PCI Express® System Interconnect Software Architecture for x86-based Systems

Application Note AN-571

Notes

By Kwok Kong and Alex Chang

Introduction

A multi-peer system using a standard-based PCI Express® multi-port switch as the System Interconnect was described in an IDT white paper by Kwok Kong[1]. That white paper described the different address domains existing in the Root Processor and the Endpoint Processor, memory map management, enumeration and initialization, peer-to-peer communication mechanisms, interrupt and error reporting, and possible redundant topologies. Since the release of the white paper, IDT has designed and implemented a multi-peer system using the x86 based system as the Root Processor (RP) and Endpoint Processor (EP) connecting through IDT's PES24NT3 NTB port and IDT's PES64H16 device as the multi-port PCIe® switch for the System Interconnect. This application note presents the software architecture of the multi-peer system as implemented by IDT. The architecture may be used as a foundation or reference to build more complex systems.

System Architecture

A multi-peer system topology using PCIe as the System Interconnect is shown in Figure 1. There is only a single Root Processor (RP) in this topology. The RP is attached to the single upstream port (UP) of the PCIe switch. The RP is responsible for the system initialization and enumeration process as in any other PCIe system. A multi-port PCIe switch is used to connect multiple Endpoint Processors (EPs) in the system. An EP is a processor with one of its PCIe interfaces configured as a PCIe endpoint.

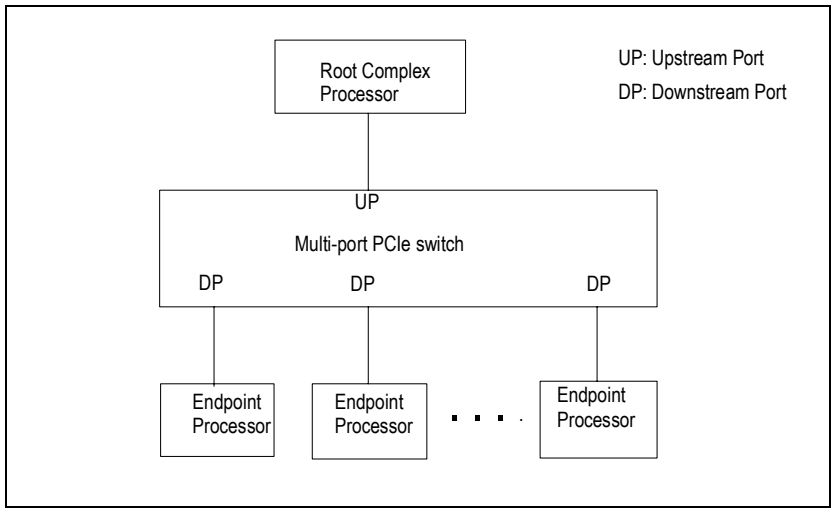

Figure 1 Multi-peer System Topology using PCIe as the System Interconnect

Notes

Root Processor

A standard x86-based PC is used as the RP. The RP uses an AMD Athlon64 CPU with the nVidia nForce4 SLI chipset to support the PCIe interface. One PCIe slot is used to connect to the multi-port PCIe switch. The system block diagram of the RP is shown in Figure 2.

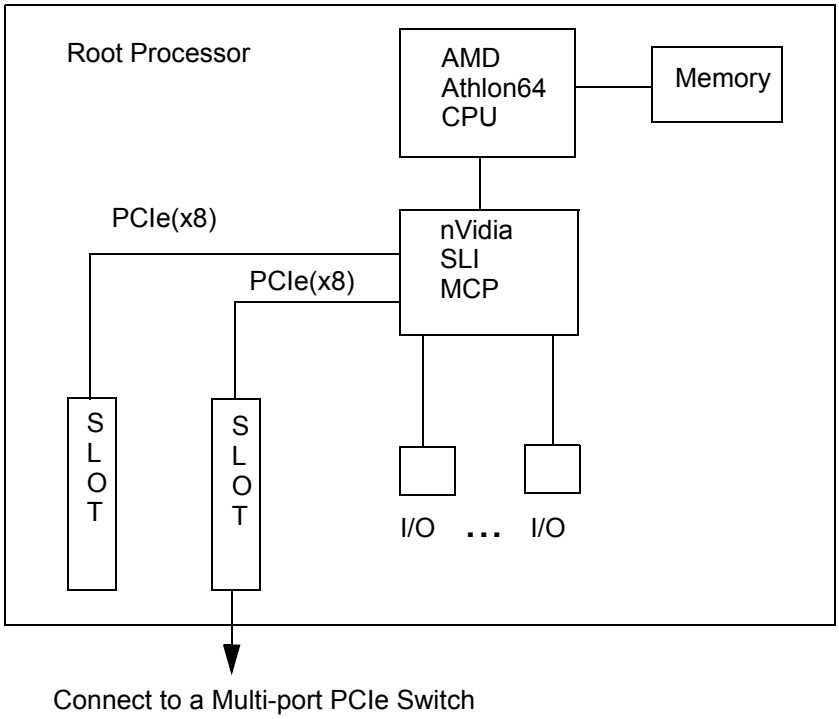

Figure 2 RP System Block Diagram

PCIe Switch

The IDT 89EBHPES64H16 evaluation board [2] (referred to hereafter as EB64H16) is used as the multi-port PCIe switch module. The system block diagram of an EB64H16 is shown in Figure 3. There is an IDT 89HPES64H16 PCIe switch[3] (referred to hereafter as PES64H16) on the evaluation board. There are 16 PCIe connectors on the EB64H16 board. A port may be configured as either a x4 or x8 port. When all the ports are configured as x8, only 8 of the PCIe connectors are used to support the 8 ports in x8 configuration. The upstream port is connected to the RP via two x4 infiniband cables. The RP is plugged directly into the PCIe connector.

Notes

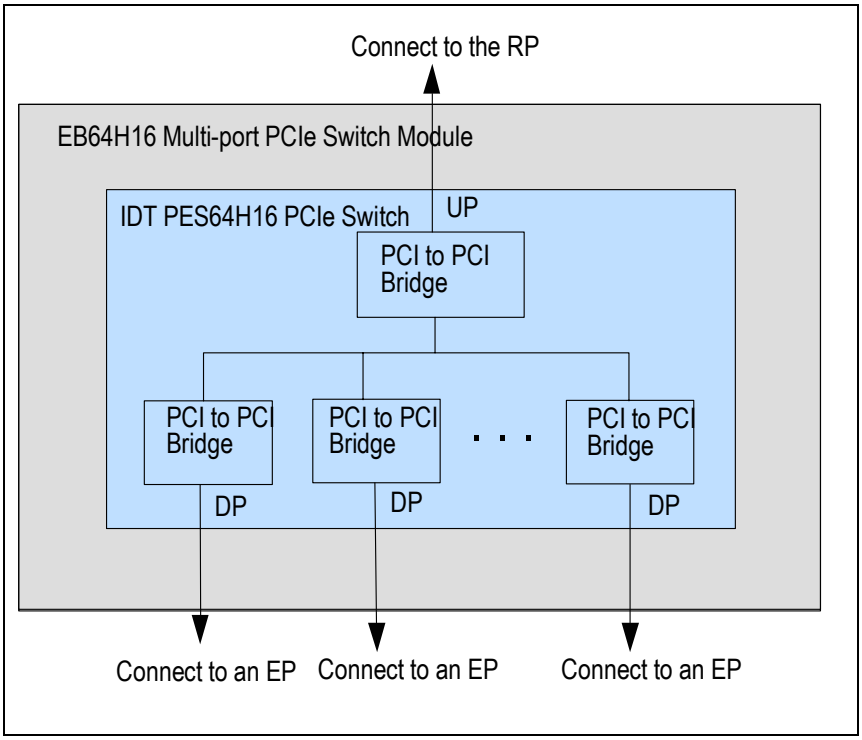

Figure 3 Multi-port PCIe Switch Module

Endpoint Processor

The x86-based EP Processor is an AMD Athlon64 CPU with the nVidia nForce4 SLI chipset to support the PCIe interface. Each x86-based PC connects to one down-stream port of the Multi-port PCIe switch via the Non-transparent Bridge (NTB) port of the IDT PES24NT3 PCIe inter-domain switch. The EP Processor system is shown in Figure 4. Note that the internal endpoint of the NTB is connected to Endpoint Processor while the external endpoint is connected to Multi-port PCIe switch mentioned above.

Notes

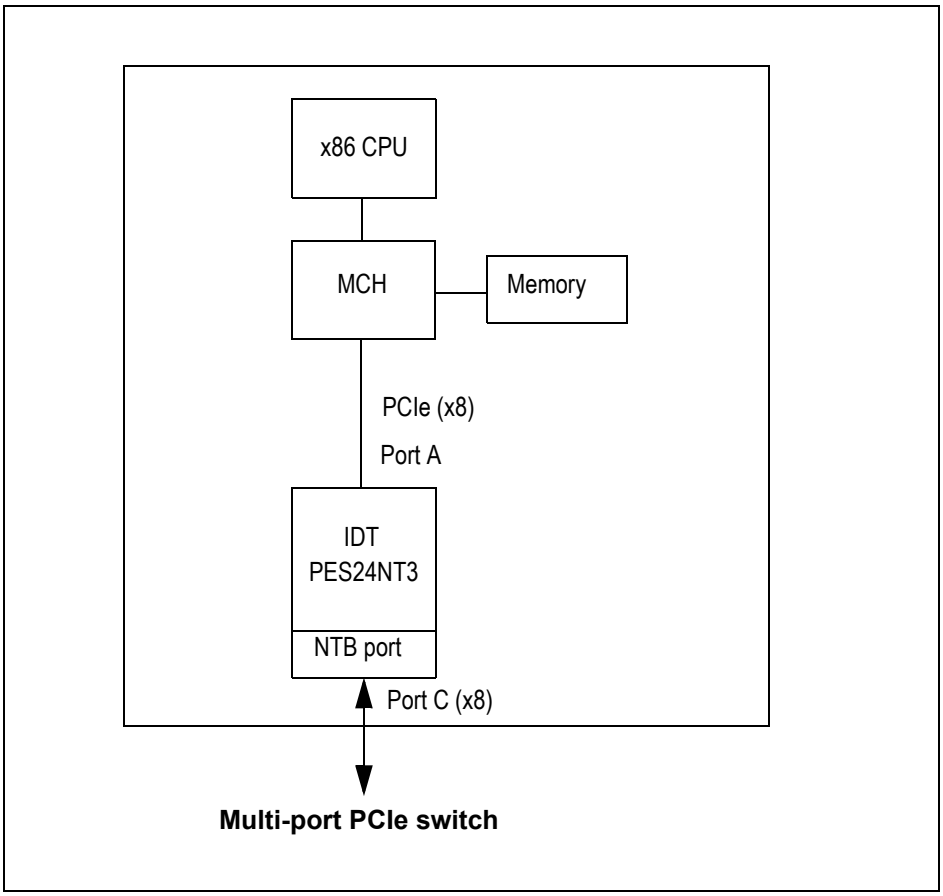

Figure 4 EP Processor Block Diagram

General Software Architecture

The software is implemented as Linux modules and device drivers on the RP and EP. The software architecture for the RP and EP are very similar. The software is divided into three layers. The Function Service layer is the top layer. It provides the device driver interface to the Linux kernel. The Message layer is the middle layer. It encapsulates and decapsulates transport messages in a common format. The lowest layer is the Transport layer. The Transport layer provides the service to send and receive data across the PCIe interface and is hardware-dependent.

RP Software Architecture

The RP software runs under Linux Fedora 3. The software architecture is shown in Figure 5. The Function Service layer provides the device driver functions to the Linux kernel. In this example, four function services are identified (the Disk Function Service is not implemented in the current version of the software). The Raw Data Function Service provides a service to exchange raw data between EPs and RP. The Ethernet Function Service provides a virtual Ethernet interface function. The RP sends and receives Ethernet packets through this function service. The Disk Function Service provides a virtual disk interface. The Configuration Function Service provides functions for system management purpose.

The Message Frame Service encapsulates the function service data in a common Message Frame header for transmission. Once the Message Frame is formatted properly, it is sent to the Transport Service layer for transfer to a remote EP. When the Message Frame Service receives a message from a remote EP, it decodes the message and passes the function data to the appropriate function in the Function Service, i.e., it passes an incoming Ethernet packet to the Ethernet Function Service.

Notes

The Transport Service layer provides the data transport between the local RP and a remote EP. This layer is EP-dependent. A unique transport service is required for each EP type.

The local Architecture Service provides a hardware abstract service, a hardware independent service to the Message Frame Service Layer and the Transport Service Layer, and other services such as the translation between virtual and physical memory addresses and translation between local and system domain addresses.

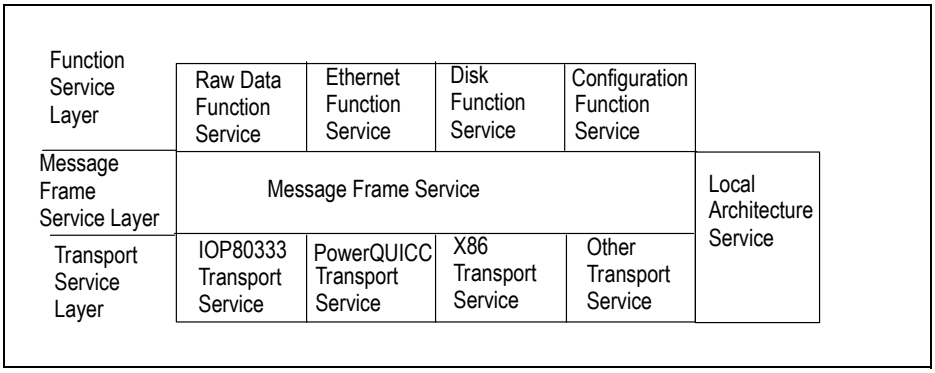

Figure 5 RP Software Architecture

EP Software Architecture

EP software architecture is shown in Figure 6. It is similar to the RP software architecture. The Function and Message Frame services on the EP are the same as on the RP. In addition to the RP software architecture components, the EP software architecture includes a few EP hardware-specific components such as local EP Inbound Transport Service and local EP to RP Transport Service. All the local EP-specific services are part of a single EP-specific device driver.

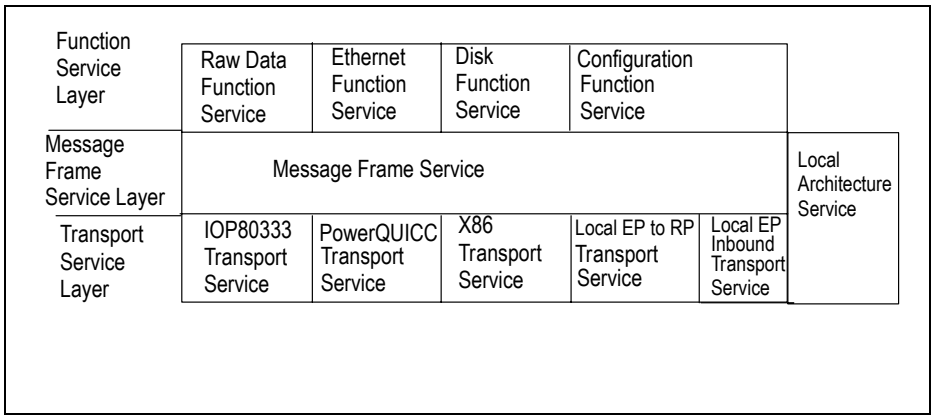

Figure 6 EP Software Architecture

The local EP Inbound Transport Service implements the Inbound Message Interrupt Service routines and notifies the Message Frame Service layer of all the inbound traffic from the RP and other EPs. The Local EP to RP Transport Service implements the outbound transport service towards the RP. The Endpoint specific Transport Service implements the outbound transport service towards the specific EP, i.e. the x86 Transport Service implements the outbound transport service towards an x86 EP. It should be noted that all the inbound traffic goes through the local EP inbound transport service while the outbound traffic goes through one of the other peer-specific transport services, such as IOP80333, FreeScale PowerQUICC III, or Local EP to RP transport services.

Notes

Since each of the peer-specific transport services is implemented as a separate device driver module on the EP and all the local EP-specific services are implemented as a single local EP device driver, any one of the peer-specific transport services may be initialized before or after the local EP device driver is completely initialized. When a peer is added through a notification from the RP, the Message Frame Service layer should immediately associate the newly added peer with its corresponding transport service if its transport service is already registered. Otherwise, the association of the peer with its transport service will be delayed until its transport service registers itself to the Message Frame Service. After the association of a peer and its transport service, the Message Frame Service notifies the function services of the new peer. During the time period when a new peer is added and the association made between the new peer and its Transport Service, the function services may receive messages from this new peer but they will be unable to respond to these messages immediately. The function services may decide to delay the processing of these inbound messages, or they may process these messages immediately and queue the response messages to be transmitted later. The case where a specific peer transport service is supported in some peers and not the others in the system is a user configuration error and not considered here.

Loadsharing, Link Failover and Hot-Swap

The IDT System Interconnect software architecture is capable of supporting advance features such as Loadsharing and Link Failover when the system is configured to a dual-star topology.

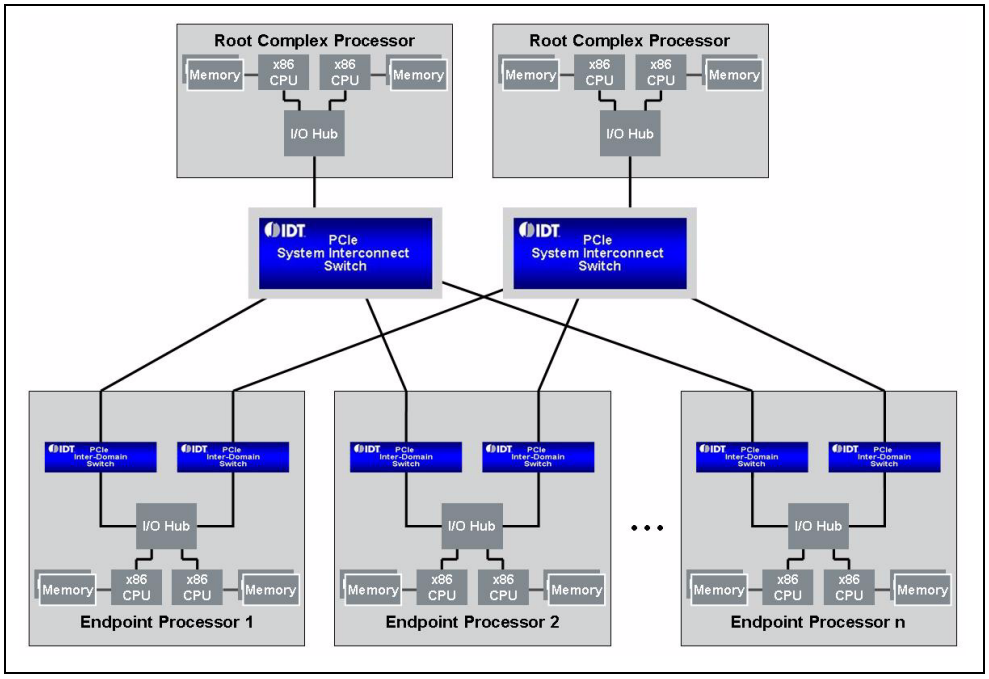

Figure 7 System Interconnect Dual-star Topology

As can be seen from the figure above, in a dual-star topology there are two Root Processors, two IDT PCIe System Interconnect switches and each Endpoint Processor contains two PCIe Inter-domain switches. Although each Endpoint Processor has two physical PCIe Inter-domain switches connected to two independent PCIe System Interconnect switches, each Endpoint Processor still behaves as a single logical entity or node with its own unique node ID.

Software Defined Domains

Due to the fact that the two Inter-domain switches connected to each Endpoint Processor are enumerated via two independent Root Processors, it is possible for the external endpoints of both Inter-domain switches to end up with the same bus, device, function (BDF) numbers. In order combat this problem the System Interconnect software supports Software Defined Domains. When the System Interconnect drivers are loaded on the Root Processor the user is able to specify a unique 8-bit domain number (see the section

Notes

titled Architecture Module for more details). This domain number is appended to the Inter-domain switches external endpoint BDF number by the System Interconnect software running on the Endpoint Processors. In the event that two Inter-domain switch external endpoints receive the same BDF numbers, the domain number is used to differentiate between them.

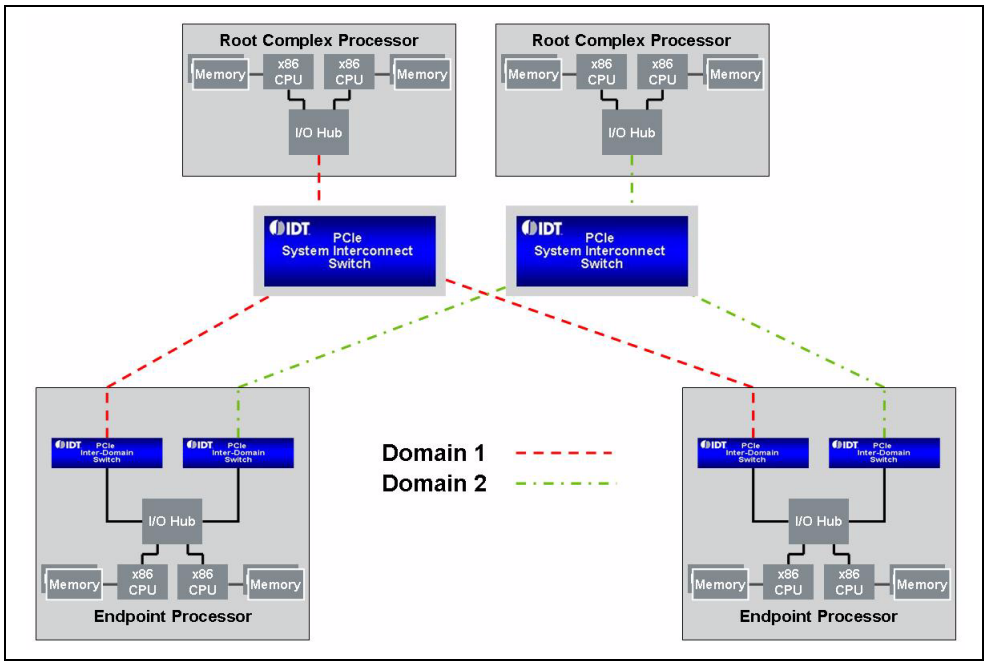

Figure 8 Software Defined Domains

Loadsharing

When loadsharing is enabled (via the System Interconnect software API) a hash function is employed to determine which link an Endpoint Processor should use when transmitting data to a specific destination node, this link is deemed to be the primary link for this destination. This function may be replaced with a user function by calling the mp_set_custom_hash(*func*) function. The *func* parameter is the address of the user hash function which should conform to the following prototype:

```
- int func(u32 srcID, u32 dstID, mp_frame *frame)
```

The *srcID* and *dstID* parameters represent the source and destination node IDs of the frame being transmitted, the *frame* parameter is a pointer to the data frame itself. The hash function should return 0 or 1 to indicated to the System Interconnect software which link to use when sending data to the destination node.

Loadsharing may be enabled in the IDT PCIe System Interconnect Virtual Ethernet driver by setting the *loadsharing* module parameter to a non-zero value when the driver is loaded (see the section titled Virtual Ethernet Device Driver for more details). Alternatively it may be enabled at runtime by calling the mp_enable_loadsharing() System Interconnect software API function with a non-zero value, passing a zero value to this function will disable loadsharing.

Hot-swap & Link Failover

When used in conjunction with the IDT PCIe Hot-Dwap Device Driver[4], it is possible to disconnect and then reconnect the same, or a different Root Processor or Endpoint Processor to the system without compromising the operational state of the rest of the system. By default, when configured in a dual-star topology with loadsharing disabled, the System Interconnect software will always attempt to transmit data on the link with the lowest software domain number. i.e. links in domain 1 will take priority over links in domain 2. The link with the lowest domain number is known as the primary link. If the primary link to a desti-

Notes

nation node is disconnected or if a transmit error occurs, the Endpoint Processor software will automatically switch to using the secondary link to reach the specified destination. The Endpoint Processor software will revert back to using the primary link once it is reconnected or the error condition no longer exists.

Application Examples

An I/O sharing application example is shown in Figure 9. In this system, there is an Ethernet interface in the EP1. This is the only Ethernet interface to connect this system to the Ethernet network. The Ethernet interface is shared by EP2, EP3, and the RP.

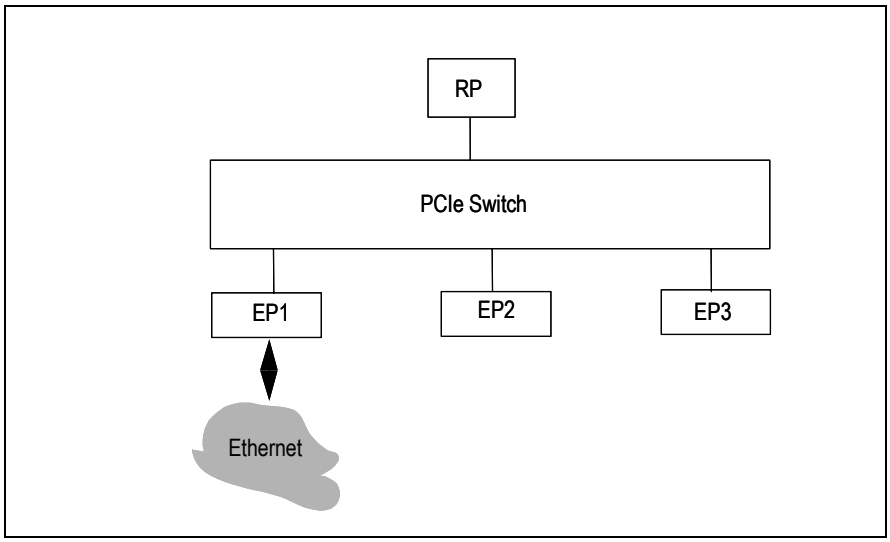

Figure 9 I/O Sharing Application Example

The protocol diagram of the sharing of Ethernet is shown in Figure 10. An Ethernet Function Service is running on an Ethernet EP. The Ethernet Function Service provides a virtual Ethernet interface to the upper layer application such as the IP stack. From the upper layer's point of view, there is a physical Ethernet interface. The Ethernet Function Service makes requests to the Message Frame Service to encapsulate the Ethernet packet in a generic message frame. The remote Transport Service then transports the message frame to its destination.

The EP that provides the sharing of its physical Ethernet interface is the Ethernet Server. The Ethernet Server provides the actual Ethernet connection to the Ethernet network. It uses the Ethernet Function Service on the PCIe interface to send/receive Ethernet packets to/from other EPs and the RP. It runs the Bridging Application to forward Ethernet packets between the Ethernet Function Service and the actual physical Ethernet interface. The Bridging Application may be replaced with an IP Routing Application such that IP packets are routed between the Ethernet Function Service and the actual physical Ethernet interface. Multiple EPs can share a single Ethernet Server and hence the actual physical Ethernet interface. The EP can communicate with other EPs directly without the involvement of the Ethernet Server.

Notes

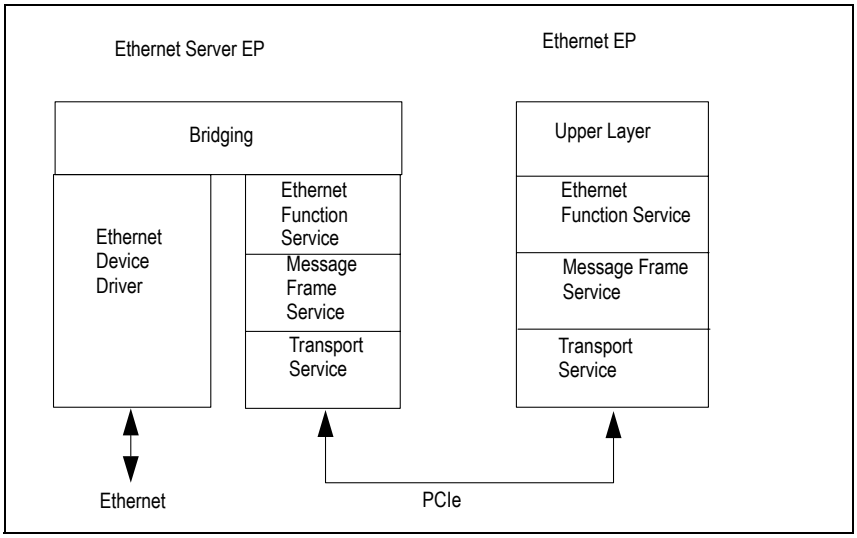

Figure 10 Ethernet Sharing Protocol Diagram

A network router application example is shown in Figure 11. In this example, there are one or many network interfaces supported by each EP. The network interfaces may be Ethernet, WAN interfaces such as DSL, T1, or OC-3. Each EP runs a routing application to forward packets between its network interfaces and the interfaces on the other EPs in the system.

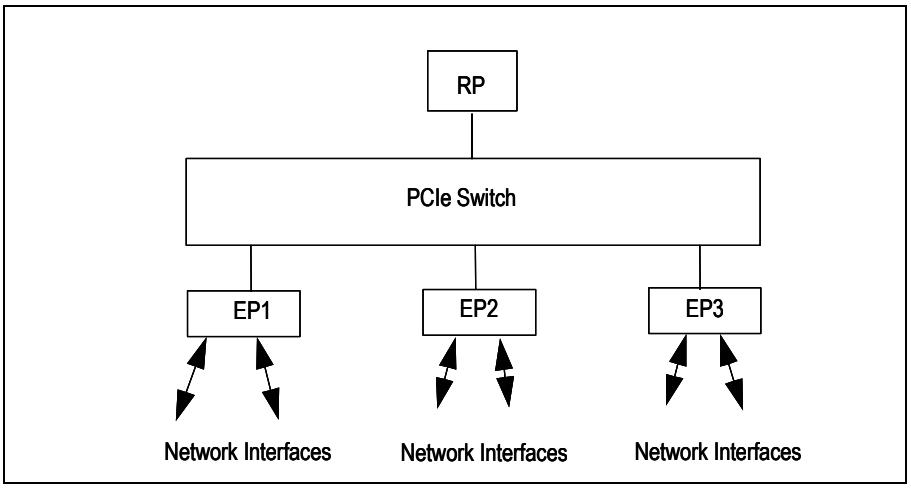

Figure 11 Router Application Example

The protocol diagram of the router application example is shown in Figure 12. The Ethernet Function Service runs on top of the PCIe interface. All the EPs communicate with each other via this virtual Ethernet interface on PCIe. A network service and routing application run on top of the local network interface device drivers and the Ethernet Function Service. All packets received by the network interfaces, including the Ethernet Function Service, are passed to the Routing Application. The Routing Application inspects the packet header and makes a packet forwarding decision. If the packet has to be sent to a different network interface on the same EP, the packet is forwarded directly to the local network interface. If the packet has to be sent to a network interface on a different EP, the packet is sent to the Ethernet Function Service to reach the destination EP. The destination EP then forwards the packet to its local destination network interface.

Notes

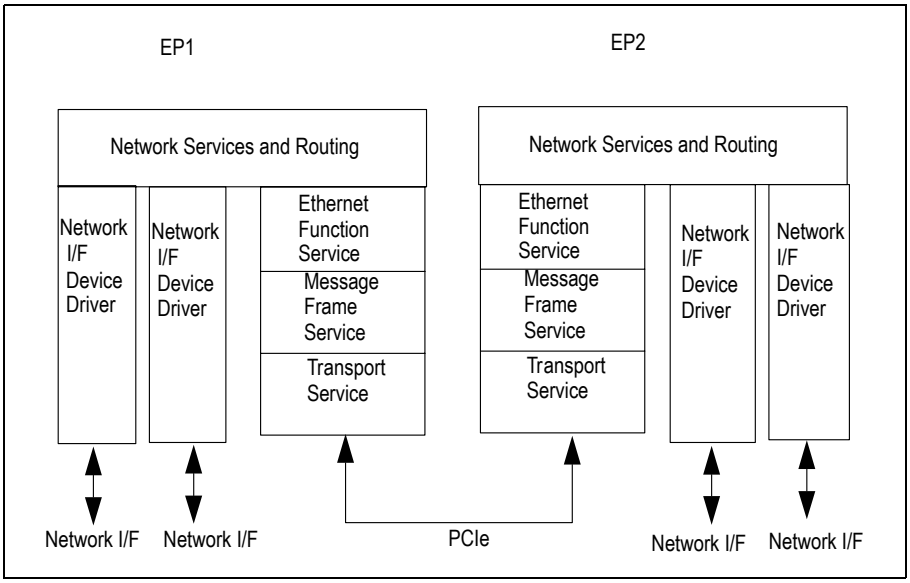

Figure 12 Router Protocol Diagram

Address Translation

There are two address domains in a multi-peer system using PCIe as the System Interconnect. The system domain is the global address domain as seen by the RP. The local domain is the address as seen by the local EP. These two domains are independent of each other. The RP is free to assign address space in the system domain and the local EP can freely assign address space in its local domain. In order to bridge address space between the system domain and the local domain, the EP supports address translation between the two domains. The address translation is a hardware function that exists in the NTB port of IDT's PES24NT3 device.

Transactions initiated on the system domain and targeted on a local EP's local domain are referred to as inbound transactions. Transactions initiated on a local EP's local domain and targeted at the system domain are referred to as outbound transactions. During inbound transactions, the Inbound Address Translation Unit converts a system domain address to the local domain address of an EP. During outbound transactions, the Outbound Address Translation Unit converts an EP's local domain address to a system domain address and initiates the data transfer on the system domain.

The PES24NT3 Inter-Domain switch supports up to 4 inbound and 4 outbound 32-bit address translation windows or 2 inbound and 2 outbound 64-bit address translation windows. The internal endpoint controls the outbound address translation windows and the external endpoint controls the inbound address translation windows. The current implementation uses 32-bit address translation windows.

When a local x86-based EP needs to access any address space in the system domain, it accesses an address within the outbound address window which is specified by the BAR register of the internal endpoint of the NTB port. When the internal endpoint detects that the local address matches its BAR, it forwards the access to the system domain and the address is also translated from the local domain to the system domain address space. A remote EP or RP accesses a location within the inbound address window (BAR of the external endpoint of the NTB port) to access from the system address space to the local address space.

The address translation windows are set up using the BARSETUP and BARTBASE registers of internal and external endpoints of the PES24NT3 switch during the initialization phase between an EP and RP. The PES24NT3 also provides Mapping Tables for both internal and external endpoints. The tables have to be configured properly for the data transfer to work. The Mapping Tables are described in more detail in section Inter-domain Switch Mapping Table on page 25.

Notes

Inbound Address Translation

Two inbound address windows are used in the x86-based EP. External endpoint BAR4 is fixed to map to the NTB endpoint configuration registers. The size of this window is fixed at 4 Kbytes. A remote RP or EP can write to the Doorbell register to interrupt the local x86-based EP. Other registers such as the Message and Scratchpad registers are also used for data transfer communication protocols during the initialization phase. External endpoint BAR1 is configured to map the system domain address to the local data space.

The two inbound address translation windows are shown in Figure 13. In this example, the system domain address space between 0x80000000 and 0x82000000 is reserved to support a 16-peer system. 2 Mbytes of address space are allocated for each one of the peers. The local x86-based EP is in peer#2 in this example. Two inbound address translation windows are set up in the local x86-based EP and the size of each window is 1 MByte. The first inbound address translation window is mapped to the local data space. The local data space contains the queue structures and data buffer for the data transport between the local x86-based EP and the remote EPs and RP. The second window address is mapped to the NTB endpoint configuration registers of the PES24NT3. The actual size of this window is 4 Kbytes. The rest of the 1 Mbyte address space is unused.

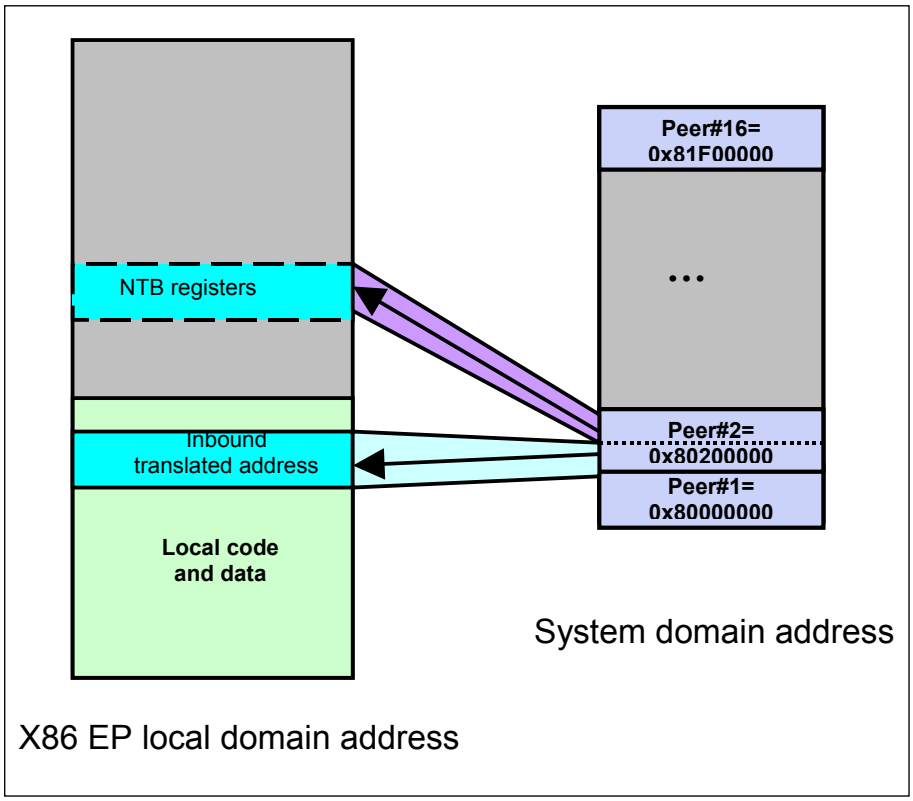

Figure 13 Inbound Address Translation Windows for x86 EP

Outbound Address Translation

Two outbound address translation windows are required. One outbound address window is set up to map to the data space of the RP such that the local x86-based EP can send data to the RP. The second outbound address window is set up to map to access the data space and NTB endpoint registers of all other EPs such that the local EP can send data to other EPs. An example of the outbound address translation windows set up is shown in Figure 14. In this example, the NTB internal endpoint BAR0 is mapped to the data space of the RP. The size of this window is currently set up as 64 Kbytes. The NTB Internal endpoint BAR1 is mapped to the data space of all the other EPs. For a 16-peer system and 2 Mbytes address space per peer, a window size of 32 Mbytes is needed.

Notes

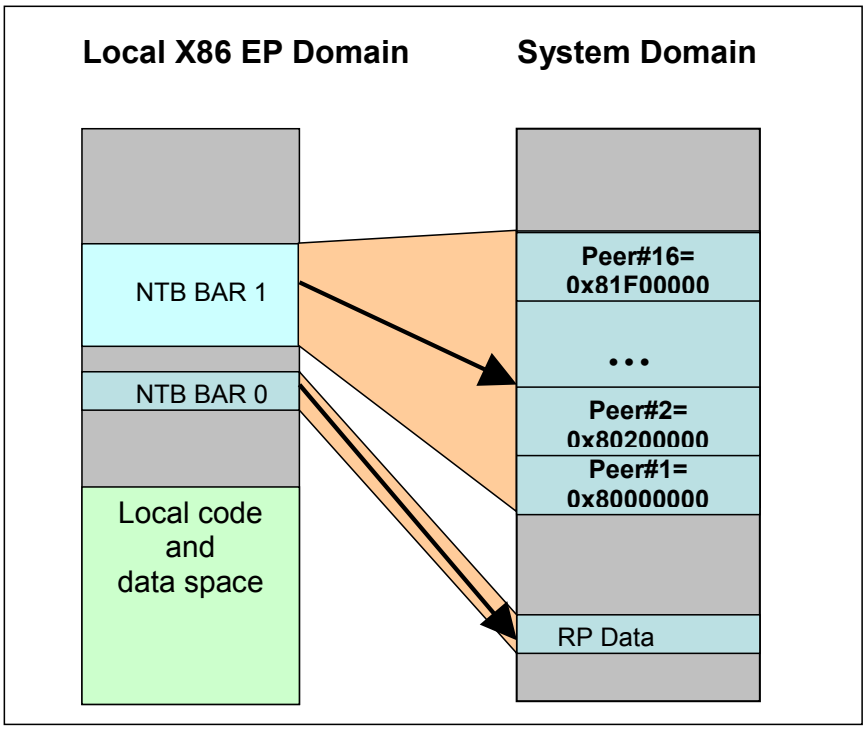

Figure 14 Outbound Address Translation Windows for x86 EP

Data Transport

Each peer sets up a FIFO buffer used to receive data from the other peers in the system. Each FIFO consists of a FIFO control structure and a data buffer. The control structure holds the FIFO state information such as the start and end address of the FIFO data buffer and the addresses where data should be written to (when transmitting data) or read from (when receiving data). Figure 15 shows the FIFO structure detail.

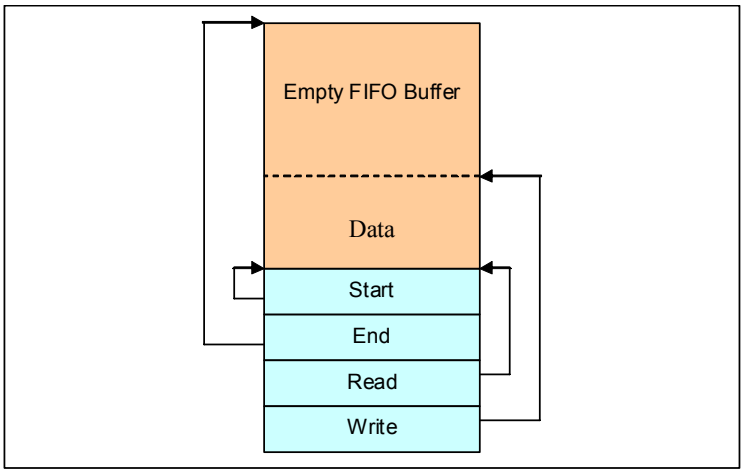

Figure 15 FIFO Structure for x86 EP and RP

Notes

The RP allocates a dedicated FIFO for each EP in the system. The physical address of this FIFO is passed to the corresponding EP during the EP initialization phase. The EP programs this address into the BARTBase register that corresponds to BAR 0 of its internal endpoint. This allows the EP to access the RP FIFO by reading or writing to the address range specified by the internal endpoints BAR 0. Figure 16 shows the usage of the RP inbound window.

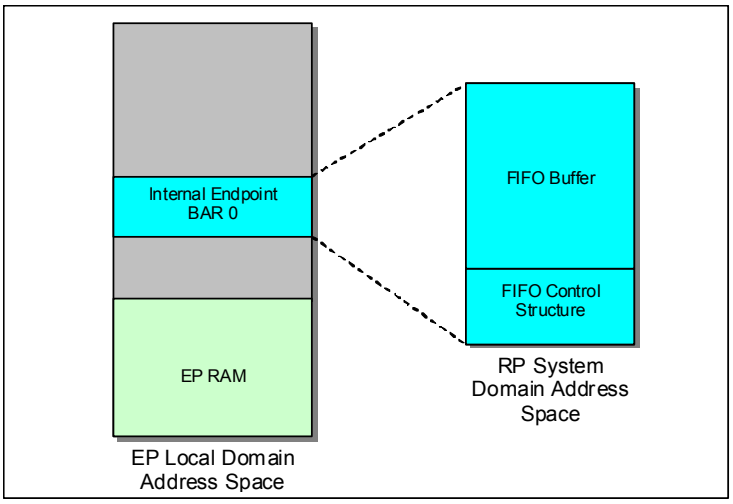

Figure 16 Inbound Window for RP

Each EP allocates one large block of memory which holds the FIFOs for the total number of peers in the system. The physical address of this memory is programmed into the BARTBase register that corresponds to external endpoint BAR 0. This allows the other peers to access the peers FIFOs by reading or writing to the address range specified by the external endpoint BAR 0. The usage of the EP Inbound Translated Address Window memory is shown in Figure 17.

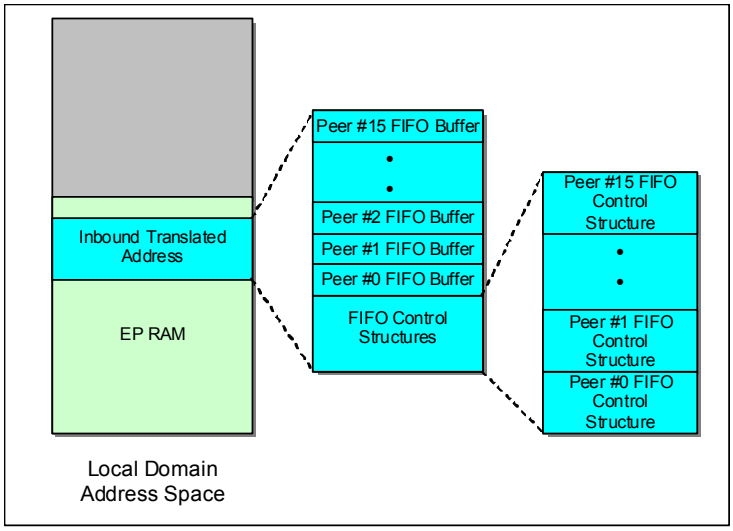

Figure 17 Inbound Translated Address Usage for x86 EP

A total of 16 FIFO control structures are defined at the beginning of the Inbound Translated Address Window memory space. The remainder of the memory allocated to the Inbound Translated Address Window is divided evenly between the 16 peers and is used as the actual FIFO buffer memory.

Notes

In the current design the FIFO control structure for peer #0 is used by the RP to send data to the EP. The other peers in the system use the FIFO control structure associated with their peer index which is determined by the RP and assigned to newly discovered peers during initialization.

In order to transmit data the local peer must first obtain the *write* address from the FIFO control structure on the remote peer that corresponds to the local peers index. Provided that the destination FIFO buffer has enough space to hold the data being transferred, the local peer transfers the data to the destination FIFO buffer and updates the *write* address to point to the next available address in the FIFO buffer. The local peer then interrupts the remote peer by setting the bit in the remote peers doorbell register that corresponds to the local peers index. The exception to this rule is when an EP is transmitting data to the RP. In this case the EP always sets bit 0 of the doorbell register on the RP.

When interrupted the remote peer can determine by the bits set in the doorbell register which FIFOs require processing. While the *read* address in the FIFO control structure does not equal the *write* address there is data to be processed in the FIFO buffer. As each data block is processed the *read* address is updated to point to the next block of data in the buffer. If either the *read* or *write* address exceed the buffer end address they will wrap back to the beginning of the FIFO buffer.

Data Movement Scenarios

A few examples are given to show the sequence of data transport between:

- a local EP to a remote EP
- a local EP to a remote RP
- a local RP to a remote EP

From a Local EP to a Remote EP

Whenever a newly discovered EP is initialized successfully, all of the other EPs in the system will be informed of the base address of its FIFOs and configuration space registers, and the peer index associated with the new EP. The new EP will be informed of similar information associated with the other EPs in the system. With this information, a local EP can initialize the FIFO associated with the new EP while the new EP can do the same thing for the other EPs in the system. The actual steps are described below:

- The local EP retrieves the *write* address from the remote EPs FIFO control structure that corresponds to the local EPs index. The *write* address must be converted from a system domain physical address to a local domain virtual address before data is written to it. If the remote EPs FIFO is full, the data is queued and a timer started to perform a retry at a later time.
- After writing the data to the remote EPs FIFO, the FIFO *write* address is updated and the local EP interrupts the remote EP by clearing and then setting the bit in the remote EPs doorbell register that corresponds to the local EPs peer index.
- When the remote EP detects the doorbell interrupt it retrieves the FIFO *read* address to determine the location of the data. The *read* address must be converted from a system domain physical address to a local domain virtual address before the data is accessed.
- The remote EP allocates a buffer in its local memory and copies the data from the FIFO buffer to the newly allocated buffer. The newly allocated buffer is then posted to the appropriate application for further processing.
- After retrieving the FIFO buffer content and passing it to upper layers, the FIFO *read* address is updated to point to the next data block in the FIFO. If the FIFO *read* address is equal to the *write* address the FIFO is deemed to be empty.

From a local EP to a remote RP

A local EP uses the window mapped with BAR0 of the Internal Endpoint for transferring data to the remote RP. During system initialization, the Transport Service module that runs on the RP allocates a dedicated block of memory in the RP's local data space for each EP in the system. Each block of allocated memory contains a single FIFO control structure and data buffer used by an EP to transfer data to the RP. The RP passes the physical address of the allocated memory to the EP. The EP uses this address to program the BARTBASE0 register of the Internal Endpoint.

Notes

The procedure to transfer data from a local EP to the RP is similar to the EP to EP case except:

- EPs always interrupt the RP by setting bit 0 in the RPs doorbell register regardless of their peer index.
- When the RP accesses the *read* address from the FIFO control structure, it only needs to convert the address from physical to virtual before accessing it.

From a Local RP to a Remote EP

The local RP uses the FIFO structure of the remote EP to transfer data from the local RP to the remote EP. The procedure is the same as transferring data from a local EP to a remote EP except:

- The RP uses bit 0 of the Doorbell register to trigger interrupts on the EP.
- When the RP retrieves the *write* address from the remote EP's FIFO, it only needs to convert the address from physical to virtual before accessing it.

Software Modules

All software components of the System Interconnect system are implemented as Linux-loadable modules. There are five and six Linux-loadable modules for x86-based RP and EPs, respectively.

The modules for RP are:

- idt-mp-i386rp-msg.ko: Message Frame module
- idt-mp-i386rp-arch.ko: Local Architecture module
- idt-mp-i386rp-i386ntb.ko: Transport module
- idt-mp-i386rp-eth.ko: Virtual Ethernet module
- idt-mp-i386rp-raw.ko: Raw Data Transfer module

The modules for EPs are:

- idt-mp-i386ntbep-msg.ko: Message Frame module
- idt-mp-i386ntbep-arch.ko: Local Architecture module
- idt-mp-i386ntbep-i386rp.ko: Transport module
- idt-mp-i386ntbep-i386ntb.ko: Transport module to EPs
- idt-mp-i386ntbep-eth.ko: Virtual Ethernet module
- idt-mp-i386ntbep-raw.ko: Raw Data Transfer module

Please note that even though the statistic function is conceptually a function service, it is implemented in the Message Frame module. This function sends and receives high priority messages to have up-to-date statistical information. Also note that even though the RP and EPs have separate binary Linux-loadable modules, some of them share the same source files. More specifically, idt-mp-i386-msg.ko shares the same source files with idt-mp-i386ntb-msg.ko and the differences are made with some compiling flags. The complete layered picture is illustrated in Figure 18.

Notes

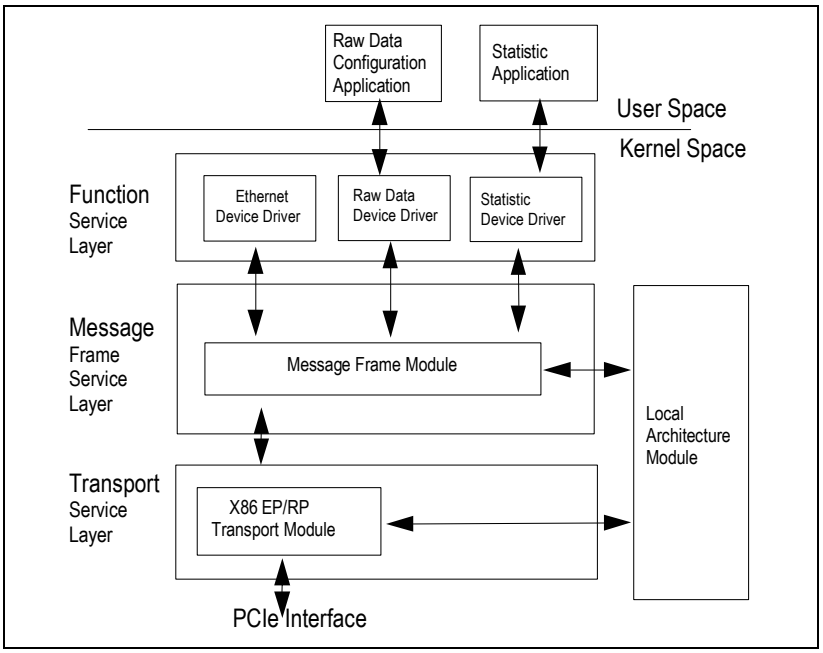

Figure 18 Software Modules and Device Drivers

Function Service Layer

There are currently two function services: the virtual Ethernet and the Raw Data Transfer. The RP and all EPs share the same source files.

Virtual Ethernet Device Driver

The virtual Ethernet module simulates a virtual Ethernet interface mp0. The module initialization function registers the virtual Ethernet interface mp0 with the Linux kernel and then registers itself with the Message module. The MAC address of the virtual Ethernet interface may be specified either on the command line when the module is loaded or when a locally administered MAC address is generated using the linux function call random_ether_address(). A MAC address to destination peer ID table is maintained in the driver. The table is initially empty. Whenever an Ethernet packet is received, the source MAC address and peer ID of the sender is added to the table. When a packet is sent, the destination MAC address is looked up in the MAC address to destination peer ID table. If there is a match, the packet is sent to the corresponding peer ID. If no match occurs, the packet is sent to the “broadcast” peer ID. The Message module sends a copy of the packet to each peer in the system. In other words, the packet is sent to all other peers in the system.

The Ethernet module’s transmit function allocates a mp_frame data structure, sets up the virtual Ethernet function header, sets up data fragments, looks up the destination MAC address for destination peer ID, then passes the frame down to the message service for transmission.

The module’s receive function extracts the virtual Ethernet function header, allocates a Linux sk_buff data structure, sets up the sk_buff fields, then calls mp_frame_sync to transfer the data. When the data transfer is completed, its callback function updates the MAC address to destination peer ID Table with the source MAC address and the source peer ID, passes the sk_buff to the Linux kernel, then releases the mp_frame data structure.

When configured in a dual-star topology, loadsharing may be enabled in the Virtual Ethernet Device Driver by specifying a non-zero value for the *loadsharing* module parameter when the driver is loaded, see below:

- `insmod idt-mp-i386ntb-eth.ko loadsharing=1`

Notes

Raw Data Service Module

The Raw Data Transfer module blindly sends the data it receives to the peer specified by the user and utilizes the new Linux 2.6 sysfs feature for interfacing with the user. The module initialization function sets up data buffers, registers itself with the multi-peer Message module, then sets up the subsystem attributes in the sysfs for interfacing with the user. The user writes the peer ID to which the received data should be forwarded into the 'forward' attribute. To generate the data, the user writes the number of frames into the 'count' attribute, writes the data content into the 'buffer' attribute, then writes the length of the data in bytes into the 'send' attribute. When the 'send' attribute is written, the corresponding store function allocates a mp_frame data structure, sets up the mp_frame data structure with the length specified, clones the frame with the number specified in the 'count' attribute minus one, then passes the frames down to message service for transmission. Its receive function allocates a buffer, then calls mp_frame_sync to transfer the data. When the data transfer is completed, its callback function allocates a new mp_frame data structure, sets up the mp_frame data structure, then passes the frames down to the message service for transfer to the destination specified by the 'forward' attribute.

Message Layer Service

The Message layer is the centerpiece of the multi-peer system. It connects and multiplexes the function and transport services to transfer data frames between peers.

Message Module

The Message module provides the interface for the Function and Transport modules to register themselves and transfer data frames. The module initialization function initializes the peer management related data structures, creates the mp workqueue for processing peer notification messages, and registers the mp subsystem with the Linux sysfs system. On the RP, when a transport service adds a new peer, the message service sends a notification of the new peer to each existing peer and a notification of each existing peer to this new peer. On the EPs, when a peer notification is received, the message service notifies the corresponding transport service of the new peer. In addition, when a peer is added, the message service creates a peer ID attribute in the sysfs to represent the known peers and interface with the user. When a function service sends a data frame, the message service looks up the destination peer type and passes the data frame to the corresponding transport service to transfer the data to the destination peer. When a transport service receives a data frame, the message service peeks into the message header to determine which function service should receive the data frame and passes the data frame accordingly.

The Message module can send messages to all other peers in the system. When the destination peer ID is unknown or "Broadcast", a message is duplicated and sent to each peer in the system.

Architecture Module

The Architecture module encapsulates the common architecture-specific functions, such as address space conversion routines. Each different type of RP and EP has its own architecture-specific module and does not share source files. The address space conversion routines convert addresses between virtual, physical, bus, and PCI addresses.

In a dual-star topology software defined domains may be enabled when the Architecture Module is loaded by specifying the *domain* module parameter, as follows:

- `insmod idt-mp-i386rp-arch.ko domain="00:05.0=1"`

In the above example, domain 1 corresponds to all devices connected to the root port with the BDF number of 00:05.0. Multiple domains may be specified on a single Root Processor provided that they are separated by a comma. I.e. domain="00:05.0=1,00:06.0=2".

Transport Service Layer

The Transport service is responsible for detecting and setting up the hardware, managing the FIFO buffers, and initiating actual data transfers. There are two separate Transport modules which do not share source files. One module runs on the RP and the other runs on the EPs.

Notes

EP Transport Module

The Transport module running on the RP is implemented as a generic PCI driver. The module initialization function initializes the transport data structure, registers itself with the message service, then registers the PCI driver with the Linux kernel. When the Linux kernel detects an external endpoint of Inter-domain, the probe function of the PCI driver is called. The probe function allocates and initializes the peer related data structures, enables the PCI device, requests the memory and interrupt resources associated with the PCI device, then communicates with the RP Transport module running on EPs to setup the memory windows and data frame buffers.

The module initialization function of the EP Transport module running on EPs initializes the transport data structure and registers itself with the message service. When the message service on EPs receives a peer-add notification, it calls the peer_add function of the EP transport service. The peer_add function initializes and registers the peer data structure and makes the new peer available for data transfers.

The transmit function is called by the message service to transmit data to an EP for both modules running on the RP and the EPs. The receive function of the EP transport service running on the RP is triggered by the interrupt handler when an EP sends data to the RP. Note that there is no receive function in the EP transport running on the EPs. The data reception is handled by the RP Transport module on the EPs.

RP Transport Module

The RP Transport module runs on the EPs. The module initialization function initializes the transport data structure, registers itself with the message service, and initializes the hardware. It then communicates with the EP Transport module running on the RP to setup the memory windows and data frame buffers. The transmit function is called by the message service to transmit data to the RP. The receive function is triggered by the interrupt handler when the RP or an EP sends data to the local EP.

Notes

System Initialization

EEPROM Setup

Before the system is powered up, all EEPROMs on PES24NT3 evaluation boards need to be programmed as shown in the following table.

Register Name	Register Offset	Register Value	Comment	
PA_SWCTL	0x0A4	0x00000054	Hot Plug Mode Port A Lane Reversal Port C Lane Reversal	= PCIe 1.0a hot-plug = true = true
PB_PCIECAP	0x1040	0x01000000	Slot Implemented	=True
PC_PCIECAP	0x2040	0x01000000	Slot Implemented	= True
PCIE_INTRLINE	0x303C	0x00000100	Use INTA	
PCEE_INTRLINE	0x383C	0x00000100	Use INTA	
PCIE_NTBCTL	0x3078	0x00000000	Opposite Side Mode	= Opposite side enabled
PCIE_BARSETUP0	0x307C	0x80000140	Memory Space Indicator Address Type Prefetchable Size Bar Enable	= Memory Space = 32-bit addressing = False = 0x14 (1 Mbyte) = True
PCEE_BARSETUP0	0x387C	0x80000150	Memory Space Indicator Address Type Prefetchable Size Bar Enable	= Memory Space = 32-bit addressing = False = 0x15 (2 Mbytes) = True
PCIE_BARSETUP1	0x3084	0x800001A0	Memory Space Indicator Address Type Prefetchable Size Bar Enable	= Memory Space = 32-bit addressing = False = 0x1A (64 Mbytes) = True
PCIE_BARSETUP4	0x309C	0x800000C0	Memory Space Indicator Address Type Prefetchable Size Bar Enable	= Memory Space = 32-bit addressing = False = 0xC (4 Kbytes) = True
PCEE_BARSETUP4	0x389C	0x800000C0	Memory Space Indicator Address Type Prefetchable Size Bar Enable	= Memory Space = 32-bit addressing = False = 0xC (4 Kbytes) = True

Notes

System Connection Topology

To ensure all assigned BAR addresses in the system domain are contiguous, the external Endpoint of Inter-domain will connect to IDT's 89HPES64H16 PCIe switch as illustrated in Figure 19.

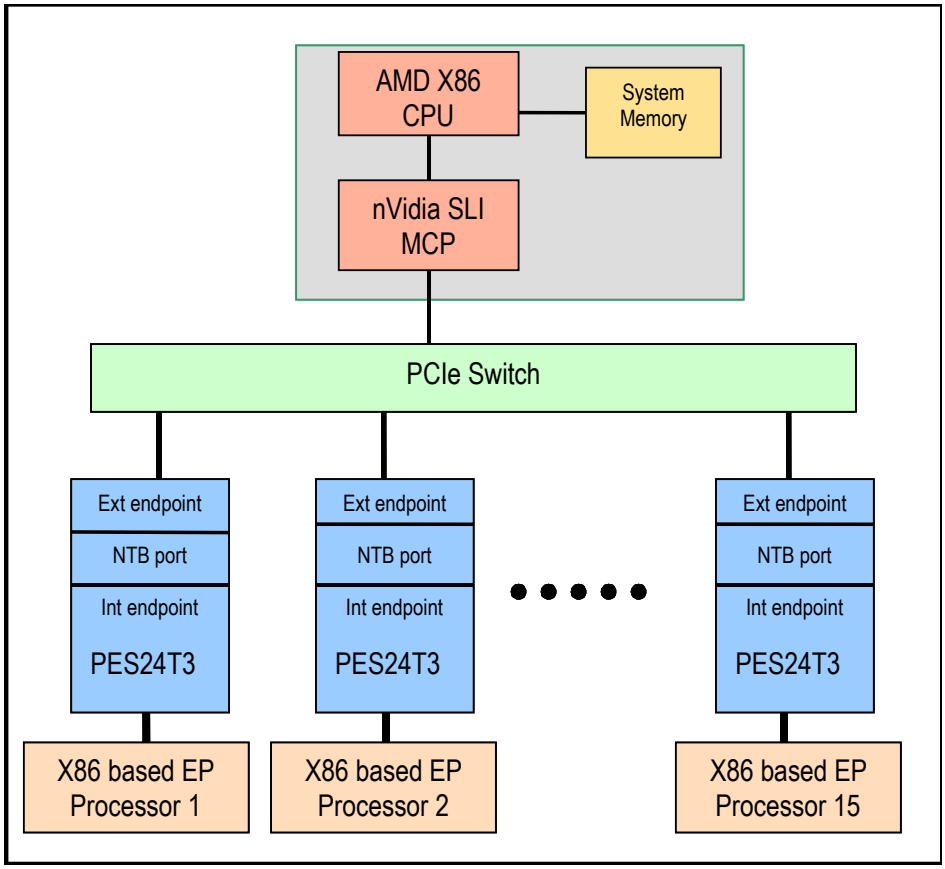

Figure 19 x86 based System Interconnect with Inter-domain

Since the external endpoint can only be visible after the internal side is powered up, all connected EPs should be powered up before the RP. After all systems boot up successfully, IDT System Interconnect modules can be loaded with the *insmod* command. Due to a dependency between these modules, the Message Service module must to be loaded first, followed by the local Architecture module, then the Transport Service module, and finally the remaining modules.

EP Initialization

When the x86 Linux kernel detects a NTB Internal Endpoint device, the probe routine in the Transport Service module is invoked. The following tasks will be done before communicating with the RP:

- Enters the state STATE_DOWN.
- Allocates peer data structure for itself and the RP.
- Initializes its device data structure, including x86 specific peer structure, timers, spin locks and interrupt related setup, etc.
- Calls PCI related API to enable the device.
- Sets up NTB Mapping Table entries.
- Allocates local memory and initializes inbound queue structure.
- Calls *ioremap* to have virtual addresses of BAR0, 1 and 4.
- Enters the state STATE_INIT, and schedules a task to write the physical address of its inbound memory block to OUTMSG1. Its current state is written into OUTMSG0.
- Notifies (using the doorbell register to interrupt the RP) the RP its readiness for communication.

Notes

Once the EP receives a notification from the RP and confirms that the RP is the state in STATE_INIT, the following tasks are performed:

- Retrieves two BARTBase addresses from INMSG1 and INMSG2 and programs its BARTBASE0 and BARTBASE1 registers.
- Retrieves two addresses from SCRATCHPAD0 and SCRATCHPAD1 to initialize mp_x86_peer structure, also passes these addresses to the local Architecture module for peer-to-peer communication later. The address from SCRATCHPAD0 is the inbound base address while SCRATCHPAD1 contains the base address of the configuration space. They are physical addresses in the system domain.
- Initializes the peer for RP by calling mp_peer_add.
- Waits for notification from the RP.

Once the EP receives a notification from the RP confirming that RP is in the STATE_MAP state, it does the following:

- Retrieves its peer index from INMSG3 and passes it to the local Architecture module.
- Retrieves its peer id from INMSG3 and calls mp_self_add with it. Peer id is composed with the Bus#, Device# and Function# of the external endpoint of the NTB port.
- Enters the STATE_MAP state and responds to the RP notification via an interrupt, informing the RP that it (EP) is also in the STATE_MAP state.
- Waits for notification from the RP.

Once the EP receives a notification from the RP confirming that RP is in the STATE_OK state, it does the following:

- Initializes the remote queue structure to RP.
- Enters the STATE_OK state and responds to the RP notification via an interrupt, informing the RP that it (EP) is also in the STATE_OK state.
- At this point, the EP has finished the initialization procedure and is ready for data transfer.

While in the STATE_OK state, if an EP receives an interrupt with the INMSG0 indicating that the RP is being removed, the EP does the following:

- Clears all OUTMSG registers.
- Nulls peer index.
- De-initializes all peer related information by calling mp_peer_del.
- Enters the STATE_INIT state and waits until it receives an interrupt from the RP to re-start the initialization procedure.

RP Initialization

When the x86 Linux kernel detects an NTB External Endpoint device, the probe routine in the Transport Service module is invoked. The following tasks will be done before communicating with the EP:

- Enters the STATE_DOWN state.
- Allocates peer data structure for the EP.
- Compiles a peer ID for the EP with Bus#, Device# and Function#.
- Initializes its device data structure, including x86 specific peer structure, timers, spin locks and interrupt-related setup, etc.
- Calls PCI-related API to enable the device.
- Sets up the NTB Mapping Table entries.
- Allocates local memory and initializes inbound queue structure.
- Calls *ioremap* to have virtual addresses of BAR0 and BAR4 of the EP.
- Enters the STATE_INIT state, schedules a task to write the physical address of its inbound memory block to OUTMSG1, and writes the 64MB aligned physical memory address (system domain address to reach all EPs) to OUTMSG2. These addresses are for EP to program the BARTBASE registers. The state is written into OUTMSG0.
- Writes addresses assigned for BAR0 and BAR4 in the SCRATCHPAD0 and SCRATCHPAD1 registers. These addresses are used to initialize inb_base and reg_base of mp_x86_peer by the EP.
- Triggers an interrupt to indicate its readiness to begin the initialization procedure with EP.

Notes

Once the RP receives notification from an EP and confirms that EP is in the STATE_INIT state, the RP is in control of the entire initialization procedure with the EP. Only the RP can notify the EP to move on to the next state, and the EP only responds after receiving this notification. The following tasks must be done before proceeding:

- Retrieves a BarTBase address from INMSG1 and programs its BARTBASE0 register accordingly.
- Calls mp_peer_add to request a peer index for the EP.
- Enters the STATE_MAP state, notifies the EP of its current state, and passes the peer index via OUTMSG3.

Once the RP receives a response from the EP and confirms that EP is in the STATE_MAP state, it does the following:

- Initializes the remote queue structure to the EP.
- Enters the STATE_OK state and notifies the EP of this current state.
- Completes the entire initialization procedure only when the EP responds again, indicating it (EP) is also in the STATE_OK state.

While in the STATE_OK state, if the RP receives an interrupt with the INMSG0 indicating that the EP is being removed, RP does the following:

- Clears all OUTMSG registers.
- Nulls peer index.
- De-initializes all peer related information by calling mp_peer_del.
- Enters the STATE_INIT state and waits until it receives an interrupt from the EP to re-start the initialization procedure.

After both endpoints of the NTB port are powered-up properly, either the EP or RP modules can be loaded first when their device initialization routines are called. Both the EP and RP will enter the STAT_INIT state and trigger an interrupt to notify the other side. Once both sides are in STATE_INIT, the RP will be in control of the remainder of the initialization procedure, and the EP will collect information from the RP and respond to the RP notification until both sides enter the STATE_OK state and are ready for data transfer.

Figure 20 illustrates an example of the state transactions between the EP and RP, assuming the EP's module is loaded first.

Notes

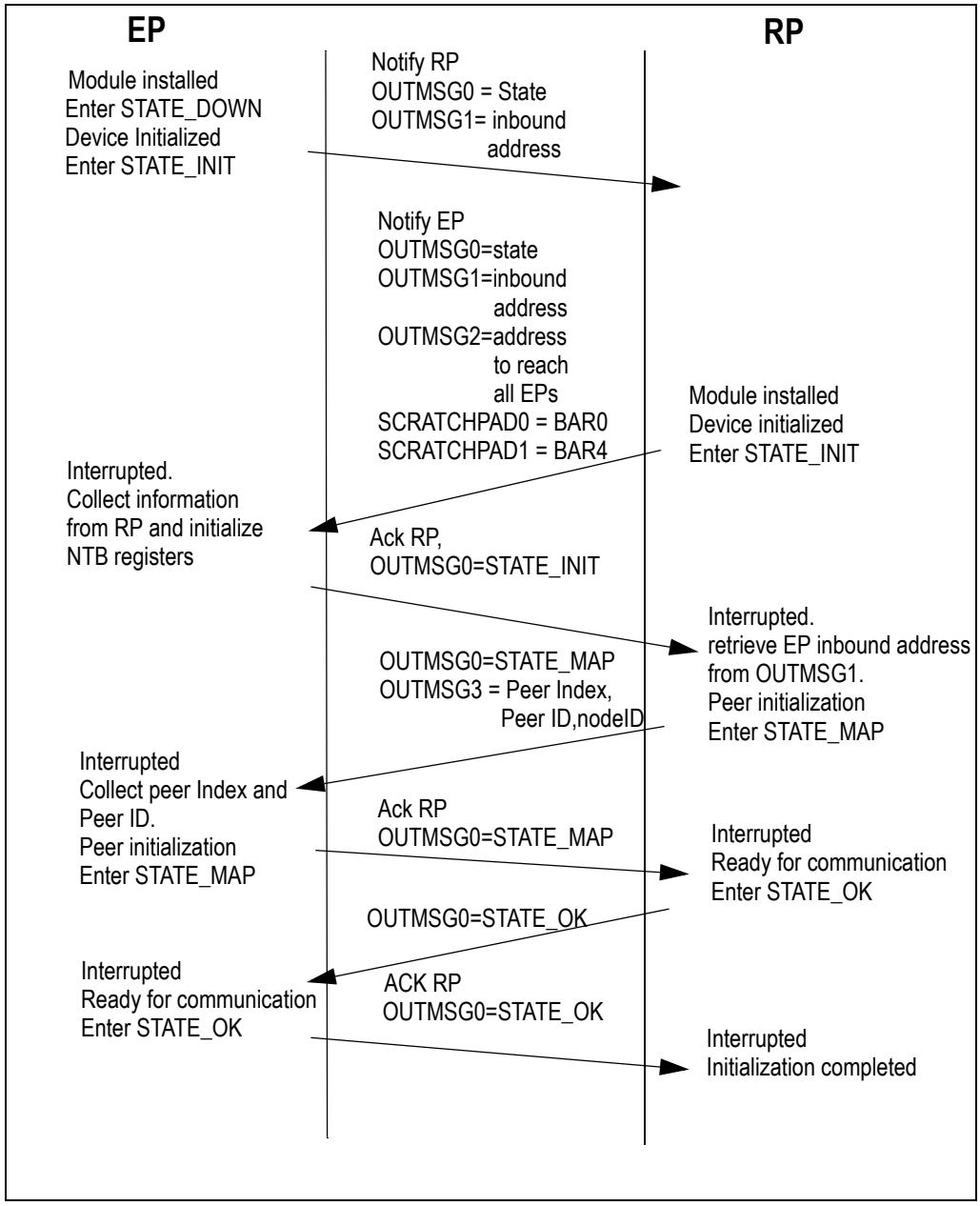

Figure 20 Example of State Transaction between Peers During Initialization

EP to EP Initialization

Since each of the peer-specific transport services is implemented as a separate device driver module (refer to the section EP Software Architecture on page 5), when the module is loaded, its module_init routine begins allocating, initializing, and registering the transport service data structure. Also, the type of peer can be identified while allocating the structure. For example, label MP_TRANS_X86 is used for transporting data to x86-based Endpoints. The data structure includes function pointers for peer_add, peer_del, frame_send, etc. These functions are called when it is necessary to add or delete a peer and when transferring data to a peer.

Notes

The Message Frame Service layer exports a function called mp_peer_add which allows EPs to call and indicate that it has finished the initialization process, including EP to EP initialization, between itself and the RP and that it is now ready to transfer data. When called, the mp_peer_add function does the following:

- Loops through its peer list to see if the peer exists. If the peer does exist, it does nothing and returns.
- If the peer is a new one, it adds the peer into the peer list.
- Initializes a kernel object for the peer and registers the object.
- Notifies other existing peers of the newly discovered peer by calling the peer_add function associated with each existing peer.

When the peer_add function of an EP gets called, the real EP to EP initialization starts:

- Loops through its peer list to see if the peer exists. If the peer does exist, it does nothing and returns.
- If the peer is a new one, it calls mp_peer_alloc to allocate a peer data structure for the new peer.
- Initializes the peer data structure.
- Each EP to EP Transport Service module is required to initialize certain peer-specific information. The following information of the newly discovered EP is passed to the existing EP: system domain address to access the EP's inbound address window, system domain address of the EP's NTB registers, and the EP's peerindex.
- Adds the peer to the Message Frame Service layer by calling mp_add_peer.
- Adds the peer to its own peer list.

When the peer_del function of an EP gets called with a non-NULL pointer to the peer data structure, the EP starts the following de-initialization steps to remove the peer:

- Removes the peer from its own peer list.
- Frees the resource the EP allocated for the peer, such as timer for transmit re-try and tasklet.
- Calls mp_peer_del to remove the peer from the Message Frame Service layer.
- Cleans up all pending jobs in its transport queue to the peer.
- Calls mp_peer_free to free the peer data structure.

Notes

Inter-domain Switch Mapping Table

The Inter-domain feature of the PES24NT3 switch provides both internal and external endpoints. The internal endpoint is facing towards Port A, the root port. The external endpoint is facing towards the external PCIe interface port C. Associated with the internal and external endpoints are 16-entry mapping tables. The mapping tables are used to translate the requester and completer IDs between the internal and external endpoints. Before any PCIe packet can be forwarded between the internal and external endpoints, the mapping table must first be configured. The mapping table contains entries of Bus, Device, and Function numbers (BDF). An example on the mapping table configuration is shown in Figure 21.

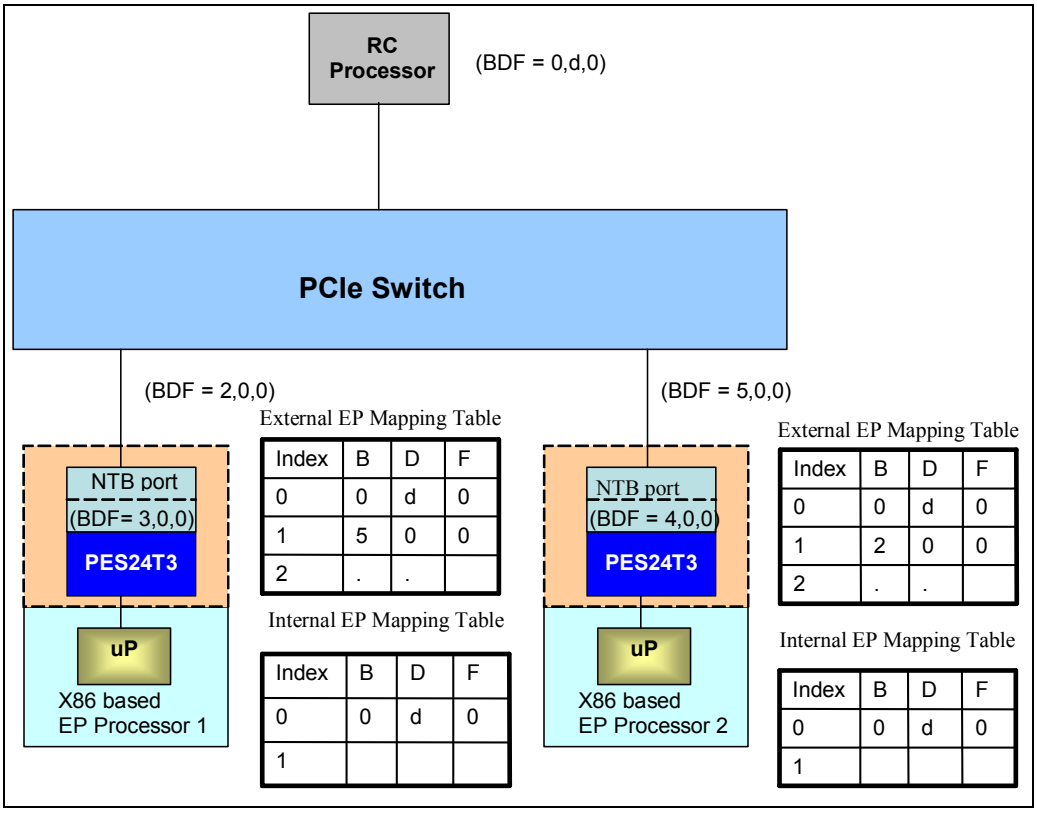

Figure 21 Inter-domain Mapping Table Configuration

This example shows an RC Processor at the top. The BDF for the RC processor are 0,d,0. When the RC Processor sends a request packet to the PCIe interface, the Requester ID is 0,d,0. When the RC Processor sends a completion packet to the PCIe interface, the Completer ID is 0,d,0.

The system domain BDF for EP Processor 1 is 2,0,0. This is the BDF of the external endpoint of the NTB port. The local domain BDF for EP Processor 1 and the internal endpoint of the NTB port are 0,d,0 and 3,0,0 respectively.

The system domain BDF for EP Processor 2 is 5,0,0. This is the BDF of the external endpoint of the NTB port. The local domain BDF for EP Processor 2 and the internal endpoint of the NTB port are 0,d,0 and 4,0,0 respectively.

The external and internal EP mapping tables for the NTB ports are configured accordingly before any data transfer can happen. For details on the initialization and configuration of the PES24NT3 Inter-Domain switch, see reference[5] at the end of this application note.

Notes

Summary

The software architecture to support PCIe System Interconnect has been presented in this document. This software has been implemented and is working under Linux with the x86 CPU as the Root Processor and the x86 CPU with the IDT PES24NT3 Inter-Domain Switch as the Endpoint Processors. Software source code is available from IDT.

The software is implemented as device drivers and modules running in the Linux Kernel space. There are three layers in the software to separate the different software functions and to allow maximum reuse of the software. The Function Service Layer is the upper layer. It provides the function service that is visible to the Operation System and upper layer application. Multiple function services have been implemented in the current release of the software: the Ethernet Function Service provides a virtual Ethernet interface to the system, the Raw Data Function Service provides transfer of user data between EPs and RP, and the Statistic Function Service provides the function to collect traffic statistics for management and diagnostic purposes. The Message Frame Layer contains the Message Frame Service which provides a common message encapsulation and de-capsulation layer to all the function services. It also notifies all other Endpoint Processors whenever a new Endpoint Processors is discovered. The Transport Service layer deals with the actual data transport between Endpoint Processors and Root Processors using the PCIe interface. The Transport service is Endpoint Processor specific. This version of the System Interconnect software supports x86-based Root Processor and Endpoint Processors.

Apart from the inter-processor communication application, this software demonstrates that I/O sharing can now be implemented using a standard PCIe switch. The sharing of a single Ethernet interface by multiple Endpoint Processors and the Root Processor has been implemented and functions properly.

The address translation unit is used to isolate and provide a bridge between different PCIe address domains. The freeQ and post Q structures are used as part of the message transport protocol.

This software release lays down the foundation to build more complex systems using the PCIe interface as the System Interconnect. The software follows a modular design which allows the addition of function services and other Endpoint Processors support without making changes to existing software modules. Complex systems, such as embedded computing, blade servers supporting I/O sharing, and communication and storage systems, can be built today using PCIe as the System Interconnect.

Reference

[1] Enabling Multi-peer Support with a Standard-Based PCI Express multi-port Switch White Paper, Kwok Kong, IDT.

[2] IDT 89EBPES64H16 Evaluation Board Manual (Eval Board: 18-624-000).

[3] IDT 89HPES64H16 PCI Express User Manual.

[4] Application Note 546, PCIe Hot-Swap Device Driver, Craig Hackney, IDT.

[5] IDT 89HPES24NT3 PCI Express User Manual.

Revision History

December 18, 2007: Initial publication.

September 25, 2008: Second publication with various revisions.