PCI Express® System Interconnect Software Architecture

Application Note AN-531

Notes

By Kwok Kong

Introduction

A multi-peer system using a standard-based PCI Express (PCIe®) multi-port switch as the system interconnect was described by Kong [1]. That paper described the different address domains existing in the Root Processor and the Endpoint Processor, the 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 Intel x86 based system as the Root Processor (RP), IDT's PES64H16 device as the multi-port PCIe switch for the system interconnect and the Intel IOP80333 [2] as the Endpoint Processor (EP). This paper presents the software architecture of the multi-peer system as implemented by IDT. This 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 Complex 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.

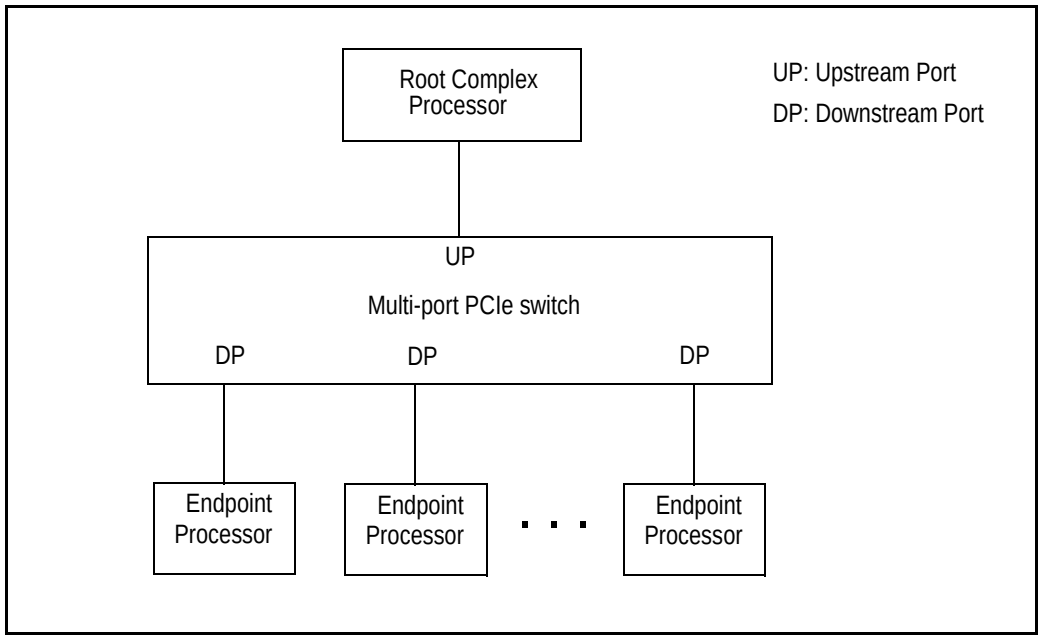

Figure 1 Multi-peer System Topology using PCIe as the System Interconnect

Root Complex Processor

A standard based PC is used as the RP. The RP uses an Intel Xeon CPU and the E7520 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.

Notes

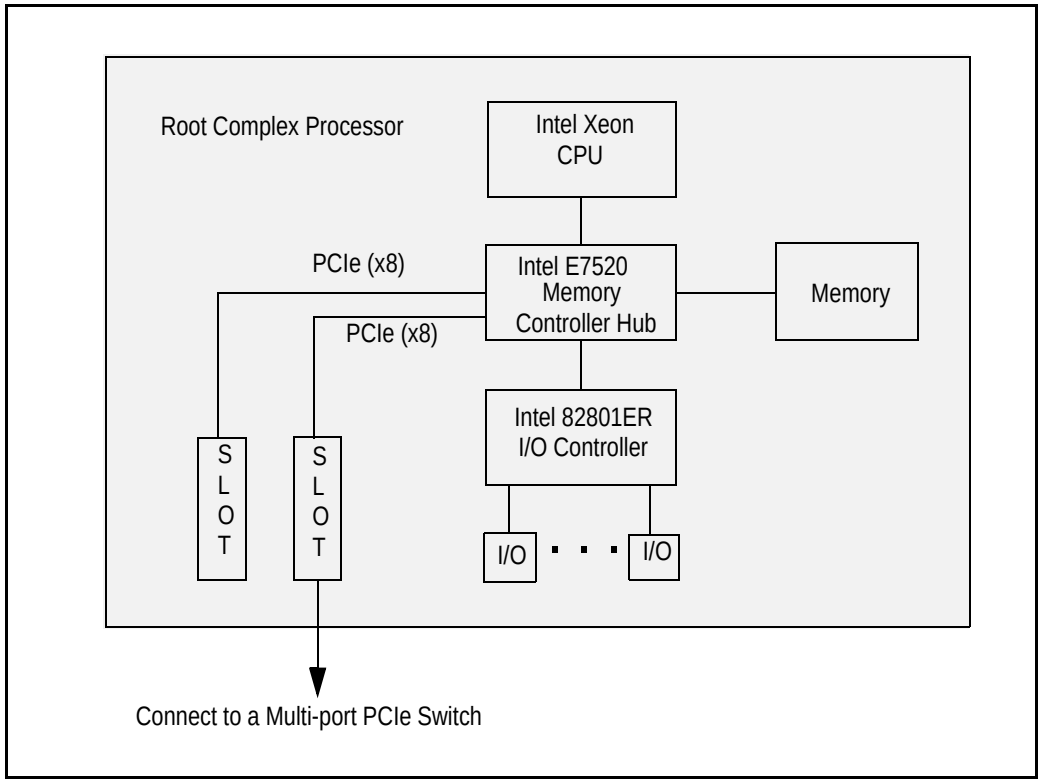

Figure 2 RP System Block Diagram

PCIe Switch

The IDT 89EBHPES64H16 evaluation board [3] (referred to 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 [4] (referred to as PES64H16) on the evaluation board. There are 16 PCI Express connectors in the EB64H16. A port may be configured as either a x4 or x8 port. When all the ports are configured to be x8 ports, only 8 of the PCI Express connectors are used to support the 8 ports of x8 configuration. This implementation configures all ports to be x8 ports.

The upstream port is connected to the RP via two x4 Infiniband cables. The RP is plugged directly into the PCI Express connector.

Notes

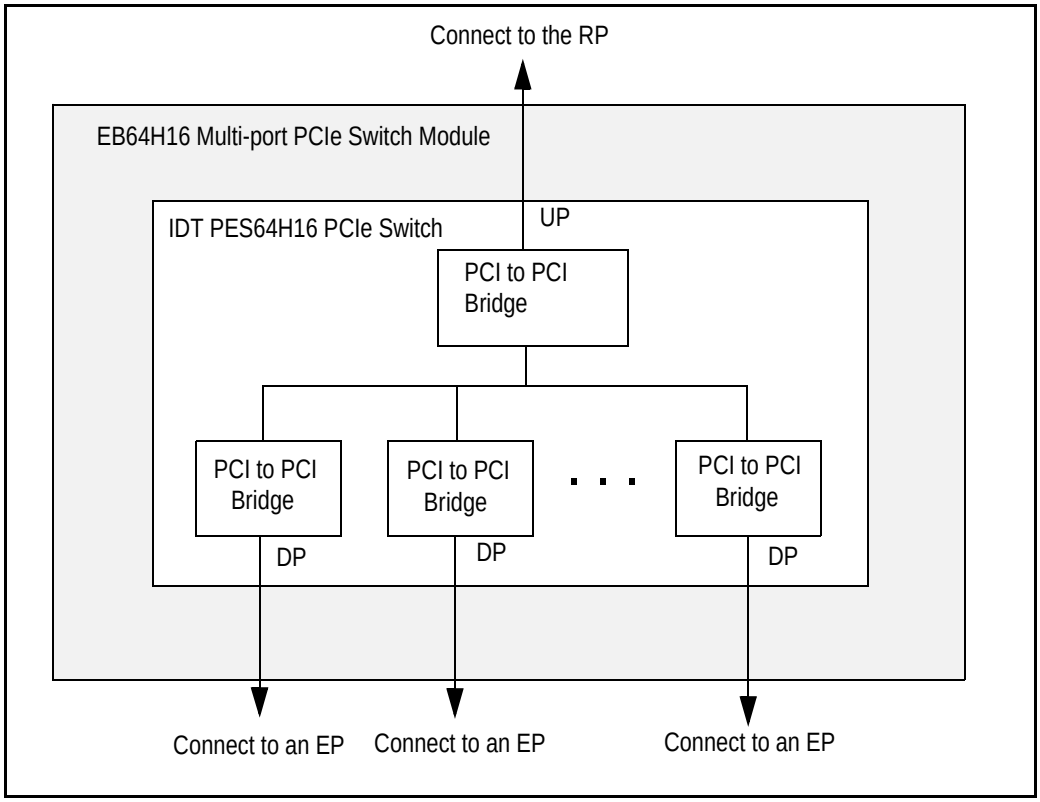

Figure 3 Multi-port PCIe Switch Module

Endpoint Processor

The Intel® IQ80333 I/O Processor Reference Board [5] is used as an Endpoint Processor (EP). This board has the form factor of a PCI-Express card which can be plugged into the EB64H16 PCIe slot directly. The system block diagram of the IQ80333 I/O Processor Reference Board is shown in Figure 4. A Gigabit Ethernet MAC controller is attached to the PCI-X interface on the IOP80333 I/O Processor. The Gigabit Ethernet is configured as a private device and can only be accessed from the local IOP80333 I/O processor.

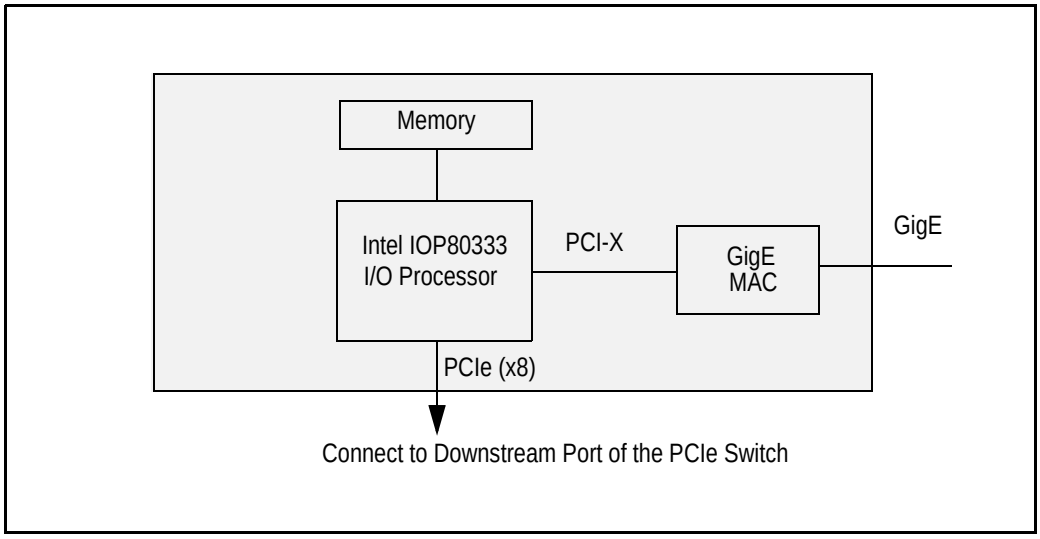

Figure 4 EP System Block Diagram

Notes

The functional block diagram of an IOP80333 is shown in Figure 5. The IOP80333 is a multi-function device that integrates the Intel XScale core with intelligent peripherals with dual PCI Express-to-PCI Bridges. There is a 2-channel DMA controller that is used to transfer data between the internal bus and the PCIe interface. The Address Translation Unit (ATU) implements the inbound and outbound address translation windows from/to the PCI-X/PCIe interface. The Message Unit implements the inter-processors communication mechanism such as inbound/outbound message and door bell registers. The Inbound and Outbound Queue Structures are also implemented in the Message Unit. Please refer to the Intel 80333 I/O Processor Developer's Manual[2] for a detailed description of the IOP80333 processor.

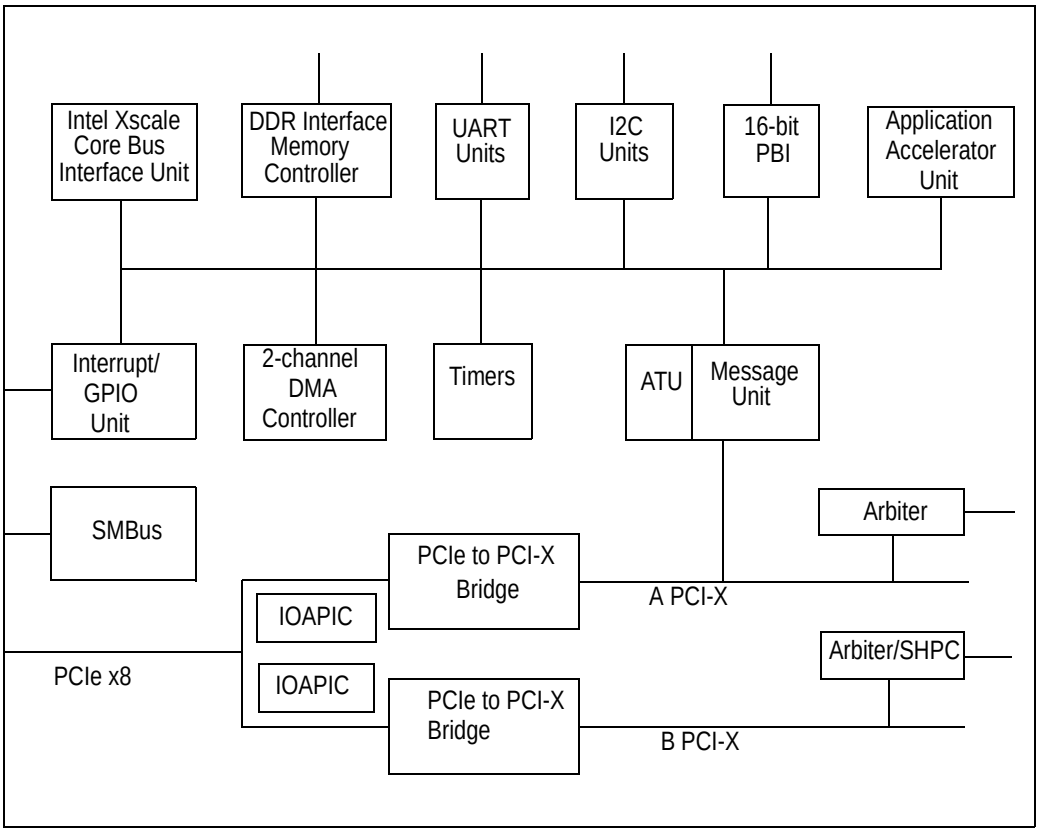

Figure 5 Intel IOP80333 Functional Block Diagram

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. The Transport Layer is hardware-dependent.

RP Software Architecture

The RP software runs under Linux Fedora 6. The software architecture is shown in Figure 6. The Function Service Layer provides the device driver functions to the Linux kernel. In this example, four function services are identified. 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. RP sends and receives Ethernet packets through this function service. The Disk Function Service provides a virtual disk interface. The Disk Function Service is not implemented for the current version of the software. The Configuration Function Service provides functions for system management purpose.

Notes

The Message Frame Service encapsulates the function service data in a common Message Frame header on transmit. 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.

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 current software release supports the IOP80333 as an EP and a x86 as an RP.

The local Architecture Service provides a hardware abstract service. It provides a hardware independent service to the Message Frame Service Layer and the Transport Service Layer, such as the translation between virtual and physical memory address, translation between local and system domain address and the hardware-specific DMA service. If the local device supports a DMA engine, the Local Architecture Service provides the device driver to the DMA engine. If the local device does not support a DMA engine, the Local Architecture Service simulates a DMA engine by doing a memory copy operation in software.

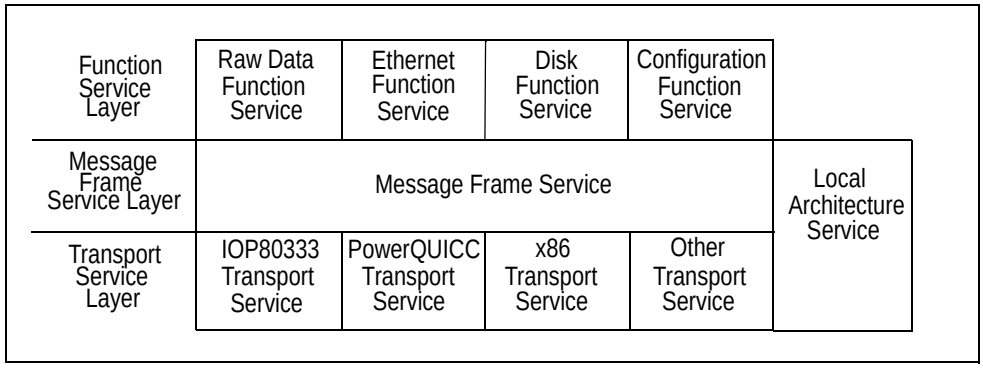

Figure 6 RP Software Architecture

EP Software Architecture

EP software architecture is shown in Figure 7. 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. For example, when the EP is IOP80333, all the local EP specific services are for the local IOP80333 implemented in a single IOP80333 EP device driver module.

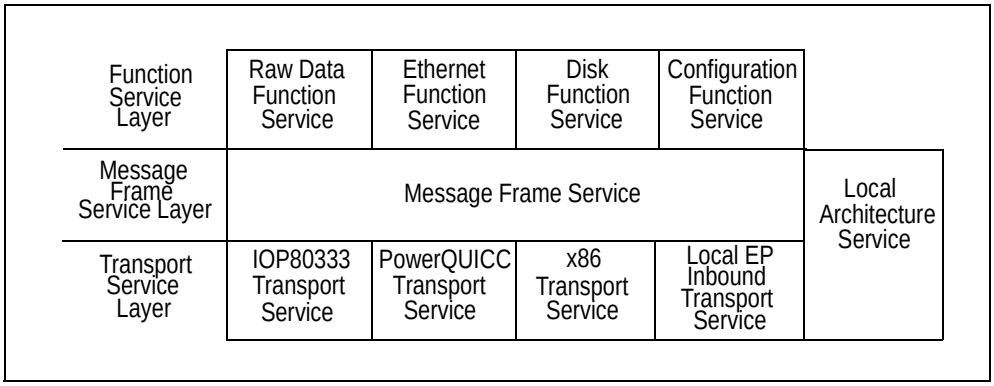

Figure 7 EP Software Architecture

Notes

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 IOP80333 Transport Service implements the outbound transport service towards an IOP80333 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.

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. Between the time a peer is added and the association between the peer and its Transport Service is made, the function services may receive messages from this new peer but are 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.

Application Examples

An I/O sharing application example is shown in Figure 8. In this system, there is an Ethernet interface in 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.

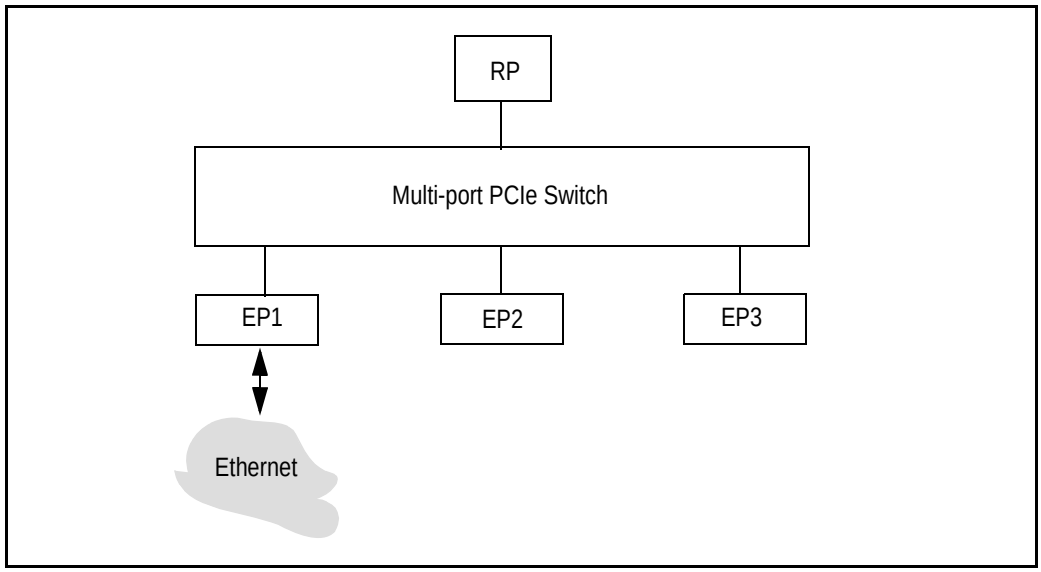

Figure 8 I/O Sharing Application Example

The protocol diagram of the sharing of Ethernet is shown in Figure 9. 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.

Notes

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.

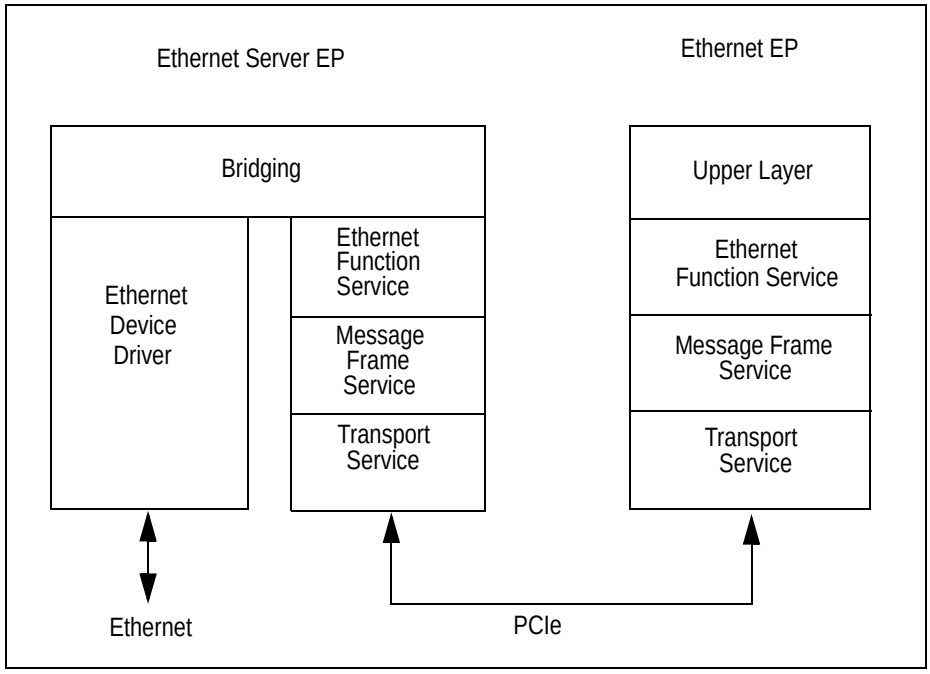

Figure 9 Ethernet Sharing Protocol Diagram

A network router application example is shown in Figure 10. 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.

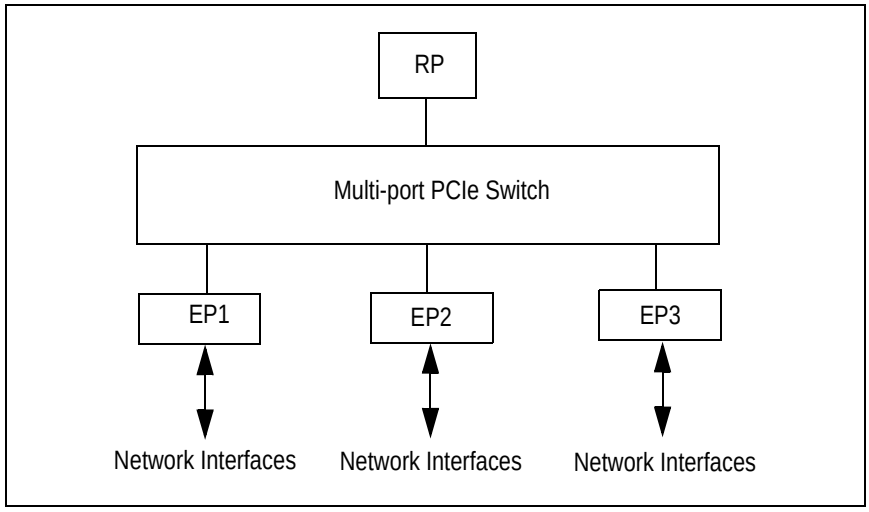

Figure 10 Router Application Example

Notes

The protocol diagram of the router application example is shown in Figure 11. 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.

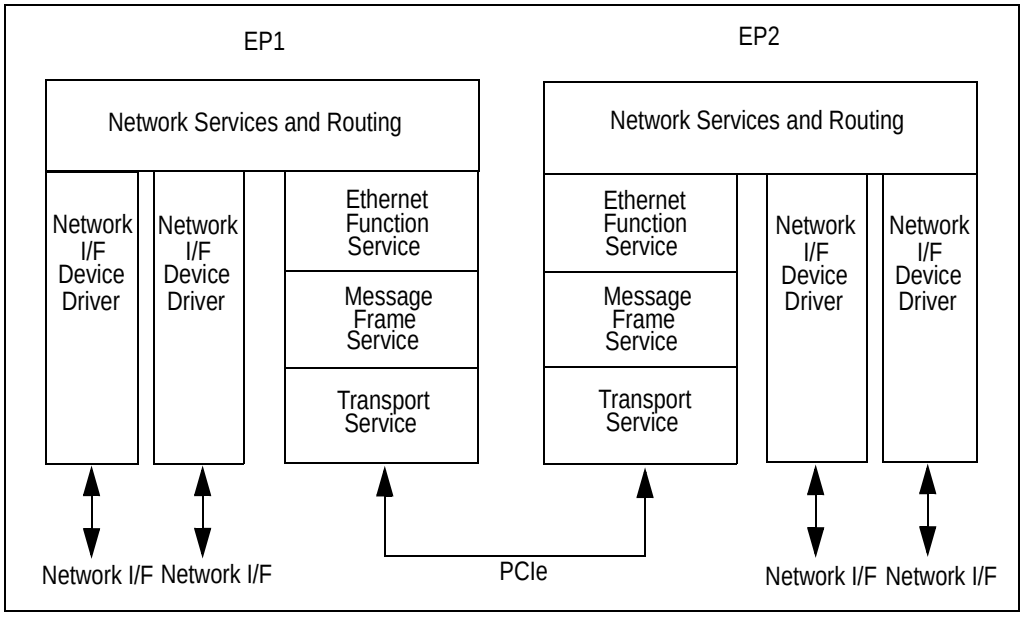

Figure 11 Router Protocol Diagram

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 EP.

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.

Inbound Address Translation

The IOP80333 supports 4 inbound address windows. This design uses a single Inbound Address Window. Inbound Address Window 0 is used. The first 4 Kbytes address of the Inbound Address Window 0 is reserved for the Message Unit. The Windows size is configured to be 1 Mbyte. 1 Mbyte window size is chosen for this implementation to provide a total of 510 buffers. The window size may be increased or decreased to provide the optimal number of buffers for a particular application. The Inbound Address Window is mapped to the EP's local data buffers. These local buffers are used by the RP and other remote EPs to send data to this local EP.

Notes

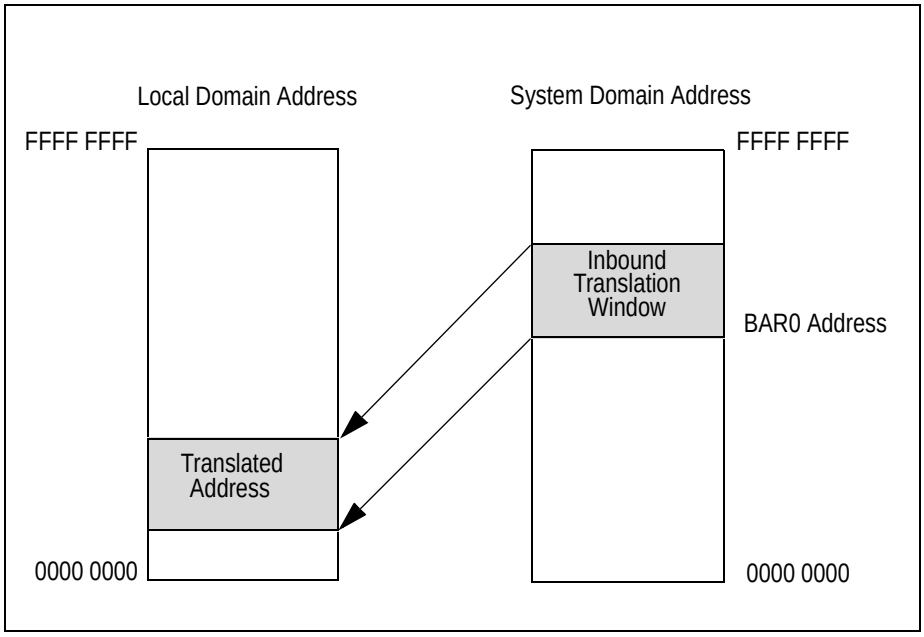

Figure 12 Inbound Address Window Translation

The Inbound Address Window is shown in Figure 12. A system domain address window is set up by the RP. During normal system initialization, the RP scans the BAR0 register of the EP's configuration registers to determine the size of the memory window required. The RP then assigns a system domain address to this EP by writing the system domain address to BAR0. The EP configures the translated address window on its local Inbound Address Translation Unit. Once the Inbound Address Window Translation initialization is completed, remote RP and EPs can access the local EP's translated address window directly. All address ranges outside the local EP's translated address window are hidden from the RP and remote EPs.

The RP sets up a unique Inbound Translation Window for each EP in the system.

Outbound Address Translation

The IOP80333 supports up to 3 outbound address windows. This design uses one outbound address window for the system interconnect communication. The outbound address window is set up to access all the remote EPs. The Outbound Address Translation set up is shown in Figure 13. The outbound address window 1 is used for the address translation. The size of the outbound address window is fixed at 64 MBytes for the IOP80333. For a system that has up to 15 IOP80333 as EPs, only 16 MBytes of system domain address space is used. When the local EP needs to access a remote EP's queue structures, the local EP performs a memory read/write from/to on its local EP Outbound Address Window. The local Address Translation Unit forwards the local memory access to the system domain and the local domain address is translated to the system domain address to access a remote EP. The queue structure to send data from a local EP to a remote RP is resident on the local EP. The local EP does not need to set up an outbound address translation window to the RP in order to access the queue structure.

The IOP80333 supports a DMA engine to transfer data between its local and the system address domains. When an address is setup in the DMA engine, it must specify if the address is in the local or system domain. When a local EP needs to transfer data to the RP, it sets up the local DMA engine to do the memory transfer from its local memory to the RP's memory. When the DMA engine is set up to transfer data to the RP, a local domain is specified in the source address and a system domain is specified in the destination address. The local EP uses the DMA engine to transfer data into RP's memory such that no outbound address window is required. The same DMA engine setup is used when a local EP transfers data to a remote EP.

Notes

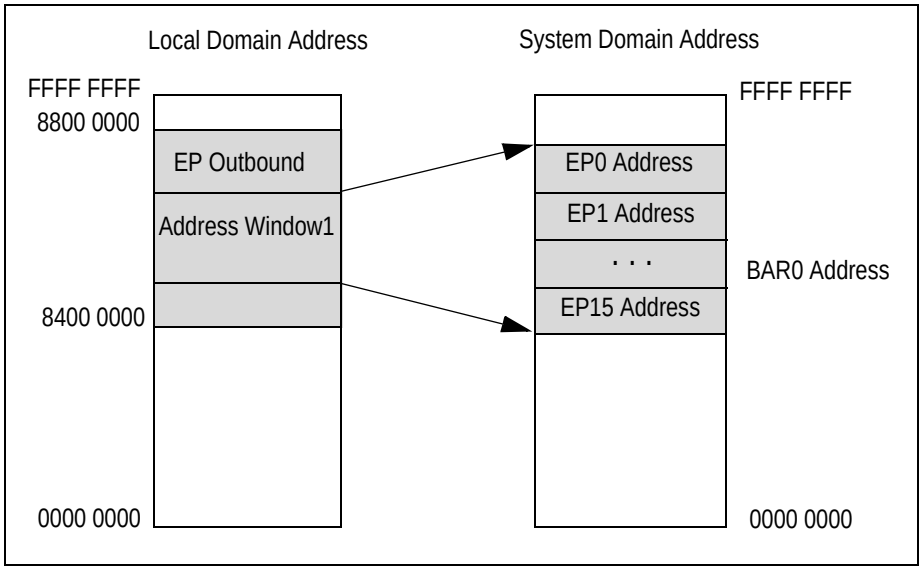

Figure 13 Outbound Address Translation

Data Transport

An EP sets up two sets of data transfer queues and uses its local DMA engine to transfer data to/from other EPs and the RP. The inbound queue structure is used to accept data message from other EPs and the RP. A local EP or RP uses the remote EP's inbound queue structure to transfer a data message to the remote EP. A local EP uses its outbound queue structure to transfer a data message to the RP.

There are two queues in the inbound queue structure: the Inbound PostQ and the Inbound FreeQ. These queues contain the Message Frame Address which is 32 bits wide. The local EP allocates local buffers in the Inbound Translated Address Window such that remote EPs have access to these buffers. These buffers are called Message Frames. The addresses of the Message Frame are added to the FreeQ. When a remote EP needs to send data to the local EP, the remote EP reads from the FreeQ to get a Message Frame first. The remote EP then transfers data to the Message Frame. After the data transfer is completed, the remote EP writes the Message Frame Address to the PostQ. The write operation by the remote EP to the PostQ causes an interrupt to the local EP. The local EP reads from the PostQ to get the Message Frame Address. It then processes the data in the Message Frame. The Message Frame is freed back to the FreeQ after the data is processed.

The IOP80333 implements the data transfer queues in hardware. A remote EP or RP performs a single memory read operation from a local EP's queue, then removes and returns the first Message Frame in the queue. A remote EP or RP performs a single memory write operation to a local EP's queue to add a Message Frame to the end of the queue.

Inbound Queues

The Inbound Queue Structure is shown in Figure 14. The local EP writes a free Message Frame Address to the FreeQ. The remote EP reads from the FreeQ when it needs to send data to the local EP. When the remote EP finishes transferring data to the Message Frame, it writes the Message Frame to the PostQ to indicate to the local EP that a Message Frame is ready to be processed. The local EP reads from the PostQ to get the next available Message Frame for processing.

A message frame size of 2 KBytes is used for this implementation. The size of the Inbound Address Window is 1 MByte. The first 4 KBytes of the Inbound Address Window is reserved for the Message Unit and hence there are 510 message frames on each IOP80333 EP. The minimum queue size that can be configured in the IOP80333 is 4K and hence a queue size of 4 K is chosen for this implementation. As the queue size for each PostQ and FreeQ is 4 K which is bigger than the number of message frames, there is no possibility of queue overflow.

Notes

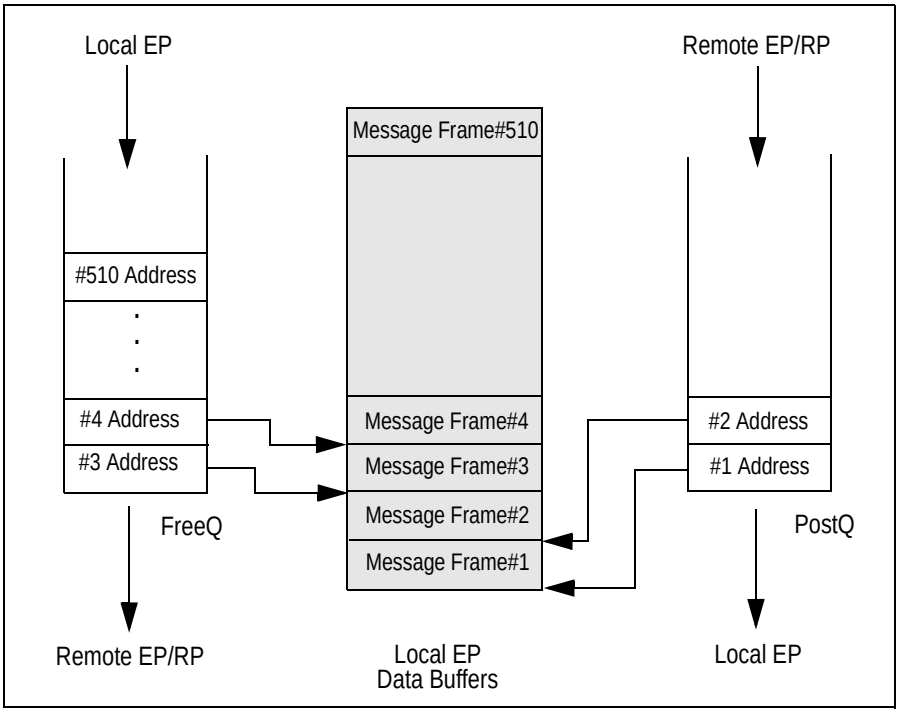

Figure 14 Inbound Queue and Date Buffer Usage

The EP to EP communication is a push model. The sending EP is responsible for transferring the data to the target EP, i.e., the sending EP pushes data to the target. The local IOP80333 uses its DMA engine to transfer data to a remote EP.

A data transfer protocol example is shown in Figure 15. In this example, a local EP needs to send a block of data to a remote EP. Local EP gets a free Message Frame by reading from the FreeQ in the remote EP. The local EP then writes the data to the Message Frame. After the data transfer is completed, the local EP writes the Message Frame Address to the PostQ in the remote EP. The remote EP reads the next available Message Frame from its PostQ and processes the data. Once data is processed, the remote EP returns the Message Frame to its FreeQ.

A local RP uses the same procedure to send data to a remote EP. Because the local RP does not support a local DMA engine, the data transfer has to be done via memory copy.

Notes

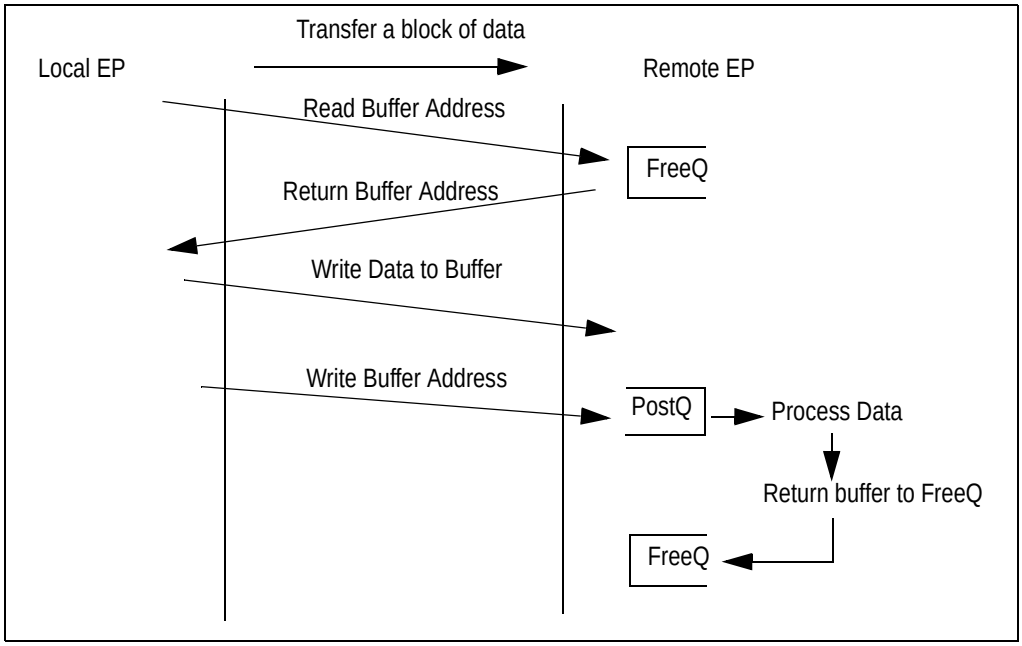

Figure 15 Data Transfer Protocol

Outbound Queues

The RP does not implement any queue structure in hardware. A local EP cannot send data to the RP using the RP's inbound queue. The outbound queue structure of the local EP is used instead. Each local EP uses its outbound queue structure to transfer data to the RP.

The outbound queue structure works very similar to the inbound queue structure. The local EP reads from the outbound FreeQ and writes to the outbound PostQ. The remote RP writes to the outbound FreeQ and reads from the outbound PostQ.

During initialization, the RP allocates Message Frames from its address space which is in the system domain. The RP writes the Message Frame Address to the outbound FreeQ of the remote EP. The RP needs to repeat the allocation of Message Frames for the EPs. A size of 64 KBytes is allocated for each EP. Each message frame is 2 Kbytes in size and hence 32 message frames are allocated per EP.

When a local EP needs to send a data message to a remote RP, it reads from the outbound FreeQ to get a Message Frame. The local EP then uses its DMA engine to transfer the data to the Message Frame. When the data transfer is completed, the local EP writes the Message Frame to its outbound PostQ. The RP reads from the outbound PostQ to process the incoming data. When the data is processed, the remote RP returns the Message Frame back to the outbound FreeQ.

Data Movement Examples

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

A local EP uses the inbound queue structure of a remote EP to transfer data to a remote EP. An example of the memory address translation window is shown in Figure 16. The local EP sets up an outbound address window at 0x8400 0000 and the window size is fixed at 64 Mbytes for an IOP80333.This window is divided into multiple 1 MByte window sizes. Each 1 MByte window is mapped to a single EP. The IOP80333 supports up to 63 remote EPs. This window is mapped to the system domain address starting at 0x8100 0000. In this example, the local EP address domain between 0x84100000 and 0x84200000 is

Notes

mapped to the system domain address between 0x81100000 and 0x81200000. The remote EP sets up its inbound address window to map the system domain address between 0x81100000 and 0x81200000 to its local address domain between 0x01000000 and 0x01100000. All these address ranges are setup at run time during system initialization.

When the local EP accesses the address within the outbound address window, the access is forwarded to the system domain. The address is translated to the address within the system domain address window. When the remote EP detects that the address is within its inbound address window, it forwards the access to its internal bus. The address is translated into the inbound address window. As an example, when the local EP writes to memory address 0x8418 0000, the write request is forwarded to the system domain. The address is translated to 0x8118 0000. When the remote EP detects that the address 0x8118 0000 is within its inbound address window, it forwards the write request to its local bus. The address is translated to its local address 0x0108 0000. The data is written to 0x0108 0000.

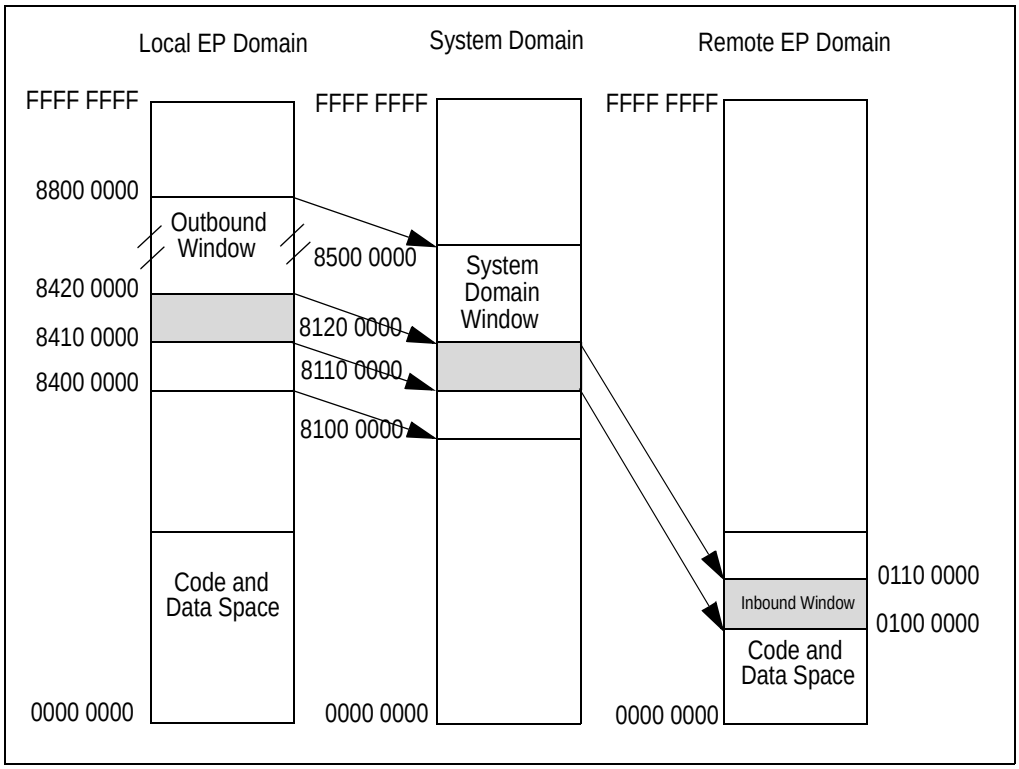

Figure 16 Address Translation Windows Example

Both the local and remote EP have their own code and data space. This space is private to the EP and can only be accessed by itself.

The inbound queue structure of the remote EP is used to send data from the local EP to the remote EP. For the Intel IOP80333, the first 4 Kbytes of the inbound address window is reserved for the Messaging Unit. The remote EP (IOP80333) address map is shown in Figure 17. The inbound queue port is at offset 0x40 which has the address of 0x01000040 in this example. The remote EP reads from the inbound queue port to get a Message Frame from the FreeQ and it writes to the inbound queue port to write a Message Frame to the PostQ.

The memory address space between 0x0100 1000 and 0x0110 0000 are for the Message Frame Buffers. The size of each Message Frame Buffer is 4 Kbytes and hence there are a total of 510 Message Frame Buffers.

Notes

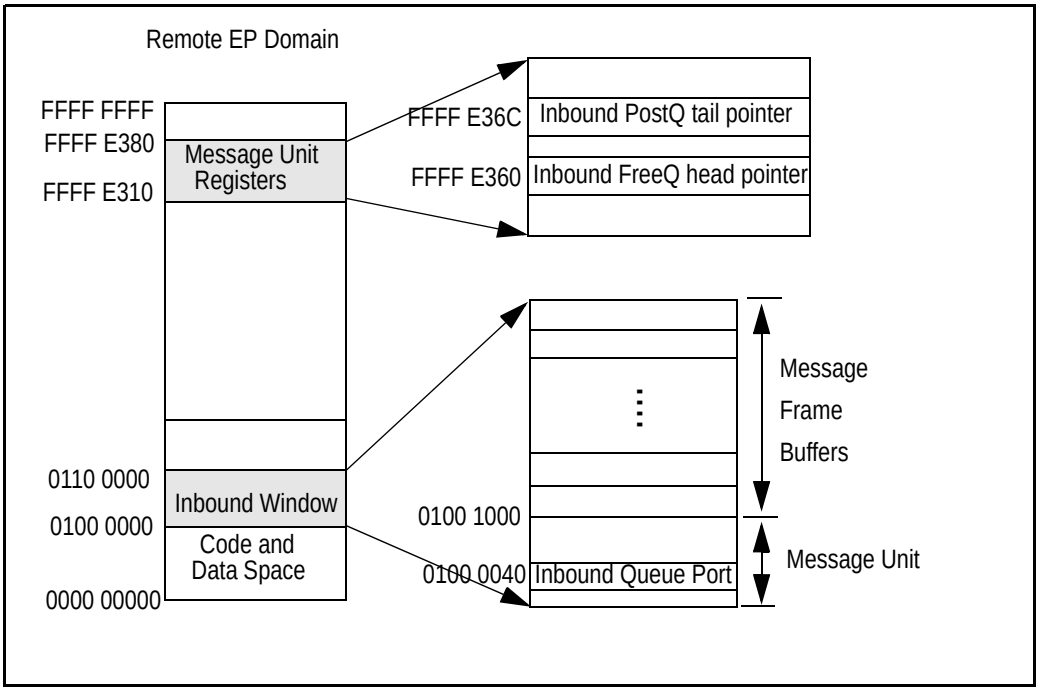

Figure 17 Remote EP Address Map

The remote EP does not use the queue port to access the FreeQ and PostQ. The remote EP (IOP80333) needs to use the queue head and tail pointers read/write from/to a queue. The Message Unit Registers of the IOP80333 are in the address range of 0xFFFF E310 to 0xFFFF E380. When the remote EP writes to a queue, it writes at the location pointed to by the queue head pointer. The queue header pointer is updated by the remote EP after the write access. When the remote EP reads from a queue, it reads from the location pointed to by the queue tail pointer. The queue tail pointer is updated by the remote EP after the read access.

The remote EP reads from the location pointed to by the inbound PostQ tail pointer to get a Message Frame from the PostQ for further processing and it writes to the location pointed to by the inbound FreeQ head pointer to add a Message Frame to the FreeQ.

In this example, the local EP reads from the address 0x84100040 to get a Message Frame from the inbound FreeQ and writes to the address 0x84100040 to add a Message Frame to the inbound PostQ of the remote EP. The remote EP reads from the location pointed to by the address 0xFFFF E36C (Inbound PostQ tail pointer) to get the next available Message Frame from the inbound PostQ and writes to the location pointed to by the 0xFFFF E360 (Inbound FreeQ head pointer) to add a Message Frame to the inbound FreeQ.

During initialization, the remote EP divides its memory at the inbound address window into multiple Message Frames. It then adds the Message Frames to the inbound FreeQ. 510 Message Frames are created in the inbound address window of the remote EP. As the first 4 Kbytes are reserved for the messaging unit, the first Message Frame starts at address 0x0100 1000 at the local address domain. The address has to be translated into the system domain address space such that other EPs can access it directly. The associated system domain address 0x8110 1000 is written into the freeQ instead. The FreeQ contains 510 Message Frames and the PostQ is empty after initialization

The steps to transport a block of data from a local EP to a remote EP are:

- The block of data is in the code and data space of the local EP domain.
- Local EP reads from the inbound FreeQ of the remote EP to get a Message Frame. (Read the address 0x84100040)
- The Message Frame 0x81101000 is returned to the local EP. The Message Frame address is in the system domain. The local EP translated the address to its outbound address window 0x84101000.

Notes

- The local EP sets up its local DMA engine to transfer the block of data from its local code and data space to the address 0x84101000. (The IOP80333 allows an address to be specified as in the system or local domain. The actual implementation sets up the DMA engine to transfer data directly to the address 0x81101000 and hence the local outbound address translation unit is bypassed.)
- As the address 0x84101000 is within the outbound address window, the address translation unit forwards the DMA request to the system domain. The address is translated to 0x81101000.
- The remote EP detects that the memory request address is 0x81101000 which falls into its inbound address window. The address translation unit on the remote EP forwards the memory request to its local bus. The address is translated to 0x01001000.
- When the DMA transfer is completed, the local EP adds the Message Frame 0x81101000 to the inbound PostQ of the remote EP. It writes 0x81101000 (system domain address) to the address 0x84100040 (remote EP's inbound PostQ port address in local address domain).
- When remote EP detects that the inbound PostQ is not empty, it reads the location pointed to by the tail pointer of the inbound PostQ to remove the Message Frame from the inbound PostQ.
- The remote EP allocates a buffer in its code and data space and copies the data from the Message Frame to the newly allocated buffer. The data buffer is posted to the appropriate application for further processing.
- The Message Frame 0x0100 1000 is returned to the inbound FreeQ by writing 0x81101000 (system domain address) to the location pointed to by the head pointer of the inbound FrereQ.

From a local EP to a remote RP

A local EP uses its outbound queue structure to transfer data from itself to the remote RP. An outbound address translation window is not set up in the local EP. The local EP always uses its local DMA engine to access memory in the remote RP.

During system initialization, the software that runs on the RP allocates a block of memory in the RP's local data space. This is the system domain address window. In this example, this window is between 0x71000000 and 0x71010000. A total of 64 KBytes of memory is reserved for data transfer Message Frames. The RP needs to allocate a separate block of memory for Message Frames for each EP in the system. The remote RP divides its memory into 32 Message Frames. It then adds these Message Frames to the outbound FreeQ of the local EP by writing to the outbound queue port of that local EP.

The steps to transport a block of data from the local EP to a remote RP are:

- The block of data is in the code and data space of the local EP domain.
- The local EP reads from the local outbound FreeQ to get a Message Frame. (Read from the location pointed to by the tail pointer of the FreeQ.)
- The Message Frame 0x7100 0000 is returned to the local EP.
- The local EP sets up its local DMA engine to transfer the block of data from its local data space to the address 0x7100 0000. The destination address is configured to be in the system domain.
- When the DMA transfer is completed, the local EP adds the Message Frame (0x7100 0000) to its outbound PostQ.
- The remote RP reads from the local EP's outbound PostQ to get the Message Frame. (0x7100 0000).
- The remote RP allocates a buffer in its code and data space. It then copies the data from the Message Frame to the newly allocated buffer. The data buffer is posted to the appropriate application for further processing.
- The remote RP returns the Message Frame 0x7100 0000 to the outbound FreeQ of the local EP.

From a Local RP to a Remote EP

The local RP uses the inbound queue structure of a remote EP to transfer data from a local RP to the remote EP. The procedure is the same as transferring data from a local EP to a remote EP. However, the DMA engine is normally not implemented on the RP to transfer data from its local memory to an external PCIe device. The RP has to copy the data from its local memory to the Message Frame in the remote EP.

Notes

Software Modules

All system interconnect system software components are implemented as Linux loadable modules. There are five and six Linux loadable modules for the RP and the IOP80333 EPs, respectively. The modules for the RP are idt-mp-i386-msg.ko, idt-mp-i386-arch.ko, idt-mp-i386-iop333.ko, idt-mp-i386-eth.ko, and idt-mp-i386-raw.ko. The modules for the IOP80333 EPs are idt-mp-iop333-msg.ko, idt-mp-iop333-arch.ko, idt-mp-iop333-rp.ko, idt-mp-iop333-iop333.ko, idt-mp-iop333-eth.ko, and idt-mp-iop333-raw.ko. The idt-mp-i386-msg.ko and idt-mp-iop333-msg.ko are the message frame modules; the idt-mp-i386-arch.ko and idt-mp-iop333-arch.ko are the local architecture modules; the idt-mp-i386-iop333.ko, idt-mp-iop333-rp.ko, and idt-mp-iop333-iop333.ko are the transport modules; the idt-mp-i386-eth.ko and idt-mp-iop333-eth.ko are the virtual Ethernet modules; and the idt-mp-i386-raw.ko and idt-mp-iop333-raw.ko are the raw data transfer modules. Refer to Figure 18 for their relative positions in the software architecture. The Application Programming Interface (API) for the modules is described in a separated document[6].

Please note that even though the statistic function is conceptually a function service, it is implemented in the message service module. More specifically, this function is part of the idt-mp-i386-msg.ko and idt-mp-iop333-msg.ko Linux loadable modules. This function sends and receives high priority messages in order to have up-to-date statistical information. Also note that even though the RP and the IOP80333 EPs have separate binary Linux loadable modules, some of them share the same source files. More specifically, idt-mp-i386-msg.ko, idt-mp-i386-eth.ko, and idt-mp-i386-raw.ko share the same source files with idt-mp-iop333-msg.ko, idt-mp-iop333-eth.ko, and idt-mp-iop333-raw.ko, respectively.

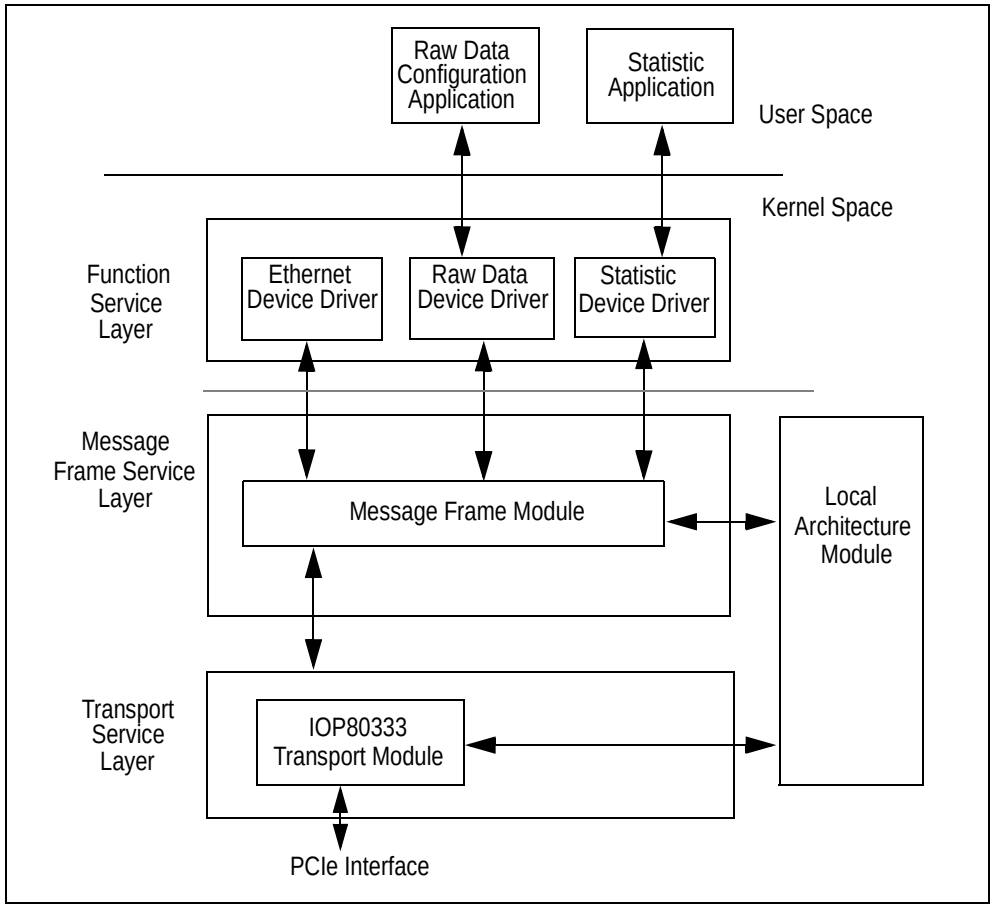

Figure 18 Software Modules and Device Drivers

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.

Notes

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.

Its 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.

Its 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.

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 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.

Notes

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 DMA transfer and address space conversion routines. Each different type of RP and EP has its own architecture-specific module and does not share source files. Depending on the capability of the hardware, the DMA transfer routines may utilize the hardware DMA engine or simulate the DMA transfer with memory copies. The address space conversion routines convert addresses between virtual, physical, bus, and PCI addresses.

Transport Service Layer

The Transport service is responsible for detecting and setting up the hardware, managing the frame buffers, and initiating the actual data transfers.

IOP80333 Transport Module

There are two separate IOP80333 transport modules which do not share source files. One module runs on the RP and the other runs on the IOP80333 EPs.

The IOP80333 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 IOP80333, 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 the IOP80333 to setup the memory windows and data frame buffers.

The module initialization function of the IOP80333 transport module running on the IOP80333 EPs initializes the transport data structure and registers itself with the message service. When the message service on the IOP80333 EPs receives an IOP80333 peer-add notification, it calls the peer_add function of the IOP80333 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 a data frame to an IOP80333 EP for both modules running on the RP and the EPs. The receive function of the IOP80333 transport service running on the RP is triggered from the interrupt handler when an IOP80333 EP sends a data frame to the RP. Note that there is no receive function in the IOP80333 transport running on the EPs. The data reception is handled by the RP transport module on the IOP80333 EPs.

RP Transport Module

The RP transport module runs on the IOP80333 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 IOP80333 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 a data frame to the RP. The receive function is triggered by the interrupt handler when any RP or EP sends a data frame to this IOP80333 EP.

System Initialization

Before the system is powered up, all EPs should be configured so that all accesses to their PCI configuration spaces are re-tried. Once the system is powered up, the PC BIOS running on the RP and all the boot-loaders running on the EPs start scanning and initializing the hardware at the same time. If the PCI BIOS running on the RP is faster than the boot-loader running on an EP, it will keep re-trying an EP configuration space access until the EP boot-loader clears the PCI configuration access retry condition. Once the EP configuration space access retry condition is cleared, PC BIOS continues the PCI bus scanning and resource assignment, and finally boots the x86 Linux kernel.

Notes

The Intel IOP80333 boot-loader sets up the Inbound ATU Limit Register 0 of the Address Translation Unit(IALR0) to request 1 MB of non-prefetchable memory mapping through Base Address Register 0 of the PCI Configuration Space(BAR0) of the EP from the RP. It also sets up inbound memory window 3 to map all local memory, outbound memory window 0, and outbound I/O window for the private PCI devices on the secondary PCI bus. The boot-loader then clears the PCI configuration space access retry conditions, continues the rest of the hardware initialization and finally boots the Intel IOP80333 Linux kernel.

The Intel IOP80333 Linux kernel initializes the Intel IOP80333 hardware and the private PCI devices. The Intel IOP80333 local message frame service inspects the IALR0 to determine the memory window requested through BAR0, allocates a corresponding memory buffer, and updates Inbound ATU Translate Value Register 0 of the Address Translation Unit(IATVR0) to point to the allocated buffer. It then initializes the message unit, populates the inbound free queue, and finally updates the Outbound Message Register 0 of the Message Unit(OMR0) with value 4096, the number of message queue entries configured. The Intel IOP80333 local inbound transport service initializes the message unit interrupts for inbound messages. The Intel IOP80333 local to RP transport service registers itself with the message frame service for outbound messages to the RP. The Intel IOP80333 local DMA service initializes the DMA channels and registers itself to provide low-level data transport services through DMA.

When the x86 Linux kernel detects an Intel IOP80333 device, the probe routine in the Intel IOP80333 PCI device driver is invoked. The probe routine initializes the new device data structure and reads the OMR0 of the newly discovered IOP80333 device. If the OMR0 is non-zero, the probe routine calls the Intel IOP80333 message queue initialization routine to populate the Intel IOP80333 outbound free queue up to the maximum number of entries indicated in the OMR0. It then updates the Inbound Message Register 0 of the Message Unit(IMR0) with the Intel IOP80333 device's ID, which encodes the bus, device, and function number of the device. If the OMR0 is zero, the Intel IOP80333 Linux is not ready and the probe routine returns immediately. The message queue initialization and IMR0 update is postponed until the Intel IOP80333 Linux updates the OMR0.

After the Intel IOP80333 outbound free queue is populated, the Intel IOP80333 device driver running on the RP notifies the message frame service of the newly discovered device. The message frame service is then responsible for notifying the other existing peers of the new device through their corresponding transport services. It also notifies the new device of the other existing peers through the new device's transport service. The notification should include the ID, architecture, and location in system memory map of the devices.

When the Intel IOP80333 hardware detects an IMR0 update, the Intel IOP80333 local inbound transport service would be notified through the associated interrupt. The Intel IOP80333 local inbound transport service updates the Outbound Memory Window Translate Value Register 1(OMWTVR1) and Outbound Upper Memory Window Translate Value Register 1(OUMWTVR1) of the Address Translation Unit with the 64 MB aligned address derived from the value assigned to the BAR0 register. It then notifies the message frame service to add the RP as its peer.

At this point, peer communication between the RP and the new EP is fully initialized and functional. The function services on both RP and EP can start sending and receiving data to each other. When another Intel IOP80333 device is discovered and initialized, the RP sends a notification message to this EP. The local EP inbound transport service passes the notification to the message frame service. The message frame service creates a new Intel IOP80333 peer and associates the new peer with the Intel IOP80333 transport service based on the type of the new peer indicated in the notification message, if the Intel IOP80333 transport service is already registered. Otherwise, it delays the association until the Intel IOP80333 transport service registers itself with the message frame service. The message frame service then notifies the function services of the new peer. Since the RP adds the new peer only after the communication between the RP and the new peer is initialized, the Intel IOP80333 transport service can start sending messages to the new peer when it is notified of the new peer's existence.

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 IOP80333 as the Endpoint Processors. The 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 functions of the software and allows 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: 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 decapsulation layer to all the function services. It also notifies the newly discovered Endpoint Processors to all other Endpoint Processors. 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 software supports the x86 as the Root Processor and the IOP80333 as the Endpoint Processor.

Apart from inter-processor communication application, this software demonstrates that IO sharing can now be implemented using a standard PCIe switch and off the shelf IO processors. 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 IO sharing, communication systems and storage system 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] Intel 80333 I/O Processor Developer's Manual, Document Number 305432001US, Intel

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

[4] IDT 89HPES64H16 PCI Express User Manual.

[5] Intel IQ80333 I/O Processor Customer Reference Board Manual, September 2005.Document Number: 306690003US.

[6] System Interconnect Software Programming Interface, Steve Shih, IDT.