DmaPause
       );

if (rc != ApiSuccess)
{
    // ERROR – Unable to pause DMA transfer
}

// Resume the DMA channel
rc = PlxDmaControl(
       PrimaryPciChannel0,
       DmaResume
       );

if (rc != ApiSuccess)
{
    // ERROR – Unable to resume DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_COMMAND	4-21

PlxDmaIsr

Syntax:

```
void
PlxDmaIsr(
    DMA_CHANNEL channel
    );
```

PLX Chip Support:

9054, 480

Description:

Services a DMA channel interrupt. If the DMA Resource Manager is used to manage the DMA resources then this function should be called by the BSP PLX interrupt service routine when a DMA interrupt occurs.

Parameters:

channel
The DMA channel which is the source of the interrupt

Return Codes:

None.

Usage:

```
void
InterruptHandler_PlxChip(
    void
    )
{
    U32         value;
    PLX_INTR    PlxIntr;
    RETURN_CODE rc;


    // Verify that the PLX chip is interrupting
    if (PlxIntrStatusGet(
            PrimaryPciBus,
            &PlxIntr,
            NULL
            ) == FALSE)
    {
        return;
    }
```

```
    /* DMA channel 0 interrupt */
    if (PlxIntr.IopDmaChannel0)
    {
        PlxDmaIsr(
            PrimaryPciChannel0
            );
    }

    /* DMA channel 1 interrupt */
    if (PlxIntr.IopDmaChannel1)
    {
        PlxDmaIsr(
            PrimaryPciChannel1
            );
    }

    /* Mailbox 0 interrupt */
    ....
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaResourceManagerInit

Syntax:

```
RETURN_CODE
PlxDmaResourceManagerInit(
    DMA_PARMS *pDmaParms
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the DMA Resource Manager.

Parameters:

pDmaParms
A pointer to an array of structures containing the DMA channel queue and memory for the SGL elements list for the DMA Resource Manager.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiDmaChannelTypeError	A DMA channel for which this function is called is not closed
ApiDmaManReady	The DMA manager is already initialized for the channel
ApiInvalidAddress	The SGL list base address of an element in the DMA_PARMS array is not aligned on a 16-byte boundary

Notes:

The DMA Resource Manager must be initialized before any DMA API functions are used.

If the DMA Resource Manager is used, it needs to perform some tasks after a DMA transfer has completed. This typically happens after a DMA completion interrupt occurs. *Refer to the function **PlxDmaIsr()** for more information.*

The DMA parameters array does not contain a size parameter. An invalid member of the array signals the end of the list.

The memory buffers provided for the *WaitQueueBase* and the *FirstSglElement* members of the DMA_PARMS structure should not be accessed after the function exits. These memory buffers are reserved for DMA Resource Manager operation.

When calling this function, the application should do the following:

1. Allocate an array of DMA_PARMS structures array containing an additional element.
2. Fill the array with the desired values.
3. Fill the last element (the extra one) of the array with one of the following:

- Set the *DmaChannel* member to –1
- Set the *FirstSglElement* member to NULL
- Set the *WaitQueueBase* member to NULL
- Set the *NumberOfElements* member to 0

Usage:

```
U32       tempAddress;
DMA_PARMS DmaParms[3];


/* Initialization for DMA channel 0 – aligned on 16-byte boundary*/
tempAddress = ( (U32) DmaElementsBuffer_0 + 0xF) & 0xFFFFFFF0;
DmaParms[0].DmaChannel       = PrimaryPciChannel0;
DmaParms[0].FirstSglElement  = (DMA_TRANSFER_ELEMENT *)tempAddress;
DmaParms[0].WaitQueueBase    = Dma0SglQueue;
DmaParms[0].NumberOfElements = DMA_CHANNEL0_SGL_SIZE;

/* Initialization for DMA channel 1 – aligned on 16-byte boundary*/
tempAddress = ( (U32) DmaElementsBuffer_1 + 0xF) & 0xFFFFFFF0;
DmaParms[1].DmaChannel       = PrimaryPciChannel1;
DmaParms[1].FirstSglElement  = (DMA_TRANSFER_ELEMENT *)tempAddress;
DmaParms[1].WaitQueueBase    = Dma1SglQueue;
DmaParms[1].NumberOfElements = DMA_CHANNEL1_SGL_SIZE;

/* Signal end of array */
DmaParms[2].DmaChannel       = (DMA_CHANNEL)(-1);
DmaParms[2].FirstSglElement  = NULL;
DmaParms[2].WaitQueueBase    = NULL;
DmaParms[2].NumberOfElements = 0;

// Initialize DMA Manager
rc = PlxDmaResourceManagerInit(
         DmaParms
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize DMA Manager
}
```

Cross Reference:

Referenced Item	Page
DMA_PARMS	4-23

PlxDmaSglBuild

Syntax:

```
RETURN_CODE
PlxDmaSglBuild(
    DMA_CHANNEL            channel,
    U32                    SglSize,
    DMA_TRANSFER_ELEMENT **pFirstElement
    );
```

PLX Chip Support:

9054, 480

Description:

Builds an empty Scatter-Gather List. The Scatter-Gather List is allocated from the DMA Manager scatter-gather list pool for the channel.

Parameters:

channel
The DMA channel to build the SGL list for.

SglSize
The desired number of DMA Transfer Elements for the Scatter-Gather List

pFirstElement
A pointer to a DMA_TRANSFER_ELEMENT pointer. This will contain the base address of the first SGL element, which is the start of the SGL list

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for SGL DMA
ApiDmaManNotReady	The DMA manager is not ready or initialized for this channel
ApiInvalidSize	The number of DMA elements is invalid
ApiDmaNoMoreElements	There are not enough elements in the SGL Free pool to build the SGL
ApiDmaManCorrupted	The DMA manager is corrupted
ApiNotSupportThisChannel	The DMA channel does not support SGL DMA

Notes:

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

Usage:

```
U8                    DestBuffer[0x100];
U8                    SourceBuffer[0x100];
U32                   PciAddress;
RETURN_CODE           rc;
DMA_TRANSFER_ELEMENT  DmaData;
DMA_TRANSFER_ELEMENT *SglList;


// Get a new SGL list with space for 2 elements
rc = PlxDmaSglBuild(
         PrimaryPciChannel0,
         2,
         &SglList
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get empty SGL list
}

// Fill in DMA transfer parameters for Local-to-PCI transfer
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = SourceBuffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 1;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = SourceBuffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 1;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif

// Insert DMA element in list
rc = PlxDmaSglFill(
         SglList,
         0,
         &DmaData
         );
```

```
if (rc != ApiSuccess)
{
    // ERROR – Unable to insert DMA element
}

// Fill in DMA transfer parameters for PCI-to-Local transfer
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = DestBuffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 0;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = DestBuffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 0;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif

// Insert DMA element in list
rc = PlxDmaSglFill(
         SglList,
         1,
         &DmaData
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to insert DMA element
}
```

```
// Perform the DMA transfers
rc = PlxDmaSglTransfer(
         PrimaryPciChannel0,
         SglList,
         FALSE           // Wait for DMA completion
         );

if (rc != ApiSuccess)
{
    ERROR – Unable to perform DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
SGL_ADDR	4-24

PlxDmaSglChannelClose

Syntax:

```
RETURN_CODE
PlxDmaSglChannelClose(
    DMA_CHANNEL channel
    );
```

PLX Chip Support:

9054, 480

Description:

Closes a DMA channel previously opened for Scatter-Gather List (SGL) mode.

Parameters:

channel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelTypeError	The DMA channel has not been opened for SGL DMA
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiDmaPaused	The DMA channel is in a paused state
ApiNotSupportThisChannel	The DMA channel given does not support SGL DMA

Notes:

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


rc = PlxDmaSglChannelClose(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to close DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaSglChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaSglChannelOpen(
    DMA_CHANNEL       channel,
    DMA_CHANNEL_DESC *pDmaChannelDesc
    );
```

PLX Chip Support:

9054, 480

Description:

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

Parameters:

channel
The DMA channel to open

pDmaChannelDesc
A pointer to the structure containing the parameters used for initializing the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaManNotReady	The DMA manager is not ready for this DMA channel
ApiDmaChannelUnavailable	The DMA channel is not closed
ApiNotSupportThisChannel	The DMA channel given does not support SGL DMA

Notes:

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

The DMA Resource Manager must be initialized before using this function.

Usage:

```
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear the DMA channel descriptor
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

#if defined(PCI9054)
    DmaDesc.EnableReadyInput    = 1;
    DmaDesc.EnableIopBurst      = 0;
    DmaDesc.DmaStopTransferMode = AssertBLAST;
    DmaDesc.DmaChannelPriority  = Rotational;
    DmaDesc.IopBusWidth         = 3;
#elif defined(PCI480)
    DmaDesc.DmaStopTransferMode = AssertBLAST;
    DmaDesc.DmaChannelPriority  = Rotational;
    DmaDesc.ValidModeEnable     = 1;
#endif

// Open the DMA channel
rc = PlxDmaSglChannelOpen(
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
     // ERROR – Unable to open DMA channel for SGL
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-14

PlxDmaSglFill

Syntax:

```
RETURN_CODE
PlxDmaSglFill(
    DMA_TRANSFER_ELEMENT *pFirstElement,
    U32                   index,
    DMA_TRANSFER_ELEMENT *pDmaData
    );
```

PLX Chip Support:

9054, 480

Description:

Inserts a DMA Transfer Element to an empty location of an existing Scatter-Gather List (SGL).

Parameters:

pFirstElement
A pointer to the first element of an SGL list

index
The index value of DMA Transfer Element, in the Scatter-Gather List, indicating the element to fill.

pDmaData
A pointer to a DMA transfer element structure containing the DMA transfer parameters

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiDmaInvalidElementIndex	The DMA Manager is unable to locate DMA element, by index, within the SGL

Notes:

Before calling this function a valid Scatter-Gather List must be obtained with *PlxDmaSglBuild()*.

Some members of the DMA_TRANSFER_ELEMENT structure differ in meanings depending upon the PLX chip and the mode of the DMA transfer.

DMA_TRANSFER_ELEMENT:

Structure Element	Signification
LowPciAddr (or Loc1.LowPciAddr)	The lower 32-bits of the PCI address
HighPciAddr	The upper 32-bits of the PCI address for Dual-address cycles
IopAddr (or Loc2.IopAddr)	The Local 32-bit address for the transfer.
TransferCount	The number of bytes to transfer
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Enable/Disable the DMA done interrupt for the specified DMA element.
IopToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored.

Usage:

```
U8                    buffer[0x100];
U32                   PciAddress;
RETURN_CODE           rc;
DMA_TRANSFER_ELEMENT  DmaData;
DMA_TRANSFER_ELEMENT *SglList;


// Get a new SGL list with space for 1 element
rc = PlxDmaSglBuild(
         PrimaryPciChannel0,
         1,
         &SglList
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get empty SGL list
}
```

```
// Fill in DMA transfer parameters
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = buffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 1;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = buffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 1;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif

// Insert DMA element in list
rc = PlxDmaSglFill(
         SglList,
         0,
         &DmaData
         );
```

Cross Reference:

Referenced Item	Page
SGL_ADDR	4-24
DMA_TRANSFER_ELEMENT	4-24

PlxDmaSglTransfer

Syntax:

```
RETURN_CODE PlxDmaSglTransfer(
    DMA_CHANNEL           channel,
    DMA_TRANSFER_ELEMENT *pFirstElement,
    BOOLEAN               ReturnImmediate
    );
```

PLX Chip Support:

9054, 480

Description:

Program and start an SGL DMA transfer on the specified DMA channel. The transfer is queued if a DMA transfer is currently in-progress.

Parameters:

channel
A previously opened and initialized DMA channel, which will be used for the transfer

pFirstElement
A pointer to the start of an SGL list to use for the DMA transfer

ReturnImmediate
If R*eturnImmediate* is FALSE, the function waits until the DMA transfer has completed.

If R*eturnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for SGL DMA
ApiDmaManNotReady	The DMA manager is not ready for this channel
ApiDmaSglInvalid	The SGL list parameter is invalid
ApiDmaSglQueueFull	The SGL pending queue for the channel is full
ApiNotSupportThisChannel	The DMA channel does not support SGL DMA transfers

Notes:

Before calling this function the specified DMA channel must have been successfully opened using *PlxDmaSglChannelOpen()*.

Once the SGL list is passed to *PlxDmaSglTransfer()*, the list should not be accessed, including no longer used with the *PlxDmaSglFill()* function.

When the DMA channel completes the transfer, an interrupt is generated and the DMA Resource Manager will return the elements in SGL list back to the SGL element free pool.

Usage:

```
U8                    DestBuffer[0x100];
U8                    SourceBuffer[0x100];
U32                   PciAddress;
RETURN_CODE           rc;
DMA_TRANSFER_ELEMENT  DmaData;
DMA_TRANSFER_ELEMENT *SglList;


// Get a new SGL list with space for 2 elements
rc = PlxDmaSglBuild(
         PrimaryPciChannel0,
         2,
         &SglList
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get empty SGL list
}

// Fill in DMA transfer parameters for Local-to-PCI transfer
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = SourceBuffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 1;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = SourceBuffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 1;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif
```

```
// Insert DMA element in list
rc = PlxDmaSglFill(
         SglList,
         0,
         &DmaData
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to insert DMA element
}

// Fill in DMA transfer parameters for PCI-to-Local transfer
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = DestBuffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 0;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = DestBuffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 0;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif

// Insert DMA element in list
rc = PlxDmaSglFill(
         SglList,
         1,
         &DmaData
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to insert DMA element
}
```

```
// Perform the DMA transfers
rc = PlxDmaSglTransfer(
         PrimaryPciChannel0,
         SglList,
         FALSE           // Wait for DMA completion
         );

if (rc != ApiSuccess)
{
    ERROR – Unable to perform DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_TRANSFER_ELEMENT	4-24
DMA_CHANNEL	4-19

PlxDmaShuttleChannelClose

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelClose(
    DMA_CHANNEL channel
    );
```

PLX Chip Support:

9054, 480

Description:

Closes a DMA channel previously opened for Shuttle mode.

Parameters:

channel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelTypeError	The DMA channel has not been opened for Shuttle DMA
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiDmaPaused	The DMA channel is in a paused state
ApiNotSupportThisChannel	The DMA channel given does not support Shuttle DMA

Notes:

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


rc = PlxDmaShuttleChannelClose(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to close DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaShuttleChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelOpen(
    DMA_CHANNEL       channel,
    DMA_CHANNEL_DESC *pDmaChannelDesc
    );
```

PLX Chip Support:

9054, 480

Description:

Opens and initializes a DMA channel for Shuttle mode.

Parameters:

channel
The DMA channel to open

pDmaChannelDesc
A pointer to the structure containing the parameters used for initializing the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaManNotReady	The DMA manager is not ready for this DMA channel
ApiDmaChannelUnavailable	The DMA channel is not closed
ApiNotSupportThisChannel	The DMA channel given does not support Shuttle DMA

Notes:

The DMA Resource Manager must be initialized before using this function.

Usage:

```
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear the DMA channel descriptor
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

#if defined(PCI9054)
    DmaDesc.EnableReadyInput    = 1;
    DmaDesc.EnableIopBurst      = 0;
    DmaDesc.DmaStopTransferMode = AssertBLAST;
    DmaDesc.DmaChannelPriority  = Rotational;
    DmaDesc.IopBusWidth         = 3;
#elif defined(PCI480)
    DmaDesc.DmaStopTransferMode = AssertBLAST;
    DmaDesc.DmaChannelPriority  = Rotational;
    DmaDesc.ValidModeEnable     = 1;
#endif

// Open the DMA channel
rc = PlxDmaShuttleChannelOpen(
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
     // ERROR – Unable to open DMA channel for Shuttle
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-14

PlxDmaShuttleTransfer

Syntax:

```
RETURN_CODE
PlxDmaShuttleTransfer(
    DMA_CHANNEL           channel,
    U32                   ElementIndex,
    DMA_TRANSFER_ELEMENT *pDmaData,
    BOOLEAN               ReturnImmediate
    );
```

PLX Chip Support:

9054, 480

Description:

Starts a Shuttle DMA transfer on a specified DMA channel.

Parameters:

channel
A previously opened and initialized DMA channel, which will be used for the transfer

ElementIndex
The index of the desired DMA transfer element

pDmaData
A pointer to a DMA transfer element specifying the transfer parameters

ReturnImmediate
If R*eturnImmediate* is FALSE, the function waits until the DMA transfer has completed.

If R*eturnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or parameters is NULL
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for Shuttle DMA
ApiDmaInvalidElementIndex	The Element index is invalid or out of range
ApiDmaInProgress	A DMA transfer is currenty in-progress on the channel
ApiNotSupportThisChannel	The DMA channel does not support SGL DMA transfers

Notes:

Before calling this function the specified DMA channel must have been successfully opened using *PlxDmaShuttleChannelOpen()*.

The maximum number for the *ElementIndex* parameter is one less the total number of elements for which the DMA channel was initialized with *PlxDmaResourceManagerInit()*.

Some members of the DMA_TRANSFER_ELEMENT structure differ in meanings depending upon the PLX chip and the mode of the DMA transfer.

DMA_TRANSFER_ELEMENT:

Structure Element	Signification
LowPciAddr (or Loc1.LowPciAddr)	The lower 32-bits of the PCI address
HighPciAddr	The upper 32-bits of the PCI address for Dual-address cycles
IopAddr (or Loc2.IopAddr)	The Local 32-bit address for the transfer
TransferCount	The number of bytes to transfer
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Enable/Disable the DMA done interrupt for the specified DMA element.
IopToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored.

Usage:

```
U8                  buffer[0x100];
RETURN_CODE         rc;
DMA_TRANSFER_ELEMENT DmaData;


// Fill in DMA transfer parameters
#if defined(PCI9054)
    DmaData.Pci9054Dma.LowPciAddr        = PciAddress;
    DmaData.Pci9054Dma.HighPciAddr       = 0x0;
    DmaData.Pci9054Dma.IopAddr           = buffer;
    DmaData.Pci9054Dma.TransferCount     = 0x100;
    DmaData.Pci9054Dma.IopToPciDma       = 1;
    DmaData.Pci9054Dma.TerminalCountIntr = 0;
#elif defined(PCI480)
    DmaData.Iop480Dma.LowPciAddr        = PciAddress;
    DmaData.Iop480Dma.HighPciAddr       = 0x0;
    DmaData.Iop480Dma.IopAddr           = buffer;
    DmaData.Iop480Dma.TransferCount     = 0x100;
    DmaData.Iop480Dma.IopToPciDma       = 1;
    DmaData.Iop480Dma.TerminalCountIntr = 0;
#endif
```

```
// Start the Shuttle DMA transfer
rc = PlxDmaShuttleTransfer(
         PrimaryPciChannel0,
         0,                  // First element
         &DmaData,
         FALSE               // Wait for completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to perform DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_TRANSFER_ELEMENT	4-24

PlxDmaShuttleTransferRestart

Syntax:

```
RETURN_CODE
PlxDmaShuttleTransferRestart(
    DMA_CHANNEL channel,
    U32         ElementIndex,
    U32         TransferSize,
    BOOLEAN     ReturnImmediate
    );
```

PLX Chip Support:

9054, 480

Description:

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

Parameters:

channel
A previously opened and initialized DMA channel, which will be used for the transfer

ElementIndex
The index of the desired DMA transfer element, which was previously programmed

TransferSize
Number of bytes to transfer

ReturnImmediate
If R*eturnImmediate* is FALSE, the function waits until the DMA transfer has completed.
If R*eturnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelTypeError	The DMA channel was not opened for Shuttle DMA
ApiDmaInvalidElementIndex	The Element index is invalid or out of range
ApiDmaInProgress	A DMA transfer is currenty in-progress on the channel
ApiNotSupportThisChannel	The DMA channel does not support SGL DMA transfers

Notes:

Before calling this function the specified DMA channel must have been successfully opened using *PlxDmaShuttleChannelOpen()* and *PlxDmaShuttleTransfer()* must have been successfully called to perform a DMA transfer using the same DMA channel and element index.

The maximum number for the *ElementIndex* parameter is one less the total number of elements for which the DMA channel was initialized with *PlxDmaResourceManagerInit()*.

In Shuttle mode DMA, when a DMA transfer, specified by an element in the Shuttle mode list, has completed, the transfer size in the element is set to 0.

Usage:

```
RETURN_CODE rc;


// A previous call to PlxDmaShuttleTransfer() is assumed

rc = PlxDmaShuttleTransferRestart(
         PrimaryPciChannel0,
         1,                    // Second element in list
         0x100,                // Provide transfer count
         TRUE                  // Don't wait for completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to restart Shuttle DMA
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaStatus

Syntax:

```
RETURN_CODE
PlxDmaStatus(
    DMA_CHANNEL channel,
    U32         ElementIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the status of the DMA engine for a specified DMA channel.

Parameters:

channel
A previously opened DMA channel

ElementIndex
The index of the desired DMA transfer element to get status of in a Shuttle DMA element list. This parameter is ignored for DMA channels opened for Block and SGL mode.

Return Codes:

Code	Description
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelClosed	The DMA channel is closed
ApiDmaDone	The DMA channel is done/ready
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiDmaPaused	The DMA channel is paused
ApiDmaInvalidElementIndex	The Element index is invalid or out of range
ApiDmaChannelError	Unable to obtain status information for the DMA channel.

Notes:

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

Usage:

```
RETURN_CODE rc;

// Start a DMA transfer
rc = PlxDmaShuttleTransfer(
         PrimaryPciChannel0,
         0,
         &DmaData,
         TRUE
         );

// Get DMA status
rc = PlxDmaStatus(
         PrimaryPciChannel0,
         0
         );

if (rc != ApiDmaInProgress)
{
    // ERROR - DmaInProgress not returned
}

// Pause the DMA channel
rc = PlxDmaControl(
         PrimaryPciChannel0,
         0,
         DmaPause
         );

// Get DMA status
rc = PlxDmaStatus(
         PrimaryPciChannel0
         0
         );

if (rc != ApiDmaPaused)
{
    // ERROR - DmaPaused not returned
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxHotSwapDisable

Syntax:

```
RETURN_CODE
PlxHotSwapDisable(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Disables the Hot Swap Capability feature of the PLX chip

Parameters:

busIndex
The bus index

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Notes:

This function disables the Hot Swap feature by removing it from the linked list of New Capabilities in the PCI registers.

Usage:

```
RETURN_CODE rc;


rc = PlxHotSwapDisable(
         PrimaryPciBus
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to disable Hot Swap
}
```

PlxHotSwapEnable

Syntax:

```
RETURN_CODE
PlxHotSwapEnable(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Enables the Hot Swap Capability feature of the PLX chip

Parameters:

busIndex
The bus index

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Notes:

This function enables the Hot Swap feature by adding it to the linked list of New Capabilities in the PCI registers.

Usage:

```
RETURN_CODE rc;


rc = PlxHotSwapEnable(
         PrimaryPciBus
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to enable Hot Swap
}
```

PlxHotSwapIdRead

Syntax:

```
U8
PlxHotSwapIdRead(
    BUS_INDEX    busIndex
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the Hot Swap Capability ID

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Notes:

A value of –1 is returned if there is an error reading the Hot Swap ID.

Usage:

```
    U8          HotSwapId;
    RETURN_CODE rc;


    HotSwapId = PlxHotSwapIdRead(
                    PrimaryPciBus,
                    &rc
                    );

    if (rc != ApiSuccess)
        // ERROR - Unable to read Hot Swap Capability ID
```

PlxHotSwapIdWrite

Syntax:

```
RETURN_CODE
PlxHotSwapIdWrite(
    BUS_INDEX busIndex
    U8        Id
    );
```

PLX Chip Support:

9054, 480

Description:

Write to the Hot Swap Capability ID.

Parameters:

busIndex
The bus index

Id
The value to write to the Hot Swap Capability ID

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Usage:

```
RETURN_CODE rc;


rc = PlxHotSwapIdWrite(
         PrimaryPciBus,
         0x0
         );

if (rc != ApiSuccess )
{
    // ERROR – Unable to write Hot Swap Capability ID
}
```

PlxHotSwapNcpRead

Syntax:

```
U8
PlxHotSwapNcpRead(
    BUS_INDEX    busIndex
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the Next Capability Pointer from the Hot Swap register.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Notes:

A value of –1 is returned if there is an error reading the Hot Swap ID.

Usage:

```
    U8          NextCapability;
    RETURN_CODE rc;


    NextCapability = PlxHotSwapNcpRead(
                         PrimaryPciBus,
                         &rc
                         );

    if (rc != ApiSuccess)
        // ERROR - Unable to read Hot Swap NCP
```

PlxHotSwapNcpWrite

Syntax:

```
RETURN_CODE
PlxHotSwapNcpWrite(
    BUS_INDEX busIndex,
    U8        value
    );
```

PLX Chip Support:

9054, 480

Description:

Writes a value to the Next Capability Pointer of the Hot Swap register.

Parameters:

busIndex
The bus index

value
The value to write to the Hot Swap Next Capability Pointer

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Usage:

```
RETURN_CODE rc;


rc = PlxHotSwapNcpWrite(
         PrimaryPciBus,
         0x4c
         );

if (rc != ApiSuccess )
{
    // ERROR – Unable to write Hot Swap Capability NCP
}
```

PlxHotSwapStatus

Syntax:

```
U8
PlxHotSwapStatus(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the Hot-Swap status of the PLX chip.

Parameters:

busIndex
The bus index

Return Codes:

If there is no error, the return value is an OR'ed combination of the following:

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 status;


status = PlxHotSwapStatus(
             PrimaryPciBus,
             );

if (status = 0xff)
{
    // ERROR: Unable to read Hot Swap status
}

if (status & HS_LED_ON)
{
    // Hot Swap LED is on
}

if (status & HS_BOARD_REMOVED)
{
    // Hot Swap – board requesting extraction
}

if (status & HS_BOARD_INSERTED)
{
    // Hot Swap – board requesting insertion
}
```

Cross Reference:

Referenced Item	Page
Hot Swap Status definition	4-30

PlxInitApi

Syntax:

```
RETURN_CODE
PlxInitApi(
    API_PARMS *ApiParms
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the passed in parameters to the PLX chip's default values, as specified in the PLX chip's data book.

Parameters:

ApiParms

A pointer to the structure containing information passed from the BSP and storage to contain PLX chip default value for the structures specified.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiAlreadyInitialized	The Local API has already been initialized
ApiInvalidAddress	The PLX chip local Base Address provided is not aligned on a 4-byte boundary

Notes:

This function must be called before any other Local API functions are used.

Usage:

```
API_PARMS       ApiInit;
RETURN_CODE     rc;
PCI_BUS_PROP    PciBusProp;
IOP_BUS_PROP    IopBus0Prop;
IOP_BUS_PROP    IopBus1Prop;
IOP_ARBIT_DESC  IopArbitDesc;
IOP_ENDIAN_DESC IopEndianDesc;
```

```
// Clear API Initialization Structure (set all to NULL)
memset(
    &ApiInit,
    0,
    sizeof(API_PARMS)
    );

/*  The PlxInitApi function will initialize the following structures
    with the default bit field settings corresponding to the PLX Chip's
    reset values.
 */

// Only initialize the data structure members needed
ApiInit.PlxIcIopBaseAddr = (void *)ADDR_PLX_CHIP; // Address of PLX chip
ApiInit.PtrPciBusProp    = &PciBusProp;
ApiInit.PtrIopBus0Prop   = &IopBus0Prop;
ApiInit.PtrIopBus1Prop   = &IopBus1Prop;
ApiInit.PtrIopArbitDesc  = &IopArbitDesc;
ApiInit.PtrIopEndianDesc = &IopEndianDesc;

// Initialize the API and fill in the structures
rc = PlxInitApi(
         &ApiInit
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize API
}

/*  NOTE: All structures used to initialize the PLX chip are now
          initalized with the default bit field settings corresponding
          to the PLX chip reset values. Consult the PLX Chip's Data
          Sheet for a detailed description of the reset values.
 */
```

Cross Reference:

Referenced Item	Page
API_PARMS	4-10
PCI_BUS_PROP	4-48
IOP_BUS_PROP	4-33
IOP_ARBIT_DESC	4-31
IOP_ENDIAN_DESC	4-40
PLX_INTR	4-54

PlxInitDone

Syntax:

```
RETURN_CODE
PlxInitDone(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Sets the 'initialization done' flag in the PLX chip to signify that the local-side is initialized and will allow accesses from external PCI masters, such as the system BIOS.

Parameters:

busIndex
The bus index

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized

Notes:

Upon completion of initialization by the local CPU, this function must be called in order to allow external PCI masters to access the PCI device. Failure to do so may result in system hang.

Usage:

```
RETURN_CODE rc;


/*************************************************************
 * At this point, the PLX Chip initialization should be
 * complete. We must now set the Local Init Done Bit so
 * to "signal" that we are ready. This allows the PLX chip
 * to accept accesses from external PCI masters, such as the
 * PCI Host or BIOS.
 *************************************************************/

rc = PlxInitDone(
         PrimaryPciBus
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set intialization done
}
```

PlxInitIopArbitration

Syntax:

```
RETURN_CODE
PlxInitIopArbitration(
    BUS_INDEX       busIndex,
    IOP_ARBIT_DESC *ArbDesc
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the Local bus arbitration of the PLX chip

Parameters:

busIndex
The bus index

ArbDesc
A pointer to a structure containing the local bus arbitration parameters.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parametera is NULL

Usage:

```
IOP_ARBIT_DESC IopArbitDesc;


// Clear structure first
memset(
    &ArbDesc,
    0,
    sizeof(IOP_ARBIT_DESC)
    );

/*************************************************************
 * Local Arbitration Properties:
 *
 * - Set the latency timer count
 * - Disable Local bus DS give up bus mode
 * - Enable the Latency timer
 *************************************************************/

IopArbitDesc.IopBusDSGiveUpBusMode    = 0;
IopArbitDesc.IopBusLatencyTimer       = 0x0C;
IopArbitDesc.EnableIopBusLatencyTimer = 1;

rc = PlxInitIopArbitration(
         PrimaryPciBus,
         &IopArbitDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize local arbitration
}
```

Cross Reference:

Referenced Item	Page
IOP_ARBIT_DESC	4-31

PlxInitIopBusProperties

Syntax:

```
RETURN_CODE
PlxInitIopBusProperties(
    BUS_INDEX     busIndex,
    IOP_SPACE     IopSpace,
    IOP_BUS_PROP *IopBusProp
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the local bus access properties for a specified PCI-to-Local space.

Parameters:

busIndex
The bus index

IopSpace
The local space to modify

IopBusProp
A pointer to a structure containing the local bus properties for the specified space

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiInvalidIopSpace	The specifed PCI-to-Local space is invalid or not supported by the PLX chip
ApiNoAction	The specifed PCI-to-Local space properties cannot be modified

Usage:

```
IOP_BUS_PROP IopSpaceProp;


// Clear bus properties structure
memset(
    &IopSpaceProp;
    0,
    sizeof(IOP_BUS_PROP)
    );

/*************************************************************
 * Local Space 0 Properties:
 *
 * - Set as 32-bit bus width
 * - Set as memory space
 * - Enable Ready Input
 * - Enable Bursting
 * - Enable PCI 2.1 mode (Delayed transaction mode)
 * - Enable BREQo so DS and DM can occur simultaneously
 *************************************************************/

IopSpaceProp.IopBusWidth      = 3;
IopSpaceProp.MapInMemorySpace = 1;
IopSpaceProp.EnableReadyInput = 1;
IopSpaceProp.EnableBursting   = 1;
IopSpaceProp.PciRev2_1Mode    = 1;
IopSpaceProp.EnableIopBREQo   = 1;

// Set PCI-to-Local Space 0 properties
rc = PlxInitIopBusProperties(
         PrimaryPciBus,
         IopSpace0,
         &IopSpaceProp
         );

if (rc != ApiSuccess)
    // ERROR – Unable to set local bus properties
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	4-44
IOP_BUS_PROP	4-48

PlxInitIopEndian

Syntax:

```
RETURN_CODE
PlxInitIopEndian(
    BUS_INDEX        busIndex,
    IOP_ENDIAN_DESC *IopEndianDesc
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the endian properties of the PLX chip for the various access types

Parameters:

busIndex
The bus index

IopEndianDesc
A pointer to a structure containing the endian setting parameters

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL

Usage:

```
IOP_ENDIAN_DESC IopEndianDesc;


// Clear Endian descriptor
memset(
    &IopEndianDesc,
    0,
    sizeof(IOP_ENDIAN_DESC)
    );

IopEndianDesc.BigEIopSpace0   = 1;
IopEndianDesc.BigEIopSpace1   = 1;
IopEndianDesc.BigEDmaChannel0 = 0;
IopEndianDesc.BigEDmaChannel1 = 0;

rc = PlxInitIopEndian(
         PrimaryPciBus,
         &IopEndianDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set Endian properties
}
```

Cross Reference:

Referenced Item	Page
IOP_ENDIAN_DESC	4-40

PlxInitLocalSpace

Syntax:

```
RETURN_CODE
PlxInitLocalSpace(
    BUS_INDEX busIndex,
    IOP_SPACE IopSpace,
    ADDRESS   LocalBase,
    UDATA     size
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the mapping for a PCI-to-Local Space window or memory controller regions (480)

Parameters:

busIndex
The bus index

IopSpace
The PCI-to-Local Space window or memory controller region (480) to modify

LocalBaseAddress
The local base address for the PCI-to-Local space window or the base address of the memory controller region (480).

size
The size, in bytes, of the window or memory controller region (480)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Loca API is not initialized
ApiInvalidIopSpace	The IOP_SPACE parameter is not supported by the PLX chip
ApiInvalidSize	The size is not a power of 2 or is too small for the specified Local Space
ApiInvalidAddress	The Base address is not a multiple of the size
ApiNoAction	The specifed PCI-to-Local space properties cannot be modified

Notes:

The *size* must be a power of 2 and the *LocalBaseAddress* must be a multiple of size.

Usage:

```
/************************************************************
 * Local Space 0 Address and size:
 *
 * - Set to DRAM address and size
 ************************************************************/

rc = PlxInitLocalSpace(
         PrimaryPciBus,
         IopSpace0,
         ADDR_LS_0,     // Local space 0 base address
         SIZE_LS_0      // Size of the PCI-to-Local space window
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize PCI-to-Local space
}

#if defined(PCI480)
    // Initialize the Chip Select 0 of the memory controller
    rc = PlxInitLocalSpace(
             PrimaryPciBus,
             MsLcs0,
             0x70000000    // Set device location in memory map
             0x00100000    // 1 MB region size
             );

    if (rc != ApiSuccess)
    {
        // ERROR – Unable to initialize Chip Select 0 region
    }
#endif
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	4-44
IOP_BUS_PROP	4-33
API_PARMS	4-10
UDATA	4-8
ADDRESS	4-8

PlxInitPciArbitration

Syntax:

```
RETURN_CODE
PlxInitPciArbitration(
    BUS_INDEX       busIndex,
    PCI_ARBIT_DESC *ArbDesc
    );
```

PLX Chip Support:

480

Description:

Initializes the PCI bus arbitration properties of the PLX chip

Parameters:

busIndex
The bus index

ArbDesc
A pointer to a structure containing the PCI bus arbitration parameters.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL

Usage:

```
RETURN_CODE    rc;
PCI_ARBIT_DESC PciArbitDesc;


// Clear descriptor structure
memset(
    &PciArbitDesc,
    0,
    sizeof(PCI_ARBIT_DESC)
    );

PciArbitDesc.PciHighPriority = 1;

rc = PlxInitPciArbitration(
         PrimaryPciBus,
         &PciArbitDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize PCI arbitration descriptor
}
```

Cross Reference:

Referenced Item	Page
PCI_ARBIT_DESC	4-47

PlxInitPciBusProperties

Syntax:

```
RETURN_CODE
PlxInitPciBusProperties(
    BUS_INDEX     busIndex,
    PCI_BUS_PROP *PciBusProp
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the PCI bus properties registers for the PLX chip on the given bus index.

Parameters:

busIndex
The bus index

PciBusProp
A pointer to a structure containing the PCI bus properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is not valid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL

Usage:

```
RETURN_CODE  rc;
PCI_BUS_PROP PciBusProp;



// Clear properies structure
memset(
    &PciBusProp,
    0,
    sizeof(PCI_BUS_PROP)
    );

/************************************************************
 * PCI bus properties:
 *
 * - Assert DMPAF# when more than 0x1c entries are in the
 *   write FIFO
 ************************************************************/

PciBusProp.WFifoAlmostFullFlagCount = 0x1C;

rc = PlxInitPciBusProperties(
         PrimaryPciBus,
         &PciBusProp
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set PCI bus properties
}
```

Cross Reference:

Referenced Item	Page
PCI_BUS_PROP	4-48

PlxInitPciSpace

Syntax:

```
RETURN_CODE
PlxInitPciSpace(
    BUS_INDEX busIndex,
    PCI_SPACE PciSpace,
    ADDRESS   PciWindowBase,
    UDATA     size
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the PLX chip properties for Local-to-PCI Space accesses *(sometimes referred to as Direct Master).*

Parameters:

busIndex
The bus index

PciSpace
The specified PCI space

PciWindowBase
The Local base address to place the Local-to-PCI window.

size
The size, in bytes, of the region

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidPciSpace	The PCI Space parameter is invalid
ApiInvalidSize	The *size* parameter is not a power of 2 or is less than 64K.
ApiInvalidAddress	The Local base address parameter is not a multiple of *size*

Notes:

The size of the region must be a power of 2 and at least 64 KB. The PCI Windows base address must be a multiple of the size.

Local masters access PCI directly through this region. As a result, the region must not conflict with any other devices on the local bus memory map; otherwise, bus conflicts may occur.

Usage:

```
/************************************************************
 * Initialize Direct Master Memory and I/O
 ************************************************************/

// Initialize DM Memory Space
rc = PlxInitPciSpace(
         PrimaryPciBus,
         PciMemSpace,
         ADDR_DM_MEM,
         SIZE_DM_MAP
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize DM memory space
}

/* Initialize DM I/O Space */
rc = PlxInitPciSpace(
         PrimaryPciBus,
         PciIoSpace,
         ADDR_DM_IO,
         SIZE_DM_MAP
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize DM I/O space
}
```

Cross Reference:

Referenced Item	Page
PCI_SPACE	4-52
UDATA	4-8
ADDRESS	4-8

PlxInitPowerManagement

Syntax:

```
RETURN_CODE
PlxInitPowerManagement(
    BUS_INDEX  busIndex,
    PM_PROP   *PmProp
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the Power Management capabilities of the PLX chip

Parameters:

busIndex
The bus index

PmProp
A pointer to a structure containing the power management properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNotInitialized	The Local API is not initialized
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNullParam	One or more parameters is NULL

Usage:

```
PM_PROP     PmProp;
RETURN_CODE rc;


// Clear properties structure
memset(
    &PmProp,
    0,
    sizeof(PM_PROP)
    );

/*************************************************************
 * Power Management properties:
 *
 * - Set version to compliance with Power Management 1.0 Spec
 * - Enable PME interrupt from D0 state
 * - Enable PME interrupt from D3Hot state
 *************************************************************/

PmProp.Version            = 1;
PmProp.AssertPMEfromD0    = 1;
PmProp.AssertPMEfromD3Hot = 1;

rc = PlxInitPowerManagement(
         PrimaryPciBus,
         &PmProp
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize Power Management properties
}
```

Cross Reference:

Referenced Item	Page
PM_PROP	4-61

PlxInitVpdAddress

Syntax:

```
RETURN_CODE
PlxInitVpdAddress(
    BUS_INDEX busIndex,
    U32       VpdBaseAddress
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the base address for the Vital Product Data, which is the EEPROM write-protect address.

Parameters:

busIndex
The bus index

VpdBaseAddress
The starting 32-bit sized index in the configuration EEPROM for the Vital Product Data. To obtain the byte offset, multiply this value by 4.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNotInitialized	The Local API is not initialized
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidAddress	The VPD base address is too large

Notes:

Warning: The PLX chip loads a preset number of bytes (refer to data book) starting from offset 0 in the EEPROM. Inadvertently overwriting these may result in erroneous conditions. User Vital Product Data should be stored after the PLX reserved portion in the EEPROM.

Usage:

```
U32         VpdBase;
RETURN_CODE rc;


// Set base depending upon PLX chip used
#if defined(PCI9054)
    VpdBase = 0x30;
#elif defined(PCI480)
    VpdBase = 0x40;
#endif

rc = PlxInitVpdAddress(
         PrimaryPciBus,
         VpdBase
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set VPD base address
}
```

PlxIntrDisable

Syntax:

```
RETURN_CODE
PlxIntrDisable(
    BUS_INDEX  busIndex,
    PLX_INTR  *pPlxIntr
    );
```

PLX Chip Support:

9054, 480

Description:

Disables specific interrupts of the PLX chip.

Parameters:

busIndex
The bus index

pPlxIntr
A pointer to the interrupt structure containing the interrupts to disable.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL

Notes:

To disable an interrupt, set fields its corresponding member in the interrupt structure. Interrupts whose corresponding member is set to 0, will NOT be modified.

Usage:

```
PLX_INTR    PlxIntr;
RETURN_CODE rc;


// Clear interrupt structure
memset(
    &PlxIntr,
    0,
    sizeof(PLX_INTR)
    );

// Set interrupts to disable
plxIntr.IopDmaChannel0 = 1;         // DMA channel 0
PlxIntr.IopDoorbell    = 1;         // Local-to-PCI doorbell
PlxIntr.InboundPost    = 1;         // Messaging Unit Inbound Post Queue

rc = PlxIntrDisable(
         PrimaryPciBus,
         &PlxIntr
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to disable interrupts
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIntrEnable

Syntax:

```
RETURN_CODE
PlxIntrEnable(
    BUS_INDEX  busIndex,
    PLX_INTR  *pPlxIntr
    );
```

PLX Chip Support:

9054, 480

Description:

Enables specific interrupts of the PLX chip.

Parameters:

busIndex
The bus index

pPlxIntr
A pointer to the interrupt structure containing the interrupts to enable.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL

Notes:

To enable an interrupt, set fields its corresponding member in the interrupt structure. Interrupts whose corresponding member is set to 0, will NOT be modified.

Usage:

```
PLX_INTR    PlxIntr;
RETURN_CODE rc;


// Clear interrupt structure
memset(
    &PlxIntr,
    0,
    sizeof(PLX_INTR)
    );

// Set interrupts to disable
PlxIntr.IopDmaChannel0 = 1;         // DMA channel 0
PlxIntr.IopDoorbell    = 1;         // Local-to-PCI doorbell
PlxIntr.InboundPost    = 1;         // Messaging Unit Inbound Post Queue
PlxIntr.IopMainInt     = 1;         // Local main interrupt enable

rc = PlxIntrEnable(
         PrimaryPciBus,
         &PlxIntr
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to enable interrupts
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIntrStatusGet

Syntax:

```
BOOLEAN
PlxIntrStatusGet(
    BUS_INDEX    busIndex,
    PLX_INTR    *pPlxIntr,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the current active interrupts of the PLX chip.

Parameters:

busIndex
The bus index

pPlxIntr
A pointer to the interrupt structure which will contain the status of all interrupts.

rc
A pointer to a buffer to store the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Note:

This function returns TRUE if any PLX chip interrupt is active or FALSE if no PLX chip interrupts are active.

If an interrupt source is active, but it is disabled (masked), its corresponding member WILL be set to 1. This function only returns the interrupt status; it does not verify that the interrupt is enabled.

The *IopMainInt* and *PciMainInt* members indicate that there is at least one Local and at least one PCI interrupt active, respectively.

Usage:

```
PLX_INTR    PlxIntr;
RETURN_CODE rc;


// Verify that an interrupt is active
if (PlxIntrStatusGet(
        PrimaryPciBus,
        &PlxIntr,
        &rc
        ) == FALSE)
{
    // No PLX chip interrupt is active
}

if (rc != ApiSuccess)
{
    // ERROR – Unable to get interrupt status
}

if (PlxIntr.IopDmaChannel0 == 1);
{
    // DMA channel 0 interrupt is active
}

if (PlxIntr.IopDoorbell == 1)
{
     // Local-to-PCI doorbell interrupt is active
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIsNewCapabilityEnabled

Syntax:

```
BOOLEAN
PlxIsNewCapabilityEnabled(
    BUS_INDEX busIndex,
    U8        CapabilityToVerify
    );
```

PLX Chip Support:

9054, 480

Description:

Returns whether one or more New Capabilities are enabled.

Parameters:

busIndex
The bus index

CapabilityToVerify
A flag indicating which new capabilities to verify

Usage:

```
    // Verify that the capabilities are enabled
    if (PlxIsNewCapabilityEnabled(
            PrimaryPciBus,
            CAPABILITY_HOT_SWAP | CAPABILITY_POWER_MANAGEMENT
            ) == FALSE)
    {
        // Hot Swap and Power Management capabilities are not enabled
    }
    else
    {
        // Hot Swap and Power Management capabilities are enabled
    }
```

Cross Reference:

Referenced Item	Page
New Capabilities Flags	4-46

PlxIsPowerLevelSupported

Syntax:

```
BOOLEAN
PlxIsPowerLevelSupported(
    BUS_INDEX       busIndex,
    PLX_POWER_LEVEL PowerLevel
    );
```

PLX Chip Support:

9054, 480

Description:

Returns whether a power state is supported and enabled by the PLX chip.

Parameters:

busIndex
The bus index

PowerLevel
The Power state to verify

Return Codes:

None

Usage:

```
// Verify that the power states are supported

if (PlxIsPowerLevelSupported(
        PrimaryPciBus,
        D0
        ) == FALSE)
{
    // Device Power State D0 not supported
}
else
{
    // Device Power State D0 is supported
}

if (PlxIsPowerLevelSupported(
        PrimaryPciBus,
        D3Hot
        ) == FALSE)
{
    // Device Power State D3Hot not supported
}
else
{
    // Device Power State D3Hot is supported
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxMuHostOutboundIndexRead

Syntax:

```
UDATA
PlxMuHostOutboundIndexRead(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

480

Description:

This function basically supports the I_2O Outbound Option discussed in V2.0 of the I_2O specification. It simply returns the value of Host Outbound Index register. This value indicates how many of the posted I_2O messages were processed by the Host.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer to store the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid

Usage:

```
UDATA       MessagesProcessed;
RETURN_CODE rc;


// Get the number of messages processed
MessagesProcessed =
    PlxMuHostOutboundIndexRead(
        PrimaryPciBus,
        &rc
        );

if (rc != ApiSuccess)
    // ERROR – Unable to read the outbound index
```

PlxMuInboundPortRead

Syntax:

```
RETURN_CODE
PlxMuInboundPortRead(
    BUS_INDEX  busIndex,
    U32       *pFramePointer
    );
```

PLX Chip Support:

9054, 480

Description:

Reads the Messaging Unit Inbound Port to retrieve the next item from the Inbound Post Queue.

Parameters:

busIndex
The bus index

pFramePointer
A pointer to a 32-bit buffer to contain the returned value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiMuNotReady	The Messaging Unit has not been initialized
ApiMuFifoEmpty	The Inbound Post Queue is empty

Notes:

The Messaging Unit must be properly initialized for this function to work properly.

Usage:

```
U32         Frame;
RETURN_CODE rc;


// Read inbound port
rc = PlxMuInboundPortRead(
         PrimaryPciBus,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read from the Inbound Port
}

// Check for an empty queue
if (Frame == (U32)-1)
{
    // ERROR – Inbound Free queue is empty
}
```

PlxMuInboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuInboundPortWrite(
    BUS_INDEX  busIndex,
    U32       *pFramePointer
    );
```

PLX Chip Support:

9054, 480

Description:

Writes to the Messaging Unit Inbound Port to add an item to the Inbound Free queue.

Parameters:

busIndex
The bus index

pFramePointer
A pointer to a 32-bit buffer containing the value to write.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX s invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiMuNotReady	The Messaging Unit has not been initialized
ApiMuFifoFull	The Inbound Free Queue is full

Usage:

```
U32         Frame;
RETURN_CODE rc;


// Prepare value
Frame = 0x0100;

// Write to the inbound port
rc = PlxMuInboundPortWrite(
         PrimaryPciBus,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to the Inbound Port
}
```

PlxMuInit

Syntax:

```
RETURN_CODE
PlxMuInit(
    BUS_INDEX busIndex,
    U32       FifoSize,
    ADDRESS   LocalAddr
    );
```

PLX Chip Support:

9054, 480

Description:

Initializes the Messaging Unit of the PLX chip. This involves setting all queue base addresses.

Parameters:

busIndex
The bus index

FifoSize
The size, in bytes, of each queue. This value must be a multiple of 4.

LocalAddr
The Local base address for the Messaging Unit queues. This must be aligned on a 1MB-boundary.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidAddress	The Local address is not aligned on a 1-MB boundary
ApiInvalidSize	The FIFO size parameter is invalid for the PLX chip

Notes:

The following table lists the possible values for the *fifoSize* parameter for each PLX chip supported. The total size of the four FIFOs has been added for reference.

PLX chip	Possible FIFO Size	Total size of all 4 queues
9054	16 KBytes	64 KBytes
	32 KBytes	128 KBytes
	64 KBytes	256 KBytes
	128 KBytes	512 KBytes
	256 KBytes	1 MBytes
480	512 Bytes	2 KBytes
	2 KBytes	8 KBytes
	8 KBytes	32 KBytes
	16 KBytes	64 KBytes
	32 KBytes	128 KBytes
	64 KBytes	256 KBytes
	128 KBytes	512 KBytes
	256 KBytes	1 MBytes

Usage:

```
#define FIFO_SIZE_512B    0x00200
#define FIFO_SIZE_16K     0x04000
#define FIFO_SIZE_64K     0x10000
#define FIFO_SIZE_128K    0x20000
#define FIFO_SIZE_256K    0x40000

RETURN_CODE rc;


rc = PlxMuInit(
        PrimaryPciBus,
        FIFO_SIZE_16K,
        0x00200000      // Start queues at local address 2MB
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to initialize Messaging Unit
}
```

Cross Reference:

Referenced Item	Page
ADDRESS	4-8

PlxMuIopOutboundIndexRead

Syntax:

```
UDATA
PlxMuIopOutboundIndexRead(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

480

Description:

This function basically supports the I_2O Outbound Option discussed in V2.0 of the I_2O specification. It simply returns the PLX chip Outbound Index register value. The value indicates where the next MFA should be written to in the Host Post List queue in Host memory.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer to store the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid

Usage:

```
UDATA       OutboundIndex;
RETURN_CODE rc;


// Get the Host queue index
OutboundIndex =
    PlxMuIopOutboundIndexRead(
        PrimaryPciBus,
        &rc
        );

if (rc != ApiSuccess)
    // ERROR - Unable to read the Outbound Index
```

PlxMuIopOutboundIndexWrite

Syntax:

```
RETURN_CODE
PlxMuIopOutboundIndexWrite(
    BUS_INDEX busIndex,
    U32       IopOutboundIndex
    );
```

PLX Chip Support:

480

Description:

This function basically supports the I_2O Outbound Option discussed in version 2.0 of the I_2O specification. It simply writes to the IOP Outbound Index register.

Parameters:

busIndex
The bus index

IopOutboundIndex
The 32-bit outbound index value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiInvalidIndex	The Outbound index parameter is invalid

Usage:

```
U32         index;
RETURN_CODE rc;


// Write to the Host outbound index
rc = PlxMuIopOutboundIndexWrite(
         PrimaryPciBus,
         index
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to write to the Outbound index
}
```

PlxMuOutboundPortRead

Syntax:

```
RETURN_CODE
PlxMuOutboundPortRead(
    BUS_INDEX  busIndex,
    U32       *pFramePointer
    );
```

PLX Chip Support:

9054, 480

Description:

Reads the Messaging Unit Outbound Port to retrieve the next item from the Outbound Free Queue.

Parameters:

busIndex
The bus index

pFramePointer
A pointer to a 32-bit buffer to contain the returned value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiMuNotReady	The Messaging Unit has not been initialized
ApiMuFifoEmpty	The Outbound Free queue is empty

Notes:

The Messaging Unit must be properly initialized for this function to work properly.

Usage:

```
U32         Frame;
HANDLE      hDevice;
RETURN_CODE rc;


// Read outbound port
rc = PlxMuOutboundPortRead(
         PrimaryPciBus,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read from the outbound Port
}

// Check for an empty queue
if (Frame == (U32)-1)
{
    // ERROR – Outbound Free queue is empty
}
```

PlxMuOutboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuOutboundPortWrite(
    BUS_INDEX  busIndex,
    U32       *pFramePointer
    );
```

PLX Chip Support:

9054, 480

Description:

Writes to the Messaging Unit Outbound Port to add an item to the Outbound Post Queue.

Parameters:

busIndex
The bus index

pFramePointer
A pointer to a 32-bit buffer containing the value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is initialized
ApiNullParam	One or more parameters is NULL
ApiMuNotReady	The Messaging Unit has not been initialized
ApiMuFifoFull	The Outbound Post queue is full

Usage:

```
U32         Frame;
RETURN_CODE rc;


// Prepare value
Frame = 0x0100;

// Write to the outbound port
rc = PlxMuOutboundPortWrite(
         PrimaryPciBus,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to the outbound Port
}
```

PlxPciAbortAddrRead

Syntax:

```
U32
PlxPciAbortAddrRead(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

480

Description:

Returns the starting PCI address of the operation that caused the PCI Abort.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Usage:

```
    U32         AbortAddr;
    RETURN_CODE rc;


    AbortAddr = PlxPciAbortAddrRead(
                    PrimaryPciBus,
                    &rc
                    );

    if (rc != ApiSuccess || AbortAddr == (U32)-1)
    {
        // ERROR – Unable to read PCI abort address
    }
```

PlxPciConfigRegisterRead

Syntax:

```
RETURN_CODE
PlxPciConfigRegisterRead(
    U32  bus,
    U32  slot,
    U32  offset,
    U32 *pData
    );
```

PLX Chip Support:

9054, 480

Description:

Reads a configuration register from a PCI device specified by bus and slot number.

Parameters:

bus
The PCI bus number of the desired device

slot
The PCI slot number of the desired device

offset
The offset of the PCI configuration register to read

pData
A pointer to a 32-bit buffer which will contain the register value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidRegister	The register offset is out of range or not aligned on a 4-byte boundary
ApiNullParam	The pointer to the user-supplied buffer is NULL
ApiConfigAccessFailed	A PCI Bus Master Abort occurred, meaning no device responded

Notes:

This function reads PCI registers of other devices on the PCI bus. In order to access PCI registers of the immediate PLX chip connected to the local CPU, use the *PlxRegisterRead()* or *PlxRegisterRead()* functions.

When accessing arbitrary PCI devices on arbitrary PCI buses of a system, the following limitations exist:

- A PCI device cannot “respond to itself”; therefore, it cannot determine its own bus & slot numbers
- The PCI specification allows for configuration accesses “downstream”, but not “upstream”. The host of a system has the ability to access PCI registers of any device on any PCI bus. A device on bus 2, for example, will not be able to access PCI registers of a PCI device on bus 0. The device on bus 0, however, is able to access PCI registers of the device on bus 2.
- From the viewpoint of the local CPU, PCI bus numbers are relative. Therefore, the immediate PCI bus that the device is plugged into is always considered bus 0. Additional buses are numbered consecutively from this.
- Accessing devices on bus 0 (local CPU relative) requires a configuration type 0 access. Buses other than 0 require a configuration type 1 access so PCI-to-PCI bridges propagate the request. This API function automatically sets the configuration access type based on the bus number.

Usage:

```
U8          bus;
U8          slot;
U32         DevVenId;
RETURN_CODE rc;


// Scan PCI buses searching for devices
for (bus = 0; bus < 32; bus++)
{
    for (slot = 0; slot < 32; alot++)
    {
        rc = PlxPciConfigRegisterRead(
                bus, ,
                slot,
                0x0,        // Offset 0x0 is the Dev/Ven ID
                &DevVenId
                );

        if ((rc == ApiSuccess) && (DevVenId != (U32)-1))
        {
            // Device found
        }
    }
}
```

PlxPciConfigRegisterWrite

Syntax:

```
RETURN_CODE
PlxPciConfigRegisterWrite(
    U32  bus,
    U32  slot,
    U32  offset,
    U32 *pData
    );
```

PLX Chip Support:

9054, 480

Description:

Writes to a configuration register of a PCI device specified by bus and slot number.

Parameters:

bus
The PCI bus number of the desired device

slot
The PCI slot number of the desired device

offset
The offset of the PCI configuration register to write

pData
A pointer to a 32-bit buffer containing the data to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidRegister	The register offset is out of range or not aligned on a 4-byte boundary
ApiNullParam	The pointer to the user-supplied buffer is NULL
ApiConfigAccessFailed	A PCI Bus Master Abort occurred, meaning no device responded

Notes:

This function writes to PCI registers of other devices on the PCI bus. In order to access PCI registers of the immediate PLX chip connected to the local CPU, use the *PlxRegisterRead()* or *PlxRegisterRead()* functions.

When accessing arbitrary PCI devices on arbitrary PCI buses of a system, the following limitations exist:

- A PCI device cannot “respond to itself”; therefore, it cannot determine its own bus & slot numbers
- The PCI specification allows for configuration accesses “downstream”, but not “upstream”. The host of a system has the ability to access PCI registers of any device on any PCI bus. A device on bus 2, for example, will not be able to access PCI registers of a PCI device on bus 0. The device on bus 0, however, is able to access PCI registers of the device on bus 2.
- From the viewpoint of the local CPU, PCI bus numbers are relative. Therefore, the immediate PCI bus that the device is plugged into is always considered bus 0. Additional buses are numbered consecutively from this.
- Accessing devices on bus 0 (local CPU relative) requires a configuration type 0 access. Buses other than 0 require a configuration type 1 access so PCI-to-PCI bridges propagate the request. This API function automatically sets the configuration access type based on the bus number.

Usage:

```
U32         PciAddress;
U32         SpaceSize;
RETURN_CODE rc;


// Implement the protocol to get the size of a PCI space

// Save the current PCI address
rc = PlxPciConfigRegisterRead(
         0,
         0x12,
         CFG_BAR0,
         &PciAddress
         );

// Set all bits of register
SpaceSize = (U32)-1;
rc = PlxPciConfigRegisterWrite(
         0,
         0x12,
         CFG_BAR0,
         &SpaceSize
         );

// Read register to get its range
rc = PlxPciConfigRegisterRead(
         0,
         0x12,
         CFG_BAR0,
         &SpaceSize
         );
```

```
// Restore PCI address
rc = PlxPciConfigRegisterWrite(
         0,
         0x12,
         CFG_BAR0,
         &PciAddress
         );

// Convert range to a numeric size
SpaceSize = ~(SpaceSize) + 1;
```

PlxPinSetup

Syntax:

```
RETURN_CODE PlxPinSetup(
    BUS_INDEX         busIndex,
    BOOLEAN           IsUserPin,
    PLX_PIN_DIRECTION direction,
    USER_PIN          UserPin
    );
```

PLX Chip Support:

480

Description:

This function select whether a user pin of the IOP 480 is configured as USER0, USER1, USER2, or LCS1#, LCS2#, CINT, respectively. It then sets up the direction to be input or output if USER pin is selected. The default configuration is all pins are selected as USER with a direction of input.

Parameters:

busIndex
The bus index

IsUserPin
A flag to indicate whether the pin is configured as a USER pin or Local Chip Select (LCS) or CINT pin

direction
If Is*UserPin* is TRUE, this is the USER pin direction, Input or Output.

If Is*UserPin* is FALSE, the direction is ignored and the default LCS pin or CINT pin direction will be used.

UserPin
The USER pin number to configure.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidUserPin	This User Pin is not supported by the PLX chip
ApiBadPinDirection	The pin direction is invalid
ApiDoNothing	A User Pin was selected which does not support custom configuration

Usage:

```
RETURN_CODE rc;


// Set USER0 direction to be output
rc = PlxPinSetup(
         PrimaryPciBus,
         TRUE,
         PLX_PIN_OUTPUT,
         USER0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to configure User pin
}
```

Cross Reference:

Referenced Item	Page
PIN_DIRECTION	4-67
USER_PIN	4-68

PlxPowerConsumedRead

Syntax:

```
U32
PlxPowerConsumedRead(
    BUS_INDEX        busIndex,
    PLX_POWER_LEVEL  PowerLevel,
    RETURN_CODE     *rc
    );
```

PLX Chip Support:

480

Description:

Returns the power consumed at a power level from the IOP 480.

Parameters:

busIndex
The bus index

PowerLevel
The power level at which to determine the power consumed

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiInvalidPowerState	An invalid Power Level parameter was passed

Notes:

The returned Power Consumed value is in units of milliWatts. –1 is returned on failure conditions.

Usage:

```
U32         Power;
RETURN_CODE rc;


// Get the Power Consumed at D0 state
Power = PlxPowerConsumedRead(
            PrimaryPciBus,
            D0,
            &rc
            );

if(rc != ApiSuccess)
{
    // ERROR – Unable to determine Power Consumed
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerConsumedWrite

Syntax:

```
RETURN_CODE
PlxPowerConsumedWrite(
    BUS_INDEX       busIndex,
    PLX_POWER_LEVEL PowerLevel,
    U32             power
    );
```

PLX Chip Support:

480

Description:

Informs the PLX chip of the amount of Power Consumed at a specified Power level.

Parameters:

busIndex
The bus index

PowerLevel
The power level at which the Power Consumed should be set

power
The power consumed, in milliWatts, by the device.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidPowerState	The Power Level parameter is invalid
ApiInvalidSize	The Power parameter is out of range

Usage:

```
RETURN_CODE rc;


// Set the PLX chip to report 5 milliWatts power consumed at D2 state
rc = PlxPowerConsumedWrite(
         PrimaryPciBus,
         D2,
         5
         );

if (rc != ApiSuccess)
{
    // Unable to set the Power Consumed
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerDissipatedRead

Syntax:

```
U32
PlxPowerDissipatedRead(
    BUS_INDEX        busIndex,
    PLX_POWER_LEVEL  PowerLevel,
    RETURN_CODE     *rc
    );
```

PLX Chip Support:

480

Description:

Reads the power dissipated at a power level from the IOP 480.

Parameters:

busIndex
The bus index

PowerLevel
The power level for at which to determine the power dissipated

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX parameter is invalid
ApiInvalidPowerState	An invalid Power Level parameter was passed

Notes:

The returned Power Dissipated value is in units of milliWatts. –1 is returned on failure conditions.

Usage:

```
U32         Power;
RETURN_CODE rc;


// Get the Power Dissipated at D0 state
Power = PlxPowerDissipatedRead(
            PrimaryPciBus,
            D0,
            &rc
            );

if(rc != ApiSuccess)
{
    // ERROR – Unable to determine Power Dissipated
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerDissipatedWrite

Syntax:

```
RETURN_CODE PlxPowerDissipatedWrite(IN BUS_INDEX busIndex,
                                    IN PLX_POWER_LEVEL powerLevel,
                                    U32 power);
```

PLX Chip Support:

IOP 480

Description:

Writes to PWRDIS (Power Dissipated Values Register) at a specific power level.

Parameters:

busIndex
The bus index

PowerLevel
The power level at which the Power Dissipated should be set

power
The power dissipated, in milliWatts, by the device.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidPowerState	The Power Level parameter is invalid
ApiInvalidSize	The Power parameter is out of range

Usage:

```
RETURN_CODE rc;


// Set the PLX chip to report 5 milliWatts Power Dissipated at D2 state
rc = PlxPowerDissipatedWrite(
         PrimaryPciBus,
         D2,
         5
         );

if (rc != ApiSuccess)
{
    // Unable to set the Power Dissipated
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerLevelGet

Syntax:

```
PLX_POWER_LEVEL
PlxPowerLevelGet(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the current power level of the PLX chip

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiUnsupportedFunction	This function is not supported by the PLX chip

Usage:

```
RETURN_CODE     rc;
PLX_POWER_LEVEL PowerLevel;


// Get the PLX chip's power level
PowerLevel =
    PlxPowerLevelGet(
        PrimaryPciBus,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get Power Level
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerLevelSet

Syntax:

```
RETURN_CODE
PlxPowerLevelSet(
    BUS_INDEX       busIndex,
    PLX_POWER_LEVEL PowerLevel
    );
```

PLX Chip Support:

9054, 480

Description:

Sets the power level of the PLX chip for the given bus index.

Parameters:

busIndex
The bus index

PowerLevel
The new power level

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiUnsupportedFunction	This function is not supported by the PLX chip
ApiInvalidPowerState	The Power Level parameter is invalid

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// Put device in Low Power state
rc = PlxPowerLevelSet(
         PrimaryPciBus,
         D3Hot
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set new Power level
}

// Restore device to Full Power state
rc = PlxPowerLevelSet(
         PrimaryPciBus,
         D0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set new Power level
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPrintf

Syntax:

```
void
PlxPrintf(
    const char *format,
    ...
    );
```

PLX Chip Support:

9054, 480

Description:

Outputs a formatted string to the serial port. This function works similarly to the C *printf()* function.

Return Codes:

None.

Usage:

```
    U8 VersionMajor;
    U8 VersionMinor;
    U8 VersionRevision;


    // Display PLX API version information
    PlxSdkVersion(
        &VersionMajor,
        &VersionMinor,
        &VersionRevision
        );

    PlxPrintf("PLX API Version: %d.%d%d\n",
              VersionMajor, VersionMinor, VersionRevision);
```

PlxRegisterDoorbellRead

Syntax:

```
UDATA
PlxRegisterDoorbellRead(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the value in the PCI-to-Local doorbell register of the PLX chip. The register is also cleared.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized

Usage:

```
U32         value;
RETURN_CODE rc;


// Read the Doorbell value after the doorbell interrupt
value = PlxRegisterDoorbellRead(
            PrimaryPciBus,
            &rc
            );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get PCI-to-Local Doorbell value
}
```

Cross Reference:

Referenced Item	Page
UDATA	4-8

PlxRegisterDoorbellSet

Syntax:

```
RETURN_CODE
PlxRegisterDoorbellSet(
    BUS_INDEX busIndex,
    UDATA     data
    );
```

PLX Chip Support:

9054, 480

Description:

Writes a value to the Local-to-PCI doorbell register of the PLX chip

Parameters:

busIndex
The bus index

data
The value to write to the doorbell

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized

Usage:

```
RETURN_CODE rc;


// Write a value to Doorbell to trigger a PCI interrupt
rc = PlxRegisterDoorbellWrite(
         PrimaryPciBus,
         (1 << 30) | (1 << 8)   // Send a “message” to the Host
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set PCI-to-Local doorbell
}
```

Cross Reference:

Referenced Item	Page
UDATA	4-8

PlxRegisterMailboxRead

Syntax:

```
UDATA
PlxRegisterMailboxRead(
    BUS_INDEX    busIndex,
    MAILBOX_ID   MailboxId,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the value in a specified mailbox register of the PLX chip

Parameters:

busIndex
The bus index

MailboxId
The ID of the mailbox to read

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidRegister	The Mailbox ID parameter is invalid

Usage:

```
UDATA       MailboxValue;
RETURN_CODE rc;


MailboxValue =
    PlxRegisterMailboxRead(
        PrimaryPciBus,
        MailBox0,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read mailbox value
}
else
{
    // Can be used for custom “message passing”
    switch (MailboxValue)
    {
        case 0x01:
            // Host is ready for more data, initiate DMA
            break;

        case 0x02:
            // Host has prepared some data, transfer and process it
            break;

        case 0x03:
            // Host completed processing, log acknowledgement
            break;
    }
}
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	4-45
UDATA	4-8

PlxRegisterMailboxWrite

Syntax:

```
RETURN_CODE
PlxRegisterMailboxWrite(
    BUS_INDEX  busIndex,
    MAILBOX_ID MailboxId,
    UDATA      data
    );
```

PLX Chip Support:

9054, 480

Description:

Writes a value to a specified mailbox register of the PLX chip.

Parameters:

busIndex
The bus index

MailboxId
The mailbox register ID

data
The value to write to the mailbox register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidRegister	The Mailbox ID parameter is invalid

Usage:

```
RETURN_CODE rc;

/*******************************************************
 * In this example, the Local CPU prepares some data or a
 * message for processing by the Host. The data is
 * placed in local memory and the communication involves
 * writing the local base address of the data to Mailbox 1
 * and then setting Bit 8 of MailBox 2 to denote the data is
 * ready.
 *
 *   - Data preparation & transfer not show here -
 ************************************************/

// Write local address of data ready for processing
rc = PlxRegisterMailboxWrite(
         PrimaryPciBus,
         MailBox1,
         0x00080000
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to Mailbox register
}

// Signal that data is ready
rc = PlxRegisterMailboxWrite(
         PrimaryPciBus,
         MailBox2,
         (1 << 8)
         );
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	4-45
UDATA	4-8

PlxRegisterRead

Syntax:

```
UDATA
PlxRegisterRead(
    BUS_INDEX    busIndex,
    U32          offset,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Reads a register of the PLX chip on the selected PLX PCI device.

Parameters:

busIndex
The bus index

offset
The local offset of the PLX register to read

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidRegister	The register offset is out of range or not aligned on a 4-byte boundary

Usage:

```
U32         RegValue;
RETURN_CODE rc;


// Check if an EEPROM is present on the 9054 device
RegValue =
    PlxRegisterRead(
        PrimaryPciBus,
        PCI9054_EEPROM_CTRL_STAT,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read PLX chip register
}
```

Cross Reference:

Referenced Item	Page
UDATA	4-8

PlxRegisterReadAll

Syntax:

```
RETURN_CODE
PlxRegisterReadAll(
    BUS_INDEX  busIndex,
    U32        StartOffset,
    U32        SizeInBytes,
    UDATA     *buffer
    );
```

PLX Chip Support:

9054, 480

Description:

Reads multiple consecutive registers of a PLX PCI device.

Parameters:

busIndex
The bus index

StartOffset
The register offset to start reading from (aligned on 4-byte boundary)

SizeInBytes
The number of bytes to read (must be a multiple of 4)

buffer
A pointer to a buffer which will contain the register values

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiInvalidRegister	The register offset is out of range or not aligned on a 4-byte boundary

Notes:

The application-provided buffer must be at least equal to or larger than the number of bytes to read.

Usage:

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


rc = PlxRegisterReadAll(
         PrimaryPciBus,
         0x0,
         0x100,
         buffer
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read register range
}
```

Cross Reference:

Referenced Item	Page
UDATA	4-8

PlxRegisterWrite

Syntax:

```
RETURN_CODE
PlxRegisterWrite(
    BUS_INDEX busIndex,
    U32       offset,
    UDATA     data
    );
```

PLX Chip Support:

9054, 480

Description:

Writes a value to a register of the PLX chip on the selected PLX PCI device.

Parameters:

busIndex
The bus index

offset
The register offset to write to (aligned on 4-byte boundary)

data
The data to write to the register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	One or more parameters is NULL
ApiInvalidRegister	The register offset is out of range or not aligned on a 4-byte boundary

Usage:

```
RETURN_CODE rc;


// Adjust the remap of a 9054 chip
rc = PlxRegisterWrite(
         PrimaryPciBus,
         PCI9054_SPACE0_REMAP,
         0x43000000 | (1 << 0)
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to PLX chip register
}
```

Cross Reference:

Referenced Item	Page
UDATA	4-8

PlxSdkVersion

Syntax:

```
RETURN_CODE
PlxSdkVersion(
    U8 *VersionMajor,
    U8 *VersionMinor,
    U8 *VersionRevision
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the SDK API version information

Parameters:

VersionMajor
A pointer to an 8-bit buffer to contain the Major version number

VersionMinor
A pointer to an 8-bit buffer to contain the Minor version number

VersionRevision
A pointer to an 8-bit buffer to contain the Revision version number

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL

Usage:

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


rc = PlxSdkVersion(
         &SdkMajor,
         &SdkMinor,
         &SdkRevision
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get API version information
}
else
{
    PlxPrintf("PLX Local API Version = %d.%d%d\n",
              SdkMajor, SdkMinor, SdkRevision);
}
```

PlxSerialEepromPresent

Syntax:

```
BOOLEAN
PlxSerialEepromPresent(
    BUS_INDEX    busIndex,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Determines whether a Serial EEPROM device is found on the PCI device

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function completed successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiEepromBlank	The EEPROM is blank (480 only)

Notes:

For PLX chips other than the 480, a blank EEPROM will result in a return value of FALSE. The PLX chip does not distinguish between blank and non-existent EEPROMs.

Usage:

```
BOOLEAN     EepromPresent;
RETURN_CODE rc;


EepromPresent =
    PlxSerialEepromPresent(
        PrimaryPciBus,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to determine EEPROM status
}

if (EepromPresent)
{
    // Programmed EEPROM exists
}
else
{
    // EEPROM does not exist or is blank
}
```

PlxSerialEepromRead

Syntax:

```
RETURN_CODE
PlxSerialEepromRead(
    BUS_INDEX    busIndex,
    EEPROM_TYPE  EepromType,
    UDATA       *pBuffer,
    U32          size
    );
```

PLX Chip Support:

9054, 480

Description:

Reads values from the configuration EEPROM connected to the PLX chip

Parameters:

busIndex
The bus index

EepromType
The type of EEPROM installed on the PCI device

pBuffer
A pointer to a buffer which will contain the data read. This buffer must be large enough to hold the amount of data requested.

size
The number of bytes to read from the EEPROM. (Must be 4-byte aligned)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiUnsupportedFunction	This function is not supported by the PLX chip
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	The user-supplied buffer pointer parameter is NULL
ApiEepromTypeNotSupported	The EEPROM type parameters unsupported for the PLX chip
ApiInvalidSize	The size parameter is 0, is too large for the EEPROM, or is not 4-byte aligned
ApiVpdNotEnabled	The VPD feature in the PLX chip is disabled.

Notes:

Attempting to access a PLX device when an EEPROM is not physically present may result in a crash.

The EEPROM data is always read starting at EEPROM offset 0.

Usage:

```
U16         EepromData[0x16];
RETURN_CODE rc;


// Read first few bytes of EEPROM data
rc = PlxSerialEepromRead(
         PrimaryPciBus,
         Eeprom93CS56,
         (UDATA *)EepromData,
         0x16 * sizeof(U16)        // Size in bytes
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read EEPROM
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	4-29
UDATA	4-8

PlxSerialEepromWrite

Syntax:

```
RETURN_CODE
PlxSerialEepromWrite(
    BUS_INDEX    busIndex,
    EEPROM_TYPE  EepromType,
    UDATA       *pBuffer,
    U32          size
    );
```

PLX Chip Support:

9054, 480

Description:

Writes values to the configuration EEPROM connected to the PLX chip

Parameters:

busIndex
The bus index

EepromType
The type of EEPROM installed on the PCI device

pBuffer
A pointer to a buffer containing the data to write

size
The number of bytes to write to the EEPROM. (Must be 4-byte aligned)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiUnsupportedFunction	This function is not supported by the PLX chip
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiNullParam	The user-supplied buffer pointer parameter is NULL
ApiEepromTypeNotSupported	The EEPROM type parameters unsupported for the PLX chip
ApiInvalidSize	The size parameter is 0, is too large for the EEPROM, or is not 4-byte aligned
ApiVpdNotEnabled	The VPD feature in the PLX chip is disabled.

Notes:

Attempting to access a PLX device when an EEPROM is not physically present may result in a crash.

The EEPROM data is always written starting at EEPROM offset 0.

Usage:

```
U16         EepromData[0x4];
RETURN_CODE rc;


// Prepare EEPROM data (assuming 9054)
EepromData[0] = 0x9054     // Device/Vendor ID
EepromData[1] = 0x10b5
EepromData[2] = 0x0068     // Class code/Revision
EepromData[3] = 0x0001

// Write EEPROM data
rc = PlxSerialEepromWrite(
          PrimaryPciBus,
          Eeprom93CS56,
          (UDATA *)eepromData,
          0x4 * sizeof(U16)     //  Size in bytes
          );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to EEPROM
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	4-29
UDATA	4-8

PlxUserRead

Syntax:

```
PLX_PIN_STATE
PlxUserRead(
    BUS_INDEX    busIndex,
    USER_PIN     UserPin,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the state of a specified USER pin of a PLX chip.

Parameters:

busIndex
The bus index

UserPin
The USER pin to read

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidUserPin	The User pin is invalid or not supported by the PLX chip
ApiBadPinDirection	This User pin is not an input pin

Usage:

```
HANDLE        hDevice;
RETURN_CODE   rc;
PLX_PIN_STATE PinState;


PinState = PlxUserRead(
               PrimaryPciBus,
               USER0,
               &rc
               );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read state of USER pin
}

if (PinState == Active)
{
    // User pin is active
}
```

Cross Reference:

Referenced Item	Page
USER_PIN	4-68
PLX_PIN_STATE	4-53

PlxUserWrite

Syntax:

```
RETURN_CODE
PlxUserWrite(
    BUS_INDEX     busIndex,
    USER_PIN      UserPin,
    PLX_PIN_STATE PinState
    );
```

PLX Chip Support:

9054, 480

Description:

Set the state of a specified User pin of the PLX chip

Parameters:

busIndex
The bus index

UserPin
The USER pin to read

PinState
The new state to set the User pin

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNotInitialized	The Local API is not initialized
ApiInvalidUserPin	The User pin is invalid or not supported by the PLX chip
ApiInvalidUserState	The User pin state is invalid
ApiBadPinDirection	This User pin is not an input pin

Usage:

```
RETURN_CODE rc;


// Write to the User pin
rc = PlxUserWrite(
         PrimaryPciBus,
         USER0,
         Active
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to User pin
}
```

Cross Reference:

Referenced Item	Page
USER_PIN	4-68
PLX_PIN_STATE	4-53

PlxVerifyEndianAccess

Syntax:

```
void
PlxVerifyEndianAccess(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Ensures that the endian setting corresponding to access of the PLX chip registers by the local CPU is correct.

Parameters:

busIndex
The bus index

Usage:

```
IOP_ENDIAN_DESC IopEndianDesc;

// Clear Endian descriptor
memset(
    &IopEndianDesc,
    0,
    sizeof(IOP_ENDIAN_DESC)
    );

IopEndianDesc.BigEIopSpace0   = 1;
IopEndianDesc.BigEIopSpace1   = 1;
IopEndianDesc.BigEDmaChannel0 = 0;
IopEndianDesc.BigEDmaChannel1 = 0;

rc = PlxInitIopEndian(
         PrimaryPciBus,
         &IopEndianDesc
         );

// Verify that the PLX chip's register access endian setting
PlxVerifyEndianAccess(
    PrimaryPciBus,
    );
```

PlxVpdDisable

Syntax:

```
RETURN_CODE
PlxVpdDisable(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Disables the Vital Product Data (VPD) Capability feature of the PLX chip

Parameters:

busIndex
The bus index

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Notes:

This function disables the VPD feature by removing it from the linked list of New Capabilities in the PCI registers.

Usage:

```
RETURN_CODE rc;


rc = PlxVpdDisable(
         PrimaryPciBus
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to disable VPD
}
```

PlxVpdEnable

Syntax:

```
RETURN_CODE
PlxVpdEnable(
    BUS_INDEX busIndex
    );
```

PLX Chip Support:

9054, 480

Description:

Enables the Vital Product Data (VPD) Capability feature of the PLX chip

Parameters:

busIndex
The bus index

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Notes:

This function enables the Hot Swap feature by adding it to the linked list of New Capabilities in the PCI registers.

Usage:

```
RETURN_CODE rc;


rc = PlxVpdEnable(
         PrimaryPciBus
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to enable VPD
}
```

PlxVpdIdRead

Syntax:

```
U8
PlxVpdIdRead(
    BUS_INDEX    busIndex
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the Vital Product Data (VPD) Capability ID

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Notes:

A value of –1 is returned if there is an error reading the VPD ID.

Usage:

```
    U8          VpdId;
    RETURN_CODE rc;


    VpdId = PlxVpdIdRead(
                PrimaryPciBus,
                &rc
                );

    if (rc != ApiSuccess)
        // ERROR - Unable to read VPD Capability ID
```

PlxVpdNcpRead

Syntax:

```
U8
PlxVpdNcpRead(
    BUS_INDEX    busIndex
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Returns the Next Capability Pointer from the VPD register.

Parameters:

busIndex
The bus index

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Notes:

A value of –1 is returned if there is an error reading the Hot Swap ID.

Usage:

```
    U8          NextCapability;
    RETURN_CODE rc;


    NextCapability = PlxVpdNcpRead(
                         PrimaryPciBus,
                         &rc
                         );

    if (rc != ApiSuccess)
        // ERROR - Unable to read VPD NCP
```

PlxVpdNcpWrite

Syntax:

```
RETURN_CODE
PlxVpdNcpWrite(
    BUS_INDEX busIndex,
    U8        value
    );
```

PLX Chip Support:

9054, 480

Description:

Writes a value to the Next Capability Pointer of the VPD register.

Parameters:

busIndex
The bus index

value
The value to write to the Hot Swap Next Capability Pointer

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Usage:

```
RETURN_CODE rc;


rc = PlxVpdNcpWrite(
         PrimaryPciBus,
         0x48
         );

if (rc != ApiSuccess )
{
    // ERROR – Unable to write VPD Capability NCP
}
```

PlxVpdRead

Syntax:

```
U32
PlxVpdRead(
    BUS_INDEX    busIndex,
    U16          offset,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

9054, 480

Description:

Reads a 32-bit value at a specified offset of the EEPROM using the Vital Product Data feature of the selected PLX chip.

Parameters:

busIndex
The bus index

offset
The is the byte offset to read from (must be aligned 32-bit boundary)

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidOffset	The offset is invalid, out of range, or nota aligned on a 32-bit boundary
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Usage:

```
U32         VpdData;
RETURN_CODE rc;

// Read the default Space 1 range (assuming a 9054)
VpdData = PlxVpdRead(
              PrimaryPciBus,
              0x48,
              &rc
              );
```

PlxVpdWrite

Syntax:

```
RETURN_CODE
PlxVpdWrite(
    BUS_INDEX busIndex,
    U16       offset,
    U32       VpdData
    );
```

PLX Chip Support:

9054, 480

Description:

Writes Vital Product Data value to the VPD Data Register.

Parameters:

busIndex
The bus index

offset
The is the byte offset to write to (must be aligned 32-bit boundary)

VpdData
The 32-bit data to write to the EEPROM.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidOffset	The offset is invalid, out of range, or nota aligned on a 32-bit boundary
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Usage:

```
RETURN_CODE rc;


/*****************************************************
 * If the offset is within the write protected area, the
 * EEPROM write-protect boundary must first be adjusted.
 *
 *  - Write-protect boundary adjustment not shown here -
 *****************************************************/

// Write the new Device/Vendor ID (assuming 9054 device)
rc = PlxVpdWrite(
         PrimaryPciBus,
         0x0,
         0x186010b5
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to VPD
}

// Write custom data to non-PLX used EEPROM space
rc = PlxVpdWrite(
         PrimaryPciBus,
         0x60,         // 9054 data ends at 0x58
         0x0024beef
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to VPD
}
```

2.4 IOP 480 Serial Port Unit (SPU) API Function Details

The functions in this section are provided for the IOP 480 Serial Port Unit. They are provided for reference purposes and to support existing applications only. The new PLX BSP, included in SDK 3.2, now provides more efficient functions for serial I/O operations. The implementation of these functions can be found in IOP 480 RDK BSP and provide a small performance advantage over the API calls listed here.

Some items should be noted regarding the IOP480 Serial Port Unit. The Baud Rate divisor is dependent upon the speed of the IOP480 CPU core, which runs at the external clock frequency. Since the software cannot directly determine the speed of the external clock, PLX software assumes the following:

- If an EEPROM is not present, assume the clock frequency is 66 MHz
- If an EEPROM is present, the code will read the 32-bit value at EEPROM byte offset 100h
    - If the value is within a preset range of reasonable Hz values, this is assumed to be the external clock speed. Valid clock frequencies ranges from 25000 Hz to 99000 Hz.
    - If the value is invalid, the default of 66 MHz is assumed.

Refer to the IOP 480 RDK BSP or the IOP 480 EEPROM screen in PLXMon for additional information.

PlxSpuBaudRateSet

Syntax:

```
RETURN_CODE
PlxSpuBaudRateSet(
    BUS_INDEX busIndex,
    U32       BaudRate,
    U32       LclkFreq
    );
```

PLX Chip Support:

480

Description:

Sets the SPU baud rate

Parameters:

busIndex
The bus index

BaudRate
The new baud rate. It should be one of the following: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200

LclkFreq
The external system clock (LCLK) frequency in Hz. This value is used in the calculation for the Baud Rate divisor, so it must match the actual local bus clock frequency.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid

Usage:

```
U32         LocalClockFrequency;   // IOP 480 Clock frequency
RETURN_CODE rc;


// Set the BAUD rate
rc = PlxSpuBaudRateSet(
         PrimaryPciBus,
         38400,
         66666 * 1000      // 66 MHz clock
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set BAUD rate
}
```

PlxSpuDataRead

Syntax:

```
RETURN_CODE
PlxSpuDataRead(
    BUS_INDEX  busIndex,
    U32       *pBufferSize,
    U8        *pBuffer
    );
```

PLX Chip Support:

480

Description:

Reads from the Serial Port Receive Buffer (SPRB) until the buffer size is reached or no more data is available.

Parameters:

busIndex
The bus index

pBufferSize
A pointer to the size of the buffer to store the data to be read. Upon API return, the size will be the number of bytes actually read. This will be no larger than the size specified when the function was called.

pBuffer
A pointer to a buffer which will contain the data read

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidSize	The buffer size parameter value is 0
ApiNullParam	One or more parameters is NULL
ApiBufferNotReady	The Serial Port Receiver Buffer is not ready, possibly an SPU error condition

Usage:

```
U8          buffer[10];
U32         size;
RETURN_CODE rc;


// Set maximum size of buffer
size = 10;

// Read data from the Serial Port
rc = PlxSpuDataRead(
         PrimaryPciBus,
         &size,
         buffer
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read data from Serial Port
}
```

PlxSpuDataWrite

Syntax:

```
RETURN_CODE
PlxSpuDataWrite(
    BUS_INDEX  busIndex,
    U32       *bufferSize,
    U8        *pbuffer
    );
```

PLX Chip Support:

480

Description:

Writes a data buffer to the Serial Port Transmit Buffer (SPTB)

Parameters:

busIndex
The bus index

pBufferSize
A pointer to the size of the buffer containing the data to write. Upon API return, the size will contain the number of bytes actually written.

pBuffer
A pointer to a buffer containing the data to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidSize	The buffer size parameter value is 0
ApiNullParam	One or more parameters is NULL
ApiBufferNotReady	The Serial Port Receiver Buffer is not ready, possibly an SPU error condition

Usage:

```
U8          buffer[10];
U32         size;
RETURN_CODE rc;

// Set the amount of data to transmit
size = 10;

// Write data to the Serial Port
rc = PlxSpuDataWrite(
         PrimaryPciBus,
         &size,
         buffer
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write data to Serial Port
}
```

PlxSpuInit

Syntax:

```
RETURN_CODE
PlxSpuInit(
    BUS_INDEX  busIndex,
    SPU_DESC  *pSpuDesc
    );
```

PLX Chip Support:

480

Description:

Initializes the Serial Port Unit of the IOP 480

Parameters:

busIndex
The bus index

pSpuDesc
A pointer a structure containing the SPU configuration properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiFailed	The initialization failed
ApiNullParam	The SPU descriptor parameter is invalid
ApiInvalidBusIndex	The BUS_INDEX is invalid

Usage:

```
SPU_DESC    SpuDesc;
RETURN_CODE rc;


// Clear descriptor
memset(
    &SpuDesc,
    0,
    sizeof(SPU_DESC)
    );

// Initialize the SPU
SpuDesc.BaudRate = BaudRate;
SpuDesc.LclkFreq = 66666 * 1000;
SpuDesc.LM       = 0;      // Normal LoopBack mode
SpuDesc.DTR      = 0;      // DTR inactive
SpuDesc.RTS      = 0;      // RTS inactive
SpuDesc.DB       = 1;      // 8 Data bits
SpuDesc.PE       = 0;      // No Parity
SpuDesc.PTY      = 0;      // Even Parity
SpuDesc.SB       = 0;      // 1 Stop bit

rc = PlxSpuInit(
         PrimaryPciBus,
         &SpuDesc
         );
```

Cross Reference:

Referenced Item	Page
SPU_DESC	4-63

PlxSpuRegisterRead

Syntax:

```
U8
PlxSpuRegisterRead(
    BUS_INDEX    busIndex,
    U32          offset,
    RETURN_CODE *rc
    );
```

PLX Chip Support:

480

Description:

Reads an SPU register at a specified offset.

Parameters:

busIndex
The bus index

offset
Offset of the SPU register to read

rc
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidOffset	The SPU register offset is out of range or not aligned on a 4-byte boundary

Usage:

```
U8          RegValue;
RETURN_CODE rc;


// Read the Serial Port Line Status register
RegValue = PlxSpuRegisterRead(
               PrimaryPciBus,
               IOP480_SPLS,
               &rc,
               );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read SPU register
}

// Check for errors and clear them if any
if (RegValue & 0x78)
{
    rc = PlxSpuRegisterWrite(
              PrimaryPciBus,
              IOP480_SPLS,
              0x78
              );
}
```

PlxSpuRegisterWrite

Syntax:

```
RETURN_CODE
PlxSpuRegisterWrite(
    BUS_INDEX busIndex,
    U32       offset,
    U8        DataByte
    );
```

PLX Chip Support:

480

Description:

Writes to an SPU register at a specified offset.

Parameters:

busIndex
The bus index

offset
Offset of the SPU register to write

DataByte
The 8-bit value to write to the register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidOffset	The SPU register offset is out of range or not aligned on a 4-byte boundary

Usage:

```
U8          RegValue;
RETURN_CODE rc;


// Read the Serial Port Line Status register
RegValue = PlxSpuRegisterRead(
               PrimaryPciBus,
               IOP480_SPLS,
               &rc,
               );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read SPU register
}

// Check for errors and clear them if any
if (RegValue & 0x78)
{
    rc = PlxSpuRegisterWrite(
              PrimaryPciBus,
              IOP480_SPLS,
              0x78
              );
}
```

PlxSpuStatus

Syntax:

```
RETURN_CODE
PlxSpuStatus(
    BUS_INDEX   busIndex,
    SPU_STATUS *pSpuStatus,
    BOOLEAN     write
    );
```

PLX Chip Support:

480

Description:

This function will read the SPU Status Structure into the buffer, or write to the SPU status registers according to parameter *write.*

Parameters:

busIndex
The bus index

pSpuStatus
A pointer to an SPU_STATUS structure

write
A flag specifying whether the action is read or write.
If *write* is TRUE, the function will write to the SPU Control register.
If *write* is FALSE, the function will read from the SPU Control register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNullParam	One or more parameters is NULL

Usage:

```
SPU_STATUS  SpuStatus;
RETURN_CODE rc;


rc = PlxSpuStatus(
         PrimaryPciBus,
         &SpuStatus,
         FALSE
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read SPU Status
}
```

Cross Reference:

Referenced Item	Page
SPU_STATUS	4-65

3 PCI Host API

The PLX PCI Host API is designed for a host environment, in which the PLX chip is accessed through the PCI bus from the host Operating System, typically through a device driver.

Note: The PLX SDK Version 3.2 Host API supports the following PLX Devices: 9080, 9054, 480, 9050, 9052 and 9030.

3.1 PCI Host API Function Quick Reference List

API Function Name	Purpose	Page
PlxPciConfigRegisterWrite	Write to a Configuration register of a PCI device.	3-87
PlxPciDeviceClose	Close a PLX device	3-89
PlxPciDeviceFind	Find a PLX device on the PCI bus.	3-90
PlxPciDeviceOpen	Open a PLX device	3-93
PlxPmIdRead	Read the Power Management Capability ID	3-96
PlxPmIdNcpRead	Read the Power Management Next Capability Pointer	3-97
PlxPowerLevelGet	Get the current power level	3-98
PlxPowerLevelSet	Set the power level	3-100
PlxRegisterDoorbellRead	Get the last value written to the Local-to-PCI Doorbell	3-102
PlxRegisterDoorbellSet	Write a value to the PCI-to-Local Doorbell	3-104
PlxRegisterMailboxRead	Read a Mailbox register	3-105
PlxRegisterMailboxWrite	Write to a Mailbox register	3-107
PlxRegisterRead	Read a PLX chip local register	3-109
PlxRegisterReadAll	Read all PLX chip local registers	3-111
PlxRegisterWrite	Write to a PLX chip local register	3-113
PlxSdkVersion	Get API Version information	3-115
PlxSerialEepromPresent	Determine if a Serial EEPROM is present	3-117
PlxSerialEepromRead	Read from the serial EEPROM	3-119
PlxSerialEepromWrite	Write to the serial EEPROM	3-121
PlxUserRead	Read a USER pin value	3-123
PlxUserWrite	Write a value to a USER pin	3-125
PlvVpdIdRead	Read the VPD Capability ID	3-127
PlxVpdNcpRead	Read the VPD Next Capability Pointer	3-128
PlxVpdRead	Read a 32-bit value from the Vital Product Data	3-129
PlxVpdWrite	Write a 32-bit value to the Vital Product Data.	3-131

3.2 PCI Host API Function Details

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

The following is a sample entry demonstrating the information provided.

Sample Function Entry

Syntax:

```
function(
    Parameters,
    ...
    );
```

This gives the declaration syntax for each function.

PLX Chip Support:

A list of PLX chips that support this function. Note: *ApiUnsupportedFunction* will be returned if an API function is called for a chip that doesn't support the function.

Description:

Summary of the function's purpose

Parameters:

The function parameters

Return Codes:

The possible codes returned by the function.

Notes:

Provides any relevant information pertaining to the function.

Usage:

Sample source code is provided to demonstrate how the function is used. Note that the sample code has not been compiled or tested. It is provided for reference purposes only.

Cross Reference:

Provides page numbers to any relevant data structures or types.

PlxBusIopRead

Syntax:

```
RETURN_CODE
PlxBusIopRead(
    HANDLE       hDevice,
    IOP_SPACE    iopSpace,
    U32          address,
    BOOLEAN      remapAddress,
    U32         *destination,
    U32          transferSize,
    ACCESS_TYPE  accessType
    );
```

PLX Chip Support:

All

Description:

This function attempts to read from the local bus of a PCI device containing a PLX chip (sometimes referred to as Direct Slave Read).

Parameters:

hDevice
Handle of an open PCI device

iopSpace
Which PCI–to-Local Space to access. The PLX chip type determines the PCI BAR register used.

address
If *remapAddress* is FALSE, *address* is an offset from the PCI-to-Local Space. The mapping will not be adjusted because the function assumes the space is already mapped correctly. The data range accessed must not be larger than the size of the PCI-to-Local Space window.

If *remapAddress* is TRUE, *address* is the base of the actual local bus address to start reading from. For 32-bit devices, this allows access to any location on the 4GB local bus space.

remapAddress
Flag that determines how the *address* parameter is treated and whether the mapping is adjusted.

destination
A pointer to a user supplied buffer that will contain the retrieved data. This buffer must be large enough to hold the amount of data requested.

transferSize
The number of bytes to read. Note: This a number of bytes, not units of data determined by *accessType*.

accessType
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInsufficientResources	The API was unable to communicate with the driver due to insufficient resources
ApiInvalidAccessType	An invalid or unsupported ACCESS_TYPE parameter
ApiInvalidAddress	The *address* parameter is not aligned based on the *accessType*
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the *accessType*

Notes:

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

This function requires that the PCI-to-Local space is valid, enabled, and the descriptors are setup properly. Incorrect settings may result in incorrect data or system crashes.

This function is not recommended when performance is a concern. Error checking, local window adjustment, and the overhead of transferring the buffer to/from the driver keeps performance less than optimal. For increased performance, use the *PlxPciBaseAddressGet()* function and access local memory directly through virtual addresses. The disadvantage to this method is that users are responsible for manually configuring the PLX chip local space re-map window. This will affect code portability, but overall performance is greater than using the API function.

The end result of this function is a read from the device local bus. If no device on the local bus responds, system crashes may result. Please make sure that valid devices are accessible and addresses are correct before using this function.

When using this function with IOP 480 Local Space 0, it is important to note that the first 400h bytes of Space 0 are mapped to the internal registers. Accesses above 400h map to the local bus starting at offset 400h, not offset 0. As a result, the first 400h bytes of any local space window are inaccessible through Space 0. One of the other spaces, such as Space 1, can be used instead. *Refer to the IOP 480 data book for additional information.*

Usage:

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


// Read from an absolute local bus address
rc = PlxBusIopRead(
         hDevice,
         IopSpace0,
         0x00100000,         // Absolute local address of 1MB
         TRUE,               // Remap since an absolute address is used
         buffer,             // Destination buffer
         sizeof(buffer),
         BitSize32           // 32-bit accesses
         );
```

```
if (rc != ApiSuccess)
{
    // ERROR - Unable to read data
}

// Read from an offset into the PCI-to-Local space window
rc = PlxBusIopRead(
         hDevice,
         IopSpace0,
         0x00000100,         // Start reading at an offset
         FALSE,              // No remapping since an offset is used
         buffer,             // Destination buffer
         sizeof(buffer),
         BitSize16           // 16-bit accesses
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read data
}
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	4-44
ACCESS_TYPE	4-9

PlxBusIopWrite

Syntax:

```
RETURN_CODE
PlxBusIopWrite(
    HANDLE       hDevice,
    IOP_SPACE    iopSpace,
    U32          address,
    BOOLEAN      remapAddress,
    U32         *source,
    U32          transferSize,
    ACCESS_TYPE  accessType
    );
```

PLX Chip Support:

All

Description:

This function attempts to write from the local bus of a PCI device containing a PLX chip (sometimes referred to as Direct Slave Write).

Parameters:

hDevice
Handle of an open PCI device

iopSpace
Which PCI–to-Local Space to access. The PLX chip type determines the PCI BAR register used.

address
If *remapAddress* is FALSE, *address* is an offset from the PCI-to-Local Space. The mapping will not be adjusted because the function assumes the space is already mapped correctly. The data range accessed must not be larger than the size of the PCI-to-Local Space window.

If *remapAddress* is TRUE, *address* is the base of the actual local bus address to start reading from. For 32-bit devices, this allows access to any location on the 4GB local bus space.

remapAddress
Flag that determines how the *address* parameter is treated and whether the mapping is adjusted.

source
A pointer to a user supplied buffer that contain the data to write.

transferSize
The number of bytes to write. Note: This a number of bytes, not units of data determined by *accessType*.

accessType
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInsufficientResources	The API was unable to communicate with the driver due to insufficient resources
ApiInvalidAccessType	An invalid or unsupported ACCESS_TYPE parameter
ApiInvalidAddress	The *address* parameter is not aligned based on the *accessType*
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the *accessType*

Notes:

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

This function requires that the PCI-to-Local space is valid, enabled, and the descriptors are setup properly. Incorrect settings may result in incorrect data or system crashes.

This function is not recommended when performance is a concern. Error checking, local window adjustment, and the overhead of transferring the buffer to/from the driver keeps performance less than optimal. For increased performance, use the *PlxPciBaseAddressGet()* function and access local memory directly through virtual addresses. . The disadvantage to this method is that users are responsible for manually configuring the PLX chip local space re-map window. This will affect code portability, but overall performance is greater than using the API function.

The end result of this function is a write to the device local bus. If no device on the local bus responds, system crashes may result. Please make sure that valid devices are accessible and addresses are correct before using this function.

When using this function with IOP 480 Local Space 0, it is important to note that the first 400h bytes of Space 0 are mapped to the internal registers. Accesses above 400h map to the local bus starting at offset 400h, not offset 0. As a result, the first 400h bytes of any local space window are inaccessible through Space 0. One of the other spaces, such as Space 1, can be used instead. *Refer to the IOP 480 data book for additional information.*

Usage:

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


// Prepare buffer
for (i=0; i < sizeof(buffer) / 4; i++)
{
    buffer[i] = i;
}
```

```
// Write to an absolute local bus address
rc = PlxBusIopWrite(
         hDevice,
         IopSpace0,
         0x00100000,         // Absolute local address of 1MB
         TRUE,               // Remap since an absolute address is used
         buffer,             // Source buffer
         sizeof(buffer),
         BitSize32           // 32-bit accesses
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to write data
}

// Write to an offset from the PCI-to-Local space window
rc = PlxBusIopWrite(
         hDevice,
         IopSpace0,
         0x00000100,         // Start writing at an offset
         FALSE,              // No remapping since an offset is used
         buffer,             // Source buffer
         sizeof(buffer),
         BitSize16           // 16-bit accesses
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to write data
}
```

Cross Reference:

Referenced Item	Page
IOP_SPACE	4-44
ACCESS_TYPE	4-9

PlxChipTypeGet

Syntax:

```
RETURN_CODE
PlxChipTypeGet(
    HANDLE  hDevice,
    U32    *pChipType,
    U8     *pRevision
    );
```

PLX Chip Support:

All

Description:

Returns the PLX chip type and its revision number

Parameters:

hDevice
Handle of an open PCI device

pChipType
A pointer to a 32-bit buffer to contain the PLX chip type

pRevision
A pointer to an 8-bit buffer to contain the PLX chip revision number

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL

Notes:

The chip type is returned as a hex number matching the chip number. For examples, 0x9054 = 9054 and 0x0480 = IOP 480.

The Revision numbering begins at 1. For example, a 9054 AB returns a revision number of 2.

Usage:

```
U8          Revision;
U32         ChipType;
RETURN_CODE rc;


rc = PlxChipTypeGet(
         hDevice
         &ChipType,
         &Revision
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get PLX chip type
}
else
{
    switch (ChipType)
    {
        case 0x9054:
            // Chip is a 9054
            if (Revision = 0x1)
                // Chip is an AA version
            if (Revision = 0x2)
                // Chip is an AB version
            break;

        case 0x480:
            // Chip is an IOP 480
            break
    }
}
```

PlxDmaBlockChannelClose

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelClose(
    HANDLE      hDevice,
    DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Closes a DMA channel previously opened for Block mode.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the 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 have been successfully opened using *PlxDmaBlockChannelOpen()*.

The DMA channel cannot be closed by this function If a DMA transfer is currently in-progress. The DMA status is read directly from the DMA status register of the PLX chip. Note that a “crashed” DMA engine reports DMA in-progress. A software reset of the PLX chip may be required in this case. DMA “crashes” are typically a result of invalid addresses provided to the DMA channel.

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


rc = PlxDmaBlockChannelClose(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // Reset the device if a DMA is in-progress
    if (rc == ApiDmaInProgress)
    {
        PlxPciBoardReset();

        // Attempt to close again
        PlxDmaBlockChannelClose(
            hDevice,
            PrimaryPciChannel0
            );
    }
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaBlockChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaBlockChannelOpen(
    HANDLE            hDevice,
    DMA_CHANNEL       dmaChannel,
    DMA_CHANNEL_DESC *dmaChannelDesc
    );
```

PLX Chip Support:

9080, 9054, 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 480.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to open

dmaChannelDesc
Structure containing the parameters used for initializing the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel is not closed

Notes:

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

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

Usage:

```
HANDLE           hDevice;
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear DMA descriptor structure
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

// Setup DMA configuration structure
DmaDesc.EnableReadyInput    = 1;
DmaDesc.DmaStopTransferMode = AssertBLAST;
DmaDesc.DmaChannelPriority  = Rotational;
DmaDesc.IopBusWidth         = 3;            // 32-bit bus

rc = PlxDmaBlockChannelOpen(
         hDevice,
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable open DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-14

PlxDmaBlockTransfer

Syntax:

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

PLX Chip Support:

9080, 9054, 480

Description:

Starts a Block DMA transfer for a given DMA channel.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to use for transfer

dmaDatal
A pointer to the data for the DMA transfer

returnImmediate
If *returnImmediate* is FALSE, the function waits until the DMA transfer has completed. Note that the function exits after a preset timeout, in case the DMA channel has “crashed”. The DMA channel may never report completion in this case.

If *returnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channelis not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel was not opened for Block DMA
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiPciTimeout	No interrupt was received to signal DMA completion

Notes:

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

Before calling this function the appropriate DMA channel must have been 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 an 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 Element	Signification
UserAddr2	Ignored.
LowPciAddr	Lower 32-bits of PCI address of the PCI buffer.
SourceAddr	Local-to-Local Channel2 source address -not used in Flyby mode.
HighPciAddr	High 32-bits of PCI address of the PCI buffer, for chips supporting Dual Address cycles
IopAddr	The IOP address for the transfer
DestAddr	Local-to-Local 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 to transfer.
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Ignored
IopToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored

Usage:

```
HANDLE               hDevice;
PCI_MEMORY           PciBuffer;
DMA_TRANSFER_ELEMENT DmaData;


// Get Common buffer information
rc = PlxPciCommonBufferGet(
         hDevice,
         &PciBuffer
         );
if (rc != ApiSuccess)
{
    // ERROR - Unable to get Common Buffer information
}
```

```
// Fill in DMA transfer parameters
switch (ChipTypeSelected)
{
    case 0x9080:
        DmaData.Pci9080Dma.LowPciAddr        = PciBuffer.PhysicalAddr;
        DmaData.Pci9080Dma.IopAddr           = 0x0;
        DmaData.Pci9080Dma.TransferCount     = 0x1000;
        DmaData.Pci9080Dma.IopToPciDma       = 1;
        DmaData.Pci9080Dma.TerminalCountIntr = 0;
        break;

    case 0x9054:
        DmaData.Pci9054Dma.LowPciAddr        = PciBuffer.PhysicalAddr;
        DmaData.Pci9054Dma.HighPciAddr       = 0x0;
        DmaData.Pci9054Dma.IopAddr           = 0x0;
        DmaData.Pci9054Dma.TransferCount     = 0x1000;
        DmaData.Pci9054Dma.IopToPciDma       = 1;
        DmaData.Pci9054Dma.TerminalCountIntr = 0;
        break;

    case 0x480:
        DmaData.Iop480Dma.LowPciAddr        = PciBuffer.PhysicalAddr;
        DmaData.Iop480Dma.HighPciAddr       = 0x0;
        DmaData.Iop480Dma.IopAddr           = 0x0;
        DmaData.Iop480Dma.TransferCount     = 0x1000;
        DmaData.Iop480Dma.IopToPciDma       = 1;
        DmaData.Iop480Dma.TerminalCountIntr = 0;
        break;
}

rc = PlxDmaBlockTransfer(
       hDevice,
       PrimaryPciChannel0,
       &DmaData,
       FALSE             // Wait for DMA completion
       );
```

```
if (rc != ApiSuccess)
{
    if (rc == ApiPciTimeout)
    {
        // Timed out waiting for DMA completion
    }
    else
    {
        // ERROR - Unable to perform DMA transfer
    }
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_TRANSFER_ELEMENT	4-24

PlxDmaBlockTransferRestart

Syntax:

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

PLX Chip Support:

9080, 9054, 480

Description:

Restarts a Block DMA transfer for a pre-programmed DMA channel. This function initiates a block DMA transfer just like *PlxDmaBlockTransfer()*, except it assumes the DMA transfer parameters are already setup, to provide a slight performance improvement.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
A previously opened and initialized DMA channel

transferSize
Number of bytes to transfer

returnImmediate
If *returnImmediate* is FALSE, the function waits until the DMA transfer has completed. Note that the function exits after a preset timeout, in case the DMA channel has "crashed". The DMA channel may never report completion in this case.

If *returnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel parameter is not supported by this PLX chip
ApiDmaChannelUnavailable	The DMA channel has not been opened for Block DMA
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiPciTimeout	No interrupt was received to signal DMA completion

Notes:

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

Before calling this function the appropriate DMA channel must have been successfully opened using *PlxDmaBlockChannelOpen()* and *PlxDmaBlockTransfer()* must have been successfully called to perform a DMA transfer using the same DMA channel.

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// A previous call to PlxDmaBlockTransfer() is assumed

rc = PlxDmaBlockTransferRestart(
         hDevice,
         PrimaryPciChannel0,
         0x100,              // Provide transfer count
         TRUE                // Don't wait for completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to restart Block DMA
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaControl

Syntax:

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

PLX Chip Support:

9080, 9054, 480

Description:

Controls the DMA engine for a given DMA channel.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
A previously opened DMA channel

dmaCommand
The action to perform on the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelUnavailable	The DMA channel has not been opened
ApiDmaNotPaused	If attempting to resume a DMA channel that is not in a paused state. Also can occur when attempting to pause a Flyby DMA transfer
ApiDmaCommandInvalid	An invalid DMA command

Notes:

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

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

Usage:

```
HANDLE               hDevice;
RETURN_CODE          rc;
DMA_TRANSFER_ELEMENT DmaData;


// Start a DMA transfer
rc = PlxDmaBlockTransfer(
       hDevice,
       PrimaryPciChannel0,
       &DmaData,
       FALSE             // Wait for DMA completion
       );

// Pause the DMA channel
rc = PlxDmaControl(
       hDevice,
       PrimaryPciChannel0,
       DmaPause
       );

if (rc != ApiSuccess)
{
    // ERROR – Unable to pause DMA transfer
}

// Resume the DMA channel
rc = PlxDmaControl(
       hDevice,
       PrimaryPciChannel0,
       DmaResume
       );

if (rc != ApiSuccess)
{
    // ERROR – Unable to resume DMA transfer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_COMMAND	4-21

PlxDmaSglChannelClose

Syntax:

```
RETURN_CODE
PlxDmaSglChannelClose(
    HANDLE      hDevice,
    DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Closes a DMA channel previously opened for Scatter-Gather List (SGL) mode.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel was not opened for Scatter-Gather DMA.
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiDmaPaused	The DMA channel is paused

Notes:

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

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


rc = PlxDmaSglChannelClose(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // Reset the device if a DMA is in-progress
    if (rc == ApiDmaInProgress)
    {
        PlxPciBoardReset();

        // Attempt to close again
        PlxDmaSglChannelClose(
            hDevice,
            PrimaryPciChannel0
            );
    }
    else
    {
        // ERROR - Unable to close DMA channel
    }
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaSglChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaSglChannelOpen(
    HANDLE            hDevice,
    DMA_CHANNEL       dmaChannel,
    DMA_CHANNEL_DESC *dmaChannelDesc
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

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

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to open

dmaChannelDesc
Structure containing the parameters used for initializing the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel is not closed

Notes:

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

If DMA_CHANNEL_DESC 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:

```
HANDLE           hDevice;
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear DMA descriptor structure
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

// Set up DMA configuration structure
DmaDesc.EnableReadyInput    = 1;
DmaDesc.DmaStopTransferMode = AssertBLAST;
DmaDesc.DmaChannelPriority  = Rotational;
DmaDesc.IopBusWidth         = 3;            // 32 bit bus

rc = PlxDmaSglChannelOpen(
         hDevice,
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable open DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-14

PlxDmaSglTransfer

Syntax:

```
RETURN_CODE
PlxDmaSglTransfer(
    HANDLE                hDevice,
    DMA_CHANNEL           dmaChannel,
    DMA_TRANSFER_ELEMENT *dmaData,
    BOOLEAN               returnImmediate
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

This function transfers a user-supplied buffer using the DMA channel. SGL mode of the DMA channel is used, but this is transparent to the application. In a Windows environment, this function works as follows:

- The PLX driver must takes the provided user-mode buffer and page-lock it into memory.
- The buffer is typically scattered throughout memory in non-contiguous pages. As a result, the driver then determines the physical address of each page of memory of the buffer and creates an SGL descriptor for each page. The descriptors are placed into an internal driver allocated buffer.
- The DMA channel is programmed to start at the first descriptor.
- After DMA transfer completion, an interrupt will occur. The driver will then perform all cleanup tasks.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to use for transfer

dmaDatal
A pointer to the data for the DMA transfer

returnImmediate
If *returnImmediate* is FALSE, the function waits until the DMA transfer has completed. Note that the function exits after a preset timeout, in case the DMA channel has “crashed”. The DMA channel may never report completion in this case.

If *returnImmediate* is TRUE, the function exits without waiting for DMA transfer completion.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channelis not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel has not been opened for SGL DMA
ApiDmaInProgress	The DMA transfer is currently in-progress
ApiPciTimeout	No interrupt was received to signal DMA completion
ApiFailed	Failed to start the DMA transfer. This is typically due to insufficient resources to build the SGL list or the driver was unable to page lock the user-mode buffer

Notes:

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

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

If large buffers will be transferred, ensure that there is enough memory to build the SGL.descriptors by increasing the `MaxSglTransferSize` registry entry.

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 Element	Signification
UserAddr	User address of the PCI buffer
LowPciAddr	Ignored
SourceAddr	Ignored
HighPciAddr	Ignored
IopAddr	The Local address for the transfer
DestAddr	Ignored
TransferCount	The number of bytes to transfer
PciSglLoc	Ignored
LastSglElement	Ignored
TerminalCountIntr	Ignored
IopToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored

Usage:

```
U8                  buffer[0x1000];
HANDLE              hDevice;
DMA_TRANSFER_ELEMENT DmaData;
```

```
switch (ChipTypeSelected)
{
    case 0x9080:
        DmaData.Pci9080Dma.UserAddr          = (U32)buffer;
        DmaData.Pci9080Dma.IopAddr           = 0x0;
        DmaData.Pci9080Dma.TransferCount     = sizeof(buffer);
        DmaData.Pci9080Dma.IopToPciDma       = 1;
        DmaData.Pci9080Dma.TerminalCountIntr = 0;
        break;

    case 0x9054:
        DmaData.Pci9054Dma.UserAddr          = (U32)buffer;
        DmaData.Pci9054Dma.IopAddr           = 0x0;
        DmaData.Pci9054Dma.TransferCount     = sizeof(buffer);
        DmaData.Pci9054Dma.IopToPciDma       = 1;
        DmaData.Pci9054Dma.TerminalCountIntr = 0;
        break;

    case 0x480:
        DmaData.Iop480Dma.UserAddr          = (U32)buffer;
        DmaData.Iop480Dma.IopAddr           = 0x0;
        DmaData.Iop480Dma.TransferCount     = sizeof(buffer);
        DmaData.Iop480Dma.IopToPciDma       = 1;
        DmaData.Iop480Dma.TerminalCountIntr = 0;
        break;
}

rc = PlxDmaSglTransfer(
         hDevice,
         PrimaryPciChannel0,
         &DmaData,
         FALSE                  // Wait for completion
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to transfer buffer
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_TRANSFER_ELEMENT	4-24

PlxDmaShuttleChannelClose

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelClose(
    HANDLE      hDevice,
    DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Closes a DMA channel previously opened for Shuttle mode.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel was not opened for Shuttle DMA
ApiDmaInProgress	A DMA transfer is currently in-progress.
ApiDmaPaused	The DMA channel is paused

Notes:

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

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;



rc = PlxDmaShuttleChannelClose(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to close DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDmaShuttleChannelOpen

Syntax:

```
RETURN_CODE
PlxDmaShuttleChannelOpen(
    HANDLE            hDevice,
    DMA_CHANNEL       dmaChannel,
    DMA_CHANNEL_DESC *dmaChannelDesc
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Opens and initializes a DMA channel for Shuttle DMA transfers.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to open

dmaChannelDesc
Structure containing the parameters used for initializing the DMA channel

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiDmaChannelInvalid	The DMA channel is not supported by the PLX chip
ApiDmaChannelUnavailable	The DMA channel is not closed

Notes:

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

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

Usage:

```
HANDLE           hDevice;
RETURN_CODE      rc;
DMA_CHANNEL_DESC DmaDesc;


// Clear DMA descriptor structure
memset(
    &DmaDesc,
    0,
    sizeof(DMA_CHANNEL_DESC)
    );

// Set up DMA configuration structure
DmaDesc.EnableReadyInput    = 1;
DmaDesc.DmaStopTransferMode = AssertBLAST;
DmaDesc.DmaChannelPriority  = Rotational;
DmaDesc.IopBusWidth         = 3;           // 32 bit bus

rc = PlxDmaShuttleChannelOpen(
         hDevice,
         PrimaryPciChannel0,
         &DmaDesc
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable open DMA channel
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_CHANNEL_DESC	4-24

PlxDmaShuttleTransfer

Syntax:

```
RETURN_CODE
PlxDmaShuttleTransfer(
    HANDLE                hDevice,
    DMA_CHANNEL           dmaChannel,
    DMA_TRANSFER_ELEMENT *dmaData
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

This function continuously transfers a user-supplied buffer using the DMA channel. SGL mode of the DMA channel is used, but this is transparent to the application. In a Windows environment, this function works as follows:

- The PLX driver must takes the provided user-mode buffer and page-lock it into memory.
- The buffer is typically scattered throughout memory in non-contiguous pages. As a result, the driver then determines the physical address of each page of memory of the buffer and creates an SGL descriptor for each page. The descriptors are placed into an internal driver allocated buffer. The last descriptor is setup to point back to the first descriptor (Shuttle mode)
- The DMA channel is programmed to start at the first descriptor.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
The DMA channel to use for transfer

dmaDatal
A pointer to the data for the DMA transfer

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	One or more parameters is NULL
ApiNullParam	The PLX device is in a power state that is lower than required for this function
ApiPowerDown	The DMA channelis not supported by the PLX chip
ApiDmaChannelInvalid	The DMA channel was not opened for SGL DMA
ApiDmaChannelUnavailable	The DMA channel has not been opened for Shuttle DMA
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiFailed	Failed to start the DMA transfer. This is typically due to insufficient resources to build the SGL list or the driver was unable to page lock the user-mode buffer

Notes:

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

Before calling this function the appropriate DMA channel must have been 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.

If large buffers will be transferred, ensure that there is enough memory to build the SGL.descriptors by increasing the `MaxSglTransferSize` registry entry.

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 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. (0 = PCI-to-Local, 1 = Local-to-PCI)
NextSglPtr	Ignored.

Usage:

```
U8                  buffer[0x1000];
HANDLE              hDevice;
DMA_TRANSFER_ELEMENT DmaData;


switch (ChipTypeSelected)
{
    case 0x9080:
        DmaData.Pci9080Dma.UserAddr          = (U32)buffer;
        DmaData.Pci9080Dma.IopAddr           = 0x0;
        DmaData.Pci9080Dma.TransferCount     = sizeof(buffer);
        DmaData.Pci9080Dma.IopToPciDma       = 1;
        DmaData.Pci9080Dma.TerminalCountIntr = 0;
        break;
```

```
        case 0x9054:
            DmaData.Pci9054Dma.UserAddr          = (U32)buffer;
            DmaData.Pci9054Dma.IopAddr           = 0x0;
            DmaData.Pci9054Dma.TransferCount     = sizeof(buffer);
            DmaData.Pci9054Dma.IopToPciDma       = 1;
            DmaData.Pci9054Dma.TerminalCountIntr = 0;
            break;

        case 0x480:
            DmaData.Iop480Dma.UserAddr          = (U32)buffer;
            DmaData.Iop480Dma.IopAddr           = 0x0;
            DmaData.Iop480Dma.TransferCount     = sizeof(buffer);
            DmaData.Iop480Dma.IopToPciDma       = 1;
            DmaData.Iop480Dma.TerminalCountIntr = 0;
            break;
    }

    rc = PlxDmaShuttleTransfer(
             hDevice,
             PrimaryPciChannel0,
             &DmaData
             );

    if (rc != ApiSuccess)
    {
        // ERROR – Unable to initiate DMA transfer
    }
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19
DMA_TRANSFER_ELEMENT	4-24

PlxDmaStatus

Syntax:

```
RETURN_CODE
PlxDmaStatus(
    HANDLE      hDevice,
    DMA_CHANNEL dmaChannel
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Returns the status of the DMA engine for a specified DMA channel.

Parameters:

hDevice
Handle of an open PCI device

dmaChannel
A previously opened DMA channel

Return Codes:

Code	Description
ApiInvalidHandle	The function was passed an invalid device handle
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaChannelUnavailable	The DMA channel has not been opened
ApiDmaDone	The DMA channel is done/ready
ApiDmaPaused	The DMA channel is paused
ApiDmaInProgress	A DMA transfer is currently in-progress

Notes:

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

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;

// Start a DMA transfer
rc = PlxDmaShuttleTransfer(
         hDevice,
         PrimaryPciChannel0,
         &DmaData
         );

// Get DMA status
rc = PlxDmaStatus(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiDmaInProgress)
{
    // ERROR - DmaInProgress not returned
}

// Pause the DMA channel
rc = PlxDmaControl(
         hDevice,
         PrimaryPciChannel0,
         DmaPause
         );

// Get DMA status
rc = PlxDmaStatus(
         hDevice,
         PrimaryPciChannel0
         );

if (rc != ApiDmaPaused)
{
    // ERROR - DmaPaused not returned
}
```

Cross Reference:

Referenced Item	Page
DMA_CHANNEL	4-19

PlxDriverVersion

Syntax:

```
RETURN_CODE
PlxDriverVersion(
    HANDLE  hDevice,
    U8     *VersionMajor,
    U8     *VersionMinor,
    U8     *VersionRevision
    );
```

PLX Chip Support:

All

Description:

Returns the version information of the PLX driver for the selected PCI device.

Parameters:

hDevice
Handle of an open PCI device

VersionMajor
A pointer to an 8-bit buffer to contain the Major version number

VersionMinor
A pointer to an 8-bit buffer to contain the Minor version number

VersionRevision
A pointer to an 8-bit buffer to contain the Revision version number

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL

Notes:

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

Usage:

```
U8          DriverMajor;
U8          DriverMinor;
U8          DriverRevision;
RETURN_CODE rc;


rc = PlxDriverVersion(
         &DriverMajor,
         &DriverMinor,
         &DriverRevision
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get Driver version information
}
else
{
    printf("PLX Driver Version = %d.%d%d\n",
           DriverMajor, DriverMinor, DriverRevision);
}
```

PlxHotSwapIdRead

Syntax:

```
U8
PlxHotSwapIdRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Hot Swap Capability ID

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Usage:

```
U8          HotSwapId;
HANDLE      hDevice;
RETURN_CODE rc;


HotSwapId = PlxHotSwapIdRead(
                hDevice,
                &rc
                );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read Hot Swap Capability ID
}
```

PlxHotSwapNcpRead

Syntax:

```
U8
PlxHotSwapNcpRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Next Capability Pointer from the Hot Swap register.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiHSNotSupported	Hot Swap is either disabled or not supported by the PLX device

Usage:

```
U8          NextCapability;
HANDLE      hDevice;
RETURN_CODE rc;


NextCapability = PlxHotSwapNcpRead(
                     hDevice,
                     &rc
                     );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read Hot Swap NCP
}
```

PlxHotSwapStatus

Syntax:

```
U8
PlxHotSwapStatus(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Hot-Swap status of the PLX chip.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

If there is no error, the return value is an OR'ed combination of the following:

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          status;
HANDLE      hDevice;
RETURN_CODE rc;


status = PlxHotSwapStatus(
             hDevice,
             &rc
             );

if (rc != ApiSuccess || status = 0xff)
{
    // ERROR: Unable to read Hot Swap status
}

if (rc == ApiHSNotSupported)
{
    // ERROR – Hot Swap is disabled or not supported by the device
}

if (status & HS_LED_ON)
{
    // Hot Swap LED is on
}

if (status & HS_BOARD_REMOVED)
{
    // Hot Swap – board requesting extraction
}

if (status & HS_BOARD_INSERTED)
{
    // Hot Swap – board requesting insertion
}
```

PlxIntrAttach

Syntax:

```
RETURN_CODE
PlxIntrAttach(
    HANDLE    hDevice,
    PLX_INTR  intrTypes,
    HANDLE   *pEventHdl
    );
```

PLX Chip Support:

All

Description:

This function is used by applications when they need to wait for a specific interrupt(s) to occur. The function registers the event and the desired interrupt source(s). The behavior of this function has changed from previous SDK releases. Please refer to the *Notes* section for further important information. The function performs the following tasks:

- The API keeps an internal list of all event handles this function allocates. It first checks if the passed event handle is in the list. If not, it allocates a new handle and some additional memory for driver communications overhead.
- The API sends a message to the driver to register the interrupt source(s) and the associated event handle.
- The API call returns the handle to the application, which is now free to wait for the event.
- Once an interrupt occurs, the application can re-attach, but should use the same event handle.

Parameters:

hDevice
Handle of an open PCI device

intrTypes
Structure containing the sources of interrupts which the application would like to be notified of. An event will occur if ANY one of the registered interrupts occurs.

pEventHdl
A pointer to a handle object which will receive the event to wait for

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInsufficientResources	Not enough memory to allocate a new event handle

Notes:

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

Due to the asynchronous behavior of the function and limitations of its original design, its behavior has changed and it should be used according to the guidelines below. Since the PLX API must register an I/O event with the driver, the memory used for the I/O operation MUST remain persistent in memory, or exceptions will occur. After returning control to the application, the API function has no way of knowing when the interrupt event occurs, so it cannot free the I/O operation memory after the interrupt event.

As a result, the function now internally allocates memory when needed to register an event, which is associated with the event handle. Every time the function is called, it will search for a match with the passed event handle in an attempt to re-use the pre-allocated memory; otherwise, a new set of resources is allocated. The following guidelines should be followed when using this function:

- Never call *CloseHandle()* to release the handle created by this function. When an application terminates, the API will release all memory created as a result of use of this function.
- Memory for the event handle pointer must remain persistent. The handle pointer should either be declared as *static* or should be allocated manually with a call to *malloc()*. If not, the API function will continue to allocate resources and the system may crash.
- Do not re-attach with an event handle in which the interrupt event has not yet occurred.
- To make the best use of memory, applications should re-use an event handle, if possible, when re-attaching, provided the event has been signaled already.

For compatibility with previous SDK versions, this function was not changed to overcome its limitations. Note that it will be modified or replaced by one or more functions in a future release of the SDK.

Usage:

```
DWORD        EventStatus;
HANDLE       hDevice;
HANDLE      *pDmaEvent;
PLX_INTR     IntSources;
RETURN_CODE  rc;


// Clear interrupt sources
memset(
    &IntSources,
    0,
    sizeof(PLX_INTR)
    );

// Allocate memory for Event Handle pointer
pDmaEvent = (HANDLE *)malloc(sizeof(HANDLE);

// Register for DMA Channel 1 interrupt notification
IntSources.PciDmaChannel1 = 1;
```

```
rc = PlxIntrAttach(
         hDevice,
         IntSources,
         pDmaEvent
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to register interrupt notification
}

/*************************************
 * Perform DMA transfer here.
 * Refer to DMA functions documentation
 *************************************/

// Wait for the DMA interrupt
EventStatus =
    WaitForSingleObject(
        *pDmaEvent,
        10 * 1000           // 10 second timeout
        );

switch (EventStatus)
{
    case WAIT_OBJECT_0:
        // DMA Interrupt occurred – re-attach with same handle
        PlxIntrAttach(
            hDevice,
            IntSources,
            pDmaEvent    // Same handle, API will not re-allocate
            );
        break;

    case WAIT_TIMEOUT:
        // ERROR - Timeout waiting for Interrupt Event
        break;

    case WAIT_FAILED:
        // ERROR - Failed while waiting for interrupt
        break;
}
```

```
    // Example of calling a function to wait for an event
    while (1)
    {
        WaitForDoorBellEvent(
            hDevice
            );
    }


void
WaitForDoorBellEvent(
    HANDLE hDevice
    )
{
    PLX_INTR      IntSources;
    static HANDLE pEvent;   // Declared as static so it is not de-allocated
                            // when the function exits

    // Clear interrupt sources
    memset(
        &IntSources,
        0,
        sizeof(PLX_INTR)
        );

    // Register for Local-to-PCI Doorbell interrupt notification
    IntSources.PciDoorbell = 1;
    PlxIntrAttach(
        hDevice,
        IntSources,
        &pEvent
        );

    // Wait for the Doorbell interrupt
    WaitForSingleObject(
        *pEvent,
        INFINITE            // Wait forever
        );
    ....
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIntrDisable

Syntax:

```
RETURN_CODE
PlxIntrDisable(
    HANDLE    hDevice,
    PLX_INTR *plxIntr
    );
```

PLX Chip Support:

All

Description:

Disables specific interrupt(s) of a PCI device containing a PLX chip.

Parameters:

hDevice
Handle of an open PCI device

plxIntr
A pointer to the interrupt structure specifying the interrupts to disable

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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      hDevice;
PLX_INTR    PlxIntr;
RETURN_CODE rc;



// Clear interrupt structure
memset(
    &PlxIntr,
    0,
    sizeof(PLX_INTR)
    );

// Set interrupts to disable
PlxIntr.InboundPost    = 1;   // Messaging Unit Inbound Post
PlxIntr.OutboundPost   = 1;   // Messaging Unit Outbound Post
PlxIntr.IopDmaChannel0 = 1;   // Local DMA Channel 0
PlxIntr.PciDmaChannel0 = 1;   // PCI DMA Channel 0
PlxIntr.PciDmaChannel1 = 1;   // PCI DMA Channel 1


rc = PlxIntrDisable(
         hDevice,
         &PlxIntr
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to disable interrupts
}
```

Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIntrEnable

Syntax:

```
RETURN_CODE
PlxIntrEnable(
    HANDLE    hDevice,
    PLX_INTR *plxIntr
    );
```

PLX Chip Support:

All

Description:

Enables specific interrupt(s) of a PCI device containing a PLX chip.

Parameters:

hDevice
Handle of an open PCI device

plxIntr
A pointer to the interrupt structure specifying the interrupts to enable

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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      hDevice;
PLX_INTR    PlxIntr;
RETURN_CODE rc;



// Clear interrupt structure
memset(
    &PlxIntr,
    0,
    sizeof(PLX_INTR)
    );

// Set interrupts to enable
PlxIntr.InboundPost    = 1;   // Messaging Unit Inbound Post
PlxIntr.OutboundPost   = 1;   // Messaging Unit Outbound Post
PlxIntr.IopDmaChannel0 = 1;   // Local DMA Channel 0
PlxIntr.PciDmaChannel0 = 1;   // PCI DMA Channel 0
PlxIntr.PciDmaChannel1 = 1;   // PCI DMA Channel 1


rc = PlxIntrEnable(
         hDevice,
         &PlxIntr
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to enable interrupts
}
```

Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIntrStatusGet

Syntax:

```
RETURN_CODE
PlxIntrStatusGet(
    HANDLE    hDevice,
    PLX_INTR *plxIntr
    );
```

PLX Chip Support:

All

Description:

Returns the status of the most recent interrupt(s) to occur.

Parameters:

hDevice
Handle of an open PCI device

plxIntr
A pointer to the interrupt structure which will contain the last active interrupt(s)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL

Notes:

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

Usage:

```
HANDLE      hDevice;
PLX_INTR    PlxIntr;
RETURN_CODE rc;


rc = PlxIntrStatusGet(
         hDevice,
         &PlxIntr
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get interrupt status
}

if (PlxIntr.OutboundOverflow == 1);
{
    // Messaging Unit Outbound Overflow interrupt was active
}

if (PlxIntr.PciDmaChannel0 == 1);
{
    // PCI DMA Channel 0 interrupt was active
}

if (PlxIntr.PciDoorbell == 1);
{
    // Local-to-PCI interrupt was active
}

if (PlxIntr.IopToPciInt == 1);
{
    // Local-to-PCI interrupt was active
}
```

Cross Reference:

Referenced Item	Page
PLX_INTR	4-54

PlxIoPortRead

Syntax:

```
RETURN_CODE
PlxIoPortRead(
    HANDLE       hDevice,
    U32          address,
    ACCESS_TYPE  bits,
    VOID        *pOutData
    );
```

***Note**: **PlxIoPortRead()** is obsolete and exported only to support existing applications. Applications should us the C-code **inp()(/outp()** I/O access functions (or equivalent) instead.*

PLX Chip Support:

All

Description:

Reads a value from an I/O port.

Parameters:

hDevice
Handle of an open PCI device

address
The I/O port address to read from

bits
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

pOutData
A pointer to a buffer to contain the data read from the I/O port

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidAccessType	An invalid or unsupported ACCESS_TYPE parameter

Notes:

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

This function performs a standard I/O port access. It is recommended that the C functions *inp()* or *oupt()* or equivalent be used instead to perform I/O port accesses. These will perform a direct access rather than use the PLX driver and will result in improved performance.

Usage:

```
U32         port;
U32         RegValue;
HANDLE      hDevice;
RETURN_CODE rc;


// Registers of some PLX devices are accessible through BAR 1 I/O space
port = PlxPciConfigRegisterRead(
           Device.bus,
           Device.slot,
           CFG_BAR1,
           &rc
           );

// Clear bit 0, which is I/O space enable
port = port & ~(1 << 0);

// Read a PLX register
rc = PlxIoPortRead(
         hDevice,
         port + 0x34,      // Read local register 34h
         BitSize32,
         &RegValue
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read I/O port
}
```

Cross Reference:

Referenced Item	Page
ACCESS_TYPE	4-9

PlxIoPortWrite

Syntax:

```
RETURN_CODE
PlxIoPortWrite(
    HANDLE       hDevice,
    U32          address,
    ACCESS_TYPE  bits,
    VOID        *pValue
    );
```

***Note**: **PlxIoPortWrite()** is obsolete and exported only to support existing applications. Applications should us the C-code **inp()(/outp()** I/O access functions (or equivalent) instead.*

PLX Chip Support:

All

Description:

Writes a value to an I/O port.

Parameters:

hDevice
Handle of an open PCI device

address
The I/O port address to read from

bits
Determines the size of each unit of data accessed: 8, 16, or 32-bit.

pValue
A pointer to the data to write to the I/O port

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The device is in a power state that is lower than required for this function
ApiInvalidAccessType	An invalid or unsupported ACCESS_TYPE parameter

Notes:

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

This function performs a standard I/O port access. It is recommended that the C functions *inp()* or *oupt()* or equivalent be used instead to perform I/O port accesses. These will perform a direct access rather than use the PLX driver and will result in improved performance.

Usage:

```
U32         port;
U32         RegValue;
HANDLE      hDevice;
RETURN_CODE rc;


// Registers of some PLX devices are accessible through BAR 1 I/O space
port = PlxPciConfigRegisterRead(
           Device.bus,
           Device.slot,
           CFG_BAR1,
           &rc
           );

// Clear bit 0, which is I/O space enable
port = port & ~(1 << 0);

// Prepare write value
RegValue = 0x00300024;

// Write to a PLX register
rc = PlxIoPortWrite(
         hDevice,
         port + 0x34,      // Write to local register 34h
         BitSize32,
         &RegValue
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read I/O port
}
```

Cross Reference:

Referenced Item	Page
ACCESS_TYPE	4-9

PlxMuHostOutboundIndexRead

Syntax:

```
U32
PlxMuHostOutboundIndexRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

480

Description:

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

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiPowerDown	The 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 for this function to work properly.

Usage:

```
U32         IndexValue;
HANDLE      hDevice;
RETURN_CODE rc;


// Read the Outbound Index value
IndexValue =
    PlxMuHostOutboundIndexRead(
        hDevice,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read Outbound Index
}
```

PlxMuHostOutboundIndexWrite

Syntax:

```
RETURN_CODE
PlxMuHostOutboundIndexWrite(
    HANDLE hDevice,
    U32    indexValue
    );
```

PLX Chip Support:

480

Description:

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

Parameters:

hDevice
Handle of an open PCI device

indexValue
The value to write to the index register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiPowerDown	The 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 for this function to work properly.

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// Write the Outbound Index value
rc = PlxMuHostOutboundIndexWrite(
         hDevice,
         0x100
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write the Outbound Index
}
```

PlxMuInboundPortRead

Syntax:

```
RETURN_CODE
PlxMuInboundPortRead(
    HANDLE  hDevice,
    U32    *framePointer
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Reads the Messaging Unit Inbound Port to retrieve the next item from the Inbound Free Queue.

Parameters:

hDevice
Handle of an open PCI device

framePointer
A pointer to a 32-bit buffer to contain the returned value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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 for this function to work properly.

Usage:

```
U32         Frame;
HANDLE      hDevice;
RETURN_CODE rc;


// Read inbound port
rc = PlxMuInboundPortRead(
         hDevice,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read from the Inbound Port
}

// Check for an empty queue
if (Frame == (U32)-1)
{
    // ERROR – Inbound Free queue is empty
}
```

PlxMuInboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuInboundPortWrite(
    HANDLE  hDevice,
    U32    *framePointer
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Writes to the Messaging Unit Inbound Port to add an item to the Inbound Post queue.

Parameters:

hDevice
Handle of an open PCI device

framePointer
A pointer to a 32-bit buffer containing the value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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 for this function to work properly.

Usage:

```
U32         Frame;
HANDLE      hDevice;
RETURN_CODE rc;


// Prepare value
Frame = 0x0100;

// Write to the inbound port
rc = PlxMuInboundPortWrite(
         hDevice,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to the Inbound Port
}
```

PlxMuOutboundPortRead

Syntax:

```
RETURN_CODE
PlxMuOutboundPortRead(
    HANDLE  hDevice,
    U32    *framePointer
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Reads the Messaging Unit Outbound Port to retrieve the next item from the Outbound Post Queue.

Parameters:

hDevice
Handle of an open PCI device

framePointer
A pointer to a 32-bit buffer to contain the returned value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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 for this function to work properly.

Usage:

```
U32         Frame;
HANDLE      hDevice;
RETURN_CODE rc;


// Read outbound port
rc = PlxMuOutboundPortRead(
         hDevice,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read from the outbound Port
}

// Check for an empty queue
if (Frame == (U32)-1)
{
    // ERROR – Outbound Free queue is empty
}
```

PlxMuOutboundPortWrite

Syntax:

```
RETURN_CODE
PlxMuOutboundPortWrite(
    HANDLE  hDevice,
    U32    *framePointer
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Writes to the Messaging Unit Outbound Port to add an item to the Outbound Free Queue.

Parameters:

hDevice
Handle of an open PCI device

framePointer
A pointer to a 32-bit buffer containing the value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters 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 for this function to work properly.

Usage:

```
U32         Frame;
HANDLE      hDevice;
RETURN_CODE rc;


// Prepare value
Frame = 0x0100;

// Write to the outbound port
rc = PlxMuOutboundPortWrite(
         hDevice,
         &Frame
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to the outbound Port
}
```

PlxPciAbortAddrRead

Syntax:

```
U32
PlxPciAbortAddrRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

480

Description:

Returns the starting PCI address of the operation that caused the PCI Abort.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function

Usage:

```
U32         AbortAddr;
HANDLE      hDevice;
RETURN_CODE rc;

AbortAddr = PlxPciAbortAddrRead(
                hDevice,
                &rc
                );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read PCI abort address
}
```

PlxPciBarRangeGet

Syntax:

```
RETURN_CODE
PlxPciBarRangeGet(
    HANDLE  hDevice,
    U32     barRegisterNumber,
    U32    *data
    );
```

PLX Chip Support:

All

Description:

Retrieves the size of any PCI-to-Local space window.

Parameters:

hDevice
Handle of an open PCI device

barRegisterNumber
The PCI base address register (BAR) number

data
A pointer to a 32-bit buffer which will contain the size of the space

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidRegister	The BAR register is invalid

Notes:

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

Usage:

```
U32         size;
U32         PciBarNum;
RETURN_CODE rc;


for (PciBarNum = 0; PciBarNum < 6; PciBarNum++)
{
    rc = PlxPciBarRangeGet(
             hDevice,
             PciBarNum,
             &size
             );

    if (rc != ApiSuccess)
    {
        // ERROR – Unable to get PCI space size
    }
    else
    {
        printf("BAR %d:  %d bytes",
               BarRegNum, size);
    }
}
```

PlxPciBaseAddressesGet

Syntax:

```
RETURN_CODE
PlxPciBaseAddressesGet(
    HANDLE             hDevice,
    VIRTUAL_ADDRESSES *virtAddr
    );
```

PLX Chip Support:

All

Description:

Gets the user-mode virtual addresses for the PCI-to-Local spaces. These can then be used to directly access local bus devices.

Parameters:

hDevice
Handle of an open PCI device

virtAddr
A pointer to a structure which will contain the virtual addresses

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL

Notes:

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

When the PLX device driver starts, it attempts map any valid PCI-to-Local space into Windows kernel memory space. When the function *PlxPciBaseAddressesGet()* is called, the driver will attempt to map the valid spaces into the calling application's user space.

All mappings, however, use up valuable Operating System Page Table Entries (PTE) and, as a result, some mappings may be unsuccessful due to insufficient resources. This is typically governed by the amount of system memory, a Windows registry entry (HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\SystemPages), and the size of the PCI space(s). Large PCI spaces, such as 16 or 32 MB, risk failed mappings due to the large number of PTEs required to map the space.

Most OS vendors do not recommend the practice of mapping PCI space directly into user application space. This is considered a breach of system security and the OS will be unable to prevent system crashes from a misbehaving application. There is no guarantee that PLX will provide this functionality in future SDK releases.

If this function successfully returns valid virtual addresses, these can be used to directly access local bus devices instead of the *PlxIopBusRead()* and *PlxIopBusWrite()* functions. Direct access will bypass the driver and result in maximum performance.

Note that it becomes the application's responsibility to make sure that the space is setup correctly before attempting to access any local bus devices. This includes, correct mapping of the PCI-to-Local space window and any bus region descriptors.

WARNING: Attempts to access invalid local bus devices or locations will result in system crashes.

Usage:

```
HANDLE            hDevice;
RETURN_CODE       rc;
VIRTUAL_ADDRESSES Va;


rc = PlxPciBaseAddressesGet(
         hDevice,
         &Va
         );

if (rc != ApiSuccess)
{
    // ERROR - Unable to get virtual addresses
}
else
{
    printf("BAR 0 VA:  0x%08x\n", Va.Va0);
    printf("BAR 1 VA:  0x%08x\n", Va.Va1);
    printf("BAR 2 VA:  0x%08x\n", Va.Va2);
    printf("BAR 3 VA:  0x%08x\n", Va.Va3);
    printf("BAR 4 VA:  0x%08x\n", Va.Va4);
    printf("BAR 5 VA:  0x%08x\n", Va.Va5);
    printf("EROM VA :  0x%08x\n", Va.VaRom);
}

// All PLX devices' local registers are accessible through BAR 0

// Read a Local register of the PLX chip
if (Va.Va0 != (U32)-1 && Va.Va0 != 0)
{
    RegValue = *(U32*)(Va.Va0 + 0x24);   // Read register at offset 0x24
}
```

```
/************************************************
 * Setup the space for access to local bus. If
 * the device is a 9054, for example, Space 0
 * can be used, which is actually at PCI BAR 2.
 * The PCI-to-Local space window must be adjusted
 * to allow access to the desired local bus address.
 *
 *        -- This is not shown here --
 ************************************************/

// Read from local bus memory (assuming Space 0 of 9054)

if (Va.Va2 == (U32)-1 && Va.Va2 == 0)
{
    // ERROR – Space 0 was not mapped or is invalid
}

// Read an 8-bit value from offset 0x100
(U8)Value = *(U8*)(Va.Va2 + 0x100);

// Read a 32-bit value from offset 0x250
(U8)Value = *(U32*)(Va.Va2 + 0x250);
```

Cross Reference:

Referenced Item	Page
VIRTUAL_ADDRESSES	4-69

PlxPciBoardReset

Syntax:

```
VOID
PlxPciBoardReset(
    HANDLE hDevice
    );
```

PLX Chip Support:

All

Description:

Performs a reset of a PCI device containing a PLX chip.

Parameters:

hDevice
Handle of an open PCI device

Return Codes:

None

Notes:

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

Usage:

```
HANDLE hDevice;


// A local bus device is crashed, reset the board
PlxPciBoardReset(
    hDevice
    );
```

PlxPciBusSearch

Syntax:

```
RETURN_CODE
PlxPciBusSearch(
    DEVICE_LOCATION *pDevData
    );
```

***Note**: **PlxPciBusSearch()** is obsolete and exported only to support existing applications. Applications should us the **PlxPciDeviceFind()** function instead.*

PLX Chip Support:

All

Description:

Searches for a PLX device on the PCI bus. When the function returns, the DEVICE_LOCATION structure will contain the information for the specified device. *PlxPciBusSearch()* queries all loaded PLX drivers for the first device matching the search data. If the device is found, the search data is completed. Otherwise *ApiInvalidDeviceInfo* is returned.

Parameters:

pDevData

A pointer to a DEVICE_LOCATION structure which provides the search data for the desired device. Items not used as search criteria and will be filled in if the device is found.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	No PCI device matched was found that matched the search criteria
ApiNoActiveDriver	There is no device driver installed into the system

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;


// Search by Device/Vendor ID
Device.VendorId        = 0x10b5;
Device.DeviceId        = 0x5401;
Device.BusNumber       = (U32)-1;
Device.SlotNumber      = (U32)-1;
Device.SerialNumber[0] = '\0';      // Set Serial # to empty string

rc = PlxPciBusSearch(
         &Device
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to search for device or device not found
}
else
{
    // Device found matching criteria
    printf("Device ID    =  %04x\n", Device.DeviceId);
    printf("Vendor ID    =  %04x\n", Device.VendorId);
    printf("Bus Number   =  %02x\n", Device.BusNumber);
    printf("Slot Number  =  %02x\n", Device.SlotNumber);
    printf("SerialNumber = %s\n",    Device.SerialNumber);
}
```

Cross Reference:

Referenced Item	Page
DEVICE_LOCATION	4-13

PlxPciCommonBufferGet

Syntax:

```
RETURN_CODE
PlxPciCommonBufferGet(
    HANDLE      hDevice,
    PCI_MEMORY *pMemoryInfo
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Provides the information about the Common buffer allocated by a PLX driver.

Parameters:

hDevice
Handle of an open PCI device

pMemoryInfo
A pointer to a structure which will contain the buffer information.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL

Notes:

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

PLX device drivers attempt to allocate a page-locked contiguous buffer in system memory for use by all PLX applications and PCI devices. Typically, applications use this buffer for block DMA transfers since a physical address is required when programming a DMA channel.

The PLX chip's DMA engine requires physical addresses when transferring to/from PCI. Conversely, Windows applications must 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 the PLX driver cannot guarantee allocation of larger buffers. Non-paged contiguous memory is a valuable system resource. The allocated buffer may be significantly smaller than the desired size.

Note that the PLX driver allocates only one buffer and does not control its use. If multiple applications and/or PCI devices attempt to share this buffer, it is the applications' responsibility to implement synchronization mechanisms or use exclusive regions of the buffer.

Usage:

```
U8         i;
HANDLE     hDevice;
PCI_MEMORY PciBuffer;


// Get Common buffer information
rc = PlxPciCommonBufferGet(
         hDevice,
         &PciBuffer
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get Common buffer information
}

if (PciBuffer.PhysicalAddr == 0 || PciBuffer.PhysicalAddr == (U32)-1 ||
    PciBuffer.UserAddr     == 0 || PciBuffer.UserAddr     == (U32)-1)
{
    // ERROR – Common buffer not allocated or mapped to user space
}

// Fill in the buffer with some data to transfer
for (i = 0; i < 0x80; i++)
{
    *(U8*)(PciBuffer.UserAddr + i) = i;
}

// Setup a block DMA to transfer the data (assuming a 9054 device)
DmaData.Pci9054Dma.LowPciAddr        = PciBuffer.PhysicalAddr;
DmaData.Pci9054Dma.HighPciAddr       = 0x0;
DmaData.Pci9054Dma.IopAddr           = 0x00100000;
DmaData.Pci9054Dma.TransferCount     = 0x80;
DmaData.Pci9054Dma.IopToPciDma       = 0;
DmaData.Pci9054Dma.TerminalCountIntr = 0;

// Transfer the data (refer to DMA block transfer function)
PlxDmaBlockTransfer(....);
```

Cross Reference:

Referenced Item	Page
PCI_MEMORY	4-51

PlxPciConfigRegisterRead

Syntax:

```
U32
PlxPciConfigRegisterRead(
    U32          bus,
    U32          slot,
    U32          registerNumber,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

All

Description:

Returns the value of a PCI configuration register of a PCI device.

Parameters:

bus
The PCI bus number of the device to read

slot
The PCI slot number of the device to read

registerNumber
Offset of the PCI configuration register read

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidRegister	The register offset parameter is out of range or not aligned on a 4-byte boundary
ApiConfigAccessFailed	The specified device either does not exist or the PCI configuration access failed

Notes:

With the advent of Plug 'n' Play, access to PCI configuration registers of any arbitrary device is not allowed, unless unsupported and possibly dangerous system by-pass techniques are used. Since Windows NT 4.0 does not support PnP, PCI registers of any device can be accessed. In newer Operating Systems, such as Windows 98/2000, drivers are not allowed to randomly access the PCI configuration space of any arbitrary device.

As a result, in PnP environments, this function can only access PCI devices for which PLX device drivers have been loaded.

Usage:

```
U8              bus;
U8              slot;
U32             RegValue;
RETURN_CODE     rc;
DEVICE_LOCATION Device;


// DEVICE_LOCATION is assumed to be filled in

// Read the Subsystem Device/Vendor ID
RegValue =
    PlxPciConfigRegisterRead(
        Device.BusNumber,
        Device.SlotNumber,
        CFG_SUB_VENDOR_ID,
        &rc
        );

// Scan for all PCI devices (only NT 4.0 systems will report all devices)
for (bus = 0; bus < 32; bus++)
{
    for (slot = 0; slot < 32; slot+)
    {
        // Read the Device/Vendor ID
        RegValue =
            PlxPciConfigRegisterRead(
                bus,
                slot,
                CFG_VENDOR_ID,
                &rc
                );

        if (rc == ApiSuccess)
        {
            // Found a valid PCI device
            printf("Device ID: %08x  [bus %02x  slot %02x]\n",
                   RegValue, bus, slot);
        }
    }
}
```

PlxPciConfigRegisterReadAll

Syntax:

```
RETURN_CODE
PlxPciConfigRegisterReadAll(
    U32  bus,
    U32  slot,
    U32 *buffer
    );
```

PLX Chip Support:

All

Description:

Reads all PCI Configuration registers of a PCI device.

Parameters:

bus
The PCI bus number of the device to read

slot
The PCI slot number of the device to read

buffer
A pointer to a buffer large enough to contain all PCI configuration register values

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiConfigAccessFailed	The specified device either does not exist or the PCI configuration access failed

Notes:

The buffer MUST be large enough to contain all PCI Configuration registers. The following table provides the minimum sizes.

PLX Chip	Minimum Buffer size
9080, 9050, 9052	64 bytes
9054, 9030	84 bytes
480	96 bytes

With the advent of Plug ‘n’ Play, access to PCI configuration registers of any arbitrary device is not allowed, unless unsupported and possibly dangerous system by-pass techniques are used. Since Windows NT 4.0 does not support PnP, PCI registers of any device can be accessed. In newer Operating Systems, such as Windows 98/2000, drivers are not allowed to randomly access the PCI configuration space of any arbitrary device.

As a result, in PnP environments, this function can only access PCI devices for which PLX device drivers have been loaded.

Usage:

```
U8          PciRegs[0x100];   // Large enough for all PCI registers
RETURN_CODE rc;


// Read PCI registers of device at a specified location
rc = PlxPciConfigRegisterReadAll(
         0x0,
         0x12,
         (U32*)PciRegs
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read PCI Configuration registers
}
```

PlxPciConfigRegisterWrite

Syntax:

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

PLX Chip Support:

All

Description:

Writes data to a configuration register on a PCI device.

Parameters:

bus
The PCI bus number of the device to write

slot
The PCI slot number of the device to write

registerNumber
Offset of the PCI configuration register write

data
A pointer to a 32-bit buffer which contains the data to write.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidRegister	The register offset parameter is out of range or not aligned on a 4-byte boundary
ApiNullParam	One or more parameters is NULL
ApiConfigAccessFailed	The specified device either does not exist or the PCI configuration access failed

Notes:

With the advent of Plug 'n' Play, access to PCI configuration registers of any arbitrary device is not allowed, unless unsupported and possibly dangerous system by-pass techniques are used. Since Windows NT 4.0 does not support PnP, PCI registers of any device can be accessed. In newer Operating Systems, such as Windows 98/2000, drivers are not allowed to randomly access the PCI configuration space of any arbitrary device.

As a result, in non-NT 4.0 environments, this function can only access devices for which PLX device drivers have been loaded.

Usage:

```
U8              bus;
U8              slot;
U32             RegValue;
RETURN_CODE     rc;
DEVICE_LOCATION Device;


// DEVICE_LOCATION is assumed to be filled in

// Read the PCI Command/Status register
RegValue =
    PlxPciConfigRegisterRead(
        Device.BusNumber,
        Device.SlotNumber,
        CFG_COMMAND,
        &rc
        );

// Check for any Errors or Aborts
if (RegValue & 0xf8000000)
{
    // Write PCI Status back to itself to clear any errors
    rc = PlxPciConfigRegisterWrite(
              Device.BusNumber,
              Device.SlotNumber,
              CFG_COMMAND,
              &RegValue
              );

    if (rc != ApiSuccess)
    {
        // ERROR – Unable to write to PCI configuration register
    }
}
```

PlxPciDeviceClose

Syntax:

```
RETURN_CODE
PlxPciDeviceClose(
    HANDLE hDevice
    );
```

PLX Chip Support:

All

Description:

Releases a PLX device object previously opened with *PlxPciDeviceOpen()*.

Parameters:

hDevice
Handle of an open PCI device

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidDeviceInfo	The device was not closed properly

Notes:

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

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


  // Release the open PLX device
  rc = PlxPciDeviceClose(
           hDevice
           );

  if (rc != ApiSuccess)
  {
      // ERROR – Unable to release PLX device
  }
```

PlxPciDeviceFind

Syntax:

```
RETURN_CODE
PlxPciDeviceFind(
    DEVICE_LOCATION *device,
    U32             *requestLimit
    );
```

PLX Chip Support:

All

Description:

Finds PLX devices on the PCI bus given a combination of bus number, slot number; vendor ID, and/or device ID, or by the Serial number.

Parameters:

device
A pointer to a DEVICE_LOCATION structure containing the search criteria and/or to contain device information once it is found.

requestLimit
A pointer to a 32-bit buffer. *Refer to the Notes section.*

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device information did not match any device in the system
ApiNoActiveDriver	There is no device driver installed into the system

Notes:

If *requestLimit* contains the value FIND_AMOUNT_MATCHED, the DEVICE_LOCATION structure should contain the search criteria

If *requestLimit* contains a value other than FIND_AMOUNT_MATCHED, the DEVICE_LOCATION structure will be filled in with information about the device indexed at *requestLimit* (numbering starts at 0).

Set items not to be used as search criteria to –1.

If the *SerialNumber* element within the DEVICE_LOCATION structure is not being used as a search criterion the first character should be set to an empty string; otherwise, the Serial number takes precedence over all other search criteria.

Usage:

```
U32             ReqLimit;
RETURN_CODE     rc;
DEVICE_LOCATION Device;


// Query to get the total number of PLX devices
ReqLimit = FIND_AMOUNT_MATCHED;

// No search criteria, select all devices
Device.BusNumber       = (U32)-1;
Device.SlotNumber      = (U32)-1;
Device.VendorId        = (U32)-1;
Device.DeviceId        = (U32)-1;
Device.SerialNumber[0] = '\0';

rc = PlxPciDeviceFind(
         &Device,
         &ReqLimit
         );

if ((rc != ApiSuccess) || (ReqLimit == 0))
{
    // ERROR – Unable to locate any valid devices
}

ReqLimit = FIND_AMOUNT_MATCHED;

// Search for the first device matching a specific Vendor ID
Device.BusNumber       = (U32)-1;
Device.SlotNumber      = (U32)-1;
Device.VendorId        = 0x10b5;     // PLX Vendor ID
Device.DeviceId        = (U32)-1;
Device.SerialNumber[0] = '\0';

rc = PlxPciDeviceFind(
         &Device,
         &ReqLimit
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to locate search for any matching devices
}
```

```
if (ReqLimit == 0)
{
    // ERROR – Unable to locate any matching devices
}
else
{
    // Found a device – fill in the device information
    PlxPciDeviceFind(
         &Device,
         &ReqLimit
         );
}
```

Cross Reference:

Referenced Item	Page
DEVICE_LOCATION	4-13

PlxPciDeviceOpen

Syntax:

```
RETURN_CODE
PlxPciDeviceOpen(
    DEVICE_LOCATION *pDevice,
    HANDLE          *pDrvHandle
    );
```

PLX Chip Support:

All

Description:

This function selects a specific PLX device, so it can be used by future API calls. The function works first by locating the device based on the criteria in the DEVICE_LOCATION, then opening the device by way of a handle, which is returned to the application for use in all subsequent PLX API calls.

Parameters:

pDevice
A pointer to the structure containing information pertaining to a specific device

pDrvHandle
A pointer to storage for the handle which will be created by the function

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device information did not match any device in the system
ApiNoActiveDriver	There is no device driver installed into the system

Notes:

Before the application terminates, the device should be released with the API call *PlxPciDeviceClose()*.

The DEVICE_LOCATION structure should contain the search criteria to locate a particular device. If no search criteria are specified, the first valid PLX device is selected.

Set items not to be used as search criteria to –1.

The function searches for a device matching the criteria as follows:

If a Serial Number is specified, the other members of the DEVICE_LOCATION structure are ignored and the Serial Number will be used to select a unique device. All remaining fields are then filled in.

If no Serial Number is specified (set to an empty string), the first device found matching the combination of bus number, slot number, Vendor ID and/or Device ID. All remaining fields are then filled in.

Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
DEVICE_LOCATION Device;


// Select the first PLX device found
Device.BusNumber       = (U32)-1;
Device.SlotNumber      = (U32)-1;
Device.DeviceId        = (U32)-1;
Device.VendorId        = (U32)-1;
Device.SerialNumber[0] = '\0';

rc = PlxPciDeviceOpen(
         &Device,
         &hDevice
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to open a PLX device
}

// Select the second 9054 device in the system
Device.BusNumber       = (U32)-1;
Device.SlotNumber      = (U32)-1;
Device.DeviceId        = (U32)-1;
Device.VendorId        = (U32)-1;
strcpy(
    Device.SerialNumber,
    "Pci9054-1"      // Numbering starts at 0
    );

rc = PlxPciDeviceOpen(
         &Device,
         &hDevice
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to open the 9054 device
}
```

```
// Select a spcific device by physical location
Device.BusNumber       = 0x01;
Device.SlotNumber      = 0x0f;
Device.DeviceId        = (U32)-1;
Device.VendorId        = (U32)-1;
Device.SerialNumber[0] = '\0';

rc = PlxPciDeviceOpen(
         &Device,
         &hDevice
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to open the PLX device
}
```

Cross Reference:

Referenced Item	Page
DEVICE_LOCATION	4-13

PlxPmIdRead

Syntax:

```
U8
PlxPmIdRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Power Management Capability ID.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiPMNotSupported	Power Management is either disabled or not supported by the PLX device

Usage:

```
U8          PowerManId;
HANDLE      hDevice;
RETURN_CODE rc;


PowerManId = PlxPmIdRead(
                hDevice,
                &rc
                );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read Power Management Capability ID
}
```

PlxPmNcpRead

Syntax:

```
U8
PlxPmNcpRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Next Capability Pointer from the Power Management register.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiPMNotSupported	Power Management is either disabled or not supported by the PLX device

Usage:

```
U8          NextCapability;
HANDLE      hDevice;
RETURN_CODE rc;


NextCapability = PlxPmNcpRead(
                     hDevice,
                     &rc
                     );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read Power Management NCP
}
```

PlxPowerLevelGet

Syntax:

```
PLX_POWER_LEVEL
PlxPowerLevelGet(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9080*, 9054, 480, 9030, 9050/52*

*These chips do not support power management. Therefore a full power state of D0 is always returned.

Description:

Returns the current power level of a PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidHandle	The function was passed an invalid device handle
ApiPMNotSupported	Power Management is either disabled or not supported by the PLX device

Notes:

Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
PLX_POWER_LEVEL PowerLevel;


PowerLevel = PlxPowerLevelGet(
                 hDevice,
                 &rc
                 );

if (rc != ApiSuccess)
{
    // ERROR – Unable to retrieve Power Level
}
else
{
    switch (PowerLevel)
    {
        case D0:
            // Device full power state (D0)
            break;

        case D1:
            // Device power state D1
            break;

        case D2:
            // Device power state D2
            break;

        case D3Hot:
            // Device power state D3Hot
            break;
    }
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxPowerLevelSet

Syntax:

```
RETURN_CODE
PlxPowerLevelSet(
    HANDLE          hDevice,
    PLX_POWER_LEVEL plxPowerLevel
    );
```

PLX Chip Support:

9080*, 9054, 480, 9030, 9050/52*

*These chips do not support power management. Therefore only a full power state of D0 is accepted.

Description:

Sets the device power level of a PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

plxPowerLevel
The new power level

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidHandle	The function was passed an invalid device handle
ApiPMNotSupported	Power Management is either disabled or not supported by the PLX device
ApiInvalidPowerState	The Power Level parameter is not supported by the PLX chip

Notes:

Setting the power state is typically a task accomplished by the Operating System through the driver. It is not recommended for applications to change the device power state because this will bypass the proper OS channels.

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// Put device in Low Power state
rc = PlxPowerLevelSet(
         hDevice,
         D3Hot
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set new Power level
}

// Restore device to Full Power state
rc = PlxPowerLevelSet(
         hDevice,
         D0
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set new Power level
}
```

Cross Reference:

Referenced Item	Page
PLX_POWER_LEVEL	4-60

PlxRegisterDoorbellRead

Syntax:

```
U32
PlxRegisterDoorbellRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Returns the last value written to the Local-to-PCI doorbell register.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidHandle	The function was passed an invalid device handle
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 device driver only keeps track of the last occurrence of the doorbell interrupt. Later interrupts will overwrite previous values. It is the application's responsibility to keep up with the doorbell interrupts if required.

Usage:

```
U32         DoorBellValue;
HANDLE      hDevice;
HANDLE      hInterrupt;
PLX_INTR    PlxIntr;
RETURN_CODE rc;


// Wait for a doorbell interrupt – refer to PlxIntrAttach function
PlxIntr.PciDoorbell = 1;

PlxIntrAttach(
    hDevice,
    &PlxIntr,
    &hInterrupt
    );

WaitForSingleObject(
    hInterrupt,
    INFINITE
    );

// Get the value written
DoorBellValue =
    PlxRegisterDoorbellRead(
        hDevice,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get last doorbell interrupt value
}
```

PlxRegisterDoorbellSet

Syntax:

```
RETURN_CODE
PlxRegisterDoorbellSet(
    HANDLE hDevice,
    U32    data
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Writes a value to the PCI-to-Local doorbell register of a PLX PCI device. This is typically used to pass 32-bit interrupt-triggered information or messages to the local CPU.

Parameters:

hDevice
Handle of an open PCI device

data
The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;

rc = PlxRegisterDoorbellSet(
         hDevice,
         (1 << 30) | (1 << 8)   // Send a “message” to the local CPU
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to set PCI-to-Local doorbell
}
```

PlxRegisterMailboxRead

Syntax:

```
U32
PlxRegisterMailboxRead(
    HANDLE       hDevice,
    MAILBOX_ID   mailboxId,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Returns the value contained in a specific mailbox register of the currently selected PLX PCI device

Parameters:

hDevice
Handle of an open PCI device

mailboxId
The mailbox register ID

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidRegister	An invalid Mailbox ID parameter was passed

Notes:

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

Usage:

```
U32         MailboxValue;
HANDLE      hDevice;
RETURN_CODE rc;


MailboxValue =
    PlxRegisterMailboxRead(
        hDevice,
        MailBox0,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read mailbox value
}
else
{
    // Can be used for custom “message passing”
    switch (MailboxValue)
    {
        case 0x01:
            // Local CPU is ready for more data, initiate DMA
            break;

        case 0x02:
            // Local CPU has prepared some data, transfer and process it
            break;

        case 0x03:
            // Local CPU completed transaction, log acknowledgement
            break;
    }
}
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	4-45

PlxRegisterMailboxWrite

Syntax:

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

PLX Chip Support:

9080, 9054, 480

Description:

Writes a 32-bit value to a specified mailbox register of a PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

mailboxId
The mailbox register ID

data
The 32-bit value to write to the mailbox register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidRegister	An invalid Mailbox ID parameter was passed

Notes:

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;

/*******************************************************
 * In this example, the Host prepares some data or a
 * message for processing by the local CPU. The data is
 * placed in local memory and the communication involves
 * writing the local base address of the data to Mailbox 1
 * and then setting Bit 8 of MailBox 2 to denote the data is
 * ready. The local CPU can either poll Mailbox 2 or wait
 * for a Mailbox 2 local interrupt
 *
 *   - Data preparation & transfer not show here -
 ************************************************/

// Write local address of data ready for processing
rc = PlxRegisterMailboxWrite(
         hDevice,
         MailBox1,
         0x00080000
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to Mailbox register
}

// Signal that data is ready
rc = PlxRegisterMailboxWrite(
         hDevice,
         MailBox2,
         (1 << 8)
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to Mailbox register
}
```

Cross Reference:

Referenced Item	Page
MAILBOX_ID	4-45

PlxRegisterRead

Syntax:

```
U32
PlxRegisterRead(
    HANDLE       hDevice,
    U32          registerOffset,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

All

Description:

Returns the value of the internal register at a specified offset of the selected PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

registerOffset
The register offset

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidRegister	An invalid or non-32-bit aligned register offset was passed

Notes:

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

Usage:

```
U32         RegValue;
HANDLE      hDevice;
RETURN_CODE rc;


// Check if an EEPROM is present on the 9054 device
RegValue =
    PlxRegisterRead(
        hDevice,
        PCI9054_EEPROM_CTRL_STAT,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read PLX chip register
}

if ((RegValue & (1 << 28)) == 0)
{
    // EEPROM is not present or is blank
}
```

PlxRegisterReadAll

Syntax:

```
RETURN_CODE
PlxRegisterReadAll(
    HANDLE  hDevice,
    U32     startOffset,
    U32     registerCount,
    U32    *buffer
    );
```

PLX Chip Support:

All

Description:

Reads multiple consecutive registers of a PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

startOffset
The register offset to start reading from (aligned on 4-byte boundary)

registerCount
The number of bytes to read

buffer
A pointer to a buffer which will contain the register values

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidRegister	An invalid or non-32-bit aligned register offset was passed or invalid range

Notes:

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

The application-provided buffer must be at least equal to or larger than the number of bytes to read.

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

Usage:

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


rc = PlxRegisterReadAll(
         hDevice,
         0x0,
         0x100,
         buffer
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read register range
}
```

PlxRegisterWrite

Syntax:

```
RETURN_CODE
PlxRegisterWrite(
    HANDLE hDevice,
    U32    registerOffset,
    U32    data
    );
```

PLX Chip Support:

All

Description:

Writes a value to a specified register of a PLX PCI device.

Parameters:

hDevice
Handle of an open PCI device

registerNumber
The register offset

data
The 32-bit value to write to the register

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidRegister	An invalid or non-32-bit aligned register offset was passed

Notes:

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// Adjust the remap of a 9054 chip
rc = PlxRegisterWrite(
         hDevice,
         PCI9054_SPACE0_REMAP,
         0x43000000 | (1 << 0)
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to PLX chip register
}
```

PlxSdkVersion

Syntax:

```
RETURN_CODE
PlxSdkVersion(
    U8 *VersionMajor,
    U8 *VersionMinor,
    U8 *VersionRevision
    );
```

PLX Chip Support:

All

Description:

Returns the SDK API version information

Parameters:

VersionMajor
A pointer to an 8-bit buffer to contain the Major version number

VersionMinor
A pointer to an 8-bit buffer to contain the Minor version number

VersionRevision
A pointer to an 8-bit buffer to contain the Revision version number

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL

Usage:

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


rc = PlxSdkVersion(
         &SdkMajor,
         &SdkMinor,
         &SdkRevision
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get API version information
}
else
{
    printf("SDK API Version = %d.%d%d\n",
           SdkMajor, SdkMinor, SdkRevision);
}
```

PlxSerialEepromPresent

Syntax:

```
BOOLEAN
PlxSerialEepromPresent(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

All

Description:

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

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function completed successfully
ApiInvalidHandle	The function was passed an invalid device handle
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()*.

For PLX chips other than the 480, a blank EEPROM will result in a return value of FALSE. The PLX chip does not distinguish between blank and non-existent EEPROMs.

Usage:

```
HANDLE      hDevice;
BOOLEAN     EepromPresent;
RETURN_CODE rc;


EepromPresent =
    PlxSerialEepromPresent(
        hDevice,
        &rc
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to determine EEPROM status
}

if (EepromPresent)
{
    // Programmed EEPROM exists
}
else
{
    // EEPROM does not exist or is blank
}
```

PlxSerialEepromRead

Syntax:

```
RETURN_CODE
PlxSerialEepromRead(
    HANDLE       hDevice,
    EEPROM_TYPE  eepromType,
    U32         *buffer,
    U32          size
    );
```

PLX Chip Support:

All

Description:

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

Parameters:

hDevice
Handle of an open PCI device

eepromType
The type of EEPROM installed on the PCI device

buffer
A pointer to a buffer which will contain the data read. This buffer must be large enough to hold the amount of data requested.

size
The number of bytes to read from the EEPROM. (Must be 4-byte aligned)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidSize	The number of bytes is invalid, larger than the EEPROM type or not 4-byte aligned

Notes:

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

Attempting to read a device that does not contain an EEPROM may result in a system crash.

The EEPROM data is always read starting at EEPROM offset 0.

Usage:

```
U16         EepromData[0x16];
HANDLE      hDevice;
RETURN_CODE rc;


// Read first few bytes of EEPROM data
rc = PlxSerialEepromRead(
          hDevice,
          Eeprom93CS56,
          (U32 *)EepromData,
          0x16 * sizeof(U16)        // Size in bytes
          );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read EEPROM
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	4-29

PlxSerialEepromWrite

Syntax:

```
RETURN_CODE
PlxSerialEepromWrite(
    HANDLE       hDevice,
    EEPROM_TYPE  eepromType,
    U32         *buffer,
    U32          size
    );
```

PLX Chip Support:

All

Description:

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

Parameters:

hDevice
Handle of an open PCI device

eepromType
The type of EEPROM installed on the PCI device

buffer
A pointer to a buffer containing the data to write

size
The number of bytes to write to the EEPROM. (Must be 4-byte aligned)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidSize	The number of bytes is invalid, larger than the EEPROM type or not 4-byte aligned

Notes:

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

Attempting to write to a device that does not contain an EEPROM may result in a system crash.

The EEPROM data is always written to starting at EEPROM offset 0.

Usage:

```
U16         EepromData[0x4];
HANDLE      hDevice;
RETURN_CODE rc;


// Prepare EEPROM data (assuming 9054)
EepromData[0] = 0x9054     // Device/Vendor ID
EepromData[1] = 0x10b5
EepromData[2] = 0x0068     // Class code/Revision
EepromData[3] = 0x0001

// Write EEPROM data
rc = PlxSerialEepromWrite(
         hDevice,
         Eeprom93CS56,
         (U32 *)eepromData,
         0x4 * sizeof(U16)     //  Size in bytes
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to EEPROM
}
```

Cross Reference:

Referenced Item	Page
EEPROM_TYPE	4-29

PlxUserRead

Syntax:

```
PLX_PIN_STATE
PlxUserRead(
    HANDLE       hDevice,
    USER_PIN     userPin,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Returns the state of a specified USER pin of a PLX chip.

Parameters:

hDevice
Handle of an open PCI device

userPin
The USERI pin to read

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidUserPin	The User pin is invalid or not supported by the selected PLX chip

Notes:

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

Usage:

```
HANDLE        hDevice;
RETURN_CODE   rc;
PLX_PIN_STATE PinState;


PinState = PlxUserRead(
               hDevice,
               USER0,
               &rc
               );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read state of USER pin
}

if (PinState == Active)
{
    // User pin is active
}
```

Cross Reference:

Referenced Item	Page
USER_PIN	4-68
PLX_PIN_STATE	4-53

PlxUserWrite

Syntax:

```
RETURN_CODE
PlxUserWrite(
    HANDLE        hDevice,
    USER_PIN      userPin,
    PLX_PIN_STATE pinState
    );
```

PLX Chip Support:

9080, 9054, 480

Description:

Set the state of a specified User pin of a PLX PCI device

Parameters:

hDevice
Handle of an open PCI device

userPin
The User pin to set the state of

pinState
The new state to set the User pin

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiPowerDown	The PLX device is in a power state that is lower than required for this function
ApiInvalidUserPin	The User pin is invalid or not supported by the selected PLX chip

Notes:

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

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


// Write to the User pin
rc = PlxUserWrite(
         hDevice,
         USER0,
         Active
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to User pin
}
```

Cross Reference:

Referenced Item	Page
USER_PIN	4-68
PLX_PIN_STATE	4-53

PlxVpdIdRead

Syntax:

```
U8
PlxVpdIdRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Vital Product Data Capability ID.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Usage:

```
U8          VpdId;
HANDLE      hDevice;
RETURN_CODE rc;


VpdId = PlxVpdIdRead(
            hDevice,
            &rc
            );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read VPD ID
}
```

PlxVpdNcpRead

Syntax:

```
U8
PlxVpdNcpRead(
    HANDLE       hDevice,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Returns the Next Capability Pointer from the VPD register.

Parameters:

hDevice
Handle of an open PCI device

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidHandle	The function was passed an invalid device handle
ApiVPDNotSupported	VPD is either disabled or not supported by the PLX device

Usage:

```
U8          NextCapability;
HANDLE      hDevice;
RETURN_CODE rc;


NextCapability = PlxVpdNcpRead(
                     hDevice,
                     &rc
                     );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read VPD NCP
}
```

PlxVpdRead

Syntax:

```
U32
PlxVpdRead(
    HANDLE       hDevice,
    U32          offset,
    RETURN_CODE *pReturnCode
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Reads a 32-bit value at a specified offset of the EEPROM using the Vital Product Data feature of the selected PLX chip.

Parameters:

hDevice
Handle of an open PCI device

offset
The is the byte offset to read from (must be aligned 32-bit boundary)

pReturnCode
A pointer to a buffer for the return code

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidHandle	The function was passed an invalid device handle

Usage:

```
U32         VpdData;
HANDLE      hDevice;
RETURN_CODE rc;


// Read the default Space 1 range (assuming a 9054)
VpdData = PlxVpdRead(
              hDevice,
              0x48,
              &rc
              );

if (rc != ApiSuccess)
{
    // ERROR – Unable to read VPD data
}
```

PlxVpdWrite

Syntax:

```
RETURN_CODE
PlxVpdWrite(
    HANDLE hDevice,
    U32    offset,
    U32    vpdData
    );
```

PLX Chip Support:

9054, 480, 9030

Description:

Write a 32-bit value to a specified offset of the EEPROM using the Vital Product Data feature of the selected PLX chip.

Parameters:

hDevice
Handle of an open PCI device

offset
The is the byte offset to write to (must be aligned 32-bit boundary)

vpdData
The 32-bit data to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidHandle	The function was passed an invalid device handle

Usage:

```
HANDLE      hDevice;
RETURN_CODE rc;


/*****************************************************
 * If the offset is within the write protected area, the
 * EEPROM write-protect boundary must first be adjusted.
 *
 *  - Write-protect boundary adjustment not shown here -
 *****************************************************/

// Write the new Device/Vendor ID (assuming 9054 device)
rc = PlxVpdWrite(
         hDevice,
         0x0,
         0x186010b5
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to VPD
}

// Write custom data to non-PLX used EEPROM space
rc = PlxVpdWrite(
         hDevice,
         0x60,         // 9054 data ends at 0x58
         0x0024beef
         );

if (rc != ApiSuccess)
{
    // ERROR – Unable to write to VPD
}
```

4 PLX SDK Data Structures Used by the API

4.1 Details of Data Structures

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

SAMPLE structure

```
typedef struct _SAMPLE
{
    U32 Member_1;
    U32 Member_2;
    ....
} SAMPLE, *PSAMPLE;
```

Affected Register Location

Structure Element	9080	9054	480	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 Local side unless stated otherwise. “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.

S8 and U8 Data Types

```
#if !defined(S8)
    typedef signed char             S8, *PS8;
#endif

#if !defined(U8)
    typedef unsigned char           U8, *PU8;
#endif
```

Affected Register Location

N/A

Purpose

These data types are used for 8 bit values.

S16 and U16 Data Types

```
#if !defined(S16)
    typedef signed short            S16, *PS16;
#endif

#if !defined(U16)
    typedef unsigned short          U16, *PU16;
#endif
```

Affected Register Location

N/A

Purpose

These data types are used for 16 bit values.

S32 and U32 Data Types

```
#if !defined(S32)
    typedef signed long             S32, *PS32;
#endif

#if !defined(U32)
    typedef unsigned long           U32, *PU32;
#endif
```

Affected Register Location

N/A

Purpose

These data types are used for 32 bit values.

S64 and U64 Data Types

```
#if !defined(S64)
    typedef union _S64
    {
        struct
        {
            U32  LowPart;
            S32  HighPart;
        }u;

        LONGLONG QuadPart;
    } S64;
#endif

#if !defined(U64)
    typedef union _U64
    {
        struct
        {
            U32  LowPart;
            U32  HighPart;
        }u;

        ULONGLONG QuadPart;
    } U64;
#endif
```

Affected Register Location

N/A

Purpose

This structure supports a standard 64-bit data structure, which allows for code compatibility between 32-bit and 64-bit environments.

LONGLONG and ULONGLONG Data Types

Local environment:

```
#if !defined(LONGLONG)
    typedef signed long long        LONGLONG;
#endif

#if !defined(ULONGLONG)
    typedef unsigned long long      ULONGLONG;
#endif
```

Win32 environments:

```
#if !defined(LONGLONG)
    typedef signed _int64           LONGLONG;
#endif

#if !defined(ULONGLONG)
    typedef unsigned _int64         ULONGLONG;
#endif
```

Affected Register Location

N/A

Purpose

These data types are provided for 64-bit signed and unsigned values. For the local-side, the compiler must support the *long long* data type, which signifies 64-bit data, even in 32-bit environments.

BOOLEAN Types

```
#if !defined(BOOLEAN)
    typedef S8                      BOOLEAN, *PBOOLEAN;
#endif

#if !defined(BOOL)
    typedef S8                      BOOL, *PBOOL;
#endif
```

Affected Register Location

N/A

Purpose

This data type defines the BOOLEAN and BOOL data types.

ADDRESS, SDATA and UDATA Data Types

```
#if defined(PLX_ADDR_64)
    typedef ULONGLONG           ADDRESS, *PADDRESS;
#else
    typedef U32                 ADDRESS, *PADDRESS;
#endif

#if defined(PLX_DATA_64)
    typedef LONGLONG            SDATA, *PSDATA;
    typedef ULONGLONG           UDATA, *PUDATA;
#else
    typedef S32                 SDATA, *PSDATA;
    typedef U32                 UDATA, *PUDATA;
#endif
```

Affected Register Location

N/A

Purpose

These data types are provided for code compatibility in embedded environments. These allow easy code porting between any combination of 32-bit or 64-bit addressing and data access size. The data size represents the maximum data size a CPU can access in a single cycle.

ACCESS_TYPE Type Enumerated Data Type

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

Affected Register Location

N/A.

Purpose

Enumerated type used for determining the access type size for a data transfer.

Members

BitSize8
Use 8-bits access

BitSize16
Use 16-bit access

BitSize32
Use 32-bit access

BitSize64
Use 64-bit access

API Parameters Structure

```
typedef struct _API_PARMS
{
    VOID            *PlxIcIopBaseAddr;
    VOID            *SpuMemBaseAddr;
    PCI_BUS_PROP    *PtrPciBusProp;
    PCI_ARBIT_DESC  *PtrPciArbitDesc;
    IOP_BUS_PROP    *PtrIopBus0Prop;
    IOP_BUS_PROP    *PtrIopBus1Prop;
    IOP_BUS_PROP    *PtrIopBus2Prop;
    IOP_BUS_PROP    *PtrIopBus3Prop;
    IOP_BUS_PROP    *PtrLcs0Prop;
    IOP_BUS_PROP    *PtrLcs1Prop;
    IOP_BUS_PROP    *PtrLcs2Prop;
    IOP_BUS_PROP    *PtrLcs3Prop;
    IOP_BUS_PROP    *PtrDramProp;
    IOP_BUS_PROP    *PtrDefaultProp;
    IOP_BUS_PROP    *PtrExpRomBusProp;
    IOP_ARBIT_DESC  *PtrIopArbitDesc;
    IOP_ENDIAN_DESC *PtrIopEndianDesc;
    PM_PROP         *PtrPMProp;
    U32             *PtrVPDBaseAddress;
}API_PARMS, *PAPI_PARMS;
```

Affected Register Location

N/A.

Purpose

This data type provides information about the board to the Local API and provides data structures to be initialized 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 Local API.

SpuMemBaseAddr
The memory mapped base address of Serial Port Unit, user should supply this value when call PlxInitApi() to pass data to the Local API (480 only).

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)

Bus Index Enum Data Type

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

Affected Register Location

N/A

Purpose

Enumerated type used for choosing the desired PCI bus.

Members

PrimaryPciBus
Use the primary PCI bus.

SecondaryPciBus
Use the secondary 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 of a PCI devices. It is used with the *PlxPciDeviceOpen()* and the *PlxPciDeviceFind()* PCI API functions.

Members

DeviceId
The Device ID for the PCI device.

VendorId
The Vendor ID for the PCI device.

BusNumber
The Bus Number where the PCI device is located.

SlotNumber
The Slot Number where the PCI device is located.

SerialNumber
A unique identifier for the PCI device. The format of the serial number is: "<device name>-<index number>". A 9054, for example, may have the Serial number "Pci9054-1", which means it is the second 9054 device in the system. Numbering starts at 0.

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	9080	9054	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	9080	9054	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 ♣
TholdForPciReads	0x130, 12-15 0x130, 28-30	0x130, 12-15 0x130, 28-30	0x21C, 12-15 0x23C, 12-15 ♣

Structure Element	9080	9054	480
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 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 the DMA Done interrupt if this bit is set. The DMA done interrupt is always set for SGL and Shuttle DMA transfers to allow for cleanup after the DMA has completed. The user is able to set or unset this bit for Block DMA transfers. However, if TerminalCountIntr is set (in DMA_TRANSFER_ELEMENT) to 1 for Block DMA transfers, the program will set *EnableDoneInt* to 1. (Please refer to page 4-27 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

N/A

Purpose

Enumerated type used for requesting a DMA channel.

Members

IopChannel0
Request the Local-to-Local DMA channel 0.

IopChannel1
Request the Local-to-Local DMA channel 1.

IopChannel2
Request the Local-to-Local DMA channel2.

PrimaryPciChannel0
Request the Primary PCI-to-Local DMA channel 0.

PrimaryPciChannel1
Request the Primary PCI-to-Local DMA channel 1.

SecondaryPciChannel0
Request the Secondary PCI-to-Local DMA channel 0.

SecondaryPciChannel1
Request the Secondary PCI-to-Local DMA channel 1.

DMA Channel Priority Enum Data Type

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

Affected Register Location

N/A

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.

DMA Command Enum Data Type

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

Affected Register Location

N/A

Purpose

Enumerated type used to control a DMA transfer.

Members

DmaStart (obselete, provided only for compatibility with older applications)
Start a DMA transfer.

DmaPause
Suspend a DMA transfer.

DmaResume
Resume a DMA transfer.

DmaAbort
Abort a DMA transfer.

DmaStatus (obselete, provided only for compatibility with older applications)
Determine the status of a DMA transfer.

DMA Direction Enum Data Type

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

Affected Register Location

N/A

Purpose

Enumerated type used for providing the DMA transfer direction.

Members

IopToIop
Transfer between two Local bus addresses.

IopToPrimaryPci
Transfer from a Local bus address to a Primary PCI bus address.

PrimaryPciToIop
Transfer from a Primary PCI bus address to a Local bus address.

IopToSecondaryPci
Transfer from a Local bus address to a Secondary PCI bus address.

SecondaryPciToIop
Transfer from a Secondary PCI bus address to a Local 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.

DMA Resource Manager Parameters Structure

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

Affected Register Location

N/A

Purpose

This data type is used for passing information from the BSP module into the DMA Resource Manager when it is initialized. The DMA manager recognizes the end of the DMA_PARMS list if any of the following conditions are true:

The *DmaChannel* member is invalid;

The *FirstSglElement* member is NULL

The *WaitQueueBase* member is NULL

The *NumberOfElements* is 0

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 this array must be on a 16-byte 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.

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;
    #endif
    } 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
    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
        U32 HighPciAddr;
    } Iop480Dma;

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

typedef PDMA_TRANSFER_ELEMENT     SGL_ADDR, *PSGL_ADDR;
```

Affected Register Location

Structure Element	9080	9054	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
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	0x110, 2	0x214, 2

Structure Element	9080	9054	480
	0x124, 2	0x124, 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 9080, 9054, and 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 9080 and 9054; Register 0x200 or 0x220, bit 10 or Register 0x240, bit 13 on 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 9054.

Iop480Dma
The structure containing the DMA data specific for the 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 the 480. Indicates the Local Memory space location in which the DMA transfers start.

Iop480Dma.Loc1.SourceAddr
Source Address Register for DMA Channel 2 of the 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.

EEPROM Type Enum Data Type

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

Affected Register Location

N/A

Purpose

Enumerated type used to specify the EEPROM type used

Members

Eeprom93CS46
National NM93CS46 or compatible.

Eeprom93CS56
National NM93CS56 or compatible.

Eeprom93CS66
National NM93CS66 or compatible.

EepromX24012
Xicor X24012 EEPROM or compatible.

EepromX24022
Xicor X24022 EEPROM or compatible.

EepromX24042
Xicor X24042 EEPROM or compatible.

EepromX24162
Xicor X24162 EEPROM or compatible.

Hot Swap Status Definition

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

Related Register Location

Definition	9054	9030	480
HS_LED_ON	0x18A, 19	0x48, 19	0x356, 19
HS_BOARD_REMOVED	0x18A, 21	0x48, 21	0x356, 21
HS_BOARD_INSERTED	0x18A, 22	0x48, 22	0x356, 22

Purpose

To specify the Hot Swap status. The Hot Swap Status definitions can be combined using the OR operator

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.

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	9080	9054	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 480 can assert the BOFF# pin when this value is set.

BOFFTimerResolution
When this value is set the LSB of the 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	9080	9054	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 0x168, 8
EnableIopBusTimeoutTimer	N/A	N/A	0x100, 14 0x114, 14 0x128, 14 0x168, 14
BREQoTimerResolution	0x94, 5	0x94, 5	N/A

Structure Element	9080	9054	480
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
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

Structure Element	9080	9054	480
			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
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 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 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 480. The value must be between 0-15.

Iop480WDLYWaitStates
This value contains the number of Write Enable Delay states to insert for the 480. The value must be between 0-15.

Iop480WHLDWaitStates
This value contains the number of Write Hold states to insert for the 480. The value must be between 0-15.

Iop480WRCVRecoverStates
This value contains the number of Write Recovery states to insert for the 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 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 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 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 480. The value must be between 0-15.

Iop480RRCVRecoverStates
This value contains the number of Read Recovery states to insert for the 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 terms 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	9080	9054	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
BigEMaster480LCS3	N/A	N/A	0x13C, 4

Structure Element	9080	9054	480
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 480 CPU is the local Bus Master, the LCS0 is configured with big endian data ordering if this value is set.

BigEMaster480LCS1
When 480 CPU is the local Bus Master, the LCS1 is configured with big endian data ordering if this value is set.

BigEMaster480LCS2
When 480 CPU is the local Bus Master, the LCS2 is configured with big endian data ordering if this value is set.

BigEMaster480LCS3
When 480 CPU is the local Bus Master, the LCS3 is configured with big endian data ordering if this value is set.

BigEMaster480Dram
When 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 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 480 CPU, the IOP configuration register is configured with big endian data ordering if this value is set.

BigEDirectMasterInternal
For internal 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

N/A

Purpose

Enumerated type used to select the desired PCI-to-Local Space when accessing the Local bus devices.

Members

IopSpaceX
Use Local Space X base address register.

MsLcsX
Use LCS X 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	9080	9054	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

MailBoxX Use mailbox X register.

New Capabilities Flags

```
#define CAPABILITY_POWER_MANAGEMENT      (1 << 0)
#define CAPABILITY_HOT_SWAP              (1 << 1)
#define CAPABILITY_VPD                   (1 << 2)
```

Related Register Location

None

Purpose

To specify which New Capabilities to check. The definitions can be combined using the OR operator

Members

PCI Arbitration Descriptor Structure

```
typedef struct _PCI_ARBIT_DESC
{
    U32 PciHighPriority;
} PCI_ARBIT_DESC, *PPCI_ARBIT_DESC;
```

Affected Register Location

Structure Element	9080	9054	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 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 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	9080	9054	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
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, the 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, the 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, the 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 480 treats 256 Master consecutive retries to a Target as a Target Abort when this value is set. Otherwise, the 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
Setting this value to 1 allows the device to behave as a bus master. Clearing this bit disables the device from generating Bus Master accesses.

EnableMemorySpaceAccess
Setting this value to 1 allows the device to respond to Memory Space accesses.

EnableIoSpaceAccess
Setting 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;
```

Affected Register Location

N/A

Purpose

This data type provides information for a contiguous page-locked buffer allocated by the device driver. This is typically used as a common buffer for DMA transfers.

Members

UserAddr
User Virtual Address for the buffer

PhysicalAddr
Physical Address of the buffer.

Size
The size of the buffer.

PCI Space Enum Data Type

```
typedef enum _PCI_SPACE
{
    PciMemSpace,
    PciIoSpace
} PCI_SPACE;
```

Affected Register Location

N/A

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.

PLX Pin State Enum Data Type

```
typedef enum _PLX_PIN_STATE
{
    Inactive,
    Active
} PLX_PIN_STATE;
```

Affected Register Location

N/A

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, setting the USER pin inactive will cause it to go high.

Active
Set the USER pin to active state. If the USER pin is active low, setting the USER pin active will 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	9080	9054	480	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 Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 24 A 0x1B8, 24	N/A
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

Structure Element	9080	9054	480	9030
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 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	E 0x1BC, 15	N/A

Structure Element	9080	9054	480	9030
		A 0xE8, 5	A 0x1B8, 15 C 0x1B8, 15	
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 A 0x1BC. 7 C 0x1BC, 7	N/A
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 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

N/A

Purpose

Enumerated type used to state the power level.

Members

D0Uninitialized
Device D0Uninitialized state. This state is the initialization state before the D0 full power state.

D0
Device D0 state. This state is the full power state and is the state for normal operation.

D1
Device D1 state.

D2
Device D2 state.

D3Hot
Device D3Hot state.

D3Cold
Device D3Cold 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	9080	9054	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	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	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

This data type is used to set or get Serial Port Handshake Status and Line 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

USER Pin Direction Enum Data Type

```
typedef enum _PLX_PIN_DIRECTION
{
    PLX_PIN_INPUT,
    PLX_PIN_OUTPUT
} PLX_PIN_DIRECTION;
```

Affected Register Location

Structure Element	9080	9054	480
PLX_PIN_INPUT	N/A	N/A	0x84, 2, 5, 9
PLX_PIN_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

PLX_PIN_INPUT
Select direction to be input

PLX_PIN_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	9080	9054	480
USER0	I 0xEC, 17 O 0xEC, 16	I 0xEC, 17 O 0xEC, 16	0x84, 0-3
USER1	N/A	N/A	0x84, 1, 4-6
USER2	N/A	N/A	0x84, 4, 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.

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 Virtual Addresses for a PCI device. The addresses correspond to the device's PCI Base Address Registers (BAR)

Members

VaX
Virtual address for PCI BAR X.

VaRom
Virtual address for the Expansion ROM BAR.

Appendix A. PLX SDK Revision Notes

A.1 New Local API functions for SDK Version 3.2

API Function	Page
PlxChipBaseAddressGet	2-9
PlxChipTypeGet	2-11
PlxIsNewCapabilityEnabled	2-90
PlxIsPowerLevelSupported	2-91
PlxVerifyEndianAccess	2-153

A.2 New Host API functions for SDK Version 3.2

API Function	Page
PlxChipTypeGet	3-10
PlxDriverVersion	3-40
PlxPciBoardReset	3-78

A.3 Changes in SDK 3.2 from SDK 3.0 & 3.1

Item Changed	Description	Page
PlxIntrAttach	Although the prototype for this function remains the same, it's internal implementation changed in order to avoid exceptions, as in previous SDK releases. Refer to the function for additional details.	3-46
ADDRESS and U64 types	These were redefined to better accommodate code portability with upcoming 64-bit devices	4-8 4-5
PIN_DIRECTION	The *INPUT* and *OUTPUT* members conflicted with an include file in the Linux environment. The type is now PLX_PIN_DIRECTION and the members are now PLX_PIN_INPUT and PLX_PIN_OUTPUT	4-67
PIN_STATE	PIN_STATE created a conflict with a type in a compiler tool. The type is now called PLX_PIN_STATE.	4-53