PLX PCI SDK

Software Development Kit

Version 2.1

Revision 1

Programmer's Reference Manual

Copyright © 1998, PLX Technology, Inc.. All rights reserved.

This document contains proprietary and confidential information of PLX Technology Inc. (PLX). The contents of this document may not be copied nor duplicated in any form, in whole or in part, without prior written consent from PLX.

PLX provides the information and data included in this document for your benefit, but it is not possible for us to entirely verify and test all of this information in all circumstances, particularly information relating to non-PLX manufactured products. PLX makes no warranties or representations relating to the quality, content or adequacy of this information. Every effort has been made to ensure the accuracy of this manual, however, PLX assumes no responsibility for any errors or omissions in this document. PLX shall not be liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual or the examples herein. PLX assumes no responsibility for any damage or loss resulting from the use of this manual; for any loss or claims by third parties which may arise through the use of this SDK; for any loss or claims by third parties which may arise through the use of this SDK; and for any damage or loss caused by deletion of data as a result of malfunction or repair. The information in this document is subject to change without notice.

Product and Company names are trademarks or registered trademarks of their respective owners.

Document number: PCISDK-REF-211

Table of Contents

1. GENERAL INFORMATION 1-1

1.1 About This Manual 1-1
1.2 Where To Go From Here 1-1
1.3 Conventions 1-1
1.3.1 Programming Conventions 1-1
1.4 Terminology 1-1
1.5 Customer Support 1-2

2. CREATING APPLICATIONS USING THE PLX API 2-1

2.1 Customizing The Board Support Package (BSP) 2-1
2.1.1 Customizing The *Main()* Function 2-1
2.1.2 Customizing The Microprocessor Initialization Function 2-4
2.1.2.1 MPC860 Initialization 2-4
2.1.2.2 IBM 401GF Initialization 2-5
2.1.2.3 Hitachi SH-3 Initialization 2-7
2.1.2.4 IDT RC32364 Initialization 2-8
2.1.3 Customizing The Board Initialization Function 2-8
2.1.4 Initialization Of PCI Devices for PCI 9080RDK-RC32364 2-15
2.2 IOP Applications 2-21
2.2.1 Setup Of The Development Environment 2-21
2.2.2 Rebuilding The Hello World Application 2-22
2.2.3 Building Custom Applications 2-22
2.3 Windows Applications 2-23
2.3.1 Creating A MS Developer's Studio Project File 2-23
2.3.2 Building A Custom Application 2-24

3. IOP API FUNCTION DESCRIPTION 3-1

3.1 IOP API Function Quick Reference List 3-1
3.2 IOP API Functions Details 3-3
PlxBusPciRead 3-4
PlxBusPciWrite 3-6
PlxDmaBlockChannelClose 3-8
PlxDmaBlockChannelOpen 3-9
PlxDmaBlockTransfer 3-11
PlxDmaBlockTransferRestart 3-14
PlxDmaIsr 3-15
PlxDmaResourceManagerInit 3-16
PlxDmaSglBuild 3-18
PlxDmaSglChannelClose 3-20

PlxDmaSglChannelOpen 3-21
PlxDmaSglFill 3-23
PlxDmaSglTransfer 3-25
PlxDmaShuttleChannelClose 3-28
PlxDmaShuttleChannelOpen 3-29
PlxDmaShuttleTransfer 3-31
PlxDmaShuttleTransferRestart 3-34
PlxEchoEnable 3-36
PlxGetChars 3-37
PlxHotSwapDisable 3-38
PlxHotSwapEnable 3-39
PlxHotSwapId 3-40
PlxHotSwapNextItemPointer 3-41
PlxHotSwapStatus 3-42
PlxInitApi 3-43
PlxInitDone 3-45
PlxInitIopArbitration 3-46
PlxInitIopBusProperties 3-47
PlxInitIopEndian 3-49
PlxInitLocalSpace 3-51
PlxInitPciArbitration 3-53
PlxInitPciBusProperties 3-54
PlxInitPciSpace 3-55
PlxInitPowerManagement 3-57
PlxInitVpdAddress 3-58
PlxIntrDisable 3-59
PlxIntrEnable 3-60
PlxIntrStatusGet 3-61
PlxMuInboundPortRead 3-62
PlxMuInboundPortWrite 3-63
PlxMuInit 3-64
PlxMuOutboundPortRead 3-66
PlxMuOutboundPortWrite 3-67
PlxPciConfigRegisterRead 3-68
PlxPciConfigRegisterWrite 3-70
PlxPowerLevelGet 3-72
PlxPowerLevelSet 3-73
PlxPrintf 3-74
PlxRegisterDoorbellRead 3-75
PlxRegisterDoorbellSet 3-76
PlxRegisterMailboxRead 3-77

PlxRegisterMailboxWrite 3-78
PlxRegisterRead 3-79
PlxRegisterReadAll 3-81
PlxRegisterWrite 3-83
PlxSdkVersion 3-85
PlxSerialEepromRead 3-86
PlxSerialEepromWrite 3-88
PlxUserRead 3-90
PlxUserWrite 3-91
PlxVpdDisable 3-93
PlxVpdEnable 3-94
PlxVpdId 3-95
PlxVpdNextItemPointer 3-96

4. PCI API FUNCTION DESCRIPTION 4-1

4.1 PCI API Function Quick Reference List 4-1
4.2 PCI API Functions Details 4-2
PlxBusIopRead 4-4
PlxBusIopWrite 4-6
PlxDmaBlockChannelClose 4-8
PlxDmaBlockChannelOpen 4-9
PlxDmaBlockTransfer 4-11
PlxDmaBlockTransferRestart 4-14
PlxDmaSglChannelClose 4-16
PlxDmaSglChannelOpen 4-17
PlxDmaSglTransfer 4-19
PlxDmaShuttleChannelClose 4-22
PlxDmaShuttleChannelOpen 4-23
PlxDmaShuttleTransfer 4-25
PlxIntrAttach 4-28
PlxIntrDisable 4-30
PlxIntrEnable 4-31
PlxIntrStatusGet 4-32
PlxIoPortRead 4-33
PlxIoPortWrite 4-34
PlxMuInboundPortRead 4-35
PlxMuInboundPortWrite 4-36
PlxMuOutboundPortRead 4-37
PlxMuOutboundPortWrite 4-38
PlxPciBarRangeGet 4-39
PlxPciBaseAddressesGet 4-40

PlxPciBusSearch 4-41
PlxPciCommonBufferGet 4-43
PlxPciConfigRegisterRead 4-44
PlxPciConfigRegisterReadAll 4-46
PlxPciConfigRegisterWrite 4-48
PlxPciDeviceClose 4-50
PlxPciDeviceFind 4-51
PlxPciDeviceOpen 4-53
PlxPowerLevelGet 4-55
PlxPowerLevelSet 4-56
PlxRegisterDoorbellRead 4-57
PlxRegisterDoorbellSet 4-58
PlxRegisterMailboxRead 4-59
PlxRegisterMailboxWrite 4-60
PlxRegisterRead 4-61
PlxRegisterReadAll 4-62
PlxRegisterWrite 4-63
PlxSdkVersion 4-64
PlxSerialEepromRead 4-65
PlxSerialEepromWrite 4-67
PlxUserRead 4-69
PlxUserWrite 4-70

5. PCI SDK DATA STRUCTURES 5-1

S8 and U8 Data Types 5-2
S16 and U16 Data Types 5-3
S32 and U32 Data Types 5-4
U64 Data Type 5-5
Access Type Enumerated Data Type 5-6
API Parameters Structure 5-7
BAR Space Enum Data Type 5-9
Bus Index Enum Data Type 5-10
Device Location Data Type 5-11
DMA Channel Descriptor Structure 5-12
DMA Channel Enum Data Type 5-15
DMA Channel Priority Enum Data Type 5-16
DMA Command Enum Data Type 5-17
DMA Direction Enum Data Type 5-18
DMA Resource Manager Parameters Structure 5-19
DMA Transfer Element Structure And SGL Address Structure 5-20
Echo State Enum Data Type 5-24

EEPROM Type Enum Data Type 5-25
IOP Arbitration Descriptor Structure 5-26
IOP Bus Properties Structure 5-28
IOP Endian Descriptor Structure 5-33
IOP Space Enum Data Type 5-36
Mailbox ID Enum Data Type 5-37
PCI Arbitration Descriptor Structure 5-38
PCI Bus Properties Structure 5-39
PCI Memory Data Type 5-41
PCI Space Enum Data Type 5-42
Pin State Enum Data Type 5-43
PLX Interrupt Structure 5-44
Power Level Enum Data Type 5-48
Power Management Properties Structure 5-49
USER Pin Enum Data Type 5-51
Virtual Addresses Data Type 5-52

License Agreement

PLX SOFTWARE LICENSE AGREEMENT

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

LICENSE Copyright (c) 1998 PLX Technology, Inc.

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

GENERAL

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

WARRANTY

PLX Technology, Inc. provides this PLX SDK AS IS, WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, 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 hereunder.

PLX Copyright Message Guidelines

The following copyright message must appear in all software products generated and distributed by PLX customers:

"**Copyright (c) 1998 PLX Technology, Inc.**"

Requirements:

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

1. General Information

The PCI SDK included in the development package is a powerful aide to software designers. The PCI SDK comes complete with a powerful Application Programmer's Interface (API) for both the I/O Platform (IOP) and Win32 platforms, sample IOP applications, and device drivers for Windows NT/98. We are confident that with the PCI SDK, your designs will be brought to market faster and more efficiently.

1.1 About This Manual

This manual provides design information on using the PLX APIs which consists of an IOP API and PCI (Host) API.

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

1.2 Where To Go From Here

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

Chapter 2, Creating Applications Using The PLX API, is an explanation of how the PCI SDK can be ported to different applications or boards.

Chapter 3, IOP API Function Description, provides a description of all the IOP API functions.

Chapter 4, PCI API Function Description, provides a description of all the PCI API functions.

Chapter 5, PCI SDK Data Structures, provides a description of the Data Structures used in the PCI SDK.

1.3 Conventions

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

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

1.3.1 Programming Conventions

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

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

1.4 Terminology

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

All references to Windows98 may be denoted as Win98.

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

All references to IOP (I/O Platform) throughout this manual denote the RDK board and all references to IOP software denote the software running on the RDK board.

All references to FLASH should be replaced with EPROM for users of the PCI 9080RDK-SH3 and PCI 9080RDK-RC32364 Reference Design Kits.

All references to BemL1 and Bem refer to the Back-End Monitor (BEM).

1.5 Customer Support

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

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

You may contact PLX customer support at:

Address: PLX Technology, Inc.
Attn. Technical Support
390 Potrero Avenue
Sunnyvale, CA 94086

Phone: 408-774-9060
Fax: 408-774-2169
Web: http://www.plxtech.com

You may send email to one of the following addresses:

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

2. Creating Applications Using The PLX API

2.1 Customizing The Board Support Package (BSP)

As mentioned in the section on the Board Support Package (BSP), in the PCI SDK User's Manual, the BSP has control of the *main()* function. This function starts by initializing the microprocessor and proceeds to initialize the IOP API, the PCI IC, the DMA Resource Manager and the Back-End Monitor. All the memory is tested to ensure that they are functioning properly. The *main()* function then begins the daisy chain loop where each task is called.

2.1.1 Customizing The *Main()* Function

The *main()* function is the first function that runs when the application starts. This function's responsibilities are to initialize all the components of the engineering board, including the microprocessor, the PCI IC and other peripheral devices, and to initialize all the software. The following code segment is an example of a *main()* function used with the PCI SDK. An explanation of each step in the code segment provided follows it.

The *Main()* Function:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments.This sample code is taken from MainRom.C file in PCI 9054RDK-860 BSP package.*

```
void
main(void)
{
  BEML1_INIT_PARMS bemL1InitData; /* variable for the init of BEM L1 */
  PU8 sdkVersion;               /* pointer to the SDK Version string */
  PU8 sdkReleaseDate;      /* pointer to the SDK Release Date string */
  GetCharsFxPtr BemL1GetChars;   /* Points to BemL1GetChars function */
  U8  tempChar;                   /* One temporary char */

  /* Initialize the MPC860 and its peripherals */
  InitPowerQuicc(); ❶

  /* Display PLX Logo */
  DisplayPlxLogo();

  /* Initialize the PLX 9054 device and the IOP API */
  BoardInit9054(); ❷

  /* Blink the boards LED to show that the board is still alive */
  BlinkLed(2, BLINK_SLOW); ❸

  /* Print PCI SDK Version and Release Date */
  PlxSdkVersion(&sdkVersion, &sdkReleaseDate); ❹
  PlxPrintf("\n\n** %s, %s **\n", sdkVersion, sdkReleaseDate);

  /* Initialize BEM L1 buffer structure */
  bemL1InitData.BufPtr = gBemL1Buffer; ❺
  bemL1InitData.BufLen = PCI9054_BEM_BUFFER;
```

```
  /* Initialize BemL1  */
  if (BemL1Init(&bemL1InitData) == FALSE ) ❻
    PlxSdkError(BspFailedToInitBemL1, TRUE); ❼

  /* Flush the serial input buffer */
  while ( PlxGetChars(&tempChar, 1) )
  {
  }

  /*
    Check whether the host wants to download an Application to RAM
    and if there is a RAM application, bypass the memory testing.
   */
  if ( (CheckPciDownloadToRam() == FALSE) && ❽
       (CheckSerialDownloadToRam() == FALSE))
  {
    /* Test on-board memory */
    if (TestAllMemory()) ❾
    {
      /* Blink the boards LED fast to show something is wrong */
      BlinkLed(2, BLINK_FAST);
      PlxSdkError(BspFailedToTestMemory, TRUE);
    }
  }

  /*
    This While loop is the heart of the SDK.  2 modules are called in
    a round robin style - BemL1 and AppMain.  Each module MUST return
    periodically to allow the other modules to obtain CPU cycles.
   */
  while(1) ❿
  {
    /*
      Make a call to BemL1. BemL1 will return a function pointer to
      its GetChars function.  This is used by the next level.
     */
    BemL1(PlxGetChars, &BemL1GetChars);

    /*
      Make a call to AppMain. AppMain is at the end of the chain so
      so it doesn't need to return a function pointer to its GetChars
     */
    AppMain(BemL1GetChars, NULL);
  }
  return;
}
```

❶ This function initializes the microprocessor on the board. This function is specific to which microprocessor is used and will be explained in more detail in section 2.1.2.

❷ This function initializes the IOP API and the PCI IC. This function will be explained in more detail in section 2.1.3.

❸ This function is useful for sanity checks. This BSP supplied function blinks an LED on the engineering board. This function assumes that there is an LED connected to a USERo pin on

the PCI IC. The LED is turned on for a period of time and then turned off. The duty cycle of the LED is adjustable by changing the parameter to the function.

❹ This function returns the current version of the PCI SDK.

❺ Some modules of the PCI SDK software require some buffers. The size of these buffers affects the operation of the module (and sometimes subsequent modules). In this case the BEM requires a buffer to store the data. Since this buffer determines how much data can be passed to the following modules the buffer pointer and its size are passed into the BEM when initialized.

❻ This function initializes the Back-End Monitor. As mentioned in the previous note the parameters to this function include a buffer to store the filtered data. The size of this buffer affects the operation of the module and affects other applications reliant on the filtered data returned. By passing the buffer into the module, the application designer can customize the buffers for optimal performance of the application.

❼ This function provides a description of the error that occurred. All API functions return a specific error code. This function receives that error code and prints the error code value and a brief description of the error. This function is part of the BSP support functions and can be modified to support user-defined error codes. This function depends on having the UART initialized and the *PlxPrintf()* configured properly.

❽ This function determines if a PCI download or a serial download is pending to the IOP RAM memory. If there is one pending this function waits for the download to complete and, once completed, this function jumps to the downloaded RAM application. When the RAM application terminates, this function returns TRUE and the *main()* routine continues.

❾ This function tests all the memory available on the board. This is a BSP support function and can be modified to support other memory configurations and memory types.

❿ This is the Daisy Chain Main loop. This is the main loop of the applications. This loop controls the order in which each task runs. This loop is explained in more detail in the next section.

The Daisy Chain Main Loop:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments.*

```
while(1)
{
  /*
     Make a call to BemL1. BemL1 will return a function pointer to
     its GetChars function.  This is used by the next level.
   */
  BemL1(PlxGetChars, &BemL1GetChars); ❶
  /*
     Make a call to AppMain. AppMain is at the end of the chain so
     so it doesn’t need to return a function pointer to its GetChars
   */
  AppMain(BemL1GetChars, NULL); ❷
}
```

❶ This function corresponds to the BEM Monitor task. This task takes two parameters: the first one is the *GetChars()* function for the UART Services module (being the *PlxGetChars()* function), and the second is its own version of the *GetChars()* function. Normally this is the first task that is run in the loop. When it is necessary to change the order in which the tasks are run the first parameter of this function would be the *GetChars()* function pointer for the previous task (in the daisy chain). This function supplies a function pointer to its *GetChars()* function for the next task in the loop needing data from the serial port.

❷ This function corresponds to the application's main routine. This function is replaced by any new application function. The parameters are the same as the other task, a *GetChars()* function provided by the previous task and a *GetChars()* function for the AppMain task which other tasks can use. The application task does not need to return a filtered buffer of data if there are no other applications in the daisy chain. By returning a stream of data, the application can be placed anywhere in the daisy chain without affecting the operation of any other task in the loop.

2.1.2 Customizing The Microprocessor Initialization Function

This PCI SDK currently supports four different microprocessors: Motorola MPC860, IBM PPC 401GF, IDT MIPS RC32364, and Hitachi SH3. This section will focus on customizing the BSP specifically for these microprocessors.

Note: *Before customizing the microprocessor initialization function, it is recommended to configure the MiniRom application , program it into the new engineering board's FLASH and test it. For more information on customizing the MiniRom application, refer to the PCI SDK User's Manual.*

2.1.2.1 MPC860 Initialization

In the MPC860 Initialization function the following components are initialized:

- The caching of instructions and data;
- The internal memory mapped registers for MPC860;
- The memory banks of the IOP bus including the FLASH, and SDRAM memory controllers;
- The C stack memory;
- The Synchronous External Master (for external master access to the IOP bus) and Retry pin;
- The relocation of code within the FLASH to SDRAM;
- The exception handler for system reset;
- The branching from assembly code to C coded functions;
- The MPC860 Clock and the IOP bus speed;
- The System Interface Unit (SIU);
- The Dual-Port internal RAM;
- The interrupt vector table and the ISRs for external and CPM interrupts;
- The Port C pins for RS-232 communication;
- The SCC1 for Serial communication;

- The SDMA and IDMA channel 2; and,
- The RISC timers.

For further information on the MPC860 initialization code please refer to the PCI or CompactPCI 9054 or 9080 RDK-860 BSP files. Every effort has been made to ensure that they are clear and consise and they are commented throughout.

2.1.2.2 IBM 401GF Initialization

Building The Binary File

The size of the binary images provided by the PCI SDK is exactly 128KB, and is shown in Figure 2-1.

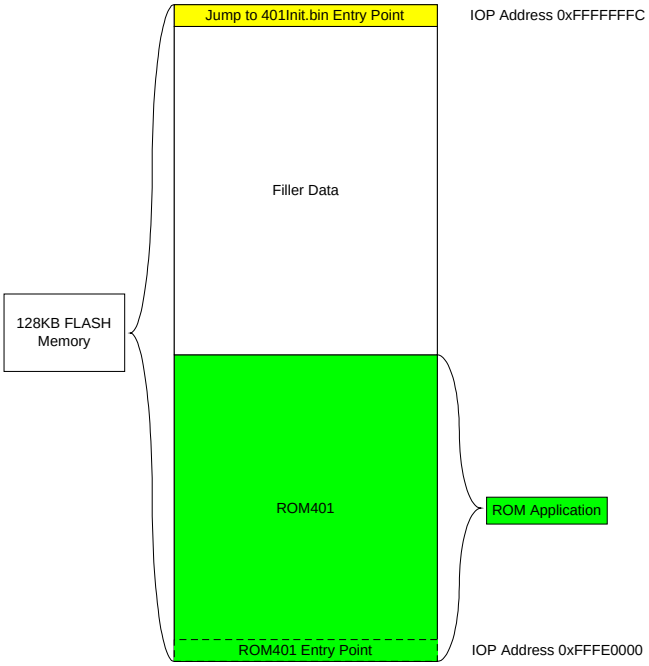

Figure 2-1 IBM FLASH Memory Map

Note: *The beginning of the binary image consists of an exception vector. Since that vector must be located on a 64K boundary, is is important to link your application at such an address.Doing otherwise could potentially hang the engineering board.*

Each image file is compiled and linked separately. Once the image file is built, it is patched with the Filler Data and one jump opcode is placed at the end of the binary file. The OS/OPEN utility, `hbranch.exe`, provides the functionality to patch the image.

The PLX 401 RDK initialization

This section explains the initialization of an IOP application. When the IBM 401GF microprocessor powers up, it starts execution at `0xFFFF FFFC`.

1. Execution starts at `0xFFFF FFFC` where a jump instruction pointing to the ROM code start address, (jumps to the `__entry` label, inside 401entry.s file).

2. The CPU performs a second jump and records the next instruction address (`__entry + 4`) into the link register. This second jump is made to the `romboot` label, located in 401init.s. The content of the link register will be used later to setup the CPU and to relocate the sections of the binary file (such as the code and data sections).
3. 401init.s file sets the microprocessor's registers allowing it to work with the specific board. The registers values are loaded from the ROM code, using the modified value of the link register. This modified value corresponds to the beginning of the 401entry.s file. The first register set is the Exception Vector Prefix (EVPR) which tells the microprocessor on which 64K boundary the exception vector is located. For the PCI SDK, the exception vector is set to `0xFFFE0000`. When the image is relocated to SRAM, this value will be updated with the equivalent SRAM address.
4. The following registers are set: the Select Little Endian Register (SLER), the Storage Guarded Register (SGR), the IO Control Register (IOCR), the Data Cache Write-through Register (DCWR) and the Bus Region Control Registers (BRCR0 – BRCR7). The values for these registers are loaded from the FLASH.
5. The following registers are setup directly, without loading their values from FLASH: the Bus Error Syndrome Register (BESR), the Exception Syndrome Register (ESR), the Timer Control Register (TCR), the Machine State Register (MSR), the Debug Status Register (DBSR) and the Fixed Point Exception Register (XER). In the PCI SDK, only the Machine Check Exception is enabled in the MSR register. The critical interrupt (for the PCI LSERR) and the external interrupt (for the PCI LINTo and UART interrupts) are not enabled at this time.
6. The reservation bit is cleared. This bit is cleared by accessing address `0x00000000`, which is in the DRAM for the RDK board.
7. The data cache and the instruction cache are invalidated, removing all data that may be in the cache.
8. The SRAM is tested. If a SRAM error is detected, the engineering board hangs.
9. The Instruction Cache Cacheability Register (ICCR) and the Data Cache Cacheability Register (DCCR) are setup by loading their values from the FLASH.
10. Once all the registers have been setup the microprocessor jumps to the `__relocation_entry` label in bootrlib.s.
11. The application is then relocated to SRAM (IOP address 0x10000000) and the data cache is flushed. This relocation is made by using the value of the link register, which did not changed since the second jump made from the `__entry` label.
12. The EVPR is set for the new code location in SRAM.

Note: *To keep an application running out of the FLASH change the command file (.cmd). Change the following lines in the command file from:*

```
#Move all sections to SRAM
SECTIONS {
 GROUP ADDRESS 0x10000000 : {
  .start:        # This section must always be first
  .text :
  .rodata:
  .data :
  .bss :
  .got :
```

```
 }
}
```

to:

```
# Keep code in flash
SECTIONS {
 GROUP ADDRESS 0xFFFE0000 : {
  .start:        # This section must always be first
  .text :
  .rodata:
}
GROUP ADDRESS 0x10000000 : {
  .data :
  .bss :
  .got :
 }
}
```

13. A jump is made to the label `__kernel_start_st`, which is located in the module 401intr.s.
14. The stack pointer is initialized and allocated, then the *main()* C function (in the BSP) is called. The `STACK_SIZE` definition defines the stack size in bytes.

OS Open and the PCI SDK

The PCI SDK uses some functions and utilities from IBM's OS Open mini operating system. Refer to the OsOpen.txt file (found in the PCI SDK PCI9080RDK-401B BSP directory) for information on the functions and utilities provided by OS Open used by the PCI SDK.

2.1.2.3 Hitachi SH-3 Initialization

The SH-3 is initialized using the following method:

1. Upon reset, the software jumps to address 0xA000 0000 (the physical address is 0x0000 0000, and the EPROM offset is 0x0 0000). The code at this location is the beginning of the .vector section, and is labeled "start".
2. The SH-3's EXPEVT internal register is read. This register contains the type of reset, either manual or power-up reset. If it is not a power-up reset, execution comtinues from step 6.
3. The Bus State Controller is initialized. This allows accesses to DRAM and SRAM.
4. The first 256 kilobytes of the DRAM is tested. In the case of failure, execution stops here.
5. The program sections are relocated. As explained in the RomXxxxx.cmd files, there are 2 ways of doing this:
   - Relocate the code and the initialized data to DRAM. Once this is done, the EPROM is not accessed anymore. This produces faster code.
   - Relocate only the initialized data section. The code will still run from the EPROM. This saves DRAM memory.
6. The stack pointer is initialized.

7. The VBR (Vector Base Register) is initialized. This allows exceptions and interrupts to be serviced.
8. The Status Register (SR) is initialized. The only goal of this step is to turn off the SR.RB.
9. A jump to the main() C routine is performed.
10. From the main() routine located in mainrom.c, the Interrupt Controller (INTC) is initialized. Once IntcInit() returns, all the other interrupt related functions can be called.
11. The Serial Communication Interface with FIFOs (SCIF) is initialized by a call to SerialInit().

For further information on the SH-3 initialization code please refer to the SH3 BSP files. Every effort has been made to ensure that they are clear and consise and they are commented throughout.

2.1.2.4 IDT RC32364 Initialization

All of the IDT RC32364 initialization is done within the IDT boot code library. The initialization sequence is as follows:

1. The microprocessor starts execution at physical address 0xBFC00000. This address contains a jump instruction to the idtstart function.
2. The idtstart function clears the RC32364's status register.
3. The first 32 MB of RAM space, and the Direct Master windows (16 MB each) virtual addresses are assigned to be equal to their physical addresses.
4. The Transaction Lookaside Buffer (TLB) is initialized.
5. RAM accesses are verified by putting different patterns on the local bus.
6. If the RAM is accessible, then the cache is configured and initialized, the uninitialized data section is initialized to zero and the stack is set up.
7. The initialized data section of ROM-version application is copied from the EPROM to RAM memory.
8. The device table is moved to RAM memory from ROM memory. This table contains the device information for the UART and Ethernet device drivers.
9. All devices in the device table are initialized.
10. The available RAM memory size is computed.
11. The breakpoint table is cleared and a jump to the main() 'C' function (in the BSP) is called.

For further information on the IDT RC32364 initialization code please refer to the BSP files. Every effort has been made to ensure that they are clear and consise and they are commented throughout.

2.1.3 Customizing The Board Initialization Function

The Board Initialization function configures the PCI IC and initializes the IOP API for the engineering board. This function should be called after the microprocessor has been initialized. Some of the initialization function calls can be skipped over when a Configuration EEPROM is installed and connected to the PCI IC. The initialization functions can only be called before the Init Done Bit of the PCI IC is set (this causes the negation of the NB# pin). An example of the board initialization function is provided with explanations for each step following it.

The Board Initialization Function:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments. This sample code is taken from MainRom.C file in 9080RDK-401B BSP package.*

```
void
BoardInit9080(void)
{
  RETURN_CODE rc;
  API_PARMS apiInit;             /* API Init structure */
  PCI_BUS_PROP pciBusProp;       /* PCI Bus Properties  */
  IOP_BUS_PROP localBusProp0;    /* IOP Space 0 Properties  */
  IOP_BUS_PROP localBusProp1;    /* IOP Space 1 Properties  */
  IOP_BUS_PROP localBusPropRom;  /* Expansion ROM Properties  */
  IOP_ARBIT_DESC arbDesc;        /* IOP Space Arbitration Properties */
  IOP_ENDIAN_DESC localEndianDesc; /* Endian properties */
  PLX_INTR plxIntr;               /* Interrupt to enable at boot */
  U32 value;


  /*
     The PlxInitApi function will initialize the following structures
     with the default bit field settings corresponding to the PCI 9080
     reset values.
   */

  /* Set apiInit Structure to ZERO */
  for ( value = 0; value < sizeof(API_PARMS); value++)
    ( (PU8) &apiInit ) [value] = 0;

  /* Only initialize the data structure members we need */
  apiInit.PlxIcIopBaseAddr = (void *)EMBED_PLX_MAP_ADDR; ❶
  apiInit.PtrPciBusProp = &pciBusProp;        /* Init Structure */
  apiInit.PtrIopBus0Prop = &localBusProp0;    /* Init Structure */
  apiInit.PtrIopBus1Prop = &localBusProp1;    /* Init Structure */
  apiInit.PtrExpRomBusProp = &localBusPropRom; /* Init Structure */
  apiInit.PtrIopArbitDesc = &arbDesc;         /* Init Structure */
  apiInit.PtrIopEndianDesc = &localEndianDesc; /* Init Structure */

  /* Initialize API and structures */
  rc = PlxInitApi(&apiInit);
  if (rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);

  /*
     WARNING:  All structures used to initialize the 9080 have now been
               initalized with the default bit field settings
               corresponding to the PCI 9080 reset values.  Consult the
               PCI 9080 Data Sheet for a detailed description of the
               reset values.
   */

  /* -------------- init. IOP endian properties -------------------- */
  /*
     The PlxInitIopEndian function MUST be called first so that all
     subsequent accesses to PCI 9080 registers are correct.
```

```
 */

 /* DMA channel 0 runs in big endian mode */
 localEndianDesc.BigEDmaChannel0 = 1; ❷
 /* DMA channel 1 runs in big endian mode */
 localEndianDesc.BigEDmaChannel1 = 1;
 /* PCI 9080 registers are accessed in big endian mode from IOP bus */
 localEndianDesc.BigEIopConfigRegAccess = 1;

 /* Now init the endian properties */
 rc = PlxInitIopEndian(PrimaryPciBus, &localEndianDesc);
 if (rc == ApiBadConfigRegEndianMode)
 {
   /*
      The API has returned that the BigEIopConfigRegAccess bit
      was setup wrong. Just switch it and try again.
    */
   localEndianDesc.BigEIopConfigRegAccess =
               1 - localEndianDesc.BigEIopConfigRegAccess;
   rc = PlxInitIopEndian(PrimaryPciBus, &localEndianDesc);
 }
 if (rc != ApiSuccess)
   PlxSdkError( (U32)rc, TRUE);

 /*
    Read the EEPROM control register.  If there is a configuration
    EEPROM present, then we assume the PCI 9080 device registers are
    being configured using the EEPROM.  If it is not present then
    configure the registers using the IOP code contained in function
    Config9080WoEeprom.

    WARNING: If there is an EEPROM and you are using a short EEPROM
             load, you will have to call Config9080WoEeprom anyway if
             you want to configure Direct Master, Iop Space 0,
             Expansion ROM or Iop Space 1. If there is an EEPROM and
             you are using a long EEPROM load, you will have to call
             part of Config9080WoEeprom if you want to configure Iop
             Space 1.
  */
 value = PlxRegisterRead(PrimaryPciBus, ❸
                         PCI9080_EEPROM_CTRL_STAT,
                         &rc);
 if (rc != ApiSuccess)
   PlxSdkError( (U32)rc, TRUE);

 if ((value & 0x10000000) == 0)
 {
   Config9080WoEeprom(&apiInit);
 }
 else
 {
   Config9080WithEeprom();
 }

 /*
    All PCI 9080 Registers should be initialized before this point.
    Now set the Local Init Done Bit so the computer's BIOS can
```

```
      continue the boot process.
      WARNING: If you do not set this bit the computer will never boot.
   */
   rc = PlxInitDone(PrimaryPciBus); ❹
   if (rc != ApiSuccess)
     PlxSdkError( (U32)rc, TRUE);

/* ---------- Init. PCI9080 interrupt control properties ---------- */

   /* Set plxIntr Structure to ZERO before enabling interrupts */
   for ( value = 0; value < sizeof(PLX_INTR); value++) ❺
     ((PU8)&plxIntr)[value] = 0;

   /*
      WARNING:  DMA interrupts must be enabled for the IOP API DMA
                Resource Manager to function properly.
    */
   plxIntr.IopDmaChannel0 = 1;         /* DMA channel 0 */
   plxIntr.IopDmaChannel1 = 1;         /* DMA channel 1 */
   plxIntr.IopDoorbell = 1;            /* pci to IOP doorbell */
   plxIntr.PciDoorbell = 1;            /* IOP to pci doorbell */
   plxIntr.PciMainInt = 1;             /* PCI interrupt enable */
   plxIntr.IopMainInt = 1;             /* IOP interrupt enable */

   rc = PlxIntrEnable(PrimaryPciBus, &plxIntr);
   if ( rc != ApiSuccess)
     PlxSdkError( (U32) rc, TRUE);

   /* Install and enables the PLX system interrupt handler */
   if (IoSetupPlxSystemInterrupt(Pci9080LserrHandler) != 0) ❻
     PlxSdkError( (U32)BspLastError, TRUE);

   /* Install and enable the PLX interrupt handler */
   if (IoSetupPlxInterrupt(Pci9080LintHandler) != 0)
     PlxSdkError( (U32)BspLastError, TRUE);

   /* turn off the LED */
   rc = PlxUserWrite(PrimaryPciBus,
                     USER0,
                     Inactive);
   if ( rc != ApiSuccess)
     PlxSdkError( (U32) rc, TRUE);

   return;
}
```

❶ Setup the IOP API Initialization function parameters. This function takes as parameters all the structures for all the IOP API PCI IC Initialization functions and sets them to the PCI IC's default values.

❷ Setup the IOP Bus Endianness. This function configures endianness of all PCI IC accesses to the IOP Bus.

❸ Determine if there is an EEPROM connected to the PCI IC. The PCI IC can determine when if there is a Configuration EEPROM connected to it. When none are present the *Config9080WoEeprom()* is called to configure the PCI IC manually, otherwise

Config9080Eeprom() is called. For more information on configuring the PCI IC manually see the Manual PCI IC Configuration Function section which describes the steps needed to configure the PCI IC manually.

❹ Finish the PCI IC initialization. This function sets the Init Done Bit of the PCI IC (this asserts the NB# pin low) allowing the accesses to the PCI Configuration registers from the PCI BIOS. Up until this point the PCI Host system waits until all PCI boards have asserted the NB# bit before it continues booting.

❺ Enable the PCI IC interrupt triggers. This section starts by clearing the PLX_INTR structure. The necessary interrupt triggers are set within the structure and the interrupts are enabled. In this example, the following interrupt triggers are enabled: the DMA channels, the IOP and the PCI doorbells, and the main IOP and PCI interrupt.

Note: *If the DMA Resource Manager will be used in an application, it is important to enable the DMA channel interrupt triggers and that the ISR connected to the PCI IC's interrupt line calls* PlxDmaIsr() *with the appropriate DMA channel number that caused the DMA interrupt.*

❻ Connect the PCI IC ISRs to the appropriate interrupt lines of the microprocessor.

The Manual PCI IC Configuration Function:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments. This sample code is taken from MainRom.C file in 9080RDK-860 BSP package.*

```
void
Config9080WoEeprom(PAPI_PARMS pInitParams)
{
  RETURN_CODE rc;                 /* Return code from API functions */

  /***************************
     Init PCI bus properties    ❶
   ***************************/

  /* assert DMPAF# when more than 0x1c entries are in write FIFO */
  pInitParams->PtrPciBusProp->WFifoAlmostFullFlagCount = 0x1C;

  rc = PlxInitPciBusProperties(PrimaryPciBus,
                               pInitParams->PtrPciBusProp);
  if ( rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);


  /*********************************
     Initialize the IOP arbitration    ❷
   *********************************/

  /* enable latency timer */
  pInitParams->PtrIopArbitDesc->EnableIopBusLatencyTimer = 1;
  /* Disable Local Bus Direct Slave Give Up Bus Mode */
  pInitParams->PtrIopArbitDesc->IopBusDSGiveUpBusMode = 0;
  /*
     number of IOP bus clocks before negating HOLD and
     releasing IOP bus
   */
```

```
pInitParams->PtrIopArbitDesc->IopBusLatencyTimer = 0x0C;

rc = PlxInitIopArbitration(PrimaryPciBus,
                           pInitParams->PtrIopArbitDesc);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);


/*************************
   Initialize IOP Space 0    ❸
 *************************/

/* map Space 0 into memory */
pInitParams->PtrIopBus0Prop->MapInMemorySpace = 1;
/* enable Ready input */
pInitParams->PtrIopBus0Prop->EnableReadyInput = 1;
/* operate in delayed transaction mode */
pInitParams->PtrIopBus0Prop->PciRev2_1Mode = 1;
pInitParams->PtrIopBus0Prop->IopBusWidth = 3;      /* 32 bits bus */
/* BREQo enable so that DS and DM can occur at the same time */
pInitParams->PtrIopBus0Prop->EnableIopBREQo = 1;

rc = PlxInitIopBusProperties(PrimaryPciBus,
                             IopSpace0,
                             pInitParams->PtrIopBus0Prop);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);

/* IOP space 0 address and size */
/* By default set the base address of IOP Space 0 to DRAM */
rc = PlxInitLocalSpace(PrimaryPciBus,
                       IopSpace0,
                       DRAM_MAP_ADDR,     /* DRAM base address */
                       LS0_WINDOW_SIZE    /* size of window */
                       );
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);


/*************************
   Initialize IOP Space 1    ❸
 *************************/
/* WARNING: IOP Space 1 MUST be initialized if I2O is enabled */

/* map Space 1 into memory */
pInitParams->PtrIopBus1Prop->MapInMemorySpace = 1;
/* enable Ready input */
pInitParams->PtrIopBus1Prop->EnableReadyInput = 1;
/* operate in delayed transaction mode */
pInitParams->PtrIopBus1Prop->PciRev2_1Mode = 1;
pInitParams->PtrIopBus1Prop->IopBusWidth = 3;      /* 32 bits bus */
/* BREQo enable so that DS and DM can occur at the same time */
pInitParams->PtrIopBus1Prop->EnableIopBREQo = 1;

rc = PlxInitIopBusProperties(PrimaryPciBus,
                             IopSpace1,
                             pInitParams->PtrIopBus1Prop);
```

```
  if (rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);

  /* IOP space 1 address and size */
  rc = PlxInitLocalSpace(PrimaryPciBus,
                         IopSpace1,
                         DRAM_MAP_ADDR,     /* DRAM base address */
                           LS1_WINDOW_SIZE    /* size of window */
                           );
  if (rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);


  /*****************************************
     Initialize Direct Master Memory and IO    ❹
   *****************************************/

  rc = PlxInitPciSpace(PrimaryPciBus,
                       PciMemSpace,
                       DM_MEM_MAP_ADDR, /* IOP base address for DM
                                           memory */
                       DM_SIZE);        /* size of window */
  if ( rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);

  /* one more time for IO space */
  rc = PlxInitPciSpace(PrimaryPciBus,
                       PciIoSpace,
                       DM_IO_MAP_ADDR, /* IOP base address for DM
                                          IO/CFG */
                          DM_SIZE);    /* size of window */

  if ( rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);

  /*
     Note: Leave all other PCI configuration registers alone so that
           they are set either by default values or by the system BIOS.
   */

  /*******************************
     Miscellaneous Initialization    ❺
   *******************************/

  rc = PlxRegisterWrite(PrimaryPciBus,
                        PCI9080_VENDOR_ID,
                        PLX_VENDOR_ID|(PLX_9080RDK_860_DEVICE_ID<<16));
  if ( rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);

  /* Expansion ROM Range Register. No Expansion ROM for PCIRDK-860 */
  rc = PlxRegisterWrite(PrimaryPciBus,
                        PCI9080_EXP_ROM_RANGE,
                        0x00000000);     /* no expansion ROM */
  if ( rc != ApiSuccess)
    PlxSdkError( (U32) rc, TRUE);
}
```

❶ Initialize the access to the PCI Bus. This function takes the PCI_BUS_PROP structure and configures the PCI IC accordingly for PCI Bus accesses. In the example the number of entries before asserting DMPAF# was modified from the default values.

❷ Initialize the IOP Bus Arbitrator. This function takes the IOP_ARBIT_DESC structure and configures the PCI IC IOP Bus Arbitrator according to the values defined within the structure. In the example, the IOP Bus Latency Timer was modified from the default values.

❸ Initialize the IOP Space Accesses. This function configures the PCI access to the IOP Bus through the Local Space Registers (BAR 2-5 of the PCI Configuration Registers). In this example only IOP Space 0 and 1 are used. Both IOP Spaces are configured so that they are mapped into PCI memory space, with Ready Input and BREQo enabled, with a 32-bit bus width and that PCI IC works in Delayed Transaction mode for IOP reads.

❹ Initialize IOP accesses to the PCI bus. This function configures accesses to the PCI Bus from the IOP bus such as the PCI base addresses and their sizes. In the example, PCI Memory and IO/CFG accesses are setup.

❺ Initialize some miscellaneous registers. This section initializes some PCI Configuration registers that are not covered by the other functions. The first register function configures the vendor and device IDs. The second configures the IOP Base address for PCI access to the Local Expansion ROM register.

2.1.4 Initialization Of PCI Devices for PCI 9080RDK-RC32364

The PCI 9080RDK-RC32364 provides a PCI bus which can support up to 3 other PCI devices. For this reason the PCI SDK contains a mini PCI BIOS to minimally support the PCI devices that are inserted into the PCI slots. This PCI BIOS allows the configuration of all PCI devices on the PCI bus and allows data transfers between the PCI 9080RDK-RC32364 and any PCI device on the bus. The following section describes the operation of the mini PCI BIOS supplied in the PCI SDK.

The *SetupPciDevices()* Function:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments.*

```
U32
SetupPciDevices(void)
{
  U32 busNum, deviceNum, barNum;      /* loop counters */
  U32 tempData;                       /* Temporary data holder */
  U32 tempRange;                      /* Temporary Range holder */
  U32 tempSize;                       /* Temporary Size holder */
  U32 mappedAddress;
  /* IDT board PCI Stat */
  PU8 pciPresentStatAddr=(PU8)PCI_PRESENT_STAT_ADDR;
  U8 pciPresentStat;                  /* IDT board PCI Stat value */
  U32 sleepWait;                      /* sleep counter */
  RETURN_CODE rc;

/* --------------------- Configure itself ------------------------ */ ❶
  if ( PlxRegisterWrite(PrimaryPciBus, PCI9080_COMMAND,
```

```
          PCIBIOS_CFG_COMMAND_VALUE) != ApiSuccess )
    return 1;

  if ( PlxRegisterWrite(PrimaryPciBus, PCI9080_REV_ID,
          PCIBIOS_CFG_REV_ID_VALUE) != ApiSuccess )
    return 1;

  if ( PlxRegisterWrite(PrimaryPciBus, PCI9080_CACHE_SIZE,
          PCIBIOS_CFG_CACHE_SIZE_VALUE) != ApiSuccess )
    return 1;


/* ---------------------- Configure BARs ------------------------- */ ❷
  /* Assign addresses to BARs */
  for ( barNum = CFG_BAR0;
        barNum <= CFG_BAR5;
        barNum += 4 )
  {
    /* Write all 1 to the register */
    if ( PlxRegisterWrite(PrimaryPciBus, barNum, 0xFFFFFFFF) ❸
        != ApiSuccess )
      return 1;

    /* Then, read it back */
    tempRange = PlxRegisterRead(PrimaryPciBus, barNum, &rc);

    if (rc != ApiSuccess)
      return 1;

    if ( (tempRange == 0) ||
         (tempRange == 1) ||
         (tempRange == 0xFFFFFFFF) )
      break; /* No more BARs to configure */

    if ( tempRange & 0x1 )
    {
      /* IO-mapped */ ❹

      /* Convert range to a size (and force it to minimum size) */
      tempSize = ((~(tempRange - 1) + 1) + (PCIBIOS_IO_BLOCK - 1))
                  & (~(PCIBIOS_IO_BLOCK - 1));

      /* Bring mgIoBase to the next available window base on
         the requested size for this BAR */
      if ((mgIoBase & tempRange) != mgIoBase)
        mgIoBase = (mgIoBase | (tempSize - 1)) + 1;

      mappedAddress = mgIoBase | 0x1;

      /* Increment mgIoBase for next assignment */
      mgIoBase += tempSize;
    }
    else
    {
      /* Memory-mapped */ ❺
```

```
      /* Convert range to a size (and force it to minimum size) */
      tempSize = ((~(tempRange) + 1) + (PCIBIOS_MEM_BLOCK - 1))
                 & (~(PCIBIOS_MEM_BLOCK - 1));

      /* Bring mgMemBase to the next available window base on
         the requested size for this BAR */
      if ((mgMemBase & tempRange) != mgMemBase)
        mgMemBase = (mgMemBase | (tempSize - 1)) + 1;

      mappedAddress = mgMemBase;

      /* Increment mgMemBase for next assignment */
      mgMemBase += tempSize;
    }
    /* Write the address to the BAR */ ❻
    if ( PlxRegisterWrite(PrimaryPciBus, barNum, mappedAddress)
         != ApiSuccess )
      return 1;
  } /* End of For loop */

/* ---------------------- Configure Interrupt line ----------------- */
  /* Request an interrupt line */ ❼
  tempData = PlxRegisterRead(PrimaryPciBus, PCI9080_INT_LINE, &rc );
  if ( rc != ApiSuccess )
    return 1;

  if ( tempData & 0x0000FF00 )
  {
    tempData &= 0xFFFFFF00;
    tempData |= (mgInterruptBase & 0xFF );
    if ( PlxRegisterWrite(PrimaryPciBus, PCI9080_INT_LINE, tempData)
         != ApiSuccess )
      return 1;
    mgInterruptBase++;
  }

/* -------------------- Configure other PCI devices -------------- */

  /* Sleep so that the PCI devices have enough time to self-configure
     before probing their PCI configuration registers */
  PlxPrintf("Sleeping...");
  for(sleepWait = 0; sleepWait < 0x140000; sleepWait++); ❽
  PlxPrintf("Awoken!\n");

  /* Read the PCI Present Status register */
  pciPresentStat = *pciPresentStatAddr;

  for(busNum = 0; busNum < PCIBIOS_MAX_BUS; busNum++)
  {
    for(deviceNum = 0; deviceNum < PCIBIOS_MAX_DEVICE; deviceNum++)
    {
      /* determine if a PCI device is present */
      if((pciPresentStat & (0x3<<(deviceNum*2))) ❾
          != (0x3<<(deviceNum*2)))
      {
        PlxPrintf("Config Bus #%d, Slot #%d\n", busNum, deviceNum + 1);
```

```
        /* configure device */
        if(ConfigPciDevice(busNum, deviceNum + 1)) ❿
        {
           PlxPrintf("Device configuration error, exiting...");
           return 1;
        }
      }
    }   /* END deviceNum loop */
  }   /* END busNum loop */

  return 0;
}
```

❶ Configure the PCI 9080RDK-RC32364's PCI IC. This section configures the PCI Command Register, the PCI Class Code Register, the PCI Cache Line Size Register and the PCI Latency Timer Register.

Note: *The values for these registers were determined by placing a PCI board into a PCI bus on a computer with a valid PCI BIOS and may not reflect their proper values as defined by the PCI specification.*

❷ Assign PCI address space for each of the Base Address Registers (BARs).

❸ The requirements of each BAR is determined by writing 0xFFFFFFFF to the BAR and reading the value returned by the PCI IC. If the register returns 0xFFFFFFFF then the BAR and all subsequent ones are not in use. The least significant bit of the returned value determines whether to assign an address range within the PCI I/O space or the PCI Memory space.

❹ Assign an PCI I/O space address. From the value returned from the BAR the range is determined and space within the PCI I/O address range is allocated.

❺ Assign an PCI Memory space address. From the value returned from the BAR the range is determined and space within the PCI Memory address range is allocated.

❻ Write the allocated address to the BAR. Once an address range is allocated the base of the space (being either in PCI Memory or I/O space) is written to the BAR.

❼ Configure and assign an interrupt line to the PCI device (as required).

❽ Most PCI devices need to do some configuration to devices on the board before the PCI BIOS accesses it. Normally the PCI BIOS will receive RETRYs until the PCI device is ready for accesses. To get around this the mini PCI BIOS waits for a period of time to allow configuration time for all PCI devices on the PCI bus.

❾ Determine if a PCI slot contains a PCI device. The PCI 9080RDK-RC32364 contains a 8 bit register that states which PCI slots are occupied by a PCI device. By reading this register the presence of a PCI device is quickly determined. This method is not recommended for a fully compliant PCI BIOS. For fully compliant PCI BIOSs it is recommended to access the PCI bus directly to determine the presence of PCI devices by testing each valid PCI slot on each valid PCI bus. Since the PCI 9080RDK-RC32364 has only one PCI bus with three PCI slots, these values are hardcoded into the max values, `PCIBIOS_MAX_BUS` and `PCIBIOS_MAX_DEVICE`, respectively.

❿ Configure a PCI device present on the PCI bus. This function will be explained in more detail in the following section.

The *ConfigPciDevice()* Function:

Note: *The following code example is provided for explanation purposes only. Please refer to the source code files provided in the PCI SDK for the valid, working code and supplemental comments.*

```
U32
ConfigPciDevice(U32 busNum,    /* Bus Number of the device */
    U32 deviceNum)              /* Slot Number of the device */
{
  U32 barNum;         /* loop counters */
  U32 tempData;       /* Temporary Data holder */
  U32 tempRange;      /* Temporary Range holder */
  U32 tempSize;       /* Temporary Size holder */
  U32 mappedAddress;
  /* IDT board PCI Stat */
  PU32 pciPresentStatAddr=(PU32)PCI_PRESENT_STAT_ADDR;
  RETURN_CODE rc;

  /* Configure the device */ ❶
  tempData = PCIBIOS_CFG_COMMAND_VALUE;
  if( (rc = PlxPciConfigRegisterWrite(busNum, deviceNum, CFG_COMMAND,
           &tempData)) != ApiSuccess )
  {
    PlxSdkError(rc, FALSE);
    return 1;
  }

  tempData = PCIBIOS_CFG_CACHE_SIZE_VALUE;
  if((rc = PlxPciConfigRegisterWrite(busNum, deviceNum, CFG_CACHE_SIZE,
          &tempData)) != ApiSuccess )
  {
    PlxSdkError(rc, FALSE);
    return 1;
  }

  /* Assign addresses to BARs */ ❷
  for ( barNum = CFG_BAR0;
        barNum <= CFG_BAR3;
        barNum += 4 )
  {
    tempRange = GetBarRange(busNum, deviceNum, barNum); ❸
    if ( (tempRange != 0) &&
         (tempRange != 1) &&
         (tempRange != 0xFFFFFFFF) )
    {
      if ( tempRange & 0x1 )
      {
        /* IO-mapped */ ❹

        /* Convert range to a size (and force it to minimum size) */
        tempSize = ((~(tempRange - 1) + 1) + (PCIBIOS_IO_BLOCK - 1))
                     & (~(PCIBIOS_IO_BLOCK - 1));

        /* Bring mgIoBase to the next available window base on
```

```
        the requested size for this BAR */
      if ((mgIoBase & tempRange) != mgIoBase)
        mgIoBase = (mgIoBase | (tempSize - 1)) + 1;

      mappedAddress = mgIoBase | 0x1;

      /* Increment mgIoBase for next assignment */
      mgIoBase += tempSize;
    }
    else
    {
      /* Memory-mapped */ ❺

      /* Convert range to a size (and force it to minimum size) */
      tempSize = ((~(tempRange) + 1) + (PCIBIOS_MEM_BLOCK - 1))
                  & (~(PCIBIOS_MEM_BLOCK - 1));

      /* Bring mgMemBase to the next available window base on
         the requested size for this BAR */
      if ((mgMemBase & tempRange) != mgMemBase)
        mgMemBase = (mgMemBase | (tempSize - 1)) + 1;

      mappedAddress = mgMemBase;

      /* Increment mgMemBase for next assignment */
      mgMemBase += tempSize;
    }
    /* Write to the BAR */
    if ( (rc = PlxPciConfigRegisterWrite(busNum, deviceNum,
                barNum, & mappedAddress )) != ApiSuccess ) ❻
    {
      PlxSdkError(rc, FALSE);
      return 1;
    }
  }
  else
    break; /* No more BARs to configure */ ❼
}   /* END barNum loop */

/* Request an interrupt line */ ❽
if ( (rc = PlxPciConfigRegisterRead(busNum, deviceNum,
      CFG_INT_LINE, &tempData )) != ApiSuccess )
{
  PlxSdkError(rc, FALSE);
  return 1;
}
if ( tempData & 0x0000FF00 )
{
  tempData &= 0xFFFFFF00;
  tempData |= (mgInterruptBase & 0xFF );
  if ( PlxPciConfigRegisterWrite(busNum, deviceNum,
                  CFG_INT_LINE, &tempData ) != ApiSuccess )
  {
    PlxSdkError(rc, FALSE);
    return 1;
  }
```

```
    mgInterruptBase++;
  }
  return (0);
}
```

❶ Configure the PCI 9080RDK-RC32364's PCI IC. This section configures the PCI Command Register and the PCI Cache Line Size Register.

Note: *The values for these registers were determined by placing a PCI board into a PCI bus on a computer with a valid PCI BIOS and may not reflect their proper values as defined by the PCI specification.*

❷ Assign PCI address space for each of the Base Address Registers (BARs).

❸ Determine the range required for the BAR. The *GetBarRange()* function determines the required range for each BAR by writing `0xFFFFFFFF` to the register and reading back the value. The return value states which address lines are used in the decoding of the PCI address and from this a range value is calculated.

❹ Assign a PCI I/O address to the BAR. The least significant bit of the range value returned from the BAR determines whether to assign an address space from the PCI Memory or PCI I/O address space. In this case assign an address range from the PCI I/O address space.

❺ Assign a PCI Memory address to the BAR.

❻ Write the assigned address to the BAR. The address can be either a PCI Memory or PCI I/O address.

❼ If the range value is `0xFFFFFFFF` then there are no more BARs to configure.

❽ Determine the interrupt requirements of the device. If the device requests an interrupt line, assign one that is not current assigned.

2.2 IOP Applications

This section discusses how to build IOP applications with the IOP API. The function of the Hello World application is to blink the LED connected to the USERo pin of the PCI IC and to print a formatted string to the serial port. The application has two different executable files, one that runs in IOP RAM memory and one that run in IOP ROM memory. The IOP RAM application runs by downloading it to IOP RAM memory; the IOP ROM application runs by programming it to FLASH and rebooting the engineering board. The PCI SDK libraries provide the functionality and support for the Hello World application.

Note*: All PCI SDK examples have been designed to run on PLX engineering boards and refer to blinking an LED. All PLX engineering boards contain an LED connected to the USERo pin of the PCI device. If the PCI SDK examples are used on a customer board without an LED, the USERo pin will toggle from low to high.*

2.2.1 Setup Of The Development Environment

The development environment for the Hello World application and all other applications can be easily setup and configured. Two files are provided to setup the environment:

- The environment setup batch file: this file sets some environment variables used by the PCI SDK for compiling and linking IOP applications; and,

- The make file: this file easily compiles and links source code to create the executables for the IOP application.

The Environment Setup Batch File

The environment setup batch file programs some environment variables that are needed by the microprocessor's development tools and the IOP application make files. Some of the variables that are set are:

- Update the PATH variable to point to the appropriate development tools;
- Set the path for the compiler's include and library directories; and,
- Set some variables required by the development tools.

The Make File

The make file is an easy way to compile and link IOP applications. It contains all the necessary information regarding the compiler and linker needed to rebuild the IOP applications. This file has variables to hold compiler and linker parameters.

The make files included in the PCI SDK hold most of the information needed to rebuild their respective IOP applications. Some of the information not included in the make file is the directory paths for the microprocessor's development tools (this information is taken from the environment variables that were set by calling the environment setup batch file explained earlier) and some command line parameters (this information controls how the make file is executed).

2.2.2 Rebuilding The Hello World Application

To rebuild the Hello World application execute the make file contained within the Hello World directory. To do this, at a command prompt, run the environment setup batch file for the desired microprocessor development environment. Change to the Hello World sample directory and type the following to build the Hello World IOP RAM application for the PCI 9080RDK-401B:

```
nmake /f 80-401B.mak
```

To rebuild the Hello World IOP ROM application, type instead:

```
nmake /f 80-401B.mak ROM="TRUE"
```

The make file builds the application properly by using the environment variables set by the batch file and by any parameters passed in from the command prompt.

In the Hello World directory there is only one source file, being the Hello World application's main file and a make file for each supported microprocessor. When rebuilding the IOP application, the make file links in the appropriate libraries from the PCI SDK library directory for all the support functions needed by the IOP application.

2.2.3 Building Custom Applications

Custom applications built with the PCI SDK use the precompiled IOP libraries. These libraries provide the necessary support functions for the IOP application. To build a custom IOP application it is recommended to start with the Hello World application. This IOP application is a basic application that can be easily ported to new engineering boards once the BSP module has

been setup. From there, the custom IOP application can be built by replacing the Hello World main task function within the BSP's *main()* function. The environment setup batch file and the make file of the Hello World application can be recycled for the custom application and updated accordingly. There is one thing to keep in mind when porting the make file. The libraries included into the PCI SDK are built with the following compiler definitions:

- `DEBUG`: code generated validates all arguments, not compiled for speed.
- `IOP_CODE`: code generated is destined for the IOP; and,
- `BIG_ENDIAN`: code generated is used with a big endian microprocessor.

Depending on the final destination of the code (either RAM or ROM) one of the following definitions is used:

- `IOP_RAM`: code generated is destined for RAM; or,
- `IOP_ROM`: code generated is destined for ROM;

for the PCI IC used:

- `PCI9080`: the code is generated for the PCI 9080 IC; or,
- `PCI9054`: the code is generated for the PCI 9054 IC,

and for the microprocessor (either IBM 401 or MPC860):

- `MPC860`: code generated is run on the MPC860 microprocessor; or,
- `IBM401B`: code generated is run on the IBM 401 microprocessor.

The make file includes these compiler definitions already. If the custom IOP application is built without using the supplied make file these definitions must be included when compiling and linking the application to the PCI SDK libraries.

2.3 Windows Applications

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

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

2.3.1 Creating A MS Developer's Studio Project File

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

1. On the File menu tab, choose New… item.
2. Create a new project file by choosing the Projects tab.
3. Choose a Win32 Application project file. Ensure that the Create New Workspace button is set.
4. Choose a project name and destination location for the project.

5. Setup the project environment.
   - Under the Project menu of Developer Studio, choose Settings... This brings up the project settings dialog box.
   - Choose the C/C++ tab.
   - Change the Category type to Preprocessor.
   - In the Preprocessor definitions window add the following entries:
     - `PCI_CODE`
     - `LITTLE_ENDIAN`
   - In the Additional include directories add a path to the PLX SDK include directory. ***Note:*** *It is recommended that a relative path from the workspace project directory to the PCI SDK include directory be given if both the include directory and the workspace project are located on the same logical Windows Disk Drive. This makes the workspace project more compatible between various Windows Systems and Windows Disk Drive Mappings.*
6. Create the application that will use the PCI API.
7. Include all the source files into the project. Copy the `PlxApi.lib` file in the PCI SDK directory into the workspace project directory. Include the `PlxApi.lib` file into the workspace project files.
8. Build the application.

2.3.2 Building A Custom Application

To start building a custom application with the PCI API create a new project for the application (see section 2.3.1 for more information). To start accessing a PCI engineering board using the PCI API follow these steps:

1. Include the following headers:
   - **PciTypes.h:** contains all the data types, enumerated types and constants used by the PCI API (for Win32 applications only);
   - **PlxTypes.h:** contains all the data types, enumerated types and constants used by both APIs (common for IOP and Win32 applications);
   - **Plx.h:** contains some constants that are common for all development platforms;
   - **PciApi.h:** contains the PCI API function prototypes; and,
   - **PlxError.h:** contains the error codes used by the PCI API.
2. Connect to an engineering board (get a device driver handle). Handle to PLX engineering boards are always obtained via the *PlxPciDeviceOpen()* function. Sometimes, the *PlxPciDeviceFind()* can help to find device driver handle for a desired engineering board. There are several methods for selecting a PCI engineering board (using the device location structure):
   - **Method 1:** Using the device's serial number. The serial number is always in this format:
     `<device driver name>-<index number>`
     where `<device driver name>` is the name of the respective device driver for the

engineering board and `<index number>` is its unique number based on the order in which the device driver found the engineering board . For example, ‘pci9080-1’.
Then a call to *PlxPciDeviceOpen()* can be performed, using the serial number as the `SerialNumber` member of the device location argument.

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

- **Method 3:** When more than one engineering board may match the search criteria, *PlxPciDeviceFind()* may be useful. Here is a procedure capable of listing all the matching engineering boards:
1. Call *PlxPciDeviceFind()* with the device location structure set with the search criteria (with the `SerialNumber` structure set to a NULL string) and the second argument set to FIND_AMOUNT_MATCHED. This will give you the number of matching engineering boards.
2. Call *PlxPciDeviceFind()* for each engineering board found. The device location structure returned will give you completed information about the engineering board.
3. Call *PlxPciDeviceOpen()* with the desired device location as argument.
4. Make connections to other engineering boards if necessary. Repeat Step 2.
5. Start using the PCI API function calls with the appropriate device driver handles.
6. When the application completes close connections to all engineering boards in use with *PlxPciDeviceClose()*.

3. IOP API Function Description

The IOP API is designed around the features of the PCI IC.

3.1 IOP API Function Quick Reference List

The following table lists all the IOP API functions available. Designers should consult section 3.2 for detailed descriptions of the API functions.

API Function Name	Purpose
PlxBusPciRead()	Read from the PCI bus.
PlxBusPciWrite()	Write to the PCI bus.
PlxDmaBlockChannelClose()	Close a Block DMA channel.
PlxDmaBlockChannelOpen()	Open a DMA channel for Block DMA.
PlxDmaBlockTransfer()	Control a Block DMA transfer.
PlxDmaBlockTransferRestart()	Restart the previous Block DMA transfer.
PlxDmaIsr()	Service a DMA interrupt.
PlxDmaResourceManagerInit()	Initialize the DMA Resource Manager.
PlxDmaSglBuild()	Build a SGL.
PlxDmaSglChannelClose()	Close a SGL DMA channel.
PlxDmaSglChannelOpen()	Open a DMA channel for SGL DMA.
PlxDmaSglFill()	Fill a SGL element.
PlxDmaSglTransfer()	Control a SGL DMA transfer.
PlxDmaShuttleChannelClose()	Close a Shuttle DMA channel.
PlxDmaShuttleChannelOpen()	Open a DMA channel for Shuttle DMA.
PlxDmaShuttleTransfer()	Control a Shuttle DMA transfer.
PlxDmaShuttleTransferRestart()	Restart the previous Shuttle DMA transfer element.
PlxEchoEnable()	Enable or disable the character echoing by the UART Services Module.
PlxGetChars()	Get stream of data from the serial port.
PlxHotSwapDisable	Disable PCI access to Hot Swap.
PlxHotSwapEnable	Enable PCI access to Hot Swap.
PlxHotSwapId	Get the Hot Swap capability ID
PlxHotSwapNextItemPointer	Return the Hot Swap next item pointer register.
PlxHotSwapStatus	Get the status of Hot Swap.
PlxInitApi()	Initialize the IOP API.

API Function Name	Purpose
PlxInitDone()	Set the Init Done bit of the PCI IC.
PlxInitIopArbitration()	Initialize the IOP Bus Arbiter.
PlxInitIopBusProperties()	Initialize the IOP Bus properties.
PlxInitIopEndian()	Initialize the endianness of the IOP Bus
PlxInitLocalSpace()	Initialize and enable accesses to the IOP Bus.
PlxInitPciArbitration()	Initialize the PCI Bus arbiter.
PlxInitPciBusProperties()	Initialize the PCI Bus properties.
PlxInitPciSpace()	Initialize and enable accesses to the PCI Bus.
PlxInitPowerManagement()	Initialize the power management properties.
PlxInitVpdAddress()	Initialize the Vital Product Data registers.
PlxIntrDisable()	Disable PCI IC interrupt triggers.
PlxIntrEnable()	Enable PCI IC interrupt triggers.
PlxIntrStatusGet()	Get the current interrupt status.
PlxMuInboundPortRead()	Read from the Inbound Post Tail Pointer.
PlxMuInboundPortWrite()	Write to the Inbound Free Head Pointer.
PlxMuInit()	Initialize the Messaging Unit.
PlxMuOutboundPortRead()	Read from the Outbound Free Tail Pointer.
PlxMuOutboundPortWrite()	Write to the Outbound Post Head Pointer.
PlxPciConfigRegisterRead()	Read from a PCI Configuration register.
PlxPciConfigRegisterWrite()	Write to a PCI Configuration register.
PlxPowerLevelGet()	Get the power level.
PlxPowerLevelSet()	Set the power level.
PlxPrintf()	Write a formatted string of characters to the serial port.
PlxRegisterDoorbellRead()	Read from then clear a Doorbell register.
PlxRegisterDoorbellSet()	Write to a Doorbell register.
PlxRegisterMailboxRead()	Read from a Mailbox register.
PlxRegisterMailboxWrite()	Write to a Mailbox register.
PlxRegisterRead()	Read from a register.
PlxRegisterReadAll()	Read a set of sequential registers.
PlxRegisterWrite()	Write to a register.
PlxSdkVersion()	Get the version and date of the PCI SDK.
PlxSerialEepromRead()	Read from the Serial EEPROM.

API Function Name	Purpose
PlxSerialEepromWrite()	Write to the Serial EEPROM.
PlxUserRead()	Read from a USER pin.
PlxUserWrite()	Write to a USER pin.
PlxVpdDisable	Disable PCI access to the Vital Product Data.
PlxVpdEnable	Enable PCI access to the Vital Product Data.
PlxVpdId	Get the Vital Product Data capability ID.
PlxVpdNextItemPointer	Return the VPD next item pointer register.

3.2 IOP API Functions Details

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

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

Note: *Devices supported by PCI SDK Version 2.1: PCI 9080 and PCI 9054.*

Sample Function Entry

Syntax:

`function(modifier` *`parameter`*`[,...]);`
This gives the declaration syntax for each function. Each parameter is *italicized*.

PCI IC Support:

A list of PCI ICs that support this function.

Description:

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

Return Value:

The value returned by the function. Some return values are only given when an API is compiled with the DEBUG label set. These return codes are shown as shaded.

Usage:

A sample of code is provided to demonstrate the function's use.

PlxBusPciRead

Syntax:

```
RETURN_CODE PlxBusPciRead( IN BUS_INDEX busIndex,
                           IN PCI_SPACE pciSpace,
                           IN U64 address,
                           OUT PU64 destination,
                           IN U32 transferSize,
                           IN ACCESS_TYPE accessType);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads values into a buffer from the PCI bus.

- *busIndex* is the bus index;
- *pciSpace* states which type of bus cycles to use, being either memory bus cycles or I/O bus cycles;
- *address* is the starting PCI address for the read;
- *destination* is a pointer to the buffer to store the data retrieved;
- *transferSize* defines the number of bytes you want to read from the PCI bus; and,
- *accessType* defines the access type size.

Note: *This API function remaps the Direct Master Space window (register value is restored when the function terminates).* destination *must be large enough to store all the data read.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidPciSpace`	The *pciSpace* parameter is neither PciMemSpace nor `PciIoSpace`.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this IOP API.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *address* parameter is not aligned based on the *accessType* provided.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *transferSize* parameter is 0 or is not aligned based on the *accessType*	PCI 9080, PCI 9054

Return Value	Description	API Support
	provided.	

Usage:

```
#define DRAM_MAP_ADDR         0x00000000

RETURN_CODE rc; /* Return code from API functions */
U32 length = 0x100;
U32 bufFrom = DRAM_MAP_ADDR, bufTo = DRAM_MAP_ADDR + length;
U64 pciStartOffset = GetPciAddress();


/* Start copying data at address DRAM_MAP_ADDR */
rc = PlxBusPciWrite(PrimaryPciBus,
                    PciMemSpace,
                    pciStartOffset,
                    (PU64)(bufFrom),
                    length,
                    BitSize8);

if (rc != ApiSuccess)
{
  PlxPrintf("Error: Unable to write data.\n");
  return -1;
}

/* Now read back the data to compare */
rc = PlxBusPciRead(PrimaryPciBus,
                   PciMemSpace,
                   pciStartOffset,
                   (PU64)(bufTo),
                   length,
                   BitSize8);

if (rc != ApiSuccess)
{
  PlxPrintf("Error: Unable to read data.\n");
  return -1;
}

/* Now compare data */
if (memcmp((const void *)bufFrom,(const void*)bufTo, length) != 0)
{
  PlxPrintf("Error: Data is corrupted.\n");
  return -1;
}
```

PlxBusPciWrite

Syntax:

```
RETURN_CODE PlxBusPciWrite( IN BUS_INDEX busIndex,
                            IN PCI_SPACE pciSpace,
                            IN U64 address,
                            IN PU64 source,
                            IN U32 transferSize,
                            IN ACCESS_TYPE accessType);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes values from the buffer provided to the PCI bus.

- *busIndex* is the bus index;
- *pciSpace* states which type of bus cycles to use, being either memory bus cycles or I/O bus cycles;
- *address* is the starting PCI address for the write;
- *source* is a pointer to the buffer containing the data; and,
- *transferSize* defines the number of bytes you want to write to the PCI bus; and,
- *accessType* defines the access type size.

Note: *This API function remaps the Direct Master Space window (register value is returned when the function terminates).*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidPciSpace`	The *pciSpace* parameter is neither PciMemSpace nor `PciIoSpace`.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this IOP API.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *address* parameter is not aligned based on the *accessType* provided.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *transferSize* parameter is 0 or is not aligned based on the *accessType*	PCI 9080, PCI 9054

Return Value	Description	API Support
	provided.	

Usage:

```
#define DRAM_MAP_ADDR         0x00000000

RETURN_CODE rc; /* Return code from API functions */
U32 length = 0x100;
U32 bufFrom = DRAM_MAP_ADDR, bufTo = DRAM_MAP_ADDR + length;
U64 pciStartOffset = GetPciAddress();


/* Start copying data at address DRAM_MAP_ADDR */
rc = PlxBusPciWrite(PrimaryPciBus,
                    PciMemSpace,
                    pciStartOffset,
                    (PU64)(bufFrom),
                    length,
                    BitSize8);

if (rc != ApiSuccess)
{
  PlxPrintf("Error: Unable to write data.\n");
  return -1;
}

/* Now read back the data to compare */
rc = PlxBusPciRead(PrimaryPciBus,
                   PciMemSpace,
                   pciStartOffset,
                   (PU64)(bufTo),
                   length,
                   BitSize8);

if (rc != ApiSuccess)
{
  PlxPrintf("Error: Unable to read data.\n");
  return -1;
}

/* Now compare data */
if (memcmp((const void *)bufFrom,(const void*)bufTo, length) != 0)
{
  PlxPrintf("Error: Data is corrupted.\n");
  return -1;
}
```

PlxDmaBlockChannelClose

Syntax:

`RETURN_CODE PlxDmaBlockChannelClose( IN DMA_CHANNEL dmaChannel);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes the Block DMA channel.

- *dmaChannel* is the DMA channel number previously opened.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;

rc = PlxDmaBlockChannelClose(PrimaryPciChannel0);
if (rc != ApiSuccess)
{
  PlxPrintf("Error closing DMA channel.\n");
  return -1;
}
```

PlxDmaBlockChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Opens and initializes a DMA channel for Block DMA transfers.

- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Note: *Before calling this function a call to PlxDmaResourceManagerInit() must be done to initialize the DMA Resource Manager.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaChannelDesc* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
DMA_CHANNEL_DESC desc;

/* Setup DMA configuration structure */
desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
desc.DmaStopTransferMode = AssertBLAST;
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
desc.HoldIopDestAddrConst = 0;
```

```
desc.DemandMode = 0;
desc.EnableTransferCountClear = 0;
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

rc = PlxDmaBlockChannelOpen(PrimaryPciChannel0, &desc);
if (rc != ApiSuccess)
{
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}
```

PlxDmaBlockTransfer

Syntax:

```
RETURN_CODE PlxDmaBlockTransfer( IN DMA_CHANNEL dmaChannel,
                                 IN DMA_COMMAND dmaCommand,
                                 IN PDMA_TRANSFER_ELEMENT dmaData,
                                 IN BOOLEAN returnImmediate);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Controls the Block DMA transfer for a given DMA channel.

- *dmaChannel* is the DMA channel number previously opened;
- *dmaCommand* is the action to perform on this DMA channel;
- *dmaData* is the data for the DMA transfer; and,
- *returnImmediate* determines if this function waits for the DMA command to complete before returning.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaData* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested	PCI 9080, PCI 9054

Return Value	Description	API Support
	and the DMA channel is not paused).	
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054

Comments:

The following table lists the possible actions used to control a SGL DMA transfer:

Action (*dmaCommand* parameter)	Description
DmaStart	Program and start a DMA channel for a simple DMA transfer.
DmaPause	Pause the DMA channel. The *dmaData* and *returnImmediate* parameters are ignored.
DmaResume	Resume a paused DMA channel. The *dmaData* and *returnImmediate* parameters are ignored.
DmaAbort	Abort the transfer currently in progress for the DMA channel. The *dmaData* and *returnImmadiate* parameters are ignored.
DmaStatus	Gives a status of the DMA channel. The *dmaData* and *returnImmediate* parameters are ignored.

The `DMA_TRANSFER_ELEMENT` structure contains members whose signification may differ depending on the DMA transfer type including ignoring some of the members. Here is the structure:

Structure Element	Signification
LowPciAddr	The low 32 bits of the PCI address for the transfer.
HighPciAddr	The high 32 bits of the PCI address for the transfer. This field is ignored for the PCI 9080.
IopAddr	The IOP address for the transfer.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Indicates if an interrupt is to be generated when the DMA transfer is done.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

```
#define DRAM_MAP_ADDR         0x00000000

RETURN_CODE rc;
DMA_TRANSFER_ELEMENT dmaData;
U32 totalSize = 0x100;
U32 bufFrom = DRAM_MAP_ADDR;


/* Setting dmaData */
dmaData.Pci9080Dma.LowPciAddr        = 0x14BC0000;
```

```
dmaData.Pci9080Dma.IopAddr           = bufFrom;
dmaData.Pci9080Dma.TransferCount     = totalSize;
dmaData.Pci9080Dma.IopToPciDma       = 1;
dmaData.Pci9080Dma.TerminalCountIntr = 0;/* no interrupts */

/*
   Start copying data from address bufFrom to pciStartOffset.
   The call returns immediately.
 */
rc = PlxDmaBlockTransfer(PrimaryPciChannel0,
                         DmaStart,
                         &dmaData,
                         TRUE);
if (rc != ApiSuccess)
{
  PlxPrintf("Error while performing a Block DMA with channel 0.\n");
  return -1;
}
```

PlxDmaBlockTransferRestart

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 totalSize = 0x100;


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

PlxDmaIsr

Syntax:

`void PlxDmaIsr( IN DMA_CHANNEL dmaChannel);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Services a DMA channel interrupt. If the application is using the DMA Resource Manager to manage the DMA resources then this function call should be placed into the BSP Lint ISR and called when the DMA interrupt trigger is the cause of the interrupt.

- *dmaChannel* is the DMA channel number causing the interrupt.

Return Value:

None.

Usage:

```
RETURN_CODE plxReturnStatus;
PLX_INTR plxIntr;

/* Check if there is actually an interrupt */
if (!PlxIntrStatusGet(PrimaryPciBus, &plxIntr, &plxReturnStatus))
{
  /* No interrupt, or function failure??? */
  return;
}

/* Look for IOP DMA channel 0 interrupt */
if (plxintr.IopDmaChannel0)
{
  /* DMA channel 0 caused interrupt */
  /* Call the DMA manager */
  PlxDmaIsr(PrimaryPciChannel0);
}

/* Look for IOP DMA channel 1 interrupt */
if (plxintr.IopDmaChannel1)
{
  /* DMA channel 1 caused interrupt */
  /* Call the DMA manager */
  PlxDmaIsr(PrimaryPciChannel1);
}
```

PlxDmaResourceManagerInit

Syntax:

```
RETURN_CODE PlxDmaResourceManagerInit(IN PDMA_PARMS dmaParms);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the DMA Resource Manager.

- *dmaParms* is an array of structures that contains the information that is defined within the BSP needed by the DMA Resource Manager.

Note: *This function must be called before any DMA API function. The DMA Manager requires control of the DMA ISR to operate properly. For this reason, modifications or replacements of the main PLX ISR should call the* PlxDmaIsr() *function when a DMA channel is the cause of the interrupt.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaParms* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	A DMA channel for which this function is called is not closed.	PCI 9080, PCI 9054
`ApiDmaManReady`	The DMA manager is already initialized for this channel.	PCI 9080, PCI 9054
`ApiInvalidAddress`	One element in the *dmaParms* array has its `FirstSglElement` pointing to an address that is not on a 16-byte boundary.	PCI 9080, PCI 9054

Comments:

The *dmaParms* structure does not contain a size parameter. Therefore, this API function scans the array of `DMA_PARMS` structures provided by the *dmaParms* pointer for an invalid member. Once an invalid parameter has been found, *PlxDmaResourceManagerInit()* stops parsing the *dmaParms* structure and returns.

When calling *PlxDmaResourceManagerInit()*, the user should do the following:

1. Allocate an array of `DMA_PARMS` structures where the size of the array contains one extra element.
2. Fill the array with the desired values.
3. Fill the last element (the extra one) of the array with one of the following:

   Set the `DmaChannel` member to –1;

   Set the `FirstSglElement` member to NULL;

   Set the `WaitQueueBase` member to NULL; or,

   Set the `NumberOfElements` member to 0.

Note: *Once the* PlxDmaResourceManagerInit() *function is called, the memory buffers provided for the* `WaitQueueBase` *and the* `FirstSglElement` *members of the* `DMA_PARMS` *structure should not be accessed. These memory buffers are reserved for the operation of the DMA Resource Manager.*

See the DMA Resource Manager Parameters Structure for more information.

Usage:

```
RETURN_CODE rc; /* Return code from API functions */
DMA_PARMS ourDmaParms[NUMBER_OF_9080_DMA_CHANNELS + 1];

/*
   DMA manager for channel 0. The actual SGL first element address
   is incremented to the next 16-bytes boundary.
 */
ourDmaParms[0].DmaChannel = PrimaryPciChannel0;
ourDmaParms[0].FirstSglElement = (PDMA_TRANSFER_ELEMENT)0x00700000;
ourDmaParms[0].WaitQueueBase = Dma0SglQueue;
ourDmaParms[0].NumberOfElements = DMA_CHANNEL0_SGL_SIZE;

/*
   DMA manager for channel 1. The actual SGL first element address
   is incremented to the next 16-bytes boundary.
 */
ourDmaParms[1].DmaChannel = PrimaryPciChannel1;
ourDmaParms[1].FirstSglElement = (PDMA_TRANSFER_ELEMENT)0x00701000;
ourDmaParms[1].WaitQueueBase = Dma1SglQueue;
ourDmaParms[1].NumberOfElements = DMA_CHANNEL1_SGL_SIZE;

/*
   Break-channel. Any of the four next lines is enough to provoke
   the break. You only need to use one of them.
 */
ourDmaParms[2].DmaChannel = (DMA_CHANNEL)(-1);
ourDmaParms[2].FirstSglElement = (PDMA_TRANSFER_ELEMENT)NULL;
ourDmaParms[2].WaitQueueBase = NULL;
ourDmaParms[2].NumberOfElements = 0;

rc = PlxDmaResourceManagerInit(ourDmaParms);
if ( rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxDmaSglBuild

Syntax:

```
RETURN_CODE PlxDmaSglBuild( IN DMA_CHANNEL dmaChannel,
                            IN U32 numberOfDmaElements,
                            OUT PSGL_ADDR sglAddress);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

- *dmaChannel* is the DMA channel number;
- *numberOfDmaElements* is the desired number of DMA Transfer Elements for the Scatter-Gather List; and,
- *sglAddress* is a pointer to the Scatter-Gather List address that is built.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *sglAddress* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaManNotReady`	The DMA manager is not ready for this channel.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for SGL DMA transfers.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *numberOfDmaElements* parameter is 0.	PCI 9080, PCI 9054
`ApiDmaNoMoreElements`	There are not enough elements in the SGL Free pool to build the SGL.	PCI 9080, PCI 9054
`ApiDmaManCorrupted`	The DMA manager is corrupted.	PCI 9080, PCI 9054

Usage:

```
#define DRAM_MAP_ADDR         0x00000000
RETURN_CODE rc;
DMA_TRANSFER_ELEMENT dmaData;
SGL_ADDR sglBasePtr,
U32 totalSize = 0x100;
U32 bufFrom = DRAM_MAP_ADDR, bufTo = DRAM_MAP_ADDR + totalSize;
memset(&dmaData, 0, sizeof(DMA_TRANSFER_ELEMENT));
/* Then we fill the resulting local buffer with 0.*/
rc = PlxDmaSglBuild(PrimaryPciChannel0, 2, &sglBasePtr);
if (rc != ApiSuccess) {
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}
/*
   Preparing a list element to perform a DMA data exchange
   from DRAM_MAP_ADDR of length totalSize to pciStartOffset
 */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufFrom;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 1;
rc = PlxDmaSglFill(sglBasePtr, 0, &dmaData);
if (rc != ApiSuccess) {
  PlxPrintf("SglFill failed\n");
  return -1;
}
/*
   Preparing a list element to perform a DMA data exchange
   from pciStartOffset of length totalSize to DRAM_MAP_ADDR
 */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufTo;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 0;
rc = PlxDmaSglFill(sglBasePtr, 1, &dmaData);
if (rc != ApiSuccess) {
  PlxPrintf("SglFill failed\n");
  return -1;
}

/*
   The DMA Manager has already been initialized.
   2 elements were allocated for channel 0.
 */
rc = PlxDmaSglTransfer(PrimaryPciChannel0,
                       DmaStart,
                       sglBasePtr,
                       FALSE);
if (rc != ApiSuccess)
{
  PlxPrintf("Error while performing a SGL DMA with channel.\n");
  return -1;
}
```

PlxDmaSglChannelClose

Syntax:

`RETURN_CODE PlxDmaSglChannelClose( IN DMA_CHANNEL dmaChannel);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes the Scatter-Gather DMA channel.

- *dmaChannel* is the DMA channel number previously opened.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for SGL DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;

rc = PlxDmaSglChannelClose(PrimaryPciChannel0);
if (rc != ApiSuccess)
{
  PlxPrintf("Error closing DMA channel.\n");
  return -1;
}
```

PlxDmaSglChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes and opens a DMA channel for Scatter-Gather DMA transfers.

- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Note: *Before calling this function a call to PlxDmaResourceManagerInit() must be done to initialize the DMA Resource Manager.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaChannelDesc* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaManNotReady`	The DMA manager is not ready for this DMA channel.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
DMA_CHANNEL_DESC desc;

desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
desc.DmaStopTransferMode = AssertBLAST;
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
```

```
desc.HoldIopDestAddrConst = 0;
desc.DemandMode = 0;
desc.EnableTransferCountClear = 0; /* don't clear count */
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

rc = PlxDmaSglChannelOpen(PrimaryPciChannel, &desc);
if (rc != ApiSuccess)
{
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}
```

PlxDmaSglFill

Syntax:

```
RETURN_CODE PlxDmaSglFill( IN SGL_ADDR sglAddress,
                           IN U32 sglIndex,
                           IN PDMA_TRANSFER_ELEMENT dmaData);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Fills a DMA Transfer Element within a Scatter-Gather List.

- *sglAddress* is the Scatter-Gather List address;
- *sglIndex* is the index value for the desired DMA Transfer Element contained within the Scatter-Gather List; and,
- *dmaData* is the data for the DMA transfer element.

Note: *Before calling this function a valid Scatter-Gather List address must be received from PlxDmaSglBuild().*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *sglAddress* parameter or the *dmaData* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaInvalidElementIndex`	The DMA Manager cannot find the *sglIndex* element within the SGL.	PCI 9080, PCI 9054

Comments:

The `DMA_TRANSFER_ELEMENT` structure contains members whose signification may differ depending on the DMA transfer type including ignoring some of the members. Here is the structure:

Structure Element	Signification
LowPciAddr	The low 32 bits of the PCI address for the transfer.
HighPciAddr	The high 32 bits of the PCI address for the transfer. This field is ignored for the PCI 9080.
IopAddr	The IOP address for the transfer.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Indicates if an interrupt is to be generated when this particular SGL element is done.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.

Structure Element	Signification
NextSglPtr	Ignored.

Usage:

```
#define DRAM_MAP_ADDR         0x00000000
RETURN_CODE rc;
DMA_TRANSFER_ELEMENT dmaData;
SGL_ADDR sglBasePtr,
U32 totalSize = 0x100;
U32 bufFrom = DRAM_MAP_ADDR, bufTo = DRAM_MAP_ADDR + totalSize;
memset(&dmaData, 0, sizeof(DMA_TRANSFER_ELEMENT));
/* Then we fill the resulting local buffer with 0.*/
rc = PlxDmaSglBuild(PrimaryPciChannel0, 2, &sglBasePtr);
if (rc != ApiSuccess) {
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}
/* Preparing a list element to perform a DMA data exchange
   from DRAM_MAP_ADDR of length totalSize to pciStartOffset */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufFrom;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 1;
rc = PlxDmaSglFill(sglBasePtr, 0, &dmaData);
if (rc != ApiSuccess) {
  PlxPrintf("SglFill failed\n");
  return -1;
}
/* Preparing a list element to perform a DMA data exchange
   from pciStartOffset of length totalSize to DRAM_MAP_ADDR */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufTo;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 0;
rc = PlxDmaSglFill(sglBasePtr, 1, &dmaData);
if (rc != ApiSuccess) {
  PlxPrintf("SglFill failed\n");
  return -1;
}
/*
   The DMA Manager has already been initialized.
   2 elements were allocated for channel 0.
 */
rc = PlxDmaSglTransfer(PrimaryPciChannel0,
                       DmaStart,
                       sglBasePtr,
                       FALSE);
if (rc != ApiSuccess)
{
  PlxPrintf("Error while performing a SGL DMA with channel.\n");
  return -1;
}
```

PlxDmaSglTransfer

Syntax:

```
RETURN_CODE PlxDmaSglTransfer( IN DMA_CHANNEL dmaChannel,
                               IN DMA_COMMAND dmaCommand,
                               IN SGL_ADDR sglAddress,
                               IN BOOLEAN returnImmediate);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Controls the Scatter-Gather DMA transfer for a given DMA channel.

- *dmaChannel* is the DMA channel number previously opened;
- *dmaCommand* is the action to perform on this DMA channel;
- *sglAddress* is the Scatter-Gather List address; and,
- *returnImmediate* determines if this function waits for the DMA command to complete before returning.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *sglAddress* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for SGL DMA.	PCI 9080, PCI 9054
`ApiDmaManNotReady`	The DMA manager is not ready for this channel.	PCI 9080, PCI 9054
`ApiDmaSglInvalid`	The *sglAddress* parameter is invalid.	PCI 9080, PCI 9054
`ApiDmaSglQueueFull`	The waiting queue is full.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080,

Return Value	Description	API Support
		PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused).	PCI 9080, PCI 9054
`ApiDmaManCorrupted`	The DMA manager is corrupted.	PCI 9080, PCI 9054
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054

Comments:.

Once the PCI IC is done with a Scatter-Gather DMA transfer, an interrupt is generated and *PlxDmaIsr()* deallocates the Scatter-Gather List represented by `sglAddress` and returns it to the free pool of Scatter-Gather List elements.

Note: *Once* PlxDmaSglTransfer() *is called, the Scatter-Gather List provided in the sglAddress parameter should no longer be used with* PlxDmaSglFill() *function.*

The following table lists the possible actions used to control a SGL DMA transfer:

Action (*dmaCommand* parameter)	Description
DmaStart	Program and start a DMA channel for a Scatter-Gather DMA transfer or queue this transfer request if one is already in progress.
DmaPause	Pause the entire DMA channel. The *dmaData*, *sglAddress* and *returnImmediate* parameters are ignored.
DmaResume	Resume a paused DMA channel. The *dmaData*, *sglAddress* and *returnImmediate* parameters are ignored.
DmaAbort	Abort the transfer currently in progress along with all pending transfers for this DMA channel. The *dmaData*, *sglAddress* and *returnImmediate* parameters are ignored.
DmaStatus	Gives a status of the DMA channel. The *dmaData*, *sglAddress* and *returnImmediate* parameters are ignored.

Usage:

```
#define DRAM_MAP_ADDR         0x00000000

RETURN_CODE rc;
DMA_TRANSFER_ELEMENT dmaData;
SGL_ADDR sglBasePtr,
U32 totalSize = 0x100;
U32 bufFrom = DRAM_MAP_ADDR, bufTo = DRAM_MAP_ADDR + totalSize;
```

```
memset(&dmaData, 0, sizeof(DMA_TRANSFER_ELEMENT));

/* Then we fill the resulting local buffer with 0.*/
rc = PlxDmaSglBuild(PrimaryPciChannel0, 2, &sglBasePtr);
if (rc != ApiSuccess)
{
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}

/*
   Preparing a list element to perform a DMA data exchange
   from DRAM_MAP_ADDR of length totalSize to pciStartOffset
 */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufFrom;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 1;
rc = PlxDmaSglFill(sglBasePtr, 0, &dmaData);
if (rc != ApiSuccess)
{
  PlxPrintf("SglFill failed\n");
  return -1;
}

/*
   Preparing a list element to perform a DMA data exchange
   from pciStartOffset of length totalSize to DRAM_MAP_ADDR
 */
dmaData.Pci9080Dma.LowPciAddr = 0x1BC0000;
dmaData.Pci9080Dma.IopAddr = bufTo;
dmaData.Pci9080Dma.TransferCount = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;
dmaData.Pci9080Dma.IopToPciDma = 0;
rc = PlxDmaSglFill(sglBasePtr, 1, &dmaData);
if (rc != ApiSuccess)
{
  PlxPrintf("SglFill failed\n");
  return -1;
}

/*
   The DMA Manager has already been initialized.
   2 elements were allocated for channel 0.
 */
rc = PlxDmaSglTransfer(PrimaryPciChannel0,
                       DmaStart,
                       sglBasePtr,
                       FALSE);
if (rc != ApiSuccess)
{
  PlxPrintf("Error while performing a SGL DMA with channel.\n");
  return -1;
}
```

PlxDmaShuttleChannelClose

Syntax:

`RETURN_CODE PlxDmaShuttleChannelClose( IN DMA_CHANNEL dmaChannel);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Stop and closes the Shuttle DMA channel.

- *dmaChannel* is the DMA channel number previously opened.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Shuttle DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;

rc = PlxDmaShuttleChannelClose(PrimaryPciChannel0);
if (rc != ApiSuccess)
{
  PlxPrintf("Error closing DMA channel.\n");
  return -1;
}
```

PlxDmaShuttleChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Opens and initializes a DMA channel for Shuttle DMA transfers. This call also starts the DMA channel.

- *dmaChannel* is the DMA channel number; and,
- *dmaChannelDesc* is a structure containing the DMA channel descriptors.

Note: *Before calling this function a call to PlxDmaResourceManagerInit() must be done to initialize the DMA Resource Manager.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaChannelDesc* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaManNotReady`	The DMA manager is not ready for this DMA channel.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
DMA_CHANNEL_DESC desc;

/* Setup DMA configuration structure */
desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
```

```
desc.DmaStopTransferMode = AssertBLAST;
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
desc.HoldIopDestAddrConst = 0;
desc.DemandMode = 0;
desc.EnableTransferCountClear = 1;
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

rc = PlxDmaShuttleChannelOpen(PrimaryPciChannel0, &desc);
if (rc != ApiSuccess)
{
  PlxPrintf("Error opening DMA channel.\n");
  return -1;
}
```

PlxDmaShuttleTransfer

Syntax:

```
RETURN_CODE PlxDmaShuttleTransfer( IN DMA_CHANNEL dmaChannel,
                                   IN U32 shuttleIndex,
                                   IN DMA_COMMAND dmaCommand,
                                   IN PDMA_TRANSFER_ELEMENT dmaData,
                                   IN BOOLEAN returnImmediate);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Controls the Shuttle DMA transfer for a given DMA channel.

- *dmaChannel* is the DMA channel number previously opened;
- *shuttleIndex* is the index value for the desired DMA Transfer Element;
- *dmaCommand* is the action to perform on this DMA channel;
- *dmaData* is the data for the DMA transfer; and,
- *returnImmediate* determines if this function waits for the DMA command to complete before returning.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *dmaData* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Shuttle DMA.	PCI 9080, PCI 9054
`ApiDmaInvalidElementIndex`	The *shuttleIndex* parameter is out of range.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080,

Return Value	Description	API Support
		PCI 9054
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused).	PCI 9080, PCI 9054
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054

Comments:

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

The following table lists the possible actions used to control a Shuttle DMA transfer:

Action (*dmaCommand* parameter)	Description
DmaStart	Program *shuttleIndex* DMA transfer element.
DmaPause	Pause the entire DMA channel. The *dmaData*, *shuttleIndex* and *returnImmediate* parameters are ignored.
DmaResume	Resume a paused DMA channel. The *dmaData*, *shuttleIndex* and *returnImmediate* parameters are ignored.
DmaAbort	Abort all Shuttle transfer elements currently programmed. The *dmaData* and *returnImmediate* parameters are ignored.
DmaStatus	Gives a status of the transfer previously programmed in the *shuttleIndex* scatter-gather element. The *dmaData* and *returnImmediate* parameters are ignored.

The DMA_TRANSFER_ELEMENT structure contains members whose signification may differ depending on the DMA transfer type. Also, some other members are ignored by the API call. Here is the structure:

Structure Element	Signification
LowPciAddr	The low 32 bits of the PCI address for the transfer.
HighPciAddr	The high 32 bits of the PCI address for the transfer. This field is ignored for the PCI 9080.
IopAddr	The IOP address for the transfer.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.
TerminalCountIntr	Indicates if an interrupt is to be generated when this particular SGL element is done.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

```
#define DRAM_MAP_ADDR         0x00000000

RETURN_CODE rc;
DMA_TRANSFER_ELEMENT dmaData;
U32 totalSize = 0x100;
U32 bufFrom = DRAM_MAP_ADDR;

/* Setting dmaData */
dmaData.Pci9080Dma.LowPciAddr        = 0x555B0000;
dmaData.Pci9080Dma.IopAddr           = bufFrom;
dmaData.Pci9080Dma.TransferCount     = totalSize;
dmaData.Pci9080Dma.TerminalCountIntr = 0;/* no interrupts */
dmaData.Pci9080Dma.IopToPciDma       = 1;

/*
   Start copying data from address bufFrom to the PCI address.
   The call returns immediately.
 */
rc = PlxDmaShuttleTransfer(PrimaryPciChannel0,
                           0,
                           DmaStart,
                           &dmaData,
                           FALSE);
if (rc != ApiSuccess)
{
  PlxPrintf("Error while performing a Shuttle DMA transfer.\n");
  return -1;
}
```

PlxDmaShuttleTransferRestart

Syntax:

```
RETURN_CODE PlxDmaShuttleTransferRestart( IN DMA_CHANNEL dmaChannel,
                                          IN U32 shuttleIndex,
                                          IN U32 transferSize,
                                          IN BOOLEAN returnImmediate);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

- *dmaChannel* is the DMA channel number previously opened;
- *shuttleIndex* is the index value for the desired DMA Transfer Element previously programmed;
- *transferSize* is the DMA transfer size; and,
- *returnImmediate* determines if this function waits for the DMA command to complete before returning.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelTypeError`	The DMA channel was not opened for Shuttle DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054
`ApiDmaInvalidElementIndex`	The *shuttleIndex* parameter is out of range.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 totalSize = 0x100;

rc = PlxDmaShuttleTransferRestart(PrimaryPciChannel0,
                                  0,
                                  totalSize,
                                  FALSE);
if (rc != ApiSuccess)
{
    PlxPrintf("Restart failed\n");
    return -1;
}
```

PlxEchoEnable

Syntax:

`ECHO_STATE PlxEchoEnable(IN ECHO_STATE echoState);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Enables or disables echoing of characters received from the serial port by the UART Services Module.

- *echoState* is the new echo state for the UART Services Module.

Return Value:

The previous echo state is returned.

Usage:

```
ECHO_STATE previousEchoState;

/* Disable Echoing of Characters */
previousEchoState = PlxEchoEnable(NoEcho);
```

PlxGetChars

Syntax:

```
U16 PlxGetChars( OUT PU8 outBuffer,
                 IN U16 outBufferSize);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Gets a stream of data from the serial port.

- *outBuffer* is a buffer to store the data stream received; and,
- *outBufferSize* is the number of characters to retrieve from the serial port.

Return Value:

The number of characters placed into the buffer.

Usage:

```
#define POWERQUICC_SPEED   40 /* In MHz */

U32 ulCount;
U8 c;
S16 resVal;
ulCount = 10 * POWERQUICC_SPEED * seconds;

while ( ulCount -- )
{
  if ( PlxGetChars(&c,1) == 1)
  {
    resVal = ((S16)c) & 0x0FF;
    return (resVal);
  }
  Sleep(1);
}
```

PlxHotSwapDisable

Syntax:

`RETURN_CODE PlxHotSwapDisable( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Disables PCI access to the Hot Swap capability of the chip.

- *busIndex* is the bus index.

***Note**: Hot Swap is enabled by default. Enabling or disabling hot swap capabilities after the Init done bit is set can cause problems with the host Operating System.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054

Usage:

```
RETURN_CODE rc;

/* Disable Hot Swap  */
rc = PlxHotSwapDisable(PrimaryPciBus);

if (rc != ApiSuccess)
  PlxSdkError(rc, TRUE);
```

PlxHotSwapEnable

Syntax:

`RETURN_CODE PlxHotSwapEnable( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Enables PCI access to the Hot Swap capability of the PCI IC.

- *busIndex* is the bus index.

***Note**: Hot Swap is enabled by default. Enabling or disabling hot swap capabilities after the Init done bit is set can cause problems with the host Operating System.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054

Usage:

```
RETURN_CODE rc;

/* Enable Hot Swap */
rc = PlxHotSwapEnable(PrimaryPciBus);

if (rc != ApiSuccess)
  PlxSdkError(rc, TRUE);
```

PlxHotSwapId

Syntax:

`U8 PlxHotSwapId( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Reads the Hot Swap Id register.

- *busIndex* is the bus index.

Return Value:

Return Value	Description	API Support
`0x06`	The function read successfully the Hot Swap ID.	PCI 9054
`0xFF`	Hot Swap is not supported or not present on this board.	PCI 9054

Usage:

```
U8 id;

/* Print the Hot Swap ID */
id = PlxHotSwapId(PrimaryPciBus);
if (id == 0xFF)
  PlxPrintf("Hot Swap is not supported\n");
else
  PlxPrintf("Hot Swap ID is 0x%02x\n", id);
```

PlxHotSwapNextItemPointer

Syntax:

`U8 PlxHotSwapNextItemPointer( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Returns the Hot Swap Next Item Pointer Register value.

- *busIndex* is the bus index.

Return Value:

Return Value	Description	API Support
`0xFF`	Hot swap is not supported or not present on this board.	PCI 9054
`0x00`	There is no more new capability.	PCI 9054
`0xXX`	Next item pointer register value (usually 0x4C).	PCI 9054

Usage:

```
U8 value;

/* Get the next item pointer register  */
value = PlxHotSwapNextItemPointer(PrimaryPciBus);
if (value == 0)
  PlxPrintf("There is no more new capability\n");
else if (value == 0xFF)
  PlxPrintf("Hot Swap is not supported on this board\n");
else
  PlxPrintf("The next item pointer register is 0x%02x\n", value);
```

PlxHotSwapStatus

Syntax:

`U8 PlxHotSwapStatus( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Returns the Hot-Swap status of the PLX chip.

- *busIndex* is the bus index.

Return Value:

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

Return Value	Description	API Support
HOT_SWAP_LED_IS_ON	The Hot Swap LED is on.	PCI 9054
HOT_SWAP_BOARD_IS_BEING_REMOVED	The board is in process of being removed.	PCI 9054
HOT_SWAP_BOARD_IS_BEING_INSERTED	The board was inserted and is being initialized.	PCI 9054

However, if Hot Swap is not supported or not present on the board, the return value will be 0xFF.

Usage:

```
U8 value;

/* Get the status  */
value = PlxHotSwapStatus(PrimaryPciBus);
if (value == 0xFF)
  PlxPrintf("Hot swap is not supported\n");
else
{
  if (value & HOT_SWAP_LED_IS_ON)
    PlxPrintf("The LED is on\n");
  if (value & HOT_SWAP_BOARD_IS_BEING_REMOVED)
    PlxPrintf("The board is about to be removed\n");
  if (value & HOT_SWAP_BOARD_IS_BEING_INSERTED)
    PlxPrintf("The board was just inserted\n");
}
```

PlxInitApi

Syntax:

`RETURN_CODE PlxInitApi( IN PAPI_PARMS apiParms);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the IOP API and sets the data structures used for initialization to the PCI IC's default values.

- *apiParms* is a structure that contains the information that is defined within the BSP needed by the IOP API.

Note: *This function must be called before any other IOP API function.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiAlreadyInitialized`	The IOP API has been initialized already.	PCI 9080
`ApiNullParam`	The *apiParms* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The PLX PCI IC IOP Base Address provided is not on a 4-byte boundary.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
API_PARMS apiInit;             /* API Init structure */
PCI_BUS_PROP pciBusProp;       /* PCI Bus Properties  */
IOP_BUS_PROP localBusProp0;    /* IOP Space 0 Properties  */
IOP_BUS_PROP localBusProp1;    /* IOP Space 1 Properties  */
IOP_BUS_PROP localBusPropRom;  /* Expansion ROM Properties  */
IOP_ARBIT_DESC arbDesc;        /* IOP Space Arbitration Properties */
IOP_ENDIAN_DESC localEndianDesc; /* Endian properties */
PLX_INTR plxIntr;               /* Interrupt to enable at boot */
U32 value;

/* Clear structure first */
memset(&apiInit, 0, sizeof(API_PARMS));

/*
   The PlxInitApi function will initialize the following structures
   with the default bit field settings corresponding to the IC
   reset values.
```

```
 */
apiInit.PlxIcIopBaseAddr = (void *)EMBED_PLX_MAP_ADDR;
apiInit.PtrPciBusProp = &pciBusProp;        /* Init Structure */
apiInit.PtrPciArbitDesc = NULL;             /* Unused */
apiInit.PtrIopBus0Prop = &localBusProp0;    /* Init Structure */
apiInit.PtrIopBus1Prop = &localBusProp1;    /* Init Structure */
apiInit.PtrIopBus2Prop = NULL;              /* Unused */
apiInit.PtrIopBus3Prop = NULL;              /* Unused */
apiInit.PtrExpRomBusProp = &localBusPropRom; /* Init Structure */
apiInit.PtrIopArbitDesc = &arbDesc;         /* Init Structure */
apiInit.PtrIopEndianDesc = &localEndianDesc; /* Init Structure */

/* Initialize API and structures */
rc = PlxInitApi(&apiInit);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitDone

Syntax:

`RETURN_CODE PlxInitDone( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Signals PCI device to allow external PCI masters (i.e. PCI BIOS) access to the PCI device on the given bus index.

- *busIndex* is the bus index.

Note: *Upon completion of initialization by the IOP processor, this function must be called in order to give access to external PCI masters to the PCI device. Failure to do so will result in system hang.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;


/*
   All PCI 9080 Registers should be initialized before this point.
   Now set the Local Init Done Bit so the computer's BIOS can
   continue the boot process.
   WARNING: If you do not set this bit the computer will never boot.
 */
rc = PlxInitDone(PrimaryPciBus);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxInitIopArbitration

Syntax:

```
RETURN_CODE PlxInitIopArbitration( IN BUS_INDEX busIndex,
                                   IN PIOP_ARBIT_DESC arbDesc);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the IOP bus arbitration registers for the PCI IC on the given bus index.

- *busIndex* is the bus index; and,
- *arbDesc* is a pointer to a structure containing the IOP bus arbitration information.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *arbDesc* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
IOP_ARBIT_DESC arbDesc;    /* IOP Space Arbitration Properties */

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

/*************************************
    Initialize the IOP arbitration
*************************************/
/* enable latency timer */
arbDesc.EnableIopBusLatencyTimer = 1;
/*
   number of IOP bus clocks before negating HOLD and
   releasing IOP bus
*/
arbDesc.IopBusLatencyTimer = 0x1C;
/* Initialize the IOP arbitration now */
rc = PlxInitIopArbitration(PrimaryPciBus, &arbDesc);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitIopBusProperties

Syntax:

```
RETURN_CODE PlxInitIopBusProperties( IN BUS_INDEX busIndex,
                                     IN IOP_SPACE iopSpace,
                                     IN PIOP_BUS_PROP iopBusProp);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the IOP bus properties registers for the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *iopSpace* is the specific local space; and,
- *iopBusProp* is a structure containing the IOP bus properties for the IOP space specified.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *iopBusProp* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidIopSpace`	The *iopSpace* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054

Usage:

```
#define DRAM_MAP_ADDR         0x00000000
#define LS0_WINDOW_SIZE       0x00800000 /* Local Space 0 8MB  */

RETURN_CODE rc;
IOP_BUS_PROP iopBusProp0;     /* IOP Space 0 Properties  */

/* Clear structure first */
memset(&iopBusProp0, 0, sizeof(IOP_BUS_PROP));

/*****************************************
   Initialize IOP Space 0
 *****************************************/
```

```
/* enable Ready input */
iopBusProp0.EnableReadyInput = 1;
/* disable read prefetching */
iopBusProp0.DisableReadPrefetch = 1;
/* enable bursting */
iopBusProp0.EnableBursting = 1;
/* enables 9080 from asserting BREQo output */
iopBusProp0.EnableIopBREQo = 1;
/* 32 bits bus */
iopBusProp0.IopBusWidth = 3;
/* Init the Local space 0 bus properties now */
rc = PlxInitIopBusProperties(PrimaryPciBus,
                             IopSpace0,
                             &iopBusProp0);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);

/* Initialize and enable local space 0. Remap it to DRAM.*/
rc = PlxInitLocalSpace(PrimaryPciBus,
                       IopSpace0,
                       DRAM_MAP_ADDR,
                       LS0_WINDOW_SIZE);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitIopEndian

Syntax:

```
RETURN_CODE PlxInitIopEndian( IN BUS_INDEX busIndex,
                              IN PIOP_ENDIAN_DESC iopEndianDesc);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the IOP Space endian type accesses for the PCI IC on the given bus index.

- *busIndex* is the bus index; and,
- *iopEndianDesc* is a pointer to a structure defining the IOP Space endian type accesses.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *iopEndianDesc* parameter is NULL.	PCI 9080, PCI 9054
`ApiBadConfigRegEndianMode`	The member `BigEIopConfigRegAccess` of *iopEndianDesc* is invalid for this board configuration.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
IOP_ENDIAN_DESC localEndianDesc; /* Endian properties */

/* Clear structure first */
memset(&localEndianDesc, 0, sizeof(IOP_ENDIAN_DESC));

/* ------------- init. IOP endian properties -------------------- */
/*
   The PlxInitIopEndian function MUST be called first so that all
   subsequent accesses to PCI 9080 registers are correct.
 */

/* DMA channel 0 runs in big endian mode */
localEndianDesc.BigEDmaChannel0 = 1;
```

```
/* DMA channel 1 runs in big endian mode */
localEndianDesc.BigEDmaChannel1 = 1;
/* PCI-9080 regs are accessed in big endian mode from local bus */
localEndianDesc.BigEIopConfigRegAccess = 1;

/* Now init the endian properties */
rc = PlxInitIopEndian(PrimaryPciBus, &localEndianDesc);
if (rc == ApiBadConfigRegEndianMode)
{
  /*
     The API has returned that the BigEIopConfigRegAccess bit
     was setup wrong. Just switch it and try again.
   */
  localEndianDesc.BigEIopConfigRegAccess =
              1 - localEndianDesc.BigEIopConfigRegAccess;
  rc = PlxInitIopEndian(PrimaryPciBus, &localEndianDesc);
}
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxInitLocalSpace

Syntax:

```
RETURN_CODE PlxInitLocalSpace( IN BUS_INDEX busIndex,
                               IN IOP_SPACE iopSpace,
                               IN U64 localBaseAddress,
                               IN U64 size);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the Local Space descriptor registers for the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *iopSpace* is the specific local space;
- *localBaseAddress* is the IOP base remap address for any PCI access to the local space; and,
- *size* is the size in bytes of the region of access (the window).

Note: size *must be a power of 2 and* localBaseAddress *must be a multiple of* size.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiInvalidIopSpace`	The *iopSpace* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *size* parameter is not a power of 2, or is too small for *iopSpace*.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *localBaseAddress* parameter is not a multiple of *size*.	PCI 9080, PCI 9054

Usage:

```
#define DRAM_MAP_ADDR         0x00000000
#define LS0_WINDOW_SIZE       0x00800000 /* Local Space 0 8MB  */

RETURN_CODE rc;
IOP_BUS_PROP iopBusProp0;     /* IOP Space 0 Properties  */
```

```
/* Clear structure first */
memset(&iopBusProp0, 0, sizeof(IOP_BUS_PROP));

/*****************************************
   Initialize IOP Space 0
 *****************************************/

/* enable Ready input */
iopBusProp0.EnableReadyInput = 1;
/* disable read prefetching */
iopBusProp0.DisableReadPrefetch = 1;
/* enable bursting */
iopBusProp0.EnableBursting = 1;
/* enables 9080 from asserting BREQo output */
iopBusProp0.EnableIopBREQo = 1;
/* 32 bits bus */
iopBusProp0.IopBusWidth = 3;
/* Init the Local space 0 bus properties now */
rc = PlxInitIopBusProperties(PrimaryPciBus,
                             IopSpace0,
                             &iopBusProp0);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);

/* Initialize and enable local space 0. Remap it to DRAM.*/
rc = PlxInitLocalSpace(PrimaryPciBus,
                       IopSpace0,
                       DRAM_MAP_ADDR,
                       LS0_WINDOW_SIZE);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitPciArbitration

Syntax:

```
RETURN_CODE PlxInitPciArbitration( IN BUS_INDEX busIndex,
                                   IN PPCI_ARBIT_DESC arbDesc);
```

PCI IC Support:

None

Description:

Initializes the PCI bus arbitration registers for the PCI IC on the given bus index.

- *busIndex* is the bus index; and,
- *arbDesc* is a pointer to a structure containing the PCI bus arbitration information.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *arbDesc* parameter is NULL.	PCI 9080, PCI 9054
`ApiUnsupportedFunction`	This function is not supported for this PCI IC.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
PCI_ARBIT_DESC pciArbitDesc;        /* PCI Arbitration Desc */

/* Clear structure first */
memset(&pciArbitDesc, 0, sizeof(PCI_ARBIT_DESC));

rc = PlxInitPciArbitration(PrimaryPciBus, &pciArbitDesc);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitPciBusProperties

Syntax:

```
RETURN_CODE PlxInitPciBusProperties( IN BUS_INDEX busIndex,
                                     IN PPCI_BUS_PROP pciBusProp);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the PCI bus properties registers for the PCI IC on the given bus index.

- *busIndex* is the PCI bus index; and,
- *pciBusProp* is a structure containing the PCI bus properties for the PCI space specified.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *pciBusProp* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
PCI_BUS_PROP pciBusProp;        /* PCI Bus Properties  */

/* Clear structure first */
memset(&pciBusProp, 0, sizeof(PCI_BUS_PROP));

rc = PlxInitPciBusProperties(PrimaryPciBus, &pciBusProp);
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitPciSpace

Syntax:

```
RETURN_CODE PlxInitPciSpace( IN BUS_INDEX busIndex,
                             IN PCI_SPACE pciSpace,
                             IN U64 pciWindowLocalBaseAddress,
                             IN U64 size);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Initializes the PCI Space descriptor registers for the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *pciSpace* is the specific PCI space;
- *pciWindowLocalBaseAddress* is the local base address for the PCI bus Direct Master window; and,
- *size* is the size in bytes of the region of access (the window).

Note: size *must be a power of 2 and* pciWindowLocalBaseAddress *must be a multiple of* size.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiInvalidPciSpace`	The *pciSpace* parameter is neither PciMemSpace nor `PciIoSpace`.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *size* parameter is not a power of 2 or is less than 64K.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *pciWindowLocalBaseAddress* parameter is not a multiple of *size*.	PCI 9080, PCI 9054

Usage:

```
#define DM_SIZE              0x08000000
#define DM_MEM_MAP_ADDR      0xB8000000
#define DM_IO_MAP_ADDR       0xB0000000

RETURN_CODE rc;
```

```
/*********************************************
   Initialize Direct Master Memory and IO
 *********************************************/
/* Init for Direct Master memory access */
rc = PlxInitPciSpace( PrimaryPciBus,
                      PciMemSpace,
                      DM_MEM_MAP_ADDR, /* local base address
                                          for DM memory */
                      DM_SIZE);   /* size of window */
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);

/* Init for Direct Master IO accesses */
rc = PlxInitPciSpace( PrimaryPciBus,
                      PciIoSpace,
                      DM_IO_MAP_ADDR,  /* local base address
                                          for DM IO/CFG */
                      DM_SIZE);  /* size of window */
if (rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxInitPowerManagement

Syntax:

```
RETURN_CODE PlxInitPowerManagement( IN BUS_INDEX busIndex,
                                    IN PPM_PROP pmProp);
```

PCI IC Support:

PCI 9054

Description:

Initializes the power management capabilities of the PLX IC on the given bus index.

- *busIndex* is the PCI bus index; and,
- *pmProp* is a structure containing the power management properties for this device.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054
`ApiNullParam`	The *pmProp* parameter is NULL.	PCI 9054

Usage:

```
RETURN_CODE rc;
PM_PROP pmProp;

/* Clear structure first */
memset(&pmProp, 0, sizeof(PM_PROP));

/* Establish startup properties for both the Power Management Command
Status Register and the Power Management Capabilites Register */

pmProp.Version = 1; /* Compliant to 1.0 of the Power Management Spec */
pmProp.PMEClockNeeded = 0; /* no clock needed for PME generation */
pmProp.DeviceSpecialInit = 0; /* no special init on switch to D0 */
pmProp.D1Supported = 0; /* The D1 Power state is not supported */
pmProp.D2Supported = 0; /* The D2 Power state is not supported */
pmProp.AssertPMEfromD0 = 1; /* A PME is asserted from the D0 state */
pmProp.AssertPMEfromD1 = 0; /* This state is not supported */
pmProp.AssertPMEfromD2 = 0; /* This state is not supported */
pmProp.AssertPMEfromD3Hot = 1; /* A PME is asserted from D3Hot state */

rc = PlxInitPowerManagement(PrimaryPciBus, &pmProp);

if (rc != ApiSuccess)
  PlxSdkError((U32)rc, TRUE);
```

PlxInitVpdAddress

Syntax:

```
RETURN_CODE PlxInitVpdAddress( IN BUS_INDEX busIndex,
                               IN U32 vpdBaseAddress);
```

PCI IC Support:

PCI 9054

Description:

Initializes the base address for the Vital Product Data to be accessed.

- *busIndex* is the PCI bus index; and,
- *vpdBaseAddress* is the starting U32-sized index in the configuration EEPROM for the Vital Product Data. Multiply this value by four to obtain the byte offset address.

Note: *Any offset less than 0x30 could contain configuration data used to initialize the PLX IC.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054
`ApiInvalidAddress`	The base address given is too large (0x80 or more)	PCI 9054

Usage:

```
RETURN_CODE rc;

rc = PlxInitVpdAddress(PrimaryPciBus,
0x30); /* 0x30 * sizeof(U32) byte offset */

if (rc != ApiSuccess)
    PlxSdkError( (U32)rc, TRUE);
```

PlxIntrDisable

Syntax:

```
RETURN_CODE PlxIntrDisable( IN BUS_INDEX busIndex,
                            IN PPLX_INTR plxIntr);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Disables specific interrupts of the PCI IC for a given bus index.

- *busIndex* is the bus index; and,
- *plxIntr* is the interrupt structure that describes which interrupts will be disabled.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *plxIntr* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;                 /* Return code from API functions */
PLX_INTR plxIntr;               /* Interrupt to enable at boot */


/* Set plxIntr Structure to ZERO before disabling interrupts */
for ( value = 0; value < sizeof(PLX_INTR); value++)
  ((PU8)&plxIntr)[value] = 0;

/* Disable the DMA interrupts */
plxIntr.IopDmaChannel0 = 1;         /* DMA channel 0 */
plxIntr.IopDmaChannel1 = 1;         /* DMA channel 1 */

rc = PlxIntrDisable(PrimaryPciBus, &plxIntr);
if ( rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxIntrEnable

Syntax:

```
RETURN_CODE PlxIntrEnable( IN BUS_INDEX busIndex,
                           IN PPLX_INTR plxIntr);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Enables specific interrupts of the PCI IC for a given bus index.

- *busIndex* is the bus index; and,
- *plxIntr* is the interrupt structure that describes which interrupts will be enabled.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080, PCI 9054
`ApiNullParam`	The *plxIntr* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;                 /* Return code from API functions */
PLX_INTR plxIntr;               /* Interrupt to enable at boot */
/* --------- Init. PCI9080 interrupt control properties ---------- */
/* Set plxIntr Structure to ZERO before enabling interrupts */
for ( value = 0; value < sizeof(PLX_INTR); value++)
  ((PU8)&plxIntr)[value] = 0;
/*
   WARNING:  DMA interrupts must be enabled for the IOP API DMA
             Resource Manager to function properly.
 */
plxIntr.IopDmaChannel0 = 1;         /* DMA channel 0 */
plxIntr.IopDmaChannel1 = 1;         /* DMA channel 1 */
plxIntr.IopDoorbell = 1;            /* PCI to IOP doorbell */
plxIntr.PciDoorbell = 1;            /* IOP to PCI doorbell */
plxIntr.PciMainInt = 1;             /* PCI interrupt enable */
plxIntr.IopMainInt = 1;             /* IOP interrupt enable */
rc = PlxIntrEnable(PrimaryPciBus, &plxIntr);
if ( rc != ApiSuccess)
  PlxSdkError( (U32) rc, TRUE);
```

PlxIntrStatusGet

Syntax:

```
BOOLEAN PlxIntrStatusGet( IN BUS_INDEX busIndex,
                          OUT PPLX_INTR plxIntr,
                          OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Determines which interrupts are currently active of the PCI IC for the given bus index.

- *busIndex* is the bus index;
- *plxIntr* is the interrupt structure that contains information detailing which interrupts are active; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

This function returns TRUE when an interrupt is active and FALSE when no interrupts are active. The status of the function call is returned via the *returnCode* parameter. The return codes are as follows:

Return Value	Description	API Support
ApiSuccess	The function returned successfully.	PCI 9080, PCI 9054
ApiInvalidBusIndex	The *busIndex* value is not valid.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE plxReturnStatus; /* Return code from API functions */
PLX_INTR plxIntr;

/* Check if there is actually an interrupt */
if (!PlxIntrStatusGet(PrimaryPciBus, &plxIntr, &plxReturnStatus))
{
  /* No interrupt, or function failure??? */
  return;
}
```

PlxMuInboundPortRead

Syntax:

```
RETURN_CODE PlxMuInboundPortRead( IN BUS_INDEX busIndex,
                                  OUT PU32 framePointer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Returns a posted message frame from the Inbound Post FIFO of the PCI IC for the given bus index.

- *busIndex* is the bus index; and
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *framePointer* parameter is NULL.	PCI 9080, PCI 9054
`ApiMuNotReady`	The Messaging Unit is not ready.	PCI 9080, PCI 9054
`ApiMuFifoEmpty`	The Inbound Post FIFO is empty.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 framePointer;

/* Read inbound port */
rc = PlxMuInboundPortRead(PrimaryPciBus, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuInboundPortWrite

Syntax:

```
RETURN_CODE PlxMuInboundPortWrite( IN BUS_INDEX busIndex,
                                   IN PU32 framePointer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Places a message frame to the Inbound Free FIFO of the PCI IC for the given bus index.

- *busIndex* is the bus index; and,
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *framePointer* parameter is NULL.	PCI 9080, PCI 9054
`ApiMuNotReady`	The Messaging Unit is not ready.	PCI 9080, PCI 9054
`ApiMuFifoEmpty`	The Inbound Post FIFO is empty.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 framePointer;

/* Write to the inbound port */
rc = PlxMuInboundPortWrite(PrimaryPciBus, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuInit

Syntax:

```
RETURN_CODE PlxMuInit( IN BUS_INDEX busIndex,
                       IN U32 fifoSize,
                       IN U64 iopAddr);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Configures the messaging unit's FIFO addresses of the PCI IC for the given bus index.

- *busIndex* is the bus index;
- *fifoSize* is the size of each FIFO in bytes; and
- *iopAddr* is the base address for the FIFOs. The base address must start on a 1MB boundary.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidAddress`	The *iopAddr* parameter is not aligned on a 1-MB boundary.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *fifoSize* parameter is invalid for this PCI IC.	PCI 9080, PCI 9054

Comments:

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

IC	Possible FIFO Sizes
PCI 9080	• 16 KBytes. Total size of the 4 FIFOs: 64 KBytes. • 32 KBytes. Total size of the 4 FIFOs: 128 KBytes. • 64 KBytes. Total size of the 4 FIFOs: 256 KBytes. • 128 KBytes. Total size of the 4 FIFOs: 512 KBytes. • 256 KBytes. Total size of the 4 FIFOs: 1 MBytes.

IC	Possible FIFO Sizes
PCI 9054	• 16 KBytes. Total size of the 4 FIFOs: 64 KBytes. • 32 KBytes. Total size of the 4 FIFOs: 128 KBytes. • 64 KBytes. Total size of the 4 FIFOs: 256 KBytes. • 128 KBytes. Total size of the 4 FIFOs: 512 KBytes. • 256 KBytes. Total size of the 4 FIFOs: 1 MBytes.

Usage:

```
#define FIFO_SIZE_16K     0x00004000

RETURN_CODE rc;


rc = PlxMuInit(PrimaryPciBus, FIFO_SIZE_16K, 0x200000);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuOutboundPortRead

Syntax:

```
RETURN_CODE PlxMuOutboundPortRead( IN BUS_INDEX busIndex,
                                   OUT PU32 framePointer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Returns a message frame from the Outbound Free FIFO of the PCI IC for the given bus index.

- *busIndex* is the bus index; and
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *framePointer* parameter is NULL.	PCI 9080, PCI 9054
`ApiMuNotReady`	The Messaging Unit is not ready.	PCI 9080, PCI 9054
`ApiMuFifoEmpty`	The Outbound Free FIFO is empty.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 framePointer;

/* Read outbound port */
rc = PlxMuOutboundPortRead(PrimaryPciBus, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuOutboundPortWrite

Syntax:

```
RETURN_CODE PlxMuOutboundPortWrite( IN BUS_INDEX busIndex,
                                    IN PU32 framePointer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Places a message frame to the Outbound Post FIFO of the PCI IC for the given bus index.

- *busIndex* is the bus index; and
- *framePointer* is the address of the Message Frame (MFA).

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *framePointer* parameter is NULL.	PCI 9080, PCI 9054
`ApiMuNotReady`	The Messaging Unit is not ready.	PCI 9080, PCI 9054
`ApiMuFifoEmpty`	The Outbound Free FIFO is empty.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U32 framePointer;

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

/* Write to the outbound port */
rc = PlxMuOutboundPortWrite(PrimaryPciBus, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxPciConfigRegisterRead

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads a configuration register from a PCI device.

- *bus* is the PCI bus number of the desired device;
- *slot* is the PCI slot number of the desired device;
- *registerNumber* is the desired configuration register; and,
- *data* is a pointer to a buffer to store the register value.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *registerNumber* parameter is out of range or not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiNullParam`	The *data* parameter is NULL.	PCI 9080, PCI 9054
`ApiConfigAccessFailed`	A PCI Bus Master Abort occurred.	PCI 9080, PCI 9054
`ApiFailed`	The API call failed indeterminately.	PCI 9054

Usage:

```
RETURN_CODE rc;                /* Return code from API functions */
U32 logbus, logdev, ven;

PlxPrintf("  bus:dev  VenID:DevID\n");
for (logbus = 0; logbus <MAX_PCI_BUS; logbus++)
{
  for (logdev = 0; logdev < MAX_PCI_DEV; logdev++)
```

```
  {
    rc = PlxPciConfigRegisterRead(logbus,
                                  logdev,
                                  PCI9080_VENDOR_ID,
                                  &ven);
    if ((rc == ApiSuccess) && (ven != 0xFFFFFFFF))
    {
        PlxPrintf("  %02x:0x%02x 0x%04x:0x%04x "
                  logbus, logdev,
                  ven & 0x0000ffff,
                  (ven & 0xffff0000) >> 16);
    }
  }
}
```

PlxPciConfigRegisterWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes data to a configuration register on a PCI device.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *registerNumber* parameter is out of range or not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiNullParam`	The *data* parameter is NULL.	PCI 9080, PCI 9054
`ApiConfigAccessFailed`	A PCI Bus Master Abort occurred.	PCI 9080, PCI 9054
`ApiFailed`	The API call failed indeterminately.	PCI 9054

Usage:

```
#define CFG_BAR0   0x10

RETURN_CODE rc;                /* Return code from API functions */
U32 address, range;

/* Save the address at BAR 0 */
rc = PlxPciConfigRegisterRead(0, 0x12, CFG_BAR0, &address);
```

```
if (rc == ApiSuccess)
{
  /* Get the range of BAR 0 */
  range = 0xFFFFFFFF;
  rc = PlxPciConfigRegisterWrite(0, 0x12, CFG_BAR0, &range);
  if (rc == ApiSuccess)
     PlxPciConfigRegisterRead(0, 0x12, CFG_BAR0, &range);

  /* Restore address to BAR 0 */
  PlxPciConfigRegisterWrite(0, 0x12, CFG_BAR0, &address);
}
```

PlxPowerLevelGet

Syntax:

```
PLX_POWER_LEVEL PlxPowerLevelGet( IN BUS_INDEX busIndex,
                                  OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9054

Description:

Gets the current power level of the PCI IC for the given bus index.

- *busIndex* is the bus index; and,
- *returnCode* is the return code of the function.

Return Value:

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

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiUnsupportedFunction`	This function is not supported by this PCI IC.	PCI 9080
`ApiInvalidPowerState`	The *powerLevel* state provided is invalid.	PCI 9054

Usage:

```
RETURN_CODE rc;
PLX_POWER_LEVEL plxPowerLevel;

/* Get the PCI IC's power level */
plxPowerLevel = PlxPowerLevelGet(PrimaryPciBus, &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxPowerLevelSet

Syntax:

```
RETURN_CODE PlxPowerLevelSet( IN BUS_INDEX busIndex,
                              IN PLX_POWER_LEVEL powerLevel);
```

PCI IC Support:

PCI 9054

Description:

Sets the power level of the PCI IC for the given bus index.

- *busIndex* is the bus index; and,
- *powerLevel* is the new power level.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiUnsupportedFunction`	This function is not supported by this PCI IC.	PCI 9080
`ApiInvalidPowerState`	The *powerLevel* state provided is invalid.	PCI 9054

Usage:

```
RETURN_CODE rc;

/* Set PCI IC to DO power level */
rc = PlxPowerLevelSet(PrimaryPciBus, D0);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxPrintf

Syntax:

`void PlxPrintf(PS8 ifmt,...);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Prints a formatted string of character to the serial port. This function works similarly to the *printf()* 'C' function.

Return Value:

None.

Usage:

```
RETURN_CODE rc;
PS8 sdkVersion;        /* pointer to the SDK Version string */
PS8 sdkReleaseDate;    /* pointer to the SDK Release Date string */


/* Print PCI SDK Version and Release Date */
rc = PlxSdkVersion(&sdkVersion, &sdkReleaseDate);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
else
  PlxPrintf("\n\n** %s, %s **\n", sdkVersion, sdkReleaseDate);
```

PlxRegisterDoorbellRead

Syntax:

```
U64 PlxRegisterDoorbellRead( IN BUS_INDEX busIndex,
                             OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads and clears the PCI to IOP doorbell register of the PCI IC on the given bus index.

- *busIndex* is the bus index; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

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

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080

Usage:

```
RETURN_CODE plxReturnStatus; /* Return code from API functions */
U32 value;

/* Local doorbell interrupt */
value = PlxRegisterDoorbellRead(PrimaryPciBus, &plxReturnStatus);
if (plxReturnStatus != ApiSuccess)
  PlxSdkError( (U32)plxReturnStatus, TRUE);
```

PlxRegisterDoorbellSet

Syntax:

```
RETURN_CODE PlxRegisterDoorbellSet( IN BUS_INDEX busIndex,
                                    IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to the IOP to PCI doorbell register of the PCI IC on the given bus index.

- *busIndex* is the bus index; and,
- *data* is a U64 value to store in the register.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080

Usage:

```
RETURN_CODE plxReturnStatus; /* Return code from API functions */
U64 value;

/* trigger a PCI doorbell interrupt */
value = 0x20;
plxReturnStatus = PlxRegisterDoorbellSet(PrimaryPciBus, value);
if (plxReturnStatus != ApiSuccess)
  PlxSdkError( (U32)plxReturnStatus, TRUE);
```

PlxRegisterMailboxRead

Syntax:

```
U64 PlxRegisterMailboxRead( IN BUS_INDEX busIndex,
                            IN MAILBOX_ID mailboxId,
                            OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads any mailbox register of the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *mailboxId* is the mailbox register ID; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

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

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *mailboxId* parameter is not a valid mailbox ID.	PCI 9080, PCI 9054
`ApiFailed`	The API call failed indeterminately.	PCI 9054

Usage:

```
RETURN_CODE rc;                /* Return code from API functions */
U64 mailboxHolder;             /* holder for Mailbox 0 */
/* Read Mailbox 0 */
mailboxHolder = PlxRegisterMailboxRead(PrimaryPciBus,
                                       MailBox0,
                                       &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
/* Write to Mailbox 1 */
rc = PlxRegisterMailboxWrite(PrimaryPciBus, MailBox1,
                             mailboxHolder);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxRegisterMailboxWrite

Syntax:

```
RETURN_CODE PlxRegisterMailboxWrite( IN BUS_INDEX busIndex,
                                     IN MAILBOX_ID mailboxId,
                                     IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to any mailbox register of the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *mailboxId* is the mailbox register ID; and,
- *data* is a U64 value to store in the register.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The Mailbox was written successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *mailboxId* parameter is not a valid mailbox ID.	PCI 9080, PCI 9054
`ApiFailed`	The API call failed indeterminately.	PCI 9054

Usage:

```
RETURN_CODE rc;                /* Return code from API functions */
U64 mailboxHolder;             /* holder for Mailbox 0 */

/* Read Mailbox 0 */
mailboxHolder = PlxRegisterMailboxRead(PrimaryPciBus,
                                       MailBox0,
                                       &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);

/* Write to Mailbox 1 */
rc = PlxRegisterMailboxWrite(PrimaryPciBus,
                             MailBox1,
                             mailboxHolder);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxRegisterRead

Syntax:

```
U64 PlxRegisterRead( IN BUS_INDEX busIndex,
                     IN U32 registerOffset,
                     OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads any register of the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *registerOffset* is the register number offset; and,
- *returnCode* is a pointer to a buffer to store the return code.

Note: *Use of this function creates application code that is not portable between PCI ICs because the PCI IC register set differs with each PCI IC. This function's use should be limited to special situations where the desired PCI IC feature or function can not be accessed by other IOP API functions.*

Return Value:

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

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiFailed`	The IOP API call failed catastrophically.	PCI 9054

Usage:

```
RETURN_CODE rc;                 /* Return code from API functions */
U32 value;                      /* temporary data holder */

/*
   The EEPROM is reloaded when bit 29 of the EEPROM_CTRL_STAT
   register goes from 0 to 1. So, first make sure it is 0.
*/
```

```
value = PlxRegisterRead(PrimaryPciBus,
                        PCI9080_EEPROM_CTRL_STAT,
                        &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);

/* set bit 28 to zero */
value &= ~(0x20000000);
rc = PlxRegisterWrite(PrimaryPciBus,
                      PCI9080_EEPROM_CTRL_STAT,
                      value);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxRegisterReadAll

Syntax:

```
RETURN_CODE PlxRegisterReadAll( IN BUS_INDEX busIndex,
                                IN U32 startOffset,
                                IN U32 registerCount,
                                OUT PU64 buffer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads multiple registers of the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *startOffset* is the register offset address to start reading at;
- *registerCount* is the number of bytes to read starting at *startOffset*; and,
- *buffer* is the storage location for the register values.

Note: registerCount *is the number of* ***bytes*** *to read.* buffer *must be large enough to store all the register values read. Use of this function creates application code that is not portable between PCI ICs because the PCI IC register set differs with each PCI IC.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *startOffset* and the *registerCount* parameters combined exceed the valid range of registers or *startOffset* is not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiNullParam`	The *buffer* parameter is NULL.	PCI 9080, PCI 9054
`ApiFailed`	The IOP API call failed catastrophically.	PCI 9054

Usage:

```
#define CONFIG_REGS_SIZE    0x40  /* Size in bytes of config regs */

RETURN_CODE rc;                 /* Return code from API functions */
U32 configRegs[CONFIG_REGS_SIZE / 4];  /* Store config regs */


rc = PlxRegisterReadAll(PrimaryPciBus,
                        PCI9080_VENDOR_ID,
                        CONFIG_REGS_SIZE,
                        &configRegs);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxRegisterWrite

Syntax:

```
RETURN_CODE PlxRegisterWrite( IN BUS_INDEX busIndex,
                              IN U32 registerOffset,
                              IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to any register of the PCI IC on the given bus index.

- *busIndex* is the bus index;
- *registerOffset* is the register offset address; and,
- *data* is a U64 value to store in the register.

Note: *Use of this function creates application code that is not portable between PCI ICs because the PCI IC register set differs with each PCI IC. This function's use should be limited to special situations where the desired PCI IC feature or function can not be accessed by other IOP API functions.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidRegister`	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiFailed`	The IOP API call failed catastrophically.	PCI 9054

Usage:

```
RETURN_CODE rc;                 /* Return code from API functions */
U32 value;                      /* temporary data holder */


/*
   The EEPROM is reloaded when bit 29 of the EEPROM_CTRL_STAT
   register goes from 0 to 1. So, first make sure it is 0.
*/
value = PlxRegisterRead(PrimaryPciBus,
                        PCI9080_EEPROM_CTRL_STAT,
```

```
                        &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);

/* set bit 28 to zero */
value &= ~(0x20000000);
rc = PlxRegisterWrite(PrimaryPciBus,
                      PCI9080_EEPROM_CTRL_STAT,
                      value);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxSdkVersion

Syntax:

```
RETURN_CODE PlxSdkVersion( S8 **versionString,
                           S8 **dateString);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Returns the version string of the PCI SDK IOP API.

- *versionString* is the version string for the IOP API; and,
- *dateString* is the version date string for the IOP API.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *versionString* or *dateString* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
PS8 sdkVersion;        /* pointer to the SDK Version string */
PS8 sdkReleaseDate;    /* pointer to the SDK Release Date string */


/* Print PCI SDK Version and Release Date */
PlxSdkVersion(&sdkVersion, &sdkReleaseDate);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
else
  PlxPrintf("\n\n** %s, %s **\n", sdkVersion, sdkReleaseDate);
```

PlxSerialEepromRead

Syntax:

```
RETURN_CODE PlxSerialEepromRead( IN BUS_INDEX busIndex,
                                 IN EEPROM_TYPE eepromType,
                                 OUT PU64 buffer,
                                 IN U32 size);
```

PCI IC Support:

PCI 9080

Note: Due to a PCI 9054-A errata, this API function is not available with the PCI 9054 device.

Description:

Reads values from the configuration EEPROM of the PCI IC for the given bus index.

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

Note: buffer *must be large enough to store all the data read from the EEPROM.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080
`ApiUnsupportedFunction`	This function is not supported by this PCI IC.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *buffer* parameter is NULL.	PCI 9080
`ApiEepromNotPresent`	There is no EEPROM connected to the PCI IC.	PCI 9080
`ApiEepromTypeNotSupported`	The *eepromType* parameter is not supported for this PCI IC.	PCI 9080
`ApiInvalidSize`	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.	PCI 9080

Usage:

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

/* Reading EEPROM into eepromData buffer */
rc = PlxSerialEepromRead(PrimaryPciBus,
                         Eeprom93CS56,
                         (U64 *)eepromData,
                         0x16 * sizeof(U16));  /* size in bytes */
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxSerialEepromWrite

Syntax:

```
RETURN_CODE PlxSerialEepromWrite( IN BUS_INDEX busIndex,
                                  IN EEPROM_TYPE eepromType,
                                  IN PU64 buffer,
                                  IN U32 size);
```

PCI IC Support:

PCI 9080

Note: Due to a PCI 9054-A errata, this API function is not available with the PCI 9054 device.

Description:

Writes values to the configuration EEPROM of the PCI IC for the given bus index.

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

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080
`ApiUnsupportedFunction`	This function is not supported by this PCI IC.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiNullParam`	The *buffer* parameter is NULL.	PCI 9080
`ApiEepromNotPresent`	There is no EEPROM connected to the PCI IC.	PCI 9080
`ApiEepromTypeNotSupported`	The *eepromType* parameter is not supported for this PCI IC.	PCI 9080
`ApiInvalidSize`	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.	PCI 9080

Usage:

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


/* Initialize the EEPROM buffer */
{...};

/* Writing to EEPROM from eepromData buffer */
rc = PlxSerialEepromWrite(PrimaryPciBus,
                          Eeprom93CS56,
                          (U64 *)eepromData,
                          0x16 * sizeof(U16));  /* size in bytes */
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxUserRead

Syntax:

```
PIN_STATE PlxUserRead( IN BUS_INDEX busIndex,
                       IN USER_PIN userPin,
                       OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads the USERI pins of the PCI IC for the given bus index.

- *busIndex* is the bus index;
- *userPin* is the USERI pin number to be read; and,
- *returnCode* is a pointer to a buffer to store the return code.

Return Value:

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

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidUserPin`	This *userPin* is not present on this PCI IC.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
PIN_STATE pinState;

/* Writing to EEPROM from eepromData buffer */
pinState = PlxUserRead(PrimaryPciBus,
                       USER0,
                       &rc);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxUserWrite

Syntax:

```
RETURN_CODE PlxUserWrite( IN BUS_INDEX busIndex,
                          IN USER_PIN userPin,
                          IN PIN_STATE pinState);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes to USERo pins of the PCI IC for the given bus index, setting them either `Active` or `Inactive`.

- *busIndex* is the bus index;
- *userPin* is the USERo pin number to be written; and,
- *pinState* is the new state to set the USERO pin.

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9080, PCI 9054
`ApiNotInitialized`	The IOP API is not initialized.	PCI 9080
`ApiInvalidUserPin`	This *userPin* is not present on this PCI IC.	PCI 9080, PCI 9054
`ApiInvalidUserState`	The *pinState* is invalid.	PCI 9080, PCI 9054

Usage:

```
U32 blinkTimes = 2;          /* blink twice */
U32 blinkSpeed = 0x4000;     /* blink fast  */
U32 loopCounter;
VU32 value;

/* Blink the LED  */
for (loopCounter = 0;
     loopCounter < blinkTimes;
     loopCounter++)
{
  /* Turn ON */
  value = 0;
  while (value < blinkSpeed )
```

```
    value ++;
  PlxUserWrite(PrimaryPciBus, USER0, Active);


  /* Turn OFF */
  value = 0;
  while (value < blinkSpeed )
    value ++;
  PlxUserWrite(PrimaryPciBus, USER0, Inactive);
}
```

PlxVpdDisable

Syntax:

`RETURN_CODE PlxVpdDisable( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Disables PCI access to the VPD capability of the chip.

- *busIndex* is the bus index.

***Note**: VPD is enabled by default. Enabling or disabling VPD after the Init done bit is set can cause problems with the host Operating System.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054

Usage:

```
RETURN_CODE rc;

/* Disable Vital Product Data  */
rc = PlxVpdDisable(PrimaryPciBus);

if (rc != ApiSuccess)
  PlxSdkError(rc, TRUE);
```

PlxVpdEnable

Syntax:

`RETURN_CODE PlxVpdEnable( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Enables PCI access to the VPD capability of the chip.

- *busIndex* is the bus index.

***Note**: VPD is enabled by default. Enabling or disabling VPD after the Init done bit is set can cause problems with the host Operating System.*

Return Value:

Return Value	Description	API Support
`ApiSuccess`	The function returned successfully.	PCI 9054
`ApiInvalidBusIndex`	The *busIndex* value is not valid.	PCI 9054

Usage:

```
RETURN_CODE rc;

/* Enable Vital Product Data */
rc = PlxVpdEnable(PrimaryPciBus);

if (rc != ApiSuccess)
  PlxSdkError(rc, TRUE);
```

PlxVpdId

Syntax:

`U8 PlxVpdId(IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Read the VPD ID register.

- *busIndex* is the bus index.

Return Value:

Return Value	Description	API Support
`0x03`	The function read successfully the VPD ID.	PCI 9054
`0xFF`	VPD is not supported or not present on this board.	PCI 9054

Usage:

```
U8 id;

/* Print the Vital Product Data ID */
id = PlxVpdId(PrimaryPciBus);
if (id == 0xFF)
  PlxPrintf("VPD is not supported\n");
else
  PlxPrintf("VPD ID is 0x%02x\n", id);
```

PlxVpdNextItemPointer

Syntax:

`U8 PlxVpdNextItemPointer( IN BUS_INDEX busIndex);`

PCI IC Support:

PCI 9054

Description:

Returns the VPD Next Item Pointer Register value.

- *busIndex* is the bus index.

Return Value:

Return Value	Description	API Support
0xFF	VPD is not supported or not present on this board.	PCI 9054
0x00	There is no more new capability.	PCI 9054
0xXX	Next item pointer register value.	PCI 9054

Usage:

```
U8 value;

/* Get the next item pointer register  */
value = PlxVpdNextItemPointer(PrimaryPciBus);
if (value == 0)
  PlxPrintf("There is no more new capability\n");
else if (value == 0xFF)
  PlxPrintf("VPD is not supported on this board\n");
else
  PlxPrintf("The next item pointer register is 0x%02x\n", value);
```

4. PCI API Function Description

The PCI API is designed around the features of the PCI IC.

4.1 PCI API Function Quick Reference List

The following table lists all the PCI API functions available. Designers should consult Section 4.2 for detailed descriptions of the API functions.

API Function Name	Purpose
PlxBusIopRead()	Read data from the IOP bus.
PlxBusIopWrite()	Write data to the IOP bus.
PlxDmaBlockChannelClose()	Close a DMA channel for Block DMA.
PlxDmaBlockChannelOpen()	Open a DMA channel for Block DMA.
PlxDmaBlockTransfer()	Control DMA Block transfer.
PlxDmaBlockTransferRestart()	Restart a previous DMA Block transfer.
PlxDmaSglChannelClose()	Close a DMA channel for SGL DMA.
PlxDmaSglChannelOpen()	Open a DMA channel for Scatter-Gather DMA.
PlxDmaSglTransfer()	Control a SGL transfer.
PlxDmaShuttleChannelClose()	Close a DMA channel for Shuttle DMA.
PlxDmaShuttleChannelOpen()	Open a DMA channel for Shuttle DMA.
PlxDmaShuttleTransfer()	Control a Shuttle DMA transfer.
PlxIntrAttach()	Attach a wait event to a PLX interrupt.
PlxIntrDisable()	Disable specific PLX interrupts.
PlxIntrEnable()	Enable specific PLX interrupts.
PlxIntrStatusGet()	Get the status of the PLX interrupts.
PlxIoPortRead()	Read data from an I/O port.
PlxIoPortWrite()	Write data to an I/O port.
PlxMuInboundPortRead()	Read the Inbound Port.
PlxMuInboundPortWrite()	Write to the Inbound Port.
PlxMuOutboundPortRead()	Read the Outbound Port.
PlxMuOutboundPortWrite()	Write to the Outbound Port.
PlxPciBarRangeGet()	Get the memory range for a base address register.
PlxPciBaseAddressesGet()	Get the virtual addresses for the base address registers.
PlxPciBusSearch()	Search for a PLX device on the PCI bus.

API Function Name	Purpose
PlxPciCommonBufferGet()	Get the information for the device driver's common buffer.
PlxPciConfigRegisterRead()	Read a configuration register of a PCI device.
PlxPciConfigRegisterReadAll()	Read all the Configuration registers of a PCI device.
PlxPciConfigRegisterWrite()	Write to a Configuration register of a PCI device.
PlxPciDeviceClose()	Close a PLX device channel.
PlxPciDeviceFind()	Find a PLX device on the PCI bus.
PlxPciDeviceOpen()	Open a PLX device channel.
PlxPowerLevelGet()	Get the power level.
PlxPowerLevelSet()	Set the power level.
PlxRegisterDoorbellRead()	Read from then clear a Doorbell register.
PlxRegisterDoorbellSet()	Set a Doorbell register.
PlxRegisterMailboxRead()	Read a Mailbox register.
PlxRegisterMailboxWrite()	Write to a Mailbox register.
PlxRegisterRead()	Read a register.
PlxRegisterReadAll()	Read a register group.
PlxRegisterWrite()	Write to a register.
PlxSdkVersion	Get the version and date strings for the PCI API.
PlxSerialEepromRead()	Read the serial EEPROM.
PlxSerialEepromWrite()	Write to the serial EEPROM.
PlxUserRead()	Read a USER pin value.
PlxUserWrite()	Write a value to a USER pin.

4.2 PCI API Functions Details

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

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

Note: *Devices supported by PCI SDK Version 2.1: PCI 9080 and PCI 9054.*

Sample Function Entry

Syntax:

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

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

PCI IC Support:

A list of PCI ICs that support this function. For this version, both the PCI 9080 and the PCI 9054 support all the PCI API functions.

Description:

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

Return Value:

The value returned by the function.

Usage:

A sample is provided to demonstrate the function's use.

PlxBusIopRead

Syntax:

```
RETURN_CODE PlxBusIopRead( IN HANDLE drvHandle,
                           IN IOP_SPACE iopSpace,
                           IN U64 address,
                           IN BOOLEAN remapAddress,
                           OUT PU64 destination,
                           IN U32 transferSize,
                           IN ACCESS_TYPE accessType);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

- *drvHandle* is the handle of the PCI device;
- *iopSpace* defines which Local Address Space register is used;
- *address* is the starting offset from the IOP space PCI remap address (*remapAddress*=TRUE) or the actual IOP bus address to start reading from (*remapAddress*=FALSE);
- *remapAddress* states how to treat the IOP address given. FALSE means that the IOP address is an offset from the IOP space PCI remap address; TRUE means the the IOP address is the actual IOP bus address;
- *destination* is a pointer to the storage buffer for the data read;
- *transferSize* defines the number of bytes to read from the IOP bus; and,
- *accessType* defines the access type size.

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). If* remapAddress *is set to* `TRUE`*, this API function will remap the IOP space window according to the* address *given.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *destination* parameter is NULL.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this device.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *address* parameter is not aligned	PCI 9080,

Return Value	Description	IC Support
	based on the *accessType* provided.	PCI 9054
`ApiInvalidSize`	The *transferSize* parameter is 0 or is not aligned based on the *accessType* provided.	PCI 9080, PCI 9054

Comments:

This function should be used with small buffers. For transfers exceeding 4 KBytes, it is recommended that DMA transfers be used instead to optimize the performance of the host system.

User's of the CompactPCI 9054RDK-860 must *disable* BTERM prior to using this API function with SDRAM. However, you must *enable* BTERM prior to using this API function with SBSRAM. Please consult the CompactPCI 9054RDK-860 Hardware Reference Manual for more details on direct slave functionality.

Note: *This function monopolizes CPU time and memory and thereby hinders the performance of the host system.*

Usage:

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

rc = PlxBusIopRead(drvHandle,
                   IopSpace0,
                   localStartOffset,
                   TRUE,       /* remap */
                   (PU64)buf,
                   length,
                   BitSize32);
if (rc != ApiSuccess)
{
  printf("Error: Unable to read data.\n");
  return -1;
}
```

PlxBusIopWrite

Syntax:

```
RETURN_CODE PlxBusIopWrite( IN HANDLE drvHandle,
                            IN IOP_SPACE iopSpace,
                            IN U64 address,
                            IN BOOLEAN remapAddress,
                            IN PU64 source,
                            IN U32 transferSize,
                            IN ACCESS_TYPE accessType);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

- *drvHandle* is the handle of the PCI device;
- *iopSpace* defines which Local Address Space register to used;
- *address* is the starting offset from the IOP space PCI remap address or the actual IOP bus address to start writing to;
- *remapAddress* states how to treat the IOP address given. FALSE means that the IOP address is an offset from the IOP space PCI remap address; TRUE means the the IOP address is the actual IOP bus address;
- *source* is a pointer to the data buffer;
- *transferSize* defines the number of bytes to write to the IOP bus; and,
- *accessType* defines the access type size.

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). If* remapAddress *is set to* `TRUE` *this API function will remap the IOP space window according to the* address *given.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *source* parameter is NULL.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this device.	PCI 9080, PCI 9054
`ApiInvalidAddress`	The *address* parameter is not aligned	PCI 9080,

Return Value	Description	IC Support
	based on the *accessType* provided.	PCI 9054
`ApiInvalidSize`	The *transferSize* parameter is 0 or is not aligned based on the *accessType* provided.	PCI 9080, PCI 9054

Comments:

This function should be used with small buffers. For transfers exceeding 4 KBytes, it is recommended that DMA transfers be used instead to optimize the performance of the host system.

User's of the CompactPCI 9054RDK-860 must *disable* BTERM prior to using this API function with SDRAM. However, you must *enable* BTERM prior to using this API function with SBSRAM. Please consult the CompactPCI 9054RDK-860 Hardware Reference Manual for more details on direct slave functionality.

Note: *This function monopolizes CPU time and memory and thereby hinders the performance of the host system.*

Usage:

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

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

/*
  Now clear an IOP buffer starting at localStartOffset
  over a length of 0x100.
*/
rc = PlxBusIopWrite(drvHandle,
                    IopSpace0,
                    localStartOffset,
                    TRUE,
                    (PU64)buf,
                    length,
                    BitSize32);

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

PlxDmaBlockChannelClose

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes the Block DMA channel.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;
if ((rc= PlxDmaBlockChannelClose(drvHandle, PrimaryPciChannel0))
    != ApiSuccess)
{
  printf ("\n\nErrors in closing channel.\n");
  printf ("Returned code %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaBlockChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Opens and initializes a DMA channel for Block DMA transfers.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). If the* dmaChannelDesc *parameter is NULL the function uses the current setting for the channel.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

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

/* Setup DMA configuration structure */
desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
desc.DmaStopTransferMode = AssertBLAST;
```

```
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
desc.HoldIopDestAddrConst = 0;
desc.DemandMode = 0;
desc.EnableTransferCountClear = 0;
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

if ((rc=PlxDmaBlockChannelOpen(drvHandle,
                               PrimaryPciChannel0,
                               &desc))
       != ApiSuccess)
{
  printf ("\n\nErrors in opening channel.\n");
  printf ("Returned code %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaBlockTransfer

Syntax:

```
RETURN_CODE PlxDmaBlockTransfer( IN HANDLE drvHandle,
                                 IN DMA_CHANNEL dmaChannel,
                                 IN DMA_COMMAND dmaCommand,
                                 IN PDMA_TRANSFER_ELEMENT dmaData,
                                 IN BOOLEAN returnImmediate);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Controls the Block DMA transfer for a given DMA channel.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *dmaData* (if *dmaCommand* is not `DmaStart`) parameters are NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054

Return Value	Description	IC Support
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused).	PCI 9080, PCI 9054
`ApiDmaManCorrupted`	The DMA manager is corrupted.	PCI 9080, PCI 9054
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054

Comments:

Block DMA transfers are useful with contiguous host buffers described by a *PCI address*. Therefore, it is incorrect to allocate a buffer (using malloc(), at compile time or in the stack) and use its address with the Block DMA functions. Instead, users should use a known PCI address (not *user address*) of, for example, the common buffer or an other device known PCI base address.

When a Block DMA transfer is done, it may generate an interrupt to the PCI which will be handled by the driver.

The table below describes the exact action that is taken for each possible value of the *dmaCommand* parameter.

Action (*dmaCommand* parameter)	Description
DmaStart	Program and start the DMA transfer. If *returnImmediate* is FALSE, then the API function will wait until the transfer is done.
DmaPause	Pause a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaResume	Resume a paused DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaAbort	Abort a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaStatus	Gives the status of a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.

The DMA_TRANSFER_ELEMENT structure contains members whose signification may differ depending on the DMA transfer type. Also, some other members are ignored by the API call. Here is the structure:

Structure Element	Signification
UserAddr	Ignored.
LowPciAddr	Low PCI address of the host buffer.
HighPciAddr	(Pci9054 only) High PCI address of the host buffer.
IopAddr	The IOP address for the transfer.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.
LastSglElement	Ignored.

Structure Element	Signification
TerminalCountIntr	Indicates if an interrupt is to be generated when the DMA transfer is done. A 1 means yes.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

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

rc = PlxPciCommonBufferGet(plxHandle, &PciMemory);
if (  rc != ApiSuccess )
{
  PlxSdkError((U32) rc );
  printf("\a\nPlxPciCommonBufferGet tested: FAILED.");
}

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

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

PlxDmaBlockTransferRestart

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Block DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054

Usage:

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

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

PlxDmaSglChannelClose

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes the Scatter-Gather DMA channel.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Scatter-Gather DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

if ((rc= PlxDmaSglChannelClose(drvHandle, PrimaryPciChannel0))
    != ApiSuccess)
{
  printf ("\n\nErrors in closing channel.\n");
  printf ("Returned code %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaSglChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). If the* dmaChannelDesc *parameter is NULL the function uses the last setting for the channel. Scatter-Gather DMA needs the DMA interrupts enabled for proper operation and thus this function enables the corresponding PCI DMA interrupt.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

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

/* Setup DMA configuration structure */
desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
```

```
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
desc.DmaStopTransferMode = AssertBLAST;
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
desc.HoldIopDestAddrConst = 0;
desc.DemandMode = 0;
desc.EnableTransferCountClear = 0;
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

if ((rc=PlxDmaSglChannelOpen(drvHandle,
                             PrimaryPciChannel0,
                             &desc))
       != ApiSuccess)
{
  printf ("\n\nErrors in opening channel.\n");
  printf ("Returned code %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaSglTransfer

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *dmaData* (if *dmaCommand* is `DmaStart`) parameters are NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for SGL DMA.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054

Return Value	Description	IC Support
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested and the DMA channel is not paused).	PCI 9080, PCI 9054
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054
`ApiFailed`	Failed to start the DMA transfer. ***Note:*** *Ensure that there is enough memory to build the Scatter-Gather List. Memory for the Scatter-Gather List can be added by increasing the* `MaxSglTransferSize` *registry entry.*	PCI 9080, PCI 9054

Comments:

Scatter-Gather transfers are useful with host buffers described by a *user address*.

When a SGL transfer is done, it will generate an interrupt to the PCI which will be handled by the driver.

The table below describes the exact action that is taken for each possible value of the *dmaCommand* parameter.

Action (*dmaCommand* parameter)	Description
DmaStart	Program and start the DMA transfer. If *returnImmediate* is FALSE, then the API function will wait until the transfer is done.
DmaPause	Pause a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaResume	Resume a paused DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaAbort	Abort a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.
DmaStatus	Gives the status of a DMA transfer. The *dmaData* and *returnImmediate* parameters are ignored.

The DMA_TRANSFER_ELEMENT structure contains members whose signification may differ depending on the DMA transfer type. Also, some other members are ignored by the API call. Here is the structure:

Structure Element	Signification
UserAddr	User address of the host buffer.
LowPciAddr	Ignored
HighPciAddr	Ignored
IopAddr	The IOP address for the transfer.
TransferCount	The number of bytes for the transfer.
PciSglLoc	Ignored.

Structure Element	Signification
LastSglElement	Ignored.
TerminalCountIntr	Ignored.
IopToPciDma	Direction of the transfer. A 1 means an IOP-to-PCI transfer.
NextSglPtr	Ignored.

Usage:

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

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

if((rc=PlxDmaSglTransfer(plxHandle,
                         PrimaryPciChannel0,
                         DmaStart,
                         &dmaData,
                         FALSE /* return upon completion */))
   != ApiSuccess)
{
  printf ("\n\nErrors in Sgl Dma.\n");
  printf ("Returned code %d\n",rc);
  getch();
  return -1;
}
```

PlxDmaShuttleChannelClose

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes the Shuttle DMA channel.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Shuttle DMA.	PCI 9080, PCI 9054
`ApiDmaInProgress`	A DMA transfer is in progress.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

if ((rc= PlxDmaShuttleChannelClose(drvHandle, PrimaryPciChannel0))
    != ApiSuccess)
{
  printf ("\n\nErrors in closing channel.\nReturned code %d",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaShuttleChannelOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Opens and initializes a DMA channel for Shuttle DMA transfers.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). If the* dmaChannelDesc *parameter is NULL the function uses the current setting for the channel.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	This *dmaChannel* parameter is not supported by this PLX IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel is not closed.	PCI 9080, PCI 9054
`ApiDmaInvalidChannelPriority`	The `DmaChannelPriority` member of *dmaChannelDesc* is not valid.	PCI 9080, PCI 9054

Usage:

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

/* Setup DMA configuration structure */
desc.EnableReadyInput = 1;
desc.EnableBTERMInput = 0;
desc.EnableIopBurst = 0;
desc.EnableWriteInvalidMode = 0;
desc.EnableDmaEOTPin = 0;
desc.DmaStopTransferMode = AssertBLAST;
```

```
desc.HoldIopAddrConst = 0;
desc.HoldIopSourceAddrConst = 0;
desc.HoldIopDestAddrConst = 0;
desc.DemandMode = 0;
desc.EnableTransferCountClear = 0;
desc.DmaChannelPriority = Rotational; /* rotational priority */
desc.WaitStates = 0;
desc.IopBusWidth = 3; /* 32 bit bus */
desc.Reserved1 = 0;
desc.TholdForIopWrites = 0;
desc.TholdForIopReads = 0;
desc.TholdForPciWrites = 0;
desc.TholdForPciReads = 0;
desc.Reserved2 = 0;

if ((rc=PlxDmaShuttleChannelOpen(drvHandle,
                            PrimaryPciChannel0,
                            &desc))
       != ApiSuccess)
{
  printf ("\n\nErrors in opening channel.\n");
  printf ("Returned code %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxDmaShuttleTransfer

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Controls a Shuttle DMA transfer for a given DMA channel.

- *drvHandle* is the handle of the PCI device;
- *dmaChannel* is the DMA channel number previously opened;
- *dmaCommand* is the action to perform on this DMA channel; and,
- *dmaData* is the data for the DMA transfer (see the table in the comments section).

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiDmaChannelInvalid`	The *dmaChannel* parameter is not supported by this PCI IC.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *dmaData* (if *dmaCommand* is not `DmaStart`) parameters are NULL.	PCI 9080, PCI 9054
`ApiDmaChannelUnavailable`	The DMA channel was not opened for Shuttle DMA.	PCI 9080, PCI 9054
`ApiDmaDone`	The DMA channel is done.	PCI 9080, PCI 9054
`ApiDmaPaused`	The DMA channel is paused.	PCI 9080, PCI 9054
`ApiDmaInProgress`	The DMA channel is in progress.	PCI 9080, PCI 9054
`ApiDmaNotPaused`	The DMA channel is in progress or done (return code returned when `DmaResume` command is requested	PCI 9080, PCI 9054

Return Value	Description	IC Support
	and the DMA channel is not paused).	
`ApiFailed`	Failed to start the DMA transfer. ***Note:*** *Ensure that there is enough memory to build the Scatter-Gather List. Memory for the Scatter-Gather List can be added by increasing the* `MaxSglTransferSize` *registry entry.*	PCI 9080, PCI 9054
`ApiDmaCommandInvalid`	The *dmaCommand* parameter is invalid.	PCI 9080, PCI 9054

Comments:

Shuttle transfers are useful with host buffers described by a *user address*.

Because shuttle transfers are perpetual, no interrupts are generated by the Shuttle DMA functions. If an interrupt would occur, it would be perpetual as well and may hang the host OS.

The table below describes the exact action that is taken for each possible value of the *dmaCommand* parameter.

Action (*dmaCommand* parameter)	Description
DmaStart	Program and start the DMA transfer. From this point, the DMA transfer occur perpetually.
DmaPause	Pause a Shuttle DMA transfer. The *dmaData* parameter is ignored.
DmaResume	Resume a paused DMA transfer. The *dmaData* parameter is ignored.
DmaAbort	Stop a Shuttle DMA transfer. The *dmaData* parameter is ignored.
DmaStatus	Gives the status of a DMA transfer. The *dmaData* parameter is ignored.

The DMA_TRANSFER_ELEMENT structure contains members whose signification may differ depending on the DMA transfer type. Also, some other members are ignored by the API call. Here is the structure:

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

Usage:

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

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

  if((rc=PlxDmaShuttleTransfer(plxHandle,
                               PrimaryPciChannel0,
                               DmaStart,
                               &dmaData));
!= ApiSuccess)
  {
    printf ("\n\nErrors in Shuttle Dma.\n");
    printf ("Returned code %d\n",rc);
    getch();
    return -1;
  }
```

PlxIntrAttach

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *pEventHdl* parameter is NULL.	PCI 9080, PCI 9054
`ApiUnsupportedFunction`	The PCI API function call is not supported by the current device.	PCI 9080, PCI 9054
`ApiInsufficientResources`	The number of attached events has been exceeded.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
PLX_INTR intfield;
HANDLE eventHandle;

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

/* Attach event on the DMA channel 1 interrupts */
intfield.PciDmaChannel1   = 1;
```

```
if ((rc=PlxIntrAttach(myPlxDevice,
                      intfield,
                      &eventHandle)) != ApiSuccess)
{
  printf ("\n\nErrors in attaching interrupt.\n");
  printf ("Returned code %d\n",rc);
  getch();
  exit(1);
}

/* Enable DMA channel 1 interrupt */
if((rc=PlxIntrEnable(myPlxDevice, &intfield)) != ApiSuccess)
{
  printf ("\n\nErrors in enabling interrupt.\n");
  printf ("Returned code %d\n",rc);
  getch();
  exit(1);
}

/* Wait for signal at end of transfer. */
val = WaitForSingleObject(eventHandle, 5000 /* msec */ );
if (val == WAIT_TIMEOUT)
  printf("\n5 seconds timeout exceeded...");
else
  printf("\nReceived the interrupt");
```

PlxIntrDisable

Syntax:

```
RETURN_CODE PlxIntrDisable( IN HANDLE drvHandle,
                            IN PPLX_INTR plxIntr);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Disables specific interrupts of a PCI device.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *plxIntr* parameter is NULL.	PCI 9080, PCI 9054

Usage:

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

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

rc = PlxIntrDisable(plxHandle, plxIntrStatus);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxIntrDisable failed with error code %08lx.",
         rc);
  PlxSdkError( (U32) rc);
}
else
  printf("\nPlxIntrDisable tested: OK.");
```

PlxIntrEnable

Syntax:

```
RETURN_CODE PlxIntrEnable( IN HANDLE drvHandle,
                           IN PPLX_INTR plxIntr);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Enables specific interrupts of a PCI device.

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

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

Return Value:

Return Value	Description
`ApiSuccess`	The function returned successfully.
`ApiNullParam`	The *drvHandle* or the *plxIntr* parameter is NULL.

Usage:

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

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

rc = PlxIntrEnable(plxHandle, &plxIntrStatus);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxIntrEnable failed with error code %08lx.",
         rc);
  PlxSdkError( (U32) rc);
}
else
  printf("\nPlxIntrEnable tested: OK.");
```

PlxIntrStatusGet

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Returns the interrupts of the PCI device are were last active. The interrupts are cleared once they are read.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *plxIntr* parameter is NULL.	PCI 9080, PCI 9054

Usage:

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

PlxIoPortRead

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads a value from an I/O port.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *pOutData* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this device.	PCI 9080, PCI 9054

Usage:

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

PlxIoPortWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to an I/O port.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *pValue* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidAccessType`	The *accessType* size is not supported for this device.	PCI 9080, PCI 9054

Usage:

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

PlxMuInboundPortRead

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads the Inbound Port and gets a Free Message Frame.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen() and the Messaging Unit must be initialized (initialized by the IOP application).*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *framePointer* parameters are NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

/* Read inbound port */
rc = PlxMuInboundPortRead(drvHandle, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuInboundPortWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes to the Inbound Port with a posted message frame.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen() and the Messaging Unit must be initialized (initialized by the IOP application).*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *framePointer* parameters are NULL.	PCI 9080, PCI 9054

Usage:

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

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

/* Write to the inbound port */
rc = PlxMuInboundPortWrite(drvHandle, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuOutboundPortRead

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads the Outbound Port and gets a posted message frame.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen() and the Messaging Unit must be initialized (initialized by the IOP application).*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *framePointer* parameters are NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

/* Read outbound port */
rc = PlxMuOutboundPortRead(drvHandle, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxMuOutboundPortWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes to the Outbound Port with a free message frame.

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

Note: *Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen() and the Messaging Unit must be initialized (initialized by the IOP application).*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *framePointer* parameters are NULL.	PCI 9080, PCI 9054

Usage:

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

/* Write to the outbound port */
rc = PlxMuOutboundPortWrite(drvHandle, &framePointer);
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxPciBarRangeGet

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Retrieves the range of any PCI base address register.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *data* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *registerNumber* parameter is out of range or does not contain a range.	PCI 9080, PCI 9054

Usage:

```
#define GETBARRANGE_MAX             6

RETURN_CODE rc;                       /* Return code */
HANDLE plxHandle;                     /* Device Handle */
U32 BarRanges[GETBARRANGE_MAX];

for ( i = 0; i < GETBARRANGE_MAX; i ++ )
{
  rc = PlxPciBarRangeGet(plxHandle, i, & BarRanges[i]);
  if ( rc != ApiSuccess )
  {
    PlxSdkError((U32) rc );
    printf("\a\nFailed to get bar range for %08lx", i);
  }
  else
    printf("\nBar range for %d is %08lX", i, BarRanges[i]);
}
```

PlxPciBaseAddressesGet

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Comments

The virtual address structure contains all the user virtual addresses for the various PCI base addresses available to access the PCI device. This structure is filled by calling the PlxPciBaseAddressesGet() function.

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *virtAddr* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;                        /* Return code */
HANDLE plxHandle;                      /* Device Handle */
VIRTUAL_ADDRESSES virtualAddresses;
rc = PlxPciBaseAddressesGet(plxHandle, &virtualAddresses);
if ( rc != ApiSuccess ) {
  printf("\a\nFailed to get virtual address with error code %08lX",
         rc);
  exit(0);
} else {
  printf("\n    U32 Va1    is %08lX", virtualAddresses.Va1);
  printf("\n    U32 Va2    is %08lX", virtualAddresses.Va2);
  printf("\n    U32 Va3    is %08lX", virtualAddresses.Va3);
  printf("\n    U32 Va4    is %08lX", virtualAddresses.Va4);
  printf("\n    U32 Va5    is %08lX", virtualAddresses.Va5);
}
```

PlxPciBusSearch

Syntax:

`RETURN_CODE PlxPciBusSearch( IN OUT PDEVICE_LOCATION pDevData);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

Note: *This function is here to support existing applications only. Use PlxPciDeviceFind() with the requestLimit parameter set to 0 instead .*

Return Value:

On success, *pDevData* contains valid device information.

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *pDevData* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidDeviceInfo`	The information in *pDevData* is not valid for any device in the system.	PCI 9080, PCI 9054
`ApiNoActiveDriver`	There is no device driver installed into the system.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054

Usage:

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

device.VendorId = PLX_VENDOR_ID;
device.DeviceId = PLX_9080RDK_860_DEVICE_ID;
device.BusNumber = 0xFFFFFFFF;
device.SlotNumber = 0xFFFFFFFF;
device.SerialNumber[0] = '\0';
rc = PlxPciBusSearch(&device);
if ( rc != ApiSuccess )
{
```

```
  printf("\a\nPlxPciBusSearch failed.");
  PlxSdkError((U32) rc);
}
else
{
  printf("\nU32 DeviceId   is %08lX",    device.DeviceId);
  printf("\nU32 VendorId   is %08lX",    device.VendorId);
  printf("\nU32 BusNumber  is %08lX",    device.BusNumber);
  printf("\nU32 SlotNumber is %08lX",    device.SlotNumber);
  printf("\nU8 SerialNumber [16] is %s", device.SerialNumber);
}
```

PlxPciCommonBufferGet

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Note about the Physical Memory Buffer:

The host device drivers allocate a physical buffer in PCI space. This buffer is meant to be used by host and IOP applications. PlxPciCommonBufferGet() provides the physical address, the virtual address and the size of the physical memory buffer.

The physical address is meant to be used by the PCI IC when you require a PCI buffer. The virtual address is meant to be used by the host applications to access the PCI buffer.

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *pMemoryInfo* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
HANDLE plxHandle;
PCI_MEMORY PciMemory;
rc = PlxPciCommonBufferGet(plxHandle, &PciMemory);
if (  rc != ApiSuccess ) {
  PlxSdkError((U32) rc );
  printf("\a\nPlxPciCommonBufferGet tested: FAILED.");
} else {
  printf("\nCommon Buffer User address is %08lX",
         PciMemory.UserAddr);
  printf("\nCommon Buffer PCI physical address is %08lX",
         PciMemory.PhysicalAddr);
  printf("\nCommon Buffer Size is %08lX", PciMemory.Size);
}
```

PlxPciConfigRegisterRead

Syntax:

```
U64 PlxPciConfigRegisterRead( IN U32 bus,
                              IN U32 slot,
                              IN U32 registerNumber,
                              OUT PPRETURN_CODE pReturnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads a configuration register from a PCI device.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *registerNumber* parameter is out of range or not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiConfigAccessFailed`	The device described by the *bus* and *slot* parameter is not present.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U64 range, address;

/* Save a copy of the address */
address =  PlxPciConfigRegisterRead(0x0,
                                    0x12,
                                    PCI9080_LOCAL_BASE0,
                                    &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  PlxSdkError( (U32) rc);
```

```
  exit(2);
}

/* Get the BAR range */
range = 0xFFFFFFFF;
rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &range);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}

range =  PlxPciConfigRegisterRead(0x0,
                                  0x12,
                                  PCI9080_LOCAL_BASE0,
                                  &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}

rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &address);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}
```

PlxPciConfigRegisterReadAll

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads all PCI Configuration registers on a PCI device.

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

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

IC	Required size (bytes)
PCI 9080	64
PCI 9054	84

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *buffer* parameter is NULL.	PCI 9080, PCI 9054
`ApiConfigAccessFailed`	The device described by the *bus* and *slot* parameter is not present.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054

Usage:

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

rc = PlxPciConfigRegisterReadAll(0x0,
                                 0x12,
                                 pciRegsBuffer);

if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterReadAll failed.");
  PlxSdkError( (U32) rc);
  exit(1);
}
```

PlxPciConfigRegisterWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes data to a configuration register on a PCI device.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *registerNumber* parameter is out of range or not on a 4-byte boundary.	PCI 9080, PCI 9054
`ApiNullParam`	The *data* parameter is NULL.	PCI 9080, PCI 9054
`ApiConfigAccessFailed`	The device described by the *bus* and *slot* parameter is not present.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
U64 range, address;

/* Save a copy of the address */
address =  PlxPciConfigRegisterRead(0x0,
                                    0x12,
                                    PCI9080_LOCAL_BASE0,
                                    &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  PlxSdkError( (U32) rc);
```

```
  exit(2);
}

/* Get the BAR range */
range = 0xFFFFFFFF;
rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &range);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}

range =  PlxPciConfigRegisterRead(0x0,
                                  0x12,
                                  PCI9080_LOCAL_BASE0,
                                  &rc);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterRead failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}

rc = PlxPciConfigRegisterWrite(0x0,
                               0x12,
                               PCI9080_LOCAL_BASE0,
                               &address);
if ( rc != ApiSuccess )
{
  printf("\a\nPlxPciConfigRegisterWrite failed.");
  PlxSdkError( (U32) rc);
  exit(2);
}
```

PlxPciDeviceClose

Syntax:

`RETURN_CODE PlxPciDeviceClose( IN HANDLE drvHandle);`

PCI IC Support:

PCI 9080, PCI 9054

Description:

Closes a PLX device channel.

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

Note: *This function should be preceded by a call to PlxPciDeviceOpen() and should be used to close a PLX device channel before the application terminates.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiInvalidDeviceInfo`	The device was not closed properly.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
DEVICE_LOCATION device;
/* Open any supported device */
device.SerialNumber[0] = 0;
device.BusNumber = MINUS_ONE_LONG;
device.SlotNumber = MINUS_ONE_LONG;
device.DeviceId = MINUS_ONE_LONG;
device.VendorId = MINUS_ONE_LONG;
rc = PlxPciDeviceOpen(&device, &myPlxHandle);
if (rc != ApiSuccess) {
  printf("Could not find any supported device.\n");
  printf ("Press any key to exit.....");
  getch();
  return;
}
/* Close a handle to the PLX device */
rc = PlxPciDeviceClose(myPlxDevice);
if (rc != ApiSuccess) {
  printf ("\n\nErrors in closing device.\n");
  printf ("Returned code is %d\n",rc);
  printf ("Press any key to exit.....");
  getch();
  return;
}
```

PlxPciDeviceFind

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

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

Note: *If the* `SerialNumber` *element within the DEVICE_LOCATION structure is not being used as a search criteria the first character should be set to '\0'.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *device* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidDeviceInfo`	The information in *device* is not valid for any device in the system.	PCI 9080, PCI 9054
`ApiNoActiveDriver`	There is no device driver installed into the system.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054

Usage:

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

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

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

if ( (rc = PlxPciDeviceFind (&device, &reqLimit) != ApiSuccess) ||
     ( reqLimit < 1 ) )
{
  printf("\a\nCould not find the 0x%04x:0x%04x board in system. ",
         device.VendorID, device.DeviceId);
  PlxSdkError((U32) rc);
  exit (2);
}
```

PlxPciDeviceOpen

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

> If no serial number if specified (pDevice->SerialNumber[0] is '\0'), the *PlxPciDeviceOpen()* calls *PlxPciDeviceFind()* and tries to get a handle to the first device matching the combination of bus number, slot number, vendor ID and/or device ID.
>
> If a serial number is specified, the other members of the *pDevice* structure are ignored and *PlxPciDeviceOpen()* will try to get a handle to the PLX device described by the unique serial number. Then, the other members of the structure are filled with the appropriate values and the structure is returned.

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

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

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

Note: *This function should be used to open a channel to a PLX device and get a handle to the device driver before any PCI API functions that require a handle can be used. The handle returned by this API function should be closed using the PlxPciDeviceClose() API function before the user application exits.*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054

Return Value	Description	IC Support
`ApiNullParam`	The *pDevice* or *pDrvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidDeviceInfo`	The information in *pDevice* is not valid for any device in the system.	PCI 9080, PCI 9054
`ApiNoActiveDriver`	There is no device driver installed into the system.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054

Usage:

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

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

PlxPowerLevelGet

Syntax:

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

PCI IC Support:

PCI 9080 (with restrictions), PCI 9054.

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

Description:

Gets the current power level of a PCI device.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054

Usage:

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

PlxPowerLevelSet

Syntax:

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

PCI IC Support:

PCI 9080 (with restrictions), PCI 9054.

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

Description:

Sets the power level of a PCI device.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidPowerState`	The *plxPowerLevel* parameter is invalid for this PCI IC.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

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

PlxRegisterDoorbellRead

Syntax:

```
U64 PlxRegisterDoorbellRead( IN HANDLE drvHandle,
                             OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Clears and reads value to any doorbell register on a PCI device.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U64 value;

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

PlxRegisterDoorbellSet

Syntax:

```
RETURN_CODE PlxRegisterDoorbellSet( IN HANDLE drvHandle,
                                    IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to any doorbell register on a PCI device.

- *drvHandle* is the handle of the PCI device;
- *data* is a U64 value to store in the register.

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U64 value;

rc = PlxRegisterDoorbellSet(myPlxDevice, 0x1234567);
if (rc != ApiSuccess)
{
  printf("\n ERROR: Setting Doorbell did not work");
  getch();
}
else
{
  if (PlxRegisterRead(myPlxDevice, 0x60, &rc) == 0x1234567)
    printf("\n OK: DoorbellSet succeeded");
  else
  {
    printf("\n ERROR: After setting doorbell and reading the "
           "register the return data was wrong!");
    printf("\n ERROR: This is normal since the BSP will clear "
           "the doorbell immediately!");
    getch();
  }
}
```

PlxRegisterMailboxRead

Syntax:

```
U64 PlxRegisterMailboxRead( IN HANDLE drvHandle,
                            IN MAILBOX_ID mailboxId,
                            OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads any mailbox register on the currently selected PCI device.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *mailboxId* parameter is not a valid mailbox ID.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U64 value;

value = PlxRegisterMailboxRead(myPlxDevice, MailBox0, &rc);
if (rc != ApiSuccess)
{
  printf("\n Error Reading a valid Mailbox");
  getch();
}
else
  printf("\n OK: Valid Mailbox Read succeeded with a value of %x",
         value);
```

PlxRegisterMailboxWrite

Syntax:

```
RETURN_CODE PlxRegisterMailboxWrite( IN HANDLE drvHandle,
                                     IN MAILBOX_ID mailboxId,
                                     IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

- *drvHandle* is the handle of the PCI device;
- *mailboxId* is the mailbox register ID; and,
- *data* is a U64 value to store in the register.

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *mailboxId* parameter is not a valid mailbox ID.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;

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

PlxRegisterRead

Syntax:

```
U64 PlxRegisterRead( IN HANDLE drvHandle,
                     IN U32 registerOffset,
                     OUT PRETURN_CODE returnCode);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads any register on the currently selected PCI device.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U64 value;

value = PlxRegisterRead(myPlxDevice, 0x18, &rc);
if (rc != ApiSuccess)
{
  printf("\n Error Reading a valid Register");
  getch();
}
else
  printf("\n OK: Valid register Read succeeded with a value of %x",
         value);
```

PlxRegisterReadAll

Syntax:

```
RETURN_CODE PlxRegisterReadAll( IN HANDLE drvHandle,
                                IN U32 startOffset,
                                IN U32 registerCount,
                                OUT PU64 buffer);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads multiple registers on a PCI device.

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

Note: registerCount *is the number of* ***bytes*** *to read. Before this function can be used, a PCI device must be selected using PlxPciDeviceOpen(). This function will not read the PCI configuration registers (use PlxPciConfigRegisterReadAll())*

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or the *buffer* parameter is NULL.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *startOffset* and the *registerCount* parameters combined exceed the valid range of registers or *startOffset* is not on a 4-byte boundary.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;
U32 buffer[0x40];
rc = PlxRegisterReadAll(myPlxDevice, 0x0, 0x100, buffer);
if (rc != ApiSuccess) {
  printf("\n ERROR: Reading All registers failed API call");
  getch();
}
```

PlxRegisterWrite

Syntax:

```
RETURN_CODE PlxRegisterWrite( IN HANDLE drvHandle,
                              IN U32 registerOffset,
                              IN U64 data);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes a value to any register on a PCI device.

- *drvHandle* is the handle of the PCI device;
- *registerNumber* is the register number; and,
- *data* is a U64 value to store in the register.

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidRegister`	The *registerOffset* parameter is out of range or is not on a 4-byte boundary.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE myPlxDevice;

rc = PlxRegisterWrite(myPlxDevice, 0x18, 0x234980C4);
if (rc != ApiSuccess)
{
  printf("\n Error Writing a valid Register");
  getch();
}
else
  printf("\n OK: Valid register Write succeeded");
```

PlxSdkVersion

Syntax:

```
RETURN_CODE PlxSdkVersion(const S8 **versionString,
                          const S8 **dateString);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Returns the version string of the PCI SDK PCI API.

- *versionString* is the version string for the PCI API; and,
- *dateString* is the version date string for the PCI API.

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *versionString* or *dateString* parameter is NULL.	PCI 9080, PCI 9054

Usage:

```
PS8 ptrDate;
PS8 ptrVer;

rc = PlxSdkVersion(&ptrVer, &ptrDate);
if (rc != ApiSuccess)
{
  printf("\n ERROR: Getting SDK Version Number. RC = %x", rc);
  getch();
}
else
{
  printf("\nCurrent version: %s", ptrVer);
  printf("\nVersion date   : %s", ptrDate);
}
```

PlxSerialEepromRead

Syntax:

```
RETURN_CODE PlxSerialEepromRead( IN HANDLE drvHandle,
                                 IN EEPROM_TYPE eepromType,
                                 OUT PU64 buffer,
                                 IN U32 size);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads values from the configuration EEPROM.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *buffer* parameters are NULL.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054
`ApiEepromNotPresent`	There is no EEPROM connected to the PCI IC.	PCI 9080, PCI 9054
`ApiEepromTypeNotSupported`	The *eepromType* parameter is not supported for this PCI IC.	PCI 9080, PCI 9054

Usage:

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

/* Reading EEPROM into eepromData buffer */
rc = PlxSerialEepromRead(plxHandle,
                         Eeprom93CS56,
                         (U64 *)eepromData,
                         0x16 * sizeof(U16));  /* size in bytes */
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxSerialEepromWrite

Syntax:

```
RETURN_CODE PlxSerialEepromWrite( IN HANDLE drvHandle,
                                  IN EEPROM_TYPE eepromType,
                                  OUT PU64 buffer,
                                  IN U32 size);
```

PCI IC Support:

PCI 9080, PCI 9054

Description:

Writes values to the configuration EEPROM.

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* or *buffer* parameters are NULL.	PCI 9080, PCI 9054
`ApiInvalidSize`	The *size* parameter is 0, is too large for this *eepromType*, or is not 2 byte aligned.	PCI 9080, PCI 9054
`ApiInsufficientResources`	There is no memory available for the PCI API.	PCI 9080, PCI 9054
`ApiEepromNotPresent`	There is no EEPROM connected to the PCI IC.	PCI 9080, PCI 9054
`ApiEepromTypeNotSupported`	The *eepromType* parameter is not supported for this PCI IC.	PCI 9080, PCI 9054

Usage:

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

/* Write eepromData buffer to the serial EEPROM */
rc = PlxSerialEepromWrite(plxHandle,
                          Eeprom93CS56,
                          (U64 *)eepromData,
                          0x16 * sizeof(U16));  /* size in bytes */
if (rc != ApiSuccess)
  PlxSdkError( (U32)rc, TRUE);
```

PlxUserRead

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

Reads the PCI device's USERI pins.

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

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

Return Value:

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

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidUserPin`	This *userPin* is not present on this PCI IC.	PCI 9080, PCI 9054

Usage:

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

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

PlxUserWrite

Syntax:

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

PCI IC Support:

PCI 9080, PCI 9054

Description:

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

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

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

Return Value:

Return Value	Description	IC Support
`ApiSuccess`	The function returned successfully.	PCI 9080, PCI 9054
`ApiNullParam`	The *drvHandle* parameter is NULL.	PCI 9080, PCI 9054
`ApiInvalidUserPin`	This *userPin* is not present on this PCI IC.	PCI 9080, PCI 9054

Usage:

```
RETURN_CODE rc;
HANDLE drvHandle;

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

5. PCI SDK Data Structures

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

Sample Data Structure

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

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
someRegister	xxx, yy	xxx, yy	xxx, yy

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

Purpose

The reasons for using this structure.

Members

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

Comments

Extra comments on how and when this structure is used.

S8 and U8 Data Types

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

Affected Register Location

Not applicable.

Purpose

These data types are used for 8 bit values.

Members

Comments

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

S16 and U16 Data Types

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

Affected Register Location

Not applicable.

Purpose

These data types are used for 16 bit values.

Members

Comments

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

S32 and U32 Data Types

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

Affected Register Location

Not applicable.

Purpose

These data types are used for 32 bit values.

Members

Comments

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

U64 Data Type

```
#ifdef IOP_CODE
  #ifdef longlong
    #define BITS_64        1
    typedef signed long long int S64, *PS64;
    typedef unsigned long long int U64, *PU64;
  #else /* longlong */
    #define BITS_64        0
    typedef signed long S64, *PS64;
    typedef unsigned long U64, *PU64;
  #endif /* longlong */

#elif PCI_CODE
  #ifdef BITS_64
    typedef LARGE_INTEGER U64, *PU64;
  #else
    typedef U32 U64, *PU64;
  #endif /* BITS_64 */
#endif /* IOP_CODE or PCI_CODE */
```

Affected Register Location

Not applicable.

Purpose

This data type is used for PCI API function compatibility between 32 bit and 64 bit systems and compilers.

Members

Comments

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

Access Type Enumerated Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used for determining the access type size.

Members

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

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

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

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

Comments

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

API Parameters Structure

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

Affected Register Location

Not applicable.

Purpose

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

Members

PlxIcIopBaseAddr
: The base IOP address for the PCI IC.

PtrPciBusProp
: A pointer to the PCI_BUS_PROP structure that will be used to initialize the PCI Bus properties of the PCI IC.

PtrPciArbitDesc
: A pointer to the PCI_ARBIT_DESC structure that will be used to initialize the PCI Bus arbiter of the PCI IC.

PtrIopBus0Prop
: A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 0 register accesses to the IOP Bus.

PtrIopBus1Prop
: A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 1 register accesses to the IOP Bus.

PtrIopBus2Prop
: A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 2 register accesses to the IOP Bus.

PtrIopBus3Prop
: A pointer to the IOP_BUS_PROP structure that will be used to initialize the Local Space 3 register accesses to the IOP Bus.

PtrExpRomBusProp

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

PtrIopArbitDesc

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

PtrIopEndianDesc

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

PtrPMProp

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

PtrVPDBaseAddress

A pointer to a U32 that will be used to initialize the Vital Product Data Write-Protected Address Boundary.

Comments

This data type provides information about the board to the IOP API and initializes initialization data structures to the PCI IC's default values.

BAR Space Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

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

Members

Bar0
: Use Base Address Register 0.

Bar1
: Use Base Address Register 1.

Bar2
: Use Base Address Register 2.

Bar3
: Use Base Address Register 3.

Bar4
: Use Base Address Register 4.

Bar5
: Use Base Address Register 5.

IopExpansionRom
: Use Expansion ROM base address register.

Comments

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

Bus Index Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired PCI bus.

Members

PrimaryPciBus
: Use the primary PCI bus.

SecondaryPciBus
: Use the secondary PCI bus.

Comments

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

Device Location Data Type

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

Purpose

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

Members

DeviceId

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

VendorId

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

BusNumber

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

SlotNumber

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

SerialNumber

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

Comments

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

DMA Channel Descriptor Structure

```
typedef struct _DMA_CHANNEL_DESC
{
  U32 EnableReadyInput          :1;
  U32 EnableBTERMInput          :1;
  U32 EnableIopBurst            :1;
  U32 EnableWriteInvalidMode    :1;
  U32 EnableDmaEOTPin           :1;
  U32 DmaStopTransferMode       :1;
  U32 HoldIopAddrConst          :1;
  U32 HoldIopSourceAddrConst    :1;
  U32 HoldIopDestAddrConst      :1;
  U32 DemandMode                :1;
  U32 EnableTransferCountClear  :1;
  U32 WaitStates                :4;
  U32 IopBusWidth               :2;
  U32 EnableDualAddressCycles   :1;
  U32 Reserved1                 :14;
  U32 TholdForIopWrites         :4;
  U32 TholdForIopReads          :4;
  U32 TholdForPciWrites         :4;
  U32 TholdForPciReads          :4;
  U32 Reserved2                 :16;
  DMA_CHANNEL_PRIORITY DmaChannelPriority;
} DMA_CHANNEL_DESC, *PDMA_CHANNEL_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableReadyInput	0x100, 6 0x114, 6	0x100, 6 0x114, 6	Currently Not Defined
EnableBTERMInput	0x100, 7 0x114, 7	0x100, 7 0x114, 7	0x154, 10
EnableIopBurst	0x100, 8 0x114, 8	0x100, 8 0x114, 8	0x154, 8
EnableWriteInvalidMode	0x100, 13 0x114, 13	0x100, 13 0x114, 13	0x200, 13 0x220, 13
EnableDmaEOTPin	0x100, 14 0x114, 14	0x100, 14 0x114, 14	0x200, 14 0x220, 14 0x240, 14
DmaStopTransferMode	0x100, 15 0x114, 15	0x100, 15 0x114, 15	0x200, 15 0x220, 15 0x240, 15
HoldIopAddrConst	0x100, 11 0x114, 11	0x100, 11 0x114, 11	0x200, 11 0x220, 11
HoldIopSourceAddrConst	*N/A*	*N/A*	0x240, 7
HoldIopDestAddrConst	*N/A*	*N/A*	0x240, 8
DemandMode	0x100, 12 0x114, 12	0x100, 12	0x200, 12 0x220, 12 0x240, 12
EnableTransferCountClear ****Note:** Can clear transfer count only if DMA chain is located on the IOP bus. Cannot clear transfer count on DMA chains located on the PCI bus.*	0x100, 16* 0x114, 16*	0x100, 16 0x114, 16	0x200, 16 0x220, 16 0x240, 16
WaitStates	0x100, 2-5 0x114, 2-5	0x100, 2-5 0x114, 2-5	Currently Not Defined

Structure Element	PCI 9080	PCI 9054	IOP 480
IopBusWidth * ***Note:*** *Bus width is shared for all DMA channels.*	0x100, 0-1 0x114, 0-1	0x100, 0-1 0x114, 0-1	0x154, 0-1*
EnableDualAddressCycles	*N/A*	0x100, 18 0x114, 18	Currently Not Defined
TholdForIopWrites	0x130, 0-3 0x130, 16-19	0x130, 0-3 0x130, 16-19	0x21C, 0-3 0x23C, 0-3
TholdForIopReads	0x130, 4-7 0x130, 20-23	0x130, 4-7 0x130, 20-23	0x21C, 4-7 0x23C, 4-7
TholdForPciWrites	0x130, 8-11 0x130, 24-27	0x130, 8-11 0x130, 24-27	0x21C, 8-11 0x23C, 8-11
TholdForPciReads	0x130, 12-15 0x130, 28-30	0x130, 12-15 0x130, 28-30	0x21C, 12-15 0x23C, 12-15
DmaChannelPriority	0x88, 19-20	0x88, 19-20	0x90, 4-5

Purpose

Structure containing the DMA channel descriptors used to configure the DMA channel.

Members

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

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

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

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

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

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

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

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

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

DemandMode
The DMA controller will work in demand mode if this value is set.

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

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

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

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

Reserved1
This value is reserved for future definitions.

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

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

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

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

Reserved2
This value is reserved for future definitions.

DmaChannelPriority
The DMA channel priority scheme is set with this value.

Comments

The DMA channel descriptor structure is used to configure the DMA channel.

DMA Channel Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used for requesting a DMA channel.

Members

IopChannel0
: Request the IOP-IOP DMA channel 0.

IopChannel1
: Request the IOP-IOP DMA channel 1.

PrimaryPciChannel0
: Request the Primary PCI-IOP DMA channel 0.

PrimaryPciChannel1
: Request the Primary PCI-IOP DMA channel 1.

SecondaryPciChannel0
: Request the Secondary PCI-IOP DMA channel 0.

SecondaryPciChannel1
: Request the Secondary PCI-IOP DMA channel 1.

Comments

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

DMA Channel Priority Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

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

Members

Channel0Highest
DMA channel 0 has the highest priority.

Channel1Highest
DMA channel 1 has the highest priority.

Channel2Highest
DMA channel 2 has the highest priority.

Channel3Highest
DMA channel 3 has the highest priority.

Rotational
Rotate the channel priority.

Comments

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

DMA Command Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used to control a DMA transfer.

Members

DmaStart
: Start a DMA transfer.

DmaPause
: Suspend a DMA transfer.

DmaResume
: Resume a DMA transfer.

DmaAbort
: Abort a DMA transfer.

DmaStatus
: Determine the status of a DMA transfer.

Comments

The DMA command enumerated type is used to control a DMA transfer.

DMA Direction Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used for providing the DMA transfer direction.

Members

IopToIop
: Transfer between two IOP bus addresses.

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

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

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

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

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

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

Comments

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

DMA Resource Manager Parameters Structure

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

Affected Register Location

Not applicable.

Purpose

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

Members

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

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

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

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

Comments

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

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

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

  struct
  {
  #if defined(PCI_CODE)
    union
    {
      U32 LowPciAddr;
      U32 UserAddr;
    };
  #else
    U32 LowPciAddr;
  #endif
    U32 IopAddr;
    U32 TransferCount;
  #ifdef LITTLE_ENDIAN
    U32 PciSglLoc          :1;
    U32 LastSglElement     :1;
    U32 TerminalCountIntr  :1;
    U32 IopToPciDma        :1;
    U32 NextSglPtr         :28;
  #elif BIG_ENDIAN
    U32 NextSglPtr         :28;
    U32 IopToPciDma        :1;
    U32 TerminalCountIntr  :1;
```

```
    U32 LastSglElement     :1;
    U32 PciSglLoc          :1;
  #else
    #error Need To Define Endian Type Used.
  #endif /* LITTLE_ENDIAN or BIG_ENDIAN */
    U32 HighPciAddr;
  } Pci9054Dma;

  struct
  {
    U32 LowPciAddr;
    U32 HighPciAddr;
    U32 IopAddr;
    U32 TransferCount;
  #ifdef LITTLE_ENDIAN
    U32 PciSglLoc          :1;
    U32 LastSglElement     :1;
    U32 TerminalCountIntr  :1;
    U32 IopToPciDma        :1;
    U32 NextSglPtr         :28;
  #elif BIG_ENDIAN
    U32 NextSglPtr         :28;
    U32 IopToPciDma        :1;
    U32 TerminalCountIntr  :1;
    U32 LastSglElement     :1;
    U32 PciSglLoc          :1;
  #else
    #error Need To Define Endian Type Used.
  #endif /* LITTLE_ENDIAN or BIG_ENDIAN */
  } Iop480Dma;

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

typedef PDMA_TRANSFER_ELEMENT SGL_ADDR, *PSGL_ADDR;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
LowPciAddr / UserAddr	0x104 0x118	0x104 0x118	0x208 0x228
HighPciAddr	N/A	0x134 0x138	0x20C 0x22C
IopAddr	0x108 0x11C	0x108 0x11C	0x210 0x230
TransferCount	0x10C 0x120	0x10C 0x120	0x214 0x234
PciSglLoc	0x110, 0 0x124, 0	0x110, 0 0x124, 0	0x218, 0 0x238, 0
LastSglElement	0x110, 1 0x124, 1	0x110, 1 0x124, 1	0x218, 1 0x238, 1
TerminalCountIntr	0x110, 2 0x124, 2	0x110, 2 0x124, 2	0x218, 2 0x238, 2

Structure Element	PCI 9080	PCI 9054	IOP 480
IopToPciDma	0x110, 3 0x124, 3	0x110, 3 0x124, 3	0x218, 3 0x238, 3
NextSglPtr	0x110, 4-31 0x124, 4-31	0x110, 4-31 0x124, 4-31	0x218, 4-31 0x238, 4-31

Purpose

Structure containing the DMA data used to program the DMA registers.

Members

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

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

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

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

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

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

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

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

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

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

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

Pci9080Dma.IopToPciDma, Pci9054Dma.IopToPciDma and Iop480Dma.IopToPciDma
The DMA transfer will transfer data from IOP memory space to PCI memory space if this

value is set. Otherwise, the data will be transferred from PCI memory space to IOP memory space.

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

Comments

The DMA Transfer Element structure is used to program the DMA Registers.

Echo State Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

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

Members

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

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

Comments

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

EEPROM Type Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

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

Members

Eeprom93CS46
Use a compatible EEPROM to the National NM93CS46 EEPROM.

Eeprom93CS56
Use a compatible EEPROM to the National NM93CS56 EEPROM.

Eeprom93CS66
Use a compatible EEPROM to the National NM93CS66 EEPROM.

EepromX24012
Use a compatible EEPROM to the Xicor X24012 EEPROM.

EepromX24022
Use a compatible EEPROM to the Xicor X24022 EEPROM.

EepromX24042
Use a compatible EEPROM to the Xicor X24042 EEPROM.

EepromX24162
Use a compatible EEPROM to the Xicor X24162 EEPROM.

Comments

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

IOP Arbitration Descriptor Structure

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

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
IopBusDSGiveUpBusMode	0x88, 21	0x88, 21	0x90, 8
EnableDSLockedSequence	0x88, 22	0x88, 22	0x90, 9
GateIopLatencyTimerBREQo	0x88, 27	0x88, 27	0x90, 10
EnableWAITInput	N/A	0x88, 31	Currently Not Defined.
EnableBOFF	N/A	N/A	0x90, 16
BOFFTimerResolution	N/A	N/A	0x90, 21
EnableIopBusLatencyTimer	0x88, 16	0x88, 16	0x8C, 8
EnableIopBusPauseTimer	0x88, 17	0x88, 17	0x8C, 24
EnableIopArbiter	N/A	N/A	0x90, 0
IopArbitrationPriority	N/A	N/A	0x90, 1-3
BOFFDelayClocks	N/A	N/A	0x90, 17-20
IopBusLatencyTimer	0x88, 0-7	0x88, 0-7	0x8C, 0-7
IopBusPauseTimer	0x88, 8-15	0x88, 8-15	0x8C, 16-23

Purpose

Structure used to describe the IOP bus arbitration.

Members

IopBusDSGiveUpBusMode

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

EnableDSLockedSequence

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

GateIopLatencyTimerBREQo

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

EnableWAITInput

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

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

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

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

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

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

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

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

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

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

Comments

The IOP arbitration descriptor structure is used to describe the IOP bus arbitration.

IOP Bus Properties Structure

```
typedef struct _IOP_BUS_PROP
{
  U32 EnableReadyInput          :1;
  U32 EnableBTERMInput          :1;
  U32 DisableReadPrefetch       :1;
  U32 EnableReadPrefetchCount   :1;
  U32 ReadPrefetchCounter       :4;
  U32 EnableBursting            :1;
  U32 EnableIopBusTimeoutTimer  :1;
  U32 BREQoTimerResolution      :1;
  U32 EnableIopBREQo            :1;
  U32 BREQoDelayClockCount      :4;
  U32 MapInMemorySpace          :1;
  U32 OddParitySelect           :1;
  U32 EnableParityCheck         :1;
  U32 MemoryWriteProtect        :1;
  U32 InternalWaitStates        :4;
  U32 PciRev2_1Mode             :1;
  U32 IopBusWidth               :2;
  U32 Reserved1                 :5;
  U32 Iop480WADWaitStates       :4;
  U32 Iop480WDDWaitStates       :4;
  U32 Iop480WDLYDelayStates     :3;
  U32 Iop480WHLDHoldStates      :3;
  U32 Iop480WRCRecoverStates    :3;
  U32 Reserved2                 :15;
  U32 Iop480RADWaitStates       :4;
  U32 Iop480RDDWaitStates       :4;
  U32 Iop480RDLYDelayStates     :3;
  U32 Iop480RRCRecoverStates    :3;
  U32 Reserved3                 :18;
} IOP_BUS_PROP, *PIOP_BUS_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableReadyInput	0x98, 6 0x98, 22 0x178, 6	0x98, 6 0x98, 22 0x178, 6	0x100, 9 0x110, 9 0x120, 9
EnableBTERMInput	0x98, 7 0x98, 23 0x178, 7	0x98, 7 0x98, 23 0x178, 7	0x100, 10 0x110, 10 0x120, 10 0x130, 10
DisableReadPrefetch * ***Note:*** *By setting the Read Prefetch Count to 00 disables Read Prefetching.*	0x98, 8 0x98, 9 0x178, 9	0x98, 8 0x98, 9 0x178, 9	0x100, 17-18* 0x110, 17-18* 0x120, 17-18* 0x130, 17-18*
EnableReadPrefetchCount	0x98, 10 0x178, 10	0x98, 10 0x178, 10	0x100, 16 0x110, 16 0x120, 16 0x130, 16
ReadPrefetchCounter	0x98, 11-14 0x178, 11-14	0x98, 11-14 0x178, 11-14	0x100, 17-18 0x110, 17-18 0x120, 17-18 0x130, 17-18

Structure Element	PCI 9080	PCI 9054	IOP 480
EnableBursting	0x98, 24 0x98, 26 0x178, 8	0x98, 24 0x98, 26 0x178, 8	0x100, 8 0x110, 8 0x120, 8 0x130, 8
EnableIopBusTimeoutTimer	N/A	N/A	0x88, 15
BREQoTimerResolution	0x94, 5	0x94, 5	N/A
EnableIopBREQo	0x94, 4	0x94, 4	N/A
BREQoDelayClockCount	0x94, 0-3	0x94, 0-3	N/A
MapInMemorySpace	0x80, 0 0x170,0	0x80, 0 0x170,0	0xA8, 0 0xB0, 0
OddParitySelect	N/A	N/A	0x100, 13 0x110, 13 0x120, 13 0x130, 13
EnableParityCheck	N/A	N/A	0x100, 12 0x110, 12 0x120, 12 0x130, 12
MemoryWriteProtect	N/A	N/A	0x100, 11 0x110, 11 0x120, 11 0x130, 11
PciRev2_1Mode	0x88, 24	0x88, 24	0x98, 25
IopBusWidth	0x98, 0-1 0x98, 16-17 0x178, 0-1	0x98, 0-1 0x98, 16-17 0x178, 0-1	0x100, 0-1 0x110, 0-1 0x120, 0-1 0x130, 0-1
InternalWaitStates	0x98, 2-5 0x98, 18-21 0x178, 2-5	0x98, 2-5 0x98, 18-21 0x178, 2-5	Currently Not Defined.
Iop480WADWaitStates	N/A	N/A	0x104, 0-3 0x114, 0-3 0x124, 0-3 0x134, 0-3
Iop480WDDWaitStates	N/A	N/A	0x104, 4-7 0x114, 4-7 0x124, 4-7 0x134, 4-7
Iop480WDLYDelayStates	N/A	N/A	0x104, 8-10 0x114, 8-10 0x124, 8-10 0x134, 8-10
Iop480WHLDHoldStates	N/A	N/A	0x104, 11-13 0x114, 11-13 0x124, 11-13 0x134, 11-13
Iop480WRCRecoverStates	N/A	N/A	0x104, 16-18 0x114, 16-18 0x124, 16-18 0x134, 16-18
Iop480RADWaitStates	N/A	N/A	0x108, 0-3 0x118, 0-3 0x128, 0-3 0x138, 0-3
Iop480RDDWaitStates	N/A	N/A	0x108, 4-7 0x118, 4-7 0x128, 4-7 0x138, 4-7
Iop480RDLYDelayStates	N/A	N/A	0x108, 8-10

Structure Element	PCI 9080	PCI 9054	IOP 480
			0x118, 8-10 0x128, 8-10 0x138, 8-10
Iop480RRCRecoverStates	N/A	N/A	0x108, 16-18 0x118, 16-18 0x128, 16-18 0x138, 16-18

Purpose

Structure used to describe the local bus characteristics.

Members

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

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

DisableReadPrefetch
Read prefetching is disabled when this value is set.

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

ReadPrefetchCounter
Stores the number of values that can be prefetched.

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

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

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

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

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

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

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

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

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

PciRev2_1Mode
The PCI device operates in Delayed Transaction mode for Direct Slave reads when this value is set. Otherwise, the PCI device does not return a TRDY# signal to the PCI host until the read data are available.

IopBusWidth
The width of the IOP bus.

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

Reserved1
This value is reserved for future definitions.

Iop480WADWaitStates
This value contains the number of Write Address-to-Data wait states to insert for the IOP 480. The value must be between 0-15.

Iop480WDDWaitStates
This value contains the number of Write Data-to-Data wait states (used when bursting) to insert for the IOP 480. The value must be between 0-15.

Iop480WDLYWaitStates
This value contains the number of Write Enable Delay states to insert for the IOP 480. The value must be between 0-15.

Iop480WHLDWaitStates
This value contains the number of Write Hold states to insert for the IOP 480. The value must be between 0-15.

Iop480WRCVWaitStates
This value contains the number of Write Recovery states to insert for the IOP 480. The value must be between 0-15.

Reserved2
This value is reserved for future definitions.

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

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

Iop480RDLYWaitStates
This value contains the number of Read Enable Delay states to insert for the IOP 480. The value must be between 0-15.

Iop480RRCVWaitStates

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

Reserved3

This value is reserved for future definitions.

Comments

The IOP bus properties structure is used to describe the IOP bus characteristics.

IOP Endian Descriptor Structure

```
typedef struct _IOP_ENDIAN_DESC
{
  U32 BigEIopSpace0           :1;
  U32 BigEIopSpace1           :1;
  U32 BigEExpansionRom        :1;
  U32 BigEIopBusRegion0       :1;
  U32 BigEIopBusRegion1       :1;
  U32 BigEIopBusRegion2       :1;
  U32 BigEIopBusRegion3       :1;
  U32 BigEDramBusRegion       :1;
  U32 BigEDefaultBusRegion    :1;
  U32 BigEDirectMasterAccess  :1;
  U32 BigEDmaChannel0         :1;
  U32 BigEDmaChannel1         :1;
  U32 BigEIopConfigRegAccess  :1;
  U32 BigEByteLaneMode        :1;
  U32 BigEByteLaneModeLBR0    :1;
  U32 BigEByteLaneModeLBR1    :1;
  U32 BigEByteLaneModeLBR2    :1;
  U32 BigEByteLaneModeLBR3    :1;
  U32 BigEByteLaneModeDRAMBR  :1;
  U32 BigEByteLaneModeDefBR   :1;
  U32 Reserved1               :12;
  U32 ReservedForFutureUse;
} IOP_ENDIAN_DESC, *PIOP_ENDIAN_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
BigELocalSpace0	0x8C, 2	0x8C, 2	N/A
BigELocalSpace1	0x8C, 5	0x8C, 5	N/A
BigEExpansionRom	0x8C, 3	0x8C, 3	N/A
BigELocalBusRegion0	N/A	N/A	0x100, 4
BigELocalBusRegion1	N/A	N/A	0x110, 4
BigELocalBusRegion2	N/A	N/A	0x120, 4
BigELocalBusRegion3	N/A	N/A	0x130, 4
BigEDramBusRegion	N/A	N/A	0x140, 4
BigEDefaultBusRegion	N/A	N/A	0x154, 4
BigEDirectMasterAccess	0x8C, 1	0x8C, 1	0x94, 1
BigEDmaChannel0	0x8C, 7	0x8C, 7	N/A
BigEDmaChannel1	0x8C, 6	0x8C, 6	N/A
BigEIopConfigRegAccess	0x8C, 0	0x8C, 0	0x94, 0
BigEByteLaneMode	0x8C, 4	0x8C, 4	N/A
BigEByteLaneModeLBR0	N/A	N/A	0x100, 3
BigEByteLaneModeLBR1	N/A	N/A	0x110, 3
BigEByteLaneModeLBR2	N/A	N/A	0x120, 3
BigEByteLaneModeLBR3	N/A	N/A	0x130, 3
BigEByteLaneModeDRAMBR	N/A	N/A	0x140, 3
BigEByteLaneModeDefBR	N/A	N/A	0x154, 3

Purpose

Structure used to describe the IOP bus endian data ordering.

Members

BigELocalSpace0
The local space 0 is configured with big endian data ordering if this value is set.

BigELocalSpace1
The local space 1 is configured with big endian data ordering if this value is set.

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

BigELocalBusRegion0
The local bus region 0 is configured with big endian data ordering if this value is set.

BigELocalBusRegion1
The local bus region 1 is configured with big endian data ordering if this value is set.

BigELocalBusRegion2
The local bus region 2 is configured with big endian data ordering if this value is set.

BigELocalBusRegion3
The local bus region 3 is configured with big endian data ordering if this value is set.

BigEDramBusRegion
The DRAM bus region is configured with big endian data ordering if this value is set.

BigEDefaultBusRegion
The default bus region is configured with big endian data ordering if this value is set.

BigEDirectMasterAccess
The direct master access is configured with big endian data ordering if this value is set.

BigEDmaChannel0
The DMA channel 0 is configured with big endian data ordering if this value is set.

BigEDmaChannel1
The DMA channel 1 is configured with big endian data ordering if this value is set.

BigEIopConfigRegAccess
The IOP configuration register access is configured with big endian data ordering if this value is set.

BigEByteLaneMode
When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeLBR0
When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus for local bus region 0. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeLBR1
When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus for local bus region 1. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeLBR2
When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus for local bus region 2. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeLBR3

When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus for local bus region 3. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

BigEByteLaneModeDRAMBR

When this value is set use D31:16 for 16-bit bus and D31:24 for 8-bit bus for DRAM bus region. Otherwise, use D15:0 for 16-bit bus and D7:0 for 8-bit bus.

Reserved1

This value is reserved for future definitions.

ReservedForFutureUse

This value is reserved for future definitions.

Comments

The IOP endian descriptor structure is used to configure the IOP bus endian data ordering.

IOP Space Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used for choosing the desired Local Address Space registers when accessing the IOP memory space.

Members

IopSpace0
: Use Local Space 0 base address register.

IopSpace1
: Use Local Space 1 base address register.

IopSpace2
: Use Local Space 2 base address register.

IopSpace3
: Use Local Space 3 base address register.

ExpansionRom
: Use Expansion ROM base address register.

Comments

The IOP space enumerated type is used to choose the Local Space Base Address register.

Mailbox ID Enum Data Type

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

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
MailBox0	0xC0	0xC0	0x180
MailBox1	0xC4	0xC4	0x184
MailBox2	0xC8	0xC8	0x188
MailBox3	0xCC	0xCC	0x18C
MailBox4	0xD0	0xD0	0x190
MailBox5	0xD4	0xD4	0x194
MailBox6	0xD8	0xD8	0x198
MailBox7	0xDC	0xDC	0x19C

Purpose

Enumerated type used for choosing the desired mailbox register.

Members

MailBox0
Use mailbox 0 register.

MailBox1
Use mailbox 1 register.

MailBox2
Use mailbox 2 register.

MailBox3
Use mailbox 3 register.

MailBox4
Use mailbox 4 register.

MailBox5
Use mailbox 5 register.

MailBox6
Use mailbox 6 register.

MailBox7: Use mailbox 7 register.

Comments

The mailbox ID enumerated type is used to select a specific mailbox register.

PCI Arbitration Descriptor Structure

```
typedef struct _PCI_ARBIT_DESC
{
  U32 Tbd;
} PCI_ARBIT_DESC, *PPCI_ARBIT_DESC;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480

Purpose

Structure used to describe the PCI bus arbitration.

Note: *This structure will be determined at a future date.*

Members

Tbd

This value and the other values of this structure will be determined at a future date.

Comments

PCI Bus Properties Structure

```
typedef struct _PCI_BUS_PROP
{
  U32 PciRequestMode           :1;
  U32 DmPciReadMode            :1;
  U32 EnablePciArbiter         :1;
  U32 EnableWriteInvalidMode   :1;
  U32 DmPrefetchLimit          :1;
  U32 PciReadNoWriteMode       :1;
  U32 PciReadWriteFlushMode    :1;
  U32 PciReadNoFlushMode       :1;
  U32 EnableRetryAbort         :1;
  U32 WFifoAlmostFullFlagCount :5;
  U32 DmWriteDelay             :2;
  U32 ReadPrefetchMode         :3;
  U32 IoRemapSelect            :3;
  U32 Reserved1                :8;
  U32 ReservedForFutureUse;
} PCI_BUS_PROP, *PPCI_BUS_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
PciRequestMode	0x88, 23	0x88, 23	0x98, 26
DmPciReadMode	0xA8, 4	0xA8, 4	0xD0, 6
EnablePciArbiter	N/A	N/A	0x98, 16
EnableWriteInvalidMode	0xA8, 9	0xA8, 9	0xD0, 7
DmPrefetchLimit	0xA8, 11	0xA8, 11	0xD0, 5
PciAddressSpaceBusWidth	N/A	N/A	Currently Not Defined.
PciReadNoWriteMode	0x88, 25	0x88, 25	0x98, 24
PciReadWriteFlushMode	0x88, 26	0x88, 26	0x98, 23
PciReadNoFlushMode	0x88, 28	0x88, 28	0x98, 22
EnableRetryAbort	N/A	N/A	0x98, 21
WFifoAlmostFullFlagCount	0xA8, 5-8 0xA8, 10	0xA8, 5-8 0xA8, 10	0xD0, 8-15
DmWriteDelay	0xA8, 14-15	0xA8, 14-15	0xD0, 13-14
ReadPrefetchMode	0xA8, 3 0xA8, 12	0xA8, 3 0xA8, 12	0xD0, 3-4
IoRemapSelect	0xA8, 13	0xA8, 13	0xD0, 15

Purpose

Structure used to describe the PCI bus characteristics.

Members

PciRequestMode

When this value is set the PCI device negates REQ0# when it asserts FRAME# during a bus master cycle. Otherwise, the PCI device leaves REQ0# asserted for the entire bus master cycle.

DmPciReadMode

When this value is set the PCI device should keep the PCI bus and de-assert IRDY# when

the read FIFO becomes full. Otherwise, the PCI device should release the PCI bus when the read FIFO becomes full.

EnablePciArbiter
The PCI arbiter is enabled when this value is set. Otherwise, the PCI arbiter is disabled and the PCI device uses REQ0# and GNT0# to acquire the PCI bus.

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

DmPrefetchLimit
The prefetching is terminated at 4K boundaries when this value is set.

PciAddressSpaceBusWidth
When this value is set the PCI bus width for PCI space is 64 bits. Otherwise, the PCI bus width is 32 bits.

PciReadNoWriteMode
When this value is set PCI device forces a retry on writes if read is pending. Otherwise, the PCI device allows writes while a read is pending.

PciReadWriteFlushMode
When this value is set PCI device submits a request to flush a pending read cycle if a write cycle is detected. Otherwise, the PCI device submits a request to not effect pending reads when a write cycle occurs.

PciReadNoFlushMode
When this value is set PCI device submits a request to not flush the read FIFO if the PCI read cycle completes. Otherwise, the PCI device submits a request to flush the read FIFO if the PCI read cycle completes.

EnableRetryAbort
The IOP 480 treats 256 Master consecutive retries to a Target as a Target Abort when this value is set. Otherwise, the IOP 480 attempts Master Retries indefinitely.

WFifoAlmostFullFlagCount
This value sets the retry limit before asserting an IOP NMI signal.

DmWriteDelay
This value sets the delay clocks placed between the PCI bus request and the start of direct master burst write cycle.

ReadPrefetchMode
This value sets the amount of data that can be prefetched from the PCI bus and enables or disables read prefetching.

IoRemapSelect
When this value is set the PCI address bits 31:16 are forced to zero. Otherwise, the address bits 31:16 in the Direct Master Remap register will be used on the PCI bus.

Reserved1
This value is reserved for future definitions.

ReservedForFutureUse: This value is reserved for future definitions.

Comments

The PCI bus properties structure is used to describe the PCI bus characteristics.

PCI Memory Data Type

```
typedef struct _PCI_MEMORY
{
    U32 UserAddr;
    U64 PhysicalAddr;
    U32 Size;
} PCI_MEMORY, *PPCI_MEMORY;
```

Purpose

This data type provides information on a Physical Memory Buffer (PBM) located in the device driver code.

Members

UserAddr
User Virtual Address for the PMB.

PhysicalAddr
Physical Address for the PMB.

Size
The size for the PMB.

Comments

This data type contains information on a Physical Memory Buffer.

PCI Space Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

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

Members

PciMemSpace
: Use PCI memory cycles when accessing the PCI bus.

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

Comments

The PCI space enumerated type is used to select the type of bus cycles when accessing the PCI bus.

Pin State Enum Data Type

```
typedef enum _PIN_STATE
{
  Inactive,
  Active
} PIN_STATE;
```

Affected Register Location

Not applicable.

Purpose

Enumerated type used to set the state of a USER pin.

Members

Inactive

Set the USER pin to inactive state. If the USER pin is active low then setting the USER pin inactive would cause it to go high.

Active

Set the USER pin to active state. If the USER pin is active low then setting the USER pin active would cause it to go low.

Comments

The pin state enumerated type is used to set the state of a USER pin.

PLX Interrupt Structure

```
typedef struct _PLX_INTR
{
  U32 InboundPost       :1;
  U32 OutboundPost      :1;
  U32 OutboundOverflow  :1;
  U32 IopDmaChannel0    :1;
  U32 PciDmaChannel0    :1;
  U32 IopDmaChannel1    :1;
  U32 PciDmaChannel1    :1;
  U32 IopDmaChannel2    :1;
  U32 PciDmaChannel2    :1;
  U32 Mailbox0          :1;
  U32 Mailbox1          :1;
  U32 Mailbox2          :1;
  U32 Mailbox3          :1;
  U32 Mailbox4          :1;
  U32 Mailbox5          :1;
  U32 Mailbox6          :1;
  U32 Mailbox7          :1;
  U32 IopDoorbell       :1;
  U32 PciDoorbell       :1;
  U32 SerialPort1       :1;
  U32 SerialPort2       :1;
  U32 BIST              :1;
  U32 PowerManagement   :1;
  U32 PciMainInt        :1;
  U32 IopToPciInt       :1;
  U32 IopMainInt        :1;
  U32 PciAbort          :1;
  U32 PciReset          :1;
  U32 PciPME            :1;
  U32 PciENUM           :1;
  U32 IopBusTimout      :1;
  U32 AbortLSERR        :1;
  U32 ParityLSERR       :1;
  U32 RetryAbort        :1;
  U32 LocalParityLSERR  :1;
  U32 Reserved          :29;
} PLX_INTR, *PPLX_INTR;
```

Affected Register Location

E: Enable interrupt bit location

A: Active interrupt bit location

Structure Element	PCI 9080	PCI 9054	IOP 480
InboundPost	E 0x168, 4 A 0x168, 5	E 0x168, 4 A 0x168, 5	E 0x28, 4 A 0x28, 5
OutboundPost	E 0xB4, 3 A 0xB0, 3	E 0xB4, 3 A 0xB0, 3	E 0x30, 3 A 0x34, 3
OutboundOverflow	E 0x168, 6 A 0x168, 7	E 0x168, 6 A 0x168, 7	E 0x28, 6 A 0x28, 7
PciDmaChannel0 **** Note:*** *DMA Interrupt is routed to*	E 0x100*, 17 A 0xE8, 21	E 0x100*, 17 A 0xE8, 21	E 0x1B4, 8 A 0x1B0, 8

Structure Element	PCI 9080	PCI 9054	IOP 480
PCI interrupt.			
IopDmaChannel0	E 0xE8, 18 A 0xE8, 21	E 0xE8, 18 A 0xE8, 21	E 0x1BC, 8 A 0x1B8, 8
PciDmaChannel1 **** Note:*** *DMA Interrupt is routed to PCI interrupt.*	E 0x114*, 17 A 0xE8, 22	E 0x114*, 17 A 0xE8, 22	E 0x1B4, 9 A 0x1B0, 9
IopDmaChannel1	E 0xE8, 19 A 0xE8, 22	E 0xE8, 19 A 0xE8, 22	E 0x1BC, 9 A 0x1B8, 9
PciDmaChannel2	N/A	N/A	E 0x1B4, 10 A 0x1B0, 10
IopDmaChannel2	N/A	N/A	E 0x1BC, 10 A 0x1B8, 10
Mailbox0 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 24 A 0x1B8, 24
Mailbox1 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 25 A 0x1B8, 25
Mailbox2 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 26 A 0x1B8, 26
Mailbox3 * ***Note:*** *Only one bit enables Mailbox 0-3 Interrupts. No individual enabling of interrupts.*	E 0xE8, 3* A 0xE8, 28	E 0xE8, 3* A 0xE8, 28	E 0x1BC, 27 A 0x1B8, 27
Mailbox4	N/A	N/A	E 0x1BC, 28 A 0x1B8, 28
Mailbox5	N/A	N/A	E 0x1BC, 29 A 0x1B8, 29
Mailbox6	N/A	N/A	E 0x1BC, 30 A 0x1B8, 30
Mailbox7	N/A	N/A	E 0x1BC, 31 A 0x1B8, 31
IopDoorbell	E 0xE8, 17 A 0xE8, 20	E 0xE8, 17 A 0xE8, 20	E 0x1BC, 11 A 0x1B8, 11
PciDoorbell	E 0xE8, 9 A 0xE8, 13	E 0xE8, 9 A 0xE8, 13	E 0x1B4, 11 A 0x1B0, 11
SerialPort1	N/A	N/A	E 0x1BC, 4 A 0x1B8, 4
SerialPort2	N/A	N/A	E 0x1BC, 5 A 0x1B8, 5
BIST	A 0xE8, 23	A 0xE8, 23	E 0x1BC, 12 A 0x1B8, 12
PowerManagement	N/A	E 0xE8, 4 A 0xE8, 5	E 0x1BC, 13 A 0x1B8, 13
PciMainInt	E 0xE8, 8 A 0xE8, 13	E 0xE8, 8 A 0xE8, 13	E 0x1BC, 14 A 0x1B8, 14
IopToPciInt	E 0xE8, 11 A 0xE8, 15	E 0xE8, 11 A 0xE8, 15	E 0x1B4, 7 E 0x1BC, 7 A 0x1B8, 7
IopMainInt ***Note:*** *There is no master bit that shows that the IOP interrupt is active. Each interrupt trigger's active bit must be checked to determine the interrupt*	E 0xE8, 16	E 0xE8, 11	E 0x1BC, 0

Structure Element	PCI 9080	PCI 9054	IOP 480
source.			
PciAbort	E 0xE8, 10 A 0xE8, 14	E 0xE8, 10 A 0xE8, 14	Currently Not Defined.
PciReset	N/A	N/A	E 0x1BC, 6 A 0x1B8, 6
PciPME	N/A	E 0xE8, 4 A 0xE8, 5	E 0x1BC, 15 A 0x1B8, 15
PciENUM	N/A	N/A	E 0x1BC, 16 A 0x1B8, 16
IopBusTimout	N/A	N/A	A 0x1B8, 3
AbortLSERR	E 0xE8, 0 A 0x06, 12-13	E 0xE8, 0 A 0x06, 12-13	E 0x1BC, 1 A 0x06, 12-13
ParityLSERR	E 0xE8, 1 A 0x06, 15	E 0xE8, 1 A 0x06, 15	E 0x1BC, 2 A 0x06, 15
RetryAbort	E 0xE8, 12 A 0x06, 12-13	E 0xE8, 12 A 0x06, 12-13	Currently Not Defined.
LocalParityLSERR	N/A	E 0xE8, 6 A 0xE8, 7	Currently Not Defined.

Purpose

Structure containing the various PLX device interrupts that are used to return active interrupts or to enable or select certain interrupts.

Members

InboundPost
The value represents the messaging unit's inbound post FIFO interrupt.

OutboundPost
The value represents the messaging unit's outbound post FIFO interrupt.

OutboundOverflow
The value represents the messaging unit's outbound FIFO overflow interrupt.

IopDmaChannel0, IopDmaChannel1 and IopDmaChannel2
The value represents DMA channel interrupt on the IOP side.

PciDmaChannel0, PciDmaChannel1 and PciDmaChannel2
The value represents DMA channel interrupt on the PCI side.

Mailbox0, Mailbox1, Mailbox2, Mailbox3, Mailbox4, Mailbox5, Mailbox6, and Mailbox7
The value represents mailbox interrupt.

IopDoorbell
The value represents the PCI to IOP doorbell interrupt.

PciDoorbell
The value represents the IOP to PCI doorbell interrupt.

SerialPort1
The value represents the serial port 1 interrupt.

SerialPort2
The value represents the serial port 2 interrupt.

BIST
The value represents the BIST interrupt.

PowerManagement
The value represents the power management interrupt.

PciMainInt
The value represents the INTA interrupt line which is the master interrupt for all PCI interrupts.

IopToPciInt
The value represents the INTI interrupt line which is an input line for the IOP to trigger PCI interrupts.

IopMainInt
The value represents the INTo interrupt line which is the master interrupt for all IOP interrupts.

PciAbort
The value represents the PCI abort interrupt.

PciReset
The value represents the PCI reset interrupt.

PciPME
The value represents the PCI PME interrupt.

PciENUM
The value represents the PCI ENUM interrupt.

IopBusTimout
The value represents the IOP bus timeout interrupt.

AbortLSERR
The value represents the IOP LSERR interrupt caused by PCI bus Target Aborts or by Master Aborts.

ParityLSERR
The value represents the IOP LSERR interrupt caused by PCI bus Target Aborts or by Master Aborts.

RetryAbort
The value enables the PCI IC to generate a Target Abort after 256 Master consecutive retries to the target.

LocalParityLSERR
The value represents the IOP LSERR interrupt caused by Direct Master Local Data Parity Check Errors.

Reserved
This value is reserved for future definitions.

Comments

The PLX interrupt structure is used to return the active interrupts or to enable or select specific interrupts.

Power Level Enum Data Type

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

Affected Register Location

Not applicable.

Purpose

Enumerated type used to state the power level.

Members

D0Uninitialized
: Change to uninitialized state. This state is the full power state and is the state for normal operation.

D0
: Change to active state. This state is the full power state and is the state for normal operation.

D1
: Change to light sleep state. This state will allow the IOP to only PCI configuration accesses. Responses to other accesses are disabled. Other background tasks on the IOP may still be running such as monitoring a network. Function context should be maintained to support transitions back to the *D0* state.

D2
: Change to deeper sleep state. This state will allow the IOP to only PCI configuration accesses. Responses to other accesses are disabled. This state should consume less power than *D1* state. Function context should be maintained to support transitions back to the *D0* state.

D3Hot
: Change to power down state. This state prepares the IOP for a hot swap. Configuration space accesses must provide responses as long as the power and clock are supplied so that the software can change the power level state back to *D0* state. This state should consume very little power (while still connected to the PCI bus).

D3Cold
: Change to no power state. This states only supports bus reset. All context is lost in this state.

Comments

The power level enumerated type is used to control the transition between the power states.

Power Management Properties Structure

```
typedef struct _PM_PROP
{
  U32 Version                    :3;
  U32 PMEClockNeeded             :1;
  U32 DeviceSpecialInit          :1;
  U32 D1Supported                :1;
  U32 D2Supported                :1;
  U32 AssertPMEfromD0            :1;
  U32 AssertPMEfromD1            :1;
  U32 AssertPMEfromD2            :1;
  U32 AssertPMEfromD3Hot         :1;
  U32 Read_Set_State             :2;
  U32 PME_Enable                 :1;
  U32 PME_Status                 :1;
  U32 PowerDataSelect            :3;
  U32 PowerDataScale             :2;
  U32 PowerDataValue             :8;
  U32 Reserved                   :4;
} PM_PROP, *PPM_PROP;
```

Affected Register Location

Structure Element	PCI 9080	PCI 9054	IOP 480
Version	N/A	0x182, 0-2	0x342, 0-2
PMEClockNeeded	N/A	0x182, 3	0x342, 3
DeviceSpecialInit	N/A	0x182, 5	0x342, 5
D1Supported	N/A	0x182, 9	0x342, 9
D2Supported	N/A	0x182, 10	0x342, 10
AssertPMEfromD0	N/A	0x182, 11	0x342, 11
AssertPMEfromD1	N/A	0x182, 12	0x342, 12
AssertPMEfromD2	N/A	0x182, 13	0x342, 13
AssertPMEfromD3Hot	N/A	0x182, 14	0x342, 14
Read_Set_State	N/A	0x184, 0-1	0x344, 0-1
PME_Enable	N/A	0x184, 8	0x344, 8
PME_Status	N/A	0x184, 15	0x344, 15
PowerDataSelect	N/A	0x184, 9-12	0x344, 9-12
PowerDataScale	N/A	0x184, 13-14	0x344, 13-14
PowerDataValue	N/A	0x187, 0-7	0x347, 0-7

Purpose

Structure used to describe the Power Management characteristics.

Members

Version

This value represents the version of the *PCI Power Management Interface Specification.*

PMEClockNeeded

When this value is set to 1, the PCI IC indicates that the device relies on the presence of PCI clock for PME# operation.

DeviceSpecialInit
When set to 1, the PCI IC requires special initializatiion following a transition to D_0 uninitialized state.

D1Supported
When set to 1, the PCI IC supports the D_1 power state.

D2Supported
When set to 1, the PCI IC supports the D_2 power state.

AssertPMEfromD0
When set to 1, PME# can be asserted from power state D_0.

AssertPMEfromD1
When set to 1, PME# can be asserted from power state D_1.

AssertPMEfromD2
When set to 1, PME# can be asserted from power state D_2.

AssertPMEfromD3Hot
When set to 1, PME# can be asserted from power state D_{3hot}.

Read_Set_State
Power state of the PCI IC.

PME_Enable
When set to 1, PME# can be asserted.

PME_Status
Indicate the status of PME#. When set to 1 and *PME_Enable* is 1, PME# is asserted.

PowerDataSelect
This value selects which data the PCI IC will report through the Power Management Data Register.

PowerDataScale
This value sets the scaling factor to use when interpreting the value of the Power Management Data Register.

PowerDataValue
This value represents the power consumed or dissipated in various power states. The exact meaning of this value is selected by the *PowerDataSelect* field and is scaled by the *PowerDataScale* field.

Reserved
This value is reserved for future definitions.

Comments

The PM_PROP structure is used to describe the Power Management characteristics.

USER Pin Enum Data Type

```
typedef enum _USER_PIN
{
  USER0,
  USER1,
  USER2,
  USER3,
  USER4,
  USER5
} USER_PIN;
```

Affected Register Location

I represents input pins.

O represents output pins.

Structure Element	PCI 9080	PCI 9054	IOP 480
USER0	I 0xEC, 17 O 0xEC, 16	I 0xEC, 17 O 0xEC, 16	0x84, 0-3
USER1	N/A	N/A	0x84, 1 0x84, 4-6
USER2	N/A	N/A	0x84, 4 0x84, 8-10
USER3	N/A	N/A	0x84, 16
USER4	N/A	N/A	0x84, 17
USER5	N/A	N/A	0x84, 18

Purpose

Enumerated type used to select a USER pin.

Members

USER0
Select USER0 pin.

USER1
Select USER1 pin.

USER2
Select USER2 pin.

USER3
Select USER3 pin.

USER4
Select USER4 pin.

USER5
Select USER5 pin.

Comments

The USER pin enumerated type is used to select a USER pin.

Virtual Addresses Data Type

```
typedef struct _VIRTUAL_ADDRESSES
{
  U32 Va0;
  U32 Va1;
  U32 Va2;
  U32 Va3;
  U32 Va4;
  U32 Va5;
  U32 VaRom;
} VIRTUAL_ADDRESSES, *PVIRTUAL_ADDRESSES;
```

Purpose

This data type provides a list of User Virtual Addresses (UVA) for a PCI device. The UVAs correspond to the device's Base Address Registers (BAR)

Members

Va0
: UVA for BAR 0.

Va1
: UVA for BAR 1.

Va2
: UVA for BAR 2.

Va3
: UVA for BAR 3.

Va4
: UVA for BAR 4.

Va5
: UVA for BAR 5.

VaRom
: UVA for the Expansion ROM.

Comments

This data type contains the UVAs for all the BARs of a PCI device.