PCI Express® System Interconnect Software Architecture for PowerQUICC™ III-based Systems

Application Note AN-573

Notes

By Craig Hackney

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 either an x86 or PowerQUICC III[4] based system as the Root Processor (RP) and a PowerQUICC III based system as the Endpoint Processor (EP) utilizing IDT's 89HPES24N3 multi-port PCIe® switch as 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 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 allow multiple Endpoint Processors (EPs) to connect to the system.

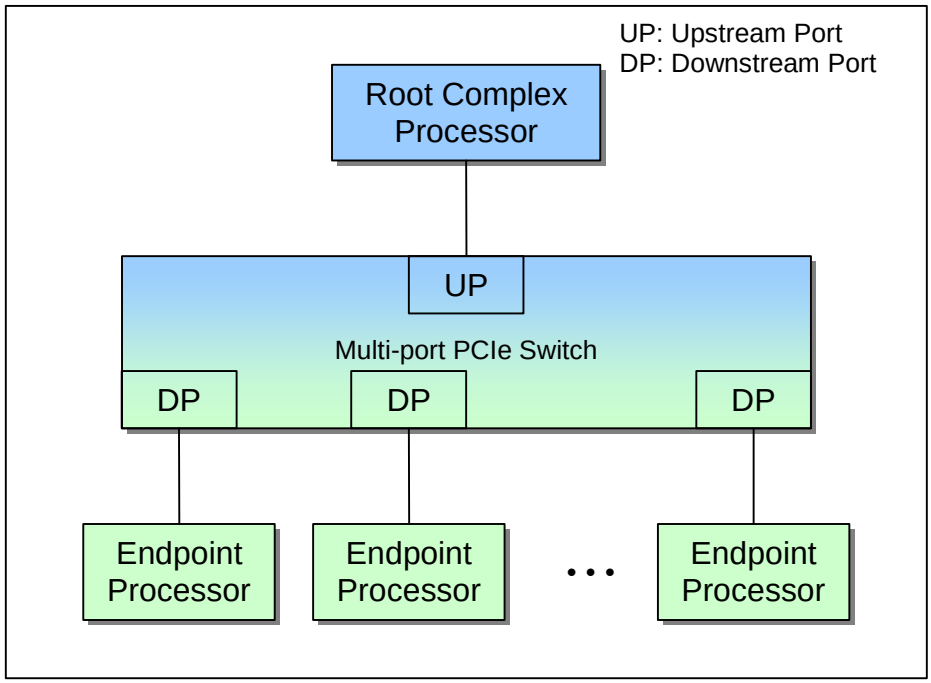

Figure 1 Multi-peer System Topology with a PCIe System Interconnect

Notes

Because the endian-ness of the RP and EPs may differ, all addresses communicated between the RP and EP's via the PCIe interface should be little endian. It is the responsibility of each peer to convert addresses from little endian to native endian format before using them.

Root Complex Processor

The x86 based RP uses an Intel Xeon CPU with the Intel 7520 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 x86 RP is shown in Figure 2.

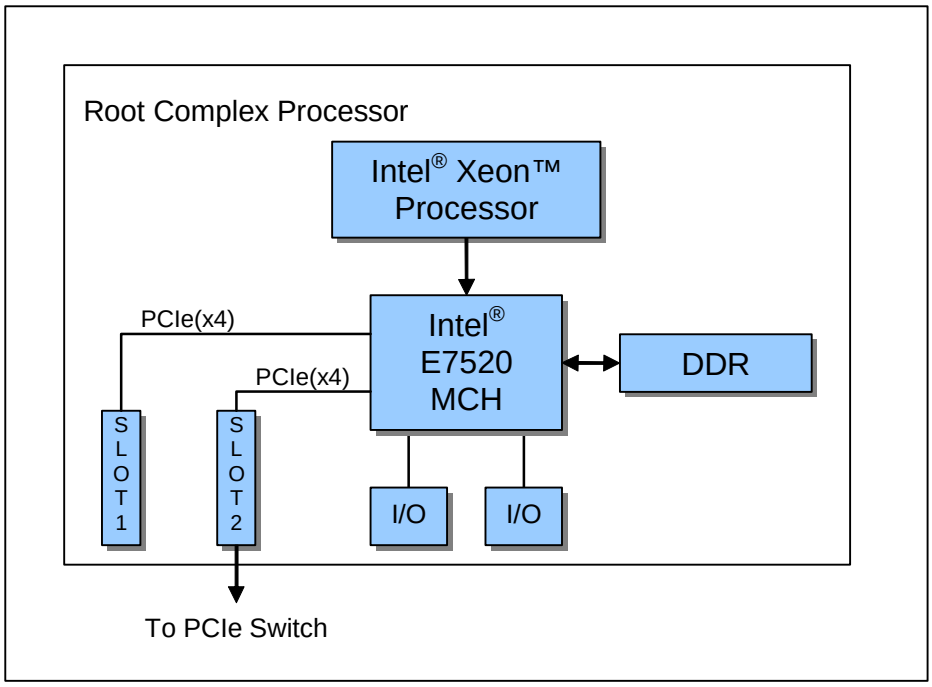

Figure 2 RP System Block Diagram

The PowerQUICC III based RP uses an EP8548A[5] with its PCIe interface configured to be a root complex. The RP's PCIe interface is connected to the upstream port of the Multi-port PCIe switch. The system block diagram of the RP is showin in Figure 4.

Notes

PCIe Switch

The IDT 89EBPES24N3 evaluation board[2] (referred to hereafter as EB24N3) is used as the multi-port PCIe switch module. The system block diagram of an EB24N3 is shown in Figure 3. The EB24N3 contains an IDT 89HPES24N3 PCIe switch[3] (referred to hereafter as PES24N3).

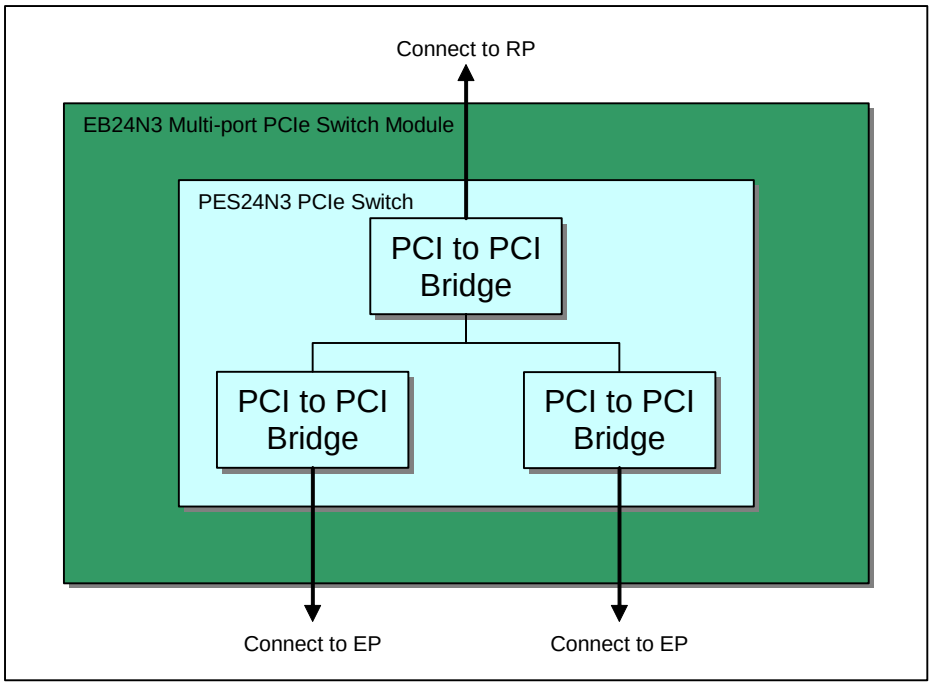

Figure 3 Multi-port PCIe Switch Module

Endpoint Processor

The EP Processor is an EP8548E with its PCIe interface configured to be an end point. Each EP connects to one downstream port of the Multi-port PCIe switch. The system block diagram of the EP is shown in Figure 4.

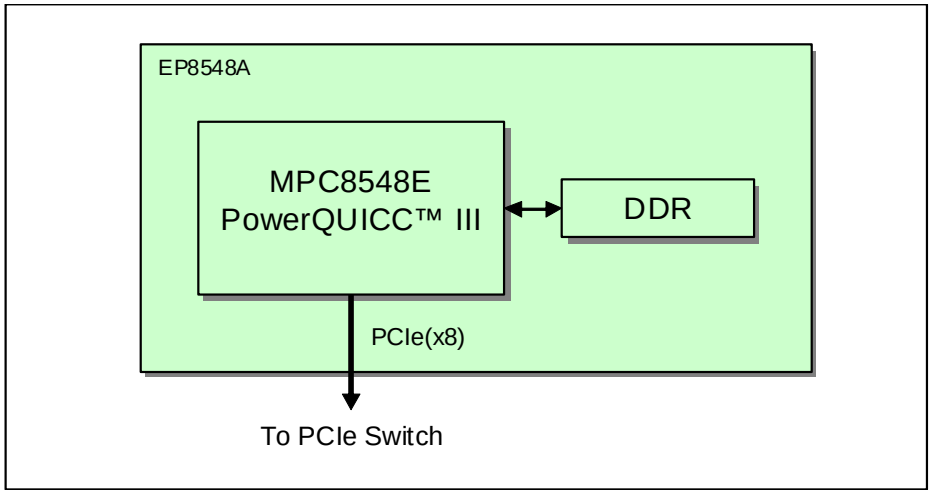

Figure 4 EP8548A System Block Diagram

Notes

General Software Architecture

The RP and EP software is very similar in design and implemented as loadable Linux modules. The software is divided into three layers:

- Function Service Layer
- Message Frame Service Layer
- Transport Service Layer

RP Software Architecture

The software architecture for the RP is shown in Figure 5. The Function Service Layer provides the device driver interface to the Linux kernel. In this example three function services are identified:

- Raw Data Function Service
- Ethernet Function Service
- Disk Function Service (Not supported in the current version of the software)

The Raw Data Function Service provides a service to exchange raw data between the RP and the EPs and is used primarily for benchmarking. The Ethernet Function Service provides a virtual Ethernet interface allowing the RP to transmit and receive Ethernet packets via PCIe, and the Disk Function Service provides a virtual disk interface allowing the RP to access disk services via PCIe.

The Message Frame Service encapsulates data from the Function Service Layer before passing it to the Transport Service Layer for transmission. Conversely data received by the Transport Service Layer and passed to the Message Frame Service is de-capsulated and passed to the Function Service Layer for further processing.

The Transport Service Layer is a hardware-dependent layer that provides the inbound and outbound transport services to the EPs in the system.

The Local Architecture Service provides hardware-independent data transfer services to the Message Frame Service Layer and the Transport Service Layer, as well as providing other services such as virtual to physical and local to system domain address translation.

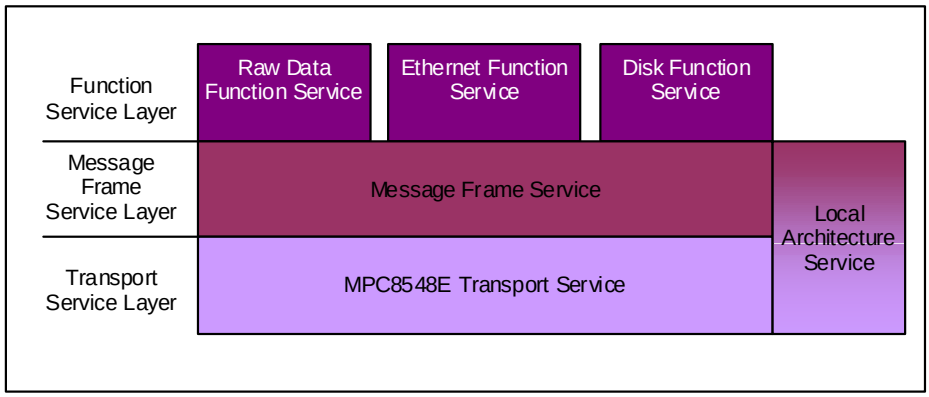

Figure 5 RP Software Architecture

EP Software Architecture

The EP software architecture shown in Figure 6 is essentially the same as the RP software architecture. The only difference is the addition of an EP to RP Transport Service that provides the outbound transport service to the RP as well as the inbound transport service from the RP and the other EPs in the system.

Notes

The EP-specific Transport Services implement the outbound transport service for the specific EP, i.e. the MPC8548E Transport Service implements the outbound transport service for an MPC8548E EP. It should be noted that all the inbound traffic goes through the EP to RP Transport Service while outbound traffic goes through one of the EP-specific transport services.

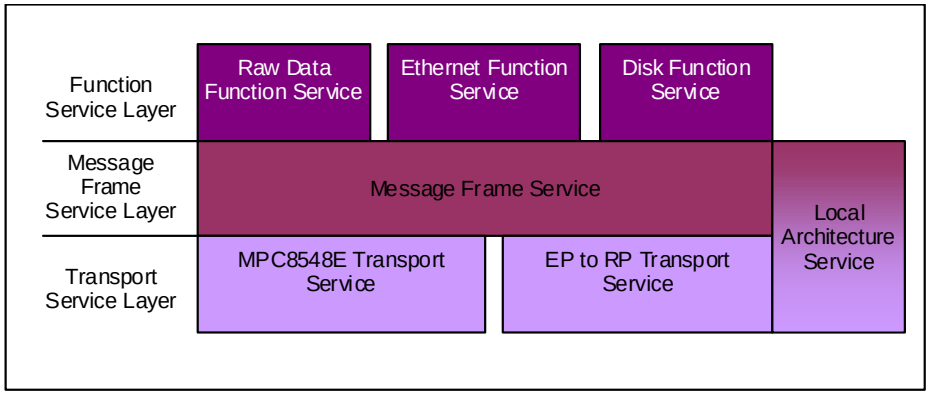

Figure 6 EP Software Architecture

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 the transport service for the newly added peer does not yet exist, the association of the peer with its transport service will be delayed until its transport service registers itself with 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 when the association is being made between a new peer and its Transport Service, the function services may receive messages from this new peer but they will be unable to respond. The function services may decide to delay the processing of these inbound messages, or process the messages immediately and queue the responses for transmission later.

The case where a specific EP transport service is supported in some peers but not others is not a supported configuration, thus it is not considered here.

Notes

Application Examples

In the I/O sharing example shown in Figure 7, the Root Processor and Endpoint Processor 2 both share the Ethernet interface on Endpoint Processor 1.

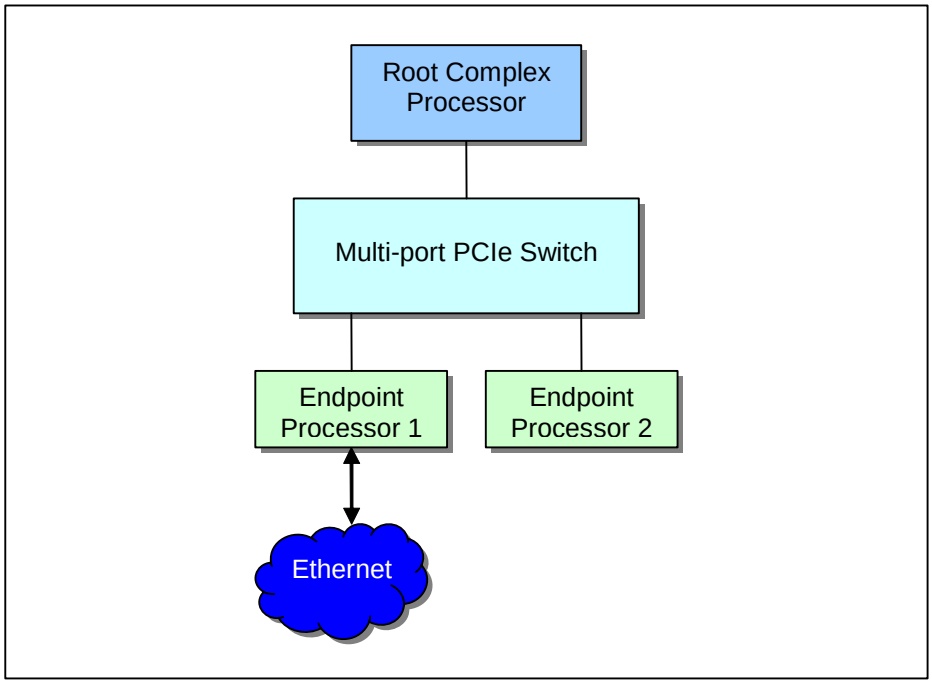

Figure 7 I/O Sharing Example Application

The protocol diagram for the Ethernet sharing application is shown in Figure 8. The Ethernet Service Function running on the Ethernet EP provides a virtual Ethernet interface to the Upper Layer application, such as a TCP/IP protocol stack. As far as the Upper Layer is concerned, this 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 Ethernet Server EP provides the physical Ethernet connection to the network. It uses the Ethernet Function Service on its PCIe interface to send/receive Ethernet packets to/from other EPs and the RP and uses a Bridging Application to forward Ethernet packets between the Ethernet Function Service and the physical Ethernet interface. The Ethernet Bridging Application may be replaced with an IP Routing Application, such that IP packets are routed between the Ethernet Function Service and the physical Ethernet interface. This type of system topology allows multiple EPs to share a single Ethernet Server and, therefore, the physical Ethernet interface.

Notes

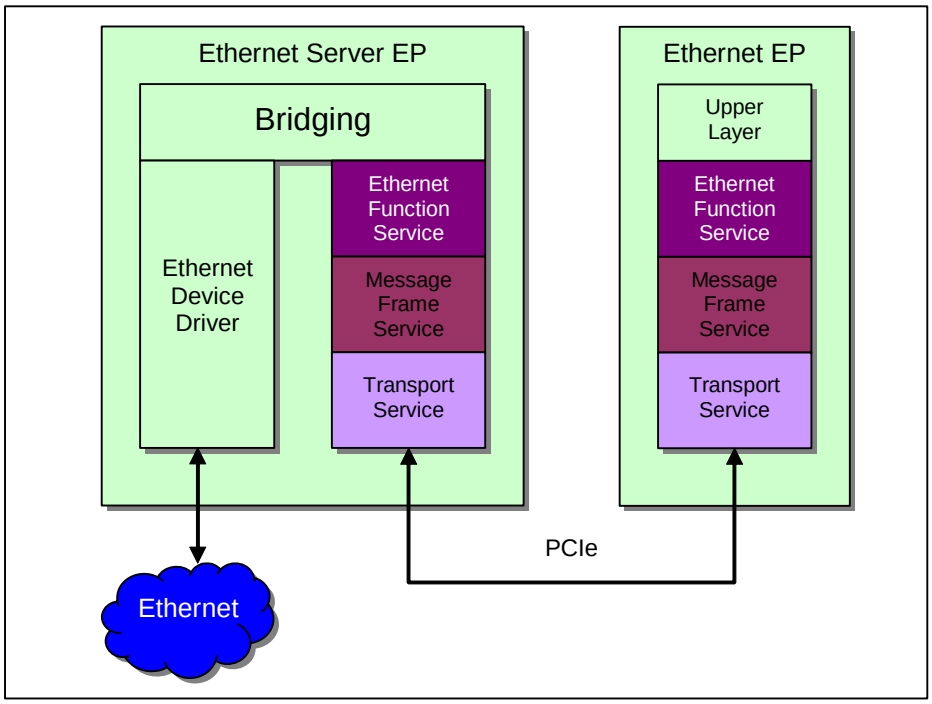

Figure 8 Ethernet Sharing Protocol Diagram

In the network router application example shown in Figure 9, each EP supports one or more network interfaces. 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.

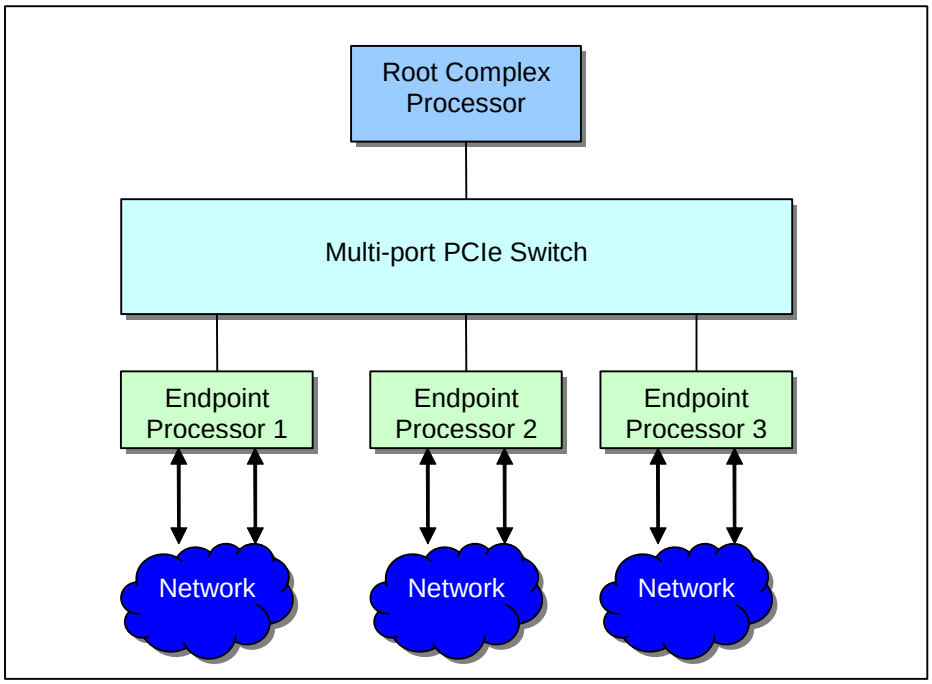

Figure 9 Router Example Application

Notes

Figure 10 depicts the protocol diagram for the router application. In this example, packets are received via an EP's local network interface and are passed to the Network Services & Routing application. The Network Services & Routing application decides if the destination for the received packet is a network interface on the local EP or a network interface on another EP in the system. If the packet destination is a network interface on the local EP, it is forwarded to the local network interface for transmission. If the packet destination is a network interface on another EP in the system, it is sent to the Ethernet Function Service where it is forwarded to the appropriate EP. Upon receiving this forwarded packet, the Network Service & Routing application on the destination EP will forward the packet to the appropriate network interface for transmission.

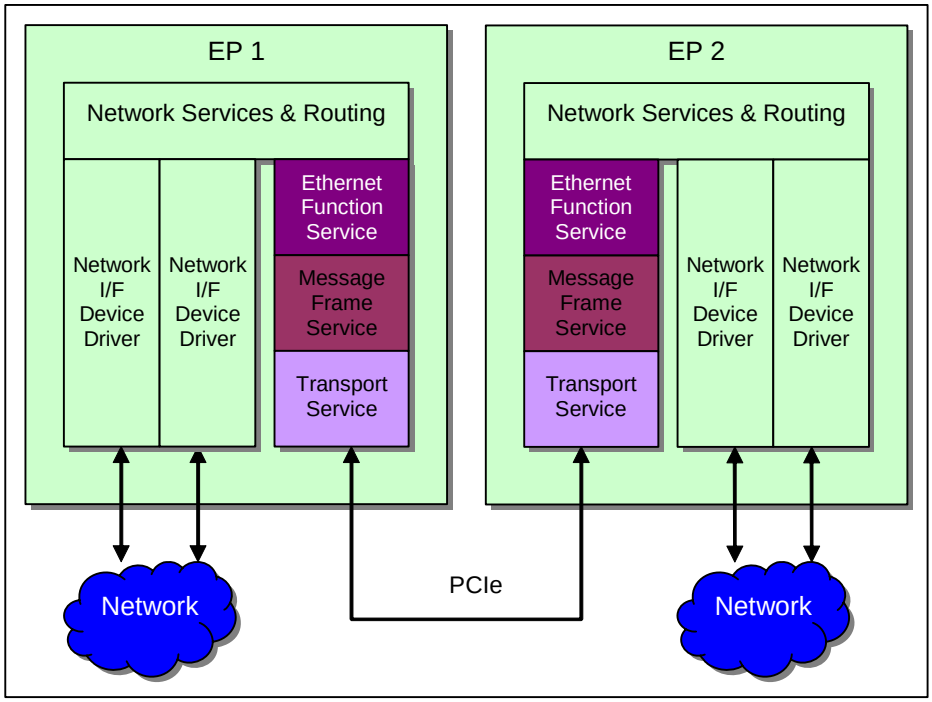

Figure 10 Router Protocol Diagram

Address Translation

There are two address domains in a multi-peer system using PCIe as the System Interconnect: the System Domain and the Local Domain. The System Domain is the global address map as seen by the RP. The Local Domain is the address map as seen by each EP. These two domains are independent of each other. The RP is free to assign address space in the System Domain, and each EP can freely assign address space in its Local Domain.

Address Translation is used to bridge between the two domains. Using Figure 11 as a reference, if Peer #1 wanted to transfer data to/from Peer #2 it would access the outbound translation address window in its Local Domain[1] that corresponds to Peer #2. These accesses would be translated to the System Domain[2] and, because this System Domain address matches the inbound translation address window defined by Peer #2, the address is translated to the Local Domain[3] of Peer #2.

Notes

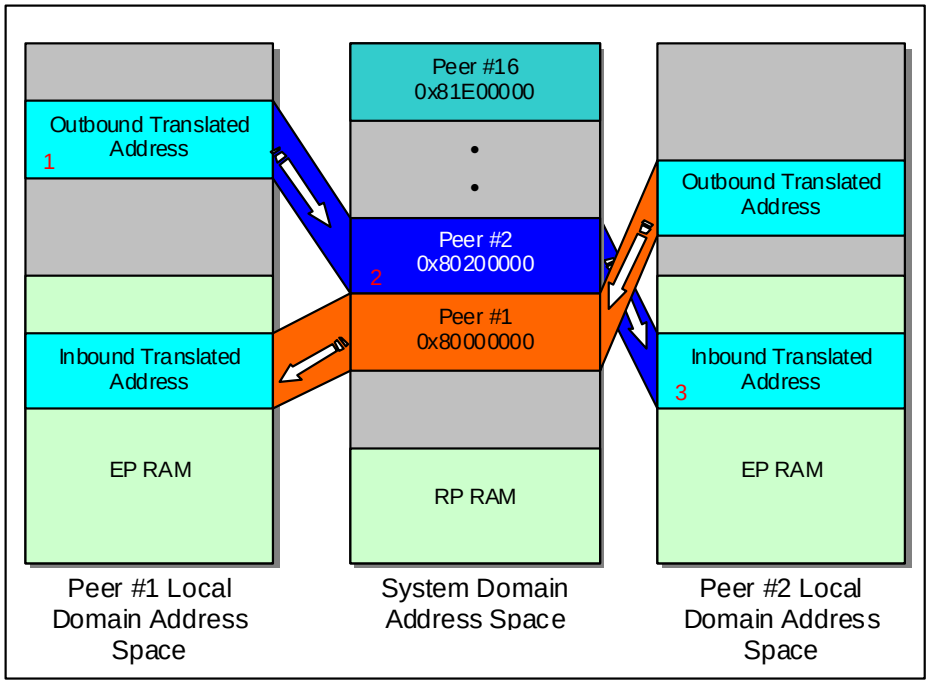

Figure 11 Address Translation

When the RP boots, it scans the PCIe bus and enumerates the endpoints. 1MB of System Domain address space is allocated to each endpoints BAR0.It is this System Domain address that is used, among other things, for remote peers to access the EP's doorbell registers. Another block of System Domain address space is allocated to BAR1 of the EP. This address range represents the System Domain address of the EP's queue structures.

Notes

Inbound Address Translation

Two inbound address translation windows are configured by each EP. Inbound Window Base Address Register 0 (IWBAR0) is configured to map 1MB of System Domain address space to the internal registers of the MPC8548E. IWBAR1 is configured to map the System Domain address space pre-allocated by the RP to local data space. The local data space contains the queue structures and data buffers for the data transport between the local EP and the remote EPs and RP. Any access to the system Domain Address Space for Peer #2 will result in an access to the Local Domain Address Space for that peer as defined by the Inbound Translation Address window.

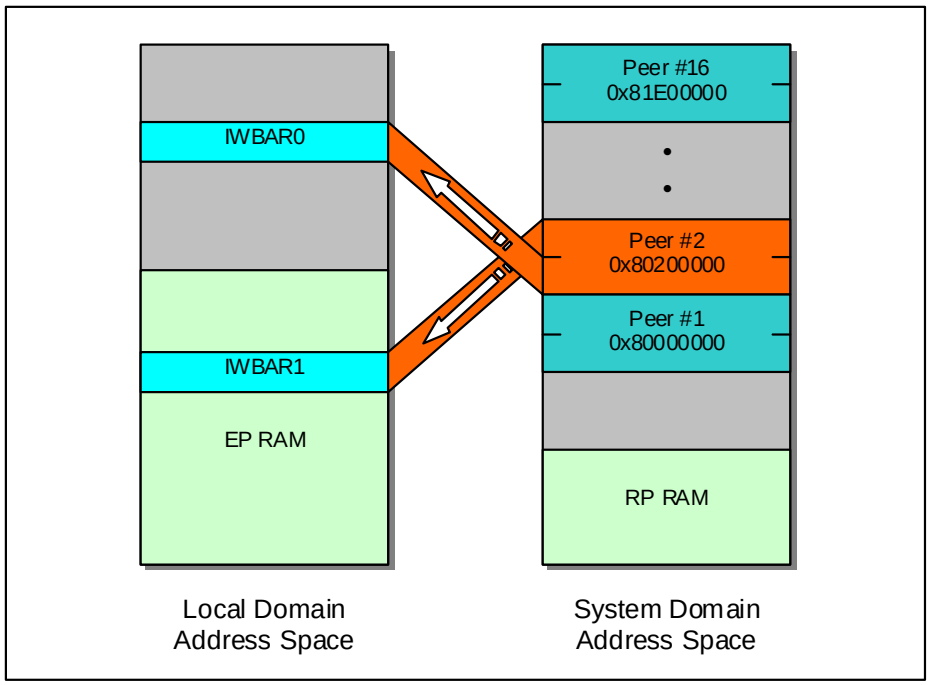

Figure 12 EP Inbound Address Translation Windows

Notes

Outbound Address Translation

Each EP requires three outbound address translation windows. The first specified by Outbound Window Base Address Register 1 (OWBAR1) is 4KB in size and is mapped to the Message Signaled Interrupt (MSI) register of the RP. The second specified by OWBAR2 is mapped to the local data space of the RP. The third specified by OWBAR3 is 64MB in size and is mapped to the data space of the other EP's in the system. An example of the outbound address translation window setup is shown in Figure 13.

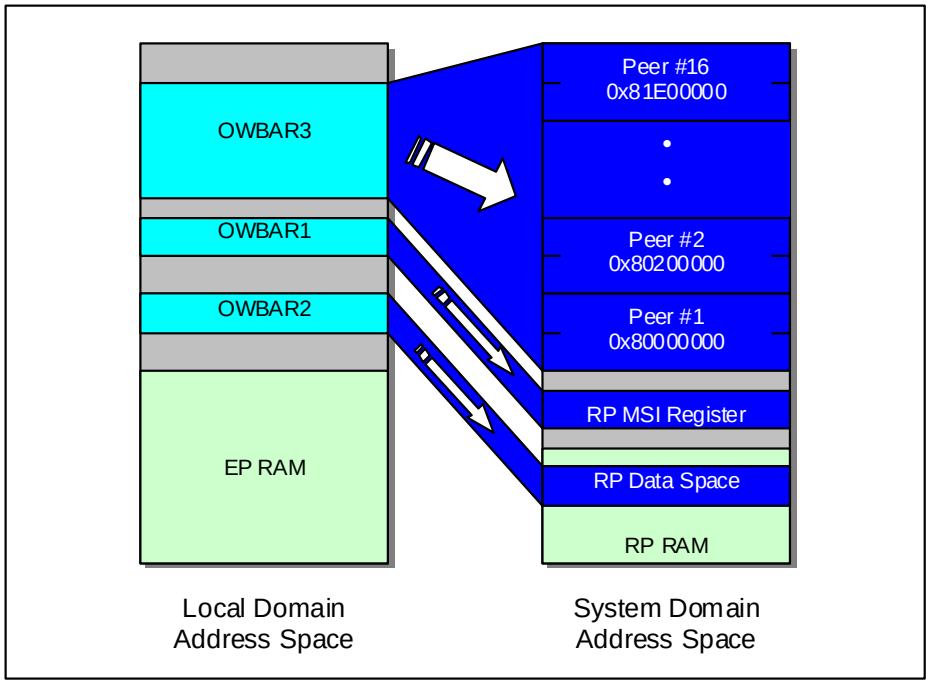

Figure 13 EP Outbound Address Translation Windows

Notes

Data Transport

Each EP sets up one Inbound Queue Structure for every peer in the system. This Queue Structure is used to receive data from the remote peer. Each queue structure consists of two queues, a PostQ and a FreeQ. Each entry in the queue is the address of a data buffer called a Message Frame, the *Buffer Size* entry in the queue structure specifies the length of the Message Frames for this queue. The Inbound Queue Structure, PostQ, FreeQ, and associated Message Frames are allocated within the inbound translation address window to allow access from remote peers. The total size of the inbound translation address window required by the Queue Structures and Message Frames is specified by the *Window Size* entry in the queue structure.

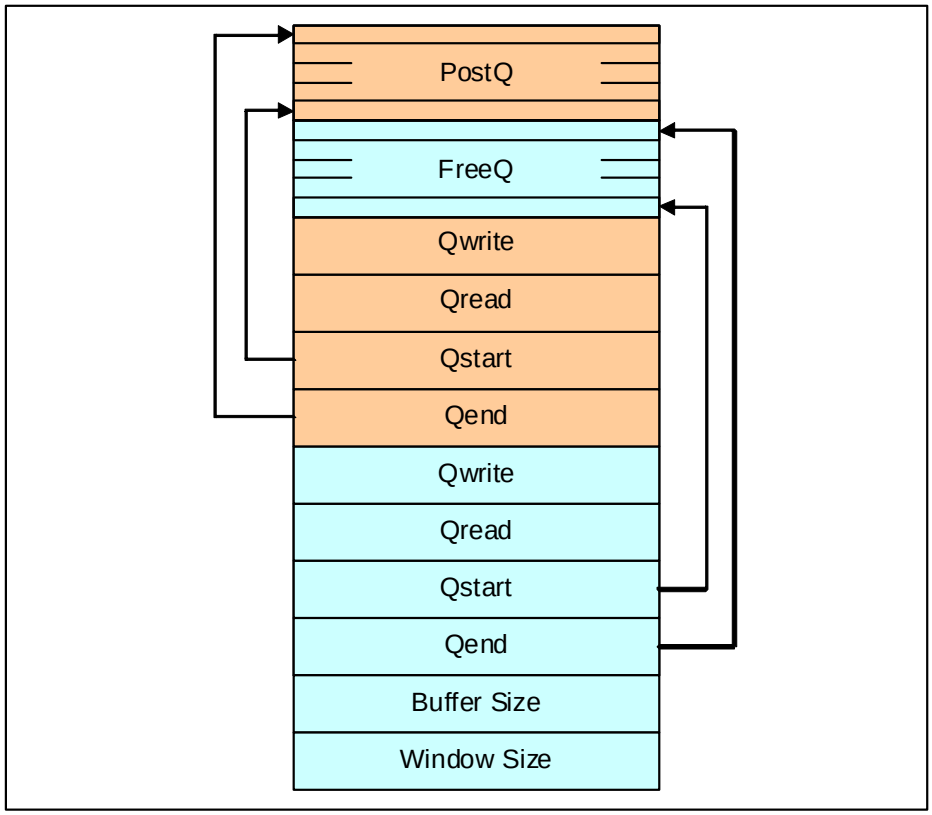

Figure 14 RP and EP Inbound Queue Structure

The beginning of the Inbound Translation Address window is used for the Inbound Queue Structures. A total of 16 Inbound Queue Structures are supported. In the current design, Peer#0 Inbound Queue Structure is used by the RP to send data to the EP's. All other remote EPs use the queue structure associated with their peer index which is determined by the RP and assigned to a newly discovered peer.

The remaining memory is used for Message Frames and is divided equally between the 16 possible peers. Figure 15 illustrates the usage of the EP Inbound Translation Address Window.

Similar to the EP, the RP allocates a block of memory to hold the Inbound Queue Structure and Message Frames for each EP in the system.

Notes

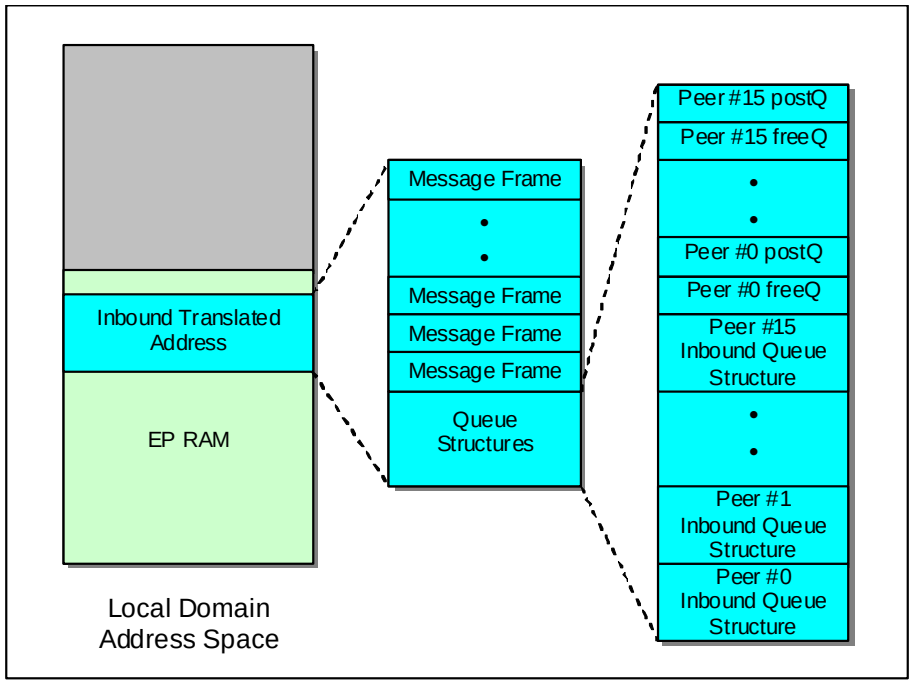

Figure 15 EP Inbound Translated Address Usage

When transmitting data, the local peer must first obtain the address of a valid Message Frame from the destination peer. This is done by popping the next entry from the destination peers FreeQ[1]. The local peer then populates this Message Frame with the data to be transmitted before pushing its address onto the PostQ[2] of the destination peer and signaling to the destination peer that its PostQ needs processing.

Upon receiving a signal to process its PostQ, a peer will pop Message Frame addresses from its PostQ[3] and process them. After each Message Frame is processed, its address is pushed on to the FreeQ[4] at which time it is free to be used again. See Figure 16.

Notes

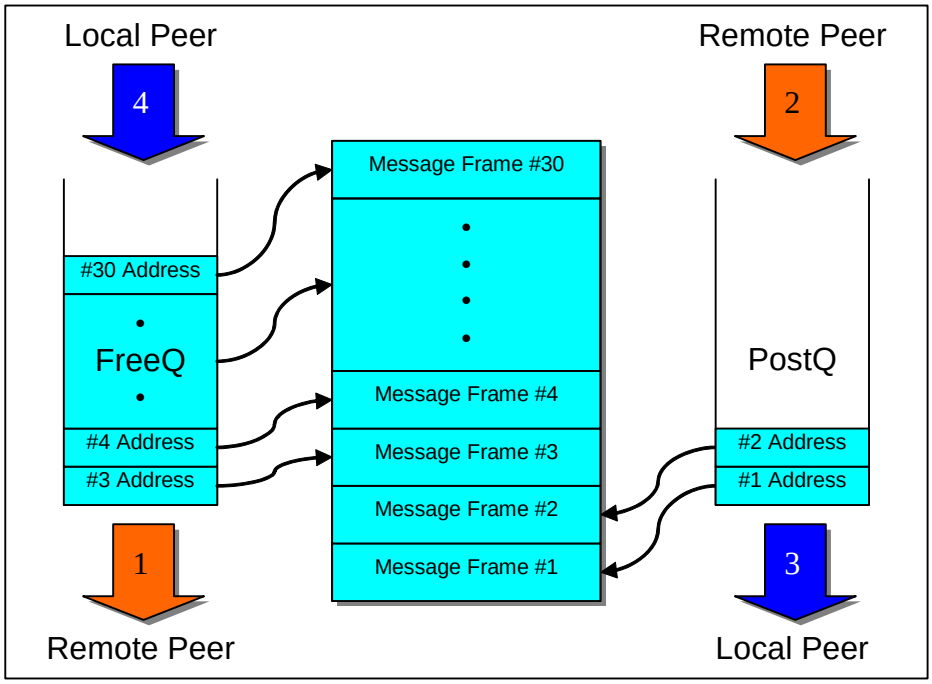

Figure 16 EP Inbound Queue and Message Frame Layout

Notes

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 existing EPs in the system will be informed of its Inbound Queue Structure base address, 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. The procedure used to transfer data from a local EP to a remote EP is described below:

- As mentioned in the Data Transport section, the local EP needs to acquire a Message Frame address from the remote EP's FreeQ. The Message Frame address needs to be converted from a System Domain physical address to a Local Domain virtual address before data can be written to it. If there are no free Message Frames available, a timer will be started to initiate a retry at a later time.
- After populating the Message Frame, the address of the Message Frame is placed in the remote EP's PostQ. The local EP then triggers an interrupt on the remote EP by setting the bit in the remote EP's *Message Shared Interrupt Register 0* that corresponds to the local EP's index.
- When the remote EP receives the interrupt, it checks the PostQ to see if there are Message Frames waiting to be processed. If so, each Message Frame address is converted from a System Domain physical address to a Local Domain virtual address before it is processed.
- The remote EP allocates a new buffer and copies the data from the Message Frame to the buffer. The data buffer is then posted to the appropriate application for further processing.
- The peer index of source EP is determined and the Message Frame is inserted into the FreeQ for the corresponding EP.

From a Local EP to a Remote RP

During system initialization, the RP's Transport Service allocates data space for each EP in the system. This data space holds the Inbound Queue Structure and Message Frames required by each EP in order for them to transfer data to the RP. Each EP is informed of its Inbound Queue Structure address on the RP so that an outbound address translation window can be configured, allowing the EP to access its Inbound Queue Structure on the RP.

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

- The EP writes the value from the PCIe MSI Message Data register to the address specified by the PCIe MSI Message Address register in order to generate an interrupt on the RP.
- When the RP receives an entry in its inbound PostQ, it converts the buffer address from System Domain physical to System Domain virtual before accessing it.
- Before releasing a Message Frame to its FreeQ, the RP converts the buffer address from System Domain virtual to System Domain physical.

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 except:

- When the RP retrieves an entry in a remote EP's inbound FreeQ, it converts the buffer address from System Domain physical to System Domain virtual before accessing it.
- When the RP posts a Message Frame to a remote EP's inbound PostQ, it converts the buffer address from System Domain virtual to System Domain physical.

Notes

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 the RP and EPs, respectively.

The modules for an x86 RP are:

- idt-mp-i386rp-msg.ko: Message Frame module
- idt-mp-i386rp-arch.ko: Local Architecture module
- idt-mp-i386rp-mpc8548Eep.ko: Transport module
- idt-mp-i386rp-eth.ko: Virtual Ethernet module
- idt-mp-i386rp-raw.ko: Raw Data Transfer module

The modules for an MPC8548E RP are:

- idt-mp-mpc8548Erp-msg.ko: Message Frame module
- idt-mp-mpc8548Erp-arch.ko: Local Architecture module
- idt-mp-mpc8548Erp-mpc8548Eep.ko: Transport module
- idt-mp-mpc8548Erp-eth.ko: Virtual Ethernet module
- idt-mp-mpc8548Erp-raw.ko: Raw Data Transfer module

The modules for EPs are:

- idt-mp-mpc8548Eep-msg.ko: Message Frame module
- idt-mp-mpc8548Eep-arch.ko: Local Architecture module
- idt-mp-mpc8548Eep-rp.ko: Transport module
- idt-mp-mpc8548Eep-mpc8548Eep.ko: Transport module to EPs
- idt-mp-mpc8548Eep-eth.ko: Virtual Ethernet module
- idt-mp-mpc8548Eep-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 in order to maintain 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. The complete layered picture is illustrated in Figure 17.

Notes

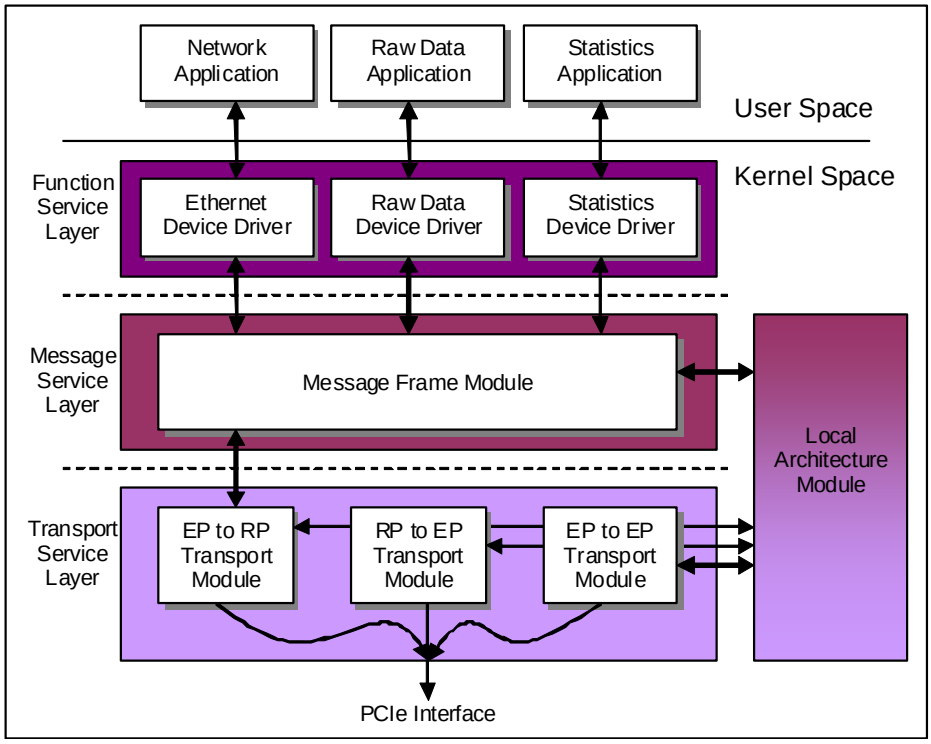

Figure 17 Software Modules and Device Drivers

The MPC8548E Transport Module exists in both the RP and EPs, whereas the EP to RP Transport Module only exists in the EPs.

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 Ethernet module simulates a virtual Ethernet interface (mp0). The module initialization function registers the virtual Ethernet interface 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 an mp_frame data structure, sets up the virtual Ethernet function header, sets up any data fragments, looks up the destination MAC address for destination peer ID, and 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 into the sk_buff. When the data transfer is complete, its callback function updates the MAC address to destination peer ID Table with the source MAC address and the source peer ID, then passes the sk_buff to the Linux kernel and releases the mp_frame data structure.

Notes

Raw Data Service Module

The Raw Data Transfer module is used to transfer received data to a user specified remote peer. The user interface to the Raw Data Transfer module utilizes the Linux 2.6 sysfs feature which allows information to be passed to the driver via the Linux file system. The module initialization function sets up data buffers, registers itself with the multi-peer Message module, and sets up the subsystem attributes in the sysfs used to interface with the user.

In order to begin a data transfer using the Raw Data Transfer module, the ID of the destination peer should be written into /sys/mp/forward followed by optionally writing the data to be transferred into /sys/mp/buffer and the number of times the data should be transferred into /sys/mp/count before writing the number of bytes to be transferred (in hexidecimal) into /sys/mp/send to begin the transfer. When the destination peer receives the raw data, it automatically transfers the data to the peer identified in /sys/mp/forward. If a peer ID has not been specified in /sys/mp/forward, the receiving peer terminates the transfer.

When /sys/mp/send is written to, the corresponding store function allocates an mp_frame data structure, sets up the mp_frame data structure with the specified length, clones the frame the number of times specified /sys/mp/count (minus one), then passes the frames down to message service for transmission.

The destination peers receive function allocates a buffer, then calls mp_frame_sync to transfer the data to the new buffer. The callback function for the data transfer allocates a new mp_frame data structure, sets up the mp_frame data structure, then passes the frame down to the message service for transmission.

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 to 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 and DMA transfer routines. Each different type of RP and EP has its own architecture-specific module. The address space conversion routines convert addresses between virtual, physical, bus, and PCI addresses. The DMA transfer functions are used to perform memory to memory and memory to PCIe transfers.

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

Notes

RP to EP Transport Module

The RP to EP 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, 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 EP to RP Transport module running on EPs to setup the memory windows and data frame buffers.

The transmit function is called by the message service to transmit a data frame to an EP. The receive function is triggered by the interrupt handler when an EP sends a data frame to the RP.

EP to RP Transport Module

The EP to 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 RP to 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 a data frame to the RP. The receive function is triggered by the interrupt handler when the RP or an EP sends a data frame to the local EP.

EP to EP Transport Module

The module initialization function of the EP to EP Transport Module running on EPs initializes the transport data structure and registers itself with the message service.

When the message service on an EP receives a peer-add notification, it calls the peer_add function of the EP to EP Transport Module. 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 another EP. Note that there is no receive function in the EP to EP Transport Module running on the EPs. All data reception is handled by the EP's EP to RP Transport Module.

Notes

System Initialization

In order for the RP to successfully detect the EPs connected to the system when the PCIe bus is probed, all EPs should be powered on before the RP. Once the RP and all of the EPs have successfully booted, the IDT System Interconnect modules may be loaded with the use of the Linux *insmod* (install module) command.

Due to the interdependences between the modules that comprise the IDT System Interconnect software the modules for an x86 RP should be installed in the order listed below.

- idt-mp-i386rp-msg.ko
- idt-mp-i386rp-arch.ko
- idt-mp-i386rp-mpc8548Eep.ko
- idt-mp-i386rp-eth.ko
- idt-mp-i386rp-raw.ko

Similarly the RP modules for an MPC8548E RP should be loaded in the following order.

- idt-mp-mpc8548Erp-msg.ko
- idt-mp-mpc8548Erp-arch.ko
- idt-mp-mpc8548Erp-mpc8548Eep.ko
- idt-mp-mpc8548Erp-eth.ko
- idt-mp-mpc8548Erp-raw.ko

and the MPC8548E EP modules should be loaded in the following order.

- idt-mp-mpc8548Eep-msg.ko
- idt-mp-mpc8548Eep-arch.ko
- idt-mp-mpc8548Eep-rp.ko
- idt-mp-mpc8548Eep-eth.ko
- idt-mp-mpc8548Eep-mpc8548Eep.ko
- idt-mp-mpc8548Eep-raw.ko

EP Initialization

When the EP to RP Transport Service module is installed, the following initialization is performed by the EP before communicating with the RP.

- Memory is allocated for the EP data structure.
- Memory is allocated for the RP data structure.
- The EP device data structure is initialized, including peer specific data, timers, and spin locks, etc.
- Memory is allocated for the EP inbound queue structure and message frames.
- IWBAR1 is setup to point to the inbound queue structure.
- System Domain translation address of IWBAR1 is programmed from the value written to BAR1 by the RP.
- Initialize the inbound queue structure.
- Allocate an interrupt for message register 0.
- Allocate and enable an interrupt for the Message Shared Interrupt Register 0.
- Enable interrupts for message register 0.

Notes

RP Initialization

When the Linux kernel detects an EP, the PCI probe sub-routine in the Transport Service module is invoked. The following tasks are performed before communicating with the EP:

- Peer data structure is allocated for the EP.
- An ID is created based on the Bus#, Device#, and Function# of the peer.
- The EP device data structure is initialized, including peer specific data, timers, and spin locks, etc.
- The EP device is enabled via the Linux PCI API.
- MSI support is enabled for the EP via the Linux PCI API.
- An interrupt is allocated for the EP, one per EP.
- Memory is allocated and initialized for the inbound queue structure, one per EP.

RP to EP Initialization

When the EP and RP transport services have completed their individual initialization, the RP can begin its initialization of the newly discovered EP. Figure 18 illustrates the initialization sequence between the RP and the EP.

During the initialization phase, before the transport services are fully initialized, information is passed between the RP and EPs using Message Registers 0-2. The RP transfers data to the EP by first writing the data to Message Register 1 and then writing the data type to Message Register 0. When Message Register 0 is written by the RP, the EP is interrupted. The EP reads the data type written to Message Register 0 to determine what action should be taken on the data written to Message Register 1. Similarly, when the EP transfers data to the RP, it writes the data to Message Register 2 and then interrupts the RP by writing the MSI data to the MSI address specified by the RP.

Notes

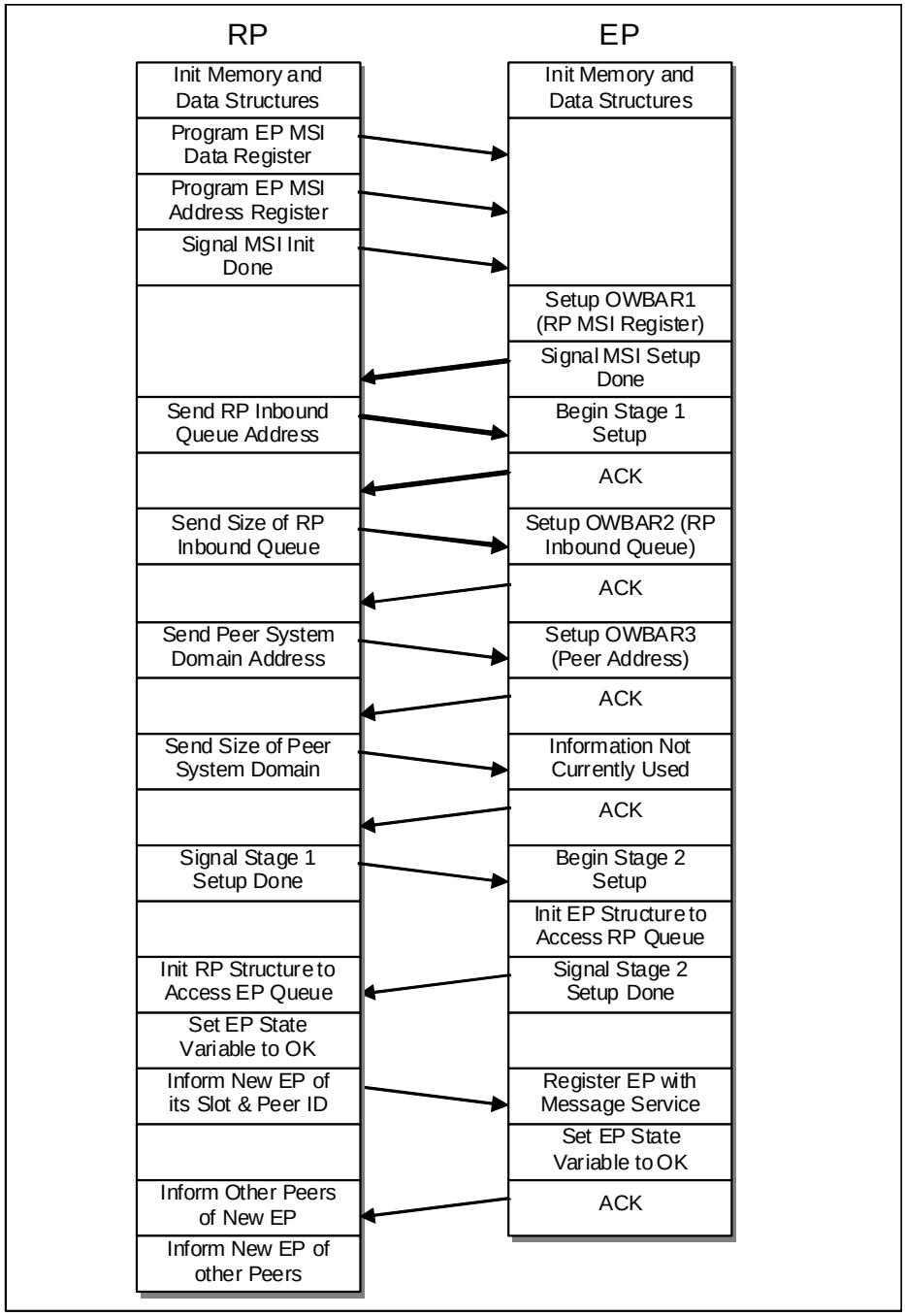

Figure 18 RP to EP Initialization Sequence

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 an x86 or an MPC8548E CPU as the Root Processor and an MPC8548E CPU 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.

Notes

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 the 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 an x86 or MPC8548E based Root Processor and MPC8548E 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 PostQ 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 Processor 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 EB24N3 Evaluation Board Manual

[3] IDT PES24N3 Product Brief

[4] MPC8548E PowerQUICC III™ Integrated Host Processor Family Reference Manual (MPC8548ERM)

[5] EP8548A User Manual (DES0212)

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.