Using the IDT PCI Express Standard System-Interconnect Solution to Deliver Predictable, Secure, High-RAS Communications in Bladed Systems

The bladed system communications challenge

Bladed system architectures enable equipment providers to produce high-density, modular equipment that can be easily scaled to match user needs. The inherent flexibility and scalability of a bladed architecture supports the expansion or reconfiguration of existing server, storage and telecommunications infrastructure, and supports the rapidly growing and shifting services demand.

This high level of scalability is achieved by disaggregating computing and scarce system I/O resources. However, it poses a significant challenge: the system needs an equally scalable interconnect architecture that enables any and all root complexes to communicate with each other and with any and all endpoints.

The IDT system-interconnect solution, the industry's first PCI Express® (PCIe®) system-interconnect solution, fulfills this demanding requirement. Developed in close collaboration with leading customer design teams, the solution is optimized specifically for demanding bladed system designs.

The four minimum operational requirements of a system-interconnect architecture are that it must:

- Be easily expandable to support extensive system scalability
- Deliver predictable, wire-speed performance under all load conditions, regardless of system form factor and capacity
- Ensure high reliability, availability and serviceability (RAS) and security
- Consume low power

However, system complexity mandates a level of functionality that simplistic peer-to-peer switching cannot deliver. The complete solution must:

- Support unconstrained inter-processor communications
- Support maximally configurable system architecture
- Enable the flexible virtualization and sharing of peripherals and scarce system I/O
- Dynamically map I/O and peripherals to computer resources
- Facilitate interdomain communications in systems that deploy heterogeneous processors
- Reuse existing hardware, software and firmware wherever possible

A logic diagram of how the IDT system-interconnect solution connects with various system resources is shown in figure 1.

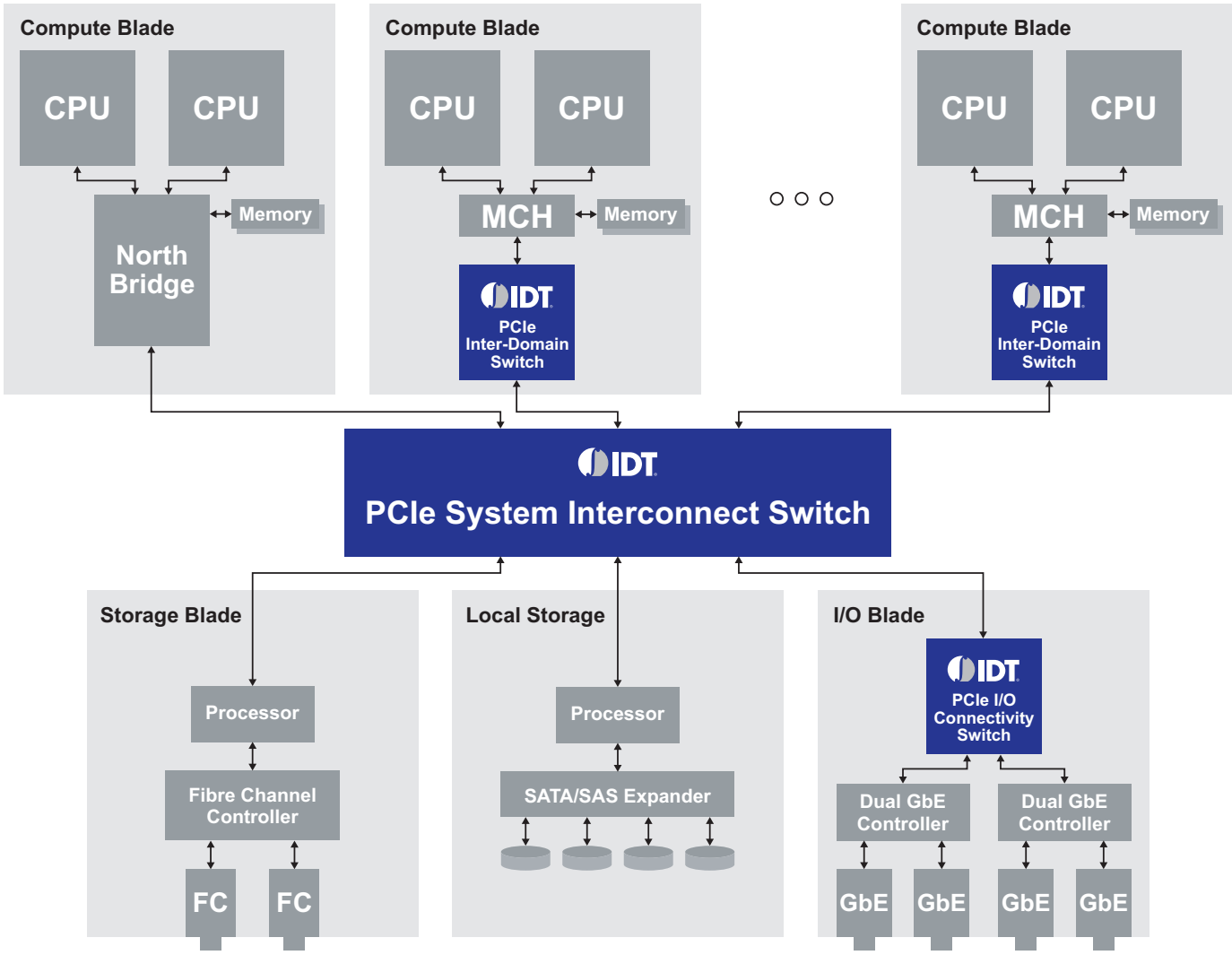


Figure 1. IDT system interconnect solution

The IDT system interconnect solution:

- Executes multiple, simultaneous wire-speed transactions with 32 GBps switching capacity, delivering scalability and predictable performance
- Enables flexible system scaling and optimal resource utilization through:
  - Multiroot configurations with inter-processor communications
  - I/O sharing and virtualization
  - Flexible I/O mapping
- Ensures high RAS and security through:
  - Redundant host support
  - End-point security and isolation
  - Advanced diagnostics
- Eases migration to bladed systems by leveraging existing hardware, firmware and software

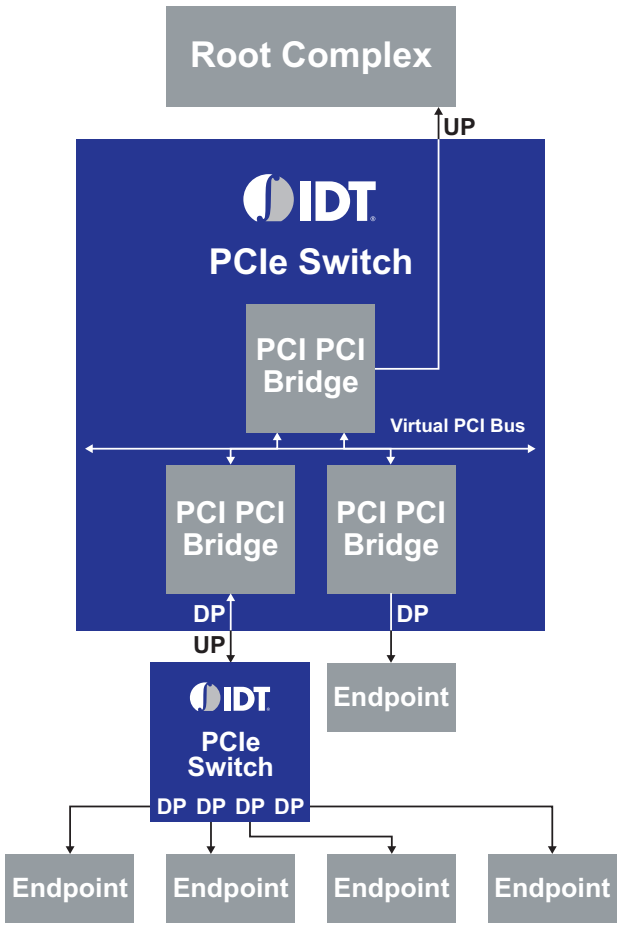
The balance of this white paper details the architecture that enables the system-interconnect solution to fulfill the requirements of unconstrained inter-processor communications.

Introduction to the PCIe communications topology

Three different types of devices exist in a native PCIe topology: root complexes, PCIe switches and endpoints. As originally conceived, only one root complex is in a PCIe tree. A root complex can be visualized as a single-processor subsystem with a single PCIe port, even though it may really consist of one or more CPUs, together with their associated RAM and memory controllers, as well as other interconnect and/or bridging functions (see figure 2).

PCIe routes use memory address or ID, depending on the transaction type. Thus, every register and device (or function within a device) in the PCIe tree must be uniquely identified. Such identification requires a process called enumeration.

During system initialization, the root complex performs the enumeration process to identify the buses in the system and the devices that reside on each bus, as well as the required address space for the devices' registers and memory. The root complex allocates numbers to all PCIe buses and configures the bus numbers to be used by the PCIe switches. A PCIe switch behaves as if it were multiple PCIe-PCIe bridges (see inset in figure 2). The root complex allocates and configures the memory and I/O address space for each PCIe switch and endpoint device.



UP: Upstream port
DP: Downstream port

Figure 2. PCIe system topology

A multipeer system has more than one processor subsystem in the PCIe tree. For example, a second root complex may be added to the system via the downstream port of a PCIe switch to act as a warm standby to the primary root complex. However, an issue arises when the second root complex also attempts the enumeration process. Assuming that as the link uses crosslink training, it transmits configuration read messages to identify other PCIe devices in the system. Configuration transactions can move only downstream. A PCIe switch does not forward or respond to configuration messages that are received on its downstream port; it silently drops all such configuration messages. Thus, the second root complex is isolated from the rest of the PCIe tree and cannot detect PCIe devices in the system. So, simply adding processors to a downstream port of a PCIe switch does not provide a multipeer processing solution. In other words, this approach does not support unconstrained inter-processor communications. Earlier attempts to solve this problem, such as nontransparent bridging (NTB), are proprietary. Clearly, a standard solution is required.

Unconstrained inter-processor communications

A multipeer system topology is shown in figure 3. Only a single root complex (RC) processor is in the topology. The RC processor is attached to the single upstream port (UP) of the PCIe switch. The RC processor is responsible for the system initialization and enumeration process as in any other PCIe system. A 16-port PCIe switch is used as an example in this paper, allowing up to 15 endpoint (EP) processors to be connected in this system. Any of these downstream ports may address a card containing only simple endpoint devices; that is, nonprocessor endpoints, such as an Ethernet network interface card (NIC) or Fibre Channel host bus adapter (HBA).

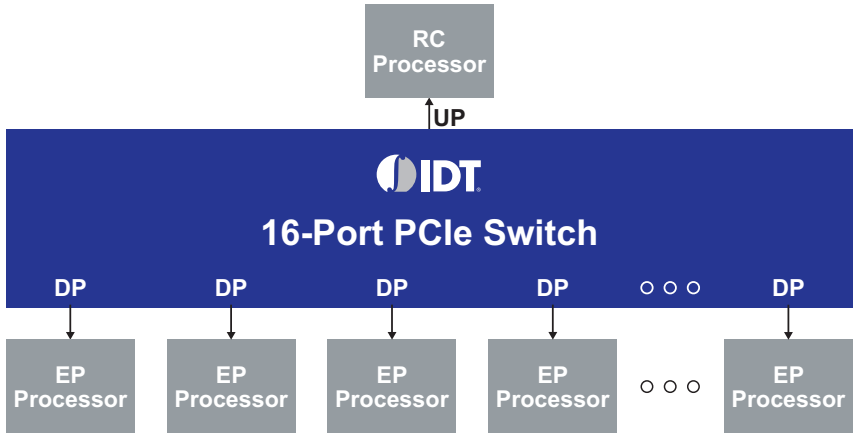


Figure 3. Multipeer PCIe system topology

An EP processor is a processor with one of its PCIe interfaces configured as a PCIe endpoint. Two examples of an EP processor are the AMCC PowerPC® 440Spe and the Freescale MPC8641. An EP processor may also be an intelligent I/O adapter that processes I/O transactions, such as redundant array of responsive disks (RAID) controllers for storage. An example of an intelligent I/O is the Intel® IOP333. figure 4 illustrates the PCIe address domains in such a system.

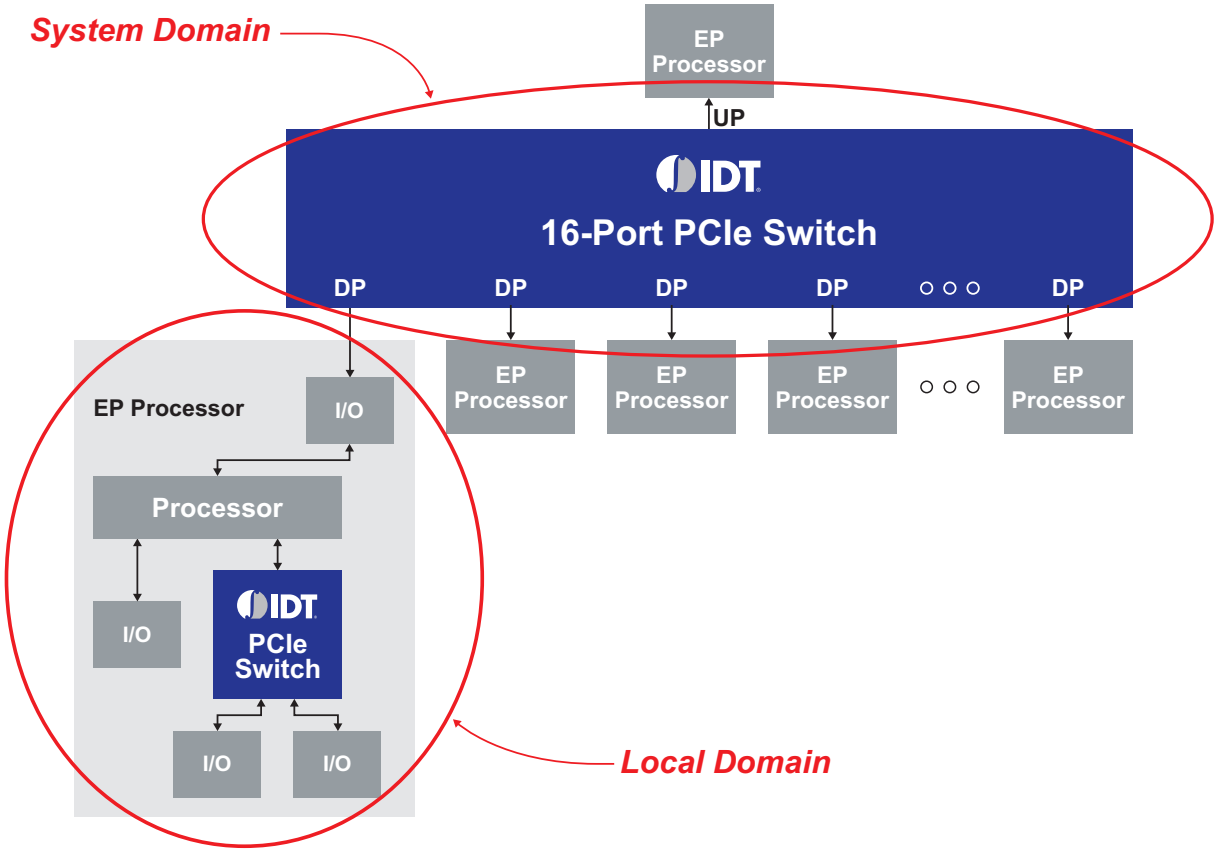


Figure 4. PCIe address domains

While acting as an endpoint in the system domain, an EP processor may have local PCIe devices on its other port or ports. The EP processor is the root complex processor for the local PCIe devices. A multipeer system has multiple PCIe domains. The system domain is owned by its RC processor and includes the RC processor, the 16-port PCIe switch and the PCIe endpoints on each of the EP processors in the system domain. The RC processor is responsible for the allocation of the PCIe bus number and address space in the system domain. The RC processor stops its device identification process when it detects the endpoint on the EP processor. The RC processor treats the EP processor as a PCIe endpoint device during its enumeration process.

The EP processor acts as the RC processor for its local PCIe devices; it is responsible for the allocation of PCIe bus numbers and address space in the local domain. The PCIe endpoint device that connects to the system domain is typically a local device integrated into the EP processor. The EP processor does not detect any device in the system domain during its enumeration process. The PCIe endpoint device on the EP processor isolates the local domain from the system domain, presenting one endpoint configuration space to the system domain and another endpoint configuration space to the EP processor in the local domain.

PCIe implementation considerations

Several considerations are involved in supporting unconstrained inter-processor communication with high RAS and security:

- Memory map management
- Enumeration and initialization
- Inter-processor communication mechanisms
- Interrupt and error reporting
- Redundancy

Memory map management

Address routing is used to transfer data to or from memory, memory mapped I/O or I/O locations. The RC processor is responsible for the allocation of memory addresses for the system domain. The EP processor is responsible for the allocation of memory addresses for its local domain. To avoid any memory-address conflicts, a general implementation uses address translation between the system and local domain. In the system domain, the RC processor reserves a block of memory address space for all the EP processors. This reserved block of memory is divided equally, for example, into 16 smaller blocks of address ranges. Each smaller block of addresses is reserved for an EP processor that connects to one of the ports of the 16-port PCIe switch. The actual physical memory of each smaller block exists in an EP processor. An example of the address map is shown in figure 5. This example assumes that the EP processors are implemented on processing blades, one slot per PCIe port, and that 1 megabytes (MB) of address space is reserved per slot. A total of 16 MB of address space is reserved, and the starting address is assumed to be 0x80000000. Whenever the RC processor detects an EP processor in a particular slot (or port of the PCIe switch) during system initialization, the RC processor assigns the address space to the EP processor based on the slot number the EP processor is in. The address space allocation and assignment is fixed based on the slot number and does not change when an EP processor blade is added or removed from a slot. The starting address, the size of the address block that is reserved per slot and the number of PCIe ports in the system should be adjusted based on system architecture. I/O transactions are not used for multipeer communication, and hence, I/O address translation is not required.

The EP processor is responsible for the address map of its local domain; it allocates and assigns memory address ranges to its local devices. This is independent of the memory map of the system domain. (Only the address ranges should correlate, although the system range may be larger than the range required by the EP processor.) An address translation unit (ATU) is required on the EP processor to translate address space between the system domain and the local domain. The EP processor supports inbound and outbound address translation. Transactions initiated in the system domain and targeted at the EP processor's local domain are referred to as inbound transactions. Transactions initiated on the EP processor's local domain and targeted at the system domain are referred to as outbound transactions.

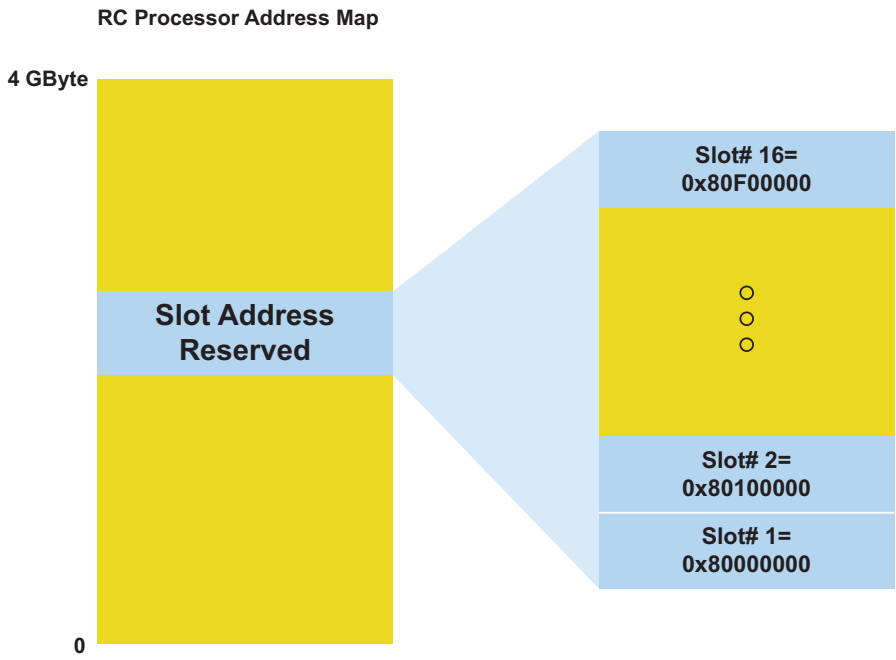


Figure 5. System domain address map

An example of the address translation is shown in figure 6. This example assumes the EP processor is in slot 2. Two outbound address translation windows are created. One outbound address window is created to access the system domain address for slot 1. The second outbound address window is created to access the system domain address for slots 3 to 16. Since this EP processor is in slot 2, there is no outbound window to address itself. Whenever the EP processor accesses any local address space that falls within one of the two outbound address windows, the ATU forwards the request to the system domain. The address is also translated from the local domain address space to the system domain address space. A single outbound address window may be used instead of creating two outbound address windows. An outbound address window is created for the entire system domain address that is reserved from slot 1 to slot 16. The software than runs on the EP processor makes sure that there is no local memory access to the address range that will be translated to its own slot (slot 2 in this example) in the system domain address map.

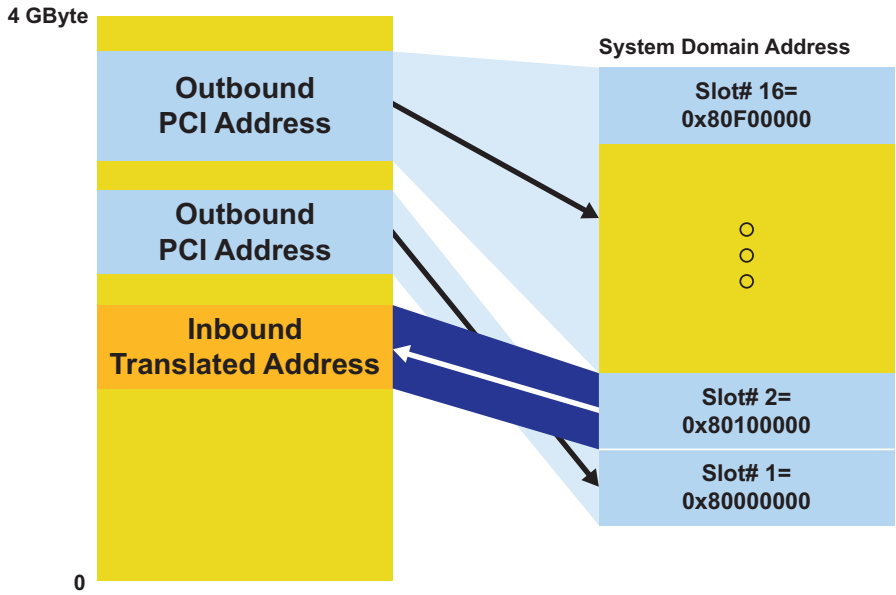


Figure 6. Address translation unit

A single inbound address translation window is set up to allow initiators in the system domain to directly access the local domain within the address range. The EP processor sets up the inbound and outbound ATUs to translate addresses between its local domain and the system domain, as shown in figure 6. When the EP processor receives a memory request from the system domain, it receives the packet only if the address in the packet header is within the memory range assigned to the EP in the system domain. (The ATU may further check if the requested address is within a tighter address window as configured in the inbound ATU.) In this example, the address window is between 0x80100000 and 0x801FFFFF. If the requested address is within the address window, the request packet is forwarded from the system domain to the local domain. The address field in the requested packet is also translated to the inbound local address. The inbound local address may represent a local buffer in memory that the EP processor will read and respond to, or it may represent a local register that affects the EP processor directly.

When the EP processor accesses the ATU via its PCIe interface, then the requester ID and completer ID in the PCIe packet also have to be translated between the local domain and the system domain. If the processor accesses the ATU via a different bus, such as the local CPU bus, then there is no need to translate the requester and completer IDs.

Enumeration and initialization

During system initialization, the RC processor scans the system domain for PCIe devices. The RC processor assigns unique bus numbers to all the PCIe links and the virtual buses within PCIe switches. The process of discovering the PCIe tree topology—and the devices and functions that reside on this topology—is referred to as the enumeration process. The normal PCIe (or PCI) enumeration process reserves bus numbers and address space for empty slots. The normal enumeration process does not change for multipeer support. All PCIe switches and endpoints within the system domain are detected at the end of the enumeration process. All EP processors are detected as PCIe endpoints in the system domain, and devices locally attached to the EP processors are not detected in the system domain at all.

After the enumeration process has been completed, the RC processor assigns memory address space to each endpoint device in the system domain. The absolute memory address range (in the system domain) allocated to an EP processor is based on the slot that the EP processor is plugged into, as shown in figure 5. This example assumes that an EP processor requests no more than 1 MB of address space in its base address register (BAR). If an EP processor requires more than 1 MB of address space, then the address map has to be adjusted accordingly. The corresponding address space is assigned to an EP processor by writing to its BAR.

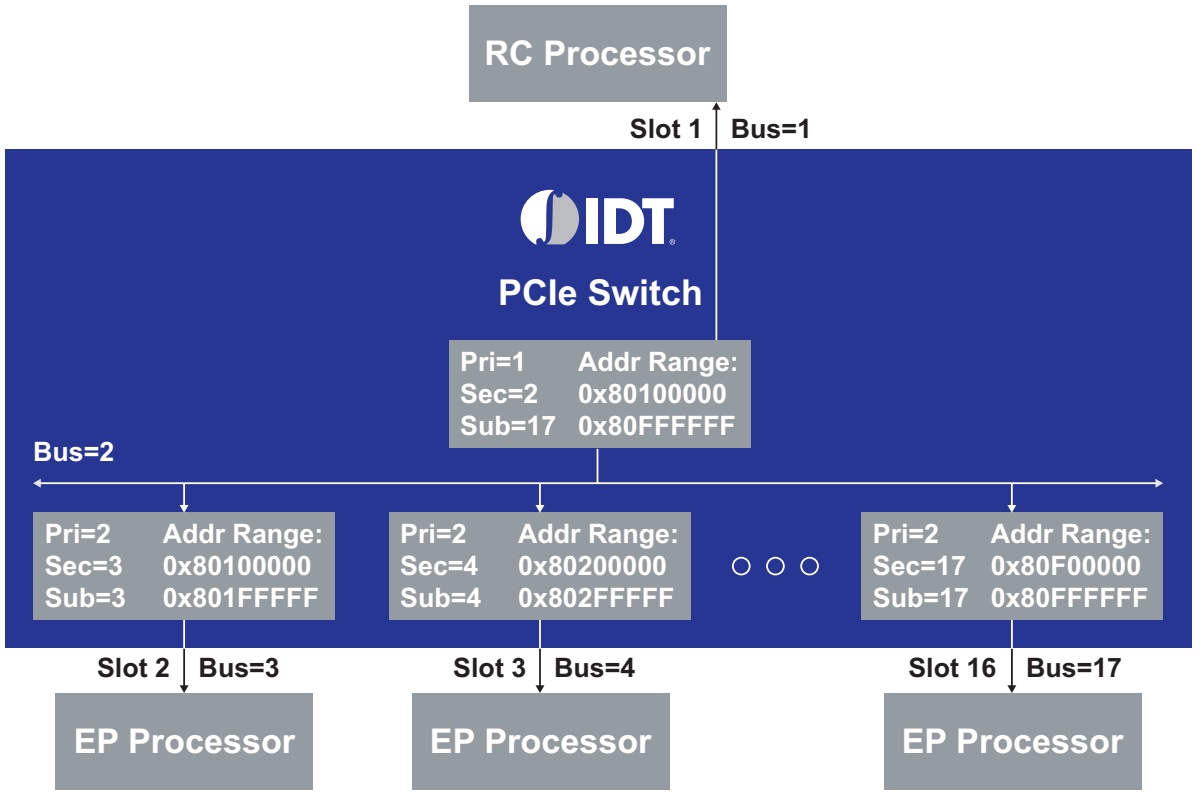


Figure 7. PCIe switch configuration after enumeration and initialization

There are multiple virtual PCIe-PCIe bridges in a PCIe switch. The configuration of the PCIe-PCIe bridges after enumeration and initialization is shown in figure 7. The primary bus number (Pri), secondary bus number (Sec), subordinate bus number (Sub) and address range for a particular slot (or port) are shown in this example. After the enumeration and initialization process is completed, memory requests can be forwarded to the targeted EP processor using address routing in the PCIe switch.

Each EP processor also runs its own enumeration and initialization process on its local domain. Because of the isolation provided by the endpoint function within the EP processor, the local and system domain enumeration and initialization processes are independent and may proceed in parallel. The EP processor may treat the PCIe interface to the system domain as a PCIe endpoint (that is, just another local device), depending on the architecture of the EP processor.

The EP processor relies on the RC processor to detect the existence of other EP processors in the system domain. The RC processor has a complete topology map of the system domain. It notifies an EP processor of the existence of other EP processors.

The RC processor is also responsible for hot-plug events in the system domain. A bus number and a memory address range are reserved for each slot in the PCIe switch. When an EP processor is plugged into the system, the pre-assigned bus number is used to access this EP processor. The BAR of the newly inserted EP processor is configured to the pre-assigned memory address offset. As shown in figure 7, bus number 4 is used, and memory address 0x80200000 is programmed into the BAR of an EP processor when the EP processor is plugged into slot 3. When the initialization of the EP processor is complete, the RC processor notifies all other EP processors that a new EP processor has been plugged into slot 3.

Conversely, when the EP processor is unplugged from slot 3, the RC processor notifies all other EP processors that the EP processor in slot 3 has been removed so that the resources associated with the removed EP processor can be released and cleaned up. However, the bus number and memory address are not reassigned since they are reserved for slot 3.

Inter-processor communication mechanisms

An inter-processor communication protocol has to be defined to enable peer-to-peer data transfer. The protocol has to define the handshake between the peers (and the message format).

There are three basic categories of communication mechanisms:

- Low volume of data traffic between an EP processor and the RC processor
- Low volume of data traffic between the EP processors
- High volume of data traffic between the EP processors (including the RC processor)

For low-volume data traffic between an EP processor and an RC processor, scratchpad registers, such as those implemented by a typical EP processor, may be used. The EP processor and the RC processor may read from and write to the scratchpad registers. The EP processor usually implements multiple (for example, eight to 16) scratchpad registers. Doorbell registers are used to send interrupts to the EP processor and the RC processor. When the RC processor needs to send some data to an EP processor, the RC processor writes the data to the scratchpad registers. It then writes to the doorbell register, interrupting the EP processor to indicate that a message in the scratchpad is ready to be consumed by the EP processor. The same procedure applies when an EP processor sends data to the RC processor.

For low-volume data traffic between two EP processors, the scratchpad and doorbell registers can be used. A semaphore register with a lock/unlock mechanism is required to control the usage of the scratchpad registers, since they are shared by all the EP processors. Before an EP processor can use the scratchpad, it has to acquire ownership of the scratchpad using the semaphore register. Once the semaphore is acquired, the EP processor can start writing to the scratchpad. Once the data is consumed, the sender releases the semaphore. An alternative is for the RC processor to relay the data between the EP processors.

Advanced queuing is needed for high-volume data traffic between the EP processors. A possible queuing structure is to follow the intelligent I/O architecture,[1] which offers two paths for data messages: an inbound queue structure and an outbound queue structure. The inbound queue structure receives data messages from any EP or RC processor in the system domain. The outbound queue structure sends data messages to the RC processor only. The assumption is that all EP processors implement the inbound and outbound queue structures, and the RC processor does not implement any inbound/outbound queue structure. When an EP processor sends data messages to another processor, the local EP processor uses the inbound queue structure of the remote EP processor. When an EP processor sends data messages to the RC processor, the local EP processor uses its local outbound queue structure. The queue structure contains a pair of first in, first out (FIFO) queues: FreeQ and PostQ. A read from the FIFO queue removes the first entry in the queue and a write to the FIFO queue adds an entry to the end of the queue. The FreeQ contains free buffer locations into which data messages can be written. The PostQ contains the locations of data messages that have been written to it (and posted by the sender).

Hardware-based queue structure

If the FIFO queue is implemented in hardware, such as the Intel IOP332/3 I/O processor, then a single memory-read operation removes the first entry from the queue, and a single memory-write operation adds an entry to the queue. An EP processor needs to have just a single pair of inbound FreeQ and PostQ to receive data from multiple EP processors.

The inbound translated address is configured as shown in figure 8. The translated address is divided into three different regions: the queue structure, some other hardware dependent control structures and data buffers. The queue structure contains only a single address location to be accessed per queue. A memory read removes the first entry from a queue and a memory write moves an entry to the end of the queue. As an example, when the EP 1 processor needs to send a data message to the EP 2 processor, the EP 1 processor performs a memory-read operation from the inbound FreeQ on EP 2 to get a free buffer. After the EP 1 processor has transferred all the data to the buffer, it performs a memory-write operation to the inbound PostQ on EP 2 to notify it that a data message is ready for processing.

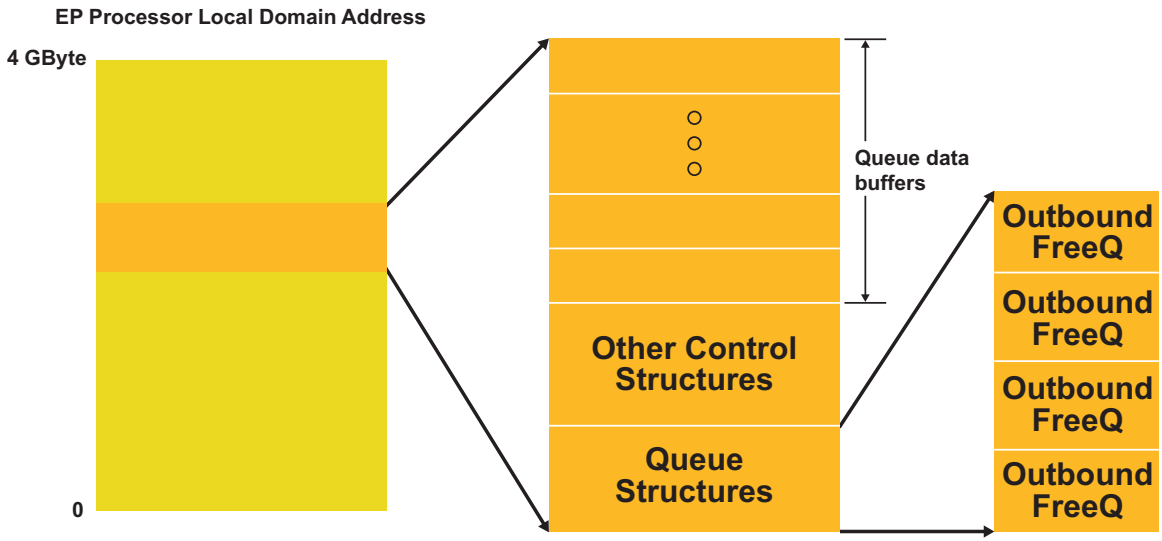


Figure 8. Hardware-based inbound translated address usage

Software-based queue structure

If hardware does not support the FIFO queue structure in the EP processor, the FIFO queue structure has to be implemented in software. Four addresses of the queue have to be maintained in the queue structure: Qstart, Qend, Qread and Qwrite. Qstart points to the beginning of the queue, and Qend points to the end of the queue. Qstart and Qend together specify the maximum number of entries in the queue. Qstart and Qend are static and do not change after initialization. An entry is removed from the location pointed to by Qread, and an entry is added to the location pointed to by Qwrite. Qread is updated to point to the next location in the FIFO queue after removing an entry, and Qwrite is updated to point to the next location after adding an entry. The FIFO queue is empty when Qread is the same as Qwrite. The FIFO queue is full when Qread is one entry ahead of Qwrite.

The operation to remove an entry from a FIFO queue requires multiple memory read/write operations:

- Read Qread
- Read Qwrite
- If Qread != Qwrite, move on to next step. Otherwise, FIFO is empty; stop here
- Read the location pointed to by Qread
- Adjust Qread to point to the next location
- Write the new value of Qread to the FIFO queue structure

The operation to add an entry to a FIFO queue requires similar operations:

- Read Qread
- Read Qwrite
- If Qread is one entry ahead of Qwrite, FIFO is full; stop here. Otherwise, move to the next step
- Write the new entry to the location pointed to by Qwrite
- Adjust Qwrite to point to the next location
- Write the new value of Qwrite to FIFO queue structure

Multiple read and write operations are required to add an entry to or remove an entry from a FIFO queue. Once an EP processor performs the first read operation on a FIFO queue, no other EP processor should access the FIFO queue until all the read and write operations are completed. Otherwise, the FIFO queue is corrupted. To guarantee that the queue add/remove operation does not get interrupted, a semaphore or multiple queues are needed. Assuming there is no semaphore support in the hardware, multiple FIFO queues are proposed here. A single pair of inbound FreeQ and PostQ has to be reserved per sender (EP or RC processor). In other words, a pair of inbound FreeQ and PostQ is dedicated to a single sender. For a 16-port (or slot) system, 16 pairs of inbound FreeQ and PostQ are required. Fifteen pairs of inbound FreeQ and PostQ should be sufficient, because an EP processor does not need to use the inbound queue structure to send data to itself. In this example, 16 pairs of inbound queue structures are used.

The inbound translated address is configured as shown in figure 9. The translated address is basically divided into three different regions: the queue structure, some other hardware dependent control structures and data buffers.

The queue structure contains queue pointers plus the queue itself to store entries. At initialization, queue pointers are configured to be empty. There are two queue structures, one for inbound and the other one for outbound, per EP processor (or per slot) in the system. There are 16 inbound and 16 outbound queue structures.

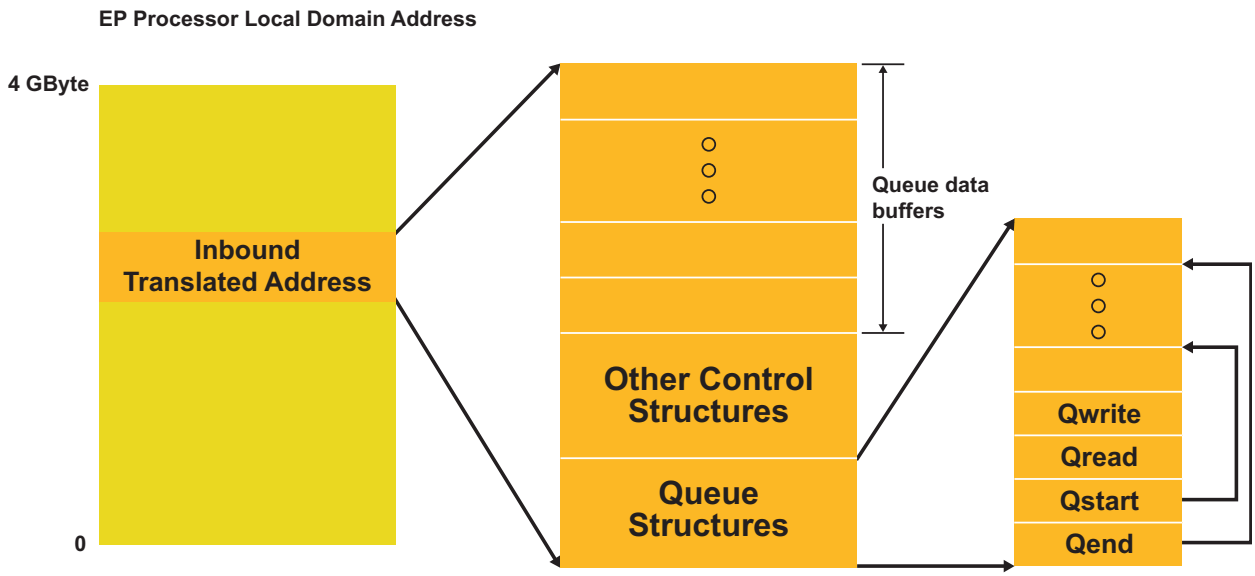


Figure 9. Software-based inbound translated address usage

Data buffer structure and data transfer protocol

A local inbound queue is used to receive data from a remote EP processor. The inbound queue and data buffer usage are shown in figure 9. In this example, the queue data space is divided into 128 equally sized buffers. The buffer size is system dependent, but 4 (kilobytes) KB is reasonable since 4 KB is a common page size. The local EP processor adds the data buffer addresses (pointers) into the FreeQ to make buffer space available for reuse. The buffer addresses may be added in any order to the FreeQ. When a remote EP processor needs to send data to this local EP processor, the remote EP processor gets a data buffer by removing the first entry from the FreeQ of the local EP processor. The data is then transferred to the data buffer referenced by that entry. Next, the data buffer address is written into the PostQ of the local EP processor. The local EP processor reads the data buffer address from its PostQ and processes the data.

The local EP processor may poll the PostQ periodically to check for any new entry in the PostQ. If doorbells are supported, the remote EP processor can assert a doorbell after adding an entry to the PostQ of the local EP processor to notify the local EP processor that a new entry has been added to its PostQ.

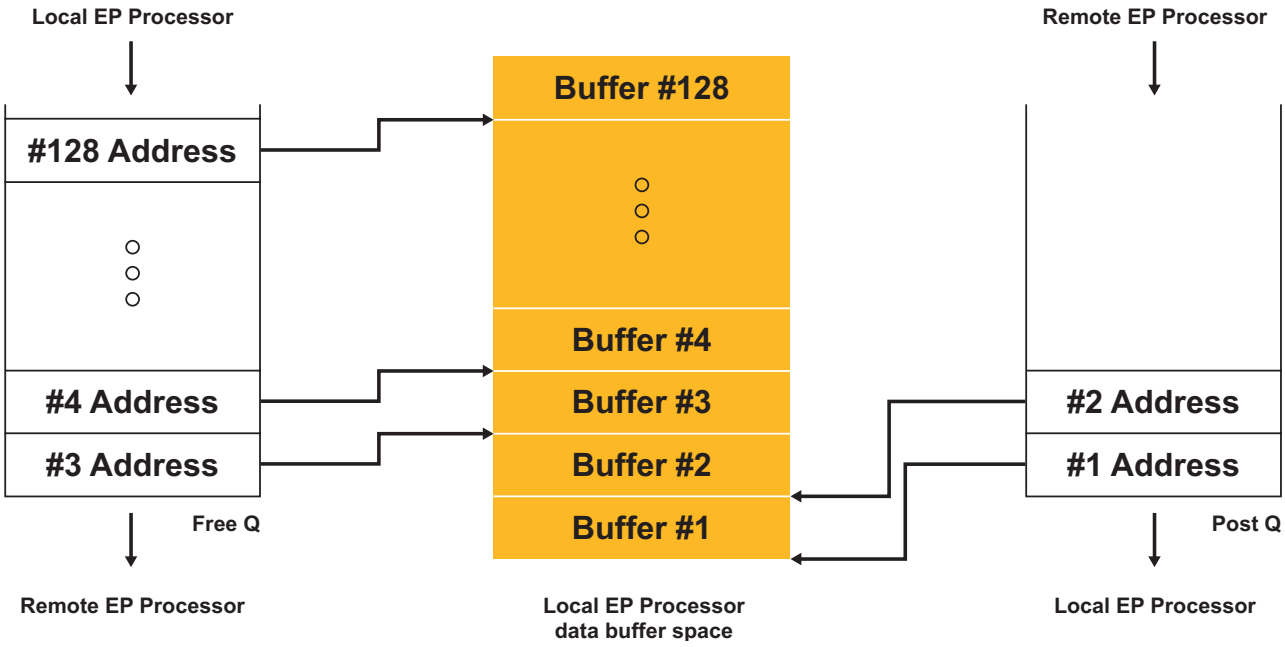


Figure 10. Inbound queue and data buffer usage

The data transfer protocol is summarized in figure 11. In this example, the EP processor in slot 3 (EPP 3) transfers a block of data to the EP processor in slot 2 (EPP 2). EPP 3 first gets a free buffer from the FreeQ on EPP 2. EPP 3 then writes data into the buffer. After all the data has been written, the buffer address is written into the PostQ of EPP 2. EPP 2 gets the buffer address from its PostQ. Then it reads and processes the data in the data buffer. Finally, the buffer is returned to the FreeQ.

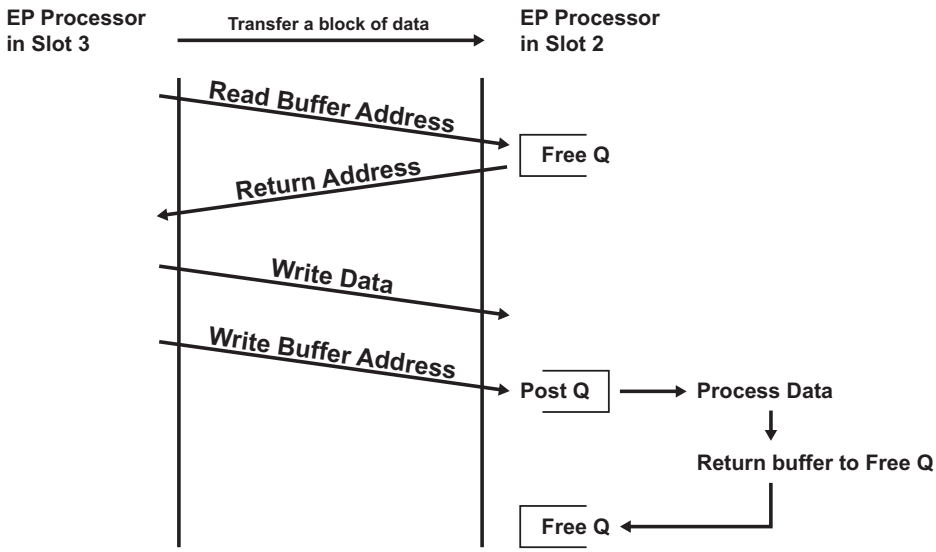


Figure 11. Data transfer protocol

Interrupt and error reporting

Interrupts in the system domain are handled in the normal PCIe way. Because the EP processor is a native PCIe device, a message signaled interrupt (MSI) is used to deliver an interrupt from it to the RC processor. A device-dependent method is needed to deliver interrupts to an EP processor. Typically, an EP processor supports a doorbell register that can be used to interrupt an EP processor.

Two methods are used to report errors: error messages or a completion status. Error messages are routed to the RC processor. A completion status is a field within the completion header that enables the transaction completer to report errors back to the requester.

An EP processor only makes memory read or write requests to another EP processor. When an error occurs in any EP-processor-to-EP-processor request, an error-status message is reported back to the requester via the completion status, enabling the EP processor to begin its error-handling procedure.

All other system-domain-related errors, such as data link layer and physical layer errors, are reported to the RC processor, which initiates the error handling procedure.

Redundancy

A system-interconnect application generally requires some level of redundancy. A dual RC processor topology has an active and a standby RC processor. An example of this dual RC processor topology is shown in figure 12. The IDT PES64H16 PCIe switch supports the redundant upstream port feature. The RC processor that is connected to the UP of the PCIe switch is the active RC processor, and the one that is connected to the redundant UP is the standby RC processor. During normal operation, the active RC processor is in control of the system and is the root complex of the system domain. The standby RC processor has no connection to the system domain. An out-of-band connection (not shown) between the RC processors supports a number of transactions. Heart beat and checkpoint messages are sent periodically from the active RC processor to the standby RC processor. The standby processor monitors the state of the active RC processor.

The standby RC processor takes over as the active RC processor when a managed switchover is requested or the standby RC processor detects a failure in the active RC processor. A managed switchover is initiated by the user for scheduled maintenance or software upgrades or in response to some form of demerit checking within the primary RC processor.

The following is the managed switchover procedure:

- The active RC 1 processor requests all EP processors to stop making requests to the system domain.
- The system waits for a short period of time for all outstanding requests to finish. The actual waiting time is system dependent.
- RC 1 configures the PES64H16 switch to swap its UP with the redundant UP. The redundant UP is now the active UP. The standby RC 2 processor becomes the active RC processor.

After the switchover, the RC 2 processor becomes the active RC processor. It initializes the PCIe switch to the same state as before the switchover, which it had previously learned via the out-of-band link with the RC 1 processor.

The RC 2 processor notifies all EP processors that the system is back to normal and that requests can now be made.

The following is the failure switchover procedure:

- The standby RC 2 processor monitors the state of the active RC 1 processor through heart beat and checkpoint messages.
- The IDT PES64H16 PCIe switch supports the watch dog timer initiated failover feature. When the active RC has not reset the watchdog timer for a certain period of time as configured during initialization, the PCIe switch failovers to its Redundant UP. When there is a failure in the active RC 1 processor, the watchdog timer will expire and triggers the failover. The standby RC 2 processor takes over as the active RC.

After the switchover, the RC 2 processor becomes the active RC processor. It initializes the PCIe switch to the same state as before the switchover, which it had previously learned via the out-of-band link with the RC 1 processor.

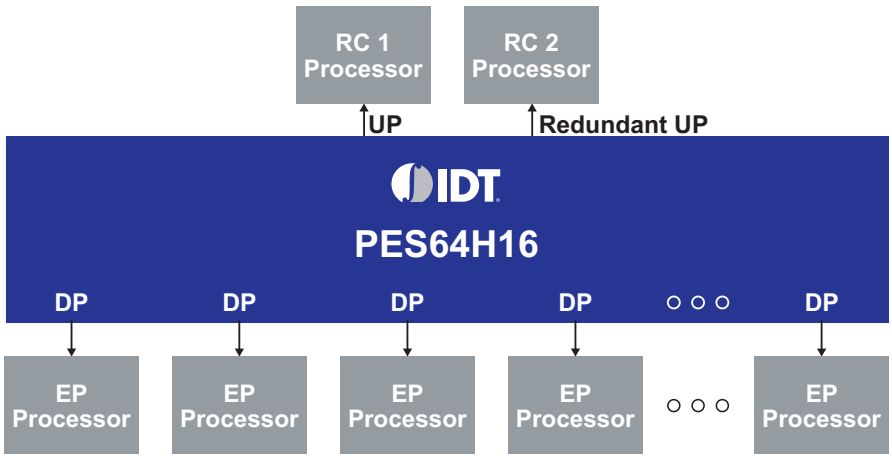


Figure 12. Dual RC processors topology

The switch is still a single point of failure in the dual RC processors topology. For a fully redundant system, a dual-star topology may be deployed as shown in figure 13. In this topology, an additional PCIe switch is added and the IDT PES24NT3 interdomain PCIe switch is added to the EP processor blade. The PCIe switch is added to the RC processor blade so that the RC processor blade has the RC processor and a 16-port PCIe switch. Each EP processor blade connects to two PCIe switches, but only one of the two connections is active. The IDT PES24NT3 interdomain switch supports nontransparent bridging on one of its port. In this topology, the port that has the standby connection is in non transparent mode.

During normal operation, all EP processors connect to the active RC 1 processor blade, and the RC 2 processor blade is in standby mode. It has the same configuration as the RC 1 processor for the system domain. During a switchover, there is no need to reinitialize the PCIe switch. The active RC processor is in control of the system and is the root complex of the system domain. There is an out-of-band connection (not shown) between the RC processors. Heart beat and checkpoint messages are sent periodically from the active RC processor to the standby RC processor. The standby RC processor monitors the state of the active RC processor.

The switchover procedure is similar for the dual RC processors and the dual-star topology.

The following is the managed switchover procedure:

- The active RC 1 processor requests all EP processors to stop making requests to the system domain.
- The system waits for a short period of time for all outstanding requests to finish. The actual waiting time is system dependent.
- RC 1 configures all the IDT PES24NT3 interdomain switches to swap their upstream ports with the non transparent ports.
- After the switchover, the RC 2 processor becomes the active RC processor.
- The RC 2 processor notifies all EP processors that the system is back to normal and that requests can now be made.

The following is the failure switchover procedure:

- The standby RC 2 processor monitors the state of the active RC 1 processor through heart beat and checkpoint messages.
- When there is a failure in the active RC 1 processor, the standby RC 2 processor takes over as the active RC processor. The RC 2 processor configures the all the IDT PES24NT3 interdomain switches to swap their upstream ports with the non transparent ports.
- After the switchover, the RC 2 processor becomes the active RC processor.

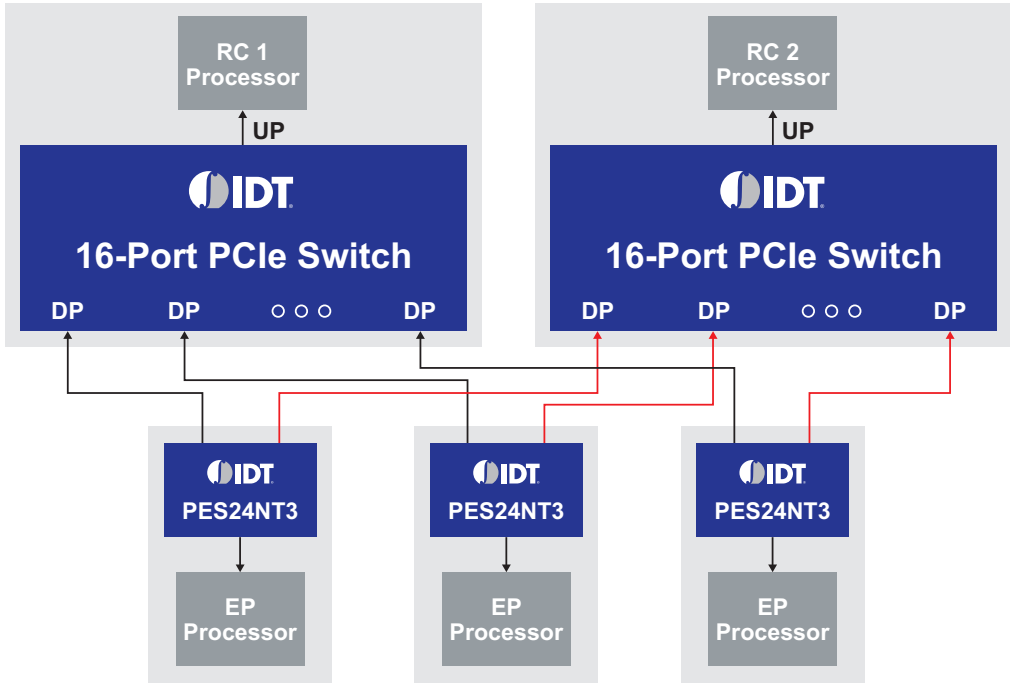


Figure 13. Dual-star topology

Conclusion

Multipeer support can be achieved using standard PCIe switches without any proprietary solutions. A few EP processors have been identified to be part of the solution of a multipeer system. System software in the RC processor has to be extended to support the allocation of the memory map in the system domain. A simple peer-to-peer communication protocol has been shown to transfer large blocks of data between EP processors using PCIe memory read and write operations.

Redundancy is important in a multipeer system. A dual RC processor topology and a dual-star topology provide different levels of redundancy.

The IDT PCIe system-interconnect solution delivers the performance, optimal resource utilization, scalability, RAS and security that are essential for the successful design and deployment of bladed systems. Designed with the needs of leading bladed systems manufacturers in mind, the solution is architected to scale with their performance and capacity needs well into the future.

Glossary

ATU	Address translation unit
BAR	Base address register
DP	Downstream port
EP	Endpoint
FIFO	First in, first out
HBA	Host-bus adapter
I/O	Input/output
KB	Kilobyte
MB	Megabyte
MSI	Message signaled interrupt
NIC	Network interface card
NTB	Nontransparent bridging
PRI	Primary bus number
RAID	Redundant array of inexpensive disks
RAS	Reliability, availability and serviceability
RC	Root complex
Sec	Secondary bus number
Sub	Subordinate bus number
UP	Upstream port

References

PCI Express Base specification Revision 1.1

[1]Intelligent I/O Architecture Specification Version 2.0, I2O Special Interest Group

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.