



# PLX SDK

## Software Development Kit

### PLX 9000 Legacy API Reference Manual

Version 6.50

September 2011

#### **WARNING**

The PLX Legacy API functions presented in this manual have been deprecated and are no longer supported by PLX. This manual is provided for reference purposes only to support existing applications that still use the Legacy API.



# Table of Contents

<b>1</b>	<b>General Information .....</b>	<b>1-1</b>
1.1	About This Manual .....	1-1
<b>2</b>	<b>PLX 9000 Legacy API .....</b>	<b>2-1</b>
2.1	PLX 9000 API Conversion Table .....	2-1
2.2	PLX 9000 API Function Details .....	2-3
	PlxBuslopRead .....	2-3
	PlxBuslopWrite .....	2-6
	PlxChipTypeGet.....	2-9
	PlxDmaBlockChannelClose.....	2-11
	PlxDmaBlockChannelOpen .....	2-13
	PlxDmaBlockTransfer .....	2-15
	PlxDmaControl.....	2-18
	PlxDmaSglChannelClose .....	2-20
	PlxDmaSglChannelOpen.....	2-22
	PlxDmaSglTransfer.....	2-24
	PlxDmaStatus .....	2-27
	PlxDriverVersion .....	2-29
	PlxHotSwapNcpRead .....	2-31
	PlxHotSwapStatus .....	2-32
	PlxIntrAttach.....	2-34
	PlxIntrDisable.....	2-35
	PlxIntrEnable.....	2-37
	PlxIntrStatusGet.....	2-39
	PlxIntrWait.....	2-41
	PlxIoPortRead.....	2-42
	PlxIoPortWrite .....	2-44
	PlxMuInboundPortRead.....	2-46
	PlxMuInboundPortWrite .....	2-48
	PlxMuOutboundPortRead.....	2-50
	PlxMuOutboundPortWrite .....	2-52
	PlxNotificationCancel .....	2-54
	PlxNotificationRegisterFor .....	2-56
	PlxNotificationWait.....	2-58
	PlxPciBarGet.....	2-60
	PlxPciBarMap .....	2-62
	PlxPciBarRangeGet.....	2-64
	PlxPciBarUnmap.....	2-66
	PlxPciBaseAddressesGet.....	2-68
	PlxPciBoardReset.....	2-69
	PlxPciCommonBufferGet.....	2-70
	PlxPciCommonBufferMap.....	2-71

PlxPciCommonBufferProperties .....	2-73
PlxPciCommonBufferUnmap .....	2-75
PlxPciConfigRegisterRead .....	2-77
PlxPciConfigRegisterWrite .....	2-79
PlxPciDeviceClose .....	2-81
PlxPciDeviceFind .....	2-82
PlxPciDeviceOpen .....	2-85
PlxPciPhysicalMemoryAllocate .....	2-88
PlxPciPhysicalMemoryFree .....	2-91
PlxPciPhysicalMemoryMap .....	2-93
PlxPciPhysicalMemoryUnmap .....	2-95
PlxPciRegisterReadFast .....	2-97
PlxPciRegisterWriteFast .....	2-98
PlxPciRegisterRead_Unsupported .....	2-100
PlxPciRegisterWrite_Unsupported .....	2-102
PlxPmNcpRead .....	2-104
PlxPowerLevelGet .....	2-105
PlxPowerLevelSet .....	2-107
PlxRegisterDoorbellRead .....	2-109
PlxRegisterDoorbellSet .....	2-111
PlxRegisterMailboxRead .....	2-112
PlxRegisterMailboxWrite .....	2-114
PlxRegisterRead .....	2-116
PlxRegisterWrite .....	2-118
PlxSdkVersion .....	2-120
PlxSerialEepromPresent .....	2-121
PlxSerialEepromRead .....	2-123
PlxSerialEepromReadByOffset .....	2-124
PlxSerialEepromWrite .....	2-126
PlxSerialEepromWriteByOffset .....	2-127
PlxVpdNcpRead .....	2-129
PlxVpdRead .....	2-130
PlxVpdWrite .....	2-131

### **3 PLX SDK Data Structures Used by the API .....3-1**

3.1 Details of Data Structures .....	3-1
SAMPLE structure .....	3-1
Basic Data Types .....	3-2
BOOLEAN Types .....	3-3
ACCESS_TYPE Type Enumerated Data Type .....	3-4
API Parameters Structure .....	3-5
Bus Index Enum Data Type .....	3-7
Device Location Data Type .....	3-8
DMA Channel Descriptor Structure .....	3-9
DMA Channel Enum Data Type .....	3-12

DMA Channel Priority Enum Data Type .....	3-13
DMA Command Enum Data Type .....	3-14
DMA Transfer Element Structure.....	3-15
EEPROM Type Enum Data Type .....	3-16
Hot Swap Status Definition .....	3-17
IOP Arbitration Descriptor Structure .....	3-18
IOP Bus Properties Structure .....	3-20
IOP Endian Descriptor Structure .....	3-22
IOP Space Enum Data Type .....	3-23
Mailbox ID Enum Data Type.....	3-24
New Capabilities Flags .....	3-25
PCI Bus Properties Structure.....	3-26
PLX Physical Memory Structure .....	3-29
PCI Space Enum Data Type.....	3-30
PLX Device Key Structure .....	3-31
PLX Device Object Structure .....	3-32
PLX Interrupt Structure .....	3-33
Power Level Data Type .....	3-38
Power Management Properties Structure .....	3-39
<b>Appendix A PLX SDK Revision Notes.....</b>	<b>3-41</b>
A.1 Host API Functions Added .....	3-41
A.2 Host API Functions Removed .....	3-42
A.3 The Structure of the Defined Type .....	3-43
A.4 Function Argument or Type Changed .....	3-43



# 1 General Information

The PLX PCI SDK has been used by numerous customers for a number of years. Starting with PLX SDK 5.0, PCI SDK v4.40 was merged with the PLX PEX SDK v2.0. To simplify development and resolve API limitations, the PLX SDK now supports a single unified API. This new API was introduced in PCI SDK v4.20 to support PLX 6000-series devices and also used in the PEX SDK to support 8000-series devices.

Starting with PLX SDK 5.0, the new PLX API also supports 9000-series devices. The original API found in PLX PCI SDK v4.40 and previous is now considered the **Legacy API** and is no longer supported. All the functionality of the Legacy API will be covered by the new PLX API.

## 1.1 About This Manual

This manual is provided as a reference only for the legacy API for existing applications. Applications written with PLX SDK v5.0 and higher must use the new API. PLX recommends any existing legacy API application should be ported to use the new API, if possible.



## 2 PLX 9000 Legacy API

This section provides the PLX 9000 Legacy API reference.

### 2.1 PLX 9000 API Conversion Table

The XTable 2-1X lists the Legacy API functions and the new API function that should be used instead. In most cases, the new API will provide the same functionality as the legacy API function. In some case, additional code may be required to exactly match the functionality of the legacy API call.

Legacy API Function	Page	New API Function to Use
PlxBuslopRead	2-3	PlxPci_PciBarSpaceRead
PlxBuslopWrite	2-6	PlxPci_PciBarSpaceWrite
PlxChipTypeGet	2-9	PlxPci_ChipTypeGet
PlxDmaBlockChannelClose	2-11	PlxPci_DmaChannelClose
PlxDmaBlockChannelOpen	2-13	PlxPci_DmaChannelOpen
PlxDmaBlockTransfer	2-15	PlxPci_DmaTransferBlock
PlxDmaControl	2-18	PlxPci_DmaControl
PlxDmaSglChannelClose	2-20	PlxPci_DmaChannelClose
PlxDmaSglChannelOpen	2-22	PlxPci_DmaChannelOpen
PlxDmaSglTransfer	2-24	PlxPci_DmaTransferUserBuffer
PlxDmaStatus	2-27	PlxPci_DmaStatus
PlxDriverVersion	2-29	PlxPci_DriverVersion
PlxHotSwapNcpRead	2-31	PlxPci_PciRegisterReadFast
PlxHotSwapStatus	2-32	PlxPci_PciRegisterReadFast
PlxIntrAttach	2-34	PlxPci_NotificationRegisterFor
PlxIntrDisable	2-35	PlxPci_InterruptDisable
PlxIntrEnable	2-37	PlxPci_InterruptEnable
PlxIntrStatusGet	2-39	PlxPci_NotificationStatus
PlxIntrWait	2-41	PlxPci_NotificationWait
PlxIoPortRead	2-42	PlxPci_IoPortRead
PlxIoPortWrite	2-44	PlxPci_IoPortWrite
PlxMulnboundPortRead	2-46	PlxPci_PlxRegisterRead
PlxMulnboundPortWrite	2-48	PlxPci_PlxRegisterWrite
PlxMuOutboundPortRead	2-50	PlxPci_PlxRegisterRead
PlxMuOutboundPortWrite	2-52	PlxPci_PlxRegisterWrite
PlxNotificationCancel	2-54	PlxPci_NotificationCancel
PlxNotificationRegisterFor	2-56	PlxPci_NotificationRegisterFor
PlxNotificationWait	2-58	PlxPci_NotificationWait
PlxPciBarGet	2-60	PlxPci_PciBarProperties
PlxPciBarMap	2-62	PlxPci_PciBarMap
PlxPciBarRangeGet	2-64	PlxPci_PciBarProperties
PlxPciBarUnmap	2-66	PlxPci_PciBarUnmap
PlxPciBaseAddressesGet	2-68	PlxPci_PciBarMap
PlxPciBoardReset	2-69	PlxPci_DeviceReset
PlxPciCommonBufferGet	2-70	PlxPci_CommonBufferProperties
PlxPciCommonBufferMap	2-71	PlxPci_CommonBufferMap
PlxPciCommonBufferProperties	2-73	PlxPci_CommonBufferProperties
PlxPciCommonBufferUnmap	2-75	PlxPci_CommonBufferUnmap
PlxPciConfigRegisterRead	2-77	PlxPci_PciRegisterRead

Legacy API Function	Page	New API Function to Use
PlxPciConfigRegisterWrite	2-79	PlxPci_PciRegisterWrite
PlxPciDeviceClose	2-81	PlxPci_DeviceClose
PlxPciDeviceFind	2-82	PlxPci_DeviceFind
PlxPciDeviceOpen	2-85	PlxPci_DeviceOpen
PlxPciPhysicalMemoryAllocate	2-88	PlxPci_PhysicalMemoryAllocate
PlxPciPhysicalMemoryFree	2-91	PlxPci_PhysicalMemoryFree
PlxPciPhysicalMemoryMap	2-93	PlxPci_PhysicalMemoryMap
PlxPciPhysicalMemoryUnmap	2-95	PlxPci_PhysicalMemoryUnmap
PlxPciRegisterReadFast	2-97	PlxPci_PciRegisterReadFast
PlxPciRegisterWriteFast	2-98	PlxPci_PciRegisterWriteFast
PlxPciRegisterRead_Unsupported	2-100	PlxPci_PciRegisterRead_BypassOS
PlxPciRegisterWrite_Unsupported	2-102	PlxPci_PciRegisterWrite_BypassOS
PlxPmNcpRead	2-104	PlxPci_PciRegisterReadFast
PlxPowerLevelGet	2-105	PlxPci_PciRegisterReadFast
PlxPowerLevelSet	2-107	PlxPci_PciRegisterWriteFast
PlxRegisterDoorbellRead	2-109	PlxPci_PlxRegisterRead
PlxRegisterDoorbellSet	2-111	PlxPci_PlxRegisterWrite
PlxRegisterMailboxRead	2-112	PlxPci_MailboxRead
PlxRegisterMailboxWrite	2-114	PlxPci_MailboxWrite
PlxRegisterRead	2-116	PlxPci_PlxRegisterRead
PlxRegisterWrite	2-118	PlxPci_PlxRegisterWrite
PlxSdkVersion	2-120	PlxPci_ApiVersion
PlxSerialEepromPresent	2-121	PlxPci_EepromPresent
PlxSerialEepromRead	2-123	PlxPci_EepromReadByOffset
PlxSerialEepromReadByOffset	2-124	PlxPci_EepromReadByOffset
PlxSerialEepromWrite	2-126	PlxPci_EepromWriteByOffset
PlxSerialEepromWriteByOffset	2-127	PlxPci_EepromWriteByOffset
PlxVpdNcpRead	2-129	PlxPci_PciRegisterReadFast
PlxVpdRead	2-130	PlxPci_VpdRead
PlxVpdWrite	2-131	PlxPci_VpdWrite

**Table 2-1 PLX API Conversion List**

## 2.2 PLX 9000 API Function Details

This section contains a detailed description of each function in the PLX 9000 Legacy API. The functions are listed in alphabetical order.

### PlxBusIopRead

---

#### Syntax:

```
RETURN_CODE  
PlxBusIopRead(  
    HANDLE        hDevice,  
    IOP_SPACE     IopSpace,  
    U32           address,  
    BOOLEAN       bRemap,  
    VOID          *pBuffer,  
    U32           ByteCount,  
    ACCESS_TYPE   AccessType  
);
```

#### PLX Chip Support:

All 9000 series & 8311

#### 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 *bRemap* 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 *bRemap* 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.

##### *bRemap*

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

##### *pBuffer*

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.

##### *ByteCount*

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 <i>address</i> parameter is not aligned based on the <i>accessType</i>
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the <i>accessType</i>

## 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.

For slightly better performance, use the *PlxPciBarMap()* function and access local memory from an application directly through a virtual address. This will completely bypass the driver and provide direct access to the local bus. The disadvantage to the direct method is that the application will be responsible for manually configuring the PLX chip local space re-map window. This will affect code portability, but overall performance is slightly 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.

## 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	3-23
ACCESS_TYPE	3-4

## PlxBusIopWrite

---

### Syntax:

```
RETURN_CODE
PlxBusIopWrite(
    HANDLE        hDevice,
    IOP_SPACE     IopSpace,
    U32           address,
    BOOLEAN       bRemap,
    VOID          *pBuffer,
    U32           ByteCount,
    ACCESS_TYPE   AccessType
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function writes 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 *bRemap* 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 *bRemap* 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.

#### *bRemap*

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

#### *pBuffer*

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

#### *ByteCount*

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 <i>address</i> parameter is not aligned based on the <i>accessType</i>
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the <i>accessType</i>

## 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.

For slightly better performance, use the *PlxPciBarMap()* function and access local memory from an application directly through a virtual address. This will completely bypass the driver and provide direct access to the local bus. The disadvantage to the direct method is that the application will be responsible for manually configuring the PLX chip local space re-map window. This will affect code portability, but overall performance is slightly 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.

## 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	3-23
ACCESS_TYPE	3-4

## PlxChipTypeGet

---

### Syntax:

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

### PLX Chip Support:

All 9000 series & 8311

### Description:

Given a handle of an open PCI device, this function stores the chip type and chip revision to memory.

### 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 example, 0x9054 = 9054 and 0x9656 = 9656.

For the 9052 chip, the chip type returned is 9050, but the revision is 2. A revision value of 1 is returned for the 9050.

## 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 == 0xAA)
                // Chip is an AA version
            if (Revision == 0xAB)
                // Chip is an AB version
            if (Revision == 0xAC)
                // Chip is an AC version
            break;

        case 0x9656:
            // Chip is 9656
            break;
    }
}
```

## PlxDmaBlockChannelClose

---

### Syntax:

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

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

Closes a DMA channel previously opened for Block mode.

### Parameters:

*hDevice*  
Handle of an open PCI device

*channel*  
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 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	3-12

## PlxDmaBlockChannelOpen

---

### Syntax:

```
RETURN_CODE  
PlxDmaBlockChannelOpen(  
    HANDLE                hDevice,  
    DMA_CHANNEL           channel,  
    DMA_CHANNEL_DESC *pDesc  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

Opens and initializes a DMA channel for Block DMA transfers.

### Parameters:

*hDevice*

Handle of an open PCI device

*channel*

The DMA channel to open

*pDesc*

Pointer to a 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 *pDesc* 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 to open DMA channel
}
```

## Cross Reference:

Referenced Item	Page
DMA_CHANNEL	3-12
DMA_CHANNEL_DESC	3-9

## PlxDmaBlockTransfer

---

### Syntax:

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

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

Starts a Block DMA transfer for a given DMA channel.

### Parameters:

*hDevice*

Handle of an open PCI device

*channel*

The DMA channel to use for transfer

*dmaData*

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 channel is 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 PciCommonBufferProperties()*) 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
u.UserVa	Ignored.
u.PciAddrLow	Lower 32-bits of PCI address of the PCI buffer.
PciAddrHigh	High 32-bits of PCI address of the PCI buffer, for chips supporting Dual Address cycles
LocalAddr	The Local address for the transfer
TransferCount	The number of bytes to transfer.
TerminalCountIntr	Ignored
LocalToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)

**Usage:**

```

HANDLE          hDevice;
PLX_PHYSICAL_MEM PciBuffer;
DMA_TRANSFER_ELEMENT DmaData;

// Get Common buffer information
rc =
    PlxPciCommonBufferProperties(
        hDevice,
        &PciBuffer
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to get Common Buffer information
}

```

```

// Fill in DMA transfer parameters
DmaData.u.PciAddrLow      = PciBuffer.PhysicalAddr;
DmaData.PciAddrHigh      = 0x0;
DmaData.LocalAddr        = 0x0;
DmaData.TransferCount    = 0x1000;
DmaData.LocalToPciDma    = 1;

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	3-12
DMA_TRANSFER_ELEMENT	3-15

# PlxDmaControl

---

## Syntax:

```
RETURN_CODE  
PlxDmaControl(  
    HANDLE        hDevice,  
    DMA_CHANNEL   channel,  
    DMA_COMMAND   command  
);
```

## PLX Chip Support:

9054, 9056, 9080, 9656, 8311

## Description:

Controls the DMA engine for a given DMA channel.

## Parameters:

*hDevice*

Handle of an open PCI device

*channel*

A previously opened DMA channel

*command*

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.
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,
        TRUE           // Don't 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	3-12
DMA_COMMAND	3-14

## PlxDmaSglChannelClose

---

### Syntax:

```
RETURN_CODE  
PlxDmaSglChannelClose(  
    HANDLE        hDevice,  
    DMA_CHANNEL  channel  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

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

### Parameters:

*hDevice*

Handle of an open PCI device

*channel*

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	3-12

## PlxDmaSglChannelOpen

---

### Syntax:

```
RETURN_CODE  
PlxDmaSglChannelOpen(  
    HANDLE          hDevice,  
    DMA_CHANNEL     channel,  
    DMA_CHANNEL_DESC *pDesc  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

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

### Parameters:

*hDevice*

Handle of an open PCI device

*channel*

The DMA channel to open

*pDesc*

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 the 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 to open DMA channel
}
```

## Cross Reference:

Referenced Item	Page
DMA_CHANNEL	3-12
DMA_CHANNEL_DESC	3-9

## PlxDmaSglTransfer

---

### Syntax:

```
RETURN_CODE  
PlxDmaSglTransfer(  
    HANDLE                hDevice,  
    DMA_CHANNEL           channel,  
    DMA_TRANSFER_ELEMENT *dmaData,  
    BOOLEAN               returnImmediate  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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. This function works as follows:

The PLX driver takes the provided user-mode buffer and page-locks 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.

This function cannot be used to build SGL descriptors. The descriptors are built automatically by the driver based upon the fragmentation of the user-mode buffer in physical memory.

### Parameters:

*hDevice*

Handle of an open PCI device

*channel*

The DMA channel to use for the transfer

*dmaData*

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 channel is 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
ApiDmaSglBuildFailed	Insufficient internal driver memory to store the SGL descriptors
ApiFailed	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()*.

The driver will always enable the DMA channel interrupt when this function is used. This is required so the driver can perform cleanup routines, such as unlock the buffer and release descriptors, after the 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
u.UserAddr	User address of the PCI buffer
u.PciAddrLow	Ignored
PciAddrHigh	Ignored
LocalAddr	The Local address for the transfer
TransferCount	The number of bytes to transfer
TerminalCountIntr	Ignored
LocalToPciDma	Direction of the transfer. (0 = PCI-to-Local, 1 = Local-to-PCI)

## Usage:

```
U8                buffer[0x1000];
HANDLE            hDevice;
DMA_TRANSFER_ELEMENT DmaData;

DmaData.u.UserAddr      = (U32)buffer;
DmaData.LocalAddr       = 0x0;
DmaData.TransferCount   = sizeof(buffer);
DmaData.LocalToPciDma   = 1;

rc =
    PlxDmaSglTransfer(
        hDevice,
        PrimaryPciChannel0,
        &DmaData,
        FALSE                // Wait for completion
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to transfer buffer
}
```

## Cross Reference:

Referenced Item	Page
DMA_CHANNEL	3-12
DMA_TRANSFER_ELEMENT	3-15

## PlxDmaStatus

---

### Syntax:

```
RETURN_CODE  
PlxDmaStatus(  
    HANDLE        hDevice,  
    DMA_CHANNEL  channel  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### Description:

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

### Parameters:

*hDevice*  
Handle of an open PCI device

*channel*  
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 =
    PlxDmaBlockTransfer(
        hDevice,
        PrimaryPciChannel0,
        &DmaData,
        TRUE           // Don't wait for DMA completion
    );

// 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	3-12

## PlxDriverVersion

---

### Syntax:

```
RETURN_CODE  
PlxDriverVersion(  
    HANDLE  hDevice,  
    U8      *VersionMajor,  
    U8      *VersionMinor,  
    U8      *VersionRevision  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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
    );
}
```

## PlxHotSwapNcpRead

---

### Syntax:

```
U8
PlxHotSwapNcpRead(
    HANDLE      hDevice,
    RETURN_CODE *pReturnCode
);
```

### PLX Chip Support:

9030, 9054, 9056, 9656, 8311

### 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:

9030, 9054, 9056, 9656, 8311

### 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  
);
```

### NOTE

*This function has been deprecated and replaced with the **PlxNotificationXxx** API functions. Please refer to these functions for interrupt notification.*

## PlxIntrDisable

---

### Syntax:

```
RETURN_CODE  
PlxIntrDisable(  
    HANDLE    hDevice,  
    PLX_INTR  *pPlxIntr  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Disables specific interrupt(s) of a PCI device containing a PLX chip.

### Parameters:

*hDevice*

Handle of an open PCI device

*pPlxIntr*

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	3-31

## PlxIntrEnable

---

### Syntax:

```
RETURN_CODE  
PlxIntrEnable(  
    HANDLE    hDevice,  
    PLX_INTR  *pPlxIntr  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Enables specific interrupt(s) of a PCI device containing a PLX chip.

### Parameters:

*hDevice*

Handle of an open PCI device

*pPlxIntr*

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	3-31

## PlxIntrStatusGet

---

### Syntax:

```
RETURN_CODE  
PlxIntrStatusGet(  
    HANDLE    hDevice,  
    PLX_INTR  *pPlxIntr  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Returns the status of the most recent interrupt(s) to occur.

### Parameters:

*hDevice*

Handle of an open PCI device

*pPlxIntr*

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	3-31

## PlxIntrWait

---

### Syntax:

```
RETURN_CODE  
PlxIntrWait(  
    HANDLE hDevice,  
    HANDLE hEvent,  
    U32     Timeout_ms  
);
```

### NOTE

*This function has been deprecated and replaced with the **PlxNotificationXxx** API functions. Please refer to these functions for interrupt notification.*

## PlxIoPortRead

---

### Syntax:

```
RETURN_CODE  
PlxIoPortRead(  
    HANDLE        hDevice,  
    U32           IoPort,  
    ACCESS_TYPE   AccessType,  
    VOID          *pValue  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Reads a value from an I/O port.

### Parameters:

#### *hDevice*

Handle of an open PCI device

#### *IoPort*

The I/O port address to read from

#### *AccessType*

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

#### *pValue*

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()*.

## 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	3-4

# PlxIoPortWrite

---

## Syntax:

```
RETURN_CODE  
PlxIoPortWrite(  
    HANDLE        hDevice,  
    U32           IoPort,  
    ACCESS_TYPE   AccessType,  
    VOID          *pValue  
);
```

## PLX Chip Support:

All 9000 series & 8311

## Description:

Writes a value to an I/O port.

## Parameters:

*hDevice*

Handle of an open PCI device

*IoPort*

The I/O port address to write to

*AccessType*

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()*.

## 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	3-4

## PlxMuInboundPortRead

---

### Syntax:

```
RETURN_CODE  
PlxMuInboundPortRead(  
    HANDLE  hDevice,  
    U32     *pFrame  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*pFrame*

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     *pFrame  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*pFrame*

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     *pFrame  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*pFrame*

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     *pFrame  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*pFrame*

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
}
```

## PlxNotificationCancel

---

### Syntax:

```
RETURN_CODE  
PlxNotificationCancel(  
    HANDLE          hDevice,  
    PLX_NOTIFY_OBJECT *pEvent  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function cancels a notification object previously registered with *PlxNotificationRegisterFor*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pEvent*

A pointer to a previously registered PLX notification object.

### 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
ApiFailed	The notification object is not valid or not registered

### Notes:

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

### Usage:

```
HANDLE          hDevice;  
PLX_INTR        IntSources;  
RETURN_CODE     rc;  
PLX_NOTIFY_OBJECT Event;  
  
// Clear interrupt sources  
memset(  
    &IntSources,  
    0,  
    sizeof(PLX_INTR)  
);
```

```

// Register for Local->PCI doorbell interrupt notification
IntSources.PciDoorbell = 1;

rc =
    PlxNotificationRegisterFor(
        hDevice,
        &IntSources,
        &Event
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to register interrupt notification
}

// Wait for the interrupt
rc =
    PlxNotificationWait(
        hDevice,
        &Event,
        10 * 1000           // 10 second timeout
    );

switch (rc)
{
    case ApiSuccess:
        // Interrupt occurred
        break;

    case ApiWaitTimeout:
        // ERROR - Timeout waiting for Interrupt Event
        break;

    case ApiWaitCanceled:
        // ERROR - Event not registered for wait
        break;
}

// Cancel interrupt notification
rc =
    PlxNotificationCancel(
        hDevice,
        &Event
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to cancel interrupt notification
}

```

## PlxNotificationRegisterFor

---

### Syntax:

```
RETURN_CODE  
PlxNotificationRegisterFor(  
    HANDLE          hDevice,  
    PLX_INTR        *pPlxIntr,  
    PLX_NOTIFY_OBJECT *pEvent  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function is used by applications that need to wait for a specific interrupt(s) to occur. The function registers the event and the desired interrupt source(s). It is used in conjunction with *PlxNotificationWait*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pPlxIntr*

A pointer to a structure containing the sources of interrupts that the application would like to be notified of. An event will occur if ANY one of the registered interrupts occurs.

*pEvent*

A pointer to a PLX notification object that can be used with *PlxNotificationWait*.

### 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()*.

Once the registration is complete, the event will continue to signal until it is cancelled. There is no need to continuously re-register for notification.

This function does **not** actually enable interrupt(s). It only registers for interrupt notification with the PLX driver. To enable an interrupt(s), refer to *PlxIntrEnable*.

### Usage:

```
HANDLE          hDevice;  
PLX_INTR        IntSources;  
RETURN_CODE     rc;  
PLX_NOTIFY_OBJECT Event;
```

```

// Clear interrupt sources
memset(
    &IntSources,
    0,
    sizeof(PLX_INTR)
);

// Register for DMA Channel 1 interrupt notification
IntSources.PciDmaChannell = 1;

rc =
    PlxNotificationRegisterFor(
        hDevice,
        &IntSources,
        &Event
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to register interrupt notification
}

/*****
 * Perform DMA transfer here.
 * Refer to DMA functions documentation
 *****/

// Wait for the DMA interrupt
rc =
    PlxNotificationWait(
        hDevice,
        &Event,
        10 * 1000          // 10 second timeout
    );

switch (rc)
{
    case ApiSuccess:
        // DMA Interrupt occurred
        break;

    case ApiWaitTimeout:
        // ERROR - Timeout waiting for Interrupt Event
        break;

    case ApiWaitCanceled:
    case ApiFailed:
    default:
        // ERROR - Failed while waiting for interrupt
        break;
}

```

**Cross Reference:**

Referenced Item	Page
PLX_INTR	3-31

## PlxNotificationWait

---

### Syntax:

```
RETURN_CODE
PlxNotificationWait(
    HANDLE                hDevice,
    PLX_NOTIFY_OBJECT *pEvent
    U32                   Timeout_ms;
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function is used to wait for a specific interrupt(s) to occur or a timeout. It will wait for the event(s) to occur which was registered with *PlxNotificationRegisterFor*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pEvent*

A pointer to a PLX notification object that can be used with *PlxNotificationWait*.

*Timeout\_ms*

The desired time to wait, in milliseconds, for the event to occur

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiFailed	The notification object is not valid or not registered
ApiWaitTimeout	Reached timeout waiting for event
ApiWaitCanceled	Wait event was cancelled

### Notes:

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

For the Linux OS, there is no support for an infinite wait. The largest 32-bit value (FFFF\_FFFFh) may be used as the timeout, which will lead to a significant wait on the order of weeks. An application can easily implement an infinite wait by calling the function again if *ApiWaitTimeout* is returned.

In Windows, you may pass the value *INFINITE* for the parameter *Timeout\_ms*. *INFINITE* is a Win32 defined constant that specifies an infinite wait for an event. *INFINITE* is not defined in Linux.

## Usage:

```
HANDLE          hDevice;
PLX_INTR        IntSources;
RETURN_CODE     rc;
PLX_NOTIFY_OBJECT Event;

// Clear interrupt sources
memset(
    &IntSources,
    0,
    sizeof(PLX_INTR)
);

// Register for Local->PCI doorbell interrupt notification
IntSources.PciDoorbell = 1;

rc =
    PlxNotificationRegisterFor(
        hDevice,
        &IntSources,
        &Event
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to register interrupt notification
}

// Wait for the interrupt
rc =
    PlxNotificationWait(
        hDevice,
        &Event,
        10 * 1000          // 10 second timeout
    );

switch (rc)
{
    case ApiSuccess:
        // Interrupt occurred
        break;

    case ApiWaitTimeout:
        // ERROR - Timeout waiting for Interrupt Event
        break;

    case ApiWaitCanceled:
        // ERROR - Event not registered for wait
        break;
}
```

## PlxPciBarGet

---

### Syntax:

```
RETURN_CODE  
PlxPciBarGet(  
    HANDLE    hDevice,  
    U8        BarIndex,  
    U32       *pPciBar,  
    BOOLEAN   *pFlag_IsIoSpace  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function returns the PCI address of a specified PCI Base Address Register (BAR). It will also specify whether the space is of type Memory or I/O.

### Parameters:

#### *hDevice*

Handle of an open PCI device

#### *BarIndex*

The index of the PCI BAR to query. Valid values are in the range 0-5.

#### *pPciBar*

Pointer to a 32-bit buffer which will contain the PCI base address.

#### *pFlag\_IsIoSpace*

Will be TRUE if the PCI space is of type I/O and FALSE if the PCI space is of type Memory.

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidIndex	PCI BAR index is not in the range of valid values

### Notes:

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

## Usage:

```
U8          i;
U32         value;
HANDLE      hDevice;
BOOLEAN     bIsIoSpace;
RETURN_CODE rc;

for (i = 0; i <= 6; i++)
{
    rc =
        PlxPciBarGet(
            hDevice,
            i,
            &value,
            &bIsIoSpace
        );

    if (rc != ApiSuccess)
    {
        // Error - Unable to read PCI BAR
    }
    else
    {
        printf(
            "PCI BAR %d = %08x (type = ",
            i, value
        );
        if (bIsIoSpace)
            printf("I/O\n");
        else
            printf("Memory\n");
    }
}
}
```

# PlxPciBarMap

---

## Syntax:

```
RETURN_CODE  
PlxPciBarMap(  
    HANDLE  hDevice,  
    U8      BarIndex,  
    VOID    *pVa  
);
```

## PLX Chip Support:

All 9000 series & 8311

## Description:

This function will map a PCI BAR into user virtual space and return the virtual address. User applications may then bypass the driver and directly access a PCI space for optimal performance.

## Parameters:

*hDevice*

Handle of an open PCI device

*BarIndex*

The index of the PCI BAR to map. Valid values are in the range 0-5.

*pVa*

Pointer to a buffer which will contain the base virtual address

## 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
ApiInvalidIndex	PCI BAR index is not in the range of valid values
ApiFailed	Virtual address mapping failed
ApiInvalidPciSpace	PCI space is of type I/O, not memory
ApiInvalidAddress	The PCI space does not contain a valid PCI address or is disabled
ApiInsufficientResources	The driver was not able to map the space due to insufficient OS resources

## Notes:

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

Virtual mappings consume Page-Table Entries (PTEs), which are a limited resource in the OS. The OS will fail a mapping attempt if the number of available PTEs is insufficient to complete the mapping. As the size of a PCI space gets larger (i.e. 4MB or more), the number of PTEs required increases, resulting in a greater risk of a failed mapping attempt.

It is important to un-map a PCI Space when the virtual address is no longer needed. This should always be done before the device is released with *PlxPciDeviceClose*. Un-mapping a space will release the PTE resources used back to the OS. Refer to *PlxPciBarUnmap*.

The virtual address will cease to be valid after the device is closed. Attempts to use the virtual address after closing a device will result in exceptions.

The PCI space, which is to be mapped into user virtual space, must be of type Memory. Mapping of I/O type spaces is not allowed. I/O type spaces can be accessed with *PlxIoPortRead* and *PlxIoPortWrite*.

### Usage:

```
U8          i;
U32         value;
U32         Va[6];
HANDLE      hDevice;
RETURN_CODE rc;

for (i = 0; i <= 5; i++)
{
    rc =
        PlxPciBarMap(
            hDevice,
            i,
            &(Va[i])
        );

    if (rc != ApiSuccess)
    {
        // Error - Unable to map PCI bar into virtual space
    }
}

printf(
    "    BAR 0 VA:  0x%08x\n",
    "    BAR 1 VA:  0x%08x\n",
    "    BAR 2 VA:  0x%08x\n",
    "    BAR 3 VA:  0x%08x\n",
    "    BAR 4 VA:  0x%08x\n",
    "    BAR 5 VA:  0x%08x\n",
    Va[0], Va[1], Va[2], Va[3], Va[4], Va[5]
);

/*****
 * NOTE:  The configuration of a PCI Space is left to the application.
 *        The space must be configured correctly before accessing it.
 *****/

// Read a 32-bit value from Space 0 (For 9054, Space 0 is at PCI BAR 2)
value = *(U32*)Va[2];

// Set bit 7 in local register 1Ch (PLX registers are at BAR 0)
Value = *(U32*)(Va[0] + 0x1c);

// Set bit 7
Value = Value | (1 << 7);

// Write register
*(U32*)(Va[0] + 0x1c) = Value;
```

# PlxPciBarRangeGet

---

## Syntax:

```
RETURN_CODE  
PlxPciBarRangeGet(  
    HANDLE  hDevice,  
    U8      BarIndex,  
    U32     *pData  
);
```

## PLX Chip Support:

All 9000 series & 8311

## Description:

Retrieves the size of any PCI-to-Local space window.

## Parameters:

*hDevice*

Handle of an open PCI device

*BarIndex*

The index of the PCI BAR to get. Valid values are in the range 0-5.

*pData*

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;
U8           PciBarNum;
RETURN_CODE rc;

for (BarIndex = 0; BarIndex <= 5; BarIndex++)
{
    rc =
        PlxPciBarRangeGet(
            hDevice,
            BarIndex,
            &size
        );

    if (rc != ApiSuccess)
    {
        // ERROR - Unable to get PCI space size
    }
    else
    {
        PlxPrintf(
            "BAR %d: %d bytes",
            BarIndex, size
        );
    }
}
```

## PlxPciBarUnmap

---

### Syntax:

```
RETURN_CODE  
PlxPciBarUnmap(  
    HANDLE hDevice,  
    VOID *pVa  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function unmaps a previously mapped PCI BAR from user virtual space.

### Parameters:

*hDevice*

Handle of an open PCI device

*pVa*

Pointer to virtual address of the PCI BAR to unmap. (The address previously obtained from *PlxPciBarMap*)

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	The virtual address is invalid or not a previously mapped address

### Notes:

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

The virtual address must be an address previously obtained with a call to *PlxPciBarMap*.

This function must be called before a device is released with *PlxPciDeviceClose*. The virtual address will cease to be valid after the device is closed.

## Usage:

```
U32          Va;
HANDLE      hDevice;
RETURN_CODE rc;

// Map PCI BAR 0 for register access
rc =
    PlxPciBarMap(
        hDevice,
        0,
        &Va
    );

if (rc != ApiSuccess)
{
    // Error - Unable to map PCI bar into virtual space
}

//
// Access registers as needed ...
//

// Unmap the space
rc =
    PlxPciBarUnmap(
        hDevice,
        &Va
    );

if (rc != ApiSuccess)
{
    // Error - Unable to unmap PCI BAR from virtual space
}
```

## PlxPciBaseAddressesGet

---

### Syntax:

```
RETURN_CODE  
PlxPciBaseAddressesGet(  
    HANDLE          hDevice,  
    VIRTUAL_ADDRESSES *pVirtAddr  
);
```

### NOTE

*This function has been deprecated and replaced with the **PlxPciBarMap** and **PlxPciBarUnmap** API functions. Please refer to these functions for user-mode virtual address mappings to PCI BAR spaces.*

## PlxPciBoardReset

---

### Syntax:

```
VOID  
PlxPciBoardReset(  
    HANDLE hDevice  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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()*.

This function utilizes the *Software Reset* feature of PLX chips. The reset sequence is dependent upon the PLX chip type. It typically involves save and restore of some registers. The API call will also issue a manual reload of the EEPROM values, if one exists, since the PLX chip will revert to default values after software reset.

### Usage:

```
HANDLE hDevice;  
  
// A local bus device is crashed, reset the board  
PlxPciBoardReset(  
    hDevice  
);
```

## PlxPciCommonBufferGet

---

### Syntax:

```
RETURN_CODE  
PlxPciCommonBufferGet(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo  
);
```

### NOTE

*This function has been deprecated and replaced with **PlxPciCommonBufferProperties** & **PlxPciCommonBufferMap** API functions. Please refer to these functions for common buffer access.*

## PlxPciCommonBufferMap

---

### Syntax:

```
RETURN_CODE  
PlxPciCommonBufferMap(  
    HANDLE  hDevice,  
    VOID    *pVa  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function will map the common buffer into user virtual space and return the base virtual address.

### Parameters:

*hDevice*

Handle of an open PCI device

*pVa*

A pointer to a buffer to hold the virtual address

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	Buffer address is invalid
ApiInsufficientResources	Insufficient resources for perform a mapping of the buffer
ApiFailed	Buffer was not allocated properly

### Notes:

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

Mapping of the common buffer into user virtual space may fail due to insufficient Page-Table Entries (PTEs). The larger the buffer size, the greater the number of PTEs required to map it into user space.

The buffer should be unmapped before calling *PlxPciDeviceClose* to close the device. The virtual address will cease to be valid after closing the device or after unmapping the buffer. Refer to *PlxPciCommonBufferUnmap*.

## Usage:

```
U8          Value;
VOID        *pBuffer;
HANDLE      hDevice;
RETURN_CODE rc;
PLX_PHYSICAL_MEM BufferInfo;

// Get the common buffer information
rc =
    PlxPciCommonBufferProperties(
        hDevice,
        &BufferInfo
    );

if (rc != ApiSucess)
{
    // Error - Unable to get common buffer properties
}

// Map the buffer into user space
rc =
    PlxPciCommonBufferMap(
        hDevice,
        &BufferVa
    );

if (rc != ApiSucess)
{
    // Error - Unable to map common buffer to user virtual space
}

// Write 32-bit value to buffer
*(U32*)(BufferVa + 0x100) = 0x12345;

// Read 8-bit value from buffer
value = *(U8*)(BufferVa + 0x54);
```

## PlxPciCommonBufferProperties

---

### Syntax:

```
RETURN_CODE  
PlxPciCommonBufferProperties(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function returns the common buffer properties.

### Parameters:

*hDevice*

Handle of an open PCI device

*pMemoryInfo*

A pointer to a PLX\_PHYSICAL\_MEM structure which will contain information about the common buffer

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
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()*.

This function will only return properties of the common buffer. It will not provide a virtual address for the buffer. Use *PlxPciCommonBufferMap* to get a virtual address.

PLX drivers allocate a common buffer for use by applications. The buffer size requested is determined by a PLX registry entry (*refer to the registry entries section of the PLX User's Manual*). The driver will attempt to allocate the buffer, but the operating system determines the success of the attempt based upon available system resources. PLX drivers will re-issue the request for a smaller-sized buffer until the call succeeds.

The common buffer is guaranteed to be physically contiguous and page-locked in memory so that it may be used for operations such as DMA. PLX drivers do not use the common buffer for any functionality. Its use is reserved for applications.

Coordination and management of access to the buffer between multiple processes or threads is left to applications. Care must be taken to avoid shared memory issues.

## Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
PLX_PHYSICAL_MEM BufferInfo;

// Get the common buffer information
rc =
    PlxPciCommonBufferProperties(
        hDevice,
        &BufferInfo
    );

if (rc != ApiSucess)
{
    // Error - Unable to get common buffer properties
}

printf(
    "Common buffer information:\n"
    "    Bus Physical Addr:  %08lx\n"
    "    CPU Physical Addr:  %08lx\n"
    "    Size                 :  %d bytes\n",
    BufferInfo.PhysicalAddr,
    BufferInfo.CpuPhysical,
    BufferInfo.Size
);
```

## Cross Reference:

Referenced Item	Page
PLX_PHYSICAL_MEM	3-29

## PlxPciCommonBufferUnmap

---

### Syntax:

```
RETURN_CODE  
PlxPciCommonBufferUnmap(  
    HANDLE hDevice,  
    VOID *pVa  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function will unmap the common buffer from user virtual space.

### Parameters:

*hDevice*

Handle of an open PCI device

*pVa*

The virtual address of the common buffer originally obtained from *PlxPciCommonBufferMap*

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	Virtual address is invalid or buffer was not allocated properly
ApiFailed	The buffer to unmap is not valid

### Notes:

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

It is important to unmap the common buffer when it is no longer needed to release mapping resources back to the system.

The buffer should be un-mapped before calling *PlxPciDeviceClose* to close the device. The virtual address will cease to be valid after closing the device or after un-mapping the buffer.

## Usage:

```
VOID                *pBuffer;
HANDLE              hDevice;
RETURN_CODE         rc;
PLX_PHYSICAL_MEM   BufferInfo;

// Get the common buffer information
rc =
    PlxPciCommonBufferProperties(
        hDevice,
        &BufferInfo
    );

if (rc != ApiSucess)
{
    // Error - Unable to get common buffer properties
}

// Map the buffer into user space
rc =
    PlxPciCommonBufferMap(
        hDevice,
        &pBuffer
    );

if (rc != ApiSucess)
{
    // Error - Unable to map common buffer to user virtual space
}

//
// Use the common buffer as needed
//

// Unmap the buffer from user space
rc =
    PlxPciCommonBufferUnmap(
        hDevice,
        &BufferVa
    );

if (rc != ApiSucess)
{
    // Error - Unable to unmap common buffer from user virtual space
}
```

## PlxPciConfigRegisterRead

---

### Syntax:

```
U32
PlxPciConfigRegisterRead(
    U8          bus,
    U8          slot,
    U16         offset,
    RETURN_CODE *pReturnCode
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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

*offset*

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:

For faster access to the PCI registers of a device that is already selected, refer to the function *PlxPciRegisterReadFast*.

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
            );
        }
    }
}
}
```

## PlxPciConfigRegisterWrite

---

### Syntax:

```
RETURN_CODE  
PlxPciConfigRegisterWrite(  
    U8    bus,  
    U8    slot,  
    U16   offset,  
    U32   *pValue  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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

*offset*

Offset of the PCI configuration register write

*pValue*

A pointer to a 32-bit buffer which contains the value 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:

For faster access to the PCI registers of a device that is already selected, refer to the function *PlxPciRegisterWriteFast*.

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:

```
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,          // PCI Command/Status register
        &rc
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read PCI configuration register
}

// Check for any PCI 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 *pDevice,  
    U32              *pRequestLimit  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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:

#### *pDevice*

A pointer to a DEVICE\_LOCATION structure containing the search criteria and/or to contain device information once it is found.

#### *pRequestLimit*

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 *pRequestLimit* contains the value FIND\_AMOUNT\_MATCHED, the DEVICE\_LOCATION structure should contain the search criteria.

If *pRequestLimit* contains a value other than FIND\_AMOUNT\_MATCHED, the DEVICE\_LOCATION structure will be filled in with information about the device indexed at *pRequestLimit* (numbering starts at 0).

When setting the elements of the DEVICE\_LOCATION structure with search criteria, any element not to be used in the search should be set 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      = (U8)-1;
Device.SlotNumber     = (U8)-1;
Device.VendorId       = (U16)-1;
Device.DeviceId       = (U16)-1;
Device.SerialNumber[0] = '\0';

rc =
    PlxPciDeviceFind(
        &Device,
        &ReqLimit
    );

if ((rc != ApiSuccess) || (ReqLimit == 0))
{
    // ERROR - Unable to locate any valid devices
}

ReqLimit = 0;

// Search for the first device matching a specific Vendor ID
Device.BusNumber      = (U8)-1;
Device.SlotNumber     = (U8)-1;
Device.VendorId       = 0x10b5;    // PLX Vendor ID
Device.DeviceId       = (U16)-1;
Device.SerialNumber[0] = '\0';

rc =
    PlxPciDeviceFind(
        &Device,
        &ReqLimit
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to locate 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	3-8

## PlxPciDeviceOpen

---

### Syntax:

```
RETURN_CODE  
PlxPciDeviceOpen(  
    DEVICE_LOCATION *pDevice,  
    HANDLE          *pHandle  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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

*pHandle*

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
ApiInvalidDriverVersion	Driver version does not match the API library version.

### 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.

When setting the elements of the `DEVICE_LOCATION` structure with search criteria, any element not to be used in the search should be set 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 will be selected. All remaining fields are then filled in.

## Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
DEVICE_LOCATION Device;

// Select the first PLX device found
Device.BusNumber      = (U8)-1;
Device.SlotNumber     = (U8)-1;
Device.DeviceId       = (U16)-1;
Device.VendorId       = (U16)-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      = (U8)-1;
Device.SlotNumber     = (U8)-1;
Device.DeviceId       = (U16)-1;
Device.VendorId       = (U16)-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 specific device by physical location
Device.BusNumber      = 0x01;
Device.SlotNumber     = 0x0f;
Device.DeviceId       = (U16)-1;
Device.VendorId       = (U16)-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	3-8

# PlxPciPhysicalMemoryAllocate

---

## Syntax:

```
RETURN_CODE  
PlxPciPhysicalMemoryAllocate(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo,  
    BOOLEAN         bSmallerOk  
);
```

## PLX Chip Support:

All 9000 series & 8311

## Description:

This function will attempt to allocate a physically contiguous, page-locked buffer.

## Parameters:

### *hDevice*

Handle of an open PCI device

### *pMemoryInfo*

A pointer to a PLX\_PHYSICAL\_MEM structure will contain the buffer information. The requested size of the buffer to allocate should be set in this structure before making the call. The actual size of the allocated buffer will be specified in the same field when the call returns.

### *bSmallerOk*

Flag to specify whether a buffer of size smaller than specified is acceptable

- If FALSE, the driver will return an error if the buffer allocation fails
- If TRUE and the allocation fails, the driver will reattempt to allocate the buffer, but decrement the size each time until the allocation succeeds.

## Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInsufficientResources	Insufficient resource to allocate buffer

## Notes:

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

The allocation of a physically contiguous page-locked buffer is dependent upon system resources and the fragmentation of memory. This type of memory is typically a limited resource in OS environments. As a result, allocation of large size buffers (> 512k) may fail.

In current versions of Linux, the size of a buffer is additionally limited. In Linux kernel version 2.2, 2MB is the maximum size allowed. In kernel version 2.4, the maximum is 4MB.

It is possible to call this function to allocate multiple buffers, even if a single call for a large buffer may fail. For example, a call to allocate a 4MB buffer may fail, but two calls to allocate two 2MB buffers may succeed. It must

be noted, however, that these buffers together do not make up a contiguous 4MB block in memory; they are separate.

The purpose of these buffers is typically for use with PLX DMA engines or local masters. Since the buffers are page-locked and physically contiguous in memory, the DMA engine can access the memory as one continuous block. When using a buffer for DMA transfers, the physical address should be used when specifying the PCI address of a block DMA transfer.

The allocated buffer is not mapped into user virtual space when allocated. To map the buffer into virtual space, use *PlxPciPhysicalMemoryMap*.

### Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
PLX_PHYSICAL_MEM Buffer_1;
PLX_PHYSICAL_MEM Buffer_2;

// Allocate a buffer that must succeed

// Set desired size
Buffer_1.Size = 0x300000;    // 3MB

rc = PlxPciPhysicalMemoryAllocate(
    hDevice,
    &Buffer_1,
    FALSE           // Do not allocate a smaller buffer on failure
);

if (rc != ApiSuccess)
{
    // Error - unable to allocate physical buffer
}

// Allocate a buffer, accepting any size

// Set desired size
RequestSize = 0x1000000;    // 16MB
Buffer_2.Size = RequestSize;

rc = PlxPciPhysicalMemoryAllocate(
    hDevice,
    &Buffer_2,
    TRUE           // A smaller size buffer is acceptable
);

if (rc != ApiSuccess)
{
    // Error - unable to allocate physical buffer
}

if (Buffer_2.Size != RequestSize)
{
    // Buffer allocated, but smaller than requested size
}
```

**Cross Reference:**

Referenced Item	Page
PLX_PHYSICAL_MEM	3-29

## PlxPciPhysicalMemoryFree

---

### Syntax:

```
RETURN_CODE  
PlxPciPhysicalMemoryFree(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function will release a buffer previously allocated with *PlxPciPhysicalMemoryAllocate*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pMemoryInfo*

A pointer to a PLX\_PHYSICAL\_MEM structure which contains the buffer information.

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidData	The buffer information is invalid or it was not allocated with <i>PlxPciPhysicalMemoryAllocate</i>

### Notes:

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

If the buffer was previously mapped to user virtual space, with *PlxPciPhysicalMemoryMap*, it should be unmapped, with *PlxPciPhysicalMemoryUnmap*, before freeing it from memory.

Once this buffer is released, any virtual mappings to it will fail and the buffer should no longer be used by hardware, such as the DMA engine. The memory will be returned to the operating system.

All allocated buffers should be unmapped and freed before releasing a device with a call to *PlxPciDeviceClose*. Buffers will become invalid once a device is released.

## Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
PLX_PHYSICAL_MEM Buffer;

// Allocate a buffer

// Set desired size
Buffer.Size = 0x1000;

rc =
    PlxPciPhysicalMemoryAllocate(
        hDevice,
        &Buffer,
        FALSE          // Do not allocate a smaller buffer on failure
    );

if (rc != ApiSuccess)
{
    // Error - unable to allocate physical buffer
}

//
// Use the buffer as needed
//

// Release the buffer
rc =
    PlxPciPhysicalMemoryFree(
        hDevice,
        &Buffer
    );

if (rc != ApiSuccess)
{
    // Error - unable to free physical buffer
}
```

## Cross Reference:

Referenced Item	Page
PLX_PHYSICAL_MEM	3-29

## PlxPciPhysicalMemoryMap

---

### Syntax:

```
RETURN_CODE  
PlxPciPhysicalMemoryMap(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function will map into user virtual space, a buffer previously allocated with *PlxPciPhysicalMemoryAllocate*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pMemoryInfo*

A pointer to a PLX\_PHYSICAL\_MEM structure which contains the buffer information.

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidData	Buffer information is invalid or buffer not allocated properly
ApiInvalidAddress	Physical address of buffer is invalid or buffer not allocated properly
ApiInsufficientResources	Insufficient resources to perform the mapping

### Notes:

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

Mapping of physical memory into user virtual space may fail due to insufficient Page-Table Entries (PTEs). The larger the buffer size, the greater the number of PTEs required to map it into user space.

The buffer should be unmapped before calling *PlxPciDeviceClose* to close the device. The virtual address will cease to be valid after closing the device or after unmapping the buffer. Refer to *PlxPciPhysicalMemoryUnmap*.

## Usage:

```
U8          Value;
HANDLE      hDevice;
RETURN_CODE rc;
PLX_PHYSICAL_MEM Buffer;

// Allocate a buffer

// Set desired size
Buffer.Size = 0x1000;

rc =
    PlxPciPhysicalMemoryAllocate(
        hDevice,
        &Buffer,
        FALSE          // Do not allocate a smaller buffer on failure
    );

if (rc != ApiSuccess)
{
    // Error - unable to allocate physical buffer
}

rc =
    PlxPciPhysicalMemoryMap(
        hDevice,
        &Buffer
    );

if (rc != ApiSuccess)
{
    // Error - unable to map physical buffer
}

// Write 32-bit value to buffer
*(U32*)(Buffer.UserAddr + 0x100) = 0x12345;

// Read 8-bit value from buffer
value = *(U8*)(Buffer.UserAddr + 0x54);
```

## Cross Reference:

Referenced Item	Page
PLX_PHYSICAL_MEM	3-29

## PlxPciPhysicalMemoryUnmap

---

### Syntax:

```
RETURN_CODE  
PlxPciPhysicalMemoryUnmap(  
    HANDLE          hDevice,  
    PLX_PHYSICAL_MEM *pMemoryInfo  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

This function will unmap a physical buffer previously mapped with *PlxPciPhysicalMemoryMap*.

### Parameters:

*hDevice*

Handle of an open PCI device

*pMemoryInfo*

A pointer to a PLX\_PHYSICAL\_MEM structure which contains the buffer information

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event occurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	The virtual address is invalid or was not previously mapped with <i>PlxPciPhysicalMemoryMap</i>
ApiInvalidData	The buffer information is invalid or it was not allocated with <i>PlxPciPhysicalMemoryAllocate</i>

### Notes:

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

It is important to unmap a physical buffer when it is no longer needed to release mapping resources back to the system.

The buffer should be un-mapped before calling *PlxPciDeviceClose* to close the device. The virtual address will cease to be valid after closing the device or after un-mapping the buffer.

## Usage:

```
HANDLE          hDevice;
RETURN_CODE     rc;
PLX_PHYSICAL_MEM Buffer;

// Allocate a buffer (not shown)

// Map buffer into user space to get virtual address
rc =
    PlxPciPhysicalMemoryMap(
        hDevice,
        &Buffer
    );

if (rc != ApiSuccess)
{
    // Error - unable to map physical buffer
}

//
// Access buffer as needed
//

// Unmap the buffer from virtual space
rc =
    PlxPciPhysicalMemoryUnmap(
        hDevice,
        &Buffer
    );

if (rc != ApiSuccess)
{
    // Error - unable to unmap physical buffer
}
```

## Cross Reference:

Referenced Item	Page
PLX_PHYSICAL_MEM	3-29

## PlxPciRegisterReadFast

---

### Syntax:

```
U32
PlxPciRegisterReadFast(
    HANDLE          hDevice,
    U16             offset,
    RETURN_CODE     *pReturnCode
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Reads the value of a PCI configuration register on the selected device.

### Parameters:

*hDevice*

Handle of an open PCI device

*offset*

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

### Usage:

```
HANDLE          hDevice;
U32             RegValue;
RETURN_CODE     rc;

// Read the Subsystem Device/Vendor ID
RegValue =
    PlxPciRegisterReadFast(
        hDevice,
        CFG_SUB_VENDOR_ID,
        &rc
    );
```

## PlxPciRegisterWriteFast

---

### Syntax:

```
RETURN_CODE  
PlxPciConfigRegisterWriteFast(  
    HANDLE hDevice,  
    U16    offset,  
    U32    value  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Writes to a PCI configuration register on the selected device.

### Parameters:

*hDevice*

Handle of an open PCI device

*offset*

Offset of the PCI configuration register write

*value*

The 32-bit value 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
ApiConfigAccessFailed	The specified device either does not exist or the PCI configuration access failed

## Usage:

```
U32          RegValue;
HANDLE       hDevice;
RETURN_CODE  rc;

// Read the PCI Command/Status register
RegValue =
    PlxPciRegisterReadFast(
        hDevice,
        CFG_COMMAND,          // PCI Command/Status register
        &rc
    );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read PCI configuration register
}

// Check for any PCI Errors or Aborts
if (RegValue & 0xf8000000)
{
    // Write PCI Status back to itself to clear any errors
    rc =
        PlxPciRegisterWriteFast(
            hDevice,
            CFG_COMMAND,
            &RegValue
        );

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

## PlxPciRegisterRead\_Unsupported

---

### Syntax:

```
U32
PlxPciRegisterRead_Unsupported(
    U8          bus,
    U8          slot,
    U16         offset,
    RETURN_CODE *pReturnCode
);
```

### PLX Chip Support:

All

### Description:

Returns the value of a PCI configuration register of a specified PCI device.

### Parameters:

*bus*

The PCI bus number of the device

*slot*

The PCI slot number of the device

*offset*

Offset of the PCI configuration register read (32-bit aligned)

*pReturnCode*

A pointer to a buffer for the return code

### Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiConfigAccessFailed	The specified device either does not exist or the PCI configuration access failed
ApiInvalidRegister	The register offset parameter is out of range or not aligned on a 4-byte boundary

### Notes:

With the support for Plug 'n' Play in newer versions of the Windows OS, access to PCI configuration registers of an arbitrary device is not directly allowed. PLX drivers provide functions to access PCI registers of arbitrary devices by bypassing the OS services and using the standard PCI I/O ports.

Due to the nature of the implementation of this function, PLX cannot guarantee its functionality in future SDK releases. For example, future versions of the OS may not allow PCI I/O port accesses. As a result, PLX does not support this function. It is provided for customers who absolutely need this functionality.

Although this function may return *ApiSuccess* in the return code, this does not necessarily indicate a successful access to the device since the driver gets no indication of success or failure. If the register value returned is FFFF\_FFFFh, it is usually an indication of an error or non-existent device in the specified bus/slot.

## Usage:

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

// Scan PCI bus for devices
for (bus=0; bus < 32; bus++)
{
    for (slot=0; slot < 32; slot++)
    {
        DevVenId =
            PlxPciRegisterRead_Unsupported(
                bus,
                slot,
                0x0,          // PCI Device/Vendor ID offset
                &rc
            );

        // Check if read was successful
        if ((rc == ApiSuccess) && (DevVenId != (U32)-1))
        {
            // Found a device at specified bus/slot
        }
        else
        {
            // No device at specified bus/slot
        }
    }
}
```

## PlxPciRegisterWrite\_Unsupported

---

### Syntax:

```
RETURN_CODE  
PlxPciRegisterWrite_Unsupported(  
    U8 bus,  
    U8 slot,  
    U16 offset,  
    U32 value  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Writes a value to a PCI configuration register of a specified PCI device.

### Parameters:

*bus*

The PCI bus number of the device

*slot*

The PCI slot number of the device

*offset*

Offset of the PCI configuration register read (32-bit aligned)

*value*

The 32-bit data value 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

### Notes:

With the support for Plug 'n' Play in newer versions of the Windows OS, access to PCI configuration registers of an arbitrary device is not directly allowed. PLX drivers provide functions to access PCI registers of arbitrary devices by bypassing the OS services and using the standard PCI I/O ports.

Due to the nature of the implementation of this function, PLX cannot guarantee its functionality in future SDK releases. For example, future versions of the OS may not allow PCI I/O port accesses. As a result, PLX does not support this function. It is provided for customers who absolutely need this functionality.

Although this function may return *ApiSuccess* in the return code, this does not necessarily indicate a successful access to the device since the driver gets no indication of success or failure. In order to verify the data was written properly, perform a PCI register read and compare values.

Use of this function is NOT recommended. Direct modification of PCI registers may result in system instability or device failure. This function is provided only for completeness and for reference purposes.

**Usage:**

```
RETURN_CODE rc;

// Perform an EEPROM write using VPD access

// Write to VPD data register
PlxPciRegisterWrite_Unsupported(
    0x01,          // Known bus
    0x07,          // Known slot
    0x50,          // PCI VPD data register
    0xFF008670    // Data to write
);

// Issue VPD command to write to EEPROM offset 34h
PlxPciRegisterWrite_Unsupported(
    0x01,          // Known bus
    0x07,          // Known slot
    0x4C,          // PCI VPD command register
    0x80340003    // VPD write command
);
```

## PlxPmNcpRead

---

### Syntax:

```
U8
PlxPmNcpRead(
    HANDLE      hDevice,
    RETURN_CODE *pReturnCode
);
```

### PLX Chip Support:

9030, 9054, 9056, 9656, 8311

### 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:

All 9000 series & 8311

\*The 9050, 9052, & 9080 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

## 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	3-38

## PlxPowerLevelSet

---

### Syntax:

```
RETURN_CODE  
PlxPowerLevelSet(  
    HANDLE          hDevice,  
    PLX_POWER_LEVEL plxPowerLevel  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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	3-38

## PlxRegisterDoorbellRead

---

### Syntax:

```
U32  
PlxRegisterDoorbellRead(  
    HANDLE          hDevice,  
    RETURN_CODE    *pReturnCode  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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    value
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*value*  
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:

9054, 9056, 9080, 9656, 8311

### 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	3-24

## PlxRegisterMailboxWrite

---

### Syntax:

```
RETURN_CODE  
PlxRegisterMailboxWrite(  
    HANDLE      hDevice,  
    MAILBOX_ID  MailboxId,  
    U32         value  
);
```

### PLX Chip Support:

9054, 9056, 9080, 9656, 8311

### 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

*value*

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	3-24

# PlxRegisterRead

---

## Syntax:

```
U32  
PlxRegisterRead(  
    HANDLE          hDevice,  
    U16             offset,  
    RETURN_CODE     *pReturnCode  
);
```

## PLX Chip Support:

All 9000 series & 8311

## 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

*offset*

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
}
```

# PlxRegisterWrite

---

## Syntax:

```
RETURN_CODE  
PlxRegisterWrite(  
    HANDLE hDevice,  
    U16    offset,  
    U32    value  
);
```

## PLX Chip Support:

All 9000 series & 8311

## Description:

Writes a value to a specified register of a PLX PCI device.

## Parameters:

- hDevice*  
Handle of an open PCI device
- offset*  
The register offset
- value*  
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;
```

```
PlxSdkVersion(  
    &SdkMajor,  
    &SdkMinor,  
    &SdkRevision  
);
```

```
PlxPrintf(  
    "SDK API Version = %d.%d%d\n",  
    SdkMajor, SdkMinor, SdkRevision  
);
```

## PlxSerialEepromPresent

---

### Syntax:

```
BOOLEAN  
PlxSerialEepromPresent(  
    HANDLE          hDevice,  
    RETURN_CODE    *pReturnCode  
);
```

### PLX Chip Support:

All 9000 series & 8311

### 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()*.

This function simply reports the status of the "EEPROM Present Bit" directly from the PLX chip. The 9050, 9052, and 9080 PLX chips do not distinguish between a blank or non-existent EEPROM. As a result, this function may return a value of FALSE even if an EEPROM is physically present, but is blank. The 9030, 9054, 9056, and 9656 chips will report that an EEPROM is present even if it is blank.

## 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  
);
```

### NOTE

*This function has been deprecated and replaced with the **PlxSerialEepromReadByOffset** API function. Please refer to this function for EEPROM access*

## PlxSerialEepromReadByOffset

---

### Syntax:

```
RETURN_CODE
PlxSerialEepromReadByOffset(
    HANDLE  hDevice,
    U16     offset,
    U32     *pValue
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Reads a 32-bit value from a specified offset from the configuration EEPROM connected to the PLX chip

### Parameters:

*hDevice*

Handle of an open PCI device

*offset*

The EEPROM offset of the value to read. (Must be 4-byte aligned)

*pValue*

A pointer to a 32-bit buffer which will contain the data read.

### 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
ApiInvalidOffset	The offset parameter is too large or not aligned on a 4-byte boundary
ApiVpdNotEnabled	The VPD feature in the PLX chip is disabled.

### 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.

## Usage:

```
U32          EepromData;
HANDLE       hDevice;
RETURN_CODE  rc;

// Read the Subsystem Device/Vendor ID of the 9054
rc =
    PlxSerialEepromReadByOffset(
        hDevice,
        0x44,          // Subsystem ID EEPROM offset
        &EepromData
    );

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

## PlxSerialEepromWrite

---

### Syntax:

```
RETURN_CODE  
PlxSerialEepromWrite(  
    HANDLE          hDevice,  
    EEPROM_TYPE    EepromType,  
    U32             *buffer,  
    U32             size  
);
```

### NOTE

*This function has been deprecated and replaced with the **PlxSerialEepromWriteByOffset** API function. Please refer to this function for EEPROM access*

## PlxSerialEepromWriteByOffset

---

### Syntax:

```
RETURN_CODE  
PlxSerialEepromWriteByOffset(  
    HANDLE hDevice,  
    U16    offset,  
    U32    value  
);
```

### PLX Chip Support:

All 9000 series & 8311

### Description:

Writes a 32-bit value to a specified offset from the configuration EEPROM connected to the PLX chip.

### Parameters:

*hDevice*

Handle of an open PCI device

*offset*

The EEPROM offset of the value to write. (Must be 4-byte aligned)

*value*

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
ApiInvalidOffset	The offset parameter is too large or not aligned on a 4-byte boundary
ApiVpdNotEnabled	The VPD feature in the PLX chip is disabled.

### 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.

**Usage:**

```
HANDLE      hDevice;
RETURN_CODE rc;

// Write EEPROM data
rc =
    PlxSerialEepromWriteByOffset(
        hDevice,
        0x14,           // Space 0 Range EEPROM offset
        0xFF000000     // 16MB range
    );

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

## PlxVpdNcpRead

---

### Syntax:

```
U8
PlxVpdNcpRead(
    HANDLE      hDevice,
    RETURN_CODE *pReturnCode
);
```

### PLX Chip Support:

9030, 9054, 9056, 9656, 8311

### 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,
    U16             offset,
    RETURN_CODE     *pReturnCode
);
```

### PLX Chip Support:

9030, 9054, 9056, 9656, 8311

### 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,  
    U16    offset,  
    U32    value  
);
```

### PLX Chip Support:

9030, 9054, 9056, 9656, 8311

### 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)

*value*

The 32-bit value 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
}
```

## Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiInvalidOffset	The offset is invalid, out of range, or not 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
}
```



## 3 PLX SDK Data Structures Used by the API

### 3.1 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;
```

#### **Purpose**

The reasons for using this structure.

#### **Members**

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

#### **Affected Register Location**

Structure Element	9080	9054	9030	9056
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.

## Basic Data Types

---

### Purpose

These data types are used for code portability between all supported environments. PLX header files automatically define the definitions depending upon the build environment.

Data Type	Storage Allocation
S8	Signed 8-bit
U8	Unsigned 8-bit
S16	Signed 16-bit
U16	Unsigned 16-bit
S32	Signed 32-bit
U32	Unsigned 32-bit
S64	Signed 64-bit
U64	Unsigned 64-bit

## **BOOLEAN**

---

```
#if !defined(BOOLEAN)
    typedef S8                BOOLEAN;
#endif
```

```
#if !defined(BOOL)
    typedef S8                BOOL;
#endif
```

### **Purpose**

This data type defines the BOOLEAN and BOOL data types.

## ACCESS\_TYPE

---

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

### 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\_PARMS

---

```
typedef struct _API_PARMS
{
    U32                PlxChipBaseAddr;
    PCI_BUS_PROP       *pPciBusProp;
    IOP_BUS_PROP       *pIopBus0Prop;
    IOP_BUS_PROP       *pIopBus1Prop;
    IOP_BUS_PROP       *pIopBus2Prop;
    IOP_BUS_PROP       *pIopBus3Prop;
    IOP_BUS_PROP       *pLcs0Prop;
    IOP_BUS_PROP       *pLcs1Prop;
    IOP_BUS_PROP       *pLcs2Prop;
    IOP_BUS_PROP       *pLcs3Prop;
    IOP_BUS_PROP       *pDramProp;
    IOP_BUS_PROP       *pDefaultProp;
    IOP_BUS_PROP       *pExpRomBusProp;
    IOP_ARBIT_DESC     *pIopArbitDesc;
    IOP_ENDIAN_DESC    *pIopEndianDesc;
    PM_PROP            *pPMProp;
    U32                *pVPDBaseAddress;
} API_PARMS;
```

### 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

#### *PlxChipBaseAddr*

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

#### *PtrPciBusProp*

A pointer to the `PCI_BUS_PROP` structure that will be used to initialize the PCI Bus properties 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*

Unused

#### *PtrLcs1Prop*

Unused

*PtrLcs2Prop*  
Unused

*PtrLcs3Prop*  
Unused

*PtrDramProp*  
Unused

*PtrDefaultProp*  
Unused

*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

---

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

### **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

---

```
typedef struct _DEVICE_LOCATION
{
    U8  BusNumber;
    U8  SlotNumber;
    U16 DeviceId;
    U16 VendorId;
    U8  SerialNumber[20];
} DEVICE_LOCATION;
```

### Purpose

This data type provides information about a PCI device. 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\_DESC

---

```
typedef struct _DMA_CHANNEL_DESC
{
    unsigned int EnableReadyInput      :1;
    unsigned int EnableBTERMInput      :1;
    unsigned int EnableIopBurst        :1;
    unsigned int EnableWriteInvalidMode :1;
    unsigned int EnableDmaEOTPin       :1;
    unsigned int DmaStopTransferMode   :1;
    unsigned int HoldIopAddrConst      :1;
    unsigned int DemandMode            :1;
    unsigned int EnableTransferCountClear :1;
    unsigned int WaitStates            :4;
    unsigned int IopBusWidth           :2;
    unsigned int EOTEndLink            :1;
    unsigned int ValidStopControl      :1;
    unsigned int ValidModeEnable       :1;
    unsigned int EnableDualAddressCycles :1;
    unsigned int TholdForIopWrites     :4;
    unsigned int TholdForIopReads      :4;
    unsigned int TholdForPciWrites     :4;
    unsigned int TholdForPciReads      :4;
    DMA_CHANNEL_PRIORITY DmaChannelPriority;
} DMA_CHANNEL_DESC;
```

### 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 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*

This value determined the DMA termination mode. The first option, *AssertBLAST*, sends a BLAST to terminate the DMA transfer (for CBUS and JBUS) or negate BDIP at the nearest 16-byte boundary (for MBUS). The other option, *EOTAsserted*, requires EOT be asserted or DREQ# negated to indicate a DMA termination.

#### *HoldIopAddrConst*

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

#### *DemandMode*

Sets the DMA controller to use demand mode if this value is set.

#### *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 local bus for the DMA channel is defined by this value.

#### *EOTEndLink*

Used only for DMA Scatter/Gather transfers. When EOT# is asserted, value of 1 indicates the DMA transfer completes the current Scatter/Gather transfer and continues with the remaining Scatter/Gather transfers. When EOT# 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 *DMADescriptorValid* 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.

#### *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.

#### *DmaChannelPriority*

The DMA channel priority scheme is set with this value.

### **Affected Register Location**

Structure Element	9080	9054	9056/9656
EnableReadyInput	0x100, 6 0x114, 6	0x100, 6 0x114, 6	0x100, 6 0x114, 6
EnableBTERMInput	0x100, 7 0x114, 7	0x100, 7 0x114, 7	0x100, 7 0x114, 7
EnableIopBurst	0x100, 8 0x114, 8	0x100, 8 0x114, 8	0x100, 8 0x114, 8

Structure Element	9080	9054	9056/9656
EnableWriteInvalidMode	0x04, 4 0x100, 13 0x114, 13	0x04, 4 0x100, 13 0x114, 13	0x04, 4 0x100, 13 0x114, 13
EnableDmaEOTPin	0x100, 14 0x114, 14	0x100, 14 0x114, 14	0x100, 14 0x114, 14
DmaStopTransferMode	0x100, 15 0x114, 15	0x100, 15 0x114, 15	0x100, 15 0x114, 15
HoldIopAddrConst	0x100, 11 0x114, 11	0x100, 11 0x114, 11	0x100, 11 0x114, 11
DemandMode	0x100, 12 0x114, 12	0x100, 12	0x100, 12 0x114, 12
EnableTransferCountClear <i>*Note: Transfer count can only be cleared if SGL is located in the Local memory and not in PCI memory.</i>	0x100, 16* 0x114, 16*	0x100, 16 * 0x114, 16 *	0x100, 16 * 0x114, 16 *
WaitStates	0x100, 2-5 0x114, 2-5	0x100, 2-5 0x114, 2-5	0x100, 2-5 0x114, 2-5
IopBusWidth	0x100, 0-1 0x114, 0-1	0x100, 0-1 0x114, 0-1	0x100, 0-1 0x114, 0-1
EOTEndLink	N/A	N/A	0x100, 19 0x114, 19
ValidStopControl	N/A	N/A	0x100, 21 0x114, 21
ValidModeEnable	N/A	N/A	0x100, 20 0x114, 20
EnableDualAddressCycles	N/A	0x100, 18 0x114, 18	0x100, 18 0x114, 18
TholdForIopWrites	0x130, 0-3 0x130, 16-19	0x130, 0-3 0x130, 16-19	0x130, 0-3 0x130, 16-19
TholdForIopReads	0x130, 4-7 0x130, 20-23	0x130, 4-7 0x130, 20-23	0x130, 4-7 0x130, 20-23
TholdForPciWrites	0x130, 8-11 0x130, 24-27	0x130, 8-11 0x130, 24-27	0x130, 8-11 0x130, 24-27
TholdForPciReads	0x130, 12-15 0x130, 28-31	0x130, 12-15 0x130, 28-31	0x130, 12-15 0x130, 28-31
DmaChannelPriority	0x88, 19-20	0x88, 19-20	0x88, 19-20
EnableDoneInt <i>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.</i>	0x100, 10 0x114, 10	0x100, 10 0x114, 10	0x100, 10 0x114, 10

## DMA\_CHANNEL

---

```
typedef enum _DMA_CHANNEL
{
    IopChannel0,
    IopChannel1,
    IopChannel2,
    PrimaryPciChannel0,
    PrimaryPciChannel1,
    PrimaryPciChannel2,
    PrimaryPciChannel3
} DMA_CHANNEL;
```

### Purpose

Enumerated type used for requesting a DMA channel.

### Members

#### *IopChannel0*

Request the Local-to-Local DMA channel 0. This is no longer supported.

#### *IopChannel1*

Request the Local-to-Local DMA channel 1. This is no longer supported.

#### *IopChannel2*

Request the Local-to-Local DMA channel 2. This is no longer supported.

#### *PrimaryPciChannel0*

Request the Primary PCI-to-Local DMA channel 0.

#### *PrimaryPciChannel1*

Request the Primary PCI-to-Local DMA channel 1.

#### *PrimaryPciChannel2*

Request the Primary PCI-to-Local DMA channel 2.

#### *PrimaryPciChannel3*

Request the Primary PCI-to-Local DMA channel 3.

## DMA\_CHANNEL\_PRIORITY

---

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

### **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

---

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

### 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.

## DMA\_TRANSFER\_ELEMENT

---

```
typedef struct _DMA_TRANSFER_ELEMENT
{
    union
    {
        U64 UserVa; // User space virtual address
        U32 PciAddrLow; // Lower 32-bits of PCI address
    } u;
    U32 PciAddrHigh; // Upper 32-bits of PCI address
    U32 LocalAddr; // Local bus address
    U32 TransferCount; // Number of bytes to transfer
    U32 TerminalCountIntr :1;
    U32 LocalToPciDma :1;
} DMA_TRANSFER_ELEMENT;
```

### Purpose

Structure used to program the DMA registers.

### Members

#### *UserVa (Host-code only)*

This member should be used by host applications for SGL only. It represents the user-mode address (virtual) of the PCI buffer for the DMA transfer.

#### *LowPciAddr / PciAddrLow*

The lower 32-bits of the PCI buffer address for the DMA transfer. This value is used to program the PCI Address Register for a given DMA channel.

#### *HighPciAddr / PciAddrHigh*

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.

#### *IopAddr / LocalAddr*

The IOP buffer address for the DMA transfer. This value is used to program the Local Address Register for a given DMA channel.

#### *TransferCount*

The number of bytes to be transferred. This value is used to program the Transfer Count Register for a given DMA channel.

#### *PciSglLoc (Local-code only)*

The next SGL element is located in PCI memory if this value is set. Otherwise, the next SGL element is located in IOP memory.

#### *LastSglElement (Local-code only)*

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.

#### *TerminalCountIntr*

A DMA interrupt will be generated after completing this SGL element's DMA transfer if this value is set.

#### *IopToPciDma / LocalToPciDma*

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.

#### *NextSglPtr (Local-code only)*

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

---

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

### 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	9030	9054/9056/9656/8311
HS_LED_ON	0x4A, 3	0x18A, 3
HS_BOARD_REMOVED	0x4A, 6	0x18A, 6
HS_BOARD_INSERTED	0x4A, 7	0x18A, 7

### 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\_ARBIT\_DESC

---

```
typedef struct _IOP_ARBIT_DESC
{
    U32 IopBusDSGiveUpBusMode      :1;
    U32 EnableDSLatchedSequence    :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;
```

### 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.

#### *EnableDSLatchedSequence*

The Direct Slave latched 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 BOFF# pin can be asserted when this value is set.

#### *BOFFTimerResolution*

When this value is set, the LSB of the 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#.

### *lopBusLatencyTimer*

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.

### *lopBusPauseTimer*

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

### *EnablelopBusTimeout*

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

### *lopBusTimeout*

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

## **Affected Register Location**

<b>Structure Element</b>	<b>9080</b>	<b>9054</b>	<b>9x56</b>
lopBusDSGiveUpBusMode	0x88, 21	0x88, 21	0x88, 21
EnableDSLatchedSequence	0x88, 22	0x88, 22	0x88, 22
GatelopLatencyTimerBREQo	0x88, 27	0x88, 27	0x88, 27
EnableWAITInput	N/A	0x88, 31	0x88, 31
EnableBOFF	N/A	0x94, 4	0x94, 4
BOFFTimerResolution	N/A	0x94, 5	0x94, 5
EnablelopBusLatencyTimer	0x88, 16	0x88, 16	0x88, 16
EnablelopBusPauseTimer	0x88, 17	0x88, 17	0x88, 17
EnablelopArbiter	N/A	N/A	N/A
lopArbitrationPriority	N/A	N/A	N/A
BOFFDelayClocks	N/A	N/A	0x94, 0-3
lopBusLatencyTimer	0x88, 0-7	0x88, 0-7	0x88, 0-7
lopBusPauseTimer	0x88, 8-15	0x88, 8-15	0x88, 8-15
EnablelopBusTimeout	N/A	N/A	N/A
lopBusTimeout	N/A	N/A	N/A

## IOP\_BUS\_PROP

---

```
typedef struct _IOP_BUS_PROP
{
    unsigned int EnableReadyRecover      :1;
    unsigned int EnableReadyInput        :1;
    unsigned int EnableBTERMInput        :1;
    unsigned int DisableReadPrefetch     :1;
    unsigned int EnableReadPrefetchCount :1;
    unsigned int ReadPrefetchCounter     :4;
    unsigned int EnableBursting          :1;
    unsigned int EnableIopBusTimeoutTimer:1;
    unsigned int BREQoTimerResolution    :1;
    unsigned int EnableIopBREQo          :1;
    unsigned int BREQoDelayClockCount    :4;
    unsigned int MapInMemorySpace        :1;
    unsigned int InternalWaitStates      :4;
    unsigned int PciRev2_1Mode           :1;
    unsigned int IopBusWidth             :2;
} IOP_BUS_PROP;
```

### Purpose

Structure used to describe the local bus characteristics.

### Members

#### *EnableReadyRecover*

Not Supported

#### *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*

Not Supported

#### *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.

#### *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.

### **Affected Register Location**

<b>Structure Element</b>	<b>Register (9080, 9054, 9056, 9656)</b>
EnableReadyRecover	N/A
EnableReadyInput	0x98, 6
	0x98, 22
	0x178, 6
EnableBTERMInput	0x98, 7
	0x98, 23
	0x178, 7
DisableReadPrefetch	0x98, 8
	0x98, 9
	0x178, 9
EnableReadPrefetchCount	0x98, 10
	0x178, 10
ReadPrefetchCounter	0x98, 11-14
	0x178, 11-14
EnableBursting	0x98, 24
	0x98, 26
	0x178, 8
EnableIopBusTimeoutTimer	N/A
BREQoTimerResolution	0x94, 5
EnableIopBREQo	0x94, 4
BREQoDelayClockCount	0x94, 0-3
MapInMemorySpace	0x80, 0
	0x170, 0
PciRev2_1Mode	0x88, 24
IopBusWidth	0x98, 0-1
	0x98, 16-17
	0x178, 0-1
InternalWaitStates	0x98, 2-5
	0x98, 18-21
	0x178, 2-5

## IOP\_ENDIAN\_DESC

---

```
typedef struct _IOP_ENDIAN_DESC
{
    unsigned int BigEIopSpace0           :1;
    unsigned int BigEIopSpace1           :1;
    unsigned int BigEExpansionRom         :1;
    unsigned int BigEDmaChannel0          :1;
    unsigned int BigEDmaChannel1         :1;
    unsigned int BigEIopConfigRegAccess   :1;
    unsigned int BigEDirectMasterAccess   :1;
    unsigned int BigEByteLaneMode         :1;
} IOP_ENDIAN_DESC;
```

### 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.

#### *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.

### Affected Register Location

Structure Element	Register (9080, 9054, 9056, 9656)
BigEIopSpace0	0x8C, 2
BigEIopSpace1	0x8C, 5
BigEExpansionRom	0x8C, 3
BigEIopConfigRegAccess	0x8C, 0
BigEDirectMasterAccess	0x8C, 1
BigEDmaChannel0	0x8C, 7
BigEDmaChannel1	0x8C, 6
BigEByteLaneMode	0x8C, 4

## IOP\_SPACE

---

```
typedef enum _IOP_SPACE
{
    IopSpace0,
    IopSpace1,
    IopSpace2,
    IopSpace3
} IOP_SPACE;
```

### **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.

## MAILBOX\_ID

---

```
typedef enum _MAILBOX_ID
{
    MailBox0,
    MailBox1,
    MailBox2,
    MailBox3,
    MailBox4,
    MailBox5,
    MailBox6,
    MailBox7
} MAILBOX_ID;
```

### Purpose

Enumerated type used for choosing the desired mailbox register.

### Members

*MailBoxX*      Use mailbox X register.

### Affected Register Location

Structure Element	9080	9054	9x56
MailBox0	0xC0	0xC0	0xC0
MailBox1	0xC4	0xC4	0xC4
MailBox2	0xC8	0xC8	0Cx8
MailBox3	0xCC	0xCC	0xCC
MailBox4	0xD0	0xD0	0xD0
MailBox5	0xD4	0xD4	0xD4
MailBox6	0xD8	0xD8	0xD8
MailBox7	0xDC	0xDC	0xDC

## New Capabilities Flags

---

```
#define CAPABILITY_POWER_MANAGEMENT    (1 << 0)
#define CAPABILITY_HOT_SWAP            (1 << 1)
#define CAPABILITY_VPD                  (1 << 2)
```

### Purpose

Specifies which New Capabilities to check. The definitions can be combined using the OR operator

## PCI\_BUS\_PROP

---

```
typedef struct _PCI_BUS_PROP
{
    unsigned int  PciRequestMode           :1;
    unsigned int  DmPciReadMode           :1;
    unsigned int  EnablePciArbiter        :1;
    unsigned int  EnableWriteInvalidMode  :1;
    unsigned int  DmPrefetchLimit         :1;
    unsigned int  PciReadNoWriteMode      :1;
    unsigned int  PciReadWriteFlushMode   :1;
    unsigned int  PciReadNoFlushMode      :1;
    unsigned int  EnableRetryAbort        :1;
    unsigned int  WfifoAlmostFullFlagCount :5;
    unsigned int  DmWriteDelay            :2;
    unsigned int  ReadPrefetchMode        :2;
    unsigned int  IoRemapSelect           :1;
    unsigned int  EnablePciBusMastering   :1;
    unsigned int  EnableMemorySpaceAccess :1;
    unsigned int  EnableIoSpaceAccess     :1;
} PCI_BUS_PROP;
```

### 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*

Not Supported

#### *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*

Not Supported

#### *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.

### **Affected Register Location**

<b>Structure Element</b>	<b>9080</b>	<b>9054</b>	<b>9x56</b>
PciRequestMode	0x88, 23	0x88, 23	0x88, 26
DmPciReadMode	0xA8, 4	0xA8, 4	0xA8, 4
EnablePciArbiter	N/A	N/A	N/A
EnableWriteInvalidMode	0x04, 4 0xA8, 9	0x04, 4 0xA8, 9	0x04, 4 0xA8, 9
DmPrefetchLimit	0xA8, 11	0xA8, 11	0xA8, 11
PciAddressSpaceBusWidth	N/A	N/A	N/A
PciReadNoWriteMode	0x88, 25	0x88, 25	0x88, 25
PciReadWriteFlushMode	0x88, 26	0x88, 26	0x88, 26
PciReadNoFlushMode	0x88, 28	0x88, 28	0x88, 28
EnableRetryAbort	N/A	N/A	N/A
WfifoAlmostFullFlagCount	0xA8, 5-8 0xA8, 10	0xA8, 5-8 0xA8, 10	0xA8, 5-8 0xA8, 10
DmWriteDelay	0xA8, 14-15	0xA8, 14-15	0xA8, 14-15
ReadPrefetchMode	0xA8, 3 0xA8, 12	0xA8, 3 0xA8, 12	0xA8, 3 0xA8, 12
IoRemapSelect	0xA8, 13	0xA8, 13	0xA8, 13

Structure Element	9080	9054	9x56
EnablePciBusMastering	N/A	0x04, 2	0x04, 2
EnableMemorySpaceAccess	N/A	N/A	0x04, 1
EnableIoSpaceAccess	N/A	N/A	0x04, 0

## PLX\_PHYSICAL\_MEM

---

```
typedef struct _PLX_PHYSICAL_MEM
{
    U64 UserAddr;
    U64 PhysicalAddr;
    U64 CpuPhysical;
    U32 Size;
} PLX_PHYSICAL_MEM;
```

### Purpose

This data type provides information for a contiguous page-locked buffer allocated by the device driver. This is typically used as a buffer for DMA transfers.

### Members

#### *UserAddr*

User Virtual Address for the buffer

#### *PhysicalAddr*

The Bus or Logical Physical address of the buffer. This address may be used to program the DMA engine.

#### *CpuPhysical*

The CPU Physical address of the buffer. This value is used internally by the PLX driver for mappings to user space.

#### *Size*

The size of the buffer.

## PCI\_SPACE

---

```
typedef enum _PCI_SPACE
{
    PciMemSpace,
    PciIoSpace
} PCI_SPACE;
```

### Purpose

Enumerated type used for choosing the desired PCI Address Space access.

### Members

#### *PciMemSpace*

Use PCI memory cycles when accessing the PCI bus.

#### *PciloSpace*

Use PCI I/O cycles when accessing the PCI bus.

## PLX\_DEVICE\_KEY

---

```
typedef struct _PLX_DEVICE_KEY
{
    U32 IsValidTag;           // Magic number to determine validity
    U8  bus;                 // Physical device location
    U8  slot;
    U8  function;
    U16 VendorId;           // Device Identifier
    U16 DeviceId;
    U16 SubVendorId;
    U16 SubDeviceId;
    U8  Revision;
    U8  ApiIndex;          // Index used internally by the API
    U8  DeviceNumber;     // Number used internally by the device driver
} PLX_DEVICE_KEY;
```

### Purpose

Uniquely identifies a PCI device in a system.

### Members

#### *IsValidTag*

Reserved for internal use by the PLX API, **do not modify**

#### *bus*

The PCI device bus number

#### *slot*

The PCI device slot number

#### *function*

The PCI device function number

#### *VendorId*

The PCI device Vendor ID

#### *DeviceId*

The PCI device Device ID

#### *SubVendorId*

The PCI device subsystem Vendor ID

#### *SubDeviceId*

The PCI device subsystem Device ID

#### *Revision*

The PCI device revision

#### *ApiIndex*

Reserved for internal use by the PLX API, **do not modify**

#### *DeviceNumber*

Reserved for internal use by the PLX device driver, **do not modify**

## PLX\_DEVICE\_OBJECT

---

```
typedef struct _PLX_DEVICE_OBJECT
{
    U32                IsValidTag;
    PLX_DEVICE_KEY     Key;
    PLX_DRIVER_HANDLE  hDevice;
    PLX_PCI_BAR_SPACE  PciBar[6];    // Used internally for PCI BAR mapping
} PLX_DEVICE_OBJECT;
```

### Purpose

Describes a selected PCI device object.

### Members

*The members in this object should never be accessed directly. Its definition may change in future SDK versions and its members are reserved for internal use by the PLX API and PLX driver use.*

## PLX\_INTR

---

```
typedef struct _PLX_INTR
{
    unsigned int InboundPost      :1;
    unsigned int OutboundPost     :1;
    unsigned int OutboundOverflow :1;
    unsigned int OutboundOption  :1;
    unsigned int IopDmaChannel0   :1;
    unsigned int PciDmaChannel0   :1;
    unsigned int IopDmaChannel1   :1;
    unsigned int PciDmaChannel1   :1;
    unsigned int IopDmaChannel2   :1;
    unsigned int PciDmaChannel2   :1;
    unsigned int Mailbox0         :1;
    unsigned int Mailbox1         :1;
    unsigned int Mailbox2         :1;
    unsigned int Mailbox3         :1;
    unsigned int Mailbox4         :1;
    unsigned int Mailbox5         :1;
    unsigned int Mailbox6         :1;
    unsigned int Mailbox7         :1;
    unsigned int IopDoorbell      :1;
    unsigned int PciDoorbell      :1;
    unsigned int BIST             :1;
    unsigned int PowerManagement  :1;
    unsigned int PciMainInt       :1;
    unsigned int IopToPciInt      :1;
    unsigned int IopMainInt       :1;
    unsigned int PciAbort         :1;
    unsigned int PciReset         :1;
    unsigned int PciPME           :1;
    unsigned int Enum             :1;
    unsigned int IopBusTimeout    :1;
    unsigned int AbortLSERR       :1;
    unsigned int ParityLSERR      :1;
    unsigned int RetryAbort       :1;
    unsigned int LocalParityLSERR :1;
    unsigned int PciSERR          :1;
    unsigned int IopToPciInt_2    :1;
    unsigned int Message          :1;
    unsigned int SwInterrupt      :1;
    unsigned int SecResetDeassert :1;
    unsigned int SecPmeDeassert   :1;
    unsigned int GPIO14           :1;
    unsigned int GPIO4            :1;
} PLX_INTR;
```

### 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, PciDmaChannel2, & DmaChannel3*

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.

*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 INT1 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 INT0 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.

*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.

### SwInterrupt

The Software Interrupt feature of PLX slave devices (9050/9052/9030).

### SecResetDeassert

For 6254/6540/6466, represents S\_RSTIN# de-assertion interrupt.

### SecPmeDeassert

For 6254/6540/6466, represents S\_PME# de-assertion interrupt

### GPIO14

For 6254/6540/6466, represents GPIO14 interrupt

### GPIO4

For 6254/6540/6466, represents GPIO4 interrupt

## 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, 9056, 9656	9050, 9052, 9030	6254/6540/6466 (NT mode)
InboundPost	E 0x168, 4 A 0x168, 5	E 0x168, 4 A 0x168, 5		
OutboundPost	E 0xB4, 3 A 0xB0, 3	E 0xB4, 3 A 0xB0, 3		
OutboundOverflow	E 0x168, 6 A 0x168, 7	E 0x168, 6 A 0x168, 7		
OutboundOption				
PciDmaChannel0 <i>Note: DMA Interrupt is routed to PCI interrupt.</i>	E 0x100, 17 A 0xE8, 21	E 0x100, 17 A 0xE8, 21		
lopDmaChannel0	E 0xE8, 18 A 0xE8, 21	E 0xE8, 18 A 0xE8, 21		
PciDmaChannel1 <i>Note: DMA Interrupt is routed to PCI interrupt.</i>	E 0x114, 17 A 0xE8, 22	E 0x114, 17 A 0xE8, 22		
lopDmaChannel1	E 0xE8, 19 A 0xE8, 22	E 0xE8, 19 A 0xE8, 22		
PciDmaChannel2				
lopDmaChannel2				
Mailbox0 <i>Note: Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.</i>	E 0xE8, 3 A 0xE8, 28	E 0xE8, 3 A 0xE8, 28		E 0xC8, 24 A 0xCC, 16 C 0xCC, 16

Structure Element	9080	9054, 9056, 9656	9050, 9052, 9030	6254/6540/6466 (NT mode)
Mailbox1 <b>Note:</b> Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.	E 0xE8, 3 A 0xE8, 29	E 0xE8, 3 A 0xE8, 29		E 0xC8, 25 A 0xCC, 17 C 0xCC, 17
Mailbox2 <b>Note:</b> Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.	E 0xE8, 3 A 0xE8, 30	E 0xE8, 3 A 0xE8, 30		E 0xC8, 26 A 0xCC, 18 C 0xCC, 18
Mailbox3 <b>Note:</b> Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.	E 0xE8, 3 A 0xE8, 31	E 0xE8, 3 A 0xE8, 31		E 0xC8, 27 A 0xCC, 19 C 0xCC, 19
Mailbox4				
Mailbox5				
Mailbox6				
Mailbox7				
IopDoorbell	E 0xE8, 17 A 0xE8, 20	E 0xE8, 17 A 0xE8, 20		
PciDoorbell	E 0xE8, 9 A 0xE8, 13	E 0xE8, 9 A 0xE8, 13		E 0xC4, 0:15 A 0xCC, 0:15 C 0xCC, 0:15
BIST <b>Note:</b> There is no enable or disable functionality for BIST for PCI9054, PCI9080, and PCI9x56	A 0xE8, 23	A 0xE8, 23		
PowerManagement		E 0xE8, 4 A 0xE8, 5 C 0xE8, 5		
PciMainInt	E 0xE8, 8	E 0xE8, 8	E 0x4C, 6	
IopToPciInt	E 0xE8, 11 A 0xE8, 15	E 0xE8, 11 A 0xE8, 15	E 0x4C, 0 + 8 A 0x4C, 2 C 0x4C, 10	
IopToPciInt_2			E 0x4C, 3 + 9 A 0x4C, 5 C 0x4C, 11	
IopMainInt <b>Note:</b> 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, 16		
PciAbort	E 0xE8, 10 A 0xE8, 14	E 0xE8, 10 A 0xE8, 14		
PciReset				
PciPME		E 0xE8, 4 A 0xE8, 5		
Enum			E 0x4A, 1 A 0x4A, 6-7	
AbortLSERR	E 0xE8, 0 A 0x06, 12-13	E 0xE8, 0 A 0x06, 12-13		

Structure Element	9080	9054, 9056, 9656	9050, 9052, 9030	6254/6540/6466 (NT mode)
ParityLSERR	E 0xE8, 1 A 0x06, 15	E 0xE8, 1 A 0x06, 15		
RetryAbort	E 0xE8, 12 A 0x06, 12-13	E 0xE8, 12 A 0x06, 12-13		
LocalParityLSERR		E 0xE8, 6 A 0xE8, 7		
PciSERR				
TargetAbort				
DMAabort				
SecResetDeassertion				E 0xC8, 28 A 0xCC, 20 C 0xCC, 20
SecPmeDeassertion				E 0xC8, 29 A 0xCC, 21 C 0xCC, 21
GPIO14 <i>Note: The actual source of the interrupt must be cleared to properly de-assert this interrupt.</i>				E 0xC8, 30 A 0xCC, 22
GPIO4 <i>Note: The actual source of the interrupt must be cleared to properly de-assert this interrupt.</i>				E 0xC8, 31 A 0xCC, 23

## PLX\_POWER\_LEVEL

---

```
typedef enum _PLX_POWER_LEVEL
{
    D0Uninitialized,
    D0,
    D1,
    D2,
    D3Hot,
    D3Cold
} PLX_POWER_LEVEL;
```

### 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.

## PM\_PROP

---

```
typedef struct _PM_PROP
{
    unsigned int Version                :3;
    unsigned int PMEClockNeeded        :1;
    unsigned int DeviceSpecialInit     :1;
    unsigned int D1Supported           :1;
    unsigned int D2Supported           :1;
    unsigned int AssertPMEfromD0      :1;
    unsigned int AssertPMEfromD1      :1;
    unsigned int AssertPMEfromD2      :1;
    unsigned int AssertPMEfromD3Hot   :1;
    unsigned int Read_Set_State        :2;
    unsigned int PME_Enable            :1;
    unsigned int PME_Status            :1;
    unsigned int PowerDataSelect       :3;
    unsigned int PowerDataScale        :2;
    unsigned int PowerDataValue       :8;
    unsigned int Reserved              :4;
} PM_PROP;
```

### 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 initialization following a transition to DR<sub>0</sub>R uninitialized state.

#### *D1Supported*

When set to 1, the PLX chip supports the DR<sub>1</sub>R power state.

#### *D2Supported*

When set to 1, the PLX chip supports the DR<sub>2</sub>R power state.

#### *AssertPMEfromD0*

When set to 1, PME# can be asserted from power state DR<sub>0</sub>R.

#### *AssertPMEfromD1*

When set to 1, PME# can be asserted from power state DR<sub>1</sub>R.

#### *AssertPMEfromD2*

When set to 1, PME# can be asserted from power state DR<sub>2</sub>R.

#### *AssertPMEfromD3Hot*

When set to 1, PME# can be asserted from power state DR<sub>3hot</sub>R.

#### *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.

**Affected Register Location**

Structure Element	9080	9054	9x56
Version	N/A	0x182, 0-2	0x182, 0-2
PMEClockNeeded	N/A	0x182, 3	0x182, 3
DeviceSpecialInit	N/A	0x182, 5	0x182, 5
D1Supported	N/A	0x182, 9	0x182, 9
D2Supported	N/A	0x182, 10	0x182, 10
AssertPMEfromD0	N/A	0x182, 11	0x182, 11
AssertPMEfromD1	N/A	0x182, 12	0x182, 12
AssertPMEfromD2	N/A	0x182, 13	0x182, 13
AssertPMEfromD3Hot	N/A	0x182, 14	0x182, 14
Read_Set_State	N/A	0x184, 0-1	0x184, 0-1
PME_Enable	N/A	0x184, 8	0x184, 8
PME_Status	N/A	0x184, 15	0x184, 15
PowerDataSelect	N/A	0x184, 9-12	0x184, 9-12
PowerDataScale	N/A	0x184, 13-14	0x184, 13-14
PowerDataValue	N/A	0x187, 0-7	0x187, 0-7

## Appendix A PLX SDK Revision Notes

### A.1 Host API Functions Added

SDK Version 4.4:

Function Name	Page
PlxPciRegisterReadFast	2-97
PlxPciRegisterWriteFast	2-98

SDK Version 4.3:

Function Name	Page
PlxNotificationCancel	2-54
PlxNotificationRegisterFor	2-56
PlxNotificationWait	2-58

SDK Version 4.1:

Function Name	Page
PlxIntrWait ( <i>Added for Windows</i> )	2-41

SDK Version 3.5:

Function Name	Page
PlxIntrWait ( <i>Linux only</i> )	2-41
PlxPciBarGet	2-60
PlxPciBarMap	2-62
PlxPciBarUnmap	2-66
PlxPciCommonBufferProperties	2-73
PlxPciCommonBufferMap	2-71
PlxPciCommonBufferUnmap	2-75
PlxPciPhysicalMemoryAllocate	2-88
PlxPciPhysicalMemoryFree	2-91
PlxPciPhysicalMemoryMap	2-93
PlxPciPhysicalMemoryUnmap	2-95

SDK Version 3.4:

Function Name	Page
PlxPciRegisterRead_Unsupported	2-100
PlxPciRegisterWrite_Unsupported	2-102

SDK Version 3.3:

Function Name	Page
PlxSerialEepromReadByOffset	2-124
PlxSerialEepromWriteByOffset	2-127

SDK Version 3.2:

API Function	Page
PlxChipTypeGet	2-9
PlxDriverVersion	2-29
PlxPciBoardReset	2-69

## A.2 Host API Functions Removed

SDK Version 4.40:

Function Name	Replacement Function(s)	Page
PlxPciCommonBufferGet	PlxPciCommonBufferProperties	2-73
PlxSerialEepromRead	PlxSerialEepromReadByOffset	2-124
PlxSerialEepromWrite	PlxSerialEepromWriteByOffset	2-127

SDK Version 4.30:

Function Name	Replacement Function(s)	Page
PlxIntrAttach	PlxNotificationRegisterFor	2-56
PlxIntrWait	PlxNotificationWait	2-58
PlxPciBaseAddressesGet	PlxPciBarMap	2-68
PlxPmIdRead	PlxPciRegisterReadFast	2-97
PlxPciHotSwapIdRead	PlxPciRegisterReadFast	2-97
PlxPciVpdIdRead	PlxPciRegisterReadFast	2-97
PlxPciAbortAddrRead	PlxRegisterRead	2-116

SDK Version 3.5:

Function Name	Replacement Function(s)	Page
PlxPciConfigRegisterReadAll	PlxPciRegisterReadFast	2-97
PlxRegisterReadAll	PlxRegisterRead	2-116
PlxDmaBlockTransferRestart	PlxDmaBlockTransfer	2-15
PlxDmaShuttleChannelOpen PlxDmaShuttleTransfer PlxDmaShuttleChannelClose	N/A	
PlxPciCommonBufferGet	PlxPciCommonBufferProperties	2-73
	PlxPciCommonBufferMap	2-71
	PlxPciCommonBufferUnmap	2-75
PlxPciBaseAddressesGet	PlxPciBarMap	2-62
	PlxPciBarUnmap	2-66

SDK Version 3.4:

Function Name	Replacement Function(s)	Page
PlxPciBusSearch	PlxPciDeviceFind	2-82
PlxPinSetup	N/A	
PlxUserRead	PlxRegisterRead	2-116
PlxUserWrite	PlxRegisterWrite	2-118

### A.3 The Structure of the Defined Type Changed

Some structures were re-defined between SDK versions. The reasons for this were in the interest of performance enhancements and code compatibility between different environments, such as Windows and Linux. The modifications are listed below.

Structure: **DMA\_TRANSFER\_ELEMENT**

File: **PlxTypes.h**

Functions: **PlxDmaBlockTransfer**

**PlxDmaSgITransfer**

Modification:

Before	After
<pre>Union {     U32 LowPciAddr;     U32 UserAddr; };</pre>	<pre>union {     U32 UserVa;     U32 PciAddrLow; } u;</pre>

Structure: **DEVICE\_LOCATION**

File: **PciTypes.h**

Functions: **PlxPciDeviceOpen**

**PlxPciDeviceFind**

Modification:

Before	After
<pre>typedef struct _DEVICE_LOCATION {     U32 DeviceId;     U32 VendorId;     U32 BusNumber;     U32 SlotNumber;     U8  SerialNumber[16]; } DEVICE_LOCATION;</pre>	<pre>typedef struct _DEVICE_LOCATION {     U8  BusNumber;     U8  SlotNumber;     U16 DeviceId;     U16 VendorId;     U8  SerialNumber[20]; } DEVICE_LOCATION;</pre>

### A.4 Function Argument or Type Changed

In SDK 3.5, the arguments of some function were changed. The following is a list of tables, which list the function prototypes that have been modified. Both the previous and current prototypes are provided.

Before	After
<pre> U32 PlxVpdRead(     HANDLE    hDevice,     U32       offset,     RETURN_CODE *pReturnCode ); </pre>	<pre> U32 PlxVpdRead(     HANDLE    hDevice,     U16       offset,     RETURN_CODE *pReturnCode ); </pre>

Before	After
<pre> RETURN_CODE PlxVpdWrite(     HANDLE hDevice,     U32    offset,     U32    vpdData ); </pre>	<pre> RETURN_CODE PlxVpdWrite(     HANDLE hDevice,     U16    offset,     U32    value ); </pre>

Before	After
<pre> RETURN_CODE PlxPciBarRangeGet(     HANDLE hDevice,     U32    barRegisterNumber,     U32    *data ); </pre>	<pre> RETURN_CODE PlxPciBarRangeGet(     HANDLE hDevice,     U8     BarIndex,     U32    *pData ); </pre>

Before	After
<pre> U32 PlxPciConfigRegisterRead(     U32    bus,     U32    slot,     U32    offset,     RETURN_CODE *pReturnCode ); </pre>	<pre> U32 PlxPciConfigRegisterRead(     U8     bus,     U8     slot,     U16    offset,     RETURN_CODE *pReturnCode ); </pre>

Before	After
<pre> RETURN_CODE PlxPciConfigRegisterWrite(     U32 bus,     U32 slot,     U32 offset,     U32 *pData ); </pre>	<pre> RETURN_CODE PlxPciConfigRegisterWrite(     U8  bus,     U8  slot,     U16 offset,     U32 *pValue ); </pre>

Before	After
<pre> U32 PlxRegisterRead(     HANDLE      hDevice,     U32         registerOffset,     RETURN_CODE *pReturnCode ); </pre>	<pre> U32 PlxRegisterRead(     HANDLE      hDevice,     U16         offset,     RETURN_CODE *pReturnCode ); </pre>

Before	After
<pre> RETURN_CODE PlxRegisterWrite(     HANDLE hDevice,     U32    registerOffset,     U32    data ); </pre>	<pre> RETURN_CODE PlxRegisterWrite(     HANDLE hDevice,     U16    offset,     U32    value ); </pre>