PCI Interconnect for CP-PDK

Application Note

March 2004

Contents

Figures

Tables

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	P. L. Srinivas
2.1	Updated for Release 2.1	December 2003	P. L. Srinivas

Part 1: Overview

1 Overview

This application note describes the high level design changes required in CP-PDK for supporting the bi-directional PCI communication between the forwarding plane software running on IXP2xxx family of network processors and control plane software running on IA-32/IXC processor. This document guides the customers on how to plug-in the existing PCI driver to the interconnect layer of CP-PDK.

1.1 Acronyms

The following table lists the terms used in this document and provides definition for each term.

Table 1. Terms and Acronyms

Acronyms	Description
AI	Angel Island
CC	Core Component
CE	Control Element
CPCI	Compact PCI
CP-PDK	Control Plane Platform Development Kit
DG	Datagram
FE	Forwarding Element
ForCES	Forwarding and Control Element Separation protocol
FPM	Forwarding Plane Module
NPF	Network Processing Forum
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card
PDK	Platform Development Kit
VIDD	Virtual Interface Device Driver

Part 2: CP-PDK Architecture Overview

2 CP-PDK Architecture Overview

The architecture of the Control Plane PDK defines three main components as shown in Figure 1:

- Control plane module
- Transport plug-ins
- Forwarding plane module

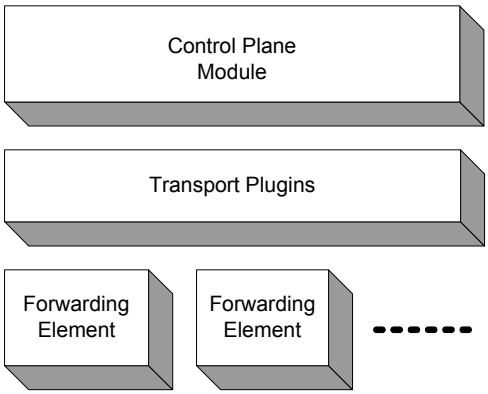


Figure 1: Control Plane PDK Architecture Components

The control plane and forwarding element(s) can have different communication mechanisms or protocols to exchange information with each other. These protocols can either be IETF standard protocols like ForCES or some proprietary implementation and so on. The planes can also be connected using different types of **interconnects**. Some examples of such interconnects are **InfiniBand, PCI, various back-plane switching fabrics and so on.**

The transport plug-in performs the following:

- Removes the type and details of the communication mechanisms from the rest of the PDK implementation
- Providesthe functionality required for separation of the control plane and forwarding plane(s).
- Enables plug-and-play functionality for different communication mechanisms with the rest of the PDK.

Thus, you can place different types of transport plug-ins between the planes to enable transparent communication between the control and forwarding Planes. This section describes the architecture for the transport plug-in.

The architecture of the transport plug-in is shown in Figure 2. The plug-in is composed of four distinct parts:

- FP plug-in API → This is an abstraction API that hides the transport plug-in details and presents a uniform API, which is invoked by the NPF API implementation modules on the control plane.
- Backend API → This API is exposed by the transport plug-in on the forwarding plane, and the the FP Module of the PDK uses it.

- Transport protocol → This is the standard or proprietary protocol such as IETF's ForCES, used to exchange information between the planes and it consists of two agents:
  - Control plane agent → Part of the transport protocol that resides on the control plane and communicates with the FP agent
  - Forwarding Plane Agent → Part of the transport protocol that resides on the forwarding plane and communicates with the CP agent
- Interconnect abstraction layer → This abstraction layer hides the interconnect details and is used by the transport protocol to send and receive messages without knowing whether the **interconnect is PCI or Infiniband or Ethernet**.

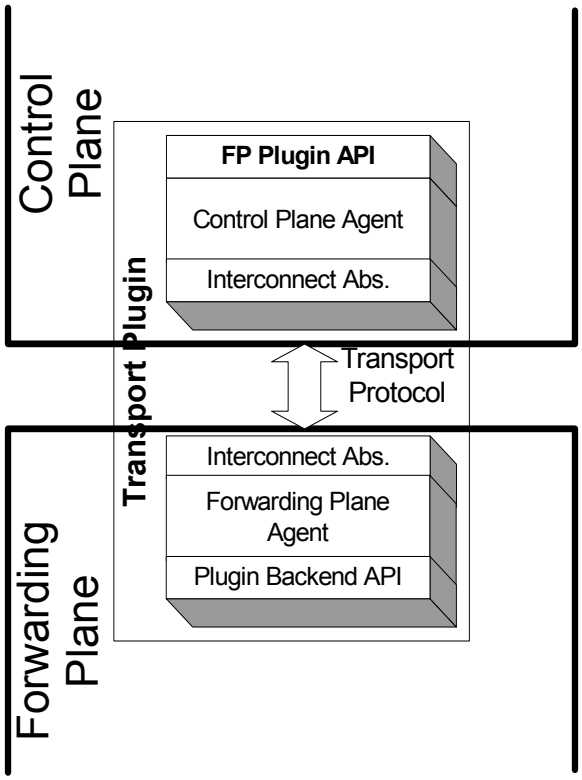


Figure 2: Transport Plug-in Architecture

Refer to CP-PDK software architecture document for more details about the transport plug-in. The interconnect layer, control plane agent and forwarding plane agent are not needed in co-located CP-PDK configuration as both the control plane and forwarding plane software runs on the same processor. Refer to co-located transport plug-in design document for details.

The interconnect layer mainly consists of two parts:

- Packet buffer management
- Datagram API

If the interconnect is changed from Ethernet to PCI, the packet buffer management APIs remain the same, while the datagram APIs must be mapped to the new interconnect layer.

Part 3: PCI Interconnect Details

3 PCI Interconnect Details

This application note proposes to have two software drivers for the PCI interconnect implementation. These two drivers must be installed on both the forwarding plane and control plane machines. Both the drivers together form the bi-directional communication system between the control plane and the forwarding plane. They both have the same design and perform the same functions but are on the opposite sides/interfaces of the PCI-to-PCI bridge, for example, i21555 or i21154.

Some of the design goals for the PCI Datagram driver (PCI DG) are as follows:

- To exist as an independent PCI device driver in the Linux kernel instead of being the network or character driver.
- To expose an abstract Datagram API, that can be used by other drivers such as network or character drivers for bi-directional communication. These datagram APIs will remain same irrespective of whether we use cPCI backplane interface or PrPMC module for PCI communication. Refer to Appendix D for details on cPCI and PMC interfaces in IXDP2400.

The network driver model makes the design simpler and very few changes are needed in the existing CP-PDK code base and the TCP/IP stack overhead follows this approach. Since the character driver model does not have the overhead of traversing the TCP/IP stack in the Linux kernel, it is also optimized.

Figure 3 shows how the PCI DG drivers fit in with the transport plug-in architecture. Another detailed figure is shown in Appendix B.

Appendix B describes the network driver abstraction for the driver and Appendix C describes the character driver abstraction.

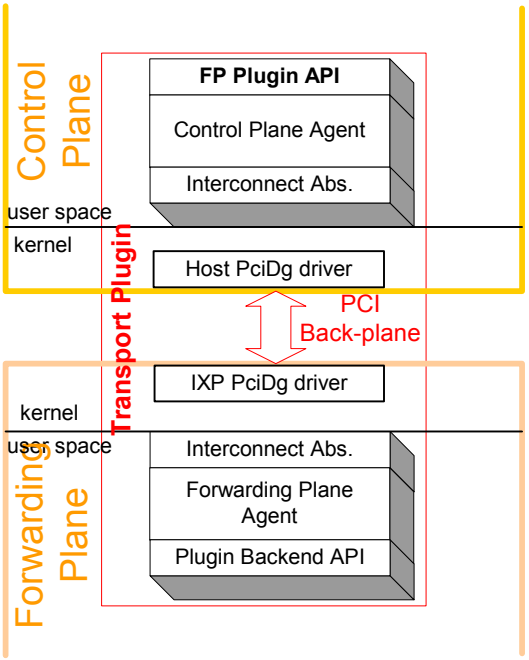


Figure 3: Transport Plug-in Architecture with the PciDg drivers

3.1 Flow of Control and Data Packets

The packets that are destined for protocols/applications on control plane must be transported from the forwarding plane. In the same way, the packets from the control plane that are transmitted to the virtual interfaces and which should be sent to the corresponding physical interfaces on the forwarding plane must be transported to the forwarding plane. Such packets are termed as *data packets*.

The packets that are transported to the forwarding plane through the transport plug-in and NPF API invocations are termed as control packets as these are the request/response invocations that control the underlying forwarding plane.

Figure 4 shows the path of data and control packets from CE to FE and vice versa. If the interconnect is Ethernet, then the control packets are sent through the transport plug-in and the data packets are sent using raw sockets (protocol id 105) from CE to FE and vice versa. This means there is a separate path for data and control packets.

The data packets are sent using IP-in-IP tunneling. The IP-in-IP tunneling is the standard mechanism used for tunneling IP packets. The IP-in-IP tunnel adds an additional IP header that identifies the new destination of the packet. For instance the control plane becomes the destination if it is an incoming packet being forwarded by the forwarding plane.). The corresponding ends of the tunnel compose or interpret the headers.

If the interconnect is changed between CE and FE, then the part of VIDD implementation that provides abstraction of the multiple forwarding plane interfaces as virtual interfaces to the control plane will not undergo any changes.

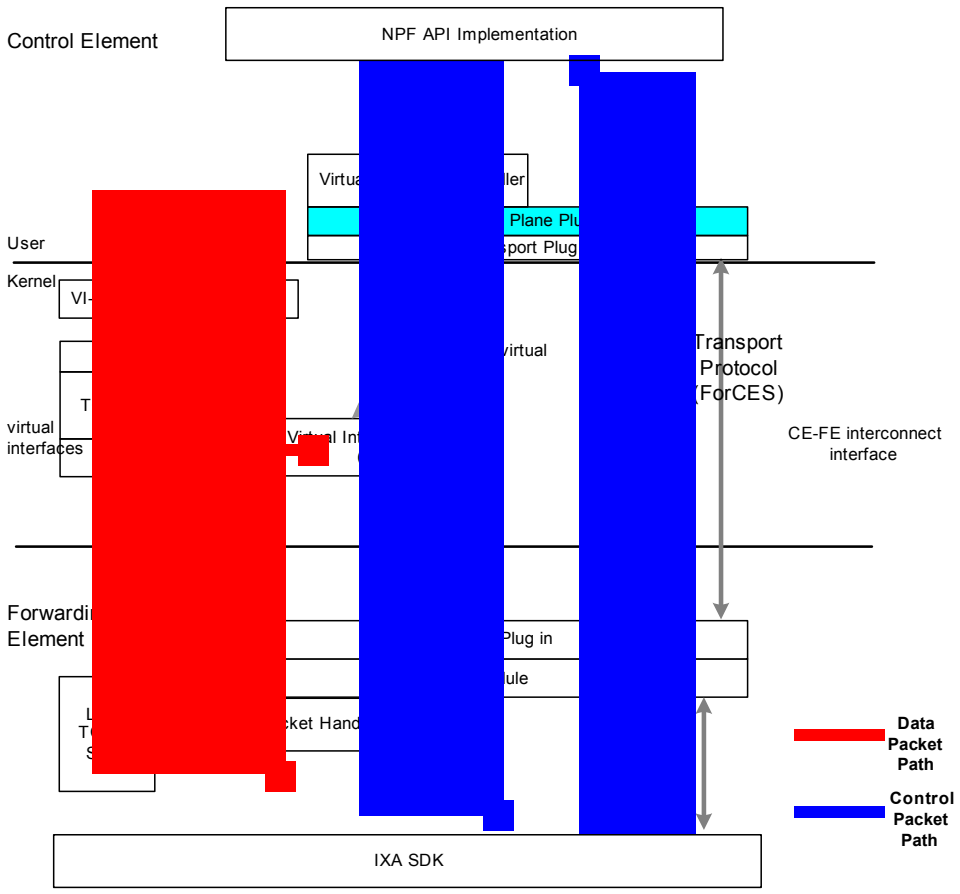


Figure 4: Flow of Control and Data Packets for Ethernet Interconnect

The main modules that are affected with the interconnect change are:

- FE packet handler
- CE packet handler for VxWorks
- VIP tunneling module for Linux
- Datagram APIs of interconnect layer

The socket interface (stream sockets for control packets and raw sockets for data packets) is used in the current implementation for Ethernet interconnect. Hence if the network driver approach is adopted you do not have to make any changes in these modules for PCI interconnect as the network driver allows opening the sockets over the PCI network driver.

The following sub-sections make reference to the PCI datagram APIs. Refer to the sample PCI datagram APIs in Appendix A for details.

If you do not adopt the network driver approach you must make changes in all the above-mentioned modules to map the datagram APIs of interconnect layer to the PCI DG API calls. Some of the

following changes must be done in various modules if the network driver model approach is not adopted:

- The IP-in-IP tunneling is not used for data packets.
- The original data packet along with the VIP header or meta data (portId, length) is sent over the PCI.
- Do not encapsulate the data packet in another IP header.
- The VIP module should be changed so that the VIP tunnel IP header is not included prior (pre-pended) to the IP packet. The VIP module tunnel transmit function internally uses the PCI datagram send function.
  The message type (param in pci_sendDatagram) distinguishes whether the message is a control packet or a data packet. You need to make the required changes to CE packet handler also as the IP-in-IP tunneling is not used.
- The portion of VIP module that handles the creation/deletion of virtual interfaces on the control plane remains the same. There will be changes to FE packet handler on the forwarding plane side.

3.1.1 FE Packet handler Flow

The packets entering the FE from the network and that are destined for the control plane enters the packet handler through the FP module Core Component. The packet handler registers a callback with the FPM CC in order to receive these packets by calling the function ix_cc_fpm_register_pkt_hdlr_cb. The FE packet handler receives the packets coming from the control plane that are destined for network.

3.1.1.1 Incoming Packets

Incoming packets follow the following sequence of steps as they traverse through the FE:

1. The FPM CC receives a packet from mostly from the stack driver) and passes it to the registered callback function. This callback function is passed bladeId, PortId, length and data buffer.
2. The packet handler invokes the `pci_sendMessage` function call to send the data packet to the CE. The VIP headers such as portId, length alone are added to the packet. The packet is not encapsulated in another IP packet.

3.1.1.2 Outgoing Packets

The outgoing packets are received from the control plane and passed to the FPM CC using the following sequence of events:

1. The PCI DG network driver receives the incoming packet by the invocation of InMessage call. The VIP header is stripped off of the packet.
2. The portId is determined from this VIP header.
3. The resulting packet is given to the FPM CC through the function ix_cc_fpm_sync_send_packet along with the portId.

3.1.2 CE Packet Handler (VxWorks)

The CE packet handler registers the FE bind/unbind events with the FP Plug-in in case of Ethernet interconnect, but it does not do so in case of PCI interconnect. When the PCI datagram driver receives the data packet from the FE, it checks the message type. Refer to Appendix E for the format of the message to be transported over the PCI interconnect of the packet. If it is in a data packet format, it is sent to the CE packet handler.

The CE packet handler extracts the VIP header from the data packet and the packet is converted to the MBLKs. The packet is then sent to the VIDD implementation of VxWorks by calling the function `pdk_vidd_receive_pkt()`. There might be some requirement to maintain the mapping between the PCI peerId and FE id in this case. The `pdk_vidd_npt_send` function is called for the outgoing packets and from this function you need to call `pciDg_sendMessage` function instead of `ph_write_packet`. The parameter message type in the `pciDg_sendMessage` API invocation should indicate that this is a data packet.

3.1.3 VIP Tunneling Module (Linux)

The various IOCTL calls implemented by the VIP tunneling module to add the IP tunnel, delete IP tunnel interface on the control plane remains same even if the interconnect is changed between the CE and FE. The PCI DG driver on receiving packet with message type as data packet sends the packet to VIP module.

The VIP module extracts the portId from the VIP header of the packet and sends the packet to the networking stack for further processing. The VIP module's packet transmit function calls the `pciDg_sendMessage` function to send the data packet to the forwarding plane for outgoing packets. This means the `pciDG_sendMessage` is called from multiple places for control and data packets. Therefore, some synchronization mechanism must be built in the PCI DG driver itself so that no two threads simultaneously call the `pciDg_sendMessage` function.

Appendix A: PCI Datagram API

This appendix provides details of the sample PCI datagram APIs. The PCI Datagram API is used by other modules or drivers in the Linux kernel or VxWorks* such as the network driver and character driver (\Refer to appendix B and C for details. These drivers register as clients of the PciDg API and use the facilities provided by the PciDg driver to read and write messages across the i21555/i21154.

1. Struct PciDg_Client used for callbacks in PCI Datagram API

Syntax

```
typedef struct {
void (*connReady)(void *context, int id);
void (*messageIn)(void *context, int srcId, int type, const char
*data,
                int dataLen);
void (*peerReady)(void *context, int id);
void (*peerGone)(void *context, int id);
void *context;
} PciDg_Client;
```

Description of fields

1. VOID (*CONNREADY)(VOID *CONTEXT, INT ID);

This function is called when the client of PCI DG driver layer that is the PCI network or character device driver is ready to send and receive This is called before your first `messageIn` call. It is OK to leave this null.

Parameters

context - A parameter specified at open time

id - Your own ID in this network

2. void (*messageIn)(void *context, int srcId, int type, const char *data, int dataLen);

This function is called whenever a message is received from another node on the network. You must declare your own callback function before you open the PCI Datagram interface. The PciDg driver owns the data buffer in this call.

Parameters

context - A parameter specified at open time

srcId - The ID of the source processor

type - The message type. The type of message is not used in the PCI network driver model but it can be used in some alternate design where the PCI DG user wants to put the message type as control or data packet

data - A buffer holding the data. When you return from the callback, this buffer is re-used. So if you want to keep the data, you must copy the data. This is 8-byte aligned.

dataLen - The length of the data in the buffer

3. void (*peerReady)(void *context, int id);

This function is called when a new system has shown up on the pciDg network. This function is called before messageIn receives a message from the new node. It is OK to leave this null.

Note: This remote node need not have its client registered. The remote node is connected and has its pciDg driver running.

Parameters

- context - A parameter specified at open time
- id - The ID of the node that is added to the system recently

4. void (*peerGone)(void *context, int id);

This function is called when a new system has left the pciDg network. You will not receive messageIn calls after the receipt of this call. It is OK to leave this null.

Parameters

- context - A parameter specified at open time
- id - The ID of the node that was just removed from the system

5. void *context;

This argument is passed into all callbacks.

2. Registering Client for PCIDG API

Syntax

```
int pciDg_registerClient(const PciDg_Client *client);
```

Description of function

Opens the PCI Datagram interface. If you have received any messages before opening, then the callback is called for each of these messages before the return of the open. If the pciDg network is already set up, then calls to both client->connReady and client->peerReady arrives before you return from this function.

Parameters

client - Your client state. This client state is copied into a buffer in the pciDg system and you can free this after return.

Return Value

0 – If success

EBUSY - If there is a client registered already.

3. De-registering Client for PCIDG API

Syntax

```
void pciDg_unregisterClient(void);
```

Description of function

Unregisters a client. Another module can register as a new client after calling this

4. Send Message Call

Syntax

```
int pciDg_sendMessage(int peer, int type, const char *data, int
dataLen, int waitOk);
```

Description of function

Sends a datagram to peer. If the peer's ring buffer is full, then the datagram is not sent and it returns one of the following errors:

- non-blocking call if waitOk is 0 ;
- Waits until the peer reads enough data from the buffer to fit the new message, if the waitOk is nonzero (blocking call)

If you use this function from interrupt level you must set the waitOk to 0. The client of the PciDg driver/caller owns the data buffer in this call and is responsible to free it.

Parameters

peer - The destination node

type - The message type

data - The data buffer

dataLen - The number of bytes in the data buffer

waitOk - If zero, do not wait for flow control. If one, wait if needed

Return Value

0 - Message sent successfully.

-EWOULDBLOCK - Recipient did not have enough buffer space for message and waitOk was 0

-ERESTARTSYS - Recipient did not have enough buffer space for message. waitOk was 1, but another signal arrived before interrupt and the data could not be transferred.

-ENODEV - The peer indicated by **dest** is not ready to accept the data currently. It denotes that the destination is yet to finish its booting.-ENOSPC - The message you are trying to send is bigger than the MTU -EINVAL - If **dest** is your own ID

Appendix B: Network Driver Abstraction

This appendix describes the PciDgNet driver. The PciDgNet driver provides the network driver abstraction for the PCI Datagram driver in the Linux kernel. It makes use of the module stacking facility provided in the Linux kernel that allows drivers to communicate with each other using well-defined APIs that are exported to the kernel. The PciDgNet driver uses the PCI Datagram API exported by the PciDg driver to communicate with it.

The PciDg driver must be successfully installed in the kernel before the installation of the PciDgNet driver. The PciDgNet driver provides an Ethernet driver abstraction for the PciDg driver to the Linux kernel. Thus for the user space applications/tools, it looks like any other Ethernet device installed on the system. The **ifconfig** command can be used to configure an IP address for the Ethernet device.

Figure 5 shows how the PciDgNet driver, PciDg driver, and the rest of the Linux kernel (TCP/IP stack) fit in with the transport plug-in architecture defined by the CP-PDK software architecture specification document. The PciDgNet driver is currently used for both the transport plug-in (since it does not require any modification to the interconnect abstraction layer) and VIDD in the CP-PDK.

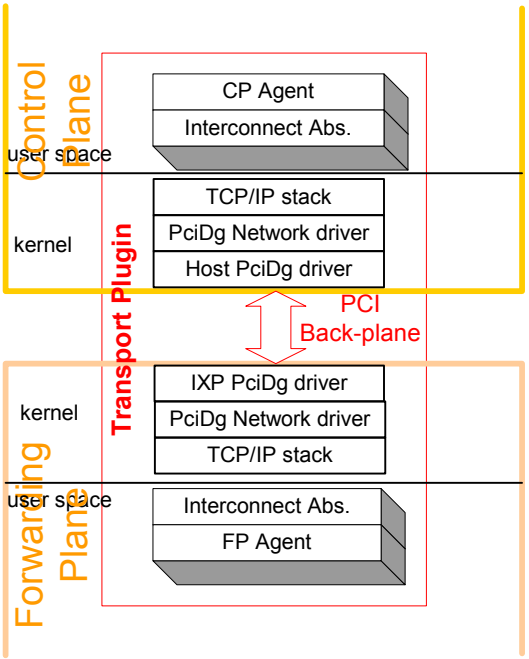


Figure 5: Transport Plug-in Architecture with the PciDg drivers

Design Overview

The PciDgNet driver uses the picDg_registerClient() call to register itself with the PciDg driver during initialization. Then it uses an obsolete Ethernet vendor address (AA 00 00 00 00) as the first five octets of the address for the device. The last octet of the address is uniquely assigned for each device and it is based on the unique Id that is passed to the driver by the PciDg driver using the ConnReady() callback function. Thus the driver uses the last octet of the Ethernet address to identify the peer to which the data is to be sent. The driver also uses ARP just like any other Ethernet driver to resolve between IP and MAC addresses.

The PciDgNet driver first identifies the peer based on the last octet in the Ethernet address for transmitting a data packet. It then calls the pciDg_sendMessage() function of the PciDg driver which takes care of sending the packet to the peer. The PciDg driver uses the MessageIn() callback function to send the packet to the PciDgNet driver during the packet reception. The PciDgNet driver then allocates the socket buffer and copies the packet to it before sending it to the TCP/IP stack in the kernel. The driver maintains the net_device_stats structure to report the statistics for the driver.

The PCIDgNet and PCI DG are kernel loadable modules in case of Linux. Therefore, their corresponding initialization routines are called at the time these modules are inserted into the kernel. In case of VxWorks, their initialization routines are called from the CP-PDK initialization routines.

The interactions between the TCP/IP stack in the Linux kernel, PciDg Network driver and the PciDg driver are illustrated in Figure 6.

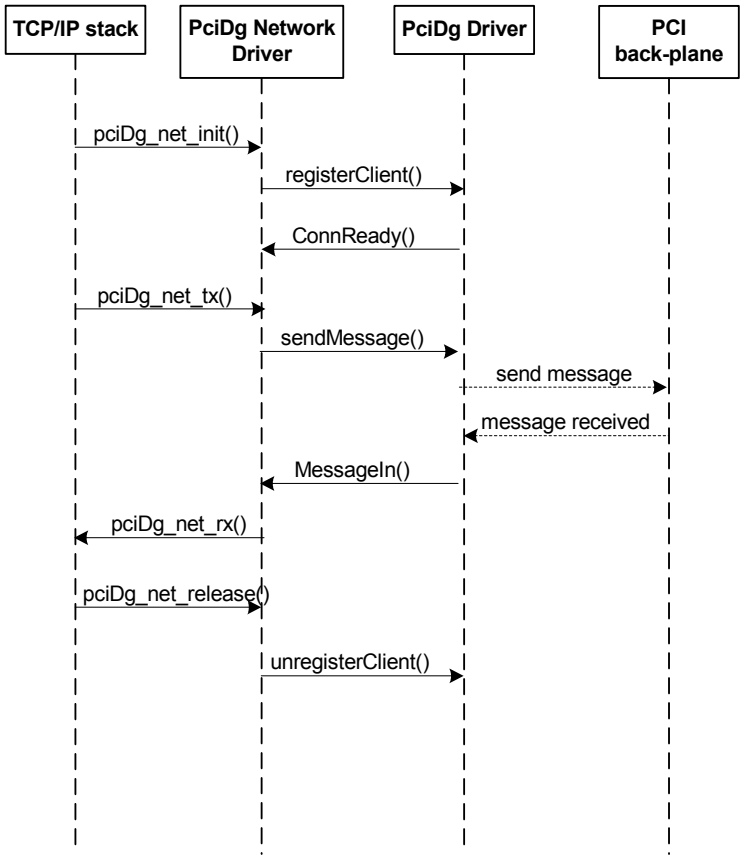


Figure 6: Interaction between the TCP/IP stack, PciDg Network driver and PciDg driver

Appendix C: Character Driver Abstraction

This appendix describes the PciDgChar driver. The PciDgChar driver provides the character driver abstraction for the PCI Datagram driver in the Linux kernel. It also makes use of the PCI Datagram API exported by the PciDg driver to communicate with it.

The PciDgChar driver should be installed after the installation of PciDg driver in the kernel. The mknod system call is used to create an entry in the /dev directory for the PciDgChar driver and associate a name with the driver's major and minor number.

The following file operations can be performed on the PciDgChar device by programs in the user space. It supports both blocking and non-blocking I/O.

```
int         open (const char *pathname, int flags);
int         close (int fd);
ssize_t     read (int fd, void *buf, size_t count);
ssize_t     write (int fd, void *buf, size_t count);
int         ioctl (int fd, int cmd, ….);
```

The ioctl operations supports three commands that are used to provide TCP/IP socket-like behavior by the character driver and, which is required for the COPS portability layer (Interconnect abstraction layer in the transport plug-in architecture). The ioctl commands are CONNECT, DISCONNECT and ACCEPT and it is similar to the connect(), accept() socket calls.

Design Overview

During initialization, the PciDgChar driver uses the pciDg_registerClient() call to register itself with the PciDg driver. The driver uses the minor number of the device to identify the peer with which it is associated. It has an array of device specific structures that are used to maintain information about each device (with different minor number) associated with the driver.

The driver first copies the user space buffer into an internal kernel buffer during a write operation. Using the minor number of the device it then identifies the peer to which the buffer should be sent and calls the pciDg_sendMessage() function of the PciDg driver which sends the buffer.

When the PciDg driver receives a buffer it uses the MessageIn() callback to send it to the PciDgChar driver. The PciDgChar driver stores it in an internal read buffer for that particular peer. When a read operation is performed on the driver, it copies the data from the internal read buffer for that device (minor number) and passes it back to the user. Appropriate locking mechanisms are used to avoid race conditions.

The interactions between the User space program, PciDg Character driver and the PciDg driver are illustrated in Figure 7

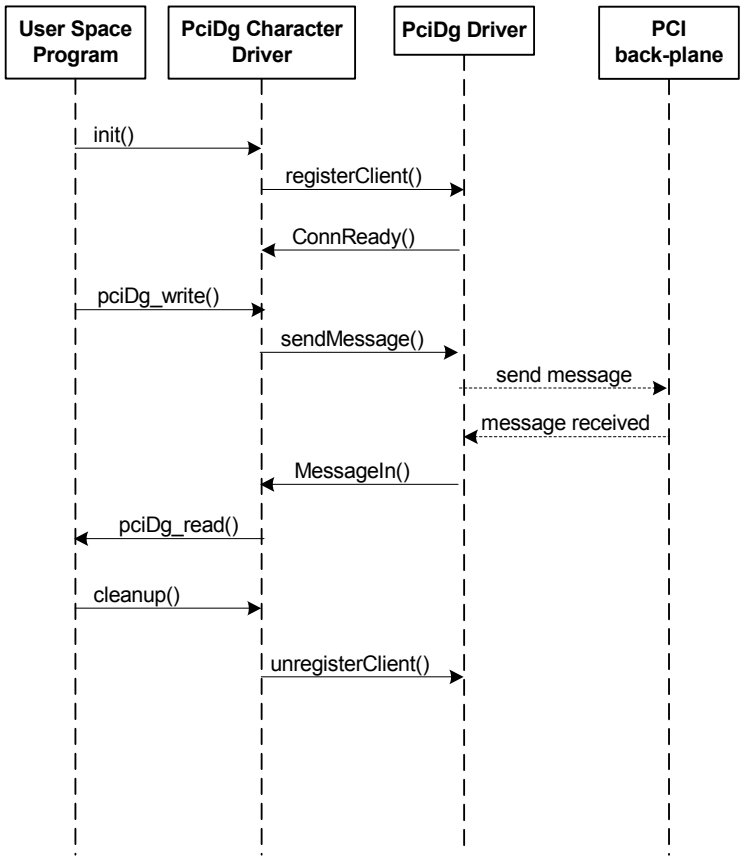


Figure 7: Interaction between the User space program, PciDg Character driver and PciDg driver

Appendix D: IXDP2400 System Overview

The IXDP2400 development platform (Angel Island) is an evaluation development platform for IXP2400. The IXMB2400 base card is the main board for IXDP2400. The IXMB2400 base card consists of two IXP2400 units arranged in a pipelined ingress-egress fashion for optimum performance of full duplex OC-48 data throughput rates. The IXMB2400 base card is housed in a chassis. The chassis supports a single IXMB2400 base card with a media card and a switch fabric interface card.

PCI subsystem in Angel Island

The following diagram is the high level representation of PCI subsystem in an Angel Island development platform. There are two options for connecting the control plane processor system (running the control plane software) to the Angel Island system (running data plane software) through the PCI interconnect:

- cPCI backplane interface
- PMC slot interface

In case the control plane processor system is connected through the cPCI backplane, the communication between the Ingress side XSCALE processor (of Angel Island) and the control plane processor will be through i21555 non-transparent PCI-PCI bridge.

Note: In case of CP-PDK, the control plane needs to directly communicate with the ingress side XSCALE processor only through the PCI interconnect. The control plane processor will never communicate directly to the egress side XSCALE processor. The communication between the ingress XSCALE and egress side XSCALE processor occurs using the resource manager framework as provided in the IXA SDK.

If the control plane processor system is connected through the PMC slot interface then the communication between the Ingress side XSCALE (of Angel Island) and the control plane processor system (that is PrPMC module) will be through the i21154 transparent PCI-PCI bridge. The low level PCI device driver design will vary depending upon the type of PCI-to-PCI bridge that is used for communication between the data plane and control plane processor.

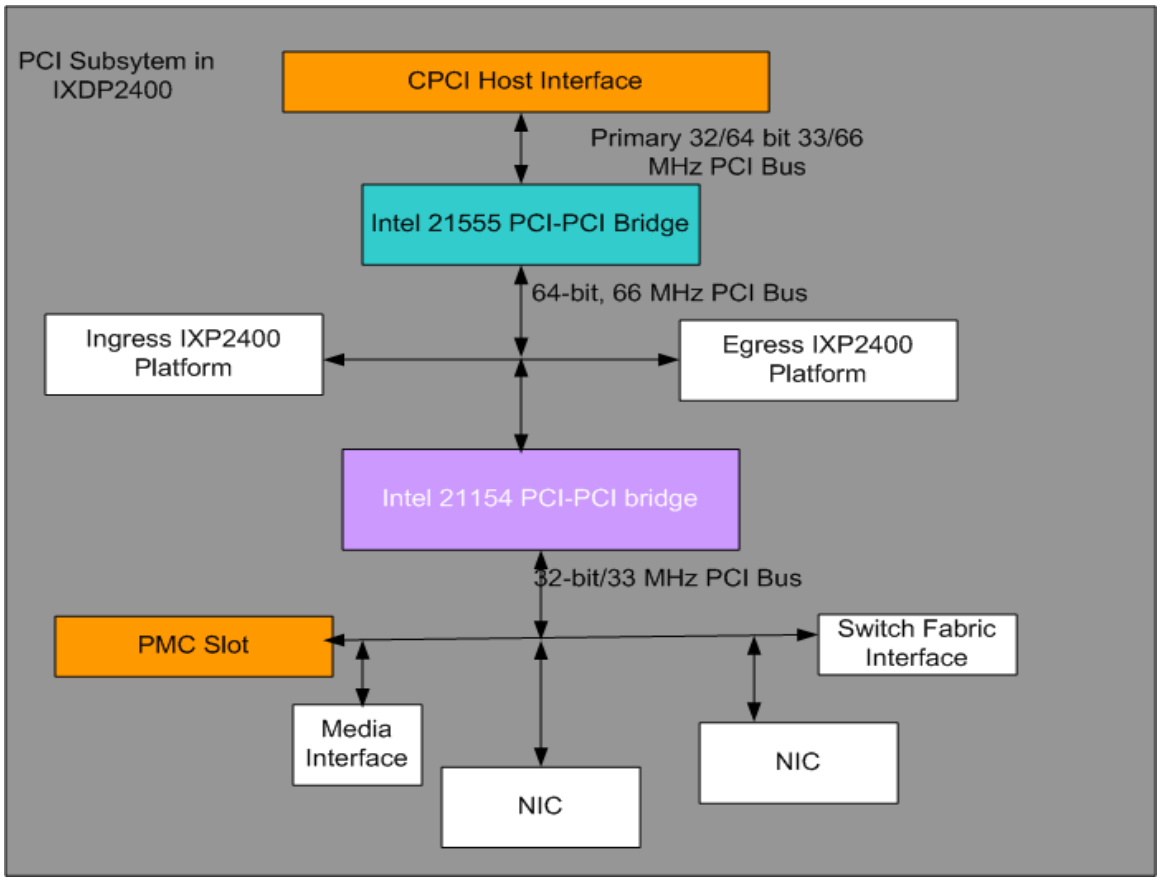


Figure 8: PCI Subsystem in Angel Island

The PCI Datagram driver enables the bi-directional PCI communication between the Angel Island (AI) platform running Linux/VxWorks and the Host PC through the i21555 non-transparent PCI-to-PCI bridge or i21544 transparent PCI-to-PCI bridge. Both the Transport Plugin and the VIDD modules of the NPF PDK use the PCI Datagram driver.

Intel 21555 Non-Transparent PCI-to-PCI Bridge

The i21555 is a non-transparent PCI bridge. It is termed as non-transparent bridge because of the following reasons:

- Although the i21555 connects two PCI busses together, the two busses are not combined into a single logical PCI bus (that is the CPU devices cannot be directly accessed on either bus by the CPU). Instead, the i21555 requires a CPU on one bus to explicitly send bus requests through the i21555 in order to access devices on the second bus.

The i21555 architecture consists of the following functional blocks:

- Data buffers
- Registers
- Control logic

The **Registers** is the main functional block that is used by the driver. The registers consist mainly of the standard PCI configuration space registers and the memory or I/O mapped control and status registers (CSR).

The x86 host processor is on the primary bus and the IXDP2400 is on the secondary bus on the Angel Island system used by the PciDg implementation

Intel 21154 Transparent PCI-to-PCI Bridge

The i21154 is a second-generation PCI-PCI bridge and is fully compliant with the electrical and protocol requirements of the PCI local bus specification revision 2.2, and the PCI-to-PCI bridge architecture specification revision 1.0.

The i21555 functionally is similar to the i21154 transparent PCI-PCI bridge, as both provide a connection path between the devices attached to two independent PCI buses.

The key difference in a i21154 while comparing with the i21555 is the presence of a transparent bridge in the connection path between the host processor and a device, and the bridge is transparent to devices and device drivers, while it is not so in the case of i21555. Since the i21555 is non transparent, the device driver for the add-in card must be aware of the presence of the i21555 and manage its resources appropriately.

The i21555 allows the entire subsystem to appear as a single virtual device to the host. This enables configuration software to identify an appropriate driver for the subsystem. Once the transparent PCI-PCI bridge is configured during system initialization, it operates without the aid of a device driver.

The transparent bridge does not require a device driver of its own, as it does not have any resources that must be managed by the software during run-time. The driver need not know about the presence of the bridge and manage its resources with a transparent bridge. The subsystem appears to the host system as individual PCI devices on a secondary PCI bus and not as a single virtual device.

Appendix E: PCI Driver Design Details

Initialization

The initialization of the driver is different depending upon whether transparent or non-transparent PCI-PCI bridge is used for communication.

Communication Overview

The PciDg driver creates a bi-directional, datagram-like communication system. The messages sent from one CPU is delivered to the callback on the second CPU with perfect guaranteed delivery. If one CPU is sending messages faster than the capacity of the recipient, then the sender may choose either to wait for the recipient to catch up or have the message lost. This is handled using the waitOk parameter as described in section 5.1.4.

The communication is built out of two unidirectional communication systems (one going in each direction) thought they are bi-directional. It is easier to consider any one of these systems to explain the design.

The communications system consists of a ring buffer, two counters, and a flag. This data structure is called the **CommWindow** and the driver allocates it during initialization. The four parts of the structure are:

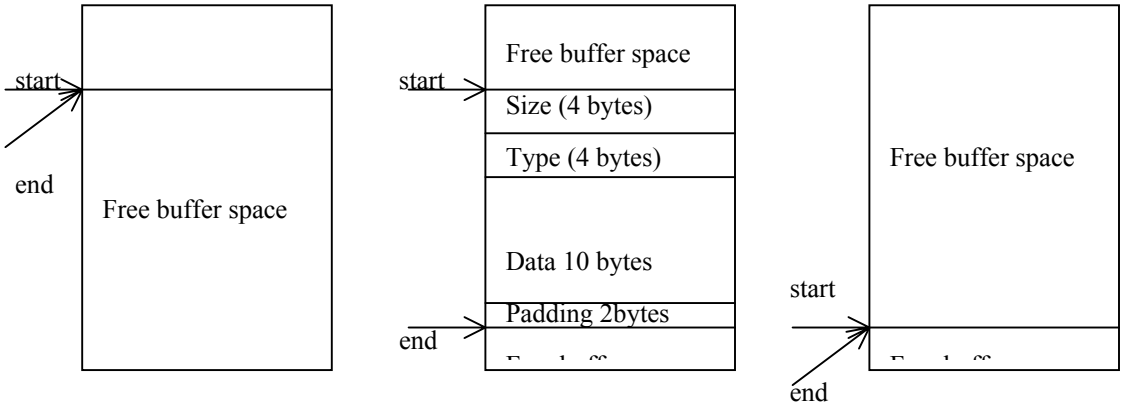
- `buffer`: A ring buffer. Messages are written here by the sender and read by the receiver.
- `dataStart`: A counter. This is the index (in the buffer) of the next message that the receiver should pick up. Every time the receiver picks up a message, the size of this message is added to the current `dataStart` value. This is never written by the sender.
- `dataEnd`: A counter. This is the index (in the `buffer`) of the first unused byte after the last message. Once the sender adds a message to the ring buffer, this is incremented. This is never written by the receiver. When `dataStart` is equal to `dataEnd`, the buffer is empty. Note that the pciDg code never fills the buffer totally as this would make `dataStart` and `dataEnd` be equal and which can be confused to an empty buffer.
- `waitingForRecv`: A flag. This flag is set when the sender is waiting for the receiver to pick up a message (thus freeing more space in the buffer). The receiver checks this flag after picking up a message and updating the `dataStart` counter. An interrupt is delivered to the sender if this flag is set.

Messages are put into the ring buffer with an 8-byte header prepended. The first four bytes are the size of the message body in little endian; the second four bytes are the type of the message in little endian. The **type** of the message is defined/passed by the client of the PciDg driver using the PciDg API, described in Appendix A.

Once a message is placed in the buffer, `dataEnd` is adjusted to point to the first 4-byte boundary after the end of the messages. Thus, messages have no restriction on alignment or length, but the header of each message is always 4-byte aligned. For example 11 bytes message consume 20 bytes in the buffer (4 bytes for length + 4 bytes for message type + 11 bytes of message + 1 byte of padding for alignment of the next message).

The message length **-1** is a special case. Messages are never put in the buffer **wrapped**. The message length of -1 is written when the entire message cannot fit in the ring buffer without having to split the message at the end of the buffer. This indicates that the next message will begin at byte 0 of the ring buffer. The sender sets `dataEnd` to 0 after writing a -1 length to the ring buffer. The receiver sets

dataStart to 0 after reading a -1 length from the ring buffer. A usage scenario for the communication system is shown in Figure 9.



1. An Empty Buffer 2. After writing 10 bytes 3. After reading 10 bytes

Figure 9: Usage Scenario of the communication system

The doorbell interrupts and mailbox registers can be used to synchronize the transfer of data between the two processors. The doorbell register is used to interrupt the processor and inform that there is a data to receive. The mailbox register is used to send the address of the data buffer from where the data should be copied.

Race Conditions

Whenever there are two processors sharing a data structure, you must always consider race conditions. The PciDg driver is no different. Follow the below-mentioned rules throughout the design to make the race conditions solve easier:

- Each of the four elements of the communications structure (the ring buffer, the counters, and the flag) is written by one processor only. This eliminates all problems involving write atomicity.
- The counters and the flag are all four bytes long and four bytes aligned. This means that a single memory access across the PCI bus will always fully update the element, so that if one processor reads and another processor writes, the reader is guaranteed to get the value before or after the write, and never a value that is half-before and half-after the write.
- The `dataStart` and `dataEnd` counters control which parts of the ring buffer are accessible by each processor. The sender will only be writing to the portion of the ring buffer after the `dataEnd` counter and before the `dataStart` counter; the receiver will only be reading from the portion of the ring buffer after `dataStart` and before `dataEnd`. Thus there will never be a case of both processors accessing the same part of the ring buffer at the same time.

Performance Considerations

The basic principle used to optimize performance is to do local reads and remote writes. Each communications system is split so that all writes are done to memory across the i21555 and all reads are done from local memory. This gives the best performance, as reading from across the PCI bus often requires the read to complete before the processor can continue. But writes to the PCI bus can often be stored in a buffer on the processor, the bus controller, or the i21555 while the processor continues to execute.

Variable/Data Structure Nomenclature

Each processor's driver has a pointer to two **CommWindow** structures. Each contains one half of the elements necessary for bi-directional communication. These structures are called **`localWindow`** (for the comm window that is in local memory) and **`peerWindow`** (for the comm window that is in the other processor's memory). Each comm window contains:

- `buffer`: A ring buffer. You write to `peerWindow`'s ring buffer to send, and to receive you read from `localWindow`'s ring buffer.
- `peerDataStart`: The `dataStart` field for the buffer in the other comm window – that is if you are receiving you read from `localWindow`'s `buffer` and then update `peerWindow`'s `peerDataStart` field.
- `localDataEnd`: The `dataEnd` field for the buffer in this comm window - if you are sending you write data to `peerWindow`'s `buffer` and then update `peerWindow`'s `localDataEnd` counter.
- `flags`: A field. The only flag defined currently is the flag which indicates that the processor across the i21555/i21154 PCI-PCI bridge is waiting for a response to free up space in the comm window.

This naming gives best performance by optimizing reads to be local (that is, not going across the PCI busses) and writes to be remote. It is important to remember that the peer of a peer is local. So, `peerWindow->peerDataStart` refers to first byte of the next message to receive in `localWindow->buffer`.

Missing features

Thread Safety

Currently, it is assumed that only one sender is in action at a time. If two kernel threads were to call the send function simultaneously, there would be many possible race conditions and problems, so modules using the PciDg system must implement their own locks. It would be better to make the PciDg API thread-safe.

Multiple Callbacks

The current design assumes to have one callback. This makes it impossible to have two modules using PciDg simultaneously. It is desirable at some point to have multiple modules moving different types of traffic over a single PciDg system at once, with the messages types differentiating the callbacks.