PLX SDK
SOFTWARE DEVELOPMENT KIT

Addendum

Version 3.5

June 2002

Website: http://www.plxtech.com

Email: apps@plxtech.com

Phone: 408 774-9060

Fax: 408 774-2169

Document number: PCI-SDK-ADDENDUM-P1-3.5

Table of Contents

1 General Information

PLX Technology offers PCI bus interface chips that address a range of adapter and embedded system applications. PLX also provides additional software to aid customers in debug and for application development.

1.1 About This Document

This document is an addendum to the SDK version 3.2 manuals; therefore, it should be used along with those manuals. It contains the changes, additions, and enhancements provided in SDK versions 3.3, 3.4, and 3.5. Unless otherwise stated, the information in the SDK 3.2 manuals still applies. This document will provide any updates, as well as corrections, from the 3.2 manuals.

1.2 Where to Go From Here

The PLX SDK ships with 3 manuals. A section in this document is devoted to each manual and contains the updates relevant to each. The sections are as follows:

Chapter 2, What's New contains general updates for SDK versions 3.3, 3.4, and 3.5.

Chapter 3, PLX SDK User's Manual Updates contains updates for the PLX SDK User's Manual, v3.2.

Chapter 4, PLX SDK Programmer's Reference Manual Updates, contains updates for the PLX SDK Programmer's Reference Manual, v3.2.

Chapter 5, PLXMon User's Manual Updates, contains updates for the PLXMon User's Manual, v3.2.

1.3 Terminology

- References to Windows NT assume Windows NT 4.0 and may be denoted as WinNT.
- References to Windows 98 may be denoted as Win98.
- References to Windows 2000 may be denoted as Win2k.
- References to Windows XP may be denoted as WinXP.
- References to Windows Me may be denoted as WinMe.
- References to Visual C/C++ or Visual C++ refer to Microsoft Visual C/C++ 6.0.
- References to Win32 denote any application that is compatible with the Windows 32-bit environment.
- Any references to IOP (I/O Platform) or Local-side throughout this manual denote a Custom board with a PLX chip or a PLX RDK board. All references to IOP or Local software denote the software running on the board.

2 What's New

2.1 SDK 3.5

2.1.1 SDK Enhancements

SDK v3.5 contains the enhancements listed below. *Refer to the SDK Readme file for details regarding changes and bug fixes.*

- Addition of Host API calls to allocate physically contiguous non-paged memory.
- Fully functional Linux Host drivers supporting all PLX chips and Linux kernels 2.2 and 2.4.
- Numerous performance enhancements to PLX Host-side API and drivers.
    - Some API calls were completely re-written to make them more efficient.
    - The "protocol" between the Host API and driver was improved by minimizing the amount of data transferred between them when an API function is called.

2.2 SDK 3.4

2.2.1 SDK Additions

SDK 3.4 provides the following additions not found in previous SDKs:

- PLX stand-alone BSP for the Compact PCI 9056RDK-860, including Ethernet features.
- PLX Local API for the PLX 9056 chip.
- New VxWorks BSP which supports the Compact PCI 9656-860 and 9056-860 RDKs
- New Windows Host API calls to access PCI configuration register of any device, including non-PLX devices, in a WDM environment.
- Special build of the Windows API DLL to support Visual Basic users who need to use the Windows Host API.

2.2.2 SDK Enhancements

SDK v3.4 contains the enhancements listed below. *Refer to the SDK Readme file for details regarding bug fixes.*

- Many enhancements and bug fixes to the PLXMon application.
- Updated Manufacturing Test to support PLX 9052 RDK-LITE
- Various Windows driver bug fixes
- First phase in update of Linux Host drivers.

2.2.3 SDK Compatibility with Windows XP

The PLX WDM drivers have been modified and tested to support Windows XP. The same WDM driver is used for all Windows Plug 'n' Play operating systems, including Win98, WinMe, Win2000, and WinXP. The Windows 2000 DDK is used when building PLX WDM drivers.

2.3 SDK 3.3

2.3.1 SDK Additions

SDK 3.3 provides the following additions not found in previous SDKs:

- PLXMon support for the PLX 9656 chip
- PLXMon support for the PLX 9056 chip
- Windows Host driver and Host API support for the PLX 9656 chip
- Windows Host driver and Host API support for the PLX 9056 chip
- PLX stand-alone BSP for the CompactPCI 9656RDK-860, including Ethernet features.
- PLX Local API for the PLX 9656 chip.
- Updated Linux Host driver to additionally support the 9030 and 9052 as well as enhancements to the existing 9054 support. *Refer to the Linux Host Support Release Notes for additional information.*

2.3.2 SDK Enhancements

SDK v3.3 contains the enhancements listed below. *Refer to the SDK Readme file for details regarding bug fixes.*

- Various PLXMon register screen enhancements
- Various Windows driver bug fixes
- Additional Host and Local API functions to access EEPROM locations at a specified offset.
- Support for the 9030 and 9050/9052 software interrupt

2.3.3 SDK Compatibility with Windows Me

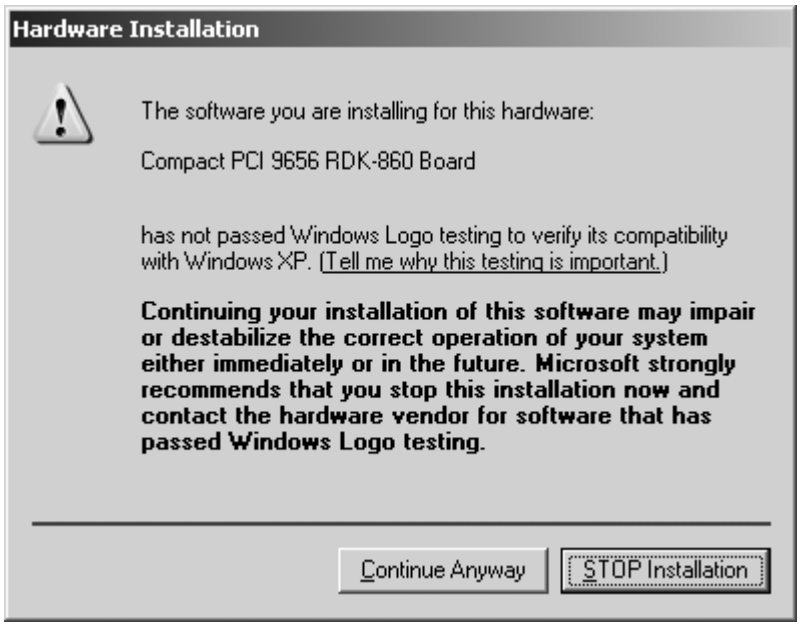
The PLX WDM drivers have been tested and are compatible with Windows Me.

3 PLX SDK User's Manual Updates

3.1 Installation of Drivers in Windows XP & Windows 2000

The installation of the PLX device drivers in Windows XP is essentially the same as Windows 2000/98 and is documented in the SDK 3.2 User's Manual. Windows XP and Windows 2000 introduced "**Driver Signing**", in which drivers are certified by Microsoft and a digital signature added to inform the OS of this.

Since PLX drivers are generic and are provided for general use and as a reference for customer development, they are have not been certified by Microsoft. As a result, during driver installation, a dialog box similar to the following will be displayed to inform the user that the driver is not signed.

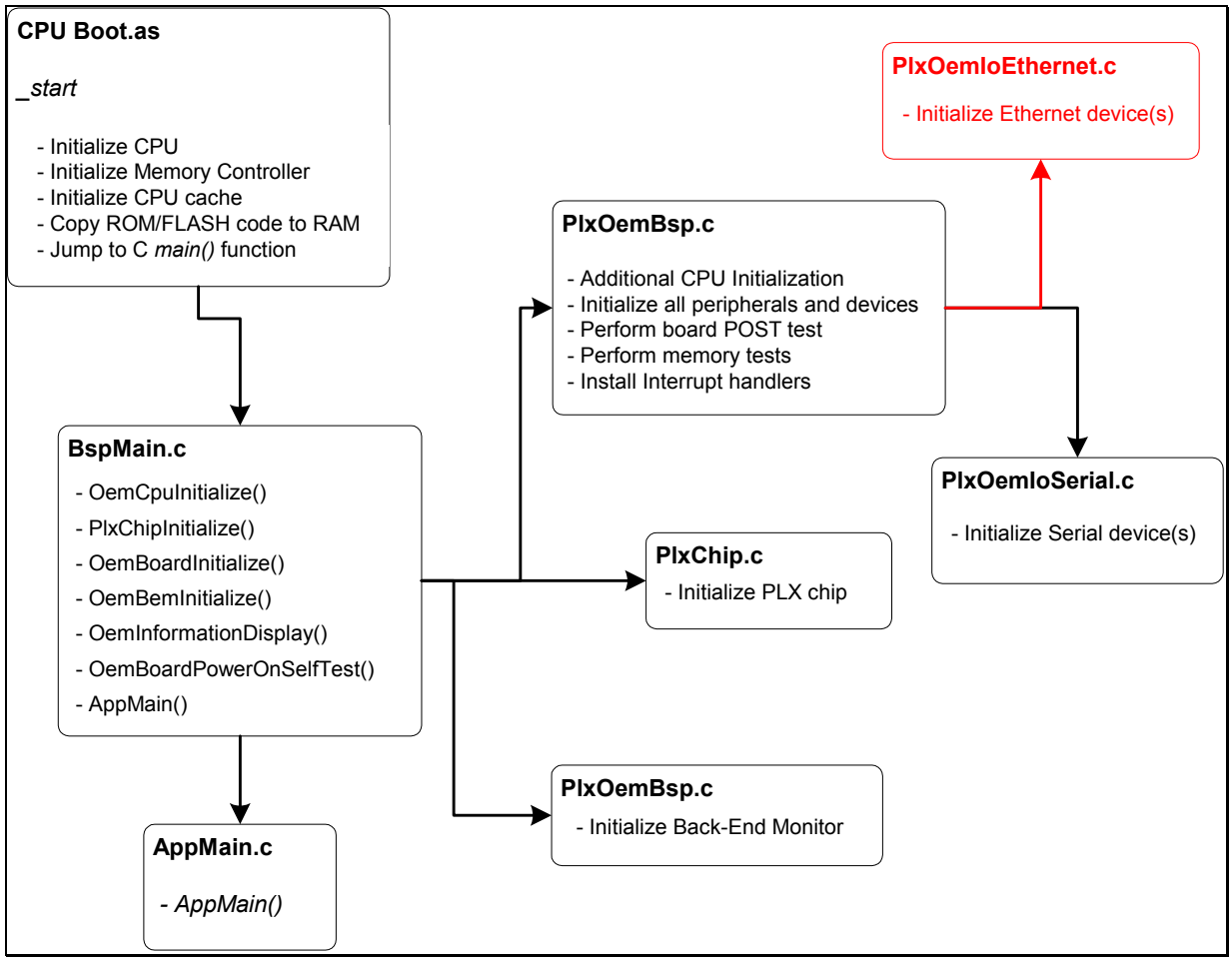


This dialog can be ignored by selecting the "*Continue Anyway*" button. There is nothing wrong with installing the PLX driver; this is just an informational message.

Users interested in additional information should refer to the Microsoft documentation on Driver Signing.

3.2 PLX BSP Flowchart

The PLX stand-alone BSP flowchart has been modified to add support for Ethernet devices. Ethernet support is added for the Compact PCI 9656-860 RDK. The following diagram shows the updated flow:



3.3 New Back-End-Monitor (BEM) Commands in SDK 3.4

The following commands were added to the BEM to support access to a specific 32-bit location in the EEPROM.

Read single 32-bit data from a specified offset of the EEPROM

***Note**: EEPROM access must be supported by the local-side. This is reported in the capabilities of the Platform Information.*

Command Message:

2 Bytes	1 Byte	1 Byte	1 Byte
Message Header	EEP_READ_BY_OFFSET	<Offset>	End-Message

Where:
Offset = The offset to read from EEPROM (must be aligned on 32-bit boundary)

Reply Message:

1 Byte	1 Byte	4 Bytes	1 Byte
Reply Header	<Status>	<Data>	End-Message

Where:
Status = REPLY_SUCCESS or REPLY_ERROR
Data = The 32-bit data read from the EEPROM if status is REPLY_SUCCESS

Write single 32-bit data to a specified offset of the EEPROM

***Note**: EEPROM access must be supported by the local-side. This is reported in the capabilities of the Platform Information.*

Command Message:

2 Bytes	1 Byte	1 Byte	*4* Bytes	1 Byte
Message Header	EEP_WRITE_BY_OFFSET	<Offset>	<Data>	End-Message

Where:
Offset = The offset to write to the EEPROM (must be aligned on 32-bit boundary)
Data = The 32-bit value to write to EEPROM

Reply Message:

1 Byte	1 Byte	1 Byte
Reply Header	<Status>	End-Message

Where:
Status = REPLY_SUCCESS or REPLY_ERROR

3.3.1 Additional Functions Required in the BSP

To support the new BEM commands listed above, two new functions are now required in the BSP. They are listed below and their implementation can be found in the *PlxOemBemSupport.c* file of the respective BSP.

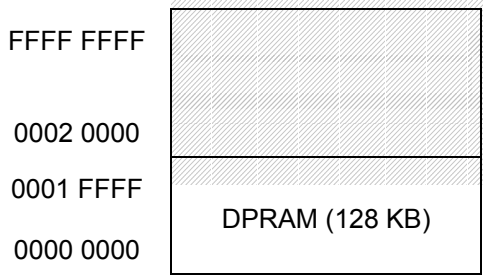
- **OemEepromReadByOffset**
- **OemEepromWriteByOffset**

3.4 Support for New PLX Rapid Development Kits (RDK)

The following sections provide relevant information for each new PLX RDKs.

3.4.1 PCI 9656 RDK-LITE

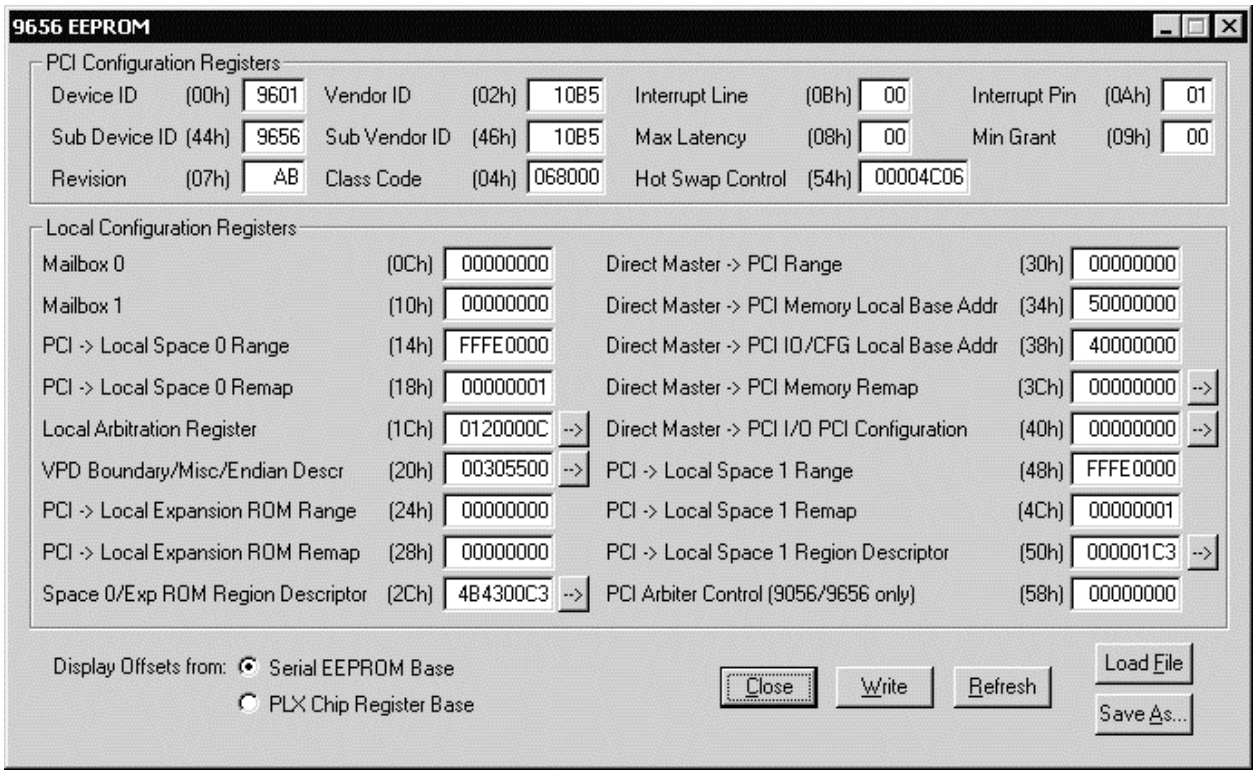
Memory map:



Board properties:

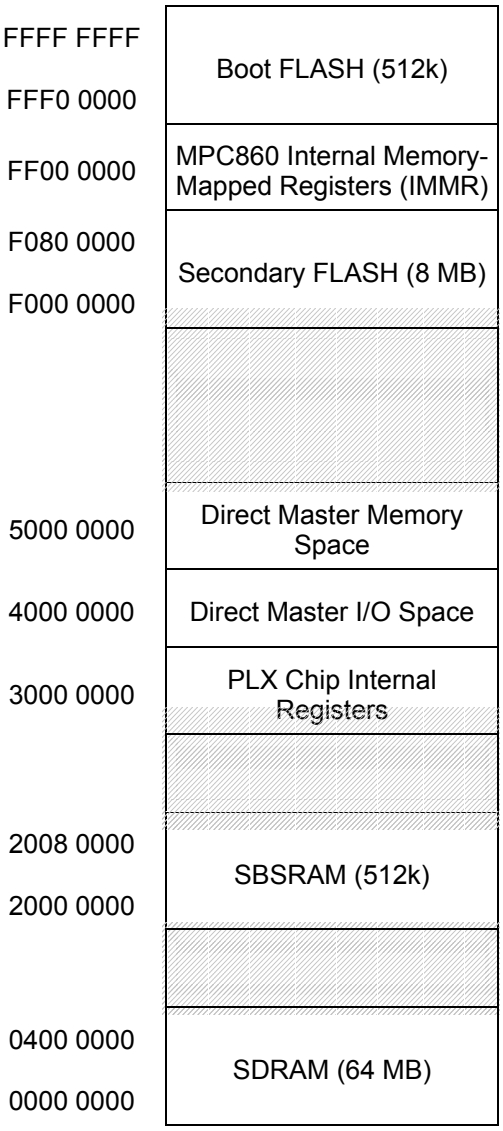
Device/Vendor ID:	9601_10b5
FLASH Type:	None
FLASH Program Offset:	N/A
EEPROM Type:	NM93CS56

EEPROM values:



3.4.2 Compact PCI 9656-860 RDK

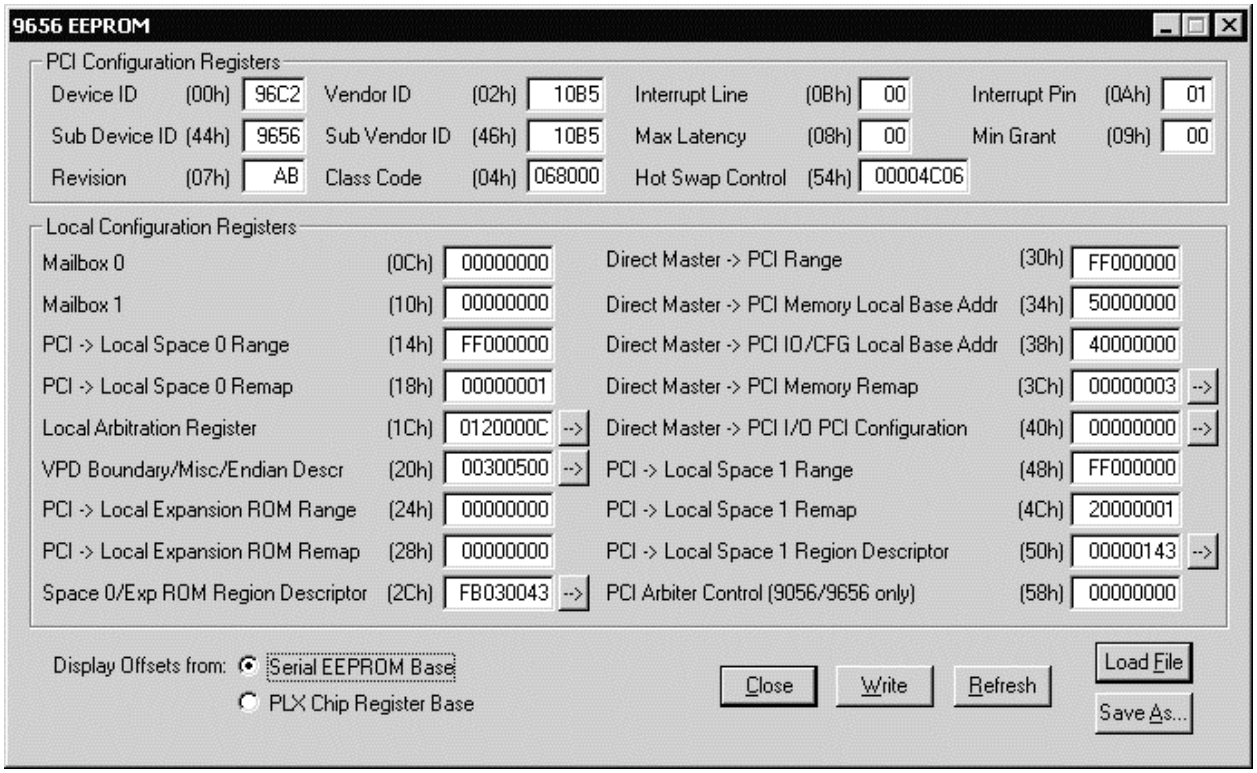
Memory map:



Board properties:

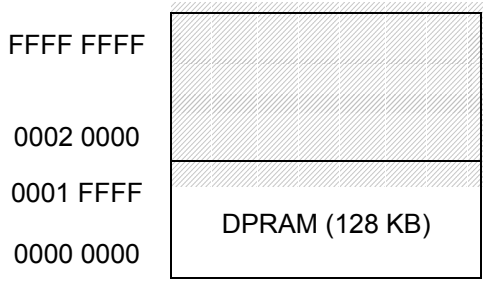
Device/Vendor ID:	96c2_10b5
FLASH Type:	ATMEL AT49LV040 or AMD 29LV040B
FLASH Program Offset:	0x0000 0000
EEPROM Type:	NM93CS56

EEPROM Values:



3.4.3 PCI 9056 RDK-LITE

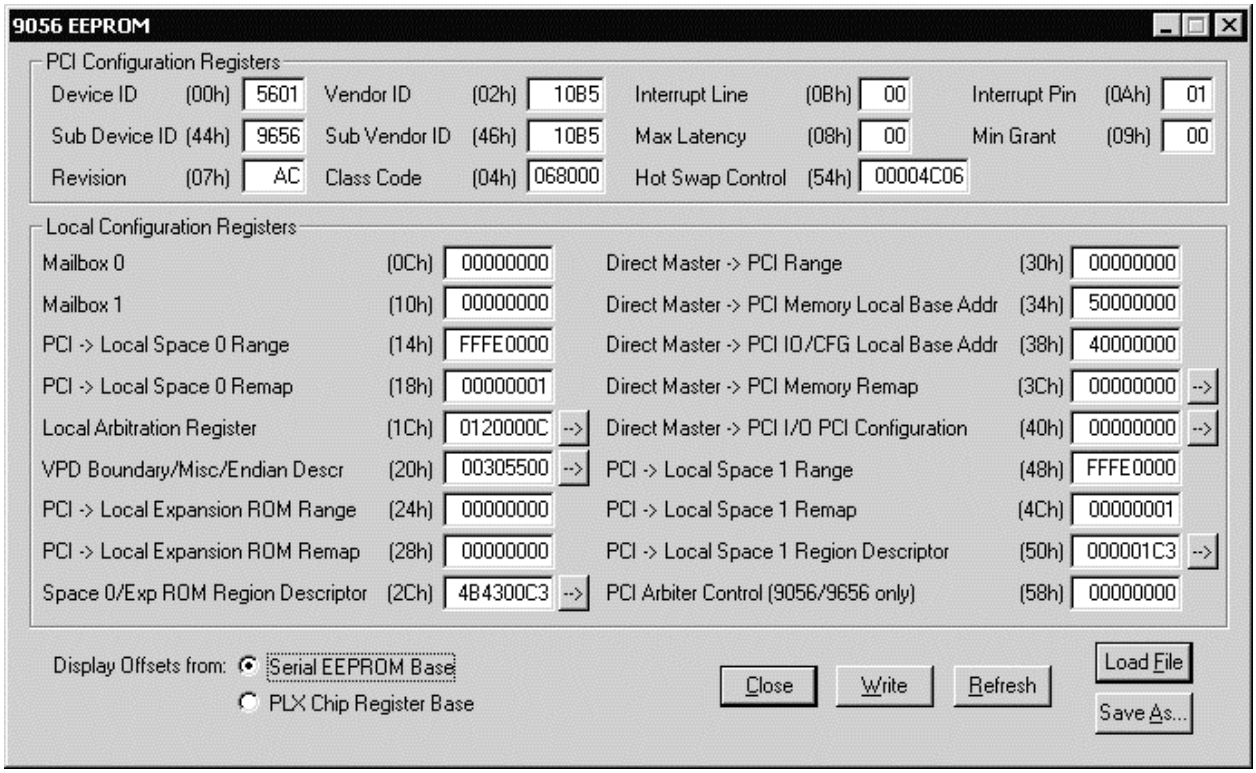
Memory map:



Board properties:

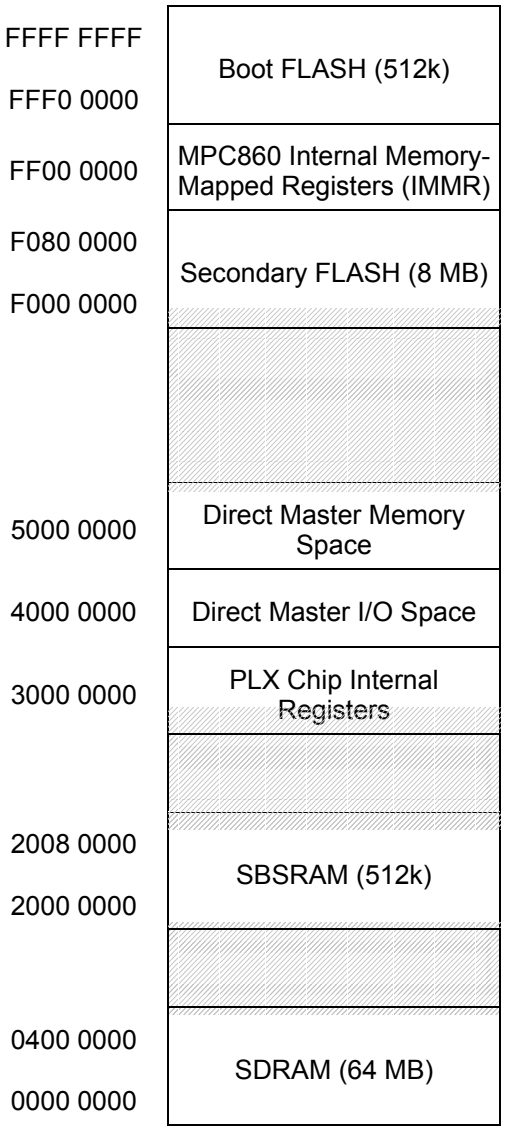
Device/Vendor ID:	5601_10b5
FLASH Type:	None
FLASH Program Offset:	N/A
EEPROM Type:	NM93CS56

EEPROM values:



3.4.4 Compact PCI 9056-860 RDK

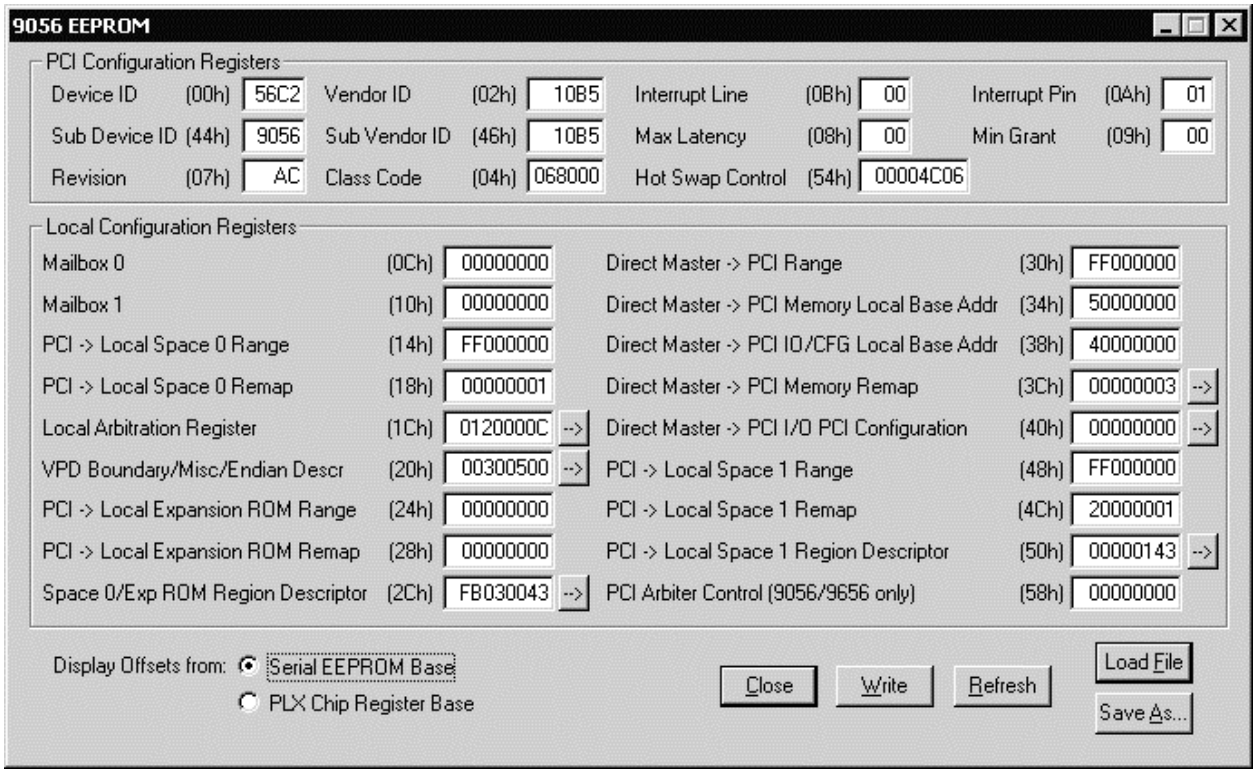
Memory map:



Board properties:

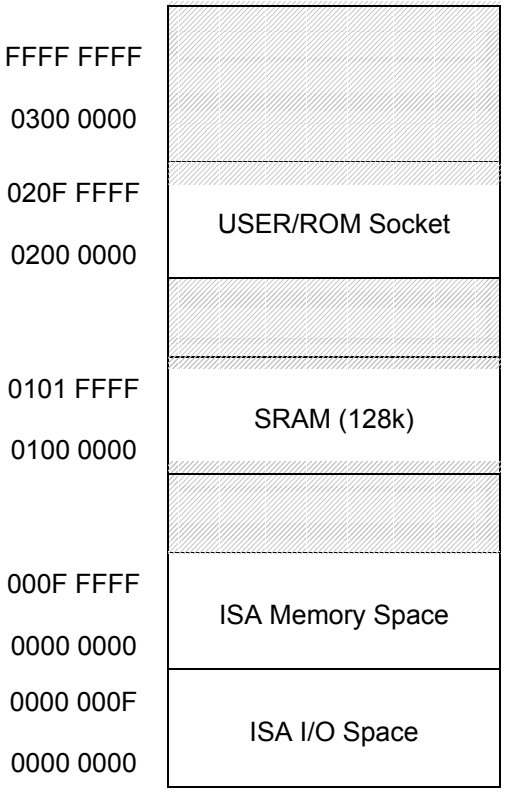
Device/Vendor ID:	56c2_10b5
FLASH Type:	ATMEL AT49LV040 or AMD 29LV040B
FLASH Program Offset:	0x0000 0000
EEPROM Type:	NM93CS56

EEPROM Values:



3.4.5 PCI 9052 RDK-LITE

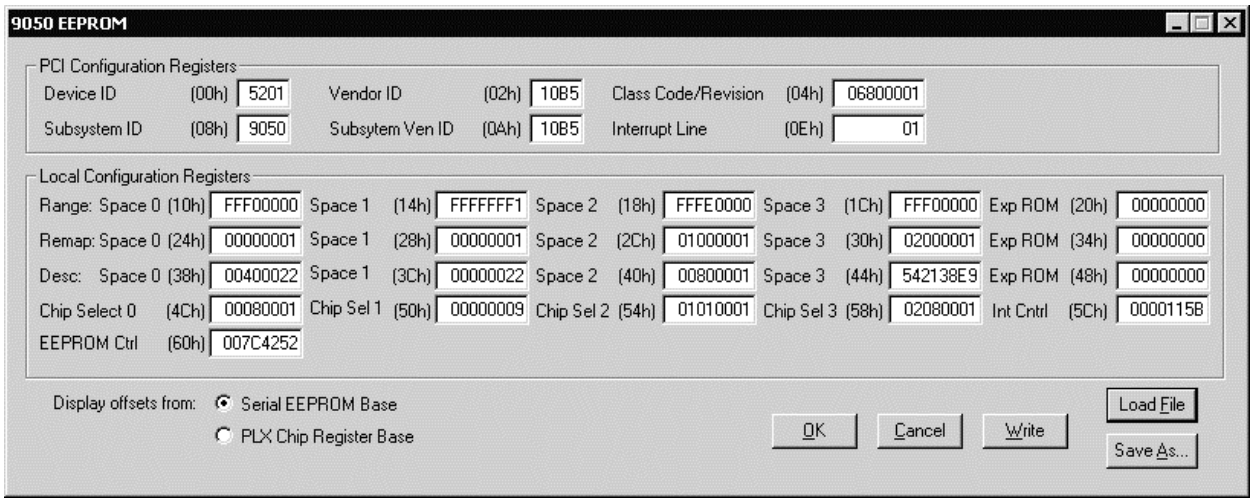
Memory map:



Board properties:

Device/Vendor ID:	5201_10b5
FLASH Type:	None
FLASH Program Offset:	N/A
EEPROM Type:	NM93CS46

EEPROM Values:



4 PLX SDK Programmer's Reference Manual Updates

4.1 9030 and 9050/9052 Software Interrupt

In SDK 3.3, support was added for the Software Interrupt feature of the 9030 and 9050/9052 chips. This interrupt allows for manual triggering of the PCI interrupt. Previous Windows drivers in SDKs ignored this interrupt, resulting in system hangs. The driver will now detect the interrupt, clear it, and signal any application that has registered for interrupt notification.

The following items have been changed:

- The Interrupt Service Routine and the ISR DPC routine of the 9030 and 9050/9052 Windows drivers have been modified to handle the interrupt.
- The field *SwInterrupt* has been added to the PLX Interrupt structure (*PLX_INTR*). *Refer to PlxTypes.h for the new structure definition.*

4.2 Local API

4.2.1 Local API Functions Removed in SDK 3.4

The following functions were removed from the Local API:

- **PlxRegisterReadAll**
- **PlxPinSetup**
- **PlxUserRead**
- **PlxUserWrite**

4.2.2 New Local API functions in SDK 3.3

The Local API now contains functions to access 32-bit values anywhere in the serial EEPROM. They are documented below.

PlxSerialEepromReadByOffset

Syntax:

```
RETURN_CODE
PlxSerialEepromReadByOffset(
    BUS_INDEX  busIndex,
    U16        offset,
    U32       *pValue
    );
```

PLX Chip Support:

9054, 9056, 9656, 480

Description:

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

Parameters:

busIndex
The bus index

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
ApiInvalidBusIndex	The BUS_INDEX is invalid
ApiNullParam	One or more parameters is NULL
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:

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

Usage:

```
U32         EepromData;
RETURN_CODE rc;


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

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

PlxSerialEepromWriteByOffset

Syntax:

```
RETURN_CODE
PlxSerialEepromWriteByOffset(
    BUS_INDEX busIndex,
    U16       offset,
    U32       value
    );
```

PLX Chip Support:

9054, 9056, 9656, 480

Description:

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

Parameters:

busIndex
The bus index

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
ApiInvalidBusIndex	The BUS_INDEX is invalid
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:

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

Usage:

```
RETURN_CODE rc;


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

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

4.3 Host API

4.3.1 SDK 3.2 Programmer's Reference Manual Correction

In the SDK 3.2 Programmer's Reference Manual, the ***PlxIoPortRead*** and ***PlxIoPortWrite*** Host API functions contain the following statement:

> ***PlxIoPortRead()*** *and* ***PlxIoPortWrite()*** *are obsolete and exported only to support existing applications. Applications should us the C-code* ***inp()(/outp()*** *I/O access functions (or equivalent) instead.*

After the release of SDK 3.2, it was determined that the Windows OS versions based on Windows NT (WinNT, WinXP, Win2000) do not support the standard C I/O Port access functions ***inp()*** and ***outp()***, and, in fact, generate an exception if these functions are used.

The PLX API functions were never actually removed, in order to support existing applications. The note mentioned above in the SDK 3.2 manual can be completely ignored. PLX will continue to provide I/O port functions in this SDK and future releases.

4.3.2 Host API Functions Removed in SDK 3.5

The following functions were removed from the Host API:

- **PlxPciConfigRegisterReadAll**
- **PlxRegisterReadAll**
- **PlxDmaBlockTransferRestart**
- **PlxDmaShuttleChannelOpen**
- **PlxDmaShuttleTransfer**
- **PlxDmaShuttleChannelClose**
    - The Shuttle DMA API calls were determined not to be useful since user applications do not have control over the physical placement of memory in a virtual memory environment.
- **PlxPciCommonBufferGet**
    - This function is still supported, but should no longer be used. It has been replaced with the new API functions: *PlxPciCommonBufferProperties*, *PlxPciCommonBufferMap*, *PlxPciCommonBufferUnmap*, which are documented in this manual.
- **PlxPciBaseAddressesGet**
    - This function is still supported, but should no longer be used. It has been replaced with the new API functions: *PlxPciBarMap* and *PlxPciBarUnmap*, which are documented in this manual.

4.3.3 Host API Functions Removed in SDK 3.4

The following functions were removed from the Host API:

- **PlxPciBusSearch**
- **PlxPinSetup**
- **PlxUserRead**
- **PlxUserWrite**

4.3.4 Host API Functions Added in SDK 3.5

SDK 3.5 adds the Host API functions listed below. *Refer to Section 4.3.7 New Host API Functions Reference for function details*.

- **PlxPciBarGet**
- **PlxPciBarMap**
- **PlxPciBarUnmap**
- **PlxPciCommonBufferProperties**
- **PlxPciCommonBufferMap**
- **PlxPciCommonBufferUnmap**
- **PlxPciPhysicalMemoryAllocate**
- **PlxPciPhysicalMemoryFree**
- **PlxPciPhysicalMemoryMap**
- **PlxPciPhysicalMemoryUnmap**
- **PlxIntrWait** (Linux only)

4.3.5 Host API Functions Added in SDK 3.4

SDK 3.4 adds the Host API functions listed below. *Refer to Section 4.3.7 New Host API Functions Reference for function details*.

- **PlxPciRegisterRead_Unsupported**
- **PlxPciRegisterWrite_Unsupported**

4.3.6 Host API Functions Added in SDK 3.3

SDK 3.3 adds the Host API functions listed below. *Refer to Section 4.3.7 New Host API Functions Reference for function details*.

- **PlxSerialEepromReadByOffset**
- **PlxSerialEepromWriteByOffset**

4.3.7 New Host API Functions Reference

This section contains details of the newly added API functions not found in the SDK 3.2 printed manual.

PlxPciBarGet

Syntax:

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

PLX Chip Support:

All

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 0-6.

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 ocurred
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

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 0-6.

pVa
Pointer to a buffer which will contain the base virtual address

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
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[7];
HANDLE      hDevice;
RETURN_CODE rc;


for (i = 0; i <= 6; 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"
    "      EROM  VA:  0x%08x\n",  // Expansion ROM space
    Va[0], Va[1], Va[2],
    Va[3], Va[4], Va[5], Va[6]
    );

/*************************************************************
 * 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;
```

PlxPciBarUnmap

Syntax:

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

PLX Chip Support:

All

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 ocurred
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
}
```

PlxPciCommonBufferProperties

Syntax:

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

PLX Chip Support:

All

Description:

This function returns the common buffer properties.

Parameters:

hDevice
Handle of an open PCI device

pMemoryInfo
A pointer to a PCI_MEMORY structure which will contain information about the common buffer

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
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;
PCI_MEMORY   BufferInfo;
RETURN_CODE  rc;


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

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

printf(
    "Common buffer information:\n"
    "     PCI address:  %08x\n"
    "     Size       :  %d bytes\n",
    BufferInfo.PhysicalAddr, BufferInfo.Size
    );
```

PlxPciCommonBufferMap

Syntax:

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

PLX Chip Support:

All

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 ocurred
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 Enties (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;
U32         BufferVa;
HANDLE      hDevice;
PCI_MEMORY  BufferInfo;
RETURN_CODE rc;


// 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);
```

PlxPciCommonBufferUnmap

Syntax:

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

PLX Chip Support:

All

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 ocurred
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:

```
U32         BufferVa;
HANDLE      hDevice;
PCI_MEMORY  BufferInfo;
RETURN_CODE rc;


// 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
}

//
// 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
}
```

PlxPciPhysicalMemoryAllocate

Syntax:

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

PLX Chip Support:

All

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 PCI_MEMORY 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 ocurred
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 2 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;
PCI_MEMORY  Buffer_1;
PCI_MEMORY  Buffer_2;
RETURN_CODE rc;


// 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
}
```

PlxPciPhysicalMemoryFree

Syntax:

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

PLX Chip Support:

All

Description:

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

Parameters:

hDevice
Handle of an open PCI device

pMemoryInfo
A pointer to a PCI_MEMORY structure which contains the buffer information.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
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 *PlxPciPhysicalMemoryAllocate*

Notes:

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

If the buffer was mapped to user virtual space (with *PlxPciPhysicalMemoryMap*), it should be unmapped 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;
PCI_MEMORY  Buffer;
RETURN_CODE rc;


// 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
}
```

PlxPciPhysicalMemoryMap

Syntax:

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

PLX Chip Support:

All

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 PCI_MEMORY structure which contains the buffer information.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
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 Enties (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;
PCI_MEMORY  Buffer;
RETURN_CODE rc;


// 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);
```

PlxPciPhysicalMemoryUnmap

Syntax:

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

PLX Chip Support:

All

Description:

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

Parameters:

hDevice
Handle of an open PCI device

pMemoryInfo
A pointer to a PCI_MEMORY structure which contains the buffer information

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
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 *PlxPciPhysicalMemoryMap*
ApiInvalidData	The buffer information is invalid or it was not allocated with *PlxPciPhysicalMemoryAllocate*

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;
PCI_MEMORY  Buffer;
RETURN_CODE rc;


// 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
}
```

PlxIntrWait

Syntax:

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

Note: *This function is only supported in a Linux environment.*

PLX Chip Support:

All

Description:

This function is used by applications that need to wait for a specific interrupt(s) to occur. The function will wait for the event(s) to occur which was registered with *PlxIntrAttach*.

Parameters:

hDevice
Handle of an open PCI device

hEvent
The handle to a wait object which was successfully returned from *PlxIntrAttach*

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 ocurred
ApiInvalidHandle	The function was passed an invalid device handle
ApiWaitTimeout	Reached timeout waiting for event
ApiWaitCanceled	Wait event was cancelled or not found in list of registered events

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.

Usage:

```
HANDLE       hDevice;
HANDLE      *pEvent;
PLX_INTR     IntSources;
RETURN_CODE  rc;


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

// Allocate memory for Event Handle pointer
pEvent = (HANDLE *)malloc(sizeof(HANDLE);

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

rc =
    PlxIntrAttach(
        hDevice,
        IntSources,
        pEvent
        );

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

// Wait for the interrupt
rc =
    PlxIntrWait(
        hdevice,
        *pEvent,
        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;
}
```

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 addition of 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 these functions, PLX cannot guarantee their functionality in future SDK releases. Future versions of the OS may trap PCI I/O port accesses. As a result, PLX does not support these functions. They are 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

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 addition of 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 these functions, PLX cannot guarantee their functionality in future SDK releases. Future versions of the OS may trap PCI I/O port accesses. As a result, PLX does not support these functions. They are 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
    );
```

PlxSerialEepromReadByOffset

Syntax:

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

PLX Chip Support:

All

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
}
```

PlxSerialEepromWriteByOffset

Syntax:

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

PLX Chip Support:

All

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
}
```

5 PLXMon User's Manual Updates

The PLXMon application is constantly updated to fix bugs, increase performance, and enhanced usability. Along with bug fixes, the latest enhancements to PLXMon involve the register dialogs and do not require supporting documentation since the interface is intuitive.

5.1 Secondary FLASH Support

The PLXMon application includes some enhancements to support new PLX hardware. This includes support for the 9056 and 9656 PLX chips and support for secondary FLASH devices on boards containing PLX chips.

To support a secondary FLASH, two modifications were made to existing PLXMon dialog boxes. These are the board properties, which informs PLXMon of the FLASH device, and the download dialog, which allows for selection of the secondary FLASH. Both of these are shown below: