PLX SDK User Manual

Version 6.40

December 2010

PLX SOFTWARE LICENSE AGREEMENT

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

This PLX Software License agreement is a legal agreement between you and PLX Technology, Inc. for the PLX Software, which is provided on the enclosed PLX CD-ROM. PLX Technology owns this PLX Software. The PLX Software is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties, and is licensed, not sold.

PLX Software License Agreement

GENERAL

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

WARRANTY

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

Table of Contents

1 General Information

1.1 About this Manual

This manual provides information about the functionality of the PLX SDK. The SDK may be used in conjunction with any PLX Rapid Development Kit (RDK) or any custom design containing a PLX 8000, 9000, or 6000 series chip. Users should consult this manual for PLX SDK installation and general information about the design architecture.

1.2 PLX SDK Features

The SDK contains software for Windows & Linux host environments where the PLX chip is accessed across the PCI/PCIe bus. This package is provided for debug phase of hardware development and also for development of custom applications:

- Windows drivers & API with source code
- Linux drivers & API with source code supporting kernel 2.4 & 2.6
- ***PLX Device Editor*** (PDE) debug utility for all PCI Express devices
- ***PLXMon*** debug utility is to support all PLX 6000 & 9000 series devices.
- Sample applications

1.3 Terminology

- References to Visual C/C++ or Visual C++ refer to Microsoft Visual C/C++ 6.0.
- Win32 references are used throughout this manual to mean any application that is compatible with the Windows environment.
- References to PCI Express may be denoted as either PCIe or PEX.
- References to Non-Transparency may be denoted as NT.
- References to Application Programming Interface may be denoted as API.

1.4 Customer Support

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

- PLX chip used
- PLX SDK version
- Host Operating System and version
- Model number of the PLX RDK (if any)
- Description of your intended design
- Detailed description of your problem
- Steps to recreate the problem.

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

Address:	PLX Technology, Inc. Attn. Technical Support 870 W Maude Avenue Sunnyvale, CA 94085
Phone:	408-774-9060
Fax:	408-774-2169
Web:	http://www.plxtech.com/support

2 Getting Started

2.1 Development Tools

Various tools were used to build the software included in the PLX SDK. There are many compatible alternative tools available for the various build environments. Customers are free to use their own preferred sets of compatible development tools; however, PLX has only verified the tools listed below and, as a result, cannot support tools not listed here. The development tools used to develop the PLX SDK components include:

Windows Applications and API DLL:
Microsoft Visual C/C++ 6.0, Service Pack 6

Windows Driver Model (WDM) Device Drivers
Microsoft Windows Device Driver Kit (DDK) or Windows Driver Kit (WDK). 2003 Server DDK or higher is required to build 64-bit versions of PLX drivers.

Linux Applications and API Library:
Standard Linux distribution, such as RedHat or Fedora, using GCC.

Linux Device Driver:
Standard Linux distribution, such as RedHat or Fedora with kernel source/development RPM installed

2.2 PLX SDK Version Compatibility

When using the PLX SDK, it is important that all components are of the same version, as follows:

- In Windows & Linux, the PLX device drivers (e.g. *.sys* files) and the PLX API library (e.g. *PlxApi.dll*) versions must match. In other words, loading a driver built with SDK 5.0 and running an application, which calls the API library from version 4.40, will result in erratic behavior.
- When building applications, it is important to use the C header files included in the installed PLX SDK version. Applications built with older SDK versions must be re-built. In some case, there may be a porting effort when upgrading to a newer SDK due to API changes.

2.3 PLX SDK Installation in Microsoft Windows

Before installing the SDK, any previously installed PLX SDK versions should be removed. Installation of multiple SDK versions may result in erratic behavior due to file conflicts. Refer to section 2.3.2 for more details.

To install the PLX SDK Software package, simply run the SDK installation package and follow the prompts.

Note: For proper Windows installation, a user with "Administrator" rights must install the SDK in order to install drivers.

2.4 PLX SDK Removal

Prior to installation of a new version of the PLX SDK, any previously installed versions should be uninstalled. Many files change between SDK releases and since these files are used for development purposes, they may be incompatible with a previous release. To remove a PLX SDK package, including device drivers, complete the following:

1. Close any open applications
2. Open the Windows Control Panel
3. Select Add/Remove Programs icon in the Control Panel window
4. Choose the PLX SDK package from the item list
5. Click the Add/Remove... button

Note*: For proper removal, a user with "Administrator" rights must remove the PLX SDK.*

***Warning**: If any files have been modified in the original PLX SDK install directory, such as C source code files, the uninstaller may delete them. Please be careful before uninstalling an SDK package. The SDK directory can first be copied (not moved) to another safe location before removal.*

2.5 Installation of PLX Device Drivers in Windows

During SDK installation, the installation package will automatically create the necessary registry entries and copy any files needed to load PLX device drivers.

2.5.1 PLX Plug and Play Device Driver Installation

The PLX Windows device drivers conform to the Microsoft Windows Driver Model (WDM). These drivers support Plug 'n' Play (PnP) and Power Management.

Since Windows is a Plug 'n' Play (PnP) Operating Systems, the SDK installation package does not automatically assign device drivers for PLX devices. The Windows PnP Manager is responsible for detecting devices and prompting the user for the correct driver. To assign a driver for a device, Windows refers to an INF file. The INF file provides instructions for Windows as to which driver files to install and which registry entries to insert.

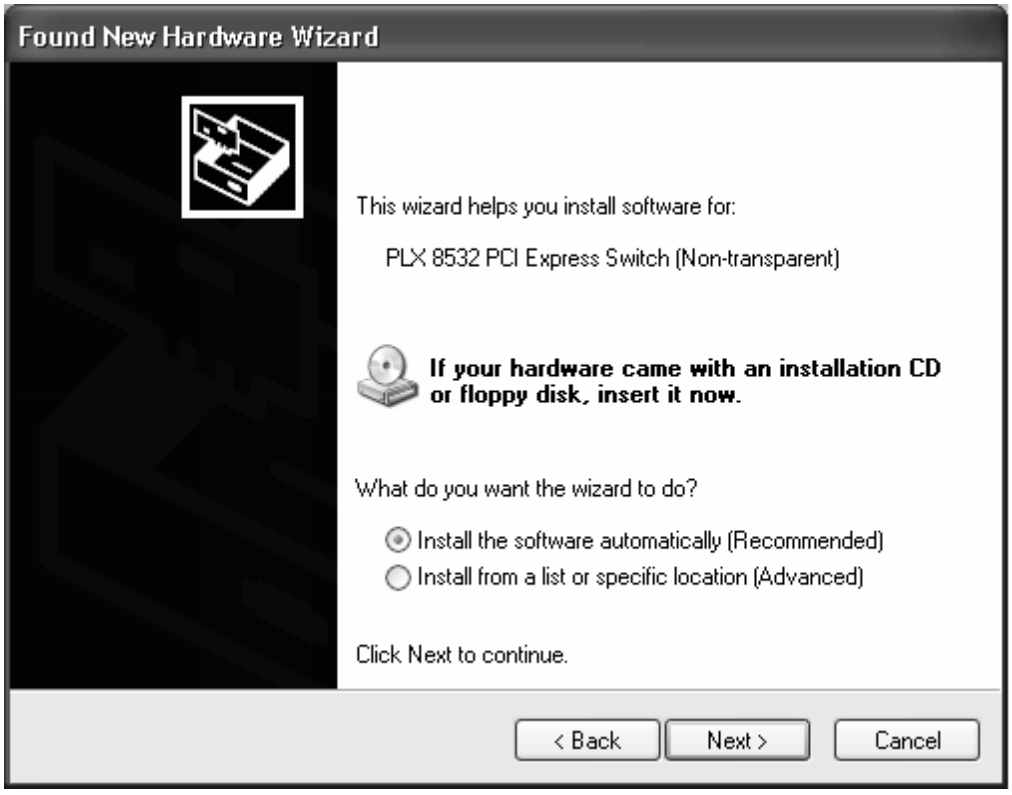
To install a driver for a board containing a PLX device in PnP Windows, complete the following steps:

1. After installing the PLX SDK successfully, shut down the computer.
2. Insert the PLX RDK board or your custom board with a PLX device into a free PCI or PCIe slot.
3. Reboot the computer. Windows should first detect the new hardware device with a "New Hardware Found" message box. Acknowledge this message box.
4. Windows then displays the "Found New Hardware" Wizard, which will search for a suitable driver.

2.5.1.1 PLX Device Driver Installation

- Once the Found New Hardware Wizard starts, the following dialog is displayed: Select ***No, not this time***.

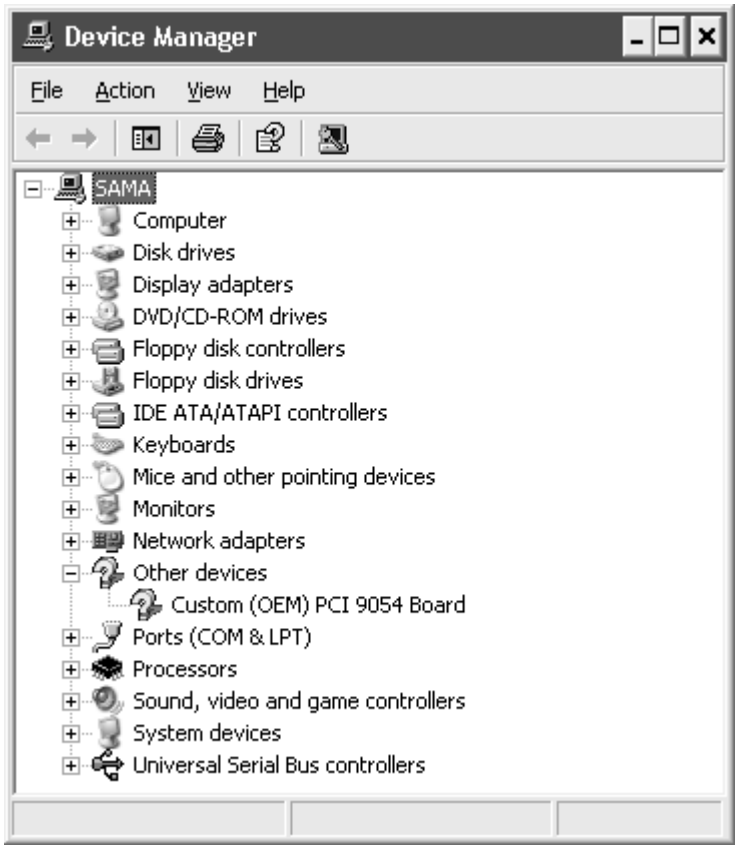
- The Wizard will now attempt to find the .INF file. By default, PLX includes the PLX INF file in <*Sdk_Install_Dir*>*Windows**Drivers*, but it also places a copy in the Windows INF folder. The wizard should be able to automatically locate the correct INF file. Select ***Install the software automatically*** option.



- Windows will then scan through INF files to find a matching device driver. Since PLX drivers are not digitally signed, Windows will prompt with the following dialog. Click ***Continue Anyway***.

- When the following dialog is displayed, the device driver installation is complete. Click the Finish button.

- If the device appears under ***Other devices***, the installation was successful. Applications that use the PLX API, such as PLXMon or the PDE, may now be used to access the device.



***Note**: If the Device/Vendor ID of the board is changed or the board is physically moved to a different PCI slot, Windows will recognize it as a completely new device and the process must be repeated.*

2.5.1.2 Modifying the PLX INF File for Use with Custom Device/Vendor IDs

When a new device is plugged into a system running Windows, the Windows Plug 'n' Play Manager will prompt the user for driver files. Windows determines which files to install through information in an INF file. PLX already provides an INF file (***PlxSdk.inf***), which contains setup information for all PLX RDKs and all PLX parts with a default ID. The INF may be found in ***<Sdk_Dir>\Windows\Driver***, but the install package also installs a copy under ***<Windows_Dir>\Inf***.

The recommended method for installing a device where the ID has been changed is to open the PLX INF file and add an entry for the device with a custom ID. The procedure for this is documented inside the INF file itself, which is a simple text file. Open the INF in a text editor, such as Notepad, and follow the instructions to add an entry for the custom ID and then re-install the device. Windows will then automatically detect the device and install the necessary driver files.

2.5.2 PLX PCI/PCIe Service Driver

The PLX Service driver (PlxSvc) is installed automatically by the SDK installation package but may also be installed manually. There are various methods to install and control the PLX Service driver, each documented in the following sections.

2.5.2.1 Install Using Service Control Manager (SCM) API

An external Windows utility may be written to install/remove and control the PLX Service driver. This utilizes the Microsoft Service functions, such as CreateService and OpenSCManager. The PLX SDK installation package and PLX Driver Options Wizard use this method to install and control the PLX Service Driver. Refer to the Microsoft on-line documentation for additional details.

2.5.2.2 Install Using Windows "sc.exe" Utility

Most versions of Windows include the utility "**SC**" to access the Service Control database. This may be used to easily perform operations on services, including add/remove and start/stop. Type "sc" in a Command Prompt window to see complete usage. Refer to Figure 2-1 for an example of basic service control functions.

To install the PLX Service Driver, perform the following:

1. Copy the correct version of PlxSvc.sys to the Windows System32\Drivers folder.
2. Issue the following command in a DOS prompt or batch file:

   sc create PlxSvc binPath= System32\Drivers\PlxSvc.sys type= kernel start= auto error= ignore DisplayName= "PLX PCI/PCIe Service Driver"

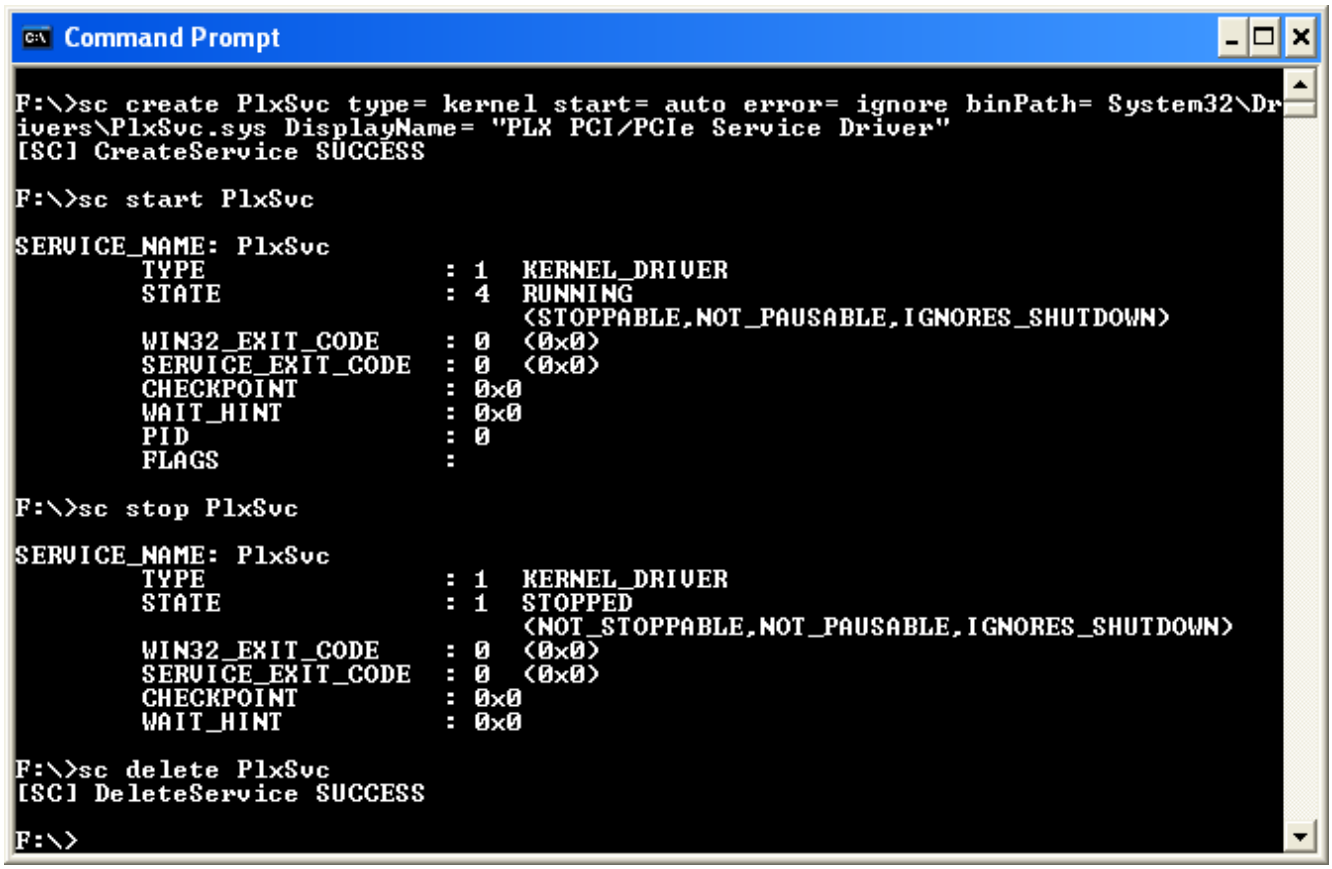


Figure 2-1: Sample 'SC' Commands

2.5.2.3 Install Manually via Registry and Reboot

To perform a manual installation, follow the steps below:

- **Add the required driver registry entries**
  Double-click the PLX Service registry file (<*Sdk_Install_Dir*>*Windows**PlxSvc.reg*) to install the required registry entries. Double-clicking the file will automatically launch *RegEdit* and add the necessary entries.
- **Copy the PLX Service driver to Windows**
  Copy the correct version (32-bit or 64-bit) of the file *PlxSvc.sys* to <*Win_Dir*>*System32**Drivers*. *PlxSvc.sys* may be found in <*Sdk_Install_Dir*>*Windows**Driver**Source.PlxSvc*.
- **Copy the PLX API library to Windows**
  Copy the PLX API DLL (e.g. *PlxApi640.dll*) to the Windows <*Win_Dir*>*System32* folder. This file is located in <*Sdk_Install_Dir*>*Windows**PlxApi*.
- **Restart the system**

2.5.2.4 Starting and Stopping the PLX Service Driver

Since the PLX PCI Service runs as a background task, it may be stopped and started dynamically. The steps below demonstrate how to control the service. Additionally, the ***PLX Driver Options Wizard*** may be used to start and stop the driver.

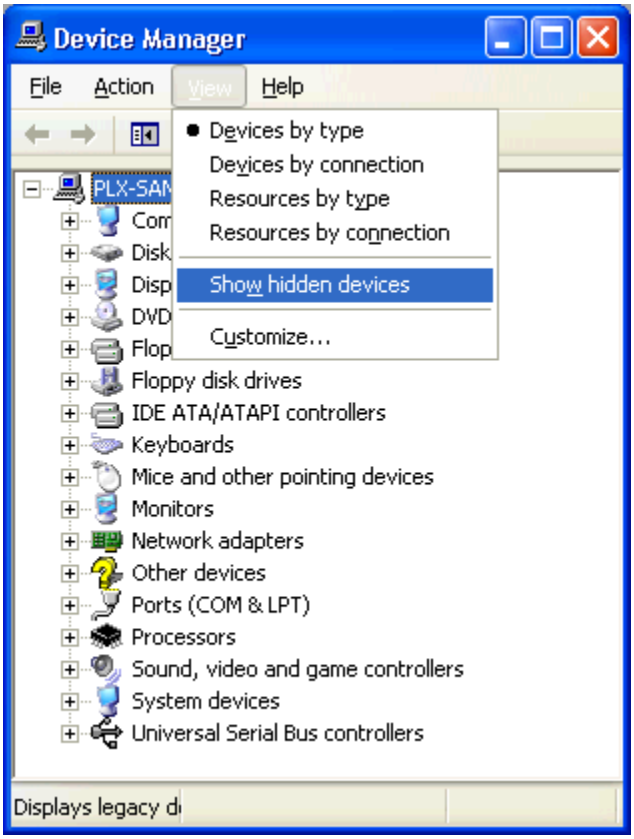
2.5.2.4.1 Use command-line utilities

- Use Microsoft 'net' utility bundled with Windows:
  net start PlxSvc
  net stop PlxSvc

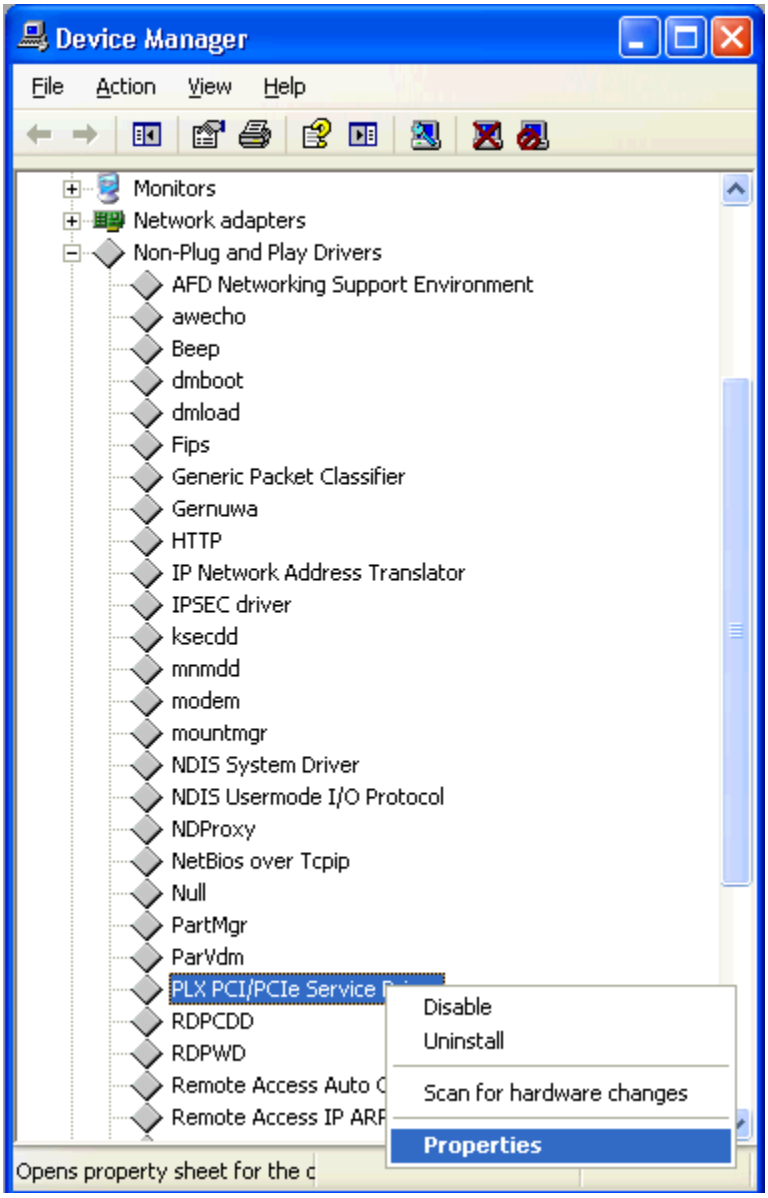
- Use Microsoft “sc” utility bundled with Windows:
  sc start PlxSvc
  sc stop PlxSvc

2.5.2.4.2 Use Device Manager

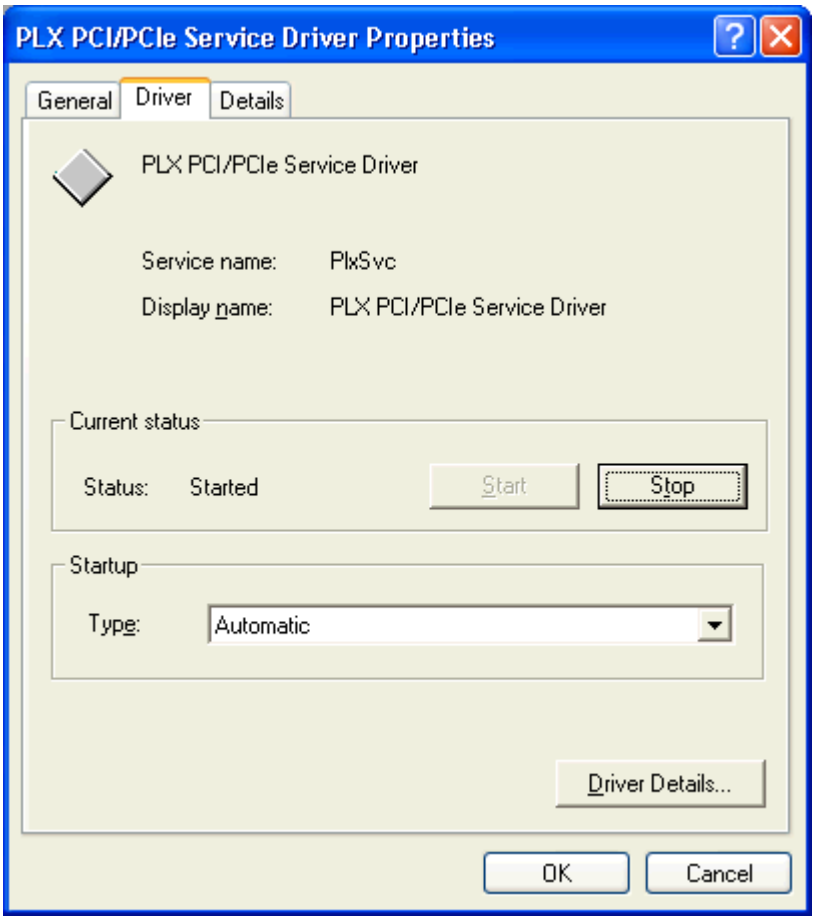
- Open the Device Manager (*My Computer* Properties, *Hardware* tab) and display the hidden devices as shown below.



- Under *Non-Plug and Play Drivers*, find the ***PLX PCI/PCIe Service Driver*** entry and double-click it.



- The following dialog will appear. The *Start* and *Stop* buttons control loading and unloading of the driver, respectfully.



2.5.3 Modifying PLX Driver Options in the Registry

All Windows drivers have entries in the Registry, which are required by the OS. Additionally, there may be driver-specific entries, which can be used to customize driver behavior. Some features of PLX drivers are customizable through registry settings and are documented below. The registry entry is located in the path specified below. Figure 2-2 demonstrates a typical entry.

HKLM\System\CurrentControlSet\Services\<DriverName>

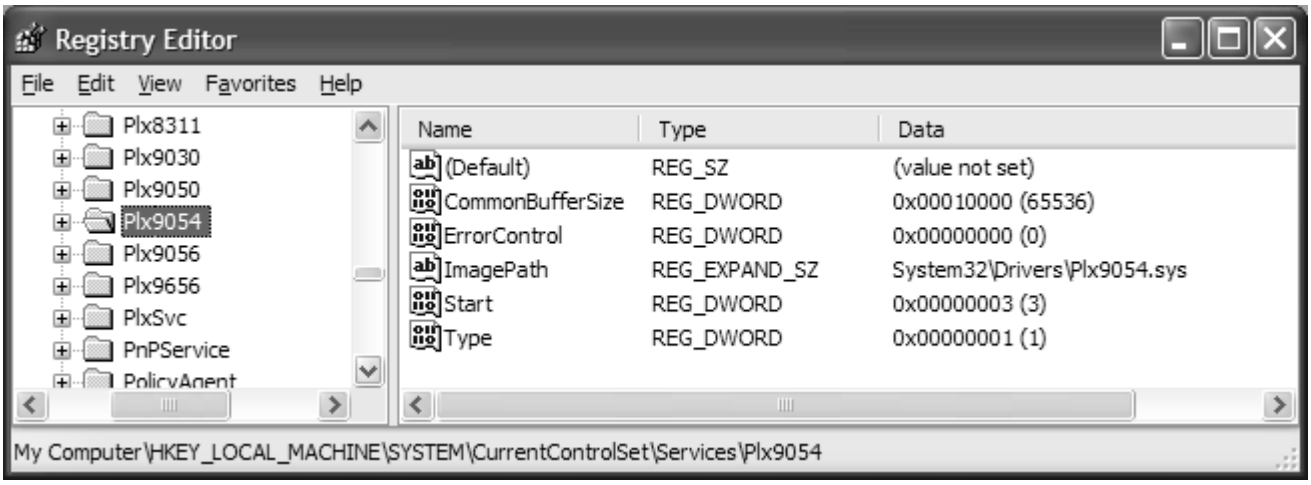


Figure 2-2 PLX Device Driver Registry Information

The registry entries are described in detail below. ***Note:*** *Only advanced users with administrative rights should modify entries in the registry. Please refer to Microsoft's documentation on modifying the registry.*

Windows required entries:

- *ErrorControl*
  Required by the operating system and should not be modified.
- *Start*
  Required by the operating system and should not be modified.
- *Type*
  Required by the operating system and should not be modified.

PLX-specific entries:

- CommonBufferSize
  This value sets the size of the Common buffer, which the driver attempts to allocate for use by all applications. This buffer is a non-paged contiguous buffer, so it can be used for DMA transfers. The default value is set to 64KB. Users may increase this value if a larger buffer size is needed.

  Note: *Changing this entry does NOT guarantee allocation of a larger buffer. The device driver makes a request to the operating system for a buffer with the size indicated by this registry entry. If the request fails, however, usually due to unavailable system resources, the driver will decrement the size and resubmit the request until the buffer allocation succeeds. The API call PlxPciCommonBufferProperties() can be used to determine the common buffer information.*

2.5.3.1 PLX Driver Options Wizard

The PLX SDK includes the *PLX Driver Options Wizard* application to manage all PLX driver settings. Using the wizard avoids the need to manually modify the registry. The wizard may be used in all supported versions of Windows. Details about each configuration option are displayed at the bottom whenever the item is selected.

After launching the wizard, select the desired driver and modify the options as needed. The updated settings will take effect when the driver is reloaded, either manually or after a system reboot.

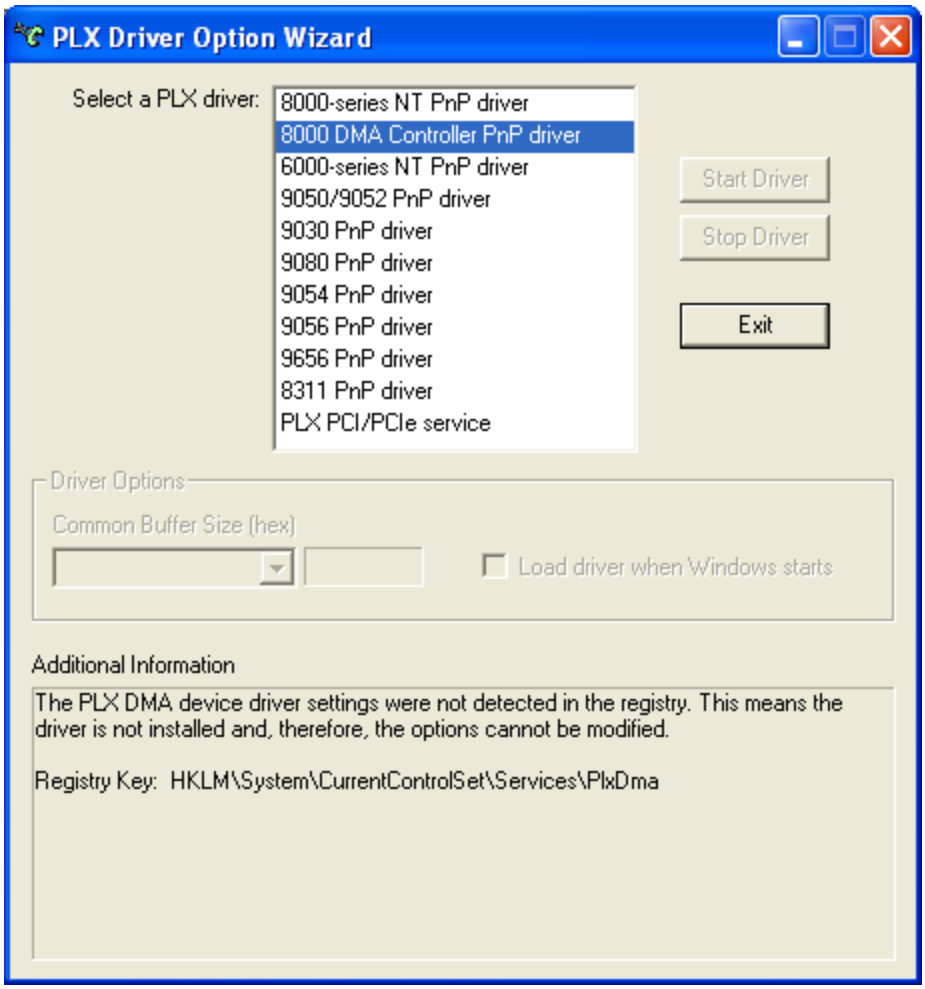


Figure 2-3 PLX Driver Options Wizard

2.6 Installation of PLX Device Drivers in Linux

The PLX SDK contains support for Linux environments. Documentation for the Linux support is not included this manual; however, much of the Windows host-side architecture applies to the Linux Host-side support as-well. Please refer to the PLX Linux Release Notes in the *<Sdk_Install_Dir>\Documentation* folder for using the PLX SDK in Linux. The PLX Linux TAR package is located in *<Sdk_Install_Dir>\Linux_Host*.

2.7 Distribution of PLX Software

2.7.1 License Agreement

For OEM customers, who have written applications with PLX software and intend to ship it with their product, please refer to the PLX Software Distribution License Agreement in the PLX SDK Release Notes. The License Agreement is not reprinted in this manual. The agreement specifies which SDK components may be redistributed to end users.

3 PLX Host-side Software

This section describes the PCI Host software components provided in the PCI SDK, which applies to Windows and Linux.

3.1 SDK Directory Structure

Figure 3-2 shows the PLX SDK directory and top level sub-folders.

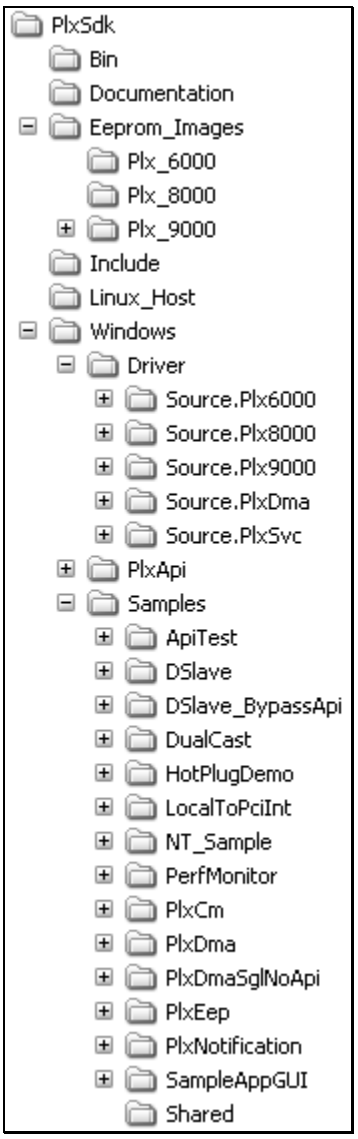


Figure 3-1. PLX SDK Directory Organization

- ***Bin***
  Contains binary executables.
- ***Documentation***
  Contains the User's Manual, readme files and other SDK documentation.
- ***EEPROM Images***
  Contains sample binary EEPROM files for all PLX devices and RDKs

- ***Include***
  Contains all the common include files used by the drivers and applications in the SDK.
- ***Linux_Host***
  Contains the PLX Linux support package
- ***Windows\Driver***
  PLX Windows drivers source code
- ***Windows\PlxApi***
  Contains the PLX API source code
- ***Windows\Samples***
  Contains sample applications that demonstrate use of the PLX API

3.2 PLX SDK Architecture Overview

The PLX SDK has three main components, the Kernel drivers, User API and User Applications. Figure 3-2 demonstrates the various components and how they fit together. The SDK is provided to handle most of the low-level functionality so users can concentrate on building their applications.

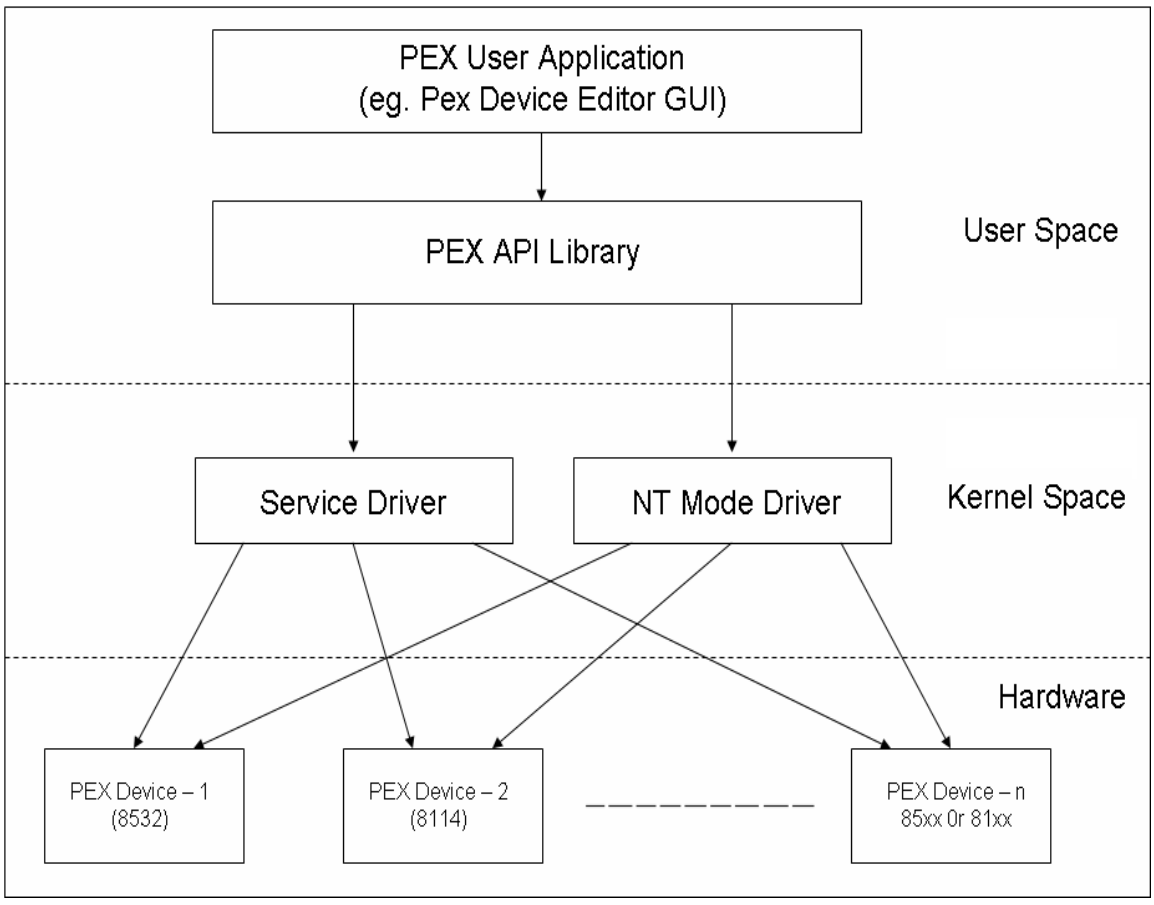


Figure 3-2 PLX SDK Software Architecture

3.3 PLX API Library

The PLX API library is provided to communicate with the PLX device drivers. When an API function is called by an application, the API library handles the call and translates it to an I/O control message and sends it to the driver. Once the driver completes the request, control returns to the API and then back to the calling application.

The PLX API consists of a library of functions, from which multiple PLX chip-based PCI boards can be accessed and used. The API covers all features of all PLX chips, such as DMA access, direct data transfers, and interrupt handling.

The PLX API libraries in the Windows environment file are implemented as Dynamically Linked Libraries (DLL). Applications linked with these libraries will attempt to load the DLL when started; therefore, the DLLs must be found somewhere in the system path. DLLs are typically placed in the Windows system directory.

The PLX API library in the Linux environment file is implemented as a statically linked library, rather than dynamically loaded. Applications will link with the API library during the build process and will, therefore, contain API library code in the executable.

3.4 Device Drivers

The PLX SDK contains two types of Windows device drivers. The first type is a Windows service driver. The service driver is used to access any PCI device in the system and also supports EEPROM access to PLX devices running in Transparent Mode. The other category of device driver is a standard Plug 'n' Play device driver. This driver is typically used for PLX devices running in Non-Transparent mode and also for all PLX 9000 devices.

A device driver is necessary for the PLX SDK software to access PLX PCI devices. Applications, such as PLXMon, cannot access PCI devices without a device driver installed. The SDK includes drivers for all supported PLX PCI chips.

The PLX device drivers contain the API implementation for the PLX chip they support and the basic functionality required by all device drivers for the OS environment. The device driver accesses the PLX chip across the PCI bus by using OS system calls. The driver is also responsible for handling PCI interrupts from the PLX chip.

Each PLX chip type has an associated driver. Device drivers are not associated with a specific board, but are generic in design to be used for any board containing the specified PLX chip. A single driver is responsible for all devices in the system containing the PLX chip the driver was written for. Each device driver communicates with the PLX API on a one-to-one basis; there is no driver-to-driver communication.

3.5 PLX API and Multi-threading

Programming in a multi-tasking environment requires understanding of many issues that do not exist in a single-threaded environment. These issues can be especially complex when they involve hardware device drivers. For those customers who intend to use PLX software in a multi-tasking environment and use it with multiple simultaneous applications, some additional work and caution may be required.

The PLX API libraries and drivers do not enforce synchronization between concurrent accesses to PLX chips. In other words, the PLX drivers do not lock all resources of the PLX chip while they are in use. Only the DMA channels are treated as a shared resource and may only be opened by one thread at a time. Each channel is independent so each can be opened by different processes.

The reasons PLX drivers do not enforce synchronization on the whole:

- Each "feature" of the chip would have to be treated as a shared resource. This includes each BAR space, each DMA channel, & the shared common buffer.
- The PLX API allows applications to map registers and BAR spaces directly into an application's virtual space for performance reasons. Once that happens, any accesses to the registers or space completely bypass the PLX driver so synchronization cannot be enforced. In other words, there's nothing stopping another process from manually writing to the DMA registers even if another process "owns" the channel.
- Synchronizing accesses to BAR spaces is not feasible. BAR space memory read/write is generally slow in relative terms. Reads are typically only 2-4MB/s. If one application wishes to read 8MB from a particular local bus location, the BAR resource must be locked for 2 seconds, which is very poor programming practice. Locking is required because the remap register, for example, must be set to access the desired local bus region. If another thread wishes to access another local bus region, it may need to adjust the remap window, which will corrupt the 1st thread.

It is left to application designers to deal with overall synchronization if it's needed. Typically this can be done by assigning mutually exclusive resources or using Inter-Process Communications (IPCs) provided by the OS to coordinate accesses to the PLX chip. Although there are numerous techniques to deal with this issue, it is left to the judgment of end-users to determine their specific requirements.

3.5.1 PLX Device Driver Directory Structure

The PLX drivers are designed to take advantage of common code; therefore many files are shared between all PLX drivers. The following figure depicts the Windows device driver directory structure as found in the PLX SDK installation.

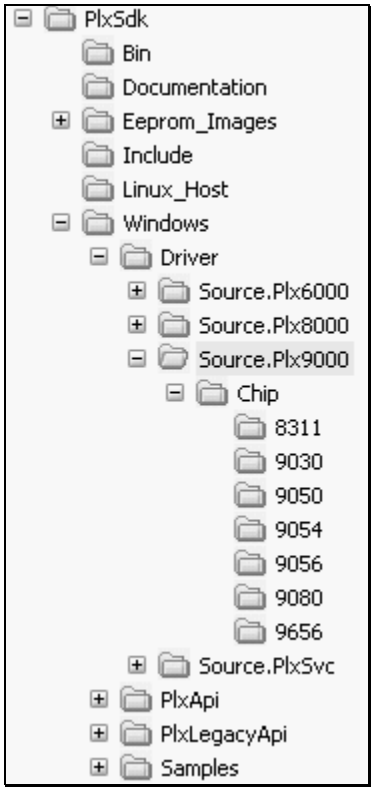


Figure 3-3 PLX Driver Directory Structure

The driver directories are described below:

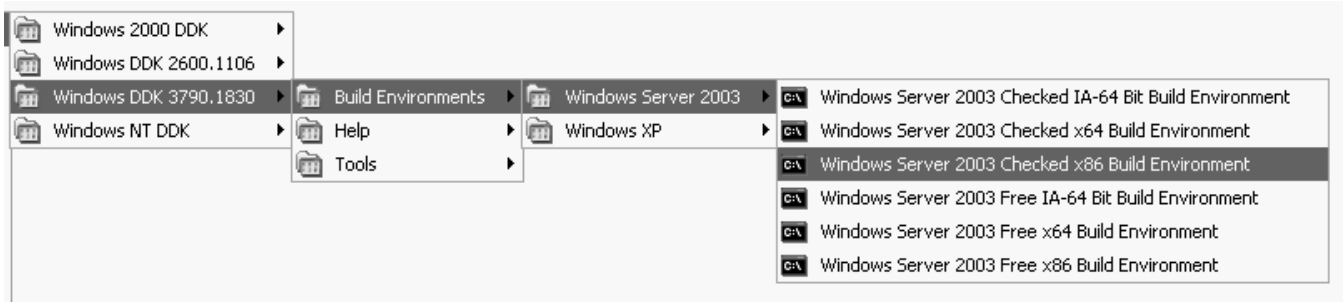
- *Driver\Source.Plx6000*
  Contains source code for the PLX 6000 Non-transparent mode driver.
- *Driver\Source.Plx8000*
  Contains source code for the PLX 8000 Non-transparent mode driver.
- *Driver\Source.Plx9000*
  Contains source code common to all PLX 9000 drivers.
- *Driver\Source.Plx9000\Chip\<ChipType>*
  Contains PLX chip-specific source code used for PLX 9000 drivers.
- *Driver\Source.PlxDma*
  Contains source code for the PLX 8000-series DMA feature.
- *Driver\Source.PlxSvc*
  Contains source code for PLX Service driver.

3.5.2 Building Windows Device Drivers

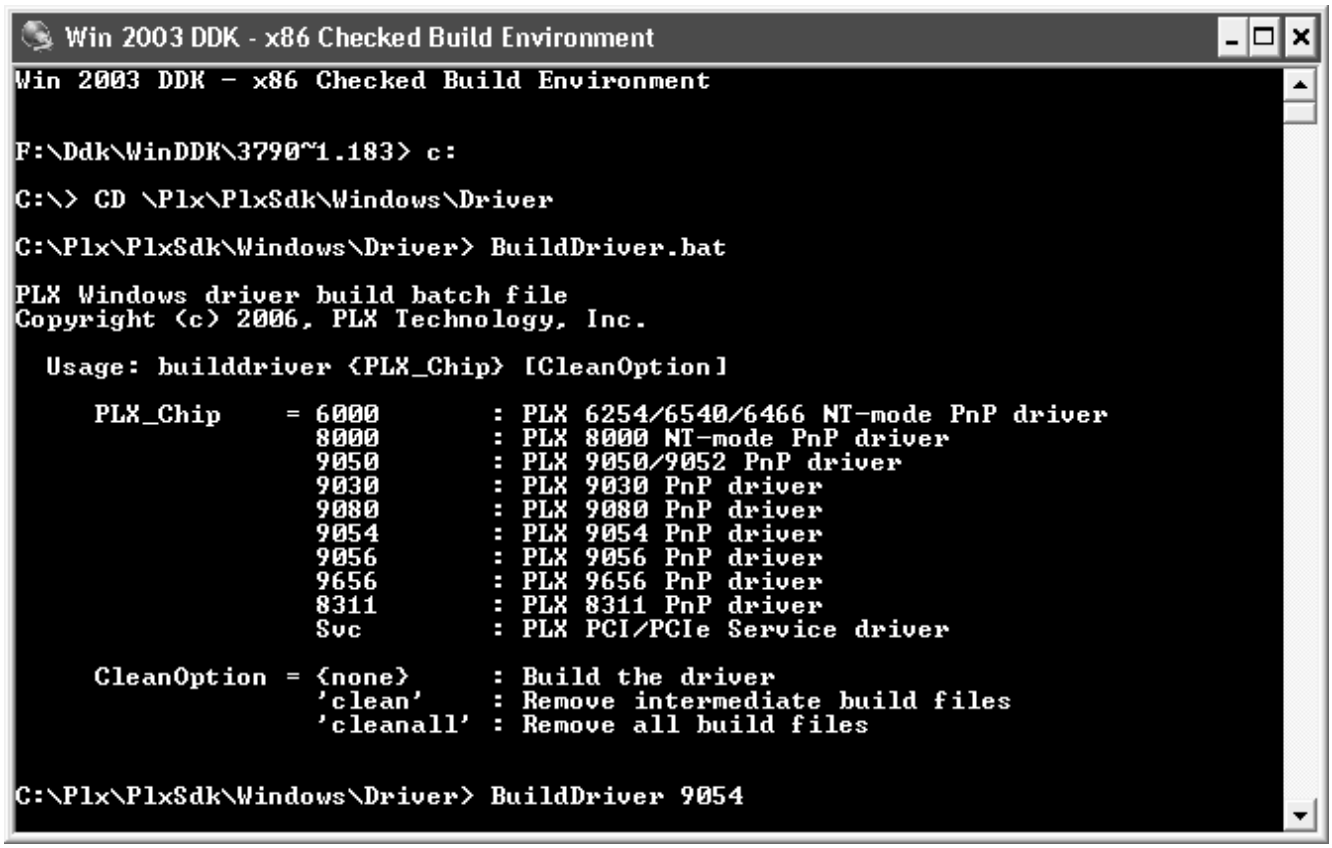
To build a driver, the Windows DDK or WDK must first be installed. Follow the steps below to build the driver. The DDK environment determines the version of the driver built; otherwise, the build process is identical for all environments.

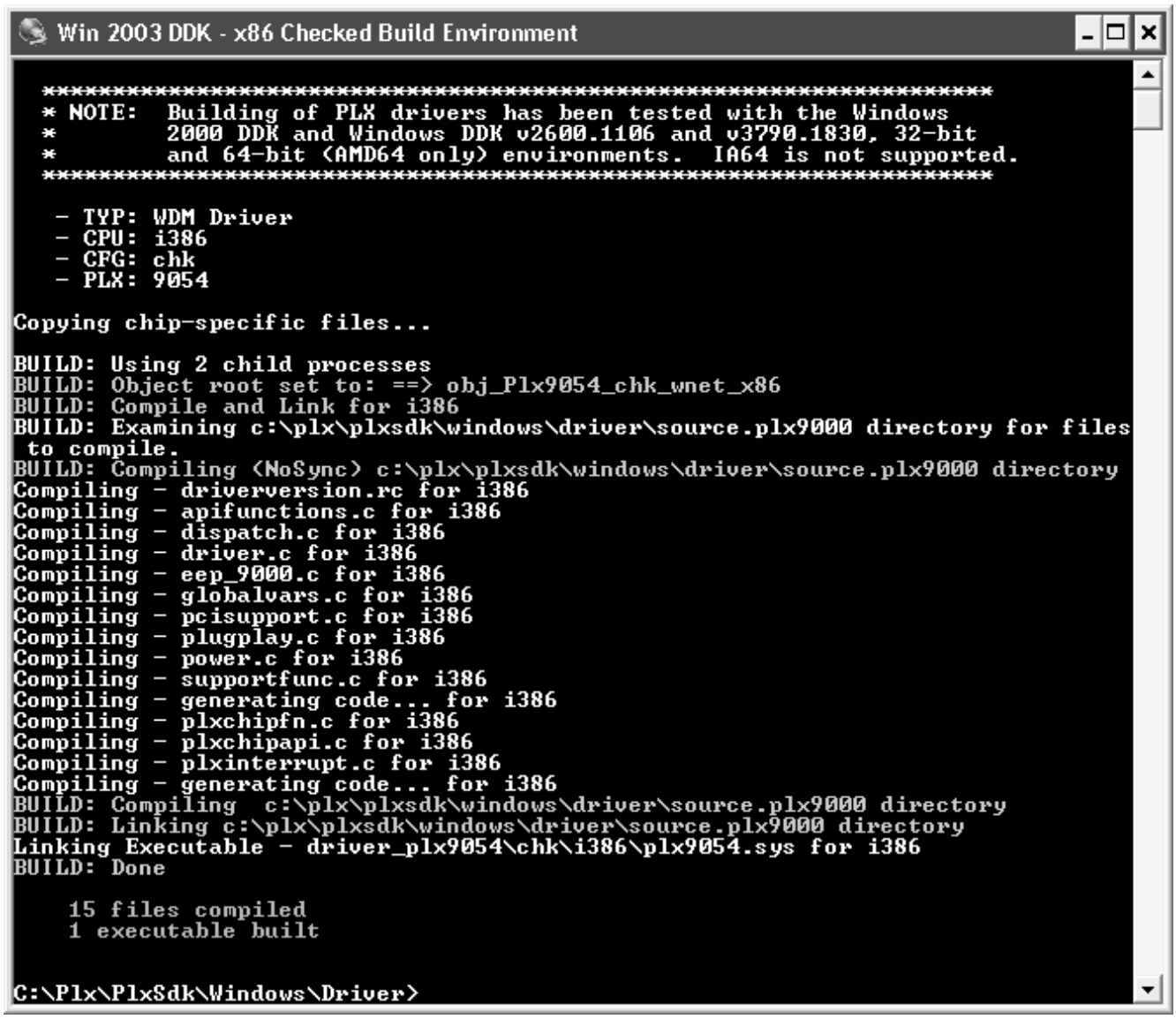
Note: Due to limitations in the **build** utility provided in the Windows DDK, the PLX-supplied batch file, **BuildDriver.bat**, must be used to build a driver. The **build** utility does not easily support compiling of files in a common directory; therefore, it is not used directly to build PLX drivers.

- Select and open the desired DDK environment (icons are installed by the DDK).



- Move to the PLX SDK driver directory. Use the ***BuildDriver*** script to build the drivers. *BuildDriver.bat* will automatically perform the necessary steps to build the desired device driver. Some sample build screenshots are provided below. Once the driver is built, the new driver file may be used in Windows. *Refer to the Windows DDK for additional information on building and debugging drivers.*





3.6 User-mode Applications

User-mode applications use the PLX API library to control any device with a PLX chip. For most situations, a user-mode application using the PLX API is sufficient to perform the desired functionality. PLX drivers are generic in design to minimize the need for driver customization. Typically, drivers are modified to take advantage of specific OEM hardware on a device, or possibly to add functionality, such as additional processing in the Interrupt Service Routine.

This section will explain some techniques for building user-mode applications and use of the API. The following text refers to Microsoft Visual C/C++ 6.0, but customers are free to use any compatible developer tool of preference.

3.6.1 PLX Sample Applications

Several sample applications, located in *<Sdk_Install_Dir>\Windows\Samples*, are included in the PLX SDK. These demonstrate how an application can use the PLX API to perform various functions with PLX devices. The included project files are for Microsoft Visual C/C++ 6.0.

3.6.2 Creating Windows PCI Host Applications

The first step in creating a Windows PCI Host application is to create a Microsoft Project File. A new project file can be created or one of the sample projects can be opened and modified. Typically, a *Win32 Console application* is used to create a project, but any C or C++ project, such as *MFC AppWizard*, is compatible with the PLX API. Figure 3-4 demonstrates the new project dialog.

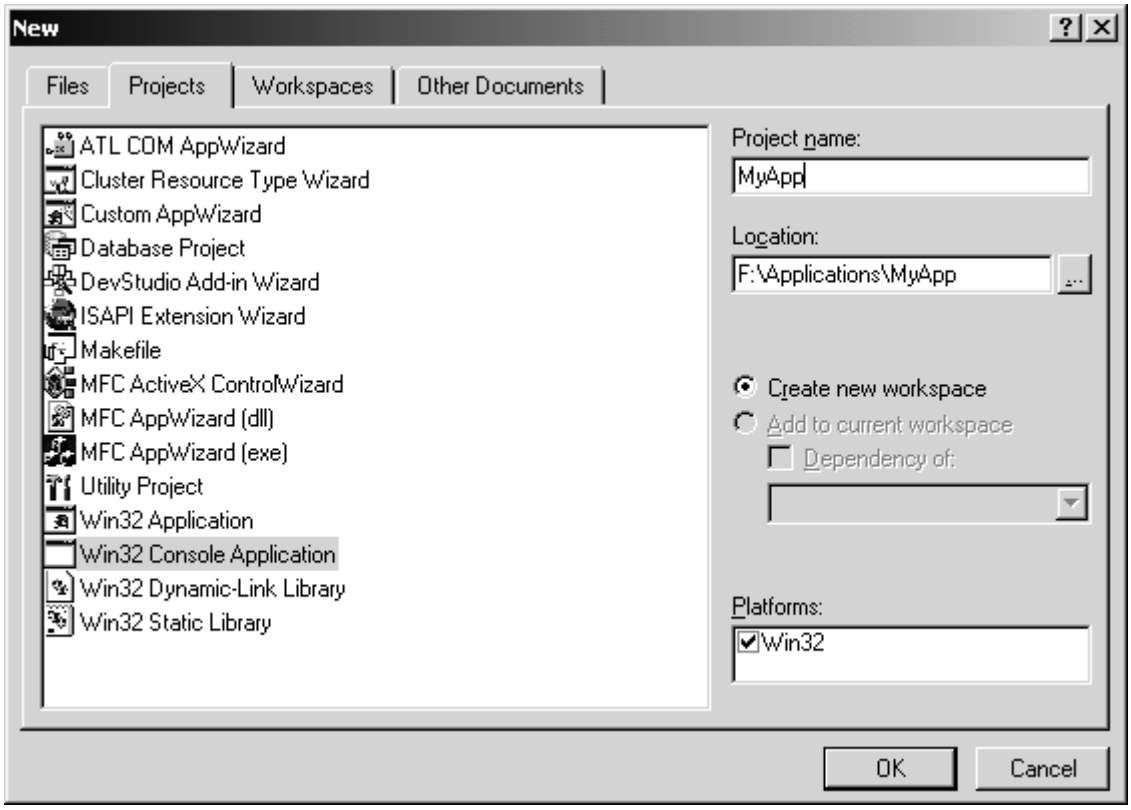


Figure 3-4 Visual C/C++ New Project Dialog

Once the project has been opened, source code can be written and inserted into the project. Before an application can be built successfully, however, the steps below must be completed. Figure 3-6 demonstrates a typical Visual C project that is configured for the PLX API.

- ***Add the PLX SDK Include directory***
  This ensures that the development tools refer to and can find the correct version of the PLX C header files. In Visual C/C++, for example, the directory is specified in the *Options* dialog, as shown in Figure 3-5.

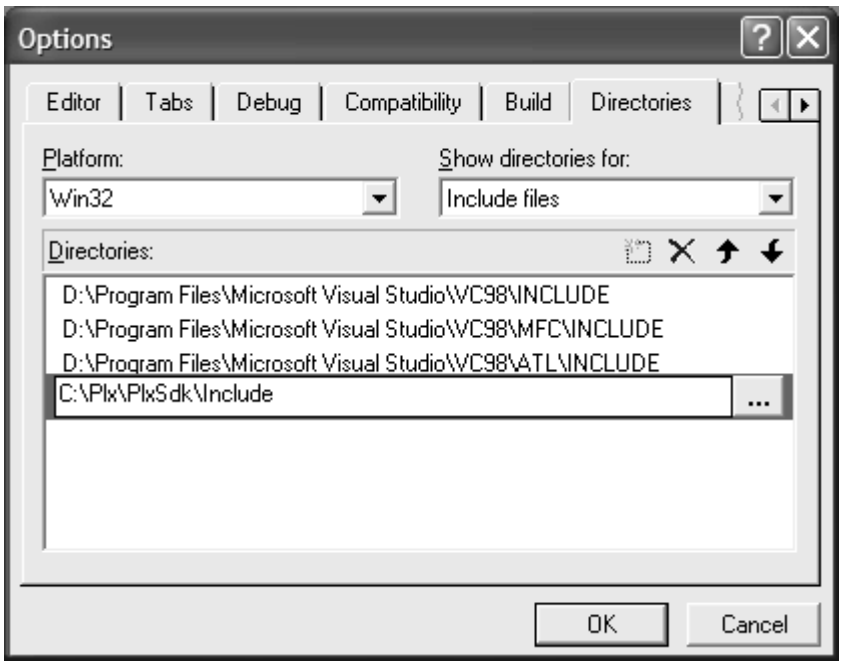


Figure 3-5 Visual C/C++ Include Files Directory

- ***Include "PlxApi.h"***
  This file must be included to provide prototypes for PLX functions and any PLX-specific data types.
- ***Insert "PlxApi.lib" into the Project***
  This library file contains link information for the *PlxApiXXX.dll* file, where <XXX> is the SDK version number, e.g. *PlxApi520.dll*. When the application is launched, the API DLL will automatically be loaded by Windows. The library file is provided in the *<Sdk_Install_Dir>\Winows\PlxApi\Release* directory.

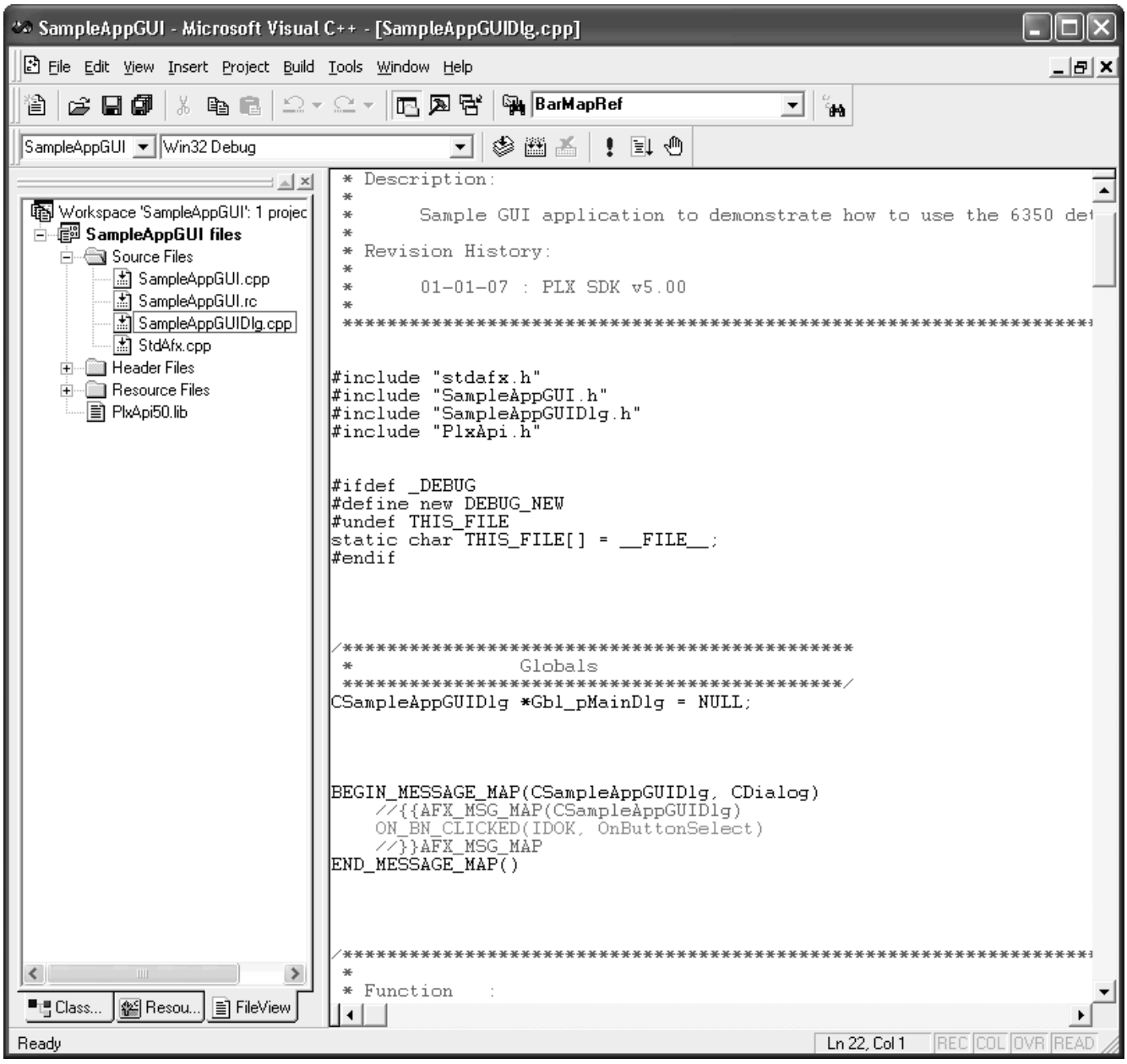


Figure 3-6 Typical Visual C/C++ Project

4 PLX Debug Utilities

4.1 PLX PEX Device Editor (PDE)

The PLX SDK includes the PEX Device Editor for working with PLX PCI Express devices. The following subsections give a high level view of the main Debug and Performance Monitoring features built into the PDE GUI utility.

From a high level the PDE GUI application supports the following features:

- Memory Mapped register access
- Config register access
- Search for registers based on address or description
- Eeprom editing and Programming directly from file
- Find differences between Eeprom and a bin file or 2 Eeprom bin files
- Lane Status Panel with Active lanes, Port types and lane widths
- I2C support to allow access to PLX device features from a different system through a USB to I2C bridge
- Register Table info from Data books accessible from PEX Device Editor GUI
- Save screen data to file, and vice-versa
- Basic Config space access to non-PLX PCI devices
- Online help from the PEX Device Editor GUI.
- PCI Device Capabilities support for Non-PLX devices
- Automated PCI Error Monitoring and Reporting
- Tree View of all PCI devices in the system
- Debug and Performance Monitoring for gen-2 & gen-3 devices:
  - Serdes Eye Width
  - Performance Monitor
  - Packet Generator
  - Probe Mode

The above features allow users to

- Configure the PLX devices to their specific needs.
- Demonstrate all major features with the help of user friendly GUI screens
- Helps in debugging & performance analysis on a live system

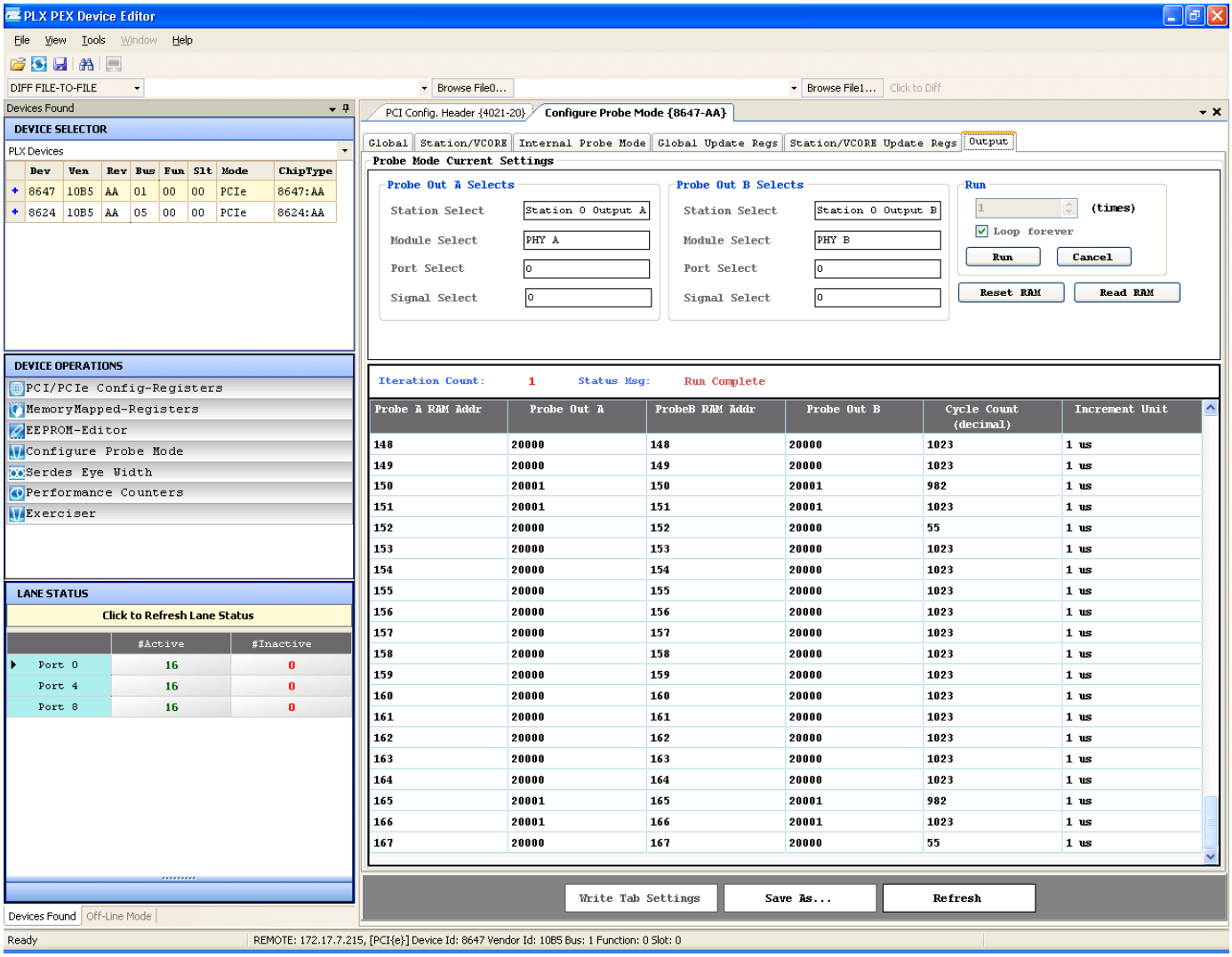
The following subsections will focus on the Debug and Performance monitoring features in detail. For addtional details on how to use the GUI features, please refer to the Help option in the PDE.

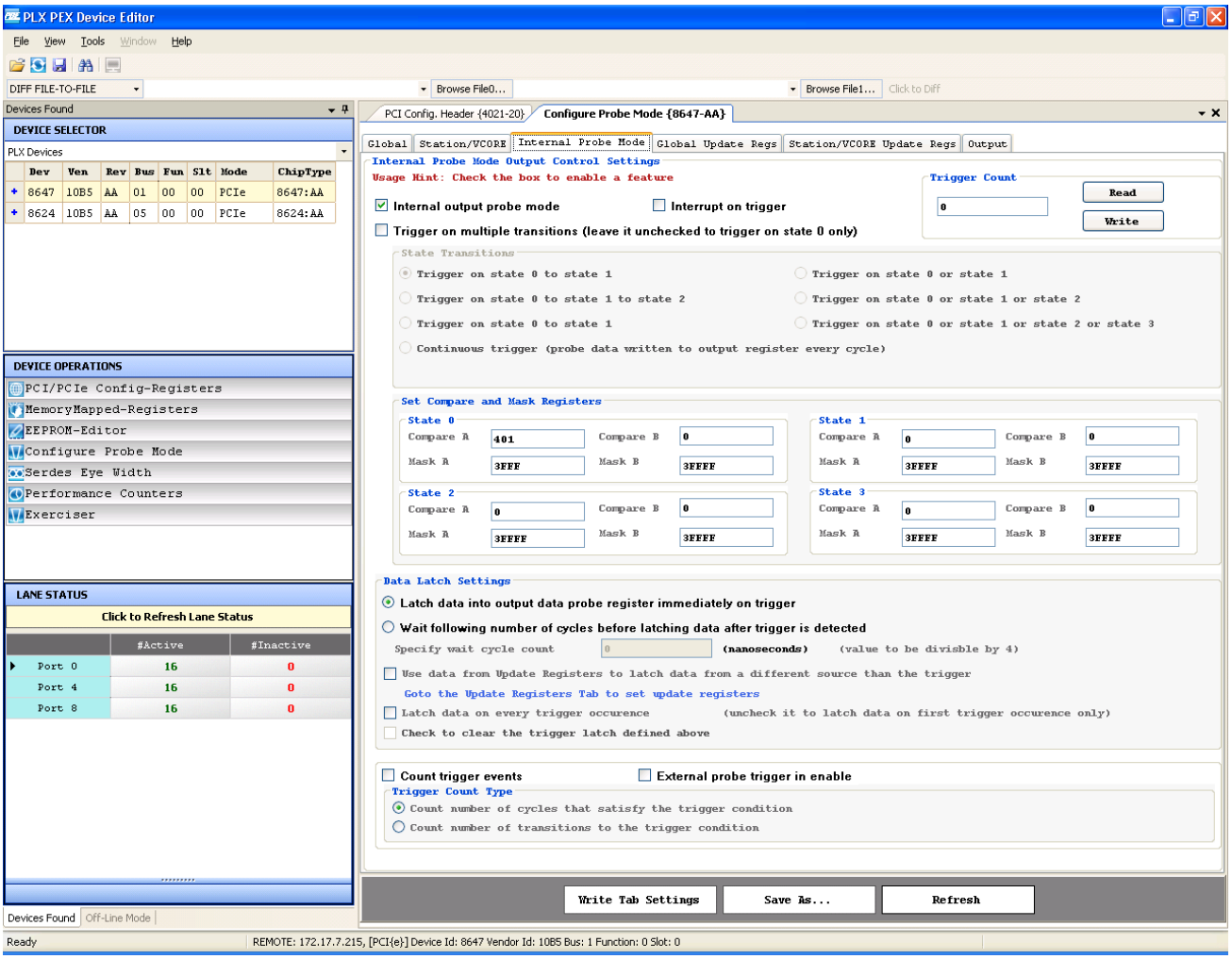
4.1.1 Probe Mode

The Probe Mode feature can do the following:

- Allows users to select all signal combinations that are allowed by Debug Mode, through a user friendly interface
- Supports both External and Internal Modes.
- Supports complex triggering options based on state transitions.

- Captured data can be saved to a file.
- Captured data from file can be displayed in GUI for analysis.





4.1.2 Selecting Signal combinations for probe mode

There are 2 levels of signal selections that can be done with the probe mode.

Level one is at the Module level. There are 2 types of modules. The ones that are present in every Port/Station combination (TIC, TEC, PHY, DLL etc) and the other type is the Core modules (Chime, I2C, EEPROM etc) which are common for the whole chip. Every module can bring out 16 different combinations of Signals to the two outputs Output-A and Output-B. You can select one particular signal combination from each module.

Once you select the appropriate signal combination from all the different modules it will turn out to be a lot more than 36 which is the maximum number of signal that can be brought out at any given time.

In order to narrow the selections down to 18+18 there is the Level two selection screen where you can select only 2 combinations out of all the selections that were done in level one.

4.1.3 External and Internal Modes

Probe mode allows the user to bring out all the selected signals to the probe pins for analysis with the help of a logic analyzer. This is the External Probe Mode.

If the user wants to capture data for a longer period then the internal mode can be used. In the internal mode all the data from the selected signal will be captured in the Debug RAM. The Debug RAM is 5376 bits deep and with 36 bits we should be able to capture to a depth of 150.

The internal mode also allows users to set up various trigger options. These options will be done in more detail once we test the hardware and see what parts of it are working.

4.1.4 Capturing, Saving and displaying data

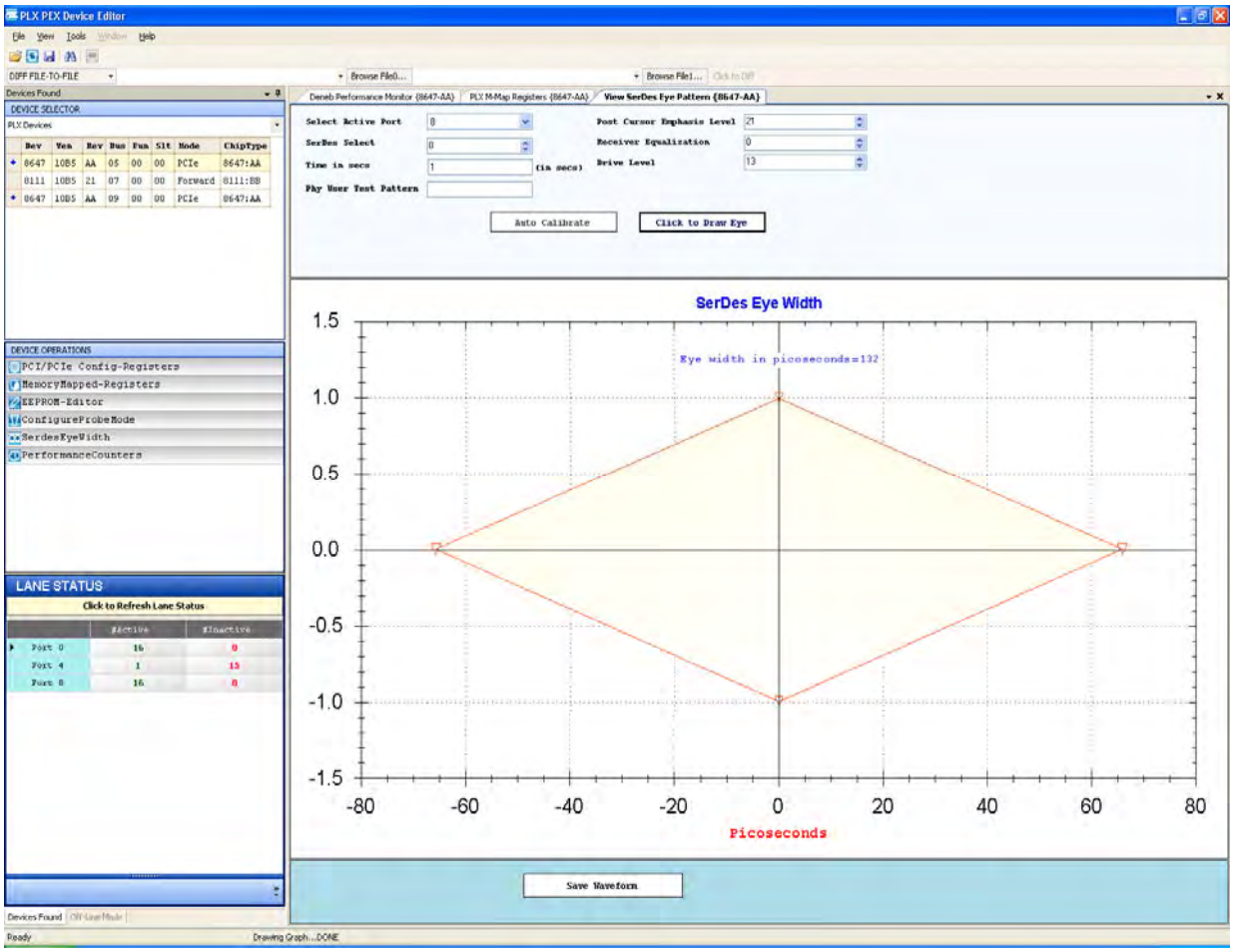
The data that is captured in the Debug RAM can be displayed in the GUI in a tabular form. This can be improved later by displaying it in the shape of a waveform. The captured data can also be saved in a text file. This file could have been sent to us from a customer who captured the data and saved it using the save feature.

Capture will be done at every clock or based on other trigger options selected.

4.1.5 Serdes Eye Width

The purpose of this feature is to allow users to tweak certain parameters of the PLX chip and get the best Serdes Eye width. This can be done at a lane level of every station.

4.1.5.1 Serdes Eye for PLX Gen2 Devices



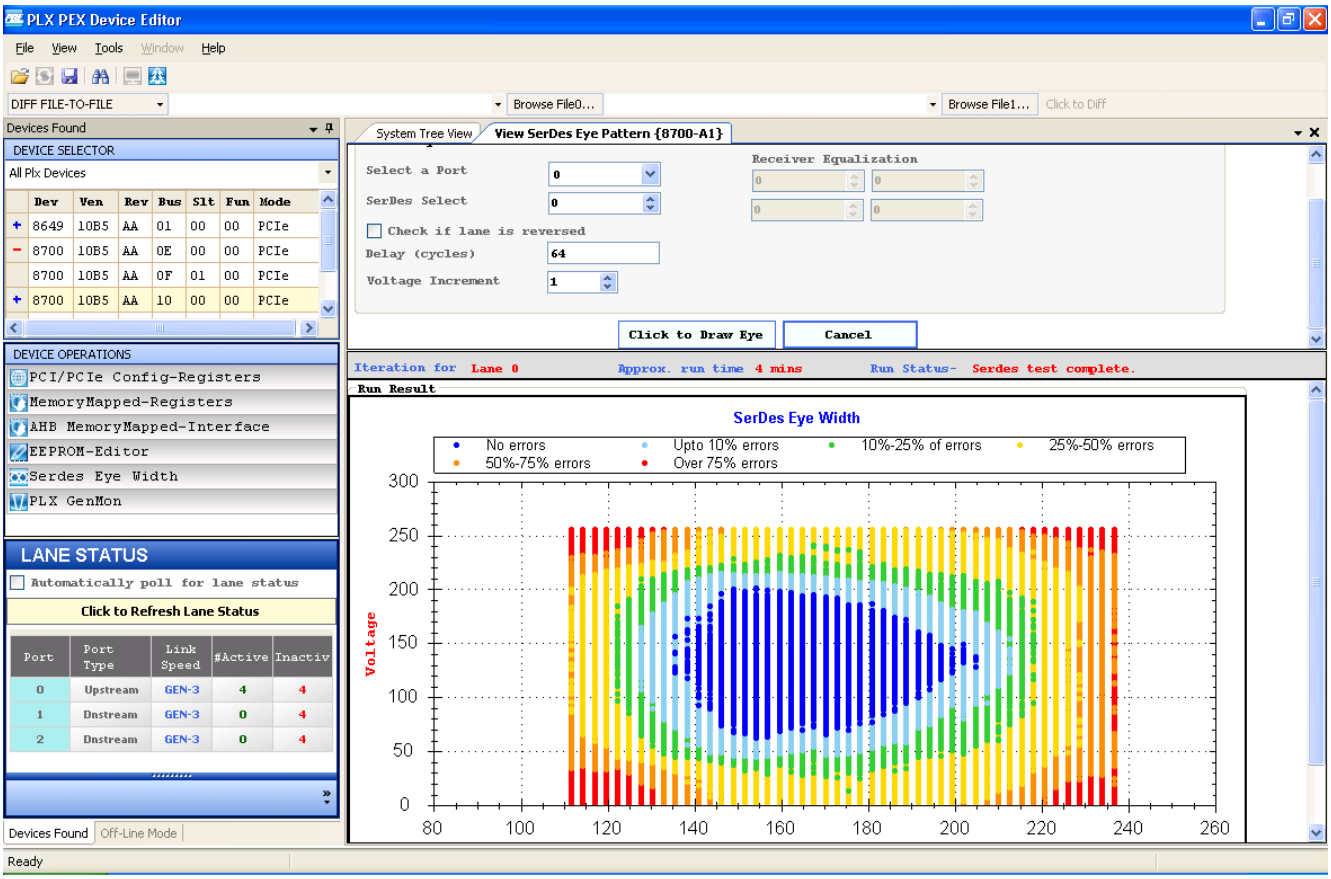
The appropriate Serdes can be put in Digital Loop back mode and a user programmable test pattern can be generated which will be sent out and received back. The received pattern will be compared with the expected pattern and an error counter is updated.

This error count is an indirect indicator of the signal level. The software will infer the voltage based on the error level. The user can also shift the Serdes clock phase in steps. At each step the signal quality can be checked until the error is at the maximum value. This indicates that we have reached the end of the eye. Then the same process is repeated by shifting the clock phase in the opposite direction to get to the other end of the eye. This gives us the total width of the eye.

Software will do the following steps:

- Put the Serdes in loop back mode
- Program the User Test Pattern and enable it.
- Shift the Serdes Clock Phase towards one direction in steps and check the error count after a few seconds. If errors are zero, keep repeating until you hit the first non-zero error count. Then go back a step and wait for a couple of minutes and check the error count. If you still get errors, go back further and continue until there are no errors. This way we get a plot of the error vs Unit Interval.
- Repeat previous step again by shifting the clock phase in the other direction.
- Plot a graph of the inferred voltage vs. Clock Phase in Unit Intervals. Inferred voltage is got from the error count.

4.1.5.2 Serdes Eye for PLX Gen3 Devices



For gen3 devices the serdes eye is much cleaner with both the width and height information being made available to the user. The algorithm used is quite different due to additional features found in the newer Serdes IP. The following algorithm is used to get the Serdes eye:

- enable eye scan feature
- enable comparator setting override
- enable eye scan error counter and wait time
- longer wait time means more accurate measurement
- wait for signal detect to go high
- wait 500ns~1us for CDR to lock
- loop (scan through the sampling points)

begin

X and Y axis of scan point

Set Lane eye delay value

set lane comparator offset override

load Y setting into comparator

wait 10ns

set AHB lane receiver equalization DFE comparator select override

set receiver equalization override latch

wait 10ns

set receiver equalization override latch

//start counting

set eye_scan_run = 1'b1

//depending on the eye scan wait time

wait 200ns

//read error

//reset counter

set eye_scan_run = 1'b0

save eye scan setting and error count

end

Sample code on how to generate the Serdes eye is provided in the SDK <*Samples**SerdesEyeTest*> folder.

4.2 PLX GenMon

The PLX GenMon application supports two features of some PLX chips. These are the Packet Generator and the Performance Monitor, which are available only on one some PLX devices. The GenMon application will provide access to only those installed PLX chips that support the feature.

4.2.1 Performace Monitor

The goal of this feature is to provide statistical information taken from performance counters found in some PLX switches. The following counters are available for every port:

- TIC Ingress TLP Posted Header
- TIC Ingress TLP Posted DW & TIC Ingress TLP Non-Posted DW
- TIC Ingress TLP Completion Header &TIC Ingress Completion DW
- TEC Ingress TLP Posted Header
- TEC Egress TLP Posted DW & TEC Egress TLP Non-Posted DW
- TEC Egress TLP Completion Header & TEC Egress Completion DW
- DLLP Ingress & DLLP Egress

Based on theses, various performance parameters can be calculated. The PLX Performance Monitor provides the following for each active port:

- Link Utilization Percentage
- Average Payload size
- Payload Byte Rate

PLX provides an API to setup and use the Performance Monitor. Sample code for utilizing this API is provided in the PLX SDK Samples folder. Details of the PLX Performance Monitor API are provided in PLX SDK API Reference section.

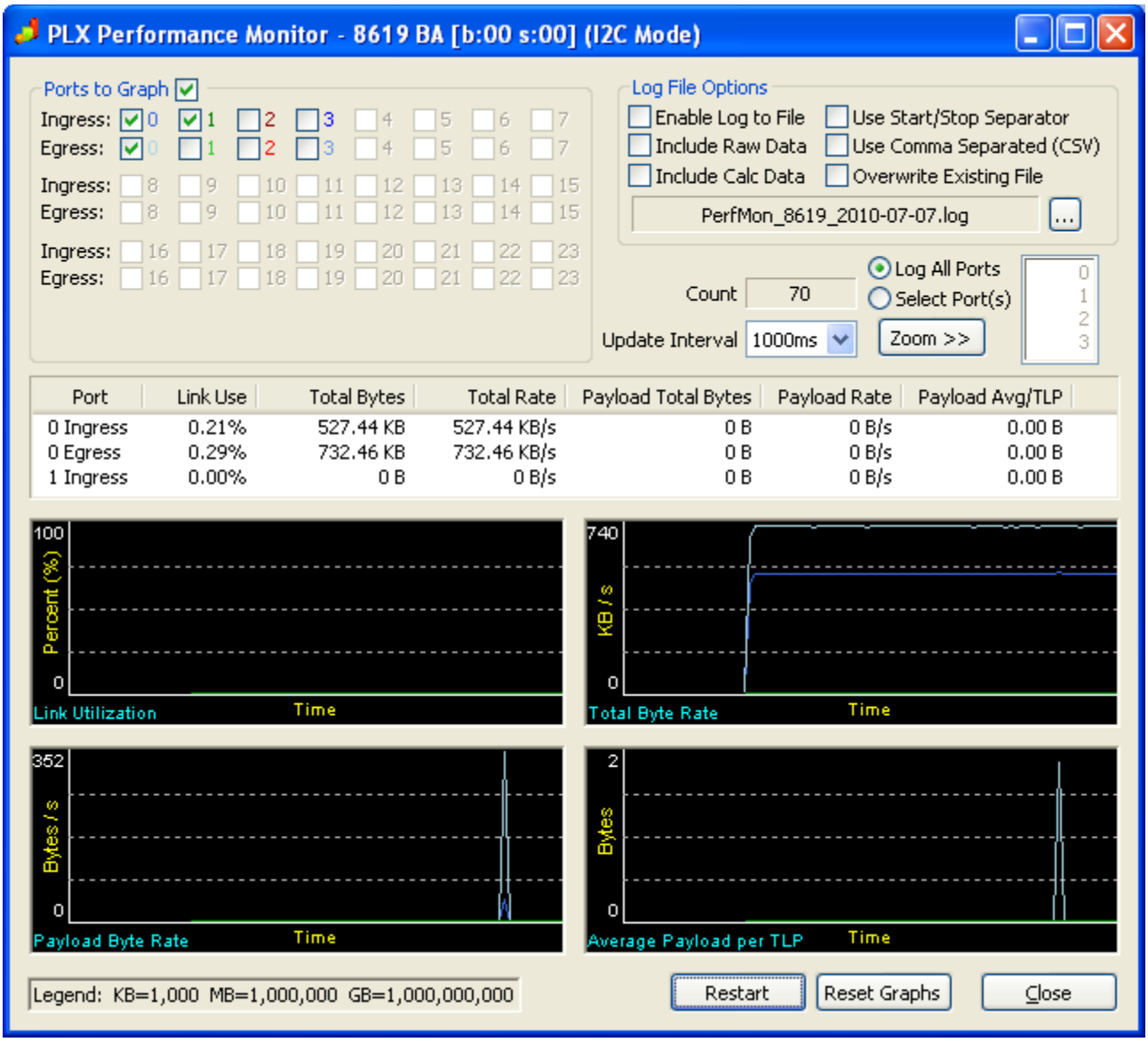


Figure 4-1 PLX Performance Monitor

4.2.2 Packet Generator

The Packet generator feature of PLX switches may be used to generate PCI compliant TLP packets. The various TLP parameters may be setup through the simple GUI.

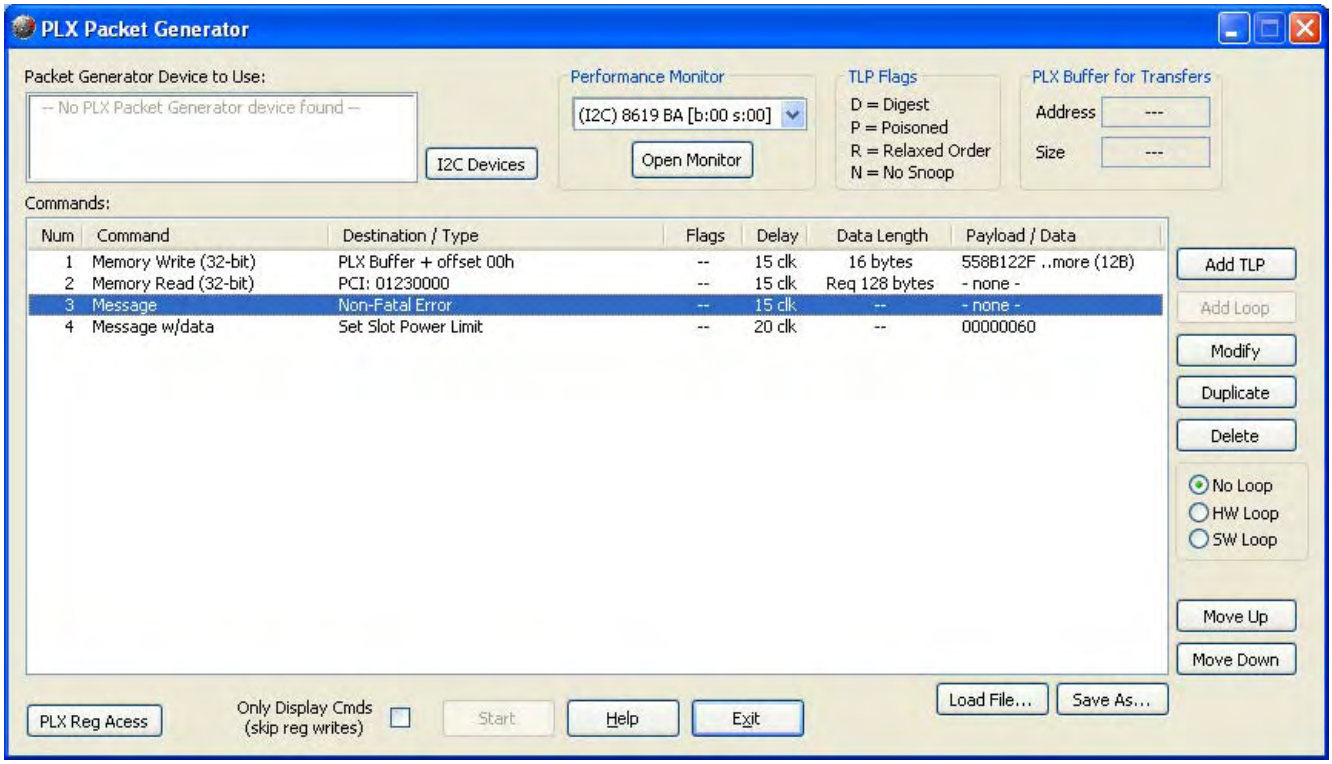


Figure 4-2 PLX Packet Generator

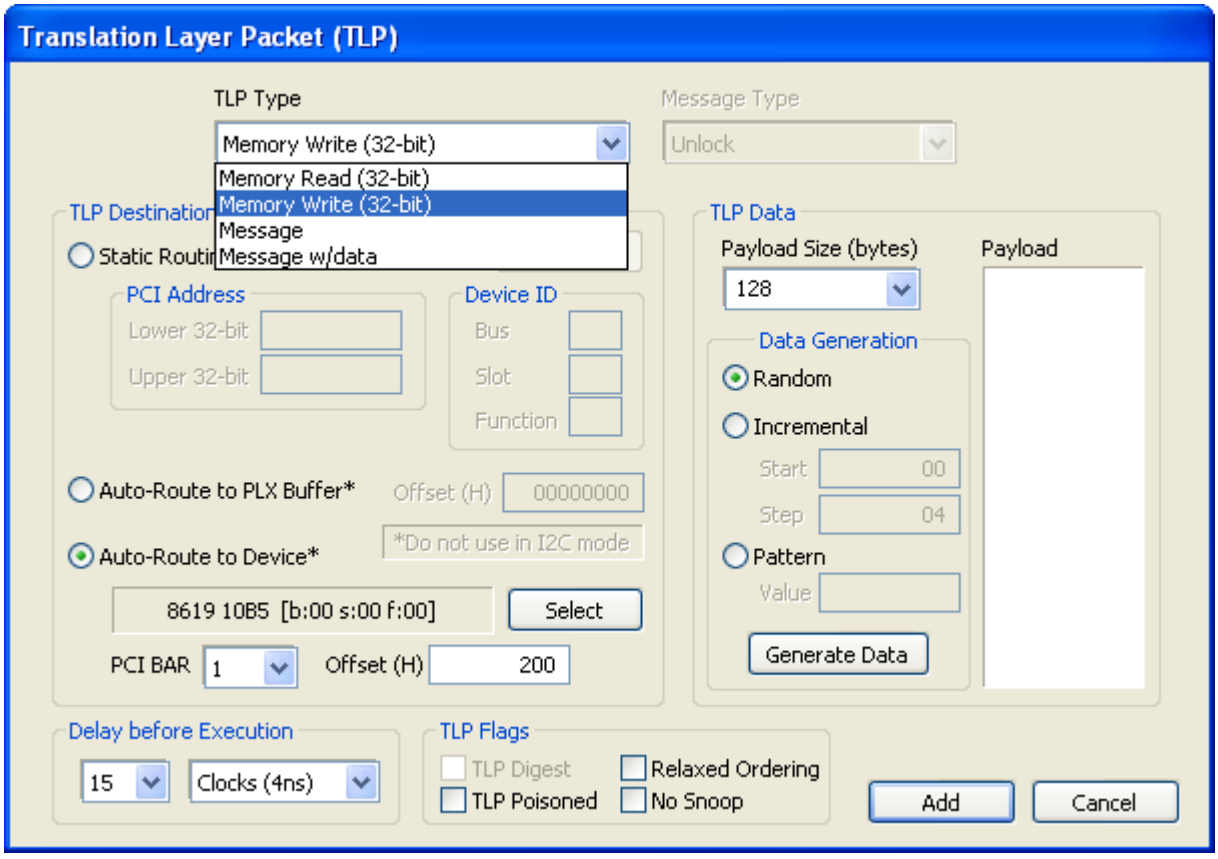


Figure 4-3 TLP Setup Options

4.3 PLXMon

The PLXMon debug utility is a powerful tool, which provides easy-to-use GUI screens for read/write of PLX chip registers, access to local bus devices, download of local software to RAM, programming of FLASH devices, and EEPROM access.

4.3.1 PLXMon Access Modes

PLXMon accesses the PLX chip in one of two ways: through the PCI bus or, if BEM compatible code is running on the local-side, through a serial cable connection. Figure 4-4 shows the PLX communication modes.

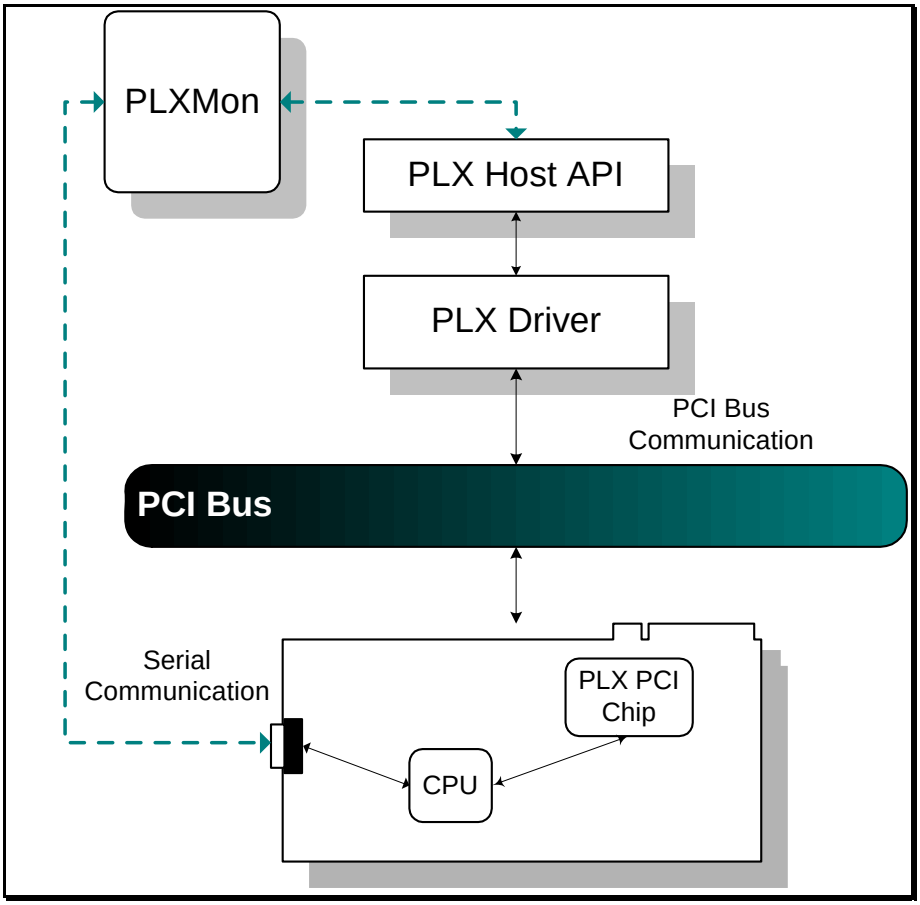


Figure 4-4 PLXMon Communications Modes

4.3.1.1 PCI Mode

In PCI mode, all accesses to the PLX chip are performed directly through the PCI bus, via the SDK API and PLX device driver. If a PLX driver is not installed/loaded, PCI mode will be unavailable. In PCI mode, the upper pane in PLXMon is disabled. The lower pane is an interpreter that accepts commands to access registers and memory.

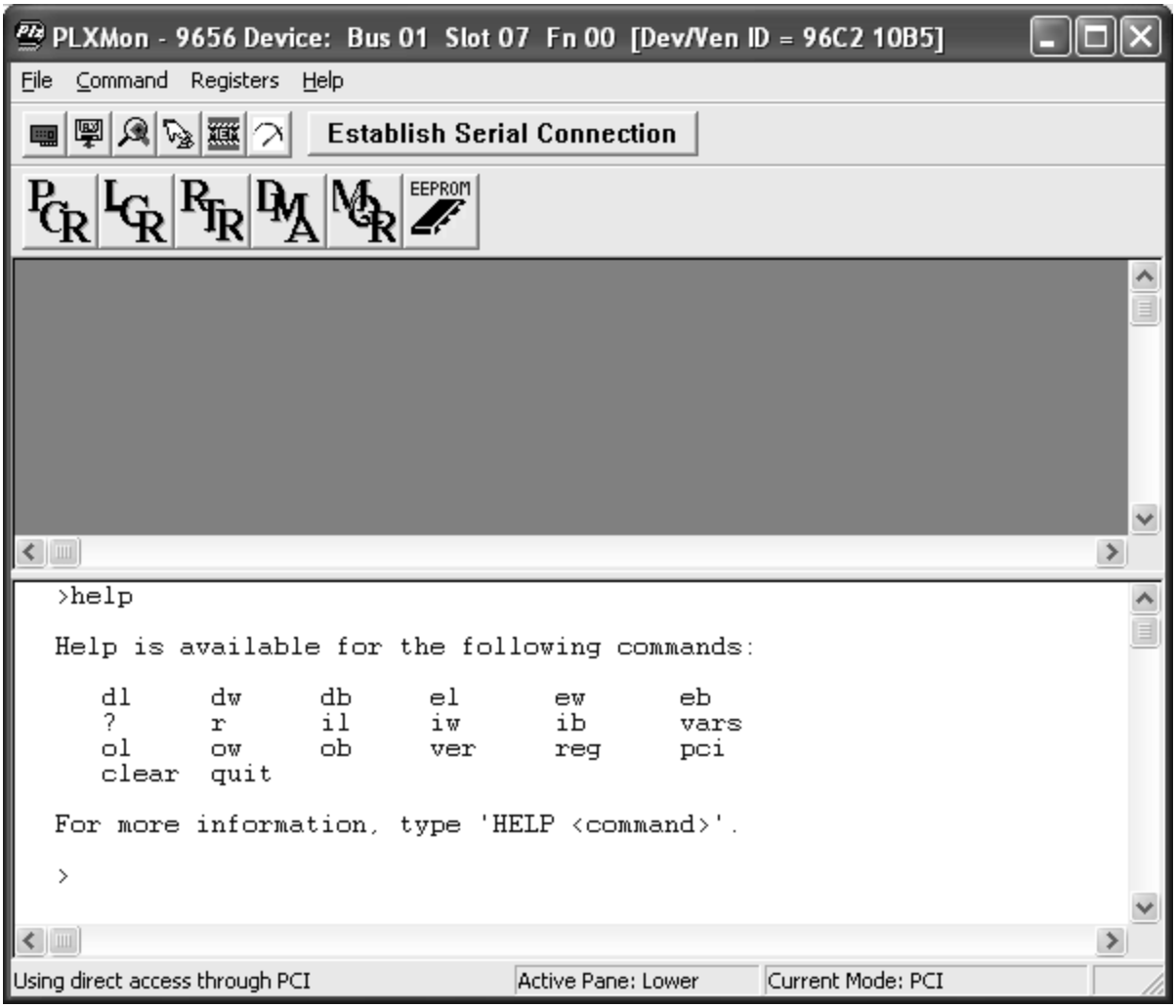


Figure 4-5 PLXMon in PCI Mode

4.3.1.2 EEPROM File Edit Mode

If a PLX device is not detected in the system, PLXMon displays a dialog (Figure 4-6), which provides two options: Enter EEPROM File Edit mode or attempt a connection to enter Serial mode.

The EEPROM edit mode is provided for those who need to create or modify EEPROM files, which will be used with an I/O programmer. In this mode, since no PLX devices are physically present in the system, PLXMon cannot program the EEPROM device directly.

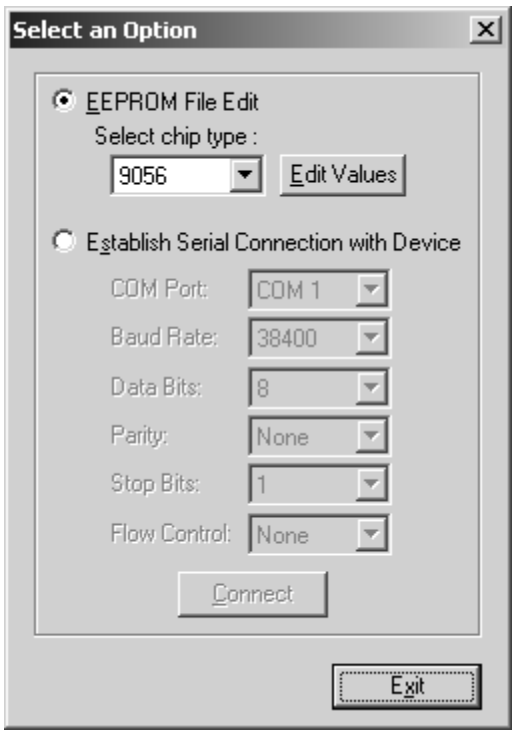


Figure 4-6: EEPROM Edit Mode

4.3.1.3 Serial Mode

In Serial Mode, PLXMon establishes a serial connection with a device. In this mode, the software executing on the local CPU (PLX BEM) accepts and carries out commands from PLXMon to perform necessary tasks. While connected, the upper pane of PLXMon provides a terminal interfaces, similar to other serial terminal applications, such as *HyperTerminal*. The lower pane is an interpreter that accepts commands to access registers and memory. It is important to note that in Serial mode, the local CPU handles commands entered in the lower pane, so memory and registers are accessed from the local CPU's point of view. In Serial mode, the command '***dl 100000***' will read from the local address location 1MB. Conversely, in PCI mode, only virtual addresses are allowed, so the same command will most likely result in an invalid address.

Figure 4-7: PLXMon in Serial Mode

4.3.2 PLXMon Toolbar

Figure 4-8: PLXMon Toolbar

The PLXMon toolbar (Figure 4-8) provides multiple options, which are described below:

- ***Select a Device*** 
  View all PLX devices found and select one to work with. Only devices, for which a PLX driver is loaded, will be available.

- ***Download to device*** 
  Opens the download dialog, which allows downloading of RAM images and programming of the FLASH ROM.
- ***View all PCI devices*** 
  Open a dialog, which displays all PCI devices in the system. Selecting one displays all PCI registers of the device
- ***Reset device*** 
  In PCI mode, resets a device by using the *Software Reset* feature of PLX chips. In Serial Mode, issues a reset command to the local CPU.
- ***Memory Access*** 
  Opens the memory access dialog.
- ***Performance Measure Dialog*** 
  In PCI mode for PLX 9000-series devices, provides a software measure for DMA and Direct Slave transfers. *Refer to the Performance Measure Dialog, section 4.3.5.*
- ***Connect to device*** 

  Attempt a serial connection to the device. If the local software implements the BEM protocol, PLXMon will establish a connection.
- ***View Register Groups*** 

  Open dialogs for the various register groups and EEPROM. The PLX chip type determines available groups.

4.3.3 Working with PLXMon Dialogs

4.3.3.1 Register Dialogs

The register dialogs in PLXMon are very simple to use. Users simply enter values, in Hexadecimal format, and PLXMon will update the value in the chip. For some registers with numerous bit-fields, PLXMon provides additional detail screens, which can be selected with the details button -  . Figure 4-9 demonstrates a typical register dialog.

Tips on working with register dialogs:

- All values are in Hexadecimal format
- The register dialogs are available in both Serial and PCI modes. In Serial mode, PLXMon sends commands to the local CPU to perform register accesses. In PCI mode, PLXMon calls the PLX Host API to access registers.
- The register offsets displayed are dependent upon the mode of operation. In Serial mode, the offsets are from the local CPU's point of view. *Refer to the PLX chip data book for more information regarding offsets*.
- In the register dialogs, PLXMon will update a register value as soon as focus shifts from the field (i.e. the TAB key or clicking on a different field with the mouse).

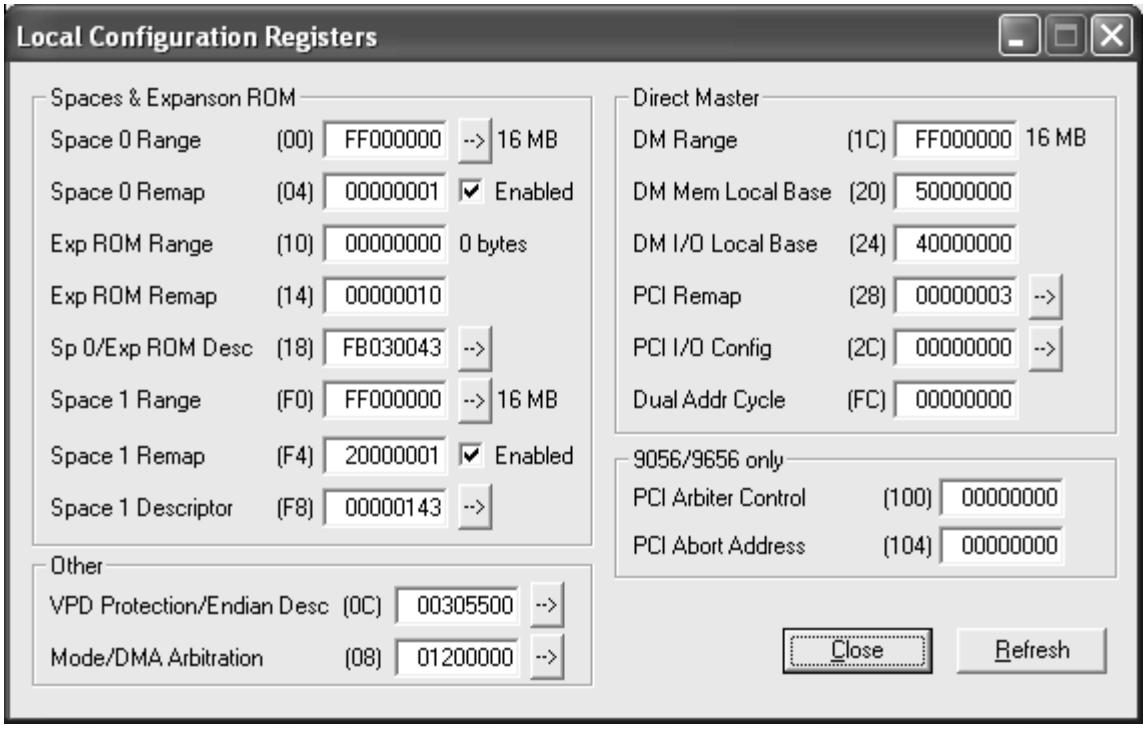


Figure 4-9: Typical PLXMon Register Dialog

4.3.3.2 EEPROM Dialogs

The EEPROM dialogs in PLXMon behave very similar to the register dialog, with a few exceptions. Additionally, the EEPROM dialogs provide options to save/load values to/from files. Figure 4-10 demonstrates a typical EEPROM dialog.

EEPROM Dialog Differences from Register Dialogs:

- Displayed offsets are from the EEPROM base (default), but offsets of the target register in the chip can be selected, as well.
- Values are not written to the EEPROM device until the ***Write*** button is selected.
- Values can be loaded from or saved to a file. When working with EEPROM files, PLXMon will only load or save enough values to fill the PLX chip's portion of the EEPROM. Additional values are discarded.

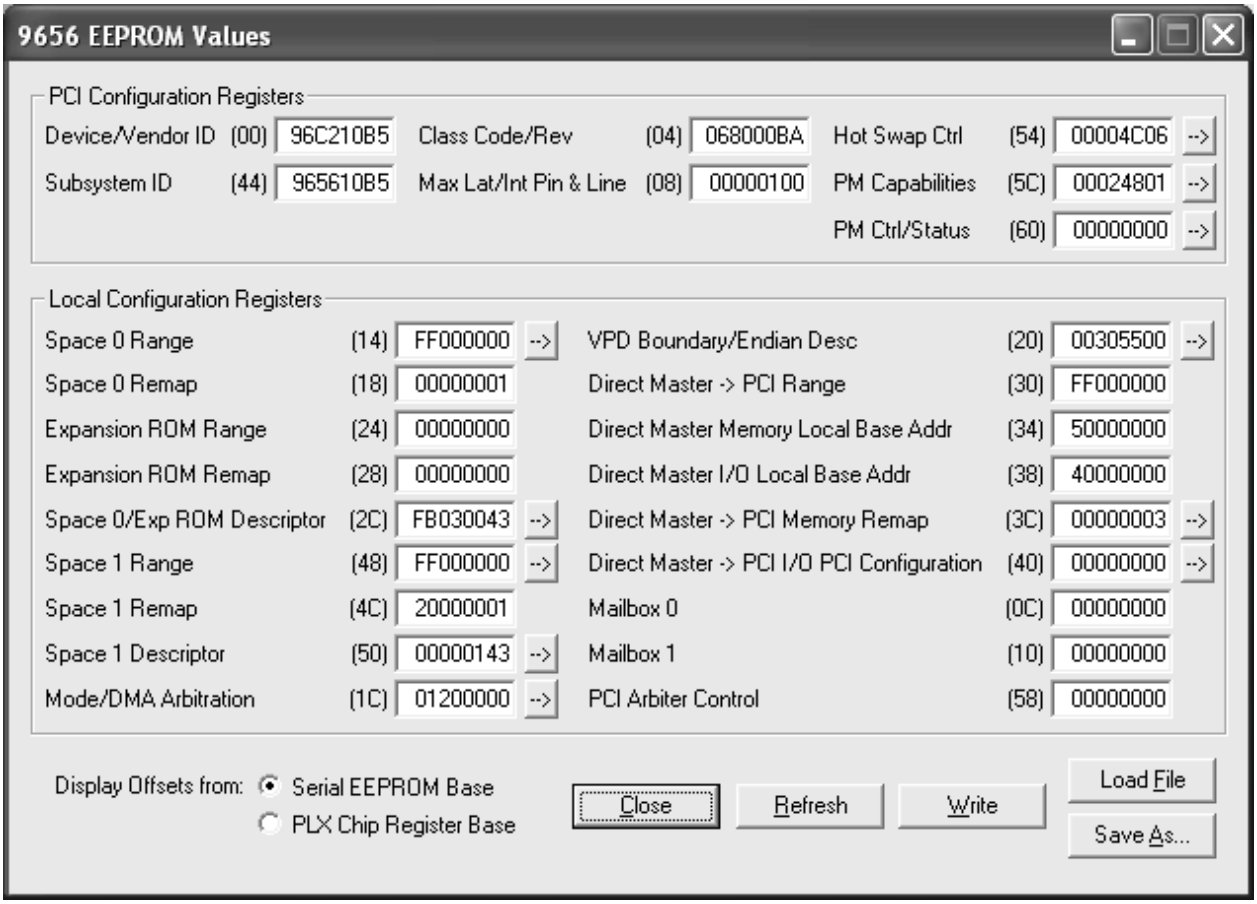


Figure 4-10: Typical EEPROM Dialog

4.3.3.3 Memory Access Dialog

Selecting the memory access button will open the dialog shown in Figure 4-11. The memory dialog allows reading of blocks of memory from the local bus or from the DMA buffer, as well as the ability to fill memory, as shown in Figure 4-12. For more control over memory accesses, use the ***db, dw, dl, eb, ew*** and ***el*** commands. Note that in PCI mode, virtual addresses are used. *Refer to Section 0 for more information*.

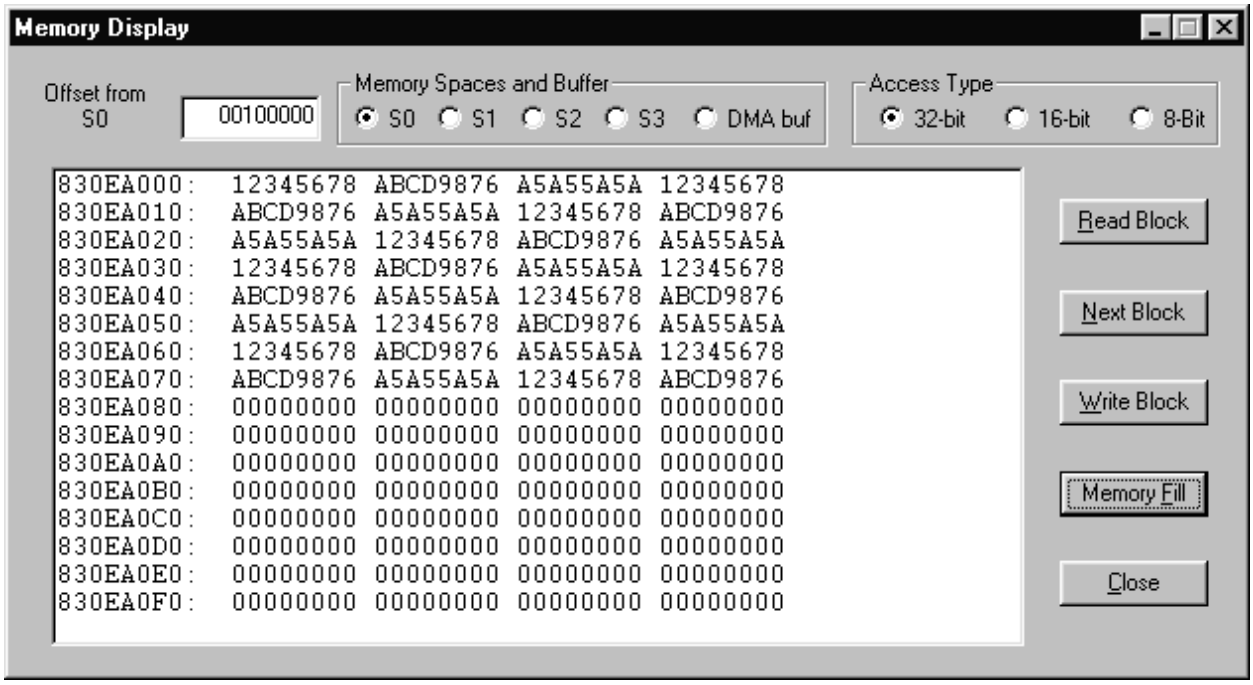


Figure 4-11: Memory Access Dialog

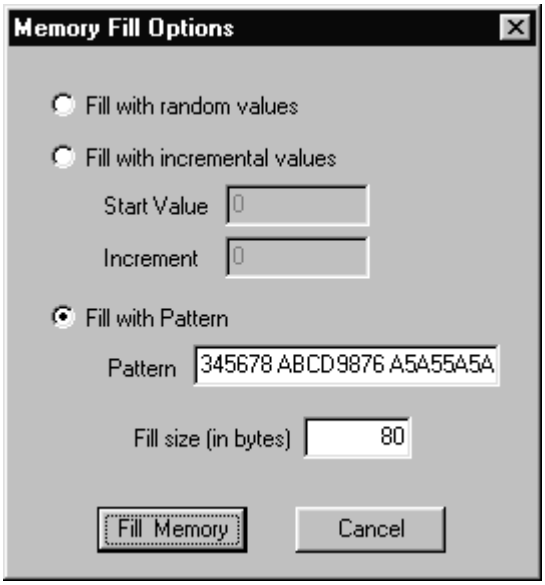


Figure 4-12: Memory Fill Dialog

4.3.4 Specifying PLX Chip Type for Unknown Devices

If the Device/Vendor ID of a PCI 6000 series bridge is modified from its default, PLX software may fail to properly identify the device as a PLX chip. In this case, PLXMon will not be able to properly display all of the PCI registers and the EEPROM contents. PLX drivers rely on known Device/Vendor ID combinations to detect PLX PCI 6000 and 8111 devices. As a result, the IDs are hard-coded into the driver source code. A customer that changes an ID will, therefore, need to modify the driver source and rebuild it. PLX software, however, provides an option to manually override the chip type in the event that it is not detected properly. This can be performed in PLXMon in the “Select a PCI Device” dialog.

Simply select a device and then select the option to manually set the chip type. Figure 7-13 shows how to manually select a chip type.

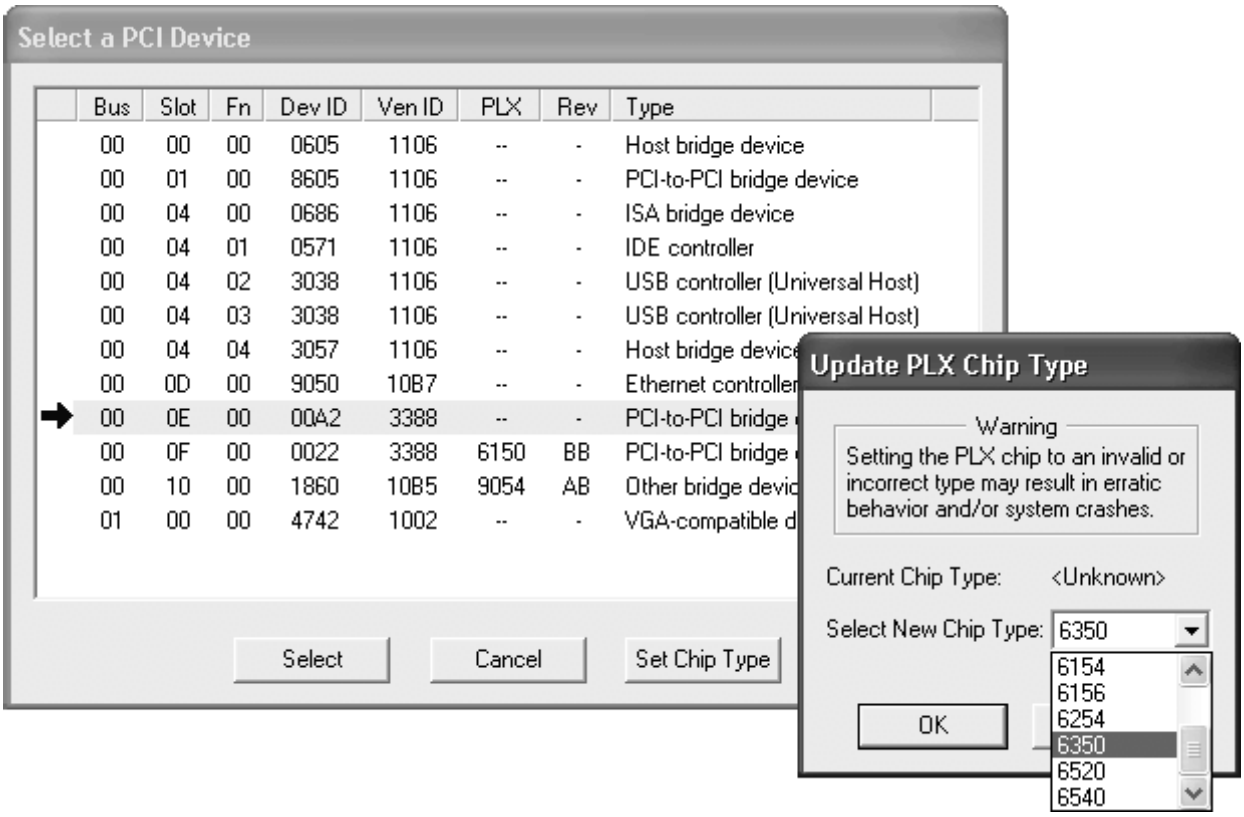


Figure 4-13 Manually Setting the PLX Chip Type

After the selection has been made, PLXMon will treat the device as the user-selected type, as can be seen in Figure 7-14. Before setting the PLX chip type, it is important to note the following:

- No error checking is performed when setting the PLX chip type. If a PLX chip is selected that does not match the installed hardware, the PLXMon and/or the system may behave erratically.
- Once the chip type is selected, the PLX driver will attempt to automatically detect the PLX revision. If this is not detected, the revision will default to the value in the PCI revision ID register.
- Modification of the PLX chip type is not permanent. It will remain in effect as long as the PLX driver is loaded and not re-started. For a permanent setting, it is recommended that the PLX PCI Service driver is modified and rebuilt to properly detect the custom ID.
- This option may only be used with PLX PCI 6000 series and 8111 devices.

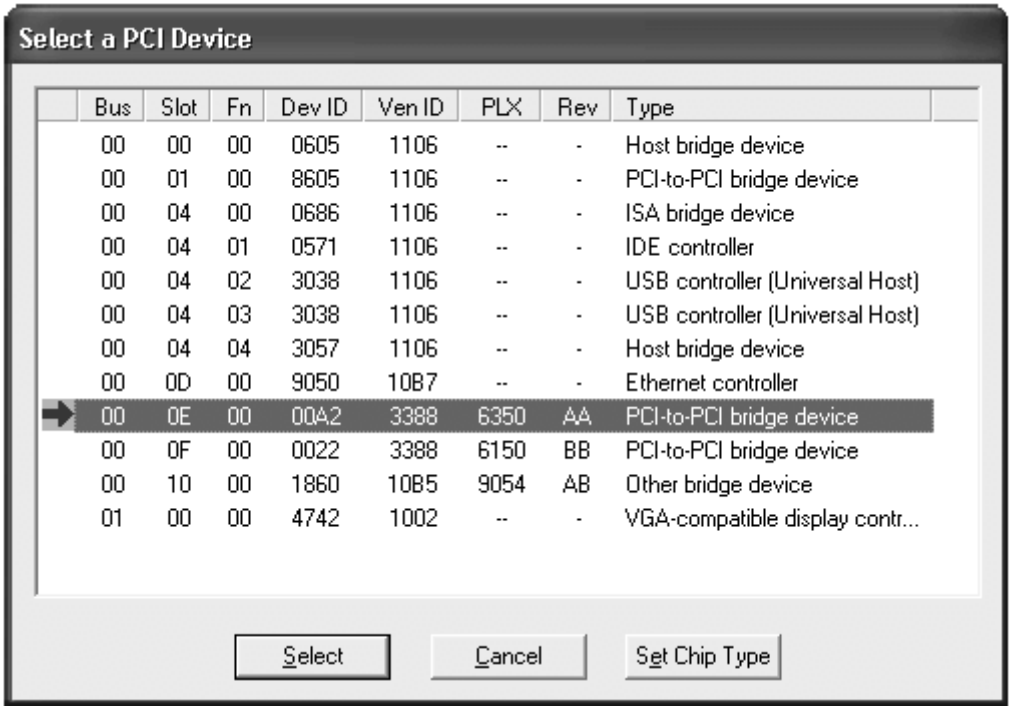


Figure 4-14 Completed PLX Chip Type Override

4.3.5 Performance Measure Dialog

PLXMon includes a performance measure dialog, which provides a software measure of data transfer performance. The dialog supports DMA and Direct Slave transfers, with multiple options for each. This section describes the details of how to use the dialog. Figure 4-15 shows a snapshot of the dialog.

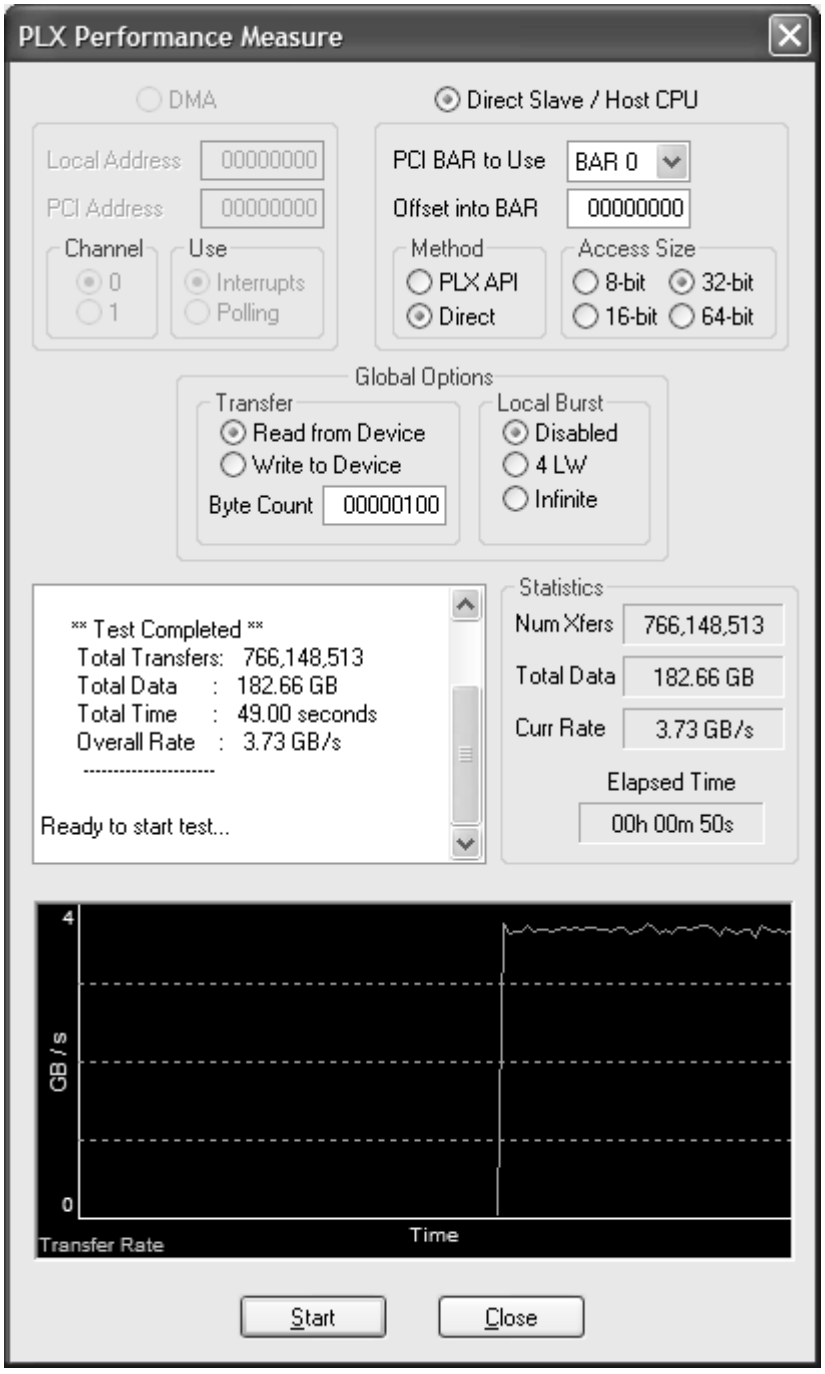


Figure 4-15 Performance Measure Dialog

4.3.5.1 Notes before Using the Performance Measure

Before using the performance dialog, it is important to be aware of the following imitations and notes:

- The Performance Measure is a simple software measurement of performance. The transfer rate is calculated by dividing the total number of bytes transferred by the total elapsed time. As a result, software overhead is a factor in the measure, although the Performance Measure is very efficient and includes very little overhead.

- The transfer rates provided by the Performance Measure should be treated as relative numbers rather than absolute values. The intention is to start with some base configuration, tweak some options and/or chip settings, then re-run the test to determine if performance has improved and repeat to achieve the optimal configuration.
- The Performance Measure does not validate the addresses used to transfer data to/from. This includes the PCI and local addresses for DMA and the local address for Direct Slave. It is left to the user to ensure that sufficient memory is provided for the transfer.
- The Performance Measure does not perform any data error checking. It is assumed that hardware is working properly.
- When selecting to use the PLX API to transfer data, it is important to note that there is a significant overhead with doing so. The API sends and receives messages from the PLX driver, which performs the actual transfer. If data transfer sizes are relatively small, the API overhead will be a significant impact to performance. As data transfer sizes get larger, the API overhead becomes less significant.
- The Performance Measure cannot guarantee burst transactions. Software has no means to force burst transactions. All software can do is enable burst in the hardware and, if conditions are right, the hardware will initiate burst transactions.
- Other than the options specified, the Performance Measure will leave chip settings intact. It is assumed that the chip is properly configured to access the intended devices. For example, if PCI BAR 2 on a 9054 will be used to access an 8-bit device, it is assumed that the Space 0 Bus Region Descriptor is configured properly and that the Space 0 Remap register is set to properly access the desired device.

4.3.5.2 Performance Measure Options

The performance measure provides numerous options to perform different type of transfers in different configurations. The individual options are explained below..

4.3.5.3 DMA Performance Test

When the DMA test is selected, the Performance Measure will perform DMA transfers to or from the specified addresses. The test continuously repeats the same DMA transfer until it is halted

The items below provide details about the individual DMA options. When the Performance Measure is initially opened and DMA is available, it will provide the DMA Common Buffer properties, which are provided by the PLX driver. This is the same information obtained with *PlxPciCommonBufferProperties*.

Note: *DMA is available only to PLX devices that include a DMA engine, including the 9080, 9054, 9056, 9656, & 8311.*

- **Local Address** This determines the starting 32-bit local address where data is transferred to/from. This value is placed directly into the Local address register of the DMA engine.
- **PCI Address** This determines the starting 32-bit PCI physical address where data is transferred to/from. This address, for example, may be the PLX DMA Common Buffer PCI address or an address taken from the PCI BAR of another PCI device, such as an Ethernet controller. This value is placed directly into the PCI address register of the DMA engine.
- **Channel** This determines which DMA channel the Performance Measure will use.
- **Use** This determines whether DMA completion is detected by waiting for the interrupt or polling the DMA done bit. In general, polling results in better transfer rates due to less overhead, but the CPU is highly utilized, so the end user system performance suffers.
- **Transfer** This determines which direction the DMA engine will transfer data.
- **Byte Count** This is the number of bytes transferred during each test iteration.
- **Bursting** This option determines whether DMA busting is enabled in the hardware. Note that the devices that the DMA engine transfers to/from must support the selected type of burst transaction.

4.3.5.4 Direct Slave Performance Test

When the Direct Slave test is selected, the Performance Measure will use the Host CPU to transfer data to/from a PLX device through one of the PCI BAR spaces. The test will repeat continuously until it is halted. Figure 4-16 depicts a completed Direct Slave test and the reported results.

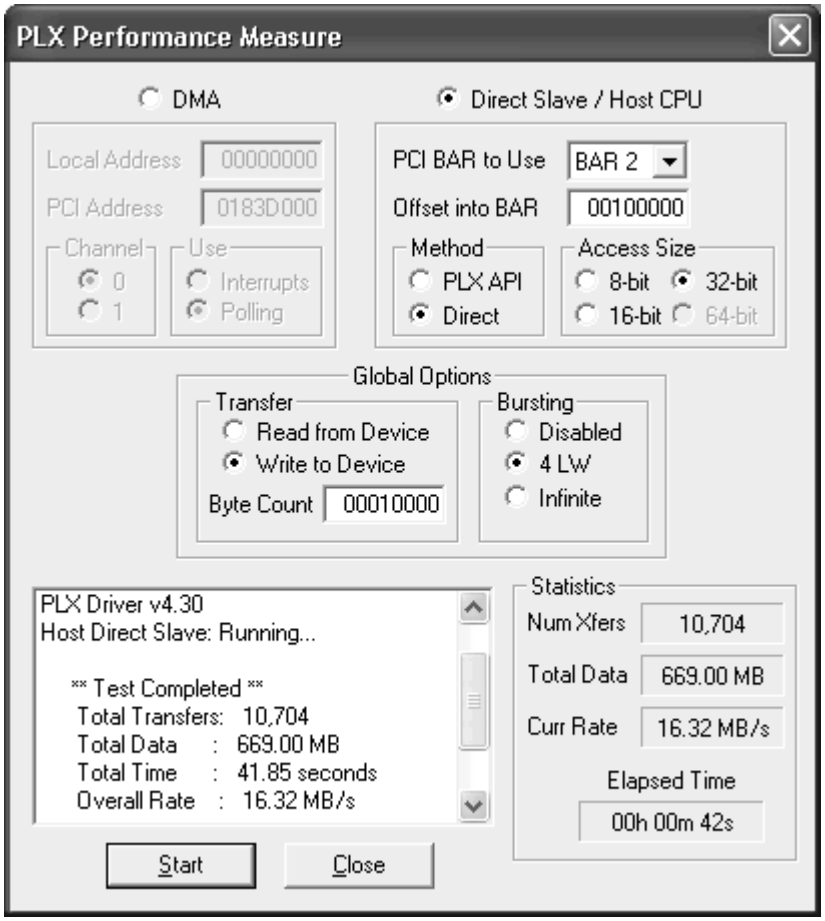


Figure 4-16 Sample Direct Slave Performance Test

The items below provide details about the individual Direct Slave options.

- **PCI BAR to Use** This determines which PCI BAR space to use for the transfer. The PCI BAR must be a valid PCI memory space that is enabled on the PLX device. I/O type spaces are not supported. It is assumed that the PCI space is properly configured to access the desired local device. This includes the remap and bus region descriptors.
- **Offset into PCI BAR** This value determines the starting offset into the PCI BAR where the Performance Measure will transfer data.
- **Method** This option determines whether the PLX API is used to transfer data or a direct access is performed. The PLX API method will use the functions *PlxBusIopRead* and *PlxBusIopWrite*, whereas, the direct method will obtain a virtual address for the PCI BAR with *PlxPciBarMap*, then use that address to directly access the PCI space. The direct method effectively bypasses the PLX API.
- **Access Size** This option determines how data is accessed, whether it is 8-bit, 16-bit, or 32-bit. This option should not be confused with the “Bus Width” of the Bus Region Descriptor for a space. The Bus Width is used to specify the port-size of the connected local device, for example, a 16-bit flash device. The Access Size determines the type of cycle issued by the Host CPU.
- **Transfer** This determines whether the Host CPU reads from or writes to the PCI BAR.
- **Byte Count** This is the number of bytes transferred during each test iteration.

- **Bursting** This option determines whether Direct Slave busting is enabled in the Bus Region descriptor for the PCI space. This option does not guarantee that burst transactions will occur, since software is not able to force bursting. In a standard PC, for example, the Host Bridge does not allow burst reads from PCI devices to the Host CPU, resulting in typically poor burst read performance. Note that the devices that data will be transferred to/from must support the selected type of burst transaction.

4.3.6 The Command-Line Interface

In the lower pane of PLXMon, a command-line interface is provided, as show in Figure 4-17. The list of available command is show in Table 4-1.

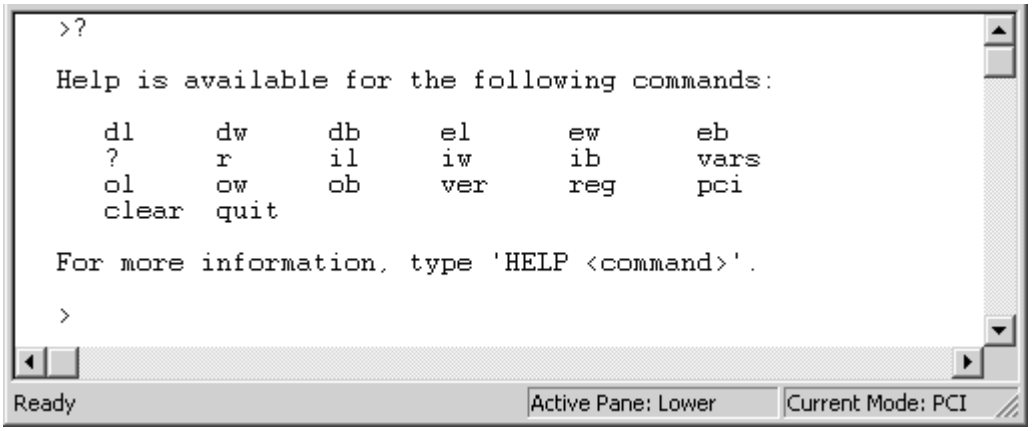


Figure 4-17: Command-line Interface

Command	Description
db, dw, dl	Read memory using Byte (8-bit), Word (16-bit), Longword (32-bit)
eb, ew, el	Write to memory using Byte (8-bit), Word (16-bit), Longword (32-bit)
ib, iw, il	Read from I/O port using Byte (8-bit), Word (16-bit), Longword (32-bit)
ob, ow, ol	Write to I/O port using Byte (8-bit), Word (16-bit), Longword (32-bit)
pci	Read/Write to a PCI register of the PLX chip
reg	Read/Write to a local register of the PLX chip
vars	Display PLXMon variables. *See Section 4.3.8*
ver	Display version information
clear	Clear the command-line pane
quit	Exits PLXMon

Table 4-1: PLXMon Command-line Commands

4.3.7 Working with Virtual Addresses

In PCI mode, PLXMon executes as an application and, therefore, must use virtual addresses to access memory. A PCI BAR address, for example, cannot be referenced directly. As a result, PLXMon relies on PLX drivers to provide a virtual mapping for all memory spaces that may be accessed. This includes any valid PCI BAR memory spaces and the DMA buffer allocated by the driver.

***Note**: Virtual addresses are not used for I/O ports, only for memory regions. Although the driver performs the actual I/O access, the referenced port address is the actual address found in the PCI BAR register. I/O regions are not mapped into virtual space.*

4.3.8 Command-Line Variables

PLXMon creates some variables to aid users with dealing with virtual addresses. Figure 4-18 demonstrates the ***vars*** command in PLXMon, which lists the default variables and the memory region they represent. Variables can be used with the ***d(b,w,l)*** or ***e(b,w,l)*** commands.

Note: *Accessing memory with these variables results in a direct memory access from PLXMon. The PLX driver just provides the initial virtual mapping, but is completely bypassed during memory accesses.*

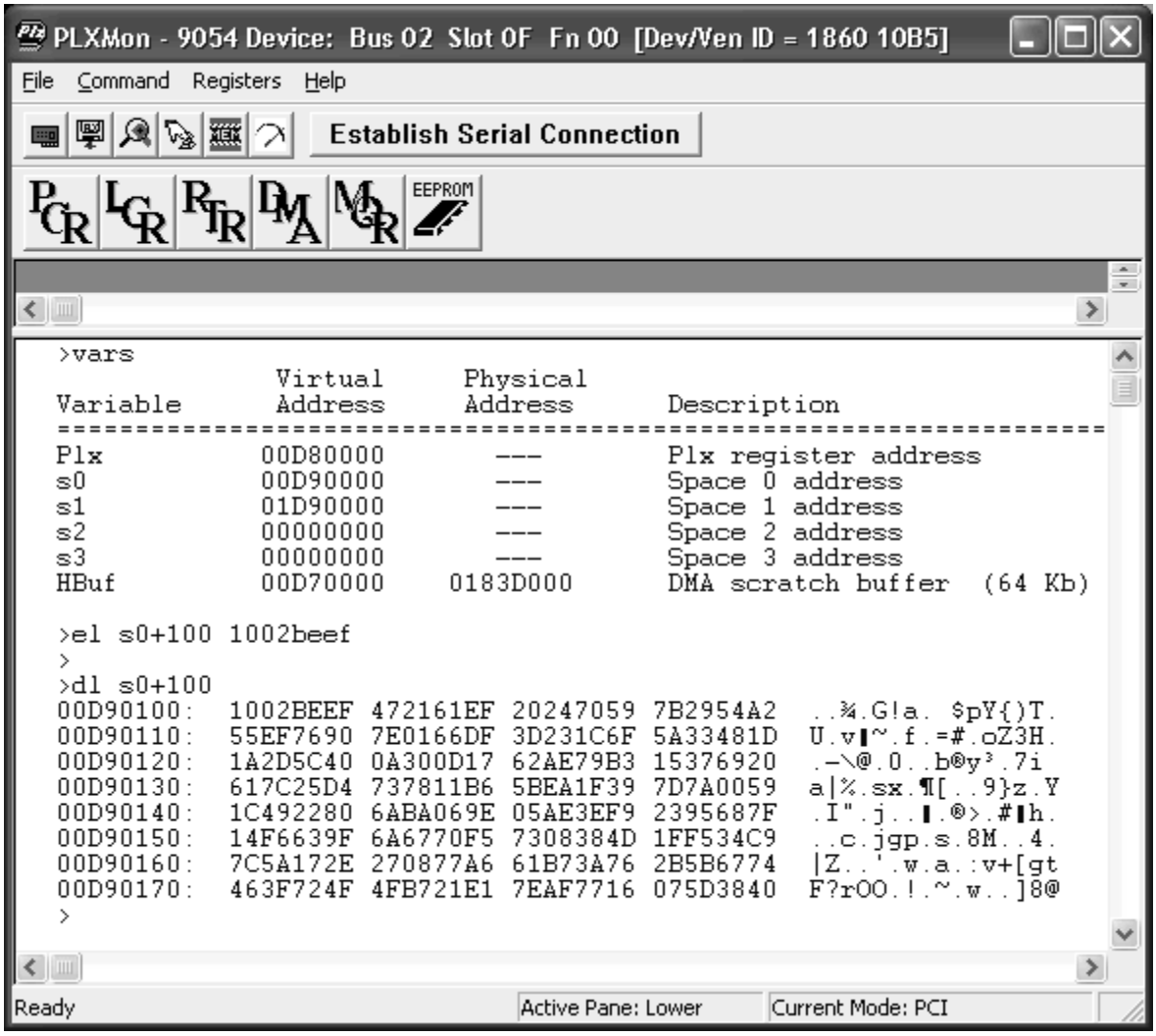


Figure 4-18: PLXMon Variables

5 PLX SDK API Reference

This section provides the details of all PLX API functions.

5.1 PLX API Functions

API Function Name	Description
PlxPci_ApiVersion	Get the PLX API library version information
PlxPci_ChipTypeGet	Get the PLX chip type and revision
PlxPci_ChipTypeSet	Set the PLX chip type
PlxPci_CommonBufferProperties	Returns the properties of the PLX driver reserved buffer
PlxPci_CommonBufferMap	Maps the common buffer to user space
PlxPci_CommonBufferUnmap	Unmaps the common buffer from user space
PlxPci_DeviceClose	Release a device
PlxPci_DeviceFind	Search for a device
PlxPci_DeviceFindEx	Search for a device with advanced options (e.g. I^2C)
PlxPci_DeviceReset	Reset a PLX device
PlxPci_DeviceOpen	Select a device
PlxPci_DmaChannelOpen	Opens & initializes a DMA channel
PlxPci_DmaChannelClose	Release a DMA channel
PlxPci_DmaGetProperties	Gets the current properties of a DMA channel
PlxPci_DmaSetProperties	Sets the properties of a DMA channel
PlxPci_DmaControl	Control a DMA channel
PlxPci_DmaStatus	Get current status of a DMA channel
PlxPci_DmaTransferBlock	Transfers a data buffer using block DMA
PlxPci_DmaTransferUserBuffer	Transfers a user-mode buffer using a DMA channel
PlxPci_DriverProperties	Get PLX driver properties
PlxPci_DriverScheduleRescan	Informs PLX Service driver to rebuild its internal device list
PlxPci_DriverVersion	Get the PLX driver version information
PlxPci_EepromPresent	Determine if an EEPROM is present on a PCI device
PlxPci_EepromProbe	Probes for the physical presence of an EEPROM
PlxPci_EepromCrcGet	Get the CRC value of the EEPROM
PlxPci_EepromCrcUpdate	Update the CRC value of the EEPROM
PlxPci_EepromSetAddressWidth	Manually sets the EEPROM addressing width
PlxPci_EepromReadByOffset	Read a 32-bit value from the EEPROM at a specified offset
PlxPci_EepromWriteByOffset	Write a 32-bit value to the EEPROM at a specified offset
PlxPci_EepromReadByOffset_16	Read a 16-bit value from the EEPROM at a specified offset
PlxPci_EepromWriteByOffset_16	Write a 16-bit value to the EEPROM at a specified offset
PlxPci_GetPortProperties	Get the port properties of the selected device
PlxPci_I2cGetPorts	Gets the installed I^2C USB devices and their availability
PlxPci_I2cVersion	Gets I^2C version information
PlxPci_IoPortRead	Reads one or more values from an I/O port
PlxPci_IoPortWrite	Writes one or more values to an I/O port
PlxPci_InterruptDisable	Disables specific interrupts of the PLX chip
PlxPci_InterruptEnable	Enables specific interrupts of the PLX chip
PlxPci_NotificationCancel	Cancels and interrupt notification object
PlxPci_NotificationRegisterFor	Registers for interrupt notification
PlxPci_NotificationStatus	Returns the status of the interrupt notification object

API Function Name	Description
PlxPci_NotificationWait	Wait for an interrupt notification event
PlxPci_Nt_LutAdd	Add an entry to the NT Requester ID LUT
PlxPci_Nt_LutDisable	Disable an entry in the NT Requester ID LUT
PlxPci_Nt_LutProperties	Return the properties of an entry in the NT Requester ID LUT
PlxPci_Nt_ReqIdProbe	Determines the Host PCIe ReqID when accessing the NT port
PlxPci_PciBarSpaceRead	Reads a block of data from the specified PCI BAR space
PlxPci_PciBarSpaceWrite	Writes a block of data to the specified PCI BAR space
PlxPci_PciBarMap	Maps a PCI BAR space to user virtual space
PlxPci_PciBarProperties	Returns the properties of a PCI BAR space
PlxPci_PciBarUnmap	Unmaps a PCI BAR space from user virtual space
PlxPci_PciRegisterRead	Read a PCI configuration register of a PCI device
PlxPci_PciRegisterWrite	Write to a PCI configuration register of a PCI device
PlxPci_PciRegisterReadFast	Reads a PCI register from the selected device
PlxPci_PciRegisterWriteFast	Writes to a PCI register on the selected device
PlxPci_PciRegisterRead_BypassOS	Reads a PCI register by bypassing the OS services
PlxPci_PciRegisterWrite_BypassOS	Writes to a PCI register by bypassing the OS services
PlxPci_PerformanceCalcStatistics	Calculates port performance statistics
PlxPci_PerformanceGetCounters	Reads the performance counters from a device
PlxPci_PerformanceInitializeProperties	Intialize the PLX performance object
PlxPci_PerformanceMonitorControl	Controls the PLX chip's perfomance monitor
PlxPci_PerformanceResetCounters	Resets the PLX chips's performance counters
PlxPci_PhysicalMemoryAllocate	Allocate Physical memory for the selected device
PlxPci_PhysicalMemoryFree	Free the allocated Physical memory for the selected device
PlxPci_PhysicalMemoryMap	Map the Physical memory to a Virtual address
PlxPci_PhysicalMemoryUnmap	Unmap Physical memory to the Virtual Address
PlxPci_PlxRegisterRead	Reads a PLX-specific register from the selected device
PlxPci_PlxRegisterWrite	Writes to a PLX-specific register on the selected device
PlxPci_PlxMappedRegisterRead	Reads a Memory mapped register from the selected device
PlxPci_PlxMappedRegisterWrite	Writes to a Memory mapped register on the selected device
PlxPci_VpdRead	Uses the VPD feature to read VPD data
PlxPci_VpdWrite	Uses the VPD feature to write VPD data

PlxPci_ApiVersion

Syntax:

```
PLX_STATUS
PlxPci_ApiVersion(
    U8 *pVersionMajor,
    U8 *pVersionMinor,
    U8 *pVersionRevision
    );
```

PLX Chip Support:

N/A

Description:

Returns the SDK API version information

Parameters:

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

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

pVersionRevision
: 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  VerMajor;
    U8  VerMinor;
    U8  VerRev;

    PlxPci_ApiVersion(
        &VerMajor,
        &VerMinor,
        &VerRev
        );

    Cons_printf(
        "PLX SDK API v%d.%d%d\n",
        VerMajor,
        VerMinor,
        VerRev
        );
```

PlxPci_ChipTypeGet

Syntax:

```
PLX_STATUS
PlxPci_ChipTypeGet(
    PLX_DEVICE_OBJECT *pDevice,
    U16               *pChipType,
    U8                *pRevision
    );
```

PLX Chip Support:

All PLX devices

Description:

Returns the PLX chip type and revision if possible.

Parameters:

pDevice
: Pointer to an open device

pChipType
: Pointer to a 16-bit buffer to contain the PLX chip type

pRevision
: Pointer to an 8-bit value to contain the revision

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device

Notes:

The chip type is returned as a hex number matching the chip number. For example, 0x6466 = 6466. For some PLX chips, different revisions are indistinguishable from each other. In the case, the revision will be the latest version.

If the PCI device is not a PLX chip or is not identified properly by the driver, a value of 0 will be returned for the chip type and revision.

Usage:

```
U8         Revision;
U16        ChipType;
PLX_STATUS rc;


rc =
    PlxPci_ChipTypeGet(
        pDevice,
        &ChipType,
        &Revision
        );

if (rc != ApiSuccess)
{
    // Error
}
else
{
    Cons_printf(
        "    Chip type:  %04X\n"
        "    Revision :    %02X\n",
        ChipType, Revision
        );
}
```

PlxPci_ChipTypeSet

Syntax:

```
PLX_STATUS
PlxPci_ChipTypeSet(
    PLX_DEVICE_OBJECT *pDevice,
    U16                ChipType,
    U8                 Revision
    );
```

PLX Chip Support:

All PLX devices

Description:

Sets the PLX chip type and revision to force a specific identification.

Parameters:

pDevice
: Pointer to an open device

ChipType
: The desired PLX chip type, in Hex, or 0 for <Unknown>. Available chip types are 8532, 8524, 8114, etc.

Revision
: The desired revision ID. If the value is 0xFF, the default chip revision will be used, which is usually taken directly from the PCI Revision ID register.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidDeviceInfo	The device object is not a valid PLX device
ApiUnsupportedFunction	The function is not supported by the installed driver (i.e. the device is in Non-Transparent mode)
ApiInvalidData	The *ChipType* parameter was invalid or not a supported type

Notes:

The chip type should be a hex number matching the chip number. For example, 0x6466 = 6466. A value of 0 may be passed to clear the chip type.

When modifying the Device/Vendor ID of a PLX PCI-to-PCI bridge, it is recommended that the PLX driver be modified to properly identify the device. PlxPci_ChipTypeSet is recommended for temporary use only for debug purposes.

Warning: This option is typically used only when a PLX PCI-to-PCI bridge Device/Vendor ID is modified and the PLX PCI Service driver is not able to properly identify the device. Setting the chip type will force the PLX driver, after it is already loaded, to treat the device as a specific PLX chip and enable chip-specific features, such as EEPROM access. Setting the chip type to an incorrect or invalid setting may result in erratic behavior system crashes.

Usage:

```
PLX_STATUS rc;


// Force the chip tpye & revision
rc =
    PlxPci_ChipTypeSet(
        pDevice,
        0x6520,            // 6520 device
        0xCA               // Revision CA
        );

if (rc != ApiSuccess)
{
    // Error
}

// Force the chip tpye, but use default revision
rc =
    PlxPci_ChipTypeSet(
        pDevice,
        0x6152,            // 6152 device
        (U8)-1             // Use default revision
        );

if (rc != ApiSuccess)
{
    // Error
}

// Clear the cuurent type to configure device as “Non-PLX”
rc =
    PlxPci_ChipTypeSet(
        pDevice,
        0,                 // Clear chip type
        0                  // Clear revision
        );

if (rc != ApiSuccess)
{
    // Error
}
```

PlxPci_CommonBufferProperties

Syntax:

```
PLX_STATUS
PlxPci_CommonBufferProperties(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PHYSICAL_MEM  *pMemoryInfo
    );
```

PLX Chip Support:

All PLX devices

Description:

Returns the common buffer properties.

Parameters:

pDevice
    Pointer to an open 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 ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid

Notes:

This function will only return properties of the common buffer. It will not provide a virtual address for the buffer. Use *PlxPci_CommonBufferMap* 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 PLX driver registry options in this 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:

```
PLX_STATUS       rc;
PLX_PHYSICAL_MEM BufferInfo;


// Get the common buffer information
rc =
    PlxPci_CommonBufferProperties(
        pDevice,
        &BufferInfo
        );

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

Cons_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
    );
```

PlxPci_CommonBufferMap

Syntax:

```
PLX_STATUS
PlxPci_CommonBufferMap(
    PLX_DEVICE_OBJECT  *pDevice,
    VOID              **pVa
    );
```

PLX Chip Support:

All PLX devices

Description:

Maps the common buffer into user virtual space and return the base virtual address.

Parameters:

pDevice
: Pointer to an open 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
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
ApiInvalidAddress	Buffer address is invalid
ApiInsufficientResources	Insufficient resources for perform a mapping of the buffer
ApiFailed	Buffer was not allocated properly

Notes:

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 *PlxPci_DeviceClose* to close the device. The virtual address will cease to be valid after closing the device or after unmapping the buffer. Refer to *PlxPci_CommonBufferUnmap*.

Usage:

```
U8                value;
VOID             *pBuffer;
PLX_STATUS        rc;
PLX_PHYSICAL_MEM  BufferInfo;


// Get the common buffer information
rc =
    PlxPci_CommonBufferProperties(
        pDevice,
        &BufferInfo
        );

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

// Map the buffer into user space
rc =
    PlxPci_CommonBufferMap(
        pDevice,
        &pBuffer
        );

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

// Write 32-bit value to buffer
*(U32*)((U8*)pBuffer + 0x100) = 0x12345;

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

PlxPci_CommonBufferUnmap

Syntax:

```
PLX_STATUS
PlxPci_CommonBufferUnmap(
    PLX_DEVICE_OBJECT  *pDevice,
    VOID              **pVa
    );
```

PLX Chip Support:

All PLX devices

Description:

Unmaps the common buffer from user virtual space.

Parameters:

pDevice
: Pointer to an open device

pVa
: The virtual address of the common buffer originally obtained from *PlxPci_CommonBufferMap*

Return Codes:

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

Notes:

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 *PlxPci_DeviceClose* 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;
PLX_STATUS        rc;
PLX_PHYSICAL_MEM  BufferInfo;


// Get the common buffer information
rc =
    PlxPci_CommonBufferProperties(
        pDevice,
        &BufferInfo
        );

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

// Map the buffer into user space
rc =
    PlxPci_CommonBufferMap(
        pDevice,
        &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 =
    PlxPci_CommonBufferUnmap(
        pDevice,
        &pBuffer
        );

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

PlxPci_DeviceClose

Syntax:

```
PLX_STATUS
PlxPci_DeviceClose(
    PLX_DEVICE_OBJECT *pDevice
    );
```

PLX Chip Support:

All devices

Description:

Releases a PLX device object previously opened with PlxPci_DeviceOpen().

Parameters:

pDevice
Pointer to an open device

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened

Usage:

```
PLX_STATUS rc;


// Release the open PLX device
rc =
    PlxPci_DeviceClose(
        pDevice
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to release PLX device
}
```

PlxPci_DeviceOpen

Syntax:

```
PLX_STATUS
PlxPci_DeviceOpen(
    PLX_DEVICE_KEY    *pKey,
    PLX_DEVICE_OBJECT *pDevice
    );
```

PLX Chip Support:

All devices

Description:

Selects a specific PCI device for later use with PLX API calls. The device is selected based on the criteria in PLX_DEVICE_KEY.

Parameters:

pKey
: Pointer to a PLX_DEVICE_KEY structure which contains one or more search criteria.

pDevice
: Pointer to a PLX_DEVICE_OBJECT structure which will describe the selected PCI device.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiInvalidDeviceInfo	The device object is invalid or the key does not match an installed device
ApiInvalidDriverVersion	The PLX driver version does not match the API library version
ApiObjectAlreadyAllocated	The device object is already open or in use

Notes:

Use PlxPci_DeviceFind to query the driver for installed PCI devices and fill in the PLX_DEVICE_KEY information.

If the function returns ApiSuccess, any missing key information will be filled in.

Usage:

```
PLX_STATUS        rc;
PLX_DEVICE_KEY    DeviceKey;
PLX_DEVICE_OBJECT Device;


// Clear key structure to select first device
memset(&DeviceKey, PCI_FIELD_IGNORE, sizeof(PLX_DEVICE_KEY));

// Open device
rc =
    PlxPci_DeviceOpen(
        &DeviceKey,
        &Device
        );

if (rc != ApiSuccess)
{
    // Error
}
else
{
    Cons_printf(
        "Selected: %04x %04x [b:%02x  s:%02x  f:%02x]\n",
        DeviceKey.DeviceId, DeviceKey.VendorId,
        DeviceKey.bus, DeviceKey.slot, DeviceKey.function
        );
}
```

PlxPci_DeviceFind

Syntax:

```
PLX_STATUS
PlxPci_DeviceFind(
    PLX_DEVICE_KEY *pKey,
    U8              DeviceNumber
    );
```

PLX Chip Support:

All devices

Description:

Locates a specific PCIe device and fills in the corresponding device key information.

Parameters:

pKey
: Pointer to a PLX_DEVICE_KEY structure containing the search criteria

DeviceNumber
: The 0-based index of the device number to select. Refer to Notes section below for details.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiInvalidDeviceInfo	The key does not match an installed device

Notes:

The fields in the PLX_DEVICE_KEY structure will be used to locate a device. If a field is set to PCI_FIELD_IGNORE, then it is ignored in the comparison. If a device matches the criteria, all ignored fields in the key will be filled in with their respective value.

The DeviceNumber parameter is an index that specifies which device to select, where '0' is the first device. If multiple devices match the criteria, the DeviceNumber specifies which device to select.

Usage:

```
PLX_STATUS     rc;
PLX_DEVICE_KEY DeviceKey;


// Clear key structure to find first device
memset(&DeviceKey, PCI_FIELD_IGNORE, sizeof(PLX_DEVICE_KEY));

rc =
    PlxPci_DeviceFind(
        &DeviceKey,
        0           // Select 1st device matching criteria
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to locate matching device
}

// Search for the third device matching a specific Vendor ID
memset(&DeviceKey, PCI_FIELD_IGNORE, sizeof(PLX_DEVICE_KEY));

// Specify Vendor ID
DeviceKey.VendorId = 0x10b5;              // PLX Vendor ID

rc =
    PlxPci_DeviceFind(
        &DeviceKey,
        2           // Select 3rd device matching criteria
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to locate matching device
}
```

PlxPci_DeviceFindEx

Syntax:

```
PLX_STATUS
PlxPci_DeviceFindEx(
    PLX_DEVICE_KEY *pKey,
    U8              DeviceNumber,
    PLX_API_MODE    ApiMode,
    PLX_MODE_PROP  *pModeProp,
    U8              DeviceNumber
    );
```

PLX Chip Support:

All devices

Description:

This function is similar to *PlxPci_DeviceFind()* but also supports finding a device using methods other than PCI/PCI Express, such as I^2C.

Parameters:

pKey
: Pointer to a PLX_DEVICE_KEY structure containing the search criteria

DeviceNumber
: The 0-based index of the device number to select. Refer to Notes section below for details.

ApiMode
: Specifies the PLX_API_MODE to use to search for a device. If *ApiMode* is PLX_API_MODE_PCI, this function behaves identical to *PlxPci_DeviceFind()*.

pModeProp
: Contains the properties used for detecting a device. The items used in the structure depend upon the value of the *ApiMode* parameter. For example, if *ApiMode* is PLX_API_MODE_I2C_AARDVARK, then only the ***I2c*** union parameters in the structure are used.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiNoActiveDriver	For PCI mode, a valid PLX driver is not loaded in the system For I^2C mode, the Aardvark USB device does not exist or driver is not installed
ApiInvalidDeviceInfo	The key does not match an installed device
ApiUnsupportedFunction	Attempt to select TCP connection which is not yet supported

Notes:

The fields in the PLX_DEVICE_KEY structure will be used to locate a device. If a field is set to PCI_FIELD_IGNORE, then it is ignored in the comparison. If a device matches the criteria, all ignored fields in the key will be filled in with their respective value.

The DeviceNumber parameter is an index that specifies which device to select, where '0' is the first device. If multiple devices match the criteria, the DeviceNumber specifies which device to select.

For I²C, if the *I2c.SlaveAddr* field is -1 (FFFFh), the API will auto-probe all possible PLX I²C addresses to detect a chip (e.g. 58->5Fh, 68->6Fh, etc).

At this time, the only I²C device supported is the TotalPhase Aardvark USB I²C /SPI tool. Other I²C devices may be supported in future versions of the SDK. The Aardvark USB driver must be loaded for the PLX API to work over I²C.

Connections over TCP/IP are not yet supported in the PLX API. This may be supported in a future version of the SDK.

Usage:

```
PLX_STATUS     rc;
PLX_MODE_PROP  ModeProp;
PLX_DEVICE_KEY DeviceKey;

// Clear key structure to find first device
memset(&DeviceKey, PCI_FIELD_IGNORE, sizeof(PLX_DEVICE_KEY));

// Set I2C properties
ModeProp.I2c.I2cPort   = 0;       // Use the first I2C USB device
ModeProp.I2c.SlaveAddr = -1;      // Auto-probe for PLX chip
ModeProp.I2c.ClockRate = 100;     // Set I2C clock rate in KHz

// Find first I2C PLX device/port
rc =
    PlxPci_DeviceFindEx(
        &DeviceKey,
        0                             // Select 1st device matching criteria
        PLX_API_MODE_I2C_AARDVARK,    // Connect over I2C
        &ModeProp
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to locate matching device
}
```

PlxPci_DeviceReset

Syntax:

```
PLX_STATUS
PlxPci_DeviceReset(
    PLX_DEVICE_OBJECT *pDevice
    );
```

PLX Chip Support:

All PLX 9000 & 8311 devices

Description:

Resets the selected PLX device

Parameters:

pDevice
Pointer to an open PCI device

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiUnsupportedFunction	Reset of the selected device is not supported

Usage:

```
PLX_DEVICE_OBJECT  Device;


// Issue reset to PLX device
PlxPci_DeviceReset(
    pDevice
    );
```

PlxPci_DmaChannelOpen

Syntax:

```
PLX_STATUS
PlxPci_DmaChannelOpen(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_PROP      *pDmaProp
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Opens and initializes a DMA channel to prepare for later transfers. Starting with SDK 6.10, it is recommended to set the *pDmaProp* parameter to NULL and use other PLX APIs to retrieve and update DMA properties. Refer to PlxPci_DmaGetProperties & PlxPci_DmaSetProperties.

Parameters:

pDevice
: Pointer to an open device

channel
: The number of the DMA channel to open

pDmaProp
: Pointer to a structure containing the properties to use for initializing the DMA channel. If this NULL, the DMA properties will not be modified.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 in use by another process

Usage:

```
// Open the DMA channel
PlxPci_DmaChannelOpen(
    pDevice,
    0,              // Channel 0
    NULL            // Do not modify current DMA properties
    );
```

PlxPci_DmaChannelClose

Syntax:

```
PLX_STATUS
PlxPci_DmaChannelClose(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Closes a previously opened DMA channel

Parameters:

pDevice
    Pointer to an open PCI device

channel
    The DMA channel number to close

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 previously opened by the caller
ApiDmaInProgress	A DMA transfer is in progress
ApiDmaPaused	The DMA channel is paused
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Notes:

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. For PLX 9000 series devices, refer to PlxPci_DeviceReset.

Usage:

```
PLX_STATUS rc;


rc =
    PlxPci_DmaBlockChannelClose(
        pDevice,
        1            // Channel 1
        );

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

        // Attempt to close again
        PlxPci_DmaChannelClose(
            pDevice,
            1
            );
    }
}
```

PlxPci_DmaGetProperties

Syntax:

```
PLX_STATUS
PlxPci_DmaGetProperties(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_PROP      *pDmaProp
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Returns the current DMA properties for a DMA channel

Parameters:

pDevice
: Pointer to an open device

channel
: The DMA channel number to access

pDmaProp
: Pointer to a structure that will contain the DMA properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 previously opened by the caller

Notes:

A DMA channel must first be opened by the caller with PlxPci_DmaChannelOpen before this function can be called.

Usage:

```
PLX_DMA_PROP DmaProp;


// Get current DMA properties
PlxPci_DmaGetProperties(
    pDevice,
    0,              // DMA channel 0
    &DmaProp
    );

// Modify desired properties based on chip type
if ((PlxChip & 0xFF00) == 0x8600) || (PlxChip & 0xFF00) == 0x8700))
{
    // Use relaxed ordering for data read requests
    DmaProp.RelOrderDataReadReq = 1;

    // Support 128B read request TLPs
    DmaProp.MaxSrcXferSize = PLX_DMA_MAX_SRC_TSIZE_128B;
}
else
{
    // Enable READY# input and burst of 4 DWORDS
    DmaProp.ReadyInput    = 1;
    DmaProp.Burst         = 1;
    DmaProp.BurstInfinite = 0;
}

// Update DMA with new properties
PlxPci_DmaSetProperties(
    pDevice,
    0,              // DMA channel 0
    &DmaProp
    );
```

PlxPci_DmaSetProperties

Syntax:

```
PLX_STATUS
PlxPci_DmaSetProperties(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_PROP      *pDmaProp
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Updates the DMA properties for a DMA channel

Parameters:

pDevice
: Pointer to an open device

channel
: The DMA channel number to access

pDmaProp
: Pointer to a structure containing the DMA properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 previously opened by the caller
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Notes:

A DMA channel must first be opened by the caller with PlxPci_DmaChannelOpen before this function can be called.

Usage:

```
PLX_DMA_PROP DmaProp;


// Fill in current DMA properties
PlxPci_DmaGetProperties(
    pDevice,
    0,              // DMA channel 0
    &DmaProp
    );

// Modify desired properties based on chip type
if ((PlxChip & 0xFF00) == 0x8600) || (PlxChip & 0xFF00) == 0x8700))
{
    // Use relaxed ordering for data read requests
    DmaProp.RelOrderDataReadReq = 1;

    // Support 128B read request TLPs
    DmaProp.MaxSrcXferSize = PLX_DMA_MAX_SRC_TSIZE_128B;
}
else
{
    // Enable READY# input and burst of 4 DWORDS
    DmaProp.ReadyInput    = 1;
    DmaProp.Burst         = 1;
    DmaProp.BurstInfinite = 0;
}

// Update DMA with new properties
PlxPci_DmaSetProperties(
    pDevice,
    0,              // DMA channel 0
    &DmaProp
    );
```

PlxPci_DmaControl

Syntax:

```
PLX_STATUS
PlxPci_DmaControl(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_COMMAND    command
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Controls the DMA engine for a given DMA channel.

Parameters:

pDevice
: Pointer to an open device

channel
: The DMA channel number to control

command
: The action to perform on the DMA channel. Refer to PLX_DMA_COMMAND for the list of valid DMA commands.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 was not previously opened by the caller
ApiDmaInProgress	If attempting to resume a DMA channel that is not in a paused state.
ApiDmaCommandInvalid	An invalid or unsupported DMA command
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Notes:

A DMA channel must first be opened by the caller with PlxPci_DmaChannelOpen before this function can be called.

Usage:

```
PLX_STATUS     rc;
PLX_DMA_PARAMS DmaParams;


// Start a DMA transfer
PlxPci_DmaTransferBlock(
    pDevice,
    0,               // Channel 0
    &DmaParams,
    0                // Don't wait for DMA completion
    );

// Pause the DMA channel
rc =
    PlxPci_DmaControl(
        pDevice,
        0,            // Channel 0
        DmaPause      // Pause the current transfer
        );

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

// Resume the DMA channel
rc =
    PlxPci_DmaControl(
        pDevice,
        0,            // Channel 0
        DmaResume     // Resume the transfer
        );

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

PlxPci_DmaStatus

Syntax:

```
PLX_STATUS
PlxPci_DmaStatus(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

Description:

Returns the status of the specified DMA channel.

Parameters:

pDevice
: Pointer to an open device

channel
: The DMA channel number to check status of

Return Codes:

Code	Description
ApiInvalidDeviceInfo	The device object is not valid
ApiDmaChannelInvalid	The DMA channel is not supported by this PLX chip
ApiDmaDone	The DMA channel is done/ready
ApiDmaPaused	The DMA channel is paused
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Usage:

```
PLX_STATUS     rc;
PLX_DMA_PARAMS DmaParams;


// Start a DMA transfer
PlxPci_DmaTransferBlock(
    pDevice,
    0,               // Channel 0
    &DmaParams,
    0                // Don't wait for DMA completion
    );

// Poll until DMA completes
do
{
    rc =
        PlxPci_DmaStatus(
            pDevice,
            0,         // Channel 0
            );
}
while (rc == ApiDmaInProgress);
```

PlxPci_DmaTransferBlock

Syntax:

```
PLX_STATUS
PlxPci_DmaTransferBlock(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_PARAMS    *pDmaParams,
    U64                Timeout_ms
    );
```

PLX Chip Support:

9054, 9056, 9080*, 9656, 8311, & 8000 DMA

Description:

Starts a Block DMA transfer for a given DMA channel.

Parameters:

pDevice
: Pointer to an open device

channel
: The open DMA channel number to use for the transfer

pDmaParams
: A pointer to a structure containing the DMA transfer parameters

Timeout_ms
: Specifies the timeout, in milliseconds, for the function to wait for DMA completion.

  If 0, the API returns immediately after starting the DMA transfer and does not wait for its completion.

  To have the function wait indefinitely for DMA completion, use the value ***PLX_TIMEOUT_INFINITE***.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 previously opened by the caller
ApiDmaInProgress	A DMA transfer is currently in-progress
ApiWaitTimeout	No interrupt was received to signal DMA completion
ApiUnsupportedFunction	The device does not support DMA or 64-bit DMA is required but not supported (9080)
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Notes:

Block DMA transfers are useful with contiguous host buffers described by a *PCI address*. The DMA channel requires a valid PCI physical addresses, not user or virtual address. 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 PlxPci_CommonBufferProperties*), for example, is a valid DMA buffer.

By default, the DMA done interrupt is automatically enabled when this function is called. It may be disabled by setting the *bIgnoreBlockInt* field of PLX_DMA_PARAMS. In this case, the DMA interrupt is disabled and will not trigger the PLX driver's Interrupt Service Routine (ISR). This also means DMA done notification events registered with *PlxPci_NotificationRegisterFor* will not signal when the DMA has completed.

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

PLX_DMA_PARAMS:

Structure Element	Description
UserVa	Ignored.
AddrSource	(8000 DMA) Source PCI address
AddrDest	(8000 DMA) Destination PCI address
PciAddr	(9000 DMA) The PCI address to transfer to/from. 64-bit is supported
LocalAddr	(9000 DMA) The Local address for the transfer
ByteCount	The number of bytes to transfer.
Direction	(8000 DMA) Ignored. *AddrSource* & *AddrDest* fields inherently imply transfer direction (9000 DMA) Direction of the transfer. Refer to PLX_DMA_DIR
bConstAddrSrc	(8000 DMA) Keeps the source address constant
bConstAddrDest	(8000 DMA) Keeps the destination address constant
bForceFlush	(8000 DMA) DMA engine will issue a Zero-length TLP to flush final writes.
bIgnoreBlockInt	Will disable the DMA done interrupt. API DMA done notification will timeout in this case.

Usage:

```
PLX_DMA_PARAMS   DmaParams;
PLX_PHYSICAL_MEM PciBuffer;


// Get Common buffer information
PlxPci_CommonBufferProperties(
    pDevice,
    &PciBuffer
    );

memset( &DmaParams, 0, sizeof(PLX_DMA_PARAMS) );

// Fill in DMA transfer parameters
DmaParams.TransferCount = 0x1000;

if (pDevObj->Key.PlxChipFamily == PLX_FAMILY_BRIDGE_P2L)
{
    // 9000/8311 DMA
    DmaParams.PciAddr   = PciBuffer.PhysicalAddr;
    DmaParams.LocalAddr = 0x0;
    DmaParams.Direction = PLX_DMA_LOC_TO_PCI;
}
else
{
    // 8000 DMA
    DmaParams.AddrSource = PciBuffer.PhysicalAddr;
    DmaParams.AddrDest   = PciBuffer.PhysicalAddr + 0x5000;
}

rc =
    PlxPci_DmaTransferBlock(
        pDevice,
        0,               // Channel 0
        &DmaParams,      // DMA transfer parameters
        (3 * 1000)       // Specify time to wait for DMA completion
        );

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

PlxPci_DmaTransferUserBuffer

Syntax:

```
PLX_STATUS
PlxPci_DmaTransferUserBuffer(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 channel,
    PLX_DMA_PARAMS    *pDmaParams,
    U64                Timeout_ms
    );
```

PLX Chip Support:

9054, 9056, 9080, 9656, 8311, & 8000 DMA

** On some versions of Windows (e.g. 2003 Server) or system with more than 4GB of RAM, the physical address of some user mode buffer pages may require 64-bit addressing. If this is detected, the PLX driver will automatically use features in the PLX chip to access these pages. For legacy PCI DMA chips, PCI dual-addressing is enabled. For newer PCI Express switch DMA, extended descriptors are used as needed. Dual-addressing is not supported on the PLX 9080 device; therefore, the API will return an error if 64-bit is required with this device.*

Description:

Transfers a user-supplied buffer using the DMA channel. SGL mode of the DMA channel is used, but this is transparent to the application. The 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 and the driver will then perform all cleanup tasks.

Parameters:

pDevice
: Pointer to an open device

channel
: The open DMA channel number to use for the transfer

pDmaParams
: A pointer to a structure containing the DMA transfer parameters

Timeout_ms
: Specifies the timeout, in milliseconds, for the function to wait for DMA completion.

  If 0, the API returns immediately after starting the DMA transfer and does not wait for its completion.

  To have the function wait indefinitely for DMA completion, use the value ***PLX_TIMEOUT_INFINITE***.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
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 previously opened by the caller
ApiDmaInProgress	The DMA transfer is currently in-progress
ApiWaitTimeout	No interrupt was received to signal DMA completion
ApiDmaSglPagesGetError	The driver was unable to obtain the page list for the user- mode buffer
ApiDmaSglPagesLockError	The driver was unable to page lock the user-mode buffer
ApiInsufficientResources	The driver was unable to allocate an internal buffer to store SGL descriptors
ApiDeviceInUse	The DMA channel is open but owned by another calling thread or process

Notes:

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

PLX_DMA_PARAMS:

Structure Element	Description
UserVa	Virtual address of the user-mode buffer to transfer
AddrSource	Ignored
AddrDest	Ignored
PciAddr	(9000 DMA) Ignored (8000 DMA) Specifies the PCI address to transfer to/from, depending upon ***Direction***
LocalAddr	(9000 DMA) The Local address for the transfer
ByteCount	The number of bytes to transfer
Direction	Direction of the transfer. Refer to PLX_DMA_DIR
bConstAddrSrc	(8000 DMA) Keeps the source address constant
bConstAddrDest	(8000 DMA) Keeps the destination address constant
bForceFlush	(8000 DMA) DMA engine will issue a Zero-length TLP to flush final writes.
bIgnoreBlockInt	Ignored. PLX driver always enables DMA done interrupt to cleanup SGL

Usage:

```
U8             *pBuffer;
PLX_DMA_PARAMS  DmaParams;


// Allocate a 500k buffer
pBuffer = malloc(500 * 1024);

// Clear DMA parameters
memset( DmaParams, 0, sizeof(PLX_DMA_PARAMS) );

// Setup DMA parameters (9000 DMA)
DmaParams.UserVa    = (PLX_UINT_PTR)pBuffer;
DmaParams.ByteCount = (500 * 1024);

if (pDevObj->Key.PlxChipFamily == PLX_FAMILY_BRIDGE_P2L)
{
    // 9000/8311 DMA
    DmaParams.LocalAddr = 0x0;
    DmaParams.Direction = PLX_DMA_LOC_TO_PCI;
}
else
{
    // 8000 DMA
    DmaParams.PciAddr   = 0x1F000000;
    DmaParams.Direction = PLX_DMA_PCI_TO_USER;
}

rc =
    PlxPci_DmaTransferUserBuffer(
        pDevice,
        0,               // Channel 0
        &DmaParams,      // DMA transfer parameters
        (3 * 1000)       // Specify time to wait for DMA completion
        );

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

PlxPci_DriverProperties

Syntax:

```
PLX_STATUS
PlxPci_DriverProperties(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_DRIVER_PROP   *pDriverProp
    );
```

PLX Chip Support:

All devices

Description:

Returns properties of the PLX driver in use for the selected device

Parameters:

pDevice
: Pointer to an open device

pDriverProp
: A pointer to PLX_DRIVER_PROP structure that will contain the driver properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid

Usage:

```
PLX_STATUS        rc;
PLX_DRIVER_PROP   DriverProp;
PLX_DEVICE_OBJECT Device;


// Determine if Service or PnP driver in use
rc =
    PlxPci_DriverProperties(
        &Device,
        &DriverProp
        );

if (rc == ApiSuccess)
{
    Cons_printf(
        “Driver Properties:\n
        “   Version  : %d.%02d\n”
        “   Name     : %s\n”
        “   Full Name: %s\n”,
        DriverProp Version,
        DriverProp.Name,
        DriverProp.FullName
        );

    if (DriverProp.bIsServiceDriver)
    {
        Cons_printf(“Using PLX Service driver\n”,);
    }
    else
    {
        Cons_printf(“Using PLX PnP driver\n”,);
     }

    Cons_printf(
        “PCIe Located at 0x%qX\n”,
        DriverProp.AcpiPcieEcam
        );
}
```

PlxPci_DriverScheduleRescan

Syntax:

```
PLX_STATUS
PlxPci_DriverScheduleRescan(
    PLX_DEVICE_OBJECT *pDevice
    );
```

Note: This function has not yet been implemented in the PLX SDK. This documentation is left here for a future SDK version when it is implemented. This function and its parameters are subject to change.

PLX Chip Support:

Any device when selected via the PLX PCI/PCIe Service driver

Description:

Makes a request to the PLX PCI Service driver to rescan the PCI/PCIe bus and rebuild its internal device list. Since the Service driver is not informed of Plug 'n' Play events (e.g. device additional/removal or resource changes), its internal list of detected devices could contain erroneous information.

Once the driver receives the request, it will perform the operation when all connections to it have been closed.

Parameters:

pDevice
Pointer to an open device

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid
ApiUnsupportedFunction	The function was called with a device that is not accessed via the Service driver

Usage:

```
PLX_STATUS status;


// Inform the service driver to rebuild its internal list
status =
    PlxPci_DriverScheduleRescan(
        pDevice
        );

// Close device to allow driver to rescan
PlxPci_DeviceClose(
    pDevice
    );
```

PlxPci_DriverVersion

Syntax:

```
PLX_STATUS
PlxPci_DriverVersion(
    PLX_DEVICE_OBJECT *pDevice,
    U8                *pVersionMajor,
    U8                *pVersionMinor,
    U8                *pVersionRevision
    );
```

PLX Chip Support:

All devices

Description:

Returns the PLX driver version information

Parameters:

pDevice
: Pointer to an open device

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

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

pVersionRevision
: 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
ApiInvalidDeviceInfo	The device object is not valid

Usage:

```
U8         DriverMajor;
U8         DriverMinor;
U8         DriverRevision;
PLX_STATUS rc;


rc =
    PlxPci_DriverVersion(
        pDevice,
        &DriverMajor,
        &DriverMinor,
        &DriverRevision
        );

if (rc != ApiSuccess)
{
    // ERROR – Unable to get Driver version information
}
else
{
    Cons_printf(
        "PLX Driver Version = %d.%d%d\n",
        DriverMajor, DriverMinor, DriverRevision
        );
}
```

PlxPci_EepromPresent

Syntax:

```
PLX_EEPROM_STATUS
PlxPci_EepromPresent(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All PLX devices

Description:

Returns the state of the EEPROM as reported by the PLX device.

Parameters:

pDevice
    Pointer to an open device

pStatus
    Pointer to a PLX_STATUS variable to hold the status. (May be NULL)

Return Codes:

If the function is successful, it will return a PLX_EEPROM_STATUS code.

If the PLX_STATUS variable is not NULL, one of the following values is returned:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	EEPROM access to device is not supported

Notes:

The EEPROM status is read directly from the PLX status register. The status is generally only valid at the time of power up or after a reset. The status may not reflect the true status of the EEPROM after reset. Modifications of EEPROM values, including the CRC, are not reflected in the chip's EEPROM status until the next reset when the EEPROM contents are loaded.

Usage:

```
PLX_STATUS        rc;
PLX_EEPROM_STATUS EepStatus;


// Check if EEPROM present
EepStatus =
    PlxPci_EepromPresent(
        pDevice,
        &rc
        );

if (rc == ApiSuccess)
{
    switch (EepStatus)
    {
        case PLX_EEPROM_STATUS_NONE:
            // No EEPROM Present
            break;

        case PLX_EEPROM_STATUS_VALID:
            // EEPROM present with valid data
            break;

        case PLX_EEPROM_STATUS_INVALID_DATA:
        case PLX_EEPROM_STATUS_BLANK:
        case PLX_EEPROM_STATUS_CRC_ERROR:
            // Present but invalid data, CRC error, or blank
            break;
    }
}
```

PlxPci_EepromProbe

Syntax:

```
BOOLEAN
PlxPci_EepromProbe (
    PLX_DEVICE_OBJECT *pDevice,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All PLX devices

Description:

Manually probes for the presence of an EEPROM. The API does this by writing to a specific EEPROM location and then reading it back to verify the write operation.

Parameters:

pDevice
Pointer to an open device

pStatus
Pointer to a PLX_STATUS variable to hold the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiUnsupportedFunction	EEPROM access to device is not supported

Usage:

```
BOOLEAN    bEepromPresent;
PLX_STATUS rc;


bEepromPresent =
    PlxPci_EepromProbe (
        pDevice,
        &rc
        );

if (rc == ApiSuccess)
{
    if (bEepromPresent)
        // Programmed EEPROM exists
    else
        // EEPROM does not exist
}
```

PlxPci_EepromCrcGet

Syntax:

```
BOOLEAN
PlxPci_EepromCrcGet(
    PLX_DEVICE_OBJECT *pDevice,
    U32               *pCrc,
    U8                *pCrcStatus
    );
```

PLX Chip Support:

All PLX 8000 devices with an EEPROM CRC feature

Description:

Reads the current CRC value from the EEPROM. The status of the CRC as reported by the PLX chip is returned.

Parameters:

pDevice
: Pointer to an open device

pCrc
: Pointer to a 32-bit buffer to contain the current CRC

pCrcStatus
: Pointer to an 8-bit buffer to store the CRC status as reported by the PLX chip. The status code will be ***PLX_CRC_VALID*** or ***PLX_CRC_INVALID***.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiUnsupportedFunction	EEPROM access to device is not supported

Notes:

Note that the CRC status is simply the status as reported by the PLX chip. This status may not be consistent with the EEPROM CRC if the EEPROM has been updated. The status of the CRC in the PLX chip is updated only upon power up when the PLX chip loads values from the EEPROM.

Usage:

```
U8  CrcStatus;
U32 Crc;


// Get current EEPROM CRC
PlxPci_EepromCrcGet(
    pDevice,
    &Crc,
    &CrcStatus
    );

Cons_printf(
    "CRC=%08x  Status=%s)\n",
    Crc,
    (CrcStatus == PLX_CRC_VALID) ? "Valid" : "Invalid"
    );
```

PlxPci_EepromCrcUpdate

Syntax:

```
BOOLEAN
PlxPci_EepromCrcUpdate(
    PLX_DEVICE_OBJECT *pDevice,
    U32               *pCrc,
    BOOLEAN            bUpdateEeprom
    );
```

PLX Chip Support:

All PLX 8000 devices with a CRC feature

Description:

Reads the current EEPROM contents and calculates an updated CRC. If requested, this function can update the CRC stored in the EEPROM.

Parameters:

pDevice
: Pointer to an open device

pCrc
: Pointer to a 32-bit buffer to contain the newly calculated CRC

bUpdateEeprom
: If TRUE, the function will update the CRC in the EEPROM. If FALSE, it will not modify the EEPROM contents.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiUnsupportedFunction	EEPROM access to device is not supported

Usage:

```
U8              CrcStatus;
U32             Crc;
U32             CrcNew;


// Get current EEPROM CRC
PlxPci_EepromCrcGet(
    pDevice,
    &Crc,
    &CrcStatus
    );

// Calculate new CRC
PlxPci_EepromCrcUpdate(
    pDevice,
    &CrcNew,
    FALSE       // Don't update EEPROM
    );

if (Crc == CrcNew)
{
    Cons_printf("CRC in EEPROM is valid\n”);
}
else
{
    Cons_printf("CRCs do not match, CRC in EEPROM not valid\n");

    // Calculate new CRC
    PlxPci_EepromCrcUpdate(
        pDevice,
        &CrcNew,
        TRUE       // Update CRC in EEPROM
        );

    Cons_printf("Updated CRC in EEPROM to valid value\n");
}
```

PlxPci_EepromSetAddressWidth

Syntax:

```
PLX_STATUS
PlxPci_EepromSetAddressWidth(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 width
    );
```

PLX Chip Support:

8111, 8112, & 8000 devices that support EEPROM address width override

Description:

Sets the EEPROM addressing width

Parameters:

pDevice
    Pointer to an open device

width
    The byte addressing to be used for EEPROM accesses. Width must by 1, 2, or 3.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	Device does not support EEPROM address width override
ApiInvalidData	The EEPROM width is not valid

Notes:

Note that this setting only remains persistent as long as the PLX driver is loaded. If it is unloaded or the system is restarted, this API call must be called again.

Usage:

```
U32        Value;
PLX_STATUS rc;


// Get EEPROM width from device
Value =
    PlxPci_PlxRegisterRead(
        pDevice,
        0x1004,      // EEPROM Control register
        &rc
        );

// Get EEPROM address width field (bits 23 & 24)
Value = (Value >> 23) & 0x3;

if (Value == 0)
{
    // EEPROM width not detected, set it manually
    PlxPci_EepromSetAddressWidth(
        pDevice,
        2             // Use 2-byte addressing
        };
}

// EEPROM can now be properly accessed
PlxPci_EepromReadByOffset(
    pDevice,
    0x10,
    &Value;
    );
```

PlxPci_EepromReadByOffset

Syntax:

```
PLX_STATUS
PlxPci_EepromReadByOffset(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U32               *pValue
    );
```

PLX Chip Support:

All PLX devices

Description:

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

Parameters:

pDevice
: Pointer to an open device

offset
: The EEPROM offset of the location to read. (Must be aligned on a 32-bit boundary)

pValue
: Pointer to a 32-bit buffer to contain the EEPROM value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	EEPROM access to device is not supported
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiInvalidOffset	Offset not aligned on 32-bit boundary

Usage:

```
U32        EepromData;
PLX_STATUS status;

// Read the Subsystem Device ID of the 9054
status =
    PlxPci_EepromReadByOffset(
        pDevice,
        0x44,            // Subsystem Device ID EEPROM offset
        &EepromData
        );

if (status != ApiSuccess)
    // ERROR – Unable to read EEPROM
```

PlxPci_EepromWriteByOffset

Syntax:

```
PLX_STATUS
PlxPci_EepromWriteByOffset(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U32                value
    );
```

PLX Chip Support:

All PLX devices

Description:

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

Parameters:

pDevice
: Pointer to an open device

offset
: The EEPROM offset of the location to write. (Must be aligned on a 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	EEPROM access to device is not supported
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiInvalidOffset	Offset not aligned on 32-bit boundary

Usage:

```
PLX_STATUS status;

// Write to the Subsystem Device ID of the 9054
status =
    PlxPci_EepromWriteByOffset(
        pDevice,
        0x44,            // Subsystem Device ID EEPROM offset
        0x524510B5
        );

if (status != ApiSuccess)
    // ERROR – Unable to write to EEPROM
```

PlxPci_EepromReadByOffset_16

Syntax:

```
PLX_STATUS
PlxPci_EepromReadByOffset_16(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U16               *pValue
    );
```

PLX Chip Support:

All PLX devices

Description:

Reads a 16-bit value from a specified offset from the configuration EEPROM connected to the PLX chip

Parameters:

pDevice
: Pointer to an open device

offset
: The EEPROM offset of the location to read. (Must be aligned on a 16-bit boundary)

pValue
: Pointer to a 16-bit buffer to contain the EEPROM value

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	EEPROM access to device is not supported
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiInvalidOffset	Offset not aligned on 16-bit boundary

Usage:

```
U16        EepromData;
PLX_STATUS status;

// Read the Subsystem Device ID of the 6540
status =
    PlxPci_EepromReadByOffset_16(
        pDevice,
        0x26,            // Subsystem Device ID EEPROM offset
        &EepromData
        );

if (status != ApiSuccess)
    // ERROR – Unable to read EEPROM
```

PlxPci_EepromWriteByOffset_16

Syntax:

```
PLX_STATUS
PlxPci_EepromWriteByOffset_16(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U16                value
    );
```

PLX Chip Support:

All PLX devices

Description:

Writes a 16-bit value to a specified offset of the EEPROM connected to the PLX chip

Parameters:

pDevice
: Pointer to an open device

offset
: The EEPROM offset of the location to write. (Must be aligned on a 16-bit boundary)

value
: The 16-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	EEPROM access to device is not supported
ApiWaitTimeout	The PLX EEPROM controller is busy and not accepting new commands
ApiInvalidOffset	Offset not aligned on 16-bit boundary

Usage:

```
PLX_STATUS status;

// Write to the Subsystem Device ID of the 9054
status =
    PlxPci_EepromWriteByOffset_16(
        pDevice,
        0x44,            // Subsystem Device ID EEPROM offset
        0x5245
        );

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

PlxPci_GetPortProperties

Syntax:

```
PLX_STATUS
PlxPci_GetPortProperties(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PORT_PROP     *pPortProp
    );
```

PLX Chip Support:

All devices

Description:

Returns properties of the PLX driver in use for the selected device

Parameters:

pDevice
Pointer to an open device

pPortProp
A pointer to PLX_PORT_PROP structure that will contain the port properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not valid

Usage:

```
PLX_PORT_PROP PortProp;


PlxPci_GetPortProperties(
    pDevice,
    &PortProp
    );

Cons_printf("Port Type  : %02d ", PortProp.PortType);

switch (PortProp.PortType)
{
    case PLX_PORT_ENDPOINT:  // PLX_PORT_NON_TRANS
        Cons_printf("(Endpoint or NT port)\n");
        break;

    case PLX_PORT_UPSTREAM:
        Cons_printf("(Upstream)\n");
        break;

    case PLX_PORT_DOWNSTREAM:
```

```
            Cons_printf("(Downstream)\n");
            break;

        case PLX_PORT_LEGACY_ENDPOINT:
            Cons_printf("(Endpoint)\n");
            break;

        case PLX_PORT_ROOT_PORT:
            Cons_printf("(Root Port)\n");
            break;

        case PLX_PORT_PCIE_TO_PCI_BRIDGE:
            Cons_printf("(PCIe-to-PCI Bridge)\n");
            break;

        case PLX_PORT_PCI_TO_PCIE_BRIDGE:
            Cons_printf("(PCI-to-PCIe Bridge)\n");
            break;

        case PLX_PORT_ROOT_ENDPOINT:
            Cons_printf("(Root Complex Endpoint)\n");
            break;

        case PLX_PORT_ROOT_EVENT_COLL:
            Cons_printf("(Root Complex Event Collector)\n");
            break;

        case PLX_PORT_UNKNOWN:
        default:
            Cons_printf("(Unknown?)\n");
            break;
    }

    Cons_printf("Port Number: %02d\n", PortProp.PortNumber);
    Cons_printf("Max Payload: %02d\n", PortProp.MaxPayloadSize);
    Cons_printf("Link Width : %d\n", PortProp.LinkWidth);
}
```

PlxPci_I2cGetPorts

Syntax:

```
PLX_STATUS
PlxPci_I2cGetPorts(
    PLX_API_MODE  ApiMode,
    U32          *pI2cPorts
    );
```

PLX Chip Support:

All devices

Description:

Returns the I^2C ports detected in the system and their availability.

Parameters:

ApiMode

Specifies the PLX_API_MODE to use. At this time, only PLX_API_MODE_I2C_AARDVARK is supported.

pI2cPorts

A 32-bit value containing information about the I^2C ports in the system. Bits [15:0] denote whether the specific port is in the system and bits [31:16] denote whether the port is in-use.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidAccessType	The ApiMode parameter is not PLX_API_MODE_I2C_AARDVARK
ApiNoActiveDriver	The Aardvark USB device does not exist or driver is not installed

Usage:

```
U8         i;
U32        I2cPorts;
PLX_STATUS status;

// Get available I2C ports
status =
    PlxPci_I2cGetPorts(
        PLX_API_MODE_I2C_AARDVARK,
        &I2cPorts
        );

if ((status != ApiSuccess) || (I2cPorts == 0))
{
    // No I2C ports detected
}
else
{
    // Parse through active ports
    for (i = 0; i < 16; i++)
    {
        // Check if port is active
        if (I2cPorts & (1 << i))
        {
            // Port exists in the system

            // Check if port is in-use
            if ((I2cPorts >> 16) & (1 << i))
            {
                // Port is in use by another application
            }
        }
    }
}
```

PlxPci_I2cVersion

Syntax:

```
PLX_STATUS
PlxPci_I2cVersion (
    U16          I2cPort,
    PLX_VERSION *pVersion
    );
```

PLX Chip Support:

All devices

Description:

Returns the version information for a specific I^2C port.

Parameters:

I2cPort
: Specifies the I^2C port.

pVersion
: A pointer to a.PLX_VERSION structure that will contain version information.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidAccessType	The ApiMode parameter is not PLX_API_MODE_I2C_AARDVARK
ApiNoActiveDriver	The Aardvark USB device does not exist or driver is not installed

Usage:

```
PLX_STATUS  status;
PLX_VERSION I2cVersion;


// Get I2C version
status =
    PlxPci_I2cVersion(
        0,                // I2C USB device
        &I2cVersion
        );

if (status != ApiSuccess)
{
    // Error – Unable to get I2C version information
}
else
{
    Cons_printf(
        “I2C Version Info:\n”
        “  API:v%01d.%02d SW:v%01d.%02d  FW:v%01d.%02d  HW:v%01d.%02d\n”,
        (I2c.ApiLibrary >> 8), I2c.ApiLibrary & 0xFF,
        (I2c.Software >> 8), I2c.Software & 0xFF,
        (I2c.Firmware >> 8), I2c.Firmware & 0xFF,
        (I2c.Hardware >> 8), I2c.Hardware & 0xFF,

    // Verify required versions
    if (I2c.SwReqByFw < I2c.Software)
        Cons_printf(“Error: I2C SW ver is not compatible with FW version\n”);

    if (I2c.FwReqBySw < I2c.Firmware)
        Cons_printf(“Error: I2C FW ver is not compatible with SW version\n”);

    if (I2c.ApiReqBySw < I2c.ApiLibrary)
        Cons_printf(“Error: I2C API ver is not compatible with SW version\n”);
}
```

PlxPci_IoPortRead

Syntax:

```
PLX_STATUS
PlxPci_IoPortRead(
    PLX_DEVICE_OBJECT *pDevice,
    U64                port,
    VOID              *pBuffer,
    U32                ByteCount,
    PLX_ACCESS_TYPE    AccessType
    );
```

PLX Chip Support:

All devices

Description:

Reads one or more values from an I/O port.

Parameters:

pDevice
: Pointer to an open device

port
: The I/O port address to read from. Must be a multiple of the AccessType.

pBuffer
: A pointer to a buffer that will contain the data read from the I/O port

ByteCount
: The number of bytes to read from the I/O port. Must be a multiple of the 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	The I/O port is not aligned on a boundary that is a multiple of the *AccessType*.
ApiInvalidAccessType	An invalid or unsupported PLX_ACCESS_TYPE parameter
ApiInvalidSize	The region to access is not a valid I/O port or the I/O port is not aligned on a boundary that is a multiple of the *AccessType*.

Usage:

```
U8         MyBuffer[0x100];
PLX_STATUS rc;


// Read from an I/O port
rc =
    PlxPci_IoPortRead(
        pDevice,
        200h,             // Specify I/O port base
        &MyBuffer,        // Buffer to place data into
        0x100,            // Number of bytes to read
        BitSize8          // Perform 8-bit reads
        );

if (rc != ApiSuccess)
{
    // ERROR - Unable to read from I/O port
}
```

PlxPci_IoPortWrite

Syntax:

```
PLX_STATUS
PlxPci_IoPortWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U64                port,
    VOID              *pBuffer,
    U32                ByteCount,
    PLX_ACCESS_TYPE    AccessType
    );
```

PLX Chip Support:

All devices

Description:

Writes one or more values to an I/O port.

Parameters:

pDevice
: Pointer to an open device

port
: The I/O port address to write to. Must be aligned on an *AccessType* boundary.

pBuffer
: A pointer to a buffer that contains the data to write to the I/O port

ByteCount
: The number of bytes to write to the I/O port. Must be a multiple of the 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	The I/O port is not aligned on a boundary that is a multiple of the *AccessType*.
ApiInvalidAccessType	An invalid or unsupported PLX_ACCESS_TYPE parameter
ApiInvalidSize	The region to access is not a valid I/O port or the I/O port is not aligned on a boundary that is a multiple of the *AccessType*.

Usage:

```
U8         MyBuffer[0x100];
PLX_STATUS rc;


// Read from an I/O port
rc =
    PlxPci_IoPortWrite(
        pDevice,
        200h,             // Specify I/O port base
        &MyBuffer,        // Buffer that contains write data
        0x100,            // Number of bytes to write
        BitSize16         // Perform 16-bit writes
        );

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

PlxPci_InterruptDisable

Syntax:

```
PLX_STATUS
PlxPci_InterruptDisable(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_INTERRUPT     *pPlxIntr
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Disables PLX-specific interrupt(s)

Parameters:

pDevice
: Pointer to an open device

pPlxIntr
: A pointer to the interrupt structure specifying the interrupts to disable

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiPowerDown	The PLX device is in a power state that is lower than required for this function

Usage:

```
PLX_STATUS    rc;
PLX_INTERRUPT PlxIntr;


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

// Set interrupts to disable
PlxIntr.LocalToPci_1 = 1;   // Generic Local-to-PCI int (LINT#)
PlxIntr.DmaChannel_0 = 1;   // PCI DMA Channel 0

rc =
    PlxPci_InterruptDisable(
        pDevice,
        &PlxIntr
        );

if (rc != ApiSuccess)
    // ERROR - Unable to disable interrupts
```

PlxPci_InterruptEnable

Syntax:

```
PLX_STATUS
PlxPci_InterruptEnable(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_INTERRUPT     *pPlxIntr
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Enables PLX-specific interrupt(s)

Parameters:

pDevice
: Pointer to an open device

pPlxIntr
: A pointer to the interrupt structure specifying the interrupts to enable

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiPowerDown	The PLX device is in a power state that is lower than required for this function

Usage:

```
PLX_STATUS    rc;
PLX_INTERRUPT PlxIntr;


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

// Set interrupts to enable
PlxIntr.LocalToPci_1 = 1;   // Generic Local-to-PCI int (LINT#)
PlxIntr.DmaChannel_0 = 1;   // PCI DMA Channel 0

rc =
    PlxPci_InterruptEnable(
        pDevice,
        &PlxIntr
        );

if (rc != ApiSuccess)
    // ERROR - Unable to enable interrupts
```

PlxPci_MailboxRead

Syntax:

```
U32
PlxPci_MailboxRead(
    PLX_DEVICE_OBJECT *pDevice,
    U16                mailbox,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, & 8000 NT

Description:

Returns the value of the specified mailbox/scratchpad register.

Parameters:

pDevice
: Pointer to an open device

mailbox
: The specified mailbox to read

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The function is not supported by the driver/device
ApiInvalidIndex	The specified mailbox is invalid for the selected device

Usage:

```
U32 MB_Value;


// Read MB
MB_Value =
    PlxPci_MailboxRead(
        pDevice,
        4,          // Mailbox 4
        NULL
        );
```

PlxPci_MailboxWrite

Syntax:

```
PLX_STATUS
PlxPci_MailboxWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U16               mailbox,
    U32               value
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, & 8000 NT

Description:

Writes a value to the specified mailbox/scratchpad register.

Parameters:

pDevice
: Pointer to an open device

mailbox
: The specified mailbox to write

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The function is not supported by the driver/device
ApiInvalidIndex	The specified mailbox is invalid for the selected device

Usage:

```
#define MSG_READY     0x1234ABCD

    // Post ready to other side
    PlxPci_MailboxWrite(
        pDevice,
        4,          // Mailbox 4
        MSG_READY
        );
```

PlxPci_MH_GetProperties

Syntax:

```
PLX_STATUS
PlxPci_MH_GetProperties(
    PLX_DEVICE_OBJECT   *pDevice,
    PLX_MULTI_HOST_PROP *pMHProp
    );
```

PLX Chip Support:

PLX 8000 virtual switches that support multi-host feature

Description:

Returns the current properties of a PLX switch capable of supporting multi-host.

Parameters:

pDevice
    Pointer to an open device

pMHProp
    A pointer to a PLX_MULTI_HOST_PROP structure that will contain the device's properties.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	Selected device does not support multi-host capabilities or device is not the management port in Virtual Switch mode

Usage:

```
PLX_STATUS          rc;
PLX_MULTI_HOST_PROP MHProp;


// Query MH switch properties
rc =
    PlxPci_MH_GetProperties(
        pDevice,
        &MHProp
        );

if (rc != ApiSuccess)
    // Error – Unable to obtain MH switch properties
else
{
    // Display properties
    if (MHProp.SwitchMode == PLX_SWITCH_MODE_STANDARD)
        Cons_printf("Switch is in standard single-host mode\n");

    if (MHProp.SwitchMode == PLX_SWITCH_MODE_MULTI_HOST)
    {
        if (MHProp.bIsMgmtPort == FALSE)
        {
            // Device properties only available through mgmt port
            Cons_printf(
                "Switch mode is multi-host but port not management\n"
                );
        }
        else
        {
            Cons_printf(
                "Properties:\n"
                "  Mode               : Multi-host\n"
                "  Curr Mgmt Port     : %d (%s)\n"
                "  Backup Mgmt Port   : %d (%s)\n"
                "  Active VS port mask: %08X\n",
                MHProp.MgmtPortNumActive,
                (MHProp.bMgmtPortActiveEn) ? "Enabled" : "Disabled",
                MHProp.MgmtPortNumRedundant,
                (MHProp.bMgmtPortRedundantEn) ? "Enabled" : "Disabled",
                MHProp.VS_EnabledMask
                );
        }
    }
}
```

PlxPci_MH_MigratePorts

Syntax:

```
PLX_STATUS
PlxPci_MH_MigratePorts(
    PLX_DEVICE_OBJECT *pDevice,
    U16               VS_Source,
    U16               VS_Dest,
    U32               DsPortMask,
    BOOLEAN           bResetSrc
    );
```

PLX Chip Support:

PLX 8000 virtual switches that support multi-host feature

Description:

Migrates one or more downstream ports from one virtual switch host to another.

Parameters:

pDevice
: Pointer to an open device

VS_Source
: The virtual host to remove downstream port(s) from.

VS_Dest
: The virtual host that will be assigned the downstream port(s).

DsPortMask
: A mask of the downstream port(s) to move. Each bit position corresponds to a port number. One or more ports may be specified but must be downstream type.

bResetSrc
: Flag to specify whether to reset the source virtual switch.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	Selected device does not support multi-host capabilities or device is not the management port in Virtual Switch mode

Usage:

```
// Move ports 2 & 5 from VS1 to VS4
status =
    PlxPci_MH_MigratePorts(
        pDevice,
        1,                      // Source port
        4,                      // Destination port
        (1 << 5) | (1 << 2),    // DS ports 2 & 5
        FALSE                   // Do not reset source port
        );

if (status == ApiSuccess)
    // Moved ports
else
    // Error – Unable to move port
```

PlxPci_NotificationCancel

Syntax:

```
PLX_STATUS
PlxPci_NotificationCancel(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_NOTIFY_OBJECT *pEvent
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Cancels a notification object previously registered with PlxPci_NotificationRegisterFor.

Parameters:

pDevice
Pointer to an open device

pEvent
A pointer to a PLX notification object previously registered with PlxPci_NotificationRegisterFor.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidHandle	The PLX driver was unable to reference the event handle
ApiInsufficientResources	Insufficient resources to create the notification object
ApiFailed	The notification object is not valid or not registered

Usage:

```
PLX_INTERRUPT     IntSources;
PLX_STATUS        rc;
PLX_NOTIFY_OBJECT Event;


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

```
// Register for interrupt notification
IntSources.Doorbell      = (1 << 16) | 0xF;  // Doorbells 16, & 0-3
IntSources.Message_0     = 1;
IntSources.ResetDeassert = 1;
IntSources.PmeDeassert   = 1;
IntSources.GPIO_4_5      = 1;
IntSources.GPIO_14_15    = 1;

rc =
    PlxPci_NotificationRegisterFor(
        pDevice,
        &IntSources,
        &Event
        );

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

// Wait for the interrupt
rc =
    PlxPci_NotificationWait(
        pDevice,
        &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 =
    PlxPci_NotificationCancel(
        pDevice,
        &Event
        );

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

PlxPci_NotificationRegisterFor

Syntax:

```
PLX_STATUS
PlxPci_NotificationRegisterFor(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_INTERRUPT     *pPlxIntr,
    PLX_NOTIFY_OBJECT *pEvent
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Registers a notification object with the PLX driver for the specified interrupt(s). It is used in conjunction with PlxPci_NotificationWait.

Parameters:

pDevice
: Pointer to an open 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 PlxPci_NotificationWait.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiInsufficientResources	Not enough memory to allocate a new event handle

Notes:

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 PlxPci_InterruptEnable.

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.

***WARNING**: For users porting applications written with PCI SDK 4.2 or older, note that you only need to call this function one time for each interrupt registration. In SDK 4.2 and older, the PlxIntrAttach API call required constant re-registration. This limitation no longer applies starting with SDK 4.3. If you continuously call PlxPci_NotificationRegisterFor, the registrations will remain persistent in an internal PLX driver list and consume system resources, possibly resulting in an unstable system.*

Usage:

```
PLX_STATUS        rc;
PLX_INTERRUPT     IntSources;
PLX_NOTIFY_OBJECT Event;


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

// Register for doorbell interrupts 1, 3, & 24
IntSources.Doorbell = (1 << 24) | (1 << 3) | (1 << 1);

// Also register for DMA channel 1
IntSources.DmaChannel_1;

rc =
    PlxPci_NotificationRegisterFor(
        pDevice,
        &IntSources,
        &Event
        );

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

// Wait for interrupt
rc =
    PlxPci_NotificationWait(
        pDevice,
        &Event,
        PLX_TIMEOUT_INFINITE       // Wait forever
        );

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

    case ApiWaitTimeout:
        // ERROR - Timeout waiting for interrupts
        break;

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

PlxPci_NotificationStatus

Syntax:

```
PLX_STATUS
PlxPci_NotificationStatus(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_NOTIFY_OBJECT *pEvent,
    PLX_INTERRUPT     *pPlxIntr
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Returns which interrupt(s) caused the provided notification event to trigger.

Parameters:

pDevice
: Pointer to an open device

pEvent
: A pointer to a PLX notification object previously registered with PlxPci_NotificationRegisterFor.

pPlxIntr
: A pointer to a PLX_INTERRUPT structure that will contain all triggered interrupts that caused the notification event to become signaled.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiInsufficientResources	Not enough memory to allocate a new event handle

Notes:

This function will set the flag for all interrupts that have caused a notification event since the last query. In other words, if two different interrupts occurred, the status will indicate two different interrupts. There is no way to determine if the same interrupt triggered multiple times since the last query.

Usage:

```
PLX_INTERRUPT     IntSources;
PLX_STATUS        rc;
PLX_NOTIFY_OBJECT Event;


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

// Wait for interrupt on previously registered event
rc =
    PlxPci_NotificationWait(
        pDevice,
        &Event,
        10 * 1000           // 10 second timeout
        );

if (rc != ApiSuccess)
{
    // ERROR – Interrupt wait failed
}

// Determine which interrupt occurred
rc =
    PlxPci_NotificationStatus(
        pDevice,
        &NotifyObject,
        &IntSources
        );

if (rc == ApiSuccess)
{
    Cons_printf("Triggered interrupt(s):");

    if (IntSources.Doorbell)
        Cons_Printf(" <Doorbell>");

    if (IntSources.DmaChannel_0)
        Cons_Printf(" <DMA 0>");

    if (IntSources.GPIO_14_15)
        Cons_Printf(" <GPIO_14_15>");

    if (IntSources.LocalToPci_1)
        Cons_Printf(" <L-to-P 1>");

    Cons_Printf("\n");
}
```

PlxPci_NotificationWait

Syntax:

```
PLX_STATUS
PlxPci_NotificationWait(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_NOTIFY_OBJECT *pEvent,
    U64                Timeout_ms
    );
```

PLX Chip Support:

All PLX 9000 devices, 8311, 8000 DMA, 6000 NT, & 8000 NT

Description:

Wait for a specific interrupt(s) associated with a PLX notification object to occur or until the timeout is reached.

Parameters:

pDevice
: Pointer to an open device

pEvent
: A pointer to a PLX notification object previously registered with PlxPci_NotificationRegisterFor.

Timeout_ms
: The desired time to wait, in milliseconds, for the event to occur. To wait forever, use the pre-defined value ***PLX_TIMEOUT_INFINITE***.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiFailed	The notification object is not valid or not registered
ApiWaitTimeout	Reached timeout waiting for event
ApiWaitCanceled	Wait event was cancelled

Usage:

```
PLX_STATUS        rc;
PLX_INTERRUPT     IntSources;
PLX_NOTIFY_OBJECT Event;


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

// Register for interrupt notification
IntSources.DmaChannel_0 = 1;

rc =
    PlxPci_NotificationRegisterFor(
        pDevice,
        &IntSources,
        &Event
        );

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

// Wait for the interrupt
rc =
    PlxPci_NotificationWait(
        pDevice,
        &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;
}
```

PlxPci_Nt_LutAdd

Syntax:

```
PLX_STATUS
PlxPci_Nt_LutAdd(
    PLX_DEVICE_OBJECT *pDevice,
    U16               *pLutIndex,
    U16                ReqId,
    U32                flags
    );
```

PLX Chip Support:

PLX 8000 NT

Description:

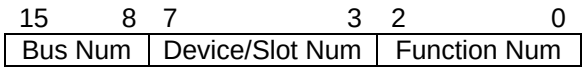
Adds a PCIe Requester ID entry to the PLX NT port Look-Up Table (LUT)

Parameters:

pDevice
: Pointer to an open device

pLutIndex
: *(May be NULL)* A pointer to a variable containing the desired LUT index. If set to -1 (FFFF), the index will be auto-determined by the driver.
On output and if not NULL, will contain the LUT index used.

ReqId
: The Requester ID to add. The format of the ID is standard PCIe format found in TLPs:



flags
: One or more flags to set in the entry. Refer to PLX_NT_LUT_FLAG.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The selected device is not being accessed through the PLX NT driver
ApiInvalidIndex	The specified LUT index was outside the range of possible values
ApiInsufficientResources	No available LUT entry was available to use

Usage:

```
// Probe for write ReqID
if (PlxPci_Nt_ReqIdProbe(
        &Device,
        FALSE,          // Probe for writes
        &ReqId_Write
        ) == FALSE)
{
    Cons_printf("ERROR: Unable to probe ReqID\n");
}
else
{
    Cons_printf(
        "Write ReqID=%04X [b:%02X s:%02X f:%01X])\n",
        ReqId_Write,
        (ReqId_Write >> 8) & 0xFF,
        (ReqId_Write >> 3) & 0x1F,
        (ReqId_Write >> 0) & 0x03
        );

    // Default to auto-selected LUT index
    LutIndex = (U16)-1;

    // Add write Req ID to LUT
    if (PlxPci_Nt_LutAdd(
            &Device,
            &LutIndex,
            ReqId_Write,
            FALSE       // Snoop must be disabled
            ) != ApiSuccess)
    {
        Cons_printf("ERROR: Unable to add LUT entry\n");
    }
}

// Probe for read ReqID
if (PlxPci_Nt_ReqIdProbe(
        &Device,
        TRUE,           // Probe for reads
        &ReqId_Read
        ) == FALSE)
    Cons_printf("ERROR: Unable to probe ReqID\n");
else
{
    Cons_printf(
        "Read ReqID=%04X [b:%02X s:%02X f:%01X])\n",
        ReqId_Read,
        (ReqId_Read >> 8) & 0xFF,
        (ReqId_Read >> 3) & 0x1F,
        (ReqId_Read >> 0) & 0x03
        );
```

```
    if (ReqId_Read == ReqId_Write)
    {
        Cons_printf("-- Read Req ID matches write, skip LUT add --\n");
    }
    else
    {
        // Default to auto-selected LUT index
        LutIndex = (U16)-1;

        // Add read Req ID to LUT
        if (PlxPci_Nt_LutAdd(
                &Device,
                &LutIndex,
                ReqId_Read,
                FALSE       // Snoop must be disabled
                ) != ApiSuccess)
        {
            Cons_printf("ERROR: Unable to add LUT entry\n");
        }
        else
        {
            Cons_printf("Ok (LUT_Index=%d No_Snoop=OFF)\n", LutIndex);
        }
    }
}
```

PlxPci_Nt_LutDisable

Syntax:

```
PLX_STATUS
PlxPci_Nt_LutDisable(
    PLX_DEVICE_OBJECT *pDevice,
    U16                LutIndex
    );
```

***** Note: Not yet implemented in the PLX SDK and will currently return ApiUnsupportedFunction *****

PLX Chip Support:

PLX 8000 NT

Description:

Disables the specified NT LUT index.

Parameters:

pDevice
    Pointer to an open device

LutIndex
    The NT LUT index to disable

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The selected device is not being accessed through the PLX NT driver
ApiInvalidIndex	The specified LUT index was outside the range of possible values

Notes:

The NT LUT is shared by all processes. On successful return, the LUT entry may still actually be enabled in the PLX chip if other active processes also added the same ReqID and the entry was re-used.

Usage:

PlxPci_Nt_LutProperties

Syntax:

```
PLX_STATUS
PlxPci_Nt_LutProperties(
    PLX_DEVICE_OBJECT *pDevice,
    U16                LutIndex,
    U16               *pReqId,
    U32               *pFlags,
    BOOLEAN           *pbEnabled
    );
```

***** Note: Not yet implemented in the PLX SDK and will currently return ApiUnsupportedFunction *****

PLX Chip Support:

PLX 8000 NT

Description:

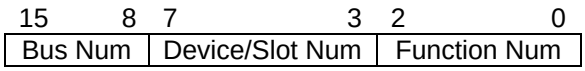
Returns the requested properties of the specified PLX NT LUT entry

Parameters:

pDevice
: Pointer to an open device

LutIndex
: The NT LUT index to retrieve properties for

pReqId
: *(May be NULL)* A pointer to contain the ReqID in the entry The format of the ID is standard PCIe format found in TLPs:



pFlags
: *(May be NULL)* A pointer to contain any additional entry properties. Refer to PLX_NT_LUT_FLAG.

pbEnabled
: *(May be NULL)* A pointer to contain a BOOLEAN specifying whether the entry is enabled or not

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The selected device is not being accessed through the PLX NT driver
ApiInvalidIndex	The specified LUT index was outside the range of possible values

Usage:

PlxPci_Nt_ReqIdProbe

Syntax:

```
PLX_STATUS
PlxPci_Nt_ReqIdProbe(
    PLX_DEVICE_OBJECT *pDevice
    BOOLEAN            bRead,
    U16               *pReqId
    );
```

PLX Chip Support:

PLX 8000 NT

Description:

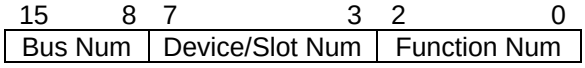
Attempts to determine the Host PCIe Requester ID when it accesses one of the PLX NT BAR spaces. The ReqID must then be added to the PLX NT LUT in order for the NT port to accept memory transactions from the Host. Refer to the *PlxPci_Nt_LutAdd* function.

Parameters:

pDevice
: Pointer to an open device

bRead
: Determines whether the algorithm probes using memory read or write access

pReqId
: A pointer to contain the detected Requester ID. The format of the ID is standard PCIe format found in TLPs:



Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The selected device is not being accessed through the PLX NT driver
ApiFailed	The determination algorithm failed to properly detect the ReqID

Notes:

The determination of the Host ReqID involves a special algorithm. This feature may not always be successful in determining the ReqID, in which case, other techniques must be used. For algorithm details, please refer to PLX driver source code.

On most systems, the PCIe ReqID used for memory reads and writes will be the same. PLX has noticed, however, that many newer chipsets will use 2 different ReqIDs. In general, the ReqID for write TLPs will be the Root Complex (bus:0 slot:0: fn:0) & the ReqID for read TLPs will be the parent PCIe Root Complex Root Port of the NT port.

Usage:

```
// Probe for write ReqID
if (PlxPci_Nt_ReqIdProbe(
        &Device,
        FALSE,          // Probe for writes
        &ReqId_Write
        ) == FALSE)
{
    Cons_printf("ERROR: Unable to probe ReqID\n");
}
else
{
    Cons_printf(
        "Write ReqID=%04X [b:%02X s:%02X f:%01X])\n",
        ReqId_Write,
        (ReqId_Write >> 8) & 0xFF,
        (ReqId_Write >> 3) & 0x1F,
        (ReqId_Write >> 0) & 0x03
        );

    // Default to auto-selected LUT index
    LutIndex = (U16)-1;

    // Add write Req ID to LUT
    if (PlxPci_Nt_LutAdd(
            &Device,
            &LutIndex,
            ReqId_Write,
            FALSE       // Snoop must be disabled
            ) != ApiSuccess)
    {
        Cons_printf("ERROR: Unable to add LUT entry\n");
    }
}

// Probe for read ReqID
if (PlxPci_Nt_ReqIdProbe(
        &Device,
        TRUE,           // Probe for reads
        &ReqId_Read
        ) == FALSE)
    Cons_printf("ERROR: Unable to probe ReqID\n");
else
{
    Cons_printf(
        "Read ReqID=%04X [b:%02X s:%02X f:%01X])\n",
        ReqId_Read,
        (ReqId_Read >> 8) & 0xFF,
        (ReqId_Read >> 3) & 0x1F,
        (ReqId_Read >> 0) & 0x03
        );

    if (ReqId_Read == ReqId_Write)
    {
        Cons_printf("-- Read Req ID matches write, skip LUT add --\n");
    }
```

```
    else
    {
        // Default to auto-selected LUT index
        LutIndex = (U16)-1;

        // Add read Req ID to LUT
        if (PlxPci_Nt_LutAdd(
                &Device,
                &LutIndex,
                ReqId_Read,
                FALSE       // Snoop must be disabled
                ) != ApiSuccess)
        {
            Cons_printf("ERROR: Unable to add LUT entry\n");
        }
        else
        {
            Cons_printf("Ok (LUT_Index=%d No_Snoop=OFF)\n", LutIndex);
        }
    }
}
```

PlxPci_PciBarSpaceRead

Syntax:

```
PLX_STATUS
PlxPci_PciBarSpaceRead(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 BarIndex,
    U32                offset,
    VOID              *pBuffer,
    U32                ByteCount,
    PLX_ACCESS_TYPE    AccessType,
    BOOLEAN            bOffsetAsLocalAddr
    );
```

PLX Chip Support:

All 9000 series & 8311

Description:

Reads from the specified PCI BAR space of a PLX chip (sometimes referred to as Direct Slave Read).

Parameters:

pDevice
: Pointer to an open device

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

offset
: If *bOffsetAsLocalAddr* is FALSE, offset is an offset from the PCI BAR 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 *bOffsetAsLocalAddr* is TRUE, offset is treated as the actual local bus base address to start reading from. For 32-bit devices, this allows access to any location on the 4GB local bus space.

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.

bOffsetAsLocalAddr (9000 & 8311 devices only)
: Determines how the API treats the *offset* parameter.

  If FALSE, *offset* is treated as an offset from the PCI BAR space.
  If TRUE, *offset* is treated as the actual local bus address. The driver will adjust the space remap register to access the 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
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 PLX_ACCESS_TYPE parameter
ApiInvalidAddress	The *offset* parameter is not aligned based on the *AccessType*
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the *AccessType*

Notes:

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

For better performance, use the PlxPci_PciBarMap 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 greater than using the API function.

The end result of this function is a read from the device's 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];


// Read from an absolute local bus address
PlxPci_PciBarSpaceRead(
    pDevice,
    2,                  // Use BAR 2
    0x00100000,         // Absolute local address of 1MB
    buffer,             // Destination buffer
    sizeof(buffer),     // Buffer size in bytes
    BitSize32,          // 32-bit accesses
    TRUE                // Treat offset as a local bus address
    );

// Read from an offset into the PCI BAR
PlxPci_PciBarSpaceRead(
    pDevice,
    3,                  // Use BAR 3
    0x00000100,         // Offset from BAR to start reading from
    buffer,             // Destination buffer
    sizeof(buffer),     // Buffer size in bytes
    BitSize16,          // 16-bit accesses
    FALSE               // Treat Offset as an offset from BAR
    );
```

PlxPci_PciBarSpaceWrite

Syntax:

```
PLX_STATUS
PlxPci_PciBarSpaceWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 BarIndex,
    U32                offset,
    VOID              *pBuffer,
    U32                ByteCount,
    PLX_ACCESS_TYPE    AccessType,
    BOOLEAN            bOffsetAsLocalAddr
    );
```

PLX Chip Support:

All 9000 series & 8311

Description:

Writes to the specified PCI BAR space of PLX chip (sometimes referred to as Direct Slave Write).

Parameters:

pDevice
: Pointer to an open device

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

offset
: If *bOffsetAsLocalAddr* is FALSE, offset is an offset from the PCI BAR 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 *bOffsetAsLocalAddr* is TRUE, offset is treated as the actual local bus base address to start reading from. For 32-bit devices, this allows access to any location on the 4GB local bus space.

pBuffer
: A pointer to a user supplied buffer that contains 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.

bOffsetAsLocalAddr (9000 & 8311 devices only)
: Determines how the API treats the *offset* parameter.

  If FALSE, *offset* is treated as an offset from the PCI BAR space.
  If TRUE, *offset* is treated as the actual local bus address. The driver will adjust the space remap register to access the 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
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 PLX_ACCESS_TYPE parameter
ApiInvalidAddress	The *address* parameter is not aligned based on the *accessType*
ApiInvalidSize	The transfer size parameter is 0 or is not aligned based on the *accessType*

Notes:

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

For better performance, use the PlxPci_PciBarMap 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 greater than using the API function.

The end result of this function is a write to the device's 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];


// Write to an absolute local bus address
PlxPci_PciBarSpaceWrite(
    pDevice,
    2,                  // Use BAR 2
    0x00100000,         // Absolute local address of 1MB
    buffer,             // Destination buffer
    sizeof(buffer),     // Buffer size in bytes
    BitSize32,          // 32-bit accesses
    TRUE                // Treat offset as a local bus address
    );

// Write to an offset from the PCI BAR window
PlxPci_PciBarSpaceWrite(
    pDevice,
    3,                  // Use BAR 3
    0x00000100,         // Offset from BAR to start reading from
    buffer,             // Source buffer
    sizeof(buffer),     // Buffer size in bytes
    BitSize16,          // 16-bit accesses
    FALSE               // Treat Offset as an offset from BAR
    );
```

PlxPci_PciBarMap

Syntax:

```
PLX_STATUS
PlxPci_PciBarMap(
    PLX_DEVICE_OBJECT  *pDevice,
    U8                  BarIndex,
    VOID              **pVa
    );
```

PLX Chip Support:

All devices

Description:

Maps a PCI BAR into user virtual space and returns the virtual address. User applications may then bypass the driver and directly access a PCI space for optimal performance.

Parameters:

pDevice
: Pointer to an open 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiUnsupportedFunction	Mapping of a PCI BAR space is not supported by the installed PLX driver
ApiInvalidIndex	PCI BAR index is not in the range of valid values
ApiFailed	Virtual address mapping failed
ApiInvalidPciSpace	The PCI space is of type I/O, not memory
ApiInvalidSize	The PCI space is of size 0 (disabled)
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:

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 PlxPci_DeviceClose. Un-mapping a space will release the PTE resources used back to the OS. Refer to PlxPci_PciBarUnmap.

The PCI space that will be mapped into user virtual space must be a PCI memory type. Mapping of I/O type spaces is not allowed. I/O type spaces should be accessed with PlxPci_IoPortRead and PlxPci_IoPortWrite.

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.

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 (e.g. 16MB or more), the number of PTEs required increases, resulting in a greater risk of a failed mapping attempt.

Usage:

```
    U8          i;
    U32         DataValue;
    VOID       *Va[6];
    PLX_STATUS  rc;


    for (i = 0; i <= 5; i++)
    {
        rc =
            PlxPci_PciBarMap(
                pDevice,
                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",
        (PLX_UINT_PTR)Va[0], (PLX_UINT_PTR)Va[1], (PLX_UINT_PTR)Va[2],
        (PLX_UINT_PTR)Va[3], (PLX_UINT_PTR)Va[4], (PLX_UINT_PTR)Va[5]
        );

/*****************************************************************
 * NOTE:  The configuration of a PCI space is left to the application
 *        The translation registers should be configured correctly
 *        before accessing the PCI space.
 *****************************************************************/

    // Read a 32-bit value from PCI BAR 0
    value = *(U32*)Va[0];

    // Write an 8-bit value to PCI BAR 1, offset 3Ch
    *((U8*)Va[1] + 0x3C) = 0x1A;
```

PlxPci_PciBarProperties

Syntax:

```
PLX_STATUS
PlxPci_PciBarProperties(
    PLX_DEVICE_OBJECT *pDevice,
    U8                 BarIndex,
    PLX_PCI_BAR_PROP  *pBarProperties
    );
```

PLX Chip Support:

All devices

Description:

Returns the properties of the specified PCI BAR space.

Parameters:

pDevice
: Pointer to an open device

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

pBarProperties
: A pointer to a PLX_PCI_BAR_PROP structure that will hold the BAR properties

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiNullParam	One or more parameters is NULL
ApiInvalidIndex	PCI BAR index is not in the range of valid values

Usage:

```
PLX_PCI_BAR_PROP BarProp

// Get BAR 2 size
PlxPci_PciBarProperties(
    pDevice,
    2,
    &BarProp
    );

Cons_Printf(
    "BAR 2:  %d bytes",
    BarProp.Size
    );
```

PlxPci_PciBarUnmap

Syntax:

```
PLX_STATUS
PlxPci_PciBarUnmap(
    PLX_DEVICE_OBJECT  *pDevice,
    VOID              **pVa
    );
```

PLX Chip Support:

All devices

Description:

Unmaps a PCI BAR space from user virtual space, previously mapped with PlxPci_PciBarMap.

Parameters:

pDevice
: Pointer to an open device

pVa
: Pointer to the virtual address of the PCI BAR to unmap, previously obtained from PlxPci_PciBarMap.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully and at least one event ocurred
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	Unmapping of a PCI BAR space is not supported by the installed PLX driver
ApiNullParam	One or more parameters is NULL
ApiInvalidAddress	The virtual address is invalid or not a previously mapped address

Notes:

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

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

Usage:

```
U32        *Va;
PLX_STATUS  rc;


// Map PCI BAR 0
rc =
    PlxPci_PciBarMap(
        pDevice,
        0,
        (VOID**)&Va
        );

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


//
// Access PCI space as needed ...
//


// Unmap the space
rc =
    PlxPci_PciBarUnmap(
        pDevice,
        (VOID**)&Va
        );

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

PlxPci_PciRegisterRead

Syntax:

```
U32
PlxPci_PciRegisterRead(
    U8          bus,
    U8          slot,
    U8          function,
    U16         offset,
    PLX_STATUS *pStatus
    );
```

PLX Chip Support:

All devices

Description:

Returns the value of a PCI configuration register at a specified offset

Parameters:

bus
: Device bus number

slot
: Device slot number

function
: Device function number

offset
: PCI register 32-bit aligned offset

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiConfigAccessFailed	The PCI configuration access failed or device does not exist

Notes:

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

Usage:

```
U8         bus;
U8         slot;
U32        RegValue;
PLX_STATUS rc;


// Scan for all PCI devices
for (bus = 0; bus < 32; bus++)
{
    for (slot = 0; slot < 32; slot+)
    {
        // Read the Device/Vendor ID
        RegValue =
            PlxPci_PciRegisterRead(
                bus,
                slot,
                0,      // Just function 0 devices
                0x0,    // Device/Vendor ID register
                &rc
                );

        if ((rc == ApiSuccess) && (RegValue != (U32)-1))
        {
            // Found a valid PCI device
            Cons_Printf(
                "Device ID: %08x  [bus %02x  slot %02x]\n",
                RegValue, bus, slot
                );
        }
    }
}
```

PlxPci_PciRegisterWrite

Syntax:

```
PLX_STATUS
PlxPci_PciRegisterWrite(
    U8  bus,
    U8  slot,
    U8  function,
    U16 offset,
    U32 value
    );
```

PLX Chip Support:

All devices

Description:

Writes a 32-bit value to a PCI configuration register at a specified offset

Parameters:

bus
: Device bus number

slot
: Device slot number

function
: Device function number

offset
: PCI register 32-bit aligned offset

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiConfigAccessFailed	The PCI configuration access failed or device does not exist

Notes:

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

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Read the PCI Command/Status register
RegValue =
    PlxPci_PciRegisterRead(
        1,
        0xe,
        0,
        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 =
        PlxPci_PciRegisterWrite(
            1,
            0xe,
            0,
            CFG_COMMAND,
            RegValue
            );

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

PlxPci_PciRegisterReadFast

Syntax:

```
U32
PlxPci_PciRegisterReadFast(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All devices

Description:

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

Parameters:

pDevice
: Pointer to an open device

offset
: PCI register 32-bit aligned offset

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiConfigAccessFailed	The PCI configuration access failed or device does not exist

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Read Device/Vendor ID
RegValue =
    PlxPci_PciRegisterReadFast(
        pDevice,
        CFG_VENDOR_ID,
        &rc
        );

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

PlxPci_PciRegisterWriteFast

Syntax:

```
PLX_STATUS
PlxPci_PciRegisterWriteFast(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U32                value
    );
```

PLX Chip Support:

All devices

Description:

Writes to a PCI configuration register on the selected device.

Parameters:

pDevice
: Pointer to an open device

offset
: PCI register 32-bit aligned offset

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiConfigAccessFailed	The PCI configuration access failed or device does not exist

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Read the PCI Command/Status register
RegValue =
    PlxPci_PciRegisterReadFast(
        pDevice,
        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 =
        PlxPci_PciRegisterWriteFast(
            pDevice,
            CFG_COMMAND,
            RegValue
            );

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

PlxPci_PciRegisterRead_BypassOS

Syntax:

```
U32
PlxPci_PciRegisterRead_BypassOS(
    U8          bus,
    U8          slot,
    U8          function,
    U16         offset,
    PLX_STATUS *pStatus
    );
```

PLX Chip Support:

All devices

Description:

Bypasses the OS services to read a specific PCI configuration register

Parameters:

bus
: Device bus number

slot
: Device slot number

function
: Device function number

offset
: PCI register 32-bit aligned offset

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiUnsupportedFunction	The function is not supported by the installed PLX driver

Notes:

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        RegValue;
PLX_STATUS rc;


// Scan for all PCI devices
for (bus = 0; bus < 32; bus++)
{
    for (slot = 0; slot < 32; slot+)
    {
        // Read the Device/Vendor ID
        RegValue =
            PlxPci_PciRegisterRead_BypassOS(
                bus,
                slot,
                0,        // Just function 0 devices
                0x0,      // Device/Vendor ID
                &rc
                );

        if ((rc == ApiSuccess) && (RegValue != 0xFFFFFFFF))
        {
            // Found a valid PCI device
            Cons_Printf(
                "Device ID: %08x  [bus %02x  slot %02x]\n",
                RegValue, bus, slot
                );
        }
    }
}
```

PlxPci_PciRegisterWrite_BypassOS

Syntax:

```
PLX_STATUS
PlxPci_PciRegisterWrite_BypassOS(
    U8  bus,
    U8  slot,
    U8  function,
    U16 offset,
    U32 value
    );
```

PLX Chip Support:

All devices

Description:

Bypasses the OS services to write to a specific PCI configuration register

Parameters:

bus
: Device bus number

slot
: Device slot number

function
: Device function number

offset
: PCI register 32-bit aligned offset

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNoActiveDriver	A valid PLX driver is not loaded in the system
ApiUnsupportedFunction	The function is not supported by the installed PLX driver

Notes:

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.

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:

```
U32        RegValue;
PLX_STATUS rc;


// Read the PCI Command/Status register
RegValue =
    PlxPci_PciRegisterRead(
        1,
        0xe,
        0,
        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 =
        PlxPci_PciRegisterWrite_BypassOS(
            1,
            0xe,
            0,
            CFG_COMMAND,
            RegValue
            );

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

PlxPci_PerformanceCalcStatistics

Syntax:

```
PLX_STATUS
PlxPci_PerformanceCalcStatistics(
    PLX_PERF_PROP  *pPerfProp,
    PLX_PERF_STATS *pPerfStats,
    U32             ElapsedTime_ms
    );
```

PLX Chip Support:

PLX PCI Express 8000 switches that support internal Performance Counters.

Description:

Uses the performance properties to calculate the resulting performance statistics for a specific port

Parameters:

pPerfProp
: Pointer to a PLX_PERF_PROP structure that contains the performance counters and properties filled in from a call to *PlxPci_PerformanceGetCounters().*

pPerfStats
: Pointer to a PLX_PERF_STATS structure that will contain the calculated performance statistics based upon the counters and elapsed time.

ElapsedTime_ms
: The elapsed time in milliseconds betweens reads of the Performance Counters (i.e. calls to *PlxPci_PerformanceGetCounters()*).

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidData	Elapsed time is invalid

Notes:

Usage:

```
U32            ElapsedTime_ms;
struct timeb   PrevTime, EndTime;
PLX_PERF_PROP  PerfProp;
PLX_PERF_STATS PerfStats;


// Initialize performance objects
PlxPci_PerformanceInitializeProperties(
    pDevice,
    &PerfProp
    );

// Start performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_START
    );

// Reset counters
PlxPci_PerformanceResetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get starting time
ftime( &PrevTime );

// Insert small delay
Plx_sleep( 1000 );

// Get statistics
PlxPci_PerformanceGetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get end time
ftime( &EndTime );

// Calculate elapsed time in milliseconds
ElapsedTime_ms = (((U32)EndTime.time * 1000) + EndTime.millitm) -
                 (((U32)PrevTime.time * 1000) + PrevTime.millitm);

// Calculate performance statistics
PlxPci_PerformanceCalcStatistics(
    &PerfProp,
    &PerfStats,
    ElapsedTime_ms
    );
```

PlxPci_PerformanceGetCounters

Syntax:

```
PLX_STATUS
PlxPci_PerformanceGetCounters(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PERF_PROP     *pPerfProp,
    U8                 NumOfObjects
    );
```

PLX Chip Support:

PLX PCI Express 8000 switches that support internal Performance Counters.

Description:

Fills in all the performance counters in the provided performance property objects

Parameters:

pDevice
    Pointer to an open device

pPerfProp
    A pointer to one or more PLX_PERF_PROP structures.

NumOfObjects
    Specifies the number of PLX_PERF_PROP objects pointed to by pPerfProp.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened or one or more PLX_PERF_PROP objects is invalid or has not been initialized.
ApiUnsupportedFunction	The PLX chip does not support Performance Counters.

Notes:

Usage:

```
U32            ElapsedTime_ms;
struct timeb   PrevTime, EndTime;
PLX_PERF_PROP  PerfProp;
PLX_PERF_STATS PerfStats;


// Initialize performance objects
PlxPci_PerformanceInitializeProperties(
    pDevice,
    &PerfProp
    );

// Start performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_START
    );

// Reset counters
PlxPci_PerformanceResetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get starting time
ftime( &PrevTime );

// Insert small delay
Plx_sleep( 1000 );

// Get statistics
PlxPci_PerformanceGetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get end time
ftime( &EndTime );

// Calculate elapsed time in milliseconds
ElapsedTime_ms = (((U32)EndTime.time * 1000) + EndTime.millitm) -
                 (((U32)PrevTime.time * 1000) + PrevTime.millitm);

// Calculate performance statistics
PlxPci_PerformanceCalcStatistics(
    &PerfProp,
    &PerfStats,
    ElapsedTime_ms
    );
```

PlxPci_PerformanceInitializeProperties

Syntax:

```
PLX_STATUS
PlxPci_PerformanceInitializeProperties(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PERF_PROP     *pPerfProp
    );
```

PLX Chip Support:

PLX PCI Express 8000 switches that support internal Performance Counters.

Description:

Initializes a performance object for use with the performance counter functions

Parameters:

pDevice
    Pointer to an open device

pPerfProp
    Pointer to a PLX_PERF_PROP object

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened or one or more PLX_PERF_PROP objects is invalid or has not been initialized.
ApiUnsupportedFunction	The PLX chip does not support Performance Counters or the port number is invalid.

Notes:

Usage:

```
U32            ElapsedTime_ms;
struct timeb   PrevTime, EndTime;
PLX_PERF_PROP  PerfProp;
PLX_PERF_STATS PerfStats;


// Initialize performance objects
PlxPci_PerformanceInitializeProperties(
    pDevice,
    &PerfProp
    );

// Start performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_START
    );

// Reset counters
PlxPci_PerformanceResetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get starting time
ftime( &PrevTime );

// Insert small delay
Plx_sleep( 1000 );

// Get statistics
PlxPci_PerformanceGetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get end time
ftime( &EndTime );

// Calculate elapsed time in milliseconds
ElapsedTime_ms = (((U32)EndTime.time * 1000) + EndTime.millitm) -
                 (((U32)PrevTime.time * 1000) + PrevTime.millitm);

// Calculate performance statistics
PlxPci_PerformanceCalcStatistics(
    &PerfProp,
    &PerfStats,
    ElapsedTime_ms
    );
```

PlxPci_PerformanceMonitorControl

Syntax:

```
PLX_STATUS
PlxPci_PerformanceMonitorControl(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PERF_CMD       command
    );
```

PLX Chip Support:

PLX PCI Express 8000 switches that support internal Performance Counters.

Description:

Controls the PLX Performance Counters

Parameters:

pDevice
    Pointer to an open device

command
    A PLX_PERF_CMD that specifies the operation to perform

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiUnsupportedFunction	The PLX chip does not support Performance Counters.
ApiInvalidData	The command parameter is not a valid PLX_PERF_CMD value.

Notes:

Usage:

```
U32            ElapsedTime_ms;
PLX_PERF_PROP  PerfProp;
PLX_PERF_STATS PerfStats;


// Set desired elapsed time
ElapsedTime_ms = 1000;

// Initialize performance objects
PlxPci_PerformanceInitializeProperties(
    pDevice,
    &PerfProp
    );

// Start performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_START
    );

// Reset counters
PlxPci_PerformanceResetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Insert small delay
Plx_sleep( ElapsedTime_ms );

// Get statistics
PlxPci_PerformanceGetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Stop performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_STOP
    );

// Calculate performance statistics
PlxPci_PerformanceCalcStatistics(
    &PerfProp,
    &PerfStats,
    ElapsedTime_ms
    );
```

PlxPci_PerformanceResetCounters

Syntax:

```
PLX_STATUS
PlxPci_PerformanceResetCounters(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PERF_PROP     *pPerfProp,
    U8                 NumOfObjects
    );
```

PLX Chip Support:

PLX PCI Express 8000 switches that support internal Performance Counters.

Description:

Resets all the performance counters in the provided performance property objects

Parameters:

pDevice
: Pointer to an open device

pPerfProp
: A pointer to one or more PLX_PERF_PROP structures.

NumOfObjects
: Specifies the number of PLX_PERF_PROP objects pointed to by pPerfProp.

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened or one or more PLX_PERF_PROP objects is invalid or has not been initialized.
ApiUnsupportedFunction	The PLX chip does not support Performance Counters.

Notes:

Usage:

```
U32            ElapsedTime_ms;
struct timeb   PrevTime, EndTime;
PLX_PERF_PROP  PerfProp;
PLX_PERF_STATS PerfStats;


// Initialize performance objects
PlxPci_PerformanceInitializeProperties(
    pDevice,
    &PerfProp
    );

// Start performance monitor
PlxPci_PerformanceMonitorControl(
    pDevice,
    PLX_PERF_CMD_START
    );

// Reset counters
PlxPci_PerformanceResetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get starting time
ftime( &PrevTime );

// Insert small delay
Plx_sleep( 1000 );

// Get statistics
PlxPci_PerformanceGetCounters(
    pDevice,
    &PerfProp,
    1           // Only one object
    );

// Get end time
ftime( &EndTime );

// Calculate elapsed time in milliseconds
ElapsedTime_ms = (((U32)EndTime.time * 1000) + EndTime.millitm) -
                 (((U32)PrevTime.time * 1000) + PrevTime.millitm);

// Calculate performance statistics
PlxPci_PerformanceCalcStatistics(
    &PerfProp,
    &PerfStats,
    ElapsedTime_ms
    );
```

PlxPci_PhysicalMemoryAllocate

Syntax:

```
PLX_STATUS
PlxPci_PhysicalMemoryAllocate(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PHYSICAL_MEM  *pMemoryInfo,
    BOOLEAN            bSmallerOk
    );
```

PLX Chip Support:

All devices

Description:

Attempts to allocate a physically contiguous, page-locked buffer which is safe for use with DMA operation.

Parameters:

pDevice
: Pointer to an open 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 ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInsufficientResources	Insufficient resource to allocate buffer

Notes:

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 2.4 & 2.6, the maximum is 4MB unless the kernel is modified.

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 for transfers across an NT port. 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 bus 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 PlxPci_PhysicalMemoryMap.

Usage:

```
PLX_STATUS       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 =
    PlxPci_PhysicalMemoryAllocate(
        pDevice,
        &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 =
    PlxPci_PhysicalMemoryAllocate(
        pDevice,
        &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
}
```

PlxPci_PhysicalMemoryFree

Syntax:

```
PLX_STATUS
PlxPci_PhysicalMemoryFree(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PHYSICAL_MEM  *pMemoryInfo
    );
```

PLX Chip Support:

All devices

Description:

Releases a buffer previously allocated with PlxPci_PhysicalMemoryAllocate.

Parameters:

pDevice
    Pointer to an open 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 ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidData	The buffer information is invalid or it was not allocated with *PlxPci_PhysicalMemoryAllocate*

Notes:

If the buffer was previously mapped to user virtual space, with PlxPci_PhysicalMemoryMap, it should be unmapped with PlxPci_PhysicalMemoryUnmap 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 PlxPci_DeviceClose. Buffers will become invalid once a device is released.

Usage:

```
PLX_STATUS       rc;
PLX_PHYSICAL_MEM Buffer;


// Allocate a buffer

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

rc =
    PlxPci_PhysicalMemoryAllocate(
        pDevice,
        &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 =
    PlxPci_PhysicalMemoryFree(
        pDevice,
        &Buffer
        );

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

PlxPci_PhysicalMemoryMap

Syntax:

```
PLX_STATUS
PlxPci_PhysicalMemoryMap(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PHYSICAL_MEM  *pMemoryInfo
    );
```

PLX Chip Support:

All devices

Description:

Maps into user virtual space a buffer previously allocated with PlxPci_PhysicalMemoryAllocate.

Parameters:

pDevice
    Pointer to an open 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 ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
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:

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 PlxPci_DeviceClose to close the device. The virtual address will cease to be valid after closing the device or after unmapping the buffer. Refer to PlxPci_PhysicalMemoryUnmap.

Usage:

```
U8               value;
PLX_STATUS       rc;
PLX_PHYSICAL_MEM Buffer;


// Allocate a buffer

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

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

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

// Map the buffer into user space
rc =
    PlxPci_PhysicalMemoryMap(
        pDevice,
        &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);
```

PlxPci_PhysicalMemoryUnmap

Syntax:

```
PLX_STATUS
PlxPci_PhysicalMemoryUnmap(
    PLX_DEVICE_OBJECT *pDevice,
    PLX_PHYSICAL_MEM  *pMemoryInfo
    );
```

PLX Chip Support:

All devices

Description:

Unmaps a physical buffer previously mapped with PlxPci_PhysicalMemoryMap.

Parameters:

pDevice
    Pointer to an open 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 ocurred
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidAddress	The virtual address is invalid or was not previously mapped with *PlxPci_PhysicalMemoryMap*
ApiInvalidData	The buffer information is invalid or it was not allocated with *PlxPci_PhysicalMemoryAllocate*

Notes:

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 PlxPci_DeviceClose to close the device. The virtual address will cease to be valid after closing the device or after un-mapping the buffer.

Usage:

```
PLX_STATUS       rc;
PLX_PHYSICAL_MEM Buffer;


// Allocate a buffer (not shown)

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

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

//
// Access buffer as needed
//

// Unmap the buffer from virtual space
rc =
    PlxPci_PhysicalMemoryUnmap(
        pDevice,
        &Buffer
        );

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

PlxPci_PlxRegisterRead

Syntax:

```
U32
PlxPci_PlxRegisterRead(
    PLX_DEVICE_OBJECT *pDevice,
    U32                offset,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All PLX devices

Description:

Reads a PLX-specific register from the selected device

Parameters:

pDevice
: Pointer to an open device

offset
: PLX register 32-bit aligned offset

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidOffset	The register offset is not aligned or is not one a PLX-specific register

Notes:

For PLX 8000 series devices, the PLX driver will internally adjust the register offset based on the device port number. For example, if the selected PCI device is Port 8 of the PLX switch, the driver will add (8 * 4k) to the *offset* parameter in order to access the correct register region for that specific port.

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Read the PCI Control register
RegValue =
    PlxPci_PlxRegisterRead(
        pDevice,
        0x100C,         // PCI Control register
        &rc
        );

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

// Determine PCI clock rate
if (RegValue & (1 << 7))
    // PCI clock is running at 66MHz
else
    // PCI clock is running at 33MHz
```

PlxPci_PlxRegisterWrite

Syntax:

```
PLX_STATUS
PlxPci_PlxRegisterWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U32                offset,
    U32                value
    );
```

PLX Chip Support:

All PLX devices

Description:

Writes to a PLX-specific register on the selected device

Parameters:

pDevice
: Pointer to an open device

offset
: PLX register 32-bit aligned offset

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidOffset	The register offset is not aligned or is not one a PLX-specific register

Notes:

For PLX 8000 series devices, the PLX driver will internally adjust the register offset based on the device port number. For example, if the selected PCI device is Port 8 of the PLX switch, the driver will add (8 * 4k) to the *offset* parameter in order to access the correct register region for that specific port.

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Write a value to the Mailbox 1 register
rc =
    PlxPci_PlxRegisterWrite (
        pDevice,
        0x1034,         // Mailbox 1 register
        0xFF001300
        );

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

PlxPci_PlxMappedRegisterRead

Syntax:

```
U32
PlxPci_PlxMappedRegisterRead(
    PLX_DEVICE_OBJECT *pDevice,
    U32                offset,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

All PLX devices

Description:

Reads a PLX-specific register from the selected device without adjusting the offset based on the port.

Parameters:

pDevice
: Pointer to an open device

offset
: PLX register 32-bit aligned offset

pStatus
: Pointer to a PLX_STATUS variable to contain the status. (May be NULL)

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidOffset	The register offset is not aligned or is not one a PLX-specific register

Notes:

This function is identical to PlxPci_PlxRegisterRead except the PLX driver will not make an internal adjustment for the port number. The register accessed is simply BAR 0 of the upstream port plus the *offset* parameter.

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Read register 264h from Port 9
RegValue =
    PlxPci_PlxMappedRegisterRead(
        pDevice,
        0x264 + (9 * (4 * 1024)),
        &rc
        );

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

PlxPci_PlxMappedRegisterWrite

Syntax:

```
PLX_STATUS
PlxPci_PlxMappedRegisterWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U32                offset,
    U32                value
    );
```

PLX Chip Support:

All PLX devices

Description:

Writes to a PLX-specific register on the selected device without adjusting the offset based on the port

Parameters:

pDevice
: Pointer to an open device

offset
: PLX register 32-bit aligned offset

value
: The 32-bit value to write

Return Codes:

Code	Description
ApiSuccess	The function returned successfully
ApiNullParam	One or more parameters is NULL
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened
ApiInvalidOffset	The register offset is not aligned or is not one a PLX-specific register

Notes:

This function is identical to PlxPci_PlxRegisterWrite except the PLX driver will not make an internal adjustment for the port number. The register accessed is simply BAR 0 of the upstream port plus the *offset* parameter.

Usage:

```
U32        RegValue;
PLX_STATUS rc;


// Write a value to register 660h from Port 8
rc =
    PlxPci_PlxMappedRegisterWrite(
        pDevice,
        0x660 + (8 * (4 * 1024)),
        0xFF001300
        );

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

PlxPci_VpdRead

Syntax:

```
U32
PlxPci_VpdRead(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    PLX_STATUS        *pStatus
    );
```

PLX Chip Support:

Any device that supports the PCI VPD capability

Description:

Reads a 32-bit value at a specified offset of the Vital Product Data.

Parameters:

pDevice
: Pointer to an open device

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

pStatus
: 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened

Usage:

```
    U32        VpdData;
    PLX_STATUS rc;


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

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

PlxPci_VpdWrite

Syntax:

```
PLX_STATUS
PlxPci_VpdWrite(
    PLX_DEVICE_OBJECT *pDevice,
    U16                offset,
    U32                value
    );
```

PLX Chip Support:

Any device that supports the PCI VPD capability

Description:

Write a 32-bit value to a specified offset of the Vital Product Data.

Parameters:

pDevice
: Pointer to an open 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
ApiInvalidDeviceInfo	The device object is not a valid PLX device or has not been opened

Usage:

```
// Write the new Device/Vendor ID (assuming 9054 device)
PlxPci_VpdWrite(
    pDevice,
    0x0,
    0x186010b5
    );

// Write custom data to non-PLX used EEPROM space
PlxPci_VpdWrite(
    pDevice,
    0x60,         // 9054 data ends at 0x58
    0x0024beef
    );
```

5.2 PLX API Data Structures and Types

This section documents details of the structures and data types used by the PLX API.

5.2.1 Standard Data Types

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
PLX_INT_PTR PLX_UINT_PTR	Types large enough to contain a pointer on the target platform. Will be 32-bit on 32-bit platforms and 64-bit on 64-bit CPU platforms. Signed (***INT***) and unsigned (***UINT***) types are provided.

5.2.1.1 Code Portability Macros

To support source code portability between platforms, the following macros are provided:

- ***PLX_PTR_TO_INT(*** *pointer* ***)*** - Converts a pointer to an integer
- ***PLX_INT_TO_PTR(*** *integer* ***)*** - Convert an integer to a pointer

5.2.2 Enumerated Types

This section contains the enumerated data types used in the PLX API.

PLX_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 (may not be supported on target platform)

PLX_API_MODE

```
typedef enum _PLX_API_MODE
{
    PLX_API_MODE_PCI,
    PLX_API_MODE_I2C_AARDVARK,
    PLX_API_MODE_TCP
} PLX_API_MODE;
```

Purpose

Enumerated type to specify the method used to access a device.

Members

PLX_API_MODE_PCI
Device is accessed via the PLX driver over PCI/PCI Express bus

PLX_API_MODE_I2C_AARDVARK
Device is accessed over I^2C using the Aadvark USB I^2C /SPI connector

PLX_API_MODE_TCP
Device is accessed over TCP/IP (*not currently supported*)

PLX_CHIP_FAMILY

```
typedef enum _PLX_CHIP_FAMILY
{
    PLX_FAMILY_NONE = 0,
    PLX_FAMILY_UNKNOWN,
    PLX_FAMILY_BRIDGE_P2L,              // 9000 series & 8311
    PLX_FAMILY_BRIDGE_PCI_P2P,          // 6000 series
    PLX_FAMILY_BRIDGE_PCIE_P2P,         // 8111,8112,8114
    PLX_FAMILY_ALTAIR,                  // 8525,8533,8547,8548
    PLX_FAMILY_ALTAIR_XL,               // 8505,8509
    PLX_FAMILY_VEGA,                    // 8516,8524,8532
    PLX_FAMILY_VEGA_LITE,               // 8508,8512,8517,8518
    PLX_FAMILY_DENEB,                   // 8612,8616,8624,8632,8647,8648
    PLX_FAMILY_SIRIUS,                  // 8604,8606,8608,8609,8613,8614,
                                        //   8615,8617,8618,8619
    PLX_FAMILY_CYGNUS,                  // 8625,8636,8649,8664,8680,8696
    PLX_FAMILY_SCOUT,                   // 8700
    PLX_FAMILY_DRACO_1,                 // 8712,8716,8724,8732,8747,8748
    PLX_FAMILY_DRACO_2,                 // 8713,8715,8717,8725,8727,8733,8749
    PLX_FAMILY_MIRA,
    PLX_FAMILY_CAPELLA
} PLX_CHIP_FAMILY;
```

Purpose

Enumerated type to specify the PLX chip family.

Members

PLX_FAMILY_NONE
Device is not a PLX chip

PLX_FAMILY_UNKOWN
The PLX chip family was unable to be determined

PLX_FAMILY_xxxx
The various PLX chip families

PLX_DMA_COMMAND

```
typedef enum _PLX_DMA_COMMAND
{
    DmaPause,
    DmaPauseImmediate,
    DmaResume,
    DmaAbort
} PLX_DMA_COMMAND;
```

Purpose

Enumerated type used to control DMA transfers.

Members

DmaPause
Pause a DMA transfer, gracefully if supported by hardware (i.e. completes pending transfers, etc).

DmaPauseImmediate
Pause a DMA transfer immediately without waiting for pending transfers to complete.

DmaResume
Resume a paused DMA transfer.

DmaAbort
Abort a DMA transfer.

PLX_DMA_DESCR_MODE

```
typedef enum _PLX_DMA_DESCR_MODE
{
    PLX_DMA_MODE_BLOCK         = 0,
    PLX_DMA_MODE_SGL           = 1,
    PLX_DMA_MODE_SGL_INTERNAL  = 2,
} PLX_DMA_DESCR_MODE;
```

Purpose

Enumerated type used to control DMA transfers.

Members

PLX_DMA_MODE_BLOCK
DMA operates in single transfer block mode

PLX_DMA_MODE_SGL
DMA operates in SGL (ring) transfer mode with descriptors held externally (off-chip mode)

PLX_DMA_MODE_SGL_INTERNAL
DMA operates in SGL (ring) transfer mode with descriptors held in internal RAM (on-chip mode)

PLX_DMA_RING_DELAY_TIME

```
typedef enum _PLX_DMA_RING_DELAY_TIME
{
    PLX_DMA_RING_DELAY_0       = 0,
    PLX_DMA_RING_DELAY_1us     = 1,
    PLX_DMA_RING_DELAY_2us     = 2,
    PLX_DMA_RING_DELAY_8us     = 3,
    PLX_DMA_RING_DELAY_32us    = 4,
    PLX_DMA_RING_DELAY_128us   = 5,
    PLX_DMA_RING_DELAY_512us   = 6,
    PLX_DMA_RING_DELAY_1ms     = 7
} PLX_DMA_RING_DELAY_TIME;
```

Purpose

In SGL mode, when DMA reaches the end of the ring and ring wrap mode is enabled, this controls the delay before the DMA wraps back to the beginning of the ring.

Members

DMA ring delay period varies from none or 1µs → 1ms via preset values. Refer to the member name for the delay time.

PLX_DMA_DIR

```
typedef enum _PLX_DMA_DIR
{
    PLX_DMA_PCI_TO_LOC   = 0,                   // PCI --> Local bus   (9000 DMA)
    PLX_DMA_LOC_TO_PCI   = 1,                   // Local bus --> PCI   (9000 DMA)
    PLX_DMA_USER_TO_PCI  = PLX_DMA_PCI_TO_LOC,  // User buffer --> PCI (8000 DMA)
    PLX_DMA_PCI_TO_USER  = PLX_DMA_LOC_TO_PCI   // PCI --> User buffer (8000 DMA)
} PLX_DMA_DIR;
```

Purpose

Enumerated type used to specify the direction of DMA transfers.

Members

PLX_DMA_PCI_TO_LOC (9000 DMA)
Sets the DMA transfer direction from PCI → Local Bus

PLX_DMA_LOC_TO_PCI (9000 DMA)
Sets the DMA transfer direction from Local Bus → PCI

PLX_DMA_USER_TO_PCI (8000 DMA)
Sets the DMA transfer direction from a user mode provided buffer → a destination PCI address

PLX_DMA_PCI_TO_USER (8000 DMA)
Sets the DMA transfer direction from a source PCI address → a user mode provided buffer

PLX_DMA_MAX_SRC_TSIZE

```
typedef enum _PLX_DMA_MAX_SRC_TSIZE
{
    PLX_DMA_MAX_SRC_TSIZE_64B  = 0,
    PLX_DMA_MAX_SRC_TSIZE_128B = 1,
    PLX_DMA_MAX_SRC_TSIZE_256B = 2,
    PLX_DMA_MAX_SRC_TSIZE_512B = 3,
    PLX_DMA_MAX_SRC_TSIZE_1K   = 4,
    PLX_DMA_MAX_SRC_TSIZE_2K   = 5,
    PLX_DMA_MAX_SRC_TSIZE_4K   = 7
} PLX_DMA_SRC_MAX_TSIZE;
```

Purpose

Sets the TLP Max Payload Size (MPS) when the DMA engine reads the source location. This should not exceed the MPS set by the system in the DMA PCIe Capabilities.

Members

DMA maximum transfer sizes vary from 64B → 4KB.. Refer to the member name for the maximum transfer size

PLX_EEPROM_PORT

```
typedef enum _PLX_EEPROM_PORT
{
    PLX_EEPROM_PORT_NONE        = 0,
    PLX_EEPROM_PORT_NT_VIRT_0   = 254,
    PLX_EEPROM_PORT_NT_LINK_0   = 253,
    PLX_EEPROM_PORT_NT_VIRT_1   = 252,
    PLX_EEPROM_PORT_NT_LINK_1   = 251,
    PLX_EEPROM_PORT_DMA_0       = 250,
    PLX_EEPROM_PORT_DMA_1       = 249,
    PLX_EEPROM_PORT_DMA_2       = 248,
    PLX_EEPROM_PORT_DMA_3       = 247,
    PLX_EEPROM_PORT_SHARED_MEM  = 246
} PLX_EEPROM_PORT
```

Purpose

Enumerated type used for specifying ports other than standard transparent to target in EEPROM values

Members

PLX_EEPROM_PORT_NONE
: Port type not specified

PLX_EEPROM_PORT_NT_xxx
: One of the NT-virtual or NT-Link ports

PLX_EEPROM_PORT_DMA_xxx
: One of the DMA functions

PLX_EEPROM_PORT_SHARED_MEM
: Shared memory in the PLX chip (8111/8112)

PLX_EEPROM_STATUS

```
typedef enum _PLX_EEPROM_STATUS
{
    PLX_EEPROM_STATUS_NONE         = 0,
    PLX_EEPROM_STATUS_VALID        = 1,
    PLX_EEPROM_STATUS_INVALID_DATA = 2,
    PLX_EEPROM_STATUS_BLANK        = PLX_EEPROM_STATUS_INVALID_DATA,
    PLX_EEPROM_STATUS_CRC_ERROR    = PLX_EEPROM_STATUS_INVALID_DATA
} PLX_EEPROM_STATUS;
```

Purpose

Enumerated type used for providing EEPROM status

Members

PLX_EEPROM_STATUS_NONE
EEPROM not present.

PLX_EEPROM_STATUS_VALID
EEPROM is present with valid data

PLX_EEPROM_STATUS_INVALID_DATA
EEPROM is present with invalid data or CRC error

PLX_EEPROM_STATUS_BLANK
EEPROM is blank. *Returns same value as PLX_EEPROM_STATUS_INVALID_DATA*

PLX_EEPROM_STATUS_CRC_ERROR
EEPROM has CRC error. *Returns same value as PLX_EEPROM_STATUS_INVALID_DATA*

PLX_NT_LUT_FLAG

```
typedef enum _PLX_NT_LUT_FLAG
{
    PLX_NT_LUT_FLAG_NONE        = 0,
    PLX_NT_LUT_FLAG_NO_SNOOP    = (1 << 0),
    PLX_NT_LUT_FLAG_READ        = (1 << 1),
    PLX_NT_LUT_FLAG_WRITE       = (1 << 2)
} PLX_NT_LUT_FLAG;
```

Purpose

Enumerated type used for reporting NT port type

Members

PLX_NT_LUT_FLAG_NONE
    No active flags

PLX_NT_LUT_FLAG_NO_SNOOP
    Enables the ***No_Snoop disable*** option for the LUT entry

PLX_NT_LUT_FLAG_READ
    Enables memory read TLP access *(Not supported in current PLX chips)*

PLX_NT_LUT_FLAG_WRITE
    Enables memory write TLP access *(Not supported in current PLX chips)*

PLX_NT_PORT_TYPE

```
typedef enum _PLX_NT_PORT_TYPE
{
    PLX_NT_PORT_NONE            = 0,                    // Not an NT port
    PLX_NT_PORT_PRIMARY         = 1,                    // NT Primary Host side
    PLX_NT_PORT_SECONDARY       = 2,                    // NT Seconday Host side
    PLX_NT_PORT_VIRTUAL         = PLX_NT_PORT_PRIMARY,  // NT Virtual-side port
    PLX_NT_PORT_LINK            = PLX_NT_PORT_SECONDARY,// NT Link-side port
    PLX_NT_PORT_UNKOWN          = 0xFF                  // NT side undetermined
} PLX_NT_PORT_TYPE;
```

Purpose

Enumerated type used for reporting NT port type

Members

PLX_NT_PORT_NONE
: Port is not an NT port

PLX_NT_PORT_PRIMARY
: Port is located on the primary side of the PLX chip

PLX_NT_PORT_SECONDARY
: Port is located on the secondary side of the PLX chip

PLX_NT_PORT_VIRTUAL
: Same as *PLX_NT_PORT_PRIMARY*

PLX_NT_PORT_LINK
: Same as *PLX_NT_PORT_SECONDARY*

PLX_NT_PORT_UNKNOWN
: PLX driver was unable to determine NT port side

PLX_PERF_CMD

```
typedef enum _PLX_PERF_CMD
{
    PLX_PERF_CMD_START,
    PLX_PERF_CMD_STOP,
} PLX_PERF_CMD;
```

Purpose

Commands to control the PLX Performance Counters

Members

PLX_PERF_CMD_START
    Starts the Performance Counters

PLX_PERF_CMD_STOP
    Stops the Performance Counters

PLX_PORT_TYPE

```
typedef enum _PLX_PORT_TYPE
{
    PLX_PORT_UNKNOWN            = 0xFF,
    PLX_PORT_ENDPOINT           = 0,
    PLX_PORT_NON_TRANS          = PLX_PORT_ENDPOINT,  // NT port is an endpoint
    PLX_PORT_LEGACY_ENDPOINT    = 1,
    PLX_PORT_ROOT_PORT          = 4,
    PLX_PORT_UPSTREAM           = 5,
    PLX_PORT_DOWNSTREAM         = 6,
    PLX_PORT_PCIE_TO_PCI_BRIDGE = 7,
    PLX_PORT_PCI_TO_PCIE_BRIDGE = 8,
    PLX_PORT_ROOT_ENDPOINT      = 9,
    PLX_PORT_ROOT_EVENT_COLL    = 10
} PLX_PORT_TYPE;
```

Purpose

Enumerated type used for providing port type information.

Members

N/A

PLX_STATE

```
typedef enum _PLX_STATE
{
    PLX_STATE_OK,
    PLX_STATE_WORKING,
    PLX_STATE_ERROR,
    PLX_STATE_ENABLED,
    PLX_STATE_DISABLED,
    PLX_STATE_INITIALIZING,
    PLX_STATE_INITIALIZED,
    PLX_STATE_IDLE,
    PLX_STATE_BUSY,
    PLX_STATE_STARTED,
    PLX_STATE_STARTING,
    PLX_STATE_STOPPED,
    PLX_STATE_STOPPING,
    PLX_STATE_CANCELED,
    PLX_STATE_DELETED,
    PLX_STATE_MARKED_FOR_DELETE,
    PLX_STATE_OK_TO_DELETE,
    PLX_STATE_TRIGGERED,
    PLX_STATE_PENDING,
    PLX_STATE_WAITING,
    PLX_STATE_REQUESTING,
    PLX_STATE_REQUESTED,
    PLX_STATE_ACCEPTING,
    PLX_STATE_ACCEPTED,
    PLX_STATE_REJECTED,
    PLX_STATE_COMPLETING,
    PLX_STATE_COMPLETED,
    PLX_STATE_CONNECTING,
    PLX_STATE_CONNECTED,
    PLX_STATE_DISCONNECTING,
    PLX_STATE_DISCONNECTED
} PLX_STATE;
```

Purpose

Enumerated type to provide generic states for general use

Members

Self-explanatory

PLX_STATUS

```
typedef enum _PLX_STATUS
{
    ApiSuccess,
    ApiFailed,
    ApiNullParam,
    ApiUnsupportedFunction,
    ApiNoActiveDriver,
    ApiConfigAccessFailed,
    ApiInvalidDeviceInfo,
    ApiInvalidDriverVersion,
    ApiInvalidOffset,
    ApiInvalidData,
    ApiInvalidSize,
    ApiInvalidAddress,
    ApiInvalidAccessType,
    ApiInvalidIndex,
    ApiInvalidPowerState,
    ApiInvalidIopSpace,
    ApiInvalidHandle,
    ApiInvalidPciSpace,
    ApiInvalidBusIndex,
    ApiInsufficientResources,
    ApiWaitTimeout,
    ApiWaitCanceled,
    ApiDmaChannelUnavailable,
    ApiDmaChannelInvalid,
    ApiDmaDone,
    ApiDmaPaused,
    ApiDmaInProgress,
    ApiDmaCommandInvalid,
    ApiDmaInvalidChannelPriority,
    ApiDmaSglPagesGetError,
    ApiDmaSglPagesLockError,
    ApiMuFifoEmpty,
    ApiMuFifoFull,
    ApiPowerDown,
    ApiHSNotSupported,
    ApiVPDNotSupported,
    ApiDeviceInUse,
    ApiPending,
    ApiObjectNotFound,
    ApiLastError               // Do not add API errors below this line
} PLX_STATUS;
```

Purpose

Enumerated type used for providing PLX status codes for all PLX API functions

Members

N/A

PLX_SWITCH_MODE

```
typedef enum _PLX_SWITCH_MODE
{
    PLX_SWITCH_MODE_STANDARD   = 0,
    PLX_SWITCH_MODE_MULTI_HOST = 2
} PLX_SWITCH_MODE;
```

Purpose

Enumerated type used for providing mode switch is in.

Members

PLX_SWITCH_MODE_STANDARD
Switch is in standard single-host mode.

PLX_SWITCH_MODE_MULTI_HOST
Switch is in multi-host mode.

5.2.3 Data Structures

This section contains the enumerated data types used in the PLX API.

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;
    U16              PlxChip;        // PLX chip type
    U8               PlxRevision;    // PLX chip revision
    U8               PlxFamily;      // PLX chip family
    U8               ApiIndex;       // Used internally by the API
    U8               DeviceNumber;   // Used internally by device drivers
    PLX_API_MODE     ApiMode;        // Mode API uses to access device
    U8               PlxPort;        // PLX port number of device
    PLX_NT_PORT_TYPE NTPortType;     // If NT port, stores NT port type
    U32              ApiInternal[2]; // Reserved for internal PLX API use
} PLX_DEVICE_KEY;
```

Purpose

Uniquely identifies a PCI device in a system. The values in the key are used throughout the PLX API and drivers and should not be modified.

Members

IsValidTag
Reserved for internal use by the PLX API

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

PlxChip
The PLX chip type. Will be 0 if non-PLX chip.

PlxRevision
The PLX chip revision

PlxFamily
The PLX chip family. *Refer to PLX_CHIP_FAMILY.*

ApiIndex
Reserved for internal use by the PLX API

DeviceNumber
Reserved for internal use by PLX device drivers

ApiMode
Mode the PLX API is using to access the device (e.g. PCI, I^2C, TCP). *Refer to PLX_API_MODE.*

PlxPort
The PCI Express port number of the PLX device

NTPortType
If the port is NT, specifies the NT port type (i.e. NT-Virtual or NT-Link side). *Refer to PLX_NT_PORT_TYPE*.

PLX_DEVICE_OBJECT

```
typedef struct _PLX_DEVICE_OBJECT
{
    U32                IsValidTag;   // Magic number to determine validity
    PLX_DEVICE_KEY     Key;          // Device location key identifier
    PLX_DRIVER_HANDLE  hDevice;      // Handle to driver
    PLX_PCI_BAR_PROP   PciBar[6];    // PCI BAR properties
    U64                PciBarVa[6];  // For PCI BAR user-mode BAR mappings
    U8                 BarMapRef[6]; // BAR map count used by API
    PLX_PHYSICAL_MEM   CommonBuffer; // Used to store common buffer information
    VOID              *pPrivateData; // Pointer storage for a user private buffer
} PLX_DEVICE_OBJECT;
```

Purpose

Opaque structure that describes a selected PCI device object.

Members

*The members in this object, other than **pPrivateData**, should never be accessed directly. The structure definition may change in future SDK versions and its members are reserved for internal use by the PLX API and PLX drivers.*

pPrivateData

Pointer in the device object which an application may use. The PLX API will not access or modify this pointer. May be useful if an application needs a private data buffer associated with an open device. The application is responsible for allocation and release of this buffer.

PLX_DMA_PARAMS

```
typedef struct _PLX_DMA_PARAMS
{
    U64         UserVa;
    U64         AddrSource;
    U64         AddrDest;
    U64         PciAddr;
    U32         LocalAddr;
    U32         ByteCount;
    PLX_DMA_DIR Direction;
    U8          bConstAddrSrc   :1;
    U8          bConstAddrDest  :1;
    U8          bForceFlush     :1;
    U8          bIgnoreBlockInt :1;
```

Purpose

Structure used to provide the parameters for a DMA transfer.

Members

UserVa
Specifies the virtual address of the user-mode buffer for the DMA transfer.

AddrSource (8000 DMA)
Specifies the source PCI address for a DMA block transfer.

AddrDest (8000 DMA)
Specifies the destination PCI address for a DMA block transfer.

PciAddr (9000 DMA)
Specifies the PCI address for a DMA block transfer. Can be 64-bit.

LocalAddr (9000 DMA)
The 32-bit local bus address for the DMA transfer.

ByteCount
The number of bytes to transfer.

Direction
Specifies the direction of the DMA transfer. Refer to PLX_DMA_DIR.

bConstAddrSrc (8000 DMA)
Specifies that the source PCI address should not be incremented

bConstAddrDest (8000 DMA)
Specifies that the destination PCI address should not be incremented

bForceFlush (8000 DMA)
Forces the DMA to use a write flush to ensure data in the final descriptor is written before the DMA engine reports DMA completion.

bIgnoreBlockInt
Specifies to disable the DMA done interrupt for the transfer. Typically used if DMA done polling is desired to eliminate the overhead of handling the DMA done interrupt. Applies only for DMA block mode transfers.

PLX_DMA_PROP

```
typedef struct _PLX_DMA_PROP
{
    // 8000 DMA properties
    U8  CplStatusWriteBack  :1;
    U8  DescriptorMode      :2;
    U8  DescriptorPollMode  :1;
    U8  RingHaltAtEnd       :1;
    U8  RingWrapDelayTime   :3;
    U8  RelOrderDescrRead   :1;
    U8  RelOrderDescrWrite  :1;
    U8  RelOrderDataReadReq :1;
    U8  RelOrderDataWrite   :1;
    U8  NoSnoopDescrRead    :1;
    U8  NoSnoopDescrWrite   :1;
    U8  NoSnoopDataReadReq  :1;
    U8  NoSnoopDataWrite    :1;
    U8  MaxSrcXferSize      :3;
    U8  MaxDestWriteSize    :3;
    U8  TrafficClass        :3;
    U8  MaxPendingReadReq   :6;
    U8  DescriptorPollTime;
    U8  MaxDescriptorFetch;
    U16 ReadReqDelayClocks;

    // 9000 DMA properties
    U8  ReadyInput          :1;
    U8  Burst               :1;
    U8  BurstInfinite       :1;
    U8  SglMode             :1;
    U8  DoneInterrupt       :1;
    U8  RouteIntToPci       :1;
    U8  ConstAddrLocal      :1;
    U8  WriteInvalidMode    :1;
    U8  DemandMode          :1;
    U8  EnableEOT           :1;
    U8  FastTerminateMode   :1;
    U8  ClearCountMode      :1;
    U8  DualAddressMode     :1;
    U8  EOTEndLink          :1;
    U8  ValidMode           :1;
    U8  ValidStopControl    :1;
    U8  LocalBusWidth       :2;
    U8  WaitStates          :4;
} PLX_DMA_PROP;
```

Purpose

Structure used to configure the DMA channel properties. For all one-bit values, 0=disable and 1=disable.

Members

8000 DMA

CplStatusWriteBack
In ring mode, determines whether DMA updates the first DWORD in a DMA descriptor to provide status information and clear valid bit after the transfer has completed for that descriptor.
(0 = No write back, 1 = Update descriptor with status information)

DescriptorMode
Sets the DMA to Block or Ring/SGL mode. Refer to PLX_DMA_DESCR_MODE.

DescriptorPollMode
** Not available in current DMA hardware Reserved for future use, set to 0. **

RingHaltAtEnd
Determines whether DMA halts when it reaches end of ring or wraps back to beginning.
(0 = Wrap, 1 = Halt)

RingWrapDelayTime
If *RingHaltAtEnd* is disabled, determines the delay before the DMA wraps to the start of the ring. Refer to PLX_DMA_RING_DELAY_TIME

RelOrderDescrRead
Use PCIe Relaxed Ordering for descriptor reads

RelOrderDescrWrite
Use PCIe Relaxed Ordering for descriptor writes

RelOrderDataReadReq
Use PCIe Relaxed Ordering for DMA data read requests

RelOrderDataWrite
Use PCIe Relaxed Ordering for DMA data writes

NoSnoopDescrRead
Set TLP No Snoop for descriptor reads

NoSnoopDescrWrite
Set TLP No Snoop for descriptor writes

NoSnoopDataReadReq
Set TLP No Snoop for DMA read requests

NoSnoopDataWrite
Set TLP No Snoop for DMA data writes

MaxSrcXferSize
Sets the maximum TLP read request size the DMA engine may request from the source address. Refer to PLX_DMA_MAX_SRC_TSIZE.

MaxDestWriteSize (*Not supported on 8600 DMA*)
Sets the maximum payload size to write to the destination

TrafficClass
Sets the PCI Express Traffic Class used for DMA transfers

MaxPendingReadReq
Determines the maximum number of pending DMA read requests from the source.

DescriptorPollTime
** Not available in current DMA hardware Reserved for future use, set to 0. **

MaxDescriptorFetch
Sets the maximum number of descriptors to prefetch at any given time

ReadReqDelayClocks
Sets the number of clocks between DMA data read requests. May be used to slow down DMA traffic.

9000 DMA

ReadyInput
Enables the Ready input (READY#)

Burst
Enables bursting for the Local bus (Burst of 4LW if *BurstInfinite* not enabled).

BurstInfinite
Enables the BTERM# input if set, which allows for infinite bursting. (*Burst* must also be set)

SglMode
Sets DMA to operate in Scatter-Gather List (SGL) mode

DoneInterrupt
Enables the DMA done interrupt

RouteIntToPci
Set the DMA interrupt to assert to the PCI side. If not set, DMA interrupt to assert on local-side.

ConstAddrLocal
Prevents the DMA engine from incrementing the local bus address

WriteInvalidMode
Enables PCI write and invalidate cycles for DMA transfers

DemandMode
Enables DMA Demand mode if set.

EnableEOT
Enables the DMA EOT# input pin

FastTerminateMode
Specifies the DMA termination mode. *0=Slow, 1=Fast*

ClearCountMode
Enable SGL DMA transfer count clear mode if set. The DMA engine will clear the transfer count of each descriptor once the data has been transferred for that descriptor.

DualAddressMode
Enables DMA dual address cycles for DMA transfers. In block mode, the upper 32-bits of the PCI address are taken from the Dual Address Cycle register. In SGL mode, SGL descriptors become 5 DWORDs instead of the standard 4 DWORDS for 32-bit transfers. The 5th DWORD in each descriptor specifies the upper 32-bits of the PCI address, which will be loaded into the Dual-Address Cycle register.

EOTEndLink
Controls DMA descriptor processing when EOT# is asserted during a DMA SGL transfer. If set (=1), when EOT# is asserted, the DMA controller halts the current SGL transfer and continues to the next descriptor. If not set (=0), when EOT# is asserted, the DMA transfer halts the current SGL transfer, but does not continue to the next descriptor.

ValidMode
Enables DMA descriptor valid mode. The DMA descriptor fetch will then only retrieve descriptors with the valid bit set.

ValidStopControl
Controls whether the DMA engine continuously polls (=0) the current descriptor's valid bit or halts the descriptor fetch (=1) when an invalid descriptor is reached.

LocalBusWidth

Specifies the local bus width for DMA transfers. *0=8-bit, 1=16-bit, 2=32-bit*

WaitStates

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

PLX_DRIVER_PROP

```
typedef struct _PLX_DRIVER_PROP
{
    U32     Version;
    char    Name[16];
    char    FullName[255];
    BOOLEAN bIsServiceDriver;
    U64     AcpiPcieEcam;
    U8      Reserved[40];
} PLX_DRIVER_PROP;
```

Purpose

Structure used to report properties of the selected PLX device driver.

Members

Version
Returns the driver version in the form Major[19:16], Minor[15:8]

Name
Returns the string name of the PLX driver being used to access the selected device

FullName
Returns the full user-friendly string name of the PLX driver being used to access the selected device

bIsServiceDriver
Returns TRUE if the PLX PCI/PCIe Service driver is being used to access the device; otherwise, a value of FALSE is returned to indicate a PLX Plug 'n' Play driver is being used.

AcpiPcieEcam
If available, returns the ACPI Enhanced Configuration Address Mechanism (ECAM) base address. The ECAM is specified in the *PCI Express Specification* and contains the memory mapped PCI configuration space for all PCI devices in the system. PLX drivers utilize this region when PCI extended configuration registers are accessed (offsets 100h & above). PLX drivers probe ACPI tables in the system to determine this address.

PLX_INTERRUPT

```
typedef struct _PLX_INTERRUPT
{
    U32 Doorbell;                       // Up to 32 doorbells
    U8  PciMain                :1;
    U8  PciAbort               :1;
    U8  LocalToPci             :2;    // Local->PCI int 1 & 2
    U8  DmaDone                :4;    // DMA channel 0-3 interrrupts
    U8  DmaPauseDone           :4;
    U8  DmaAbortDone           :4;
    U8  DmaImmedStopDone       :4;
    U8  DmaInvalidDescr        :4;
    U8  DmaError               :4;
    U8  MuInboundPost          :1;
    U8  MuOutboundPost         :1;
    U8  MuOutboundOverflow     :1;
    U8  TargetRetryAbort       :1;
    U8  Message                :4;    // 6000 NT 0-3 message interrupts
    U8  SwInterrupt            :1;
    U8  ResetDeassert          :1;
    U8  PmeDeassert            :1;
    U8  GPIO_4_5               :1;
    U8  GPIO_14_15             :1;
    U8  NTV_LE_Correctable     :1;   // NT Virtual - Link-side error interrupts
    U8  NTV_LE_Uncorrectable   :1;
    U8  NTV_LE_LinkStateChange :1;
    U8  NTV_LE_UncorrErrorMsg  :1;
    U8  HotPlugAttention       :1;
    U8  HotPlugPowerFault      :1;
    U8  HotPlugMrlSensor       :1;
    U8  HotPlugChangeDetect    :1;
    U8  HotPlugCmdCompleted    :1;
} PLX_INTERRUPT;
```

Purpose

Contains the supported PLX device interrupts used to return active interrupts, enable/disable interrupts, or select certain interrupts. For all one-bit values, 0=disable and 1=disable.

For multi-bit interrupts, interrupt numbers are associated with bit positions. For example, the ***DmaDone*** field is 4 bits, representing up to 4 DMA channel done interrupts. Bit 0 = Channel 0, Bit 1 = Channel 1, Bit 2 = Channel 2, & Bit 3 = Channel 3.

Members

Doorbell
Represents up to 32 (0→31) doorbell interrupts

PciMain
Represents the main PCI interrupt line. This field is only used in interrupt enable/disable API functions.

PciAbort
Represents the PCI abort interrupt.

LocalToPci
Represents the generic Local→PCI interrupts (bit 0 = L→P #1, bit 1 = L→P #2)

DmaDone
Represents the DMA channel transfer complete interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

DmaPauseDone
Represents the DMA pause complete interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

DmaAbortDone
Represents the DMA abort complete interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

DmaImmedStopDone
Represents the DMA immediate pause/stop complete interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

DmaInvalidDescr
Represents the DMA invalid descriptor detected interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

DmaError
Represents the general DMA error interrupts (bit 0=Ch 0, bi1=Ch 1, etc)

MuInboundPost
Represents the messaging unit's inbound post FIFO interrupt

MuOutboundPost
Represents the messaging unit's outbound post FIFO interrupt

MuOutboundOverflow
The value represents the messaging unit's outbound FIFO overflow interrupt

TargetRetryAbort
Represents the PLX chip's Target Abort interrupt after 256 Master consecutive retries to the target

Message
For 6254/6540/6466 NT mode, represents the four message interrupts (bit 0=Msg 0, bit 1=Msg 1, etc.)

SwInterrupt
Represents the Software-triggered interrupt of PLX 9000 slave devices (9050/9052/9030)

ResetDeassert
For 6254/6540/6466, represents S_RSTIN# or P_RSTIN# de-assertion interrupt

PmeDeassert
For 6254/6540/6466, represents S_PME# or P_PME# de-assertion interrupt

GPIO_4_5
For 6254/6540/6466, represents GPIO4 (primary-side) or GPIO5 (secondary-side) interrupt

GPIO_14_15
For 6254/6540/6466, represents GPIO14 (primary-side) or GPIO15 (secondary-side) interrupt

NT_LE_Correctable
(8000-series NT Virtual side) NT Link interface detected a correctable TLP error

NT_LE_Uncorrectable
(8000-series NT Virtual side) NT Link interface detected an uncorrectable TLP error

NT_LE_LinkStateChange
(8000-series NT Virtual side) Link interface link state changed (Link Down or Link Up)

NT_LE_UncorrErrorMsg
(8000-series NT Virtual side) Link interface received and uncorrectable error message TLP

HotPlugAttention
Represents the Hot Plug Attention button pressed interrupt.

HotPlugPowerFault
Represents the Hot Plug Power Fault interrupt

HotPlugMrlSensor
Represents the Hot Plug MRL Sensor interrupt

HotPlugChangeDetect
Represents the Hot Plug Change Detected interrupt

HotPlugCmdCompleted
Represents the Hot Plug Command Completed interrupt

PLX_MULTI_HOST_PROP

```
typedef struct _PLX_MULTI_HOST_PROP
{
    U8      SwitchMode;
    U16     VS_EnabledMask;
    U8      VS_UpstreamPortNum[8];
    U32     VS_DownstreamPorts[8];
    BOOLEAN bIsMgmtPort;
    BOOLEAN bMgmtPortActiveEn;
    U8      MgmtPortNumActive;
    BOOLEAN bMgmtPortRedundantEn;
    U8      MgmtPortNumRedundant;
} PLX_MULTI_HOST_PROP;
```

Purpose

Contains properties of PLX multi-root switches.

Members

SwitchMode
Current switch mode. Refer to PLX_SWITCH_MODE.

VS_EnabledMask
Bit for each enabled Virtual Switch

VS_UpstreamPortNum
Upstream port number of each Virtual Switch

VS_DownstreamPorts
Downstream ports associated with a Virtual Switch

bIsMgmtPort
Specifies whether the selected port is the management port. Will always be TRUE in standard host mode. In Multi-host mode, properties are only available through the management port; otherwise, they are invalid.

bMgmtPortActiveEn
Specifies whether the active management port is enabled

MgmtPortNumActive
Active management port number

bMgmtPortRedundantEn
Specifies whether the redundant management port is enabled

MgmtPortNumRedundant
Redundant management port number

PLX_MODE_PROP

```
typedef struct _PLX_MODE_PROP
{
    union
    {
        struct
        {
            U16 I2cPort;
            U16 SlaveAddr;
            U32 ClockRate;
        } I2c;

        struct
        {
            U64 IpAddress;
        } Tcp;
    };
} PLX_MODE_PROP;
```

Purpose

Used to provide API mode properties for finding/selecting a device.

Members

I2c.I2cPort
Contains the port number for the I^2C USB device to use. For Aardvark I^2C, starts at '0'.

I2c.SlaveAddr
The I^2C bus address assigned to the PLX chip to access.

I2c.ClockRate
Specifies the I^2C clock rate in KHz

Tcp.IpAddress
Specifies the TCP IP address of the device to access (*not currently supported*)

PLX_NOTIFY_OBJECT

```
typedef struct _PLX_NOTIFY_OBJECT
{
    U32 IsValidTag;         // Magic number to determine validity
    U64 pWaitObject;        // -- INTERNAL -- Wait object used by the driver
    U64 hEvent;             // User event handle (HANDLE can be 32 or 64 bit)
} PLX_NOTIFY_OBJECT;
```

Purpose

Opaque structure that used for interrupt notification functions

Members

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

PLX_PCI_BAR_PROP

```
typedef struct _PLX_PCI_BAR_PROP
{
    U32      BarValue;
    U64      Physical;
    U64      Size;
    BOOLEAN  bIoSpace;
    BOOLEAN  bPrefetchable;
    BOOLEAN  b64bit;
} PLX_PCI_BAR_PROP;
```

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

BarValue
Actual value in the PCI BAR register

Physical
The physical address assigned to the BAR

Size
The size of the BAR space

bIoSpace
TRUE if the BAR space is type I/O; FALSE if BAR space is type Memory

bPrefetchable
TRUE if the BAR space is configured as Prefetchable; FALSE if not

b64bit
TRUE if the BAR is a 64-bit space; FALSE if it is 32-bit

PLX_PERF_PROP

```
typedef struct _PLX_PERF_PROP
{
    U32 IsValidTag;   // Magic number to determine validity

    // Port properties
    U8  PortNumber;
    U8  LinkWidth;
    U8  LinkSpeed;
    U8  Station;
    U8  StationPort;

    // Ingress counters
    U32 IngressPostedHeader;
    U32 IngressPostedDW;
    U32 IngressNonpostedDW;
    U32 IngressCplHeader;
    U32 IngressCplDW;
    U32 IngressDllp;
    U32 IngressPhy;

    // Egress counters
    U32 EgressPostedHeader;
    U32 EgressPostedDW;
    U32 EgressNonpostedDW;
    U32 EgressCplHeader;
    U32 EgressCplDW;
    U32 EgressDllp;
    U32 EgressPhy;

    // Previous Ingress counters
    U32 Prev_IngressPostedHeader;
    U32 Prev_IngressPostedDW;
    U32 Prev_IngressNonpostedDW;
    U32 Prev_IngressCplHeader;
    U32 Prev_IngressCplDW;
    U32 Prev_IngressDllp;
    U32 Prev_IngressPhy;

    // Previous Egress counters
    U32 Prev_EgressPostedHeader;
    U32 Prev_EgressPostedDW;
    U32 Prev_EgressNonpostedDW;
    U32 Prev_EgressCplHeader;
    U32 Prev_EgressCplDW;
    U32 Prev_EgressDllp;
    U32 Prev_EgressPhy;
}
```

Purpose

Used to store the current and previous performance counters obtained from the PLX chip.

Members

These members are not documented because they are reserved for internal use by PLX software tools.

PLX_PERF_STATS

```
typedef struct _PLX_PERF_PROP
{
    S64         IngressTotalBytes;          // Total bytes including overhead
    long double IngressTotalByteRate;       // Total byte rate
    S64         IngressCplAvgPerReadReq;    // Avg completion TLPs per read req
    S64         IngressCplAvgBytesPerTlp;   // Avg bytes per completion TLP
    S64         IngressPayloadReadBytes;    // Payload bytes read (Cpl TLPs)
    S64         IngressPayloadReadBytesAvg; // Avg read payload bytes (Cpl TLPs)
    S64         IngressPayloadWriteBytes;   // Payload bytes written (Posted TLPs)
    S64         IngressPayloadWriteBytesAvg;// Avg write payload bytes (P. TLPs)
    S64         IngressPayloadTotalBytes;   // Payload total bytes
    double      IngressPayloadAvgPerTlp;    // Payload average size per TLP
    long double IngressPayloadByteRate;     // Payload byte rate
    long double IngressLinkUtilization;     // Total link utilization

    S64         EgressTotalBytes;           // Total byte including overhead
    long double EgressTotalByteRate;        // Total byte rate
    S64         EgressCplAvgPerReadReq;     // Avg completion TLPs per read req
    S64         EgressCplAvgBytesPerTlp;    // Avg bytes per completion TLPs
    S64         EgressPayloadReadBytes;     // Payload bytes read (Cpl TLPs)
    S64         EgressPayloadReadBytesAvg;  // Avg read payload bytes (Cpl TLPs)
    S64         EgressPayloadWriteBytes;    // Payload bytes written (Posted TLPs)
    S64         EgressPayloadWriteBytesAvg; // Avg write payload bytes (P. TLPs)
    S64         EgressPayloadTotalBytes;    // Payload total bytes
    double      EgressPayloadAvgPerTlp;     // Payload average size per TLP
    long double EgressPayloadByteRate;      // Payload byte rate
    long double EgressLinkUtilization;      // Total link utilization
}
```

Purpose

Used to store the calculated performance values for a particular port

Members

These members are not documented because they are reserved for internal use by PLX software tools.

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.

Notes

The CPU address is the physical address from the point of view of the CPU. The Bus or Logical physical address is the address from the point of view of a device. The bus address should be used when programming PCI addresses in hardware (e.g. DMA controllers). On x86 platforms, CPU and Logical addresses are the same because no I/O Memory Management Unit (IOMMU) exists on these systems. On other platforms, the CPU address may not be equal to the Logical address.

PLX software already includes placeholders for the various addresses. If the correct field is used when code is written, applications should work properly on all target platforms, regardless of whether an IOMMU exists or not.

PLX_PORT_PROP

```
typedef struct _PLX_PORT_PROP
{
    U8      PortType;
    U8      PortNumber;
    U8      LinkWidth;
    U8      MaxLinkWidth;
    U8      LinkSpeed;
    U8      MaxLinkSpeed;
    U16     MaxReadReqSize;
    U16     MaxPayloadSize;
    U16     MaxPayloadSupported;
    BOOLEAN bNonPcieDevice;
} PLX_PORT_PROP;
```

Purpose

Structure used to report PCI Express port properties.

Members

PortType
Contains the port type (*refer to PLX_PORT_TYPE*)

PortNumber
Contains the port number

LinkWidth
Specifies the negotiated link width

MaxLinkWidth
Specifies the maximum link width the device is capable of

LinkSpeed
Specifies the negotiated link speed (*1 = 2.5 Gbps, 2 = 5 Gbps*)

MaxLinkSpeed
Specifies the maximum link speed the device is capable of

MaxReadReqSize
Specifies the maximum amount of data the device may request in a single PCI Express read packet

MaxPayloadSize
Specifies the current maximum TLP payload size (MPS) setting in the device

MaxPayloadSupported
Specifies the maximum TLP payload size (MPS) supported by the device

bNonPcieDevice
Flag to specify whether the device is not a PCI Express device (i.e. does not support PCI Express Capability)

PLX_VERSION

```
typedef struct _PLX_VERSION
{
    PLX_API_MODE ApiMode;

    union
    {
        struct
        {
            U16 ApiLibrary;
            U16 Software;
            U16 Firmware;
            U16 Hardware;
            U16 SwReqByFw;
            U16 FwReqBySw;
            U16 ApiReqBySw;
            U32 Features;
        } I2c;
    };
} PLX_VERSION;
```

Purpose

Structure used to report version information. All 16-bit version numbers are in the format (Major << 8) | (Minor). For example, the number 0114h = v1.20.

Members

ApiMode
Contains the ApiMode that the version information is for. This determines which union in the structure is contains valid information. (R*efer to PLX_API_MODE*)

*I2c.*ApiLibrary
Version of the I2C API libraray

I2c.Software
Version of the I2C software

I2c.Firmware
Version of the firmware in the I2C USB device

I2c.Hardware
Version of the I2C USB hardware

I2c.SwReqByFw
Firmware requires that software version must be >= this version

I2c.ApiReqBySw
Software requires that the API version must be >= this version

I2c.Features
Bitmask of features supported by the device. At the time of this writing, these are the features:

```
#define AA_FEATURE_SPI          0x00000001
#define AA_FEATURE_I2C          0x00000002
#define AA_FEATURE_GPIO         0x00000008
#define AA_FEATURE_I2C_MONITOR  0x00000010
```