    - Read the value of the target location.
    - Write the “swap” value back to the target location.
    - Return the original value of the target location.
- CAS (Compare and Swap): Request contains two operands, a “compare” value and a “swap” value
    - Read the value of the target location.
    - Compare that value to the “compare” value.
    - If equal, write the “swap” value back to the target location.
    - Return the original value of the target location.

A given AtomicOp transaction has an associated operand size, and the same size is used for the target location accesses and the returned value. FetchAdd and Swap support operand sizes of 32 and 64 bits. CAS supports operand sizes of 32, 64, and 128 bits.

AtomicOp capabilities are optional normative. Endpoints and Root Ports are permitted to implement AtomicOp Requester capabilities. PCI Express Functions with Memory Space BARs as well as all Root Ports are permitted to implement AtomicOp Completer capabilities. Routing elements (Switches, as well as Root Complexes supporting peer-to-peer access between Root Ports) require AtomicOp routing capability in order to route AtomicOp Requests. AtomicOps are architected for device-to-host, device-to-device, and host-to-device transactions. In each case, the Requester, Completer, and all intermediate routing elements must support the associated AtomicOp capabilities.

AtomicOp capabilities are not supported on PCI Express to PCI/PCI-X Bridges. If need be, Locked Transactions can be used for devices below such Bridges. AtomicOps and Locked Transactions can operate concurrently on the same hierarchy.

Software discovers specific AtomicOp Completer capabilities via three new bits in the Device Capabilities 2 register (see Section 7.5.3.15). For increased interoperability, Root Ports are required to implement certain AtomicOp Completer capabilities in sets if at all (see Section 6.15.3.1). Software discovers AtomicOp routing capability via the AtomicOp Routing Supported bit in the Device Capabilities 2 register. Software discovery of AtomicOp Requester capabilities is outside the scope of this specification, but software must set the AtomicOp Requester Enable bit in a Function’s Device Control 2 register before the Function can initiate AtomicOp Requests (see Section 7.5.3.16).

With routing elements, software can set an AtomicOp Egress Blocking bit (see Section 7.5.3.16) on a Port-by-Port basis to avoid AtomicOp Requests being forwarded to components that shouldn’t receive them, and might handle each as a Malformed TLP, which by default is a Fatal Error. Each blocked Request is handled as an AtomicOp Egress Blocked error, which by default is an Advisory Non-Fatal Error.

AtomicOps are Memory Transactions, so existing standard mechanisms for managing Memory Space access (e.g., Bus Master Enable, Memory Space Enable, and Base Address registers) apply.

6.15.1 AtomicOp Use Models and Benefits

AtomicOps enable advanced synchronization mechanisms that are particularly useful when there are multiple producers and/or multiple consumers that need to be synchronized in a non-blocking fashion. For example, multiple producers can safely enqueue to a common queue without any explicit locking.

AtomicOps also enable lock-free statistics counters, for example where a device can atomically increment a counter, and host software can atomically read and clear the counter.

Direct support for the three chosen AtomicOps over PCI Express enables easier migration of existing high-performance SMP applications to systems that use PCI Express as the interconnect to tightly-coupled accelerators, co-processors, or GP-GPUs. For example, a ported application that uses PCI Express-attached accelerators may be able to use the same synchronization algorithms and data structures as the earlier SMP application.

An AtomicOp to a given target generally incurs latency comparable to a Memory Read to the same target. Within a single hierarchy, multiple AtomicOps can be “in flight” concurrently. AtomicOps generally create negligible disruption to other PCI Express traffic.

Compared to Locked Transactions, AtomicOps provide lower latency, higher scalability, advanced synchronization algorithms, and dramatically less impact to other PCI Express traffic.

6.15.2 AtomicOp Transaction Protocol Summary

Detailed protocol rules and requirements for AtomicOps are distributed throughout the rest of this specification, but here is a brief summary plus some unique requirements.

- AtomicOps are Non-Posted Memory Transactions, supporting 32- and 64-bit address formats.
- FetchAdd, Swap, and CAS each use a distinct type code.
- The Completer infers the operand size from the Length field value and type code in the AtomicOp Request.
- The endian format used by AtomicOp Completers to read and write data at the target location is implementation specific, and permitted to be whatever the Completer determines to be appropriate for the target memory (e.g., little-endian, big-endian, etc.). See Section 2.2.2 .
- If an AtomicOp Requester supports Address Translation Services (ATS), the Requester is permitted to use a Translated address in an AtomicOp Request only if the Translated address has appropriate access permissions. Specifically, the Read (R) and Write (W) fields must both be Set, and the Untranslated access only (U) field must be Clear. See Section 2.2.4.1 .
- If a component supporting Access Control Services (ACS) supports AtomicOp routing or AtomicOp Requester capability, it handles AtomicOp Requests and Completions the same as with other Memory Requests and Completions with respect to ACS functionality.
- The No Snoop attribute is applicable and permitted to be Set with AtomicOp Requests, but atomicity must be guaranteed regardless of the No Snoop attribute value.
- The Relaxed Ordering attribute is applicable and permitted to be Set with AtomicOp Requests, where it affects the ordering of both the Requests and their associated Completions.
- Ordering requirements for AtomicOp Requests are similar to those for Non-Posted Write Requests. Thus, if a Requester wants to ensure that an AtomicOp Request is observed by the Completer before a subsequent Posted or Non-Posted Request, the Requester must wait for the AtomicOp Completion before issuing the subsequent Request.
- Ordering requirements for AtomicOp Completions are similar to those for Read Completions.
- Unless there’s a higher precedence error, an AtomicOp-aware Completer must handle a Poisoned AtomicOp Request as a Poisoned TLP Received error, and must also return a Completion with a Completion Status of Unsupported Request (UR). See Section 2.7.2.2 . The value of the target location must remain unchanged.
- If the Completer of an AtomicOp Request encounters an uncorrectable error accessing the target location or carrying out the Atomic operation, the Completer must handle it as a Completer Abort (CA). The subsequent state of the target location is implementation specific.
- AtomicOp-aware Completers are required to handle any properly formed AtomicOp Requests with types or operand sizes they don’t support as an Unsupported Request (UR). If the Length field in an AtomicOp Request contains an unarchitected value, the Request must be handled by an AtomicOp-aware Completer as a Malformed TLP. See Section 2.2.7 .
- If any Function in a Multi-Function Device supports AtomicOp Completer or AtomicOp routing capability, all Functions with Memory Space BARs in that device must decode properly formed AtomicOp Requests and handle any they don’t support as an Unsupported Request (UR). Note that in such devices, Functions lacking

AtomicOp Completer capability are forbidden to handle properly formed AtomicOp Requests as Malformed TLPs.

- If an RC has any Root Ports that support AtomicOp routing capability, all RCiEPs in the RC reachable by forwarded AtomicOp Requests must decode properly formed AtomicOp Requests and handle any they don't support as an Unsupported Request (UR).
- With an AtomicOp Request having a supported type and operand size, the AtomicOp-aware Completer is required either to carry out the Request or handle it as Completer Abort (CA) for any location in its target Memory Space. Completers are permitted to support AtomicOp Requests on a subset of their target Memory Space as needed by their programming model (see Section 2.3.1). Memory Space structures defined or inherited by PCI Express (e.g., the MSI-X Table structure) are not required to be supported as AtomicOp targets unless explicitly stated in the description of the structure.
- For a Switch or an RC, when AtomicOp Egress Blocking is enabled in an Egress Port, and an AtomicOp Request targets going out that Egress Port, the Egress Port must handle the Request as an AtomicOp Egress Blocked error[127] (see Figure 6-2) and must also return a Completion with a Completion Status of UR. If the severity of the AtomicOp Egress Blocked error is non-fatal, this case must be handled as an Advisory Non-Fatal Error as described in Section 6.2.3.2.4.1 .

6.15.3 Root Complex Support for AtomicOps

RCs have unique requirements and considerations with respect to AtomicOp capabilities.

6.15.3.1 Root Ports with AtomicOp Completer Capabilities

AtomicOp Completer capability for a Root Port indicates that the Root Port supports receiving at its Ingress Port AtomicOp Requests that target host memory or Memory Space allocated by a Root Port BAR. This is independent of any RCiEPs that have AtomicOp Completer capabilities.

If a Root Port implements any AtomicOp Completer capability for host memory access, it must implement all 32-bit and 64-bit AtomicOp Completer capabilities. Implementing 128-bit CAS Completer capability is optional.

If an RC has one or more Root Ports that implement AtomicOp Completer capability, the RC must ensure that host memory accesses to a target location on behalf of a given AtomicOp Request are performed atomically with respect to each host processor or device access to that target location range.

If a host processor supports atomic operations via its instruction set architecture, the RC must also ensure that host memory accesses on behalf of a given AtomicOp Request preserve the atomicity of any host processor atomic operations.

6.15.3.2 Root Ports with AtomicOp Routing Capability

As with other PCI Express Transactions, the support for peer-to-peer routing of AtomicOp Requests and Completions between Root Ports is optional and implementation dependent. If an RC supports AtomicOp routing capability between two or more Root Ports, it must indicate that capability in each associated Root Port via the AtomicOp Routing Supported bit in the Device Capabilities 2 register.

127. Though an AtomicOp Egress Blocked error is handled by returning a Completion with UR Status, the error is not otherwise handled as an Unsupported Request. For example, it does not set the Unsupported Request Detected bit in the Device Status register.

An RC is not required to support AtomicOp routing between all pairs of Root Ports that have the AtomicOp Routing Supported bit Set. An AtomicOp Request that would require routing between unsupported pairs of Root Ports must be handled as an Unsupported Request (UR), and reported by the "sending" Port.

The AtomicOp Routing Supported bit must be Set for any Root Port that supports forwarding of AtomicOp Requests initiated by host software or RCiEPs. The AtomicOp Routing Supported bit must be Set for any Root Ports that support forwarding of AtomicOp Requests received on their Ingress Port to RCiEPs.

6.15.3.3 RCs with AtomicOp Requester Capabilities

An RC is permitted to implement the capability for either host software or RCiEPs to initiate AtomicOp Requests.

Software discovery of AtomicOp Requester capabilities is outside the scope of this specification.

If an RC supports software-initiated AtomicOp Requester capabilities, the specific mechanisms for how software running on a host processor causes the RC to generate AtomicOp Requests is outside the scope of this specification.

IMPLEMENTATION NOTE

Generating AtomicOp Requests via Host Processor Software

If a host processor instruction set architecture (ISA) supports atomic operation instructions that directly correspond to one or more PCI Express AtomicOps, an RC might process the associated internal atomic transaction that targets PCI Express Memory Space much like it processes the internal read transaction resulting from a processor load instruction. However, instead of "exporting" the internal read transaction as a PCI Express Memory Read Request, the RC would export the internal atomic transaction as a PCI Express AtomicOp Request. Even if an RC uses the "export" approach for some AtomicOp types and operand sizes, it would not need to use this approach for all.

For AtomicOp types and operand sizes where the RC does not use the "export" approach, the RC might use an RC register-based mechanism similar to one where some PCI host bridges use CONFIG_ADDRESS and CONFIG_DATA registers to generate Configuration Requests. Refer to the [PCI] for details.

The "export" approach may permit a large number of concurrent AtomicOp Requests without becoming RC register limited. It may also be easier to support AtomicOp Request generation from user space software using this approach.

The RC register-based mechanism offers the advantage of working for all AtomicOp types and operand sizes even if the host processor ISA doesn't support the corresponding atomic instructions. It might also support a polling mode for waiting on AtomicOp Completions as opposed to stalling the processor while waiting for a Completion.

6.15.4 Switch Support for AtomicOps

If a Switch supports AtomicOp routing capability for any of its Ports, it must do so for all of them.

6.16 Dynamic Power Allocation (DPA) Capability

A common approach to managing power consumption is through a negotiation between the device driver, operating system, and executing applications. Adding Dynamic Power Allocation for such devices is anticipated to be done as an extension of that negotiation, through software mechanisms that are outside of the scope of this specification. Some

devices do not have a device specific driver to manage power efficiently. The DPA Capability provides a mechanism to allocate power dynamically for these types of devices. DPA is optional normative functionality applicable to Endpoint Functions that can benefit from the dynamic allocation of power and do not have an alternative mechanism. If supported, the Emergency Power Reduction State, over-rides the mechanisms listed here (see Section 6.25).

The DPA Capability enables software to actively manage and optimize Function power usage when in the D0 state. DPA is not applicable to power states D1-D3 therefore the DPA Capability is independently managed from the PCI-PM Capability.

DPA defines a set of power substates, each of which with an associated power allocation. Up to 32 substates [0..31] can be defined per Function. Substate 0, the default substate, indicates the maximum power the Function is ever capable of consuming.

Substates must be contiguously numbered from 0 to Substate_Max, as defined in Section 7.9.12.2 . Each successive substate has a power allocation lower than or equal to that of the prior substate. For example, a Function with four substates could be defined as follows:

1. Substate 0 (the default) defines a power allocation of 25 Watts.
2. Substate 1 defines a power allocation of 20 Watts.
3. Substate 2 defines a power allocation of 20 Watts.
4. Substate 3 defines a power allocation of 10 Watts.

When the Function is initialized, it will operate within the power allocation associated with substate 0. Software is not required to progress through intermediate substates. Over time, software may dynamically configure the Function to operate at any of the substates in any sequence it chooses. Software is permitted to configure the Function to operate at any of the substates before the Function completes a previously initiated substate transition.

On the completion of the substate transition(s) the Function must compare its substate with the configured substate. If the Function substate does not match the configured substate, then the Function must begin transition to the configured substate. It is permitted for the Function to dynamically alter substate transitions on Configuration Requests instructing the Function to operate in a new substate.

In the prior example, software can configure the Function to transition to substate 4, followed by substate 1, followed by substate 3, and so forth. As a result, the Function must be able to transition between any substates when software configures the associated control field.

The Substate Control Enabled bit provides a mechanism that allows the DPA Capability to be used in conjunction with the software negotiation mechanism mentioned above. When Set, power allocation is controlled by the DPA Capability. When Clear, the DPA Capability is disabled, and the Function is not permitted to directly initiate substate transitions based on configuration of the Substate Control register field. At an appropriate point in time, software participating in the software negotiation mechanism mentioned above clears the bit, effectively taking over control of power allocation for the Function.

It is required that the Function respond to Configuration Space accesses while in any substate.

At any instant, the Function must never draw more power than it indicates through its Substate Status. When the Function is configured to transition from a higher power substate to a lower power substate, the Function's Substate Status must indicate the higher power substate during the transition, and must indicate the lower power substate after completing the transition. When the Function is configured to transition from a lower power substate to a higher power substate, the Function's Substate Status must indicate the higher power substate during the transition, as well as after completing the transition.

Due to the variety of applications and the wide range of maximum power required for a given Function, the transition time required between any substates is implementation specific. To enable software to construct power management policies (outside the scope of this specification), the Function defines two Transition Latency Values. Each of the Function substates associates its maximum Transition Latency with one of the Transition Latency Values, where the

maximum Transition Latency is the time it takes for the Function to enter the configured substate from any other substate. A Function is permitted to complete the substate transition faster than the maximum Transition Latency for the substate.

6.16.1 DPA Capability with Multi-Function Devices

It is permitted for some or all Functions of a Multi-Function Device to implement a DPA Capability. The power allocation for the Multi-Function Device is the sum of power allocations set by the DPA Capability for each Function. It is permitted for the DPA Capability of a Function to include the power allocation for the Function itself as well as account for power allocation for other Functions that do not implement a DPA Capability. The association between multiple Functions for DPA is implementation specific and beyond the scope of this specification.

6.17 TLP Processing Hints (TPH)

TLP Processing Hints is an optional feature that provides hints in Request TLP headers to facilitate optimized processing of Requests that target Memory Space. These Processing Hints enable the system hardware (e.g., the Root Complex and/or Endpoints) to optimize platform resources such as system and memory interconnect on a per TLP basis. The TPH mechanism defines Processing Hints that provide information about the communication models between Endpoints and the Root-complex. Steering Tags are system-specific values used to identify a processing resource that a Requester explicitly targets. System software discovers and identifies TPH capabilities to determine the Steering Tag allocation for each Function that supports TPH.

6.17.1 Processing Hints

The Requester provides hints to the Root Complex or other targets about the intended use of data and data structures by the host and/or device. The hints are provided by the Requester, which has knowledge of upcoming Request patterns, and which the Completer would not be able to deduce autonomously (with good accuracy). Cases of interest to distinguish with such hints include:

DWHR: Device writes then host reads soon

HWDR: Device reads data that the Host is believed to have recently written

D*D*: Device writes/reads, then device reads/writes soon

Includes DWDW, DWDR, DRDW, DRDR

Bi-Directional: Data structure that is shared and has equal read/write access by host and device.

The usage models are mapped to the Processing Hint encodings as described in Table 6-12 .

Table 6-12 Processing Hint Mapping

PH[1:0] (b)	Processing Hint	Usage Model
00	Bi-directional data structure	Bi-Directional shared data structure
01	Requester	D*D*
10	Target	DWHR HWDR

PH[1:0] (b)	Processing Hint	Usage Model
11	Target with Priority	Same as target but with temporal re-use priority

6.17.2 Steering Tags

Functions that intend to target a TLP towards a specific processing resource such as a host processor or system cache hierarchy require topological information of the target cache (e.g., which host cache). Steering Tags are system-specific values that provide information about the host or cache structure in the system cache hierarchy. These values are used to associate processing elements within the platform with the processing of Requests.

Software programmable Steering Tag values to be used are stored in an ST Table that is permitted to be located either in the TPH Requester Extended Capability structure (see Section 7.9.13) or combined with the MSI-X Table (see Section 7.7), but not in both locations for a given Function. When the ST Table is combined with the MSI-X Table, the 2 most significant bytes of the Vector Control register of each MSI-X Table entry are used to contain the Steering Tag value.

The choice of ST Table location is implementation specific and is discoverable by software. A Function that implements MSI-X is permitted to locate the ST Table in either location (see Section 7.9.13.2). A Function that implements both MSI and MSI-X is permitted to combine the ST Table with the MSI-X Table and use it, even when MSI-X is disabled (i.e. when MSI is enabled). Each ST Table entry is 2 bytes. The size of the ST Table is indicated in the TPH Requester Extended Capability structure.

For some usage models the Steering Tags are not required or not provided, and in such cases a Function is permitted to use a value of all zeroes in the ST field to indicate no ST preference. The association of each Request with an ST Table entry is device specific and outside the scope of this specification.

6.17.3 ST Modes of Operation

The ST Table Location field in the TPH Requester Extended Capability structure indicates where (if at all) the ST Table is implemented by the Function. If an ST Table is implemented, software can program it with the system-specific Steering Tag values.

Table 6-13 ST Modes of Operation

ST Mode Select [2:0] (b)	ST Mode Name	Description
000	No ST Mode	The Function must use a value of all zeroes for all Steering Tags.
001	Interrupt Vector Mode	Each Steering Tag is selected by an MSI/MSI-X interrupt vector number. The Function is required to use the Steering Tag value from an ST Table entry that can be indexed by a valid MSI/MSI-X interrupt vector number.
010	Device Specific Mode	It is recommended for the Function to use a Steering Tag value from an ST Table entry, but it is not required.
All other encodings	Reserved	Reserved for future use.

In the No ST Mode of operation, the Function must use a value of all zeroes for each Steering Tag, enabling the use of Processing Hints without software-provided Steering Tags.

In the Interrupt Vector Mode of operation, Steering Tags are selected from the ST Table using MSI/MSI-X interrupt vector numbers. For Functions that have MSI enabled, the Function is required to select tags within the range specified by the Multiple Message Enable field in the MSI Capability structure. For Functions that have MSI-X enabled, the Function is required to select tags within the range of the MSI-X Table size. If the ST Table Size is smaller than the enabled range of interrupt vector numbers, the Function is permitted to either not use TPH for certain transactions, to use TPH with a Steering Tag of 0 or to use TPH with an implementation defined mechanism used to select a Steering Tag value from the ST Table. If the ST Table Size is larger than the enabled range of interrupt vector numbers, ST Table Entries corresponding to out of range interrupt vector numbers are ignored by the Function.

In the Device Specific Mode of operation, the assignment of the Steering Tags to Requests is device specific. The number of Steering Tags used by the Function is permitted to be different than the number of interrupt vectors allocated for the Function, irrespective of the ST Table location, and Steering Tag values used in Requests are not required to come from the architected ST Table.

A Function that is capable of generating TPH Requests is required to support the No ST Mode of operation. Support for other ST Modes of operation is optional. Only one ST Mode of operation can be selected at a time by programming ST Mode Select.

IMPLEMENTATION NOTE

ST Table Programming

To ensure that deterministic Steering Tag values are used in Requests, it is recommended that software either quiesce the Function or disable the TPH Requester capability during the process of performing ST Table updates. Failure to do so may result in non-deterministic values of ST values being used during ST Table updates.

6.17.4 TPH Capability

TPH capabilities are optional normative. Each Function capable of generating Request TLPs with TPH is required to implement a TPH Requester Extended Capability structure. Functions that support processing of TLPs with TPH as Completers are required to indicate TPH Completer capability via the Device Capabilities 2 register. TPH is architected to be applied for transactions that target Memory Space, and is applicable for transaction flows between device-to-host, device-to-device and host-to-device. In each case for TPH to be supported, the Requester, Completer, and all intermediate routing elements must support the associated TPH capabilities.

Software discovers the Requester capabilities via the TPH Requester Extended Capability structure and Completer capabilities via the Device Capabilities 2 Register (see Section 7.5.3.15). Software must program the TPH Requester Enable field in the TPH Requester Extended Capability structure to enable the Function to initiate Requests with TPH.

TPH only provides additional information to enable optimized processing of Requests that target Memory Space, so existing mechanisms and rules for managing Memory Space access (e.g., Bus Master Enable, Memory Space Enable, and Base Address registers) are not altered.

6.18 Latency Tolerance Reporting (LTR) Mechanism

The Latency Tolerance Reporting (LTR) mechanism enables Endpoints to report their service latency requirements for Memory Reads and Writes to the Root Complex, so that power management policies for central platform resources (such as main memory, RC internal interconnects, and snoop resources) can be implemented to consider Endpoint service requirements. The LTR Mechanism does not directly affect Link power management or Switch internal power management, although it is possible that indirect effects will occur.

The implications of “latency tolerance” will vary significantly between different device types and implementations. When implementing this mechanism, it will generally be desirable to consider if service latencies impact functionality or only performance, if performance impacts are linear, and how much it is possible for the device to use buffering and/or other techniques to compensate for latency sensitivities.

The Root Complex is not required to honor the requested service latencies, but is strongly encouraged to provide a worst case service latency that does not exceed the latencies indicated by the LTR mechanism.

LTR support is discovered and enabled through reporting and control registers described in Chapter 7 . Software must not enable LTR in an Endpoint unless the Root Complex and all intermediate Switches indicate support for LTR. Note that it is not required that all Endpoints support LTR to permit enabling LTR in those Endpoints that do support it. When enabling the LTR mechanism in a hierarchy, devices closest to the Root Port must be enabled first.

If an LTR Message is received at a Downstream Port that does not support LTR or if LTR is not enabled, the Message must be treated as an Unsupported Request.

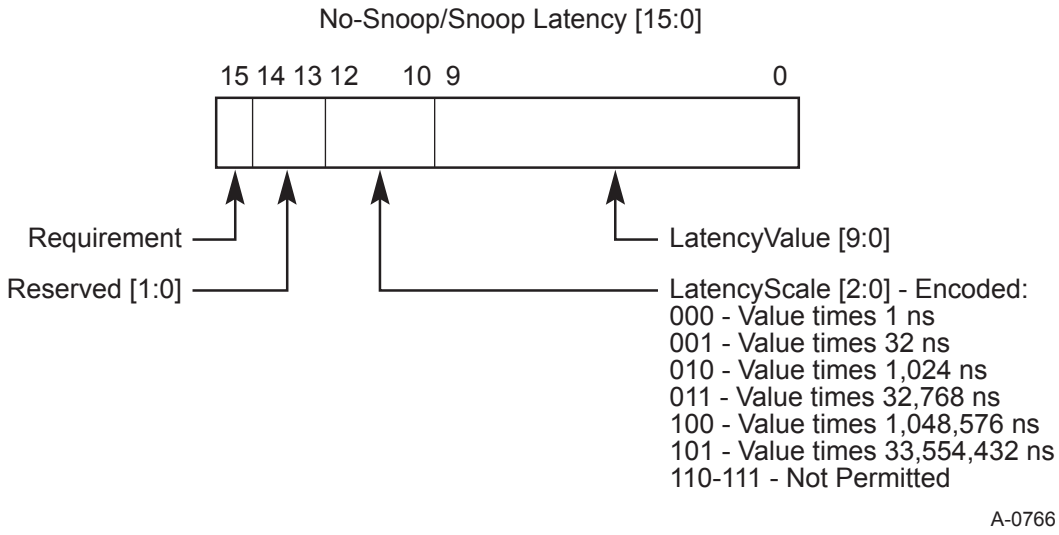

Figure 6-15 Latency Fields Format for LTR Messages

No-Snoop Latency and Snoop Latency: As shown in Figure 6-15 , these fields include a Requirement bit that indicates if the device has a latency requirement for the given type of Request. If the Requirement bit is Set, the LatencyValue and LatencyScale fields describe the latency requirement. If the Requirement bit is Clear, there is no latency requirement and the LatencyValue and LatencyScale fields are ignored. With any LTR Message transmission, it is permitted for a device to indicate that a requirement is being reported for only no-snoop Requests, for only snoop Requests, or for both types of Requests. It is also permitted for a device to indicate that it has no requirement for either type of traffic, which it does by clearing the Requirement bit in both fields.

Each field also includes value and scale fields that encode the reported latency. Values are multiplied by the indicated scale to yield an absolute time value, expressible in a range from 1 ns to $2^{25}*(2^{10}-1)$ = 34,326,183,936 ns.

Setting the value and scale fields to all 0’s indicates that the device will be impacted by any delay and that the best possible service is requested.

If a device doesn’t implement or has no service requirements for a particular type of traffic, then it must have the Requirement bit clear for the associated latency field.

When directed to a non-D0 state by a Write to the PMCSR register, if a device's most recently transmitted LTR message (since the last DL_Down to DL_Up transition) reported one or both latency fields with any Requirement bit set, then it must send a new LTR Message with both of the Requirement bits clear prior to transitioning to the non-D0 state.

When the LTR Mechanism Enable bit is cleared, if a device's most recently sent LTR Message (since the last DL_DOWN to DL_Up transition) reported latency tolerance values with any Requirement bit set, then one of the following applies:

- If the bit was cleared due to a Configuration Write to the Device Control 2 register, the device must send a new LTR Message with all the Requirement bits clear.
- If the bit was cleared due to an FLR, it is strongly recommended that the device send a new LTR Message with all the Requirement bits clear.

When a Downstream Port goes to DL_Down status, any previous latencies recorded for that Port must be treated as invalid.

An LTR Message from a device reflects the tolerable latency from the perspective of the device, for which the platform must consider the service latency itself, plus the delay added by the use of Clock Power Management (CLKREQ#), if applicable. The service latency itself is defined as follows:

- When a device issues a Non-Posted Request, service latency of that Request is the delay from transmission of the last symbol of the Request TLP to the receipt of the first symbol of the first Completion TLP for that Request[128] .
- When a device issues one or more Posted Requests such that it cannot issue another Posted Request due to Flow Control backpressure, service latency of the blocked Request is the delay from the transmission of the last symbol of the previous Posted Request to the receipt of the first symbol of the DLLP returning the credit(s) that allows transmission of the blocked Request[118].

If Clock Power Management is used, then the platform implementation-dependent period between when a device asserts CLKREQ# and the device receives a valid clock signal constitutes an additional component of the platform service latency that must be comprehended by the platform when setting platform power management policy.

It is recommended that Endpoints transmit an LTR Message Upstream shortly after LTR is enabled.

It is strongly recommended that Endpoints send no more than two LTR Messages within any 500 µs time period, except where required to by the specification. Downstream Ports must not generate an error if more than two LTR Messages are received within a 500 µs time period, and must properly handle all LTR messages regardless of the time interval between them.

Multi-Function Devices (MFDs) associated with an Upstream Port must transmit a "conglomerated" LTR Message Upstream according to the following rules:

- The acceptable latency values for the Message sent Upstream by the MFD must reflect the lowest values associated with any Function.
  - It is permitted that the snoop and no-snoop latencies reported in the conglomerated Message are associated with different Functions.
  - If none of the Functions report a requirement for a certain type of traffic (snoop/no-snoop), the Message sent by the MFD must not set the Requirement bit corresponding to that type of traffic.
- The MFD must transmit a new LTR Message Upstream when any Function of the MFD changes the values it has reported internally in such a way as to change the conglomerated value earlier reported by the MFD.

128. For this definition, all of the symbols of a DLLP or TLP are included. For 8b/10b, the first and last symbols are framing symbols (SDP, STP or END, see Section 4.2.1). For 128b/130b, the first symbol of a packet is the first symbol of a framing token (SDP or STP) and the last symbol of a packet is the last symbol of a CRC or LCRC (see Section 4.2.2).

Switches must collect the Messages from Downstream Ports that have the LTR mechanism enabled and transmit a "conglomerated" Message Upstream according to the following rules:

- If a Switch supports the LTR feature, it must support the feature on its Upstream Port and all Downstream Ports.
- A Switch Upstream Port is permitted to transmit LTR Messages only when its LTR Mechanism Enable bit is Set or shortly after software clears its LTR Mechanism Enable bit as described earlier in this section.
- The acceptable latency values for the Message sent Upstream by the Switch must be calculated as follows:
  - If none of the Downstream Ports receive an LTR Message containing a requirement for a certain type of traffic (snoop/no-snoop), then any LTR Message sent by the Switch must not set the Requirement bit corresponding to that type of traffic.
  - Define LTRdnport[N] as the value reported in the LTR Message received at Downstream Port N, with these adjustments if applicable:
    - LTRdnport[N] is effectively infinite if the Requirement bit is clear or if a Not Permitted LatencyScale value is used
    - LTRdnport[N] must be 0 if the Requirement bit is 1 and the LatencyValue field is all 0's regardless of the LatencyScale value
  - Define LTRdnportMin as the minimum value of LTRdnport[N] across all Downstream Ports
  - Define Lswitch as all latency induced by the Switch
    - If Lswitch dynamically changes based on the Switch's operational mode, the Switch must not allow Lswitch to exceed 20% of LTRdnportMin, unless Lswitch for the Switch's lowest latency mode is greater, in which case the lowest latency state must be used
  - Calculate the value to transmit upstream, LTRconglomerated, as LTRdnportMin - Lswitch, unless this value is less than 0 in which case LTRconglomerated is 0
  - If LTRconglomerated is 0, both the LatencyValue and LatencyScale fields must be all 0's in the conglomerated LTR message
- A new LTR message must be transmitted Upstream if the conglomerated latencies are changed as a result of DL_Down invalidating the previous latencies recorded for that Port.
- If a Switch Downstream Port has the LTR Mechanism Enable bit cleared, the Latency Tolerance values recorded for that Port must be treated as invalid, and the latencies to be transmitted Upstream updated and a new conglomerated Message transmitted Upstream if the conglomerated latencies are changed as a result.
- A Switch must transmit an LTR Message Upstream when any Downstream Port/Function changes the latencies it has reported in such a way as to change the conglomerated latency reported by the Switch.
- A Switch must not transmit LTR Messages Upstream unless triggered to do so by one of the events described above.

The RC is permitted to delay processing of device Request TLPs provided it satisfies the device's service requirements.

When the latency requirement is updated during a series of Requests, it is required that the updated latency figure be comprehended by the RC prior to servicing a subsequent Request. In all cases the updated latency value must take effect within a time period equal to or less than the previously reported latency requirement. It is permitted for the RC to comprehend the updated latency figure earlier than this limit.

IMPLEMENTATION NOTE
Optimal Use of LTR

It is recommended that Endpoints transmit an updated LTR Message each time the Endpoint's service requirements change. If the latency tolerance is being reduced, it is recommended to transmit the updated LTR Message ahead of the first anticipated Request with the new requirement, allowing the amount of time indicated in the previously issued LTR Message. If the tolerance is being increased, then the update should immediately follow the final Request with the preceding latency tolerance value.

Typically, the Link will be in ASPM L1, and, if Clock Power Management (Clock PM) is supported, CLKREQ# will be deasserted, at the time an Endpoint reaches an internal trigger that causes the Endpoint to initiate Requests to the RC. The following text shows an example of how LTR is applied in such a case. Key time points are illustrated in Figure 6-16 .

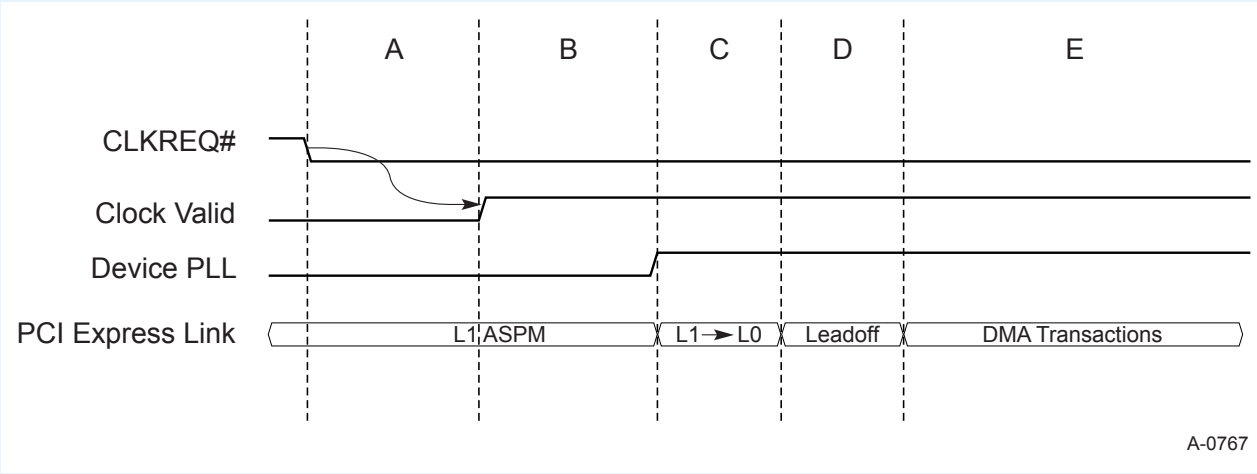

Figure 6-16 CLKREQ# and Clock Power Management

Time A is a platform implementation-dependent period between when a device asserts CLKREQ# and the device receives a valid clock signal. This value will not exceed the latency in effect.

Time B is the device implementation-dependent period between when a device has a valid clock and it can initiate the retraining sequence to transition from L1 ASPM to L0.

Time C is the period during which the transition from L1 ASPM to L0 takes place

Time D for a Read transaction is the time between the transmission of the END symbol in the Request TLP to the receipt of the STP symbol in the Completion TLP for that Request. Time D for a Write transaction is the time between the transmission of the END symbol of the TLP that exhausts the FC credits to the receipt of the SDP symbol in the DLLP returning more credits for that Request type. This value will not exceed the latency in effect.

Time E is the period where the data path from the Endpoint to system memory is open, and data transactions are not subject to the leadoff latency.

The LTR latency semantic reflects the tolerable latency seen by the device as measured by one or both of the following:

Case 1: the device may or may not support Clock PM, but has not deasserted its CLKREQ# signal - The latency observed by the device is represented in Figure 6-16 as the sum of times C and D.

Case 2: the device supports Clock PM and has deasserted CLKREQ#- The latency observed by the device is represented as the sum of times A, C, and D.

To effectively use the LTR mechanism in conjunction with Clock PM, the device will know or be able to measure times B and C, so that it knows when to assert CLKREQ#. The actual values of Time A, Time C, and Time D may vary dynamically, and it is the responsibility of the platform to ensure the sum will not exceed the latency.

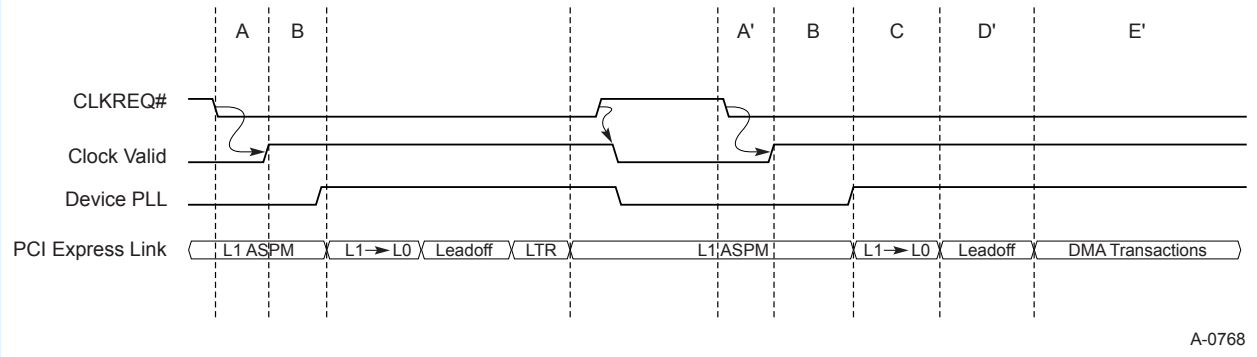

Figure 6-17 Use of LTR and Clock Power Management

In a very simple model, an Endpoint may choose to implement LTR as shown in Figure 6-17 . When an Endpoint determines that it is idle, it sends an LTR Message with the software configured maximum latency or the maximum latency the Endpoint can support.

When the Endpoint determines that it has a need to maintain sustained data transfers with the Root Complex, the Endpoint sends a new LTR Message with a shorter latency (at Time E). This LTR Message is sent prior to the next data flush by a time equal to the maximum latency sent before (the time between Time E and Time D'). In between Time E and Time A', the Endpoint can return to a low power state, while the platform transitions to a state where it can provide the shorter latency when the device next needs to transmit data.

Note that the RC may delay processing of device Request TLPs, provided it satisfies the device's service requirements. If, for example, an Endpoint connected to Root Port 1 reports a latency tolerance of 100 µs, and an Endpoint on Root Port 2 report a value of 30 µs, the RC might implement a policy of stalling an initial Request following an idle period from Root Port 1 for 70 µs before servicing the Request with a 30 µs latency, thus providing a perceived service latency to the first Endpoint of 100 µs. This RC behavior provides the RC the ability to batch together Requests for more efficient servicing.

It is possible that, after it is determined that the RC can service snoop and no-snoop Requests from all Endpoints within the maximum snoop and maximum no-snoop time intervals, this information may be communicated to Endpoints by updating the Max Snoop LatencyValue, Max Snoop LatencyScale and Max No-Snoop LatencyValue, Max No-Snoop LatencyScale fields. The intention of this communication would be to prevent Endpoints from sending needless LTR updates.

When an Endpoint's LTR value for snoop Requests changes to become larger (looser) than the value indicated in the Max Snoop LatencyValue/Scale fields, it is recommended that the Endpoint send an LTR message with the snoop LTR value indicated in the Max Snoop LatencyValue/Scale fields. Likewise, when an Endpoint's LTR value for no-snoop Requests changes to become larger (looser) than the value indicated in the Max No-Snoop

LatencyValue/Scale fields, it is recommended that the Endpoint send an LTR message with the no-snoop LTR value indicated in the Max No-Snoop LatencyValue/Scale fields.

It is recommended that Endpoints buffer Requests as much as possible, and then use the full Link bandwidth in bursts that are as long as the Endpoint can practically support, as this will generally lead to the best overall platform power efficiency.

Note that LTR may be enabled in environments where not all Endpoints support LTR, and in such environments, Endpoints that do not support LTR may experience suboptimal service.

6.19 Optimized Buffer Flush/Fill (OBFF) Mechanism

The Optimized Buffer Flush/Fill (OBFF) Mechanism enables a Root Complex to report to Endpoints (throughout a hierarchy) time windows when the incremental platform power cost for Endpoint bus mastering and/or interrupt activity is relatively low. Typically this will correspond to time that the host CPU(s), memory, and other central resources associated with the Root Complex are active to service some other activity, for example the operating system timer tick. The nature and determination of such windows is platform/implementation specific.

An OBFF indication is a hint - Functions are still permitted to initiate bus mastering and/or interrupt traffic whenever enabled to do so, although this will not be optimal for platform power and should be avoided as much as possible.

OBFF is indicated using either of the WAKE# signal or a message (see Section 2.2.8.9). The message is to be used exclusively on interconnects where the WAKE# signal is not available. WAKE# signaling of OBFF or CPU Active must only be initiated by a Root Port when the system is in an operational state, which in an ACPI compliant system corresponds to the S0 state. Functions that are in a non-D0 state must not respond to OBFF or CPU Active signaling.

The OBFF message routing is defined as 100b, for point-to-point, and is only permitted to be transmitted in the Downstream direction. There are multiple OBFF events distinguished. When using the OBFF Message, the OBFF Code field is used to distinguish between different OBFF cases:

1111b “CPU Active” - System fully active for all Device actions including bus mastering and interrupts

0001b “OBFF” - System memory path available for Device memory read/write bus master activities

0000b “Idle” - System in an idle, low power state

All other codes are Reserved.

These codes correspond to various assertion patterns of WAKE# when using WAKE# signaling, as shown in Figure 6-18 . There is one negative-going transition when signaling OBFF and two negative going transitions each time CPU Active is signaled. The electrical parameters required when using WAKE# are defined in the WAKE# Signaling section of [CEM-2.0] (or later).

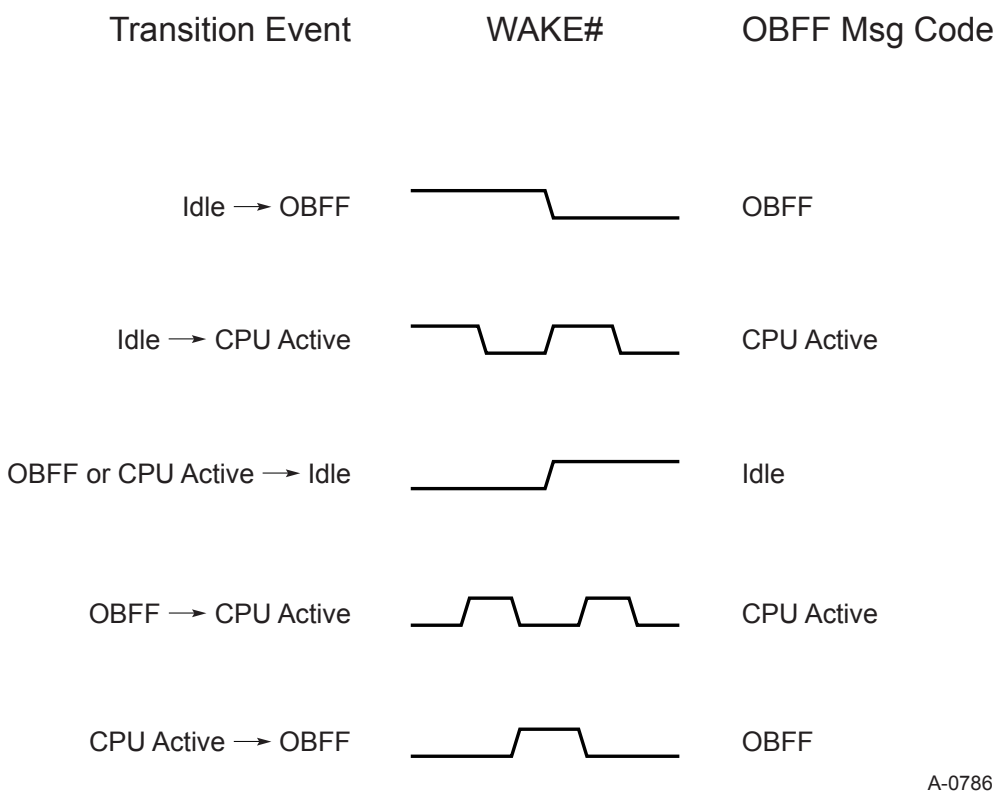

Figure 6-18 Codes and Equivalent WAKE# Patterns

When an OBFF Message is received that indicates a Reserved code, the Receiver, if OBFF is enabled, must treat the indication as a "CPU Active" indication.

An OBFF Message received at a Port that does not implement OBFF or when OBFF is not enabled must be handled as an Unsupported Request (UR). This is a reported error associated with the receiving Port (see Section 6.2). If a Port has OBFF enabled using WAKE# signaling, and that Port receives an OBFF Message, the behavior is undefined.

OBFF indications reflect central resource power management state transitions, and are signaled using WAKE# when this is supported by the platform topology, or using a Message when WAKE# is not available. OBFF support is discovered and enabled through reporting and control registers described in Chapter 7 . Software must not enable OBFF in an Endpoint unless the platform supports delivering OBFF indications to that Endpoint.

When the platform indicates the start of a CPU Active or OBFF window, it is recommended that the platform not return to the Idle state in less than 10 µs. It is permitted to indicate a return to Idle in advance of actually entering platform idle, but it is strongly recommended that this only be done to prevent late Endpoint activity from causing an immediate exit from the idle state, and that the advance time be as short as possible.

It is recommended that Endpoints not assume CPU Active or OBFF windows will remain open for any particular length of time.

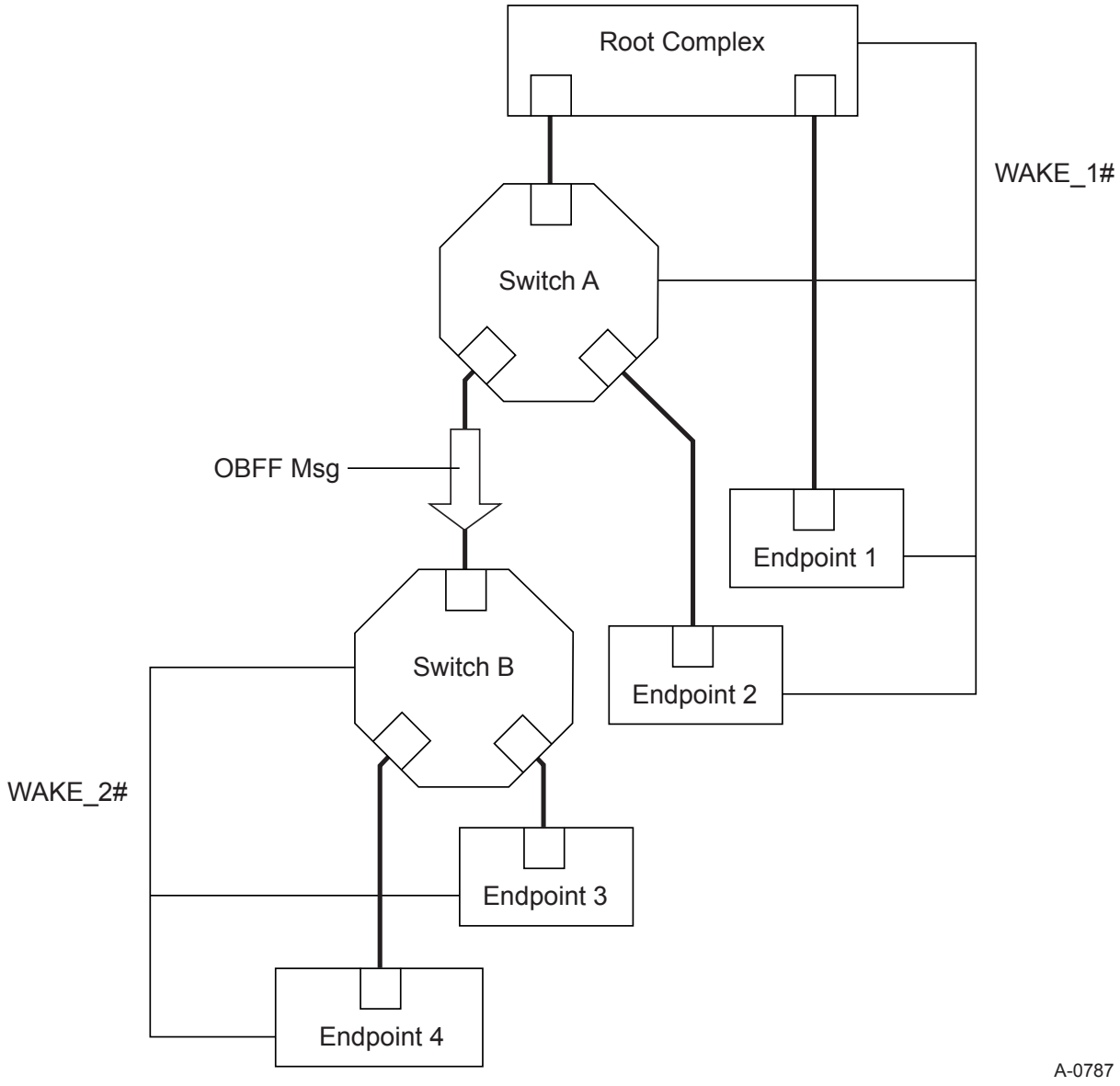

Figure 6-19 Example Platform Topology Showing a Link Where OBFF is Carried by Messages

Figure 6-19 shows an example system where it is necessary for a Switch (A) to translate OBFF indications received using WAKE# into OBFF Messages, which in this case are received by another Switch (B) and translated back to using WAKE# signaling. A HwInit configuration mechanism (set by hardware or firmware) is used to identify cases such as shown in this example (where the link between Switch A and Switch B requires the use of OBFF Messages), and system firmware/software must configure OBFF accordingly.

When a Switch is configured to use OBFF Message signaling at its Upstream Port and WAKE# at one or more Downstream Ports, or vice-versa, when enabled for OBFF, the Switch is required to convert all OBFF indications received at the Upstream Port into the appropriate form at the Downstream Port(s).

When using WAKE#, the enable for any specific Root Port enables the global use of WAKE# unless there are multiple WAKE# signals, in which case only the associated WAKE# signals are affected. When using Message signaling for OBFF, the enable for a particular Root Port enables transmission of OBFF messages from that Root Port only. To ensure OBFF is

fully enabled in a platform, all Root Ports indicating OBFF support must be enabled for OBFF. It is permitted for system firmware/software to selectively enable OBFF, but such enabling is beyond the scope of this specification.

To minimize power consumption, system firmware/software is strongly recommended to enable Message signaling of OBFF only when WAKE# signaling is not available for a given link.

OBFF signaling using WAKE# must only be reported as supported by all components connected to a Switch if it is a shared WAKE# signal. In these topologies it is permitted for software to enable OBFF for components connected to the Switch even if the Switch itself does not support OBFF.

It is permitted, although not encouraged, to indicate the same OBFF event more than once in succession.

When a Switch is propagating OBFF indications Downstream, it is strongly encouraged to propagate all OBFF indications. However, especially when using Messages, it may be necessary for the Switch to discard or collapse OBFF indications. It is permitted to discard and replace an earlier indication of a given type when an indication of the same or a different type is received.

Downstream Ports can be configured to transmit OBFF Messages in two ways, which are referred to as Variation A and Variation B. For Variation A, the Port must transmit the OBFF Message if the Link is in the L0 state, but discard the Message when the Link is in the Tx_L0s or L1 state. This variation is preferred when the Downstream Port leads to Devices that are expected to have communication requirements that are not time-critical, and where Devices are expected to signal a non-urgent need for attention by returning the Link state to L0. For Variation B, the Port must transmit the OBFF Message if the Link is in the L0 state, or, if the Link is in the Tx_L0s or L1 state, it must direct the Link to the L0 state and then transmit the OBFF Message. This variation is preferred when the Downstream Port leads to devices that can benefit from timely notification of the platform state.

When initially configured for OBFF operation, the initial assumed indication must be the CPU Active state, regardless of the logical value of the WAKE# signal, until the first transition is observed.

When enabling Ports for OBFF, it is recommended that all Upstream Ports be enabled before Downstream Ports, and Root Ports must be enabled after all other Ports have been enabled. For hot pluggable Ports this sequence will not generally be possible, and it is permissible to enable OBFF using WAKE# to an unconnected hot pluggable Downstream Port. It is recommended that unconnected hot pluggable Downstream Ports not be enabled for OBFF message transmission.

IMPLEMENTATION NOTE

OBFF Considerations for Endpoints

It is possible that during normal circumstances, events could legally occur that could cause an Endpoint to misinterpret transitions from an Idle window to a CPU Active window or OBFF window. For example, a non-OBFF Endpoint could assert WAKE# as a wakeup mechanism, masking the system's transitions of the signal. This could cause the Endpoint to behave in a manner that would be less than optimal for power or performance reasons, but should not be unrecoverable for the Endpoint or the host system.

In order to allow an Endpoint to maintain the most accurate possible view of the host state, it is recommended that the Endpoint place its internal state tracking logic in the CPU Active state when it receives a request that it determines to be host-initiated, and at any point where the Endpoint has a pending interrupt serviced by host software.

6.20 *PASID TLP Prefix*

The PASID TLP Prefix is an End-End TLP Prefix as defined in Section 2.2.1 . Layout of the PASID TLP Prefix is shown in Figure 6-20 and Table 6-14 .

When a PASID TLP Prefix is present, the PASID value in the prefix, in conjunction with the requester ID, identifies the Process Address Space ID associated with the Request. Each Function has a distinct set of PASID values. PASID values used by one Function are unrelated to PASID values used by any other Function.

A PASID TLP Prefix is permitted on:

- Memory Requests (including AtomicOp Requests) with Untranslated Addresses (see Section 2.2.4.1).
- Address Translation Requests, ATS Invalidation Messages, Page Request Messages, and PRG Response Messages (see Section 10.1.3).

  The PASID TLP Prefix is not permitted on any other TLP.

6.20.1 Managing PASID TLP Prefix Usage

Usage of PASID TLP Prefixes must be specifically enabled. Unless enabled, a component is not permitted to transmit a PASID TLP Prefix.

For Endpoint Functions (including Root Complex Integrated Devices), the following rules apply:

- A Function is not permitted to send and receive TLPs with a PASID TLP Prefix unless PASID Enable is Set (see Section 7.8.8.3):
- A Function must have a mechanism for dynamically associating use of a PASID with a particular Function context. This mechanism is device specific.
- A Function must have a mechanism to request that it gracefully stop using a specific PASID. This mechanism is device specific but must satisfy the following rules:
  - A Function may support a limited number of simultaneous PASID stop requests. Software should defer issuing new stop requests until older stop requests have completed.
  - A stop request in one Function must not affect operation of any other Function.
  - A stop request must not affect operation of any other PASID within the Function.
  - A stop request must not affect operation of transactions that are not associated with a PASID.
  - When the stop request mechanism indicates completion, the Function has:
    - Stopped queuing new Requests for this PASID.
    - Completed all Non-Posted Requests associated with this PASID.
    - Flushed to the host all Posted Requests addressing host memory in all TCs that were used by the PASID. The mechanism used for this is device specific (for example: a non-relaxed Posted Write to host memory or a processor read of the Function can flush TC0; a zero length read to host memory can flush non-zero TCs).
    - Optionally flushed all Peer-to-Peer Posted Requests to their destination(s). The mechanism used for this is device specific.
    - Complied with additional rules described in Address Translation Services (Chapter 10) if Address Translations or Page Requests were issued on the behalf of this PASID.

For Root Complexes, the following rules apply:

- A Root Complex must have a device specific mechanism for indicating support for PASID TLP Prefixes.
- A Root Complex that supports PASID TLP Prefixes must have a device specific mechanism for enabling them. By default usage of PASID TLP Prefixes is disabled.
- A Root Complex that supports PASID TLP Prefixes may optionally have a device specific mechanism for enabling them at a finer granularity than the entire Root Complex (e.g., distinct enables for a specific Root Port, Requester ID, Bus Number, Requester ID, or Requester ID/PASID combination).

6.20.2 PASID TLP Layout

A TLP may contain at most one PASID TLP Prefix.

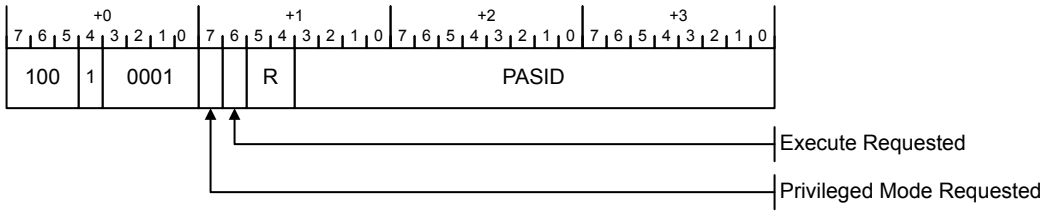

Figure 6-20 PASID TLP Prefix

Table 6-14 PASID TLP Prefix

Bits	Description
Byte 0 bits 7:5	100b - indicating TLP Prefix
Byte 0 bit 4	1b - indicating End-End TLP Prefix
Byte 0 bits 3:0	0001b - indicating PASID TLP Prefix
Byte 1 bit 7	***Privileged Mode Requested*** - If Set indicates a Privileged Mode entity in the Endpoint is issuing the Request, If Clear, indicates a Non-Privileged Mode entity in the Endpoint is issuing the Request. Usage of this bit is specified in Section 6.20.2.3 .
Byte 1 bit 6	***Execute Requested*** - If Set indicates the Endpoint is requesting Execute Permission. If Clear, indicates the Endpoint is not requesting Execute Permission. Usage of this bit is specified in Section 6.20.2.2 .
Byte 1 bit 3: Byte3 bit 0	**Process Address Space ID (PASID)** - This field contains the PASID value associated with the TLP. Usage of this field is defined in Section 6.20.2.1 .

6.20.2.1 PASID field

The PASID field identifies the user process associated with a Request. This field is present in all PASID TLP Prefixes.

The PASID field is 20 bits wide. Endpoints and Root Complexes need not support the entire range of the field. For Endpoints, the Max PASID Width field indicates the supported range of PASID values (Section 7.28.2). For Root Complexes, an implementation specific mechanism is used to provide this information.

Endpoints are not permitted to send TLPs with a PASID TLP Prefix unless the PASID Enable bit (Section 7.28.3) is Set. Endpoints that support the PASID TLP Prefix must signal Unsupported Request (UR) when they receive a TLP with a PASID TLP Prefix and the PASID Enable bit is Clear.

Root Complexes may optionally support TLPs with PASID TLP Prefixes. The mechanism used to detect whether a Root Complex supports the PASID TLP Prefix is implementation specific.

For Endpoints, the following rules apply:

- The Endpoint is not permitted to send TLPs with a PASID value greater than or equal to $2^{\text{Max PASID Width}}$.
- The Endpoint is optionally permitted to signal an error when it receives a Request with a PASID value greater than or equal to $2^{\text{Max PASID Width}}$. This is an Unsupported Request error associated with the Receiving Port (see Section 6.2).

For Root Complexes, the following rules apply:

- A Root Complex is not permitted to send a TLP with a PASID value greater than it supports.
- A Root Complex is optionally permitted to signal an error when it receives a Request with a PASID value greater than it supports. This is an Unsupported Request error associated with the Receiving Port (see Section 6.2).

  For Completers, the following rules apply:

- For Untranslated Memory Requests, the PASID value and the Untranslated Address are both used in determining the Translated Address used in satisfying the Request.
  For address translation related TLPs, usage of this field is defined in Address Translation Services (Chapter 10).

IMPLEMENTATION NOTE
PASID Width Homogeneity

The PASID value is unique per Function and thus the original intent was that the width of the PASID value supported by that Function could be based on the needs of that Function. However, current system software typically does not follow that model and instead uses the same PASID value in all Functions that access a specific address space. To enable this, system software will typically ensure a common system PASID width for Root Complex and persistent translation agents. Such system software will typically disable ATS on any hot plugged Endpoint Functions or translation agents reporting PASID width support which is less than that of the common system PASID width.

The Root Complex, Endpoints, and translation agents, are often implemented independently of system software, therefore it is highly recommended that hardware implement the maximum width of 20 bits to ensure interoperability with system software.

Endpoints may, in an implementation-specific way, be able to map the 20 bit system PASID to an internal representation carrying a smaller width. If this is done, it is critical that the Endpoint do so without impacting system software, which has no mechanism to differentiate such implementation from those that implement the full 20 bit width natively.

6.20.2.2 Execute Requested

If the Execute Requested bit is Set, the Endpoint is requesting permission for the Endpoint to Execute instructions in the memory range associated with this request. The meaning of Execute permission is outside the scope of this specification.

Endpoints are not permitted to send TLPs with the Execute Requested bit Set unless the Execute Permission Supported bit (Section 7.8.8.2) and the Execute Permission Enable bit (Section 7.8.8.3) are both Set.

For Root Complexes, the following rules apply:

- Support for Execute Requested by the Root Complex is optional. The mechanism used to determine whether a Root Complex supports Execute Requested is implementation specific.
- A Root Complex that supports the Execute Requested bit should have an implementation specific mechanism to enable it to use the bit.
- A Root Complex that supports the Execute Requested bit may have an implementation specific mechanism to enable use of the bit at a finer granularity (e.g., for a specific Root Port, for a specific Bus Number, for a specific Requester ID, or for a specific Requester ID/PASID combination), and its default value is implementation specific.
  For Completers, the following rules apply:
- Completers have a concept of an effective value of the bit. For a given Request, if the Execute Requested bit is supported and it usage is enabled for the Request, the effective value of the bit is the value in the Request; otherwise the effective value of the bit is 0b.
- For Untranslated Memory Read Requests, Completers use the effective value of the bit as part of the protection check. If this protection check fails, Completers treat the Request as if the memory was not mapped.
- For Untranslated Memory Requests, other than an Untranslated Memory Read Request, the bit is Reserved.
  For address translation related TLPs, usage of this bit is defined in Address Translation Services (Chapter 10).

6.20.2.3 Privileged Mode Requested

If Privileged Mode Requested is Set, the Endpoint is issuing a Request that targets memory associated with Privileged Mode. If Privileged Mode Requested is Clear, the Endpoint is issuing a Request that targets memory associated with Non-Privileged Mode.

The meaning of Privileged Mode and Non-Privileged Mode and what it means for an Endpoint to be operating in Privileged or Non-Privileged Mode depends on the protection model of the system and is outside the scope of this specification.

Endpoints are not permitted to send a TLP with the Privileged Mode Requested bit Set unless both the Privileged Mode Supported bit (Section 7.8.8.2) and the Privileged Mode Enable bit (Section 7.8.8.3) are Set.

For Root Complexes, the following rules apply:

- Support for the Privileged Mode Requested bit by the Root Complex is optional. The mechanism used to determine whether a Root Complex supports the Privileged Mode Requested bit is implementation specific.
- A Root Complex that supports the Privileged Mode Requested bit should have an implementation specific mechanism to enable it to use the bit.
- A Root Complex that supports the Privileged Mode Requested bit may have an implementation specific mechanism to enable use of the bit at a finer granularity (e.g., for a specific Root Port, for a specific Bus Number, for a specific Requester ID, or for a specific Requester ID/PASID combination).

  For Completers, the following rules apply:

- Completers have the concept of an effective value of the bit. For a given Request, if the Privileged Mode Requested bit is supported and its usage is enabled for the Request, the effective value of the bit is the value in the Request; otherwise the effective value of the bit is the 0b.
- For Untranslated Memory Requests, Completers use the effective value of the bit as part of its protection check. If this protection check fails, Completers treat the Request as if the memory was not mapped.
- For address translation related TLPs, usage of this bit is defined in Address Translation Services (Chapter 10).

6.21 Lightweight Notification (LN) Protocol

Lightweight Notification (LN) protocol enables Endpoints to register interest in specific cachelines in host memory, and be notified via a hardware mechanism when they are updated. An LN Requester (LNR) is a client subsystem in an Endpoint that sends LN Read/Write Requests and receives LN Messages. An LN Completer (LNC) is a service subsystem in the host that receives LN Read/Write Requests, and sends LN Messages when registered cachelines are updated.

LN protocol provides a notification service for when cachelines of interest are updated. Most commonly, an LNR sends an LN Read to a Memory Space range that has an associated LNC, requesting a copy of a cacheline (abbreviated “line” in this section). The LNC returns the requested line to the LNR and records that the LNR has requested notification when that line is updated; that is, the LNC “registers” the line. Later, the LNC notifies the LNR via an LN Message when some entity updates the line, so the LNR can take appropriate action. A similar notification service exists for LN Writes, where an LNR writing a line can be notified later when the line is updated.

LN protocol permits multiple LNRs each to register the same line concurrently. An Endpoint with LNR is permitted to write to a line at any time, regardless of whether the LNR has registered the line.

A typical system consists of host processors, host memory, a host internal fabric, Root Ports, Switches, and Endpoints. Figure 6-21 below illustrates a simplified view of the elements and how they are interconnected to provide a context for describing how LN protocol operates.

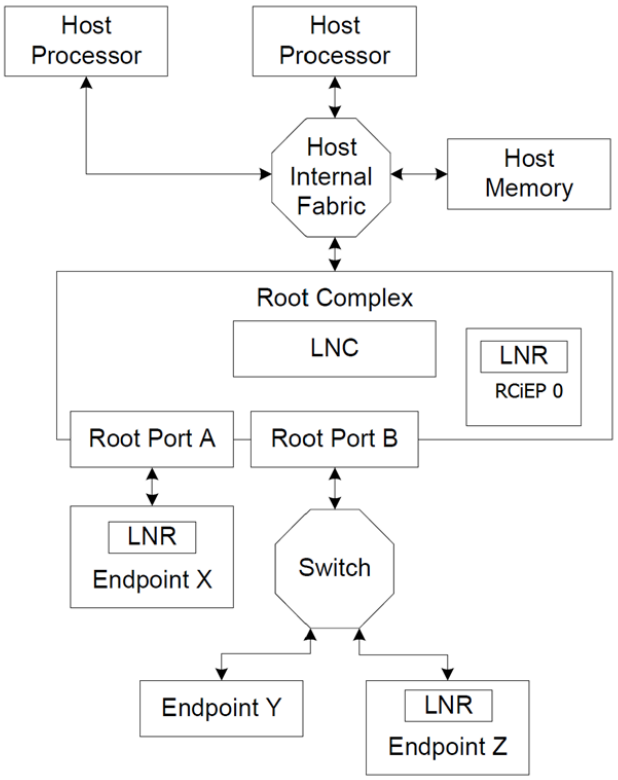

Figure 6-21 Sample LN System Block Diagram

In the figure above, Endpoint X, Endpoint Z, and RCIEP 0 each contain an LNR. The Root Complex contains an LNC.

6.21.1 LN Protocol Operation

LN is a simple protocol that satisfies several key use models with minimum complexity and cost.

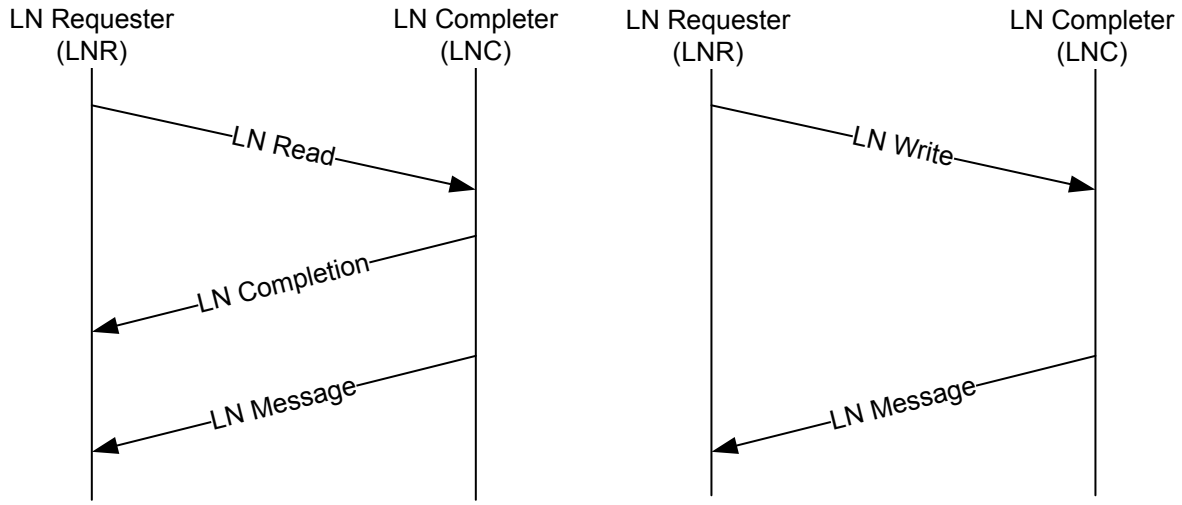

Figure 6-22 LN Protocol Basic Operation

LN protocol operation with a single LNR is shown in the diagram above. For the case of reads, as shown on the left: (1) an LNR requests a copy of a line from host memory using an LN Read; (2) the LNC returns the line in an LN Completion and records that the LNR has registered the line; and (3) the LNC later notifies the LNR using an LN Message when the registered line has been updated. For the case of writes, as shown on the right: (1) an LNR writes to a line to host memory using an LN Write; (2) the LNC records that the LNR has registered the line; and (3) the LNC later notifies the LNR using an LN Message when the registered line has been updated.

An LNC must send an LN Message either if a registered line is updated by some entity (e.g., a CPU or a device) or if the LNC is no longer going to track the state of that line. The latter case is referred to as an eviction, and is indicated by the Notification Reason (NR) field in the LN Message.

If an LN Requester does an LN Read or LN Write to a line that it already has registered, then the LN Requester generally can't determine if a subsequent LN Message it receives for that line was for the most recent LN Read/Write Request or for a previous one.

LN protocol permits multiple LNRs to register the same line concurrently. In this case the LNC notifies the multiple LNRs either by sending a directed LN Message to each LNR, or by sending a broadcast LN Message to each affected Hierarchy Domain.

Once the LNC updates or evicts a given line and sends one or more LN Messages to notify all LNRs with associated registrations, the LNC will not send additional LN Messages for that specific line unless it receives a new LN Read or LN Write to that line.

An LNC is permitted to have an implementation-specific limit on how many LNRs it can independently track for each given line. At or below this limit, the LNC generally uses directed LN Messages for notifications. Above this limit, the LNC uses broadcast LN Messages. An implementation is permitted to have a limit of zero, and use broadcast LN Messages exclusively.

A single LN Message with a specific NR field value can indicate that all lines registered to that LNR have been evicted. Both directed and broadcast versions of this "all evicted" LN Message are permitted.

An LNC is permitted to have an implementation-specific limitations on how many lines or sets of lines it can register concurrently. If the LNC receives a Request to register a new line, but the LNC has insufficient resources to do so, the LNC

is permitted to evict one or more old lines in order to free up the necessary resources, or send a LN Message indicating that the new line was evicted.

IMPLEMENTATION NOTE
Excessive Use of Broadcast LN Messages

In order to avoid performance issues, LNCs that use broadcast LN Messages should be implemented to minimize the number of Hierarchy Domains each broadcast LN Message is sent to, and also to keep the rate of broadcast LN Messages within reasonable bounds. Each broadcast LN Message consumes Link bandwidth, and some Endpoints may process broadcast LN Messages at a relatively low rate.

In particular, Endpoints that do not support LN Requester capability may handle received broadcast LN Messages as an exception case using a low performance mechanism, e.g., decoding the Message with device firmware instead of in hardware. Each Message could conceivably take microseconds to process[129] , and an excessive rate of them could easily cause back-pressure of Posted Requests within the fabric, causing severe performance problems.

While Endpoints that do not support LN Requester capability may also handle directed LN Messages as an exception case using a low performance mechanism, this should not cause performance problems. With LN protocol, LNCs send directed LN Messages only to Endpoints that support LN Requester capability, and presumably those Endpoints will be able to process both directed and broadcast LN Messages at a rate that avoids performance issues.

6.21.2 LN Registration Management

Since LNCs have limited resources for registering cachelines, an LNR Capability structure in each LNR provides mechanisms for software to limit the LNR's maximum number of outstanding registrations. Software reads the LNR Registration Max field to discover the maximum number an LNR is capable of having outstanding. If necessary, software can set the LNR Registration Limit field to impose a specified limit.

LNC registration resource limitations may be highly implementation specific, perhaps including factors such as set associativity, maximum number of sets, and distinct sets of resources for different regions of host memory. How software discovers these resource limitations is outside the scope of this specification.

To manage its number of outstanding registrations, an LNR can deregister outstanding registrations by sending a zero-length LN Write to each line it wants to deregister.

The LNC is not required to accept registrations for all locations in host memory. However, the granularity for memory regions accepting registrations is required to be no finer than aligned 4KB regions. To determine if a given aligned 4KB region accepts registrations, an LNR can perform an LN Read to any cacheline within the region, and see if an LN Completion is returned. If an LNR wishes to "probe" a region for registration capability without actually creating a registration, the LNR can use a zero-length LN Read, which is required to provide this semantic.

6.21.3 LN Ordering Considerations

LN Reads have the same ordering requirements as other Memory Read Requests, and LN Writes have the same ordering requirements as other Memory Write Requests. LN Completions (for LN Reads) have the same ordering requirements as Completions for other Memory Reads. LN Messages have the same ordering requirements as other Posted Requests.

129. The Posted Request Acceptance Limit permits an Endpoint to take up to 10 µs to process each received Posted Request. See Section 2.3.1 .

For a given line, when an LN Completer receives an LN Read followed by an update to that line, the LN Completer is permitted to send the LN Completion and the LN Message in either order.

For a given line, when an LN Completer receives an LN Read that triggers an eviction LN Message, the LN Completer is permitted to send the LN Completion and the LN Message in either order.

For a given line, when an LN Completer receives an LN Write that triggers an eviction LN Message, followed by an update to that line, the LN Completer is permitted to send the two LN Messages in either order.

For different lines, an LN Completer is permitted to send LN Messages in any order.

6.21.4 LN Software Configuration

LN protocol supports 2 cache line sizes (CLSs) - 64 bytes and 128 bytes. Support for each CLS is optional, both for Root Complexes and Endpoints. The CLS in effect for the system is determined by the host, and indicated by the LN System CLS field in applicable Root Ports and RCRBs. All Root Ports and RCRBs that indicate LN Completer support must indicate the same CLS; otherwise, the results are undefined. The host must not change the system CLS while any operating system is running; otherwise, the results are undefined.

Endpoints supporting LN protocol must support one or both CLSs, and indicate this via the LNR-64 Supported and LNR-128 Supported capability bits. When enabling each LN Requester, software must ensure that the associated LNR CLS control bit is configured to match the system CLS; otherwise, the results are undefined. An LN Requester that supports only one CLS is permitted to hardwire its LNR CLS control bit to the corresponding value.

Software must not change the value of the LNR CLS control bit or the LNR Registration Limit field unless its LNR Enable control bit is already Clear or is being Cleared with the same Configuration Write; otherwise, the results are undefined. If the LNR Enable bit is already Clear, software is permitted to change the values of the LNR CLS bit and LNR Registration Limit field concurrently with Setting the LNR Enable bit, using a single Configuration Write.

Software is permitted to Clear the LNR Enable bit at any time. When the LNR Enable bit is Clear, the LNR must clear all its internal registration state.

6.21.5 LN Protocol Summary

Detailed rules and requirements for LN protocol are distributed throughout the rest of this specification, but here is a general summary plus some unique requirements.

- An LN Read is a Memory Read Request whose TLP Header has the LN bit Set. An LN Completion is a Completion whose TLP Header has the LN bit Set. An LN Write is a Memory Write Request whose TLP Header has the LN bit Set.
- All requirements for Memory Read Requests apply to LN Reads unless explicitly stated otherwise.
  - An LN Read must access no more than a single cacheline, as determined by the system CLS. An LN Completer must handle a violation of this rule as a Completer Abort unless the Completer detects a higher precedence error. Partial cacheline LN Reads, including zero-length LN Reads, are permitted.
  - If an LN Completer handles an LN Read as an Uncorrectable Error or an Advisory Non-Fatal Error, the LN Completer must not register notification service for that Request.
  - An LN Completer must handle a zero-length LN Read as a “probe” of the targeted memory region for registration capability. See Section 6.20.2 . If the Completion Status is Successful Completion, the LN bit in the TLP Header must indicate if the region supports registration capability, but the LNC must not create a registration for this case. LN Completers must support registration capability with a granularity no finer than aligned 4KB regions.

    - Ordering and Flow Control rules for LN Reads are identical to those for Memory Read Requests.
- All requirements for Memory Read Completions apply to LN Completions unless explicitly stated otherwise.
    - The Completion for an LN Read must be an LN Completion (i.e., have the LN bit Set in its TLP Header) if the Completer is an LN Completer, the targeted memory region accepts registrations, and the Completion Status is Successful Completion; otherwise the Completion must have the LN bit Clear in its TLP Header. Note that a poisoned Completion will have a Completion Status of Successful Completion. See Section 2.7.2.2 .
    - Ordering and Flow Control rules for LN Completions are identical to Completions whose TLP Header has the LN bit Clear.
- All requirements for Memory Write Requests apply to LN Writes unless explicitly stated otherwise.
    - An LN Write must access no more than a single cacheline, as determined by the system CLS. An LN Completer must handle a violation of this rule as a Completer Abort unless the Completer detects a higher precedence error.
    - If an LN Completer handles an LN Write as an Uncorrectable Error, the LN Completer must not register notification service for that Request. Note that depending upon its configuration, a Completer may handle a poisoned LN Write as an Uncorrectable Error, an Advisory Non-Fatal Error, or a masked error.
    - An LN Completer must handle a zero-length LN Write as a request to deregister an existing registration. See Section 6.20.2 . If the cacheline targeted by a zero-length LN Write was not previously registered, it must remain unregistered.
    - Ordering, Flow Control, and Data Poisoning rules for LN Writes are identical to those for Memory Write Requests.
    - A Requester must not generate LN Writes for MSI or MSI-X interrupts. An LN Completer must handle an LN Write targeting an interrupt address range as a Completer Abort unless the Completer detects a higher precedence error.
- For LN Reads and LN Writes, the address must be the proper type, as indicated by the Address Type (AT) field. See Section 2.2.4.1 . The proper type depends upon whether a Translation Agent (TA) is being used. See Chapter 10 .
    - If a TA is being used, the address must be a Translated address. An LN Requester must support ATS in order to acquire and use Translated addresses.
    - If a TA is not being used, the address must be a Default/Untranslated Address.
    - An LN Completer detecting a violation of the above rules must handle the Request as a Completer Abort (CA), unless a higher precedence error is detected.

6.22 Precision Time Measurement (PTM) Mechanism

6.22.1 Introduction

Precision Time Measurement (PTM) enables precise coordination of events across multiple components with independent local time clocks. Ordinarily, such precise coordination would be difficult given that individual time clocks have differing notions of the value and rate of change of time. To work around this limitation, PTM enables components to calculate the relationship between their local times and a shared PTM Master Time: an independent time domain associated with a PTM Root.

Enhanced Precision Time Management (ePTM) places additional requirements on PTM Devices. Support for ePTM is indicated by the ePTM Capable bit.

PTM defines the following:

- PTM Requester - A Function capable of using PTM as a consumer associated with an Endpoint or an Upstream Port.
- PTM Responder - A Function capable of using PTM to supply PTM Master Time associated with a Port or an RCRB.
- Time Source - A local clock associated with a PTM Responder.
- PTM Root - The source of PTM Master Time for a PTM Hierarchy. A PTM Root must also be a Time Source and is typically also a PTM Responder.

Each PTM Root supplies a single PTM Master Time to all of the PTM Hierarchy: a set of PTM Requesters associated with a single PTM Root.

Figure 6-23 illustrates some example system topologies using PTM. These are only illustrative examples, and are not intended to imply any limits or requirements.

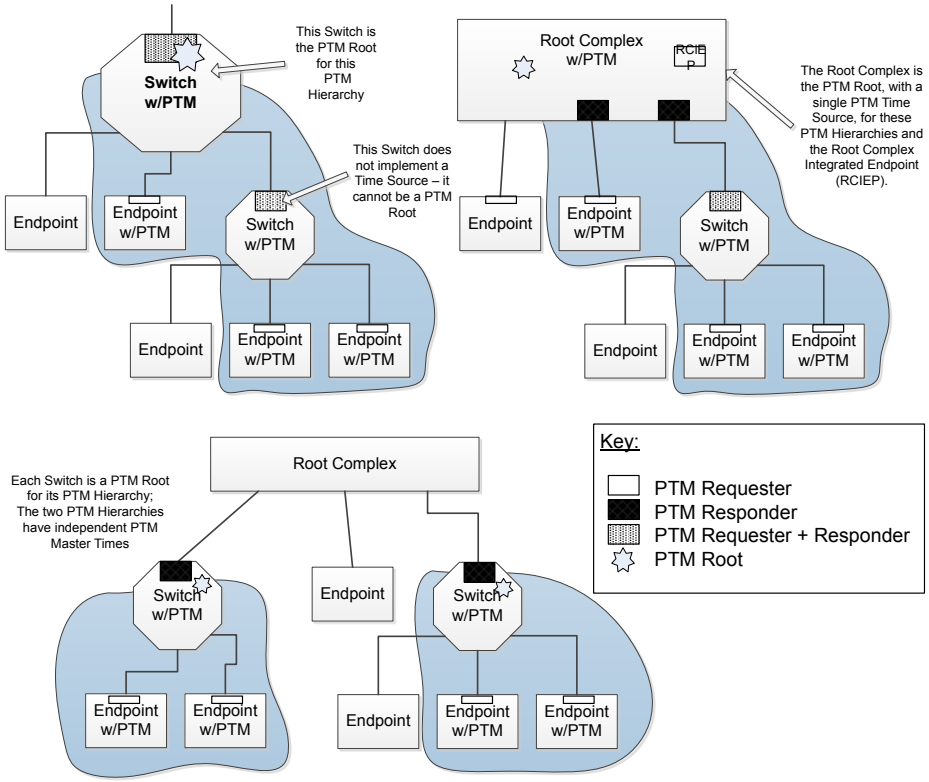

Figure 6-23 Example System Topologies using PTM

IMPLEMENTATION NOTE

PTM and Retimers

PCIe Retimers can impact PTM accuracy by introducing asymmetric link delays. Retimers designed to maintain symmetric link delays will enable the best PTM accuracy. The larger and more variable the asymmetry, the greater the impact to PTM. Consult the manufacturer's documentation to determine the suitability of a Retimer implementation for use with PTM.

6.22.2 PTM Link Protocol

When using PTM between two components on a Link, the Upstream Port, which acts on behalf of the PTM Requester, sends PTM Requests to the Downstream Port on the same Link, which acts on behalf of the PTM Responder.

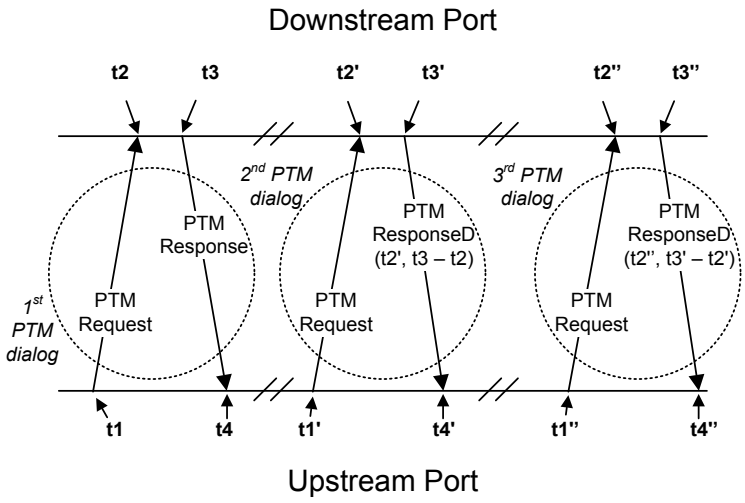

Figure 6-24 Precision Time Measurement Link Protocol

Figure 6-24 illustrates the PTM link protocol. The points t1, t2, t3, and t4 in the above diagram represent timestamps captured locally by each Port as they transmit and receive PTM Messages. The component associated with each Port stores these timestamps from the 1st PTM dialog in internal registers for use in the 2nd PTM dialog, and so on for subsequent PTM dialogs.

The Upstream Port, on behalf of the PTM Requester, initiates the PTM dialog by transmitting a PTM Request message.

The Downstream Port, on behalf of the PTM Responder, has knowledge of or access (directly or indirectly) to the PTM Master Time.

During each dialog, the Downstream Port populates the PTM ResponseD message based on timestamps stored during previous PTM dialogs, as defined in Section 6.22.3.2 .

Once each component has historical timestamps from the preceding dialog, the component associated with the Upstream Port can combine its timestamps with those passed in the PTM ResponseD message to calculate the PTM Master Time using the following formula:

$$\text{PMT Master Time at } t1' = \frac{\left((t4 - t1) - (t3 - t2)\right)}{2}$$

Equation 6-2 PTM Master Time

The values t1, t2, t3, t4, and t2' indicate the timestamps captured during the PTM dialog as illustrated in Figure 6-24 .

PTM capable components would typically record the results of these timestamp calculations, and may make them available to software via implementation specific means. Herein, this document refers to this resultant timing information as the component's "PTM context".

For a Switch implementing PTM, the time synchronization mechanism(s) within the Switch itself are implementation-specific.

IMPLEMENTATION NOTE
PTM Theory and Operation

The timestamps captured during the PTM dialogs enable the calculation of the timing relationship between the PTM Requester and PTM Responder. The value (t3-t2) measures the time consumed by the PTM Responder for a given PTM dialog. The time (t4-t1) is the time from request to response. Therefore ((t4 - t1) - (t3 - t2)) effectively gives the round trip message transit time between the two components, and that quantity divided by 2 approximates the Link delay - the time difference between t1 and t2. It is assumed that the Link transit times from PTM Requester to PTM Responder and back again are symmetric, which is typically a good assumption (see also the Implementation Note on PTM Timestamp Capture Mechanisms).

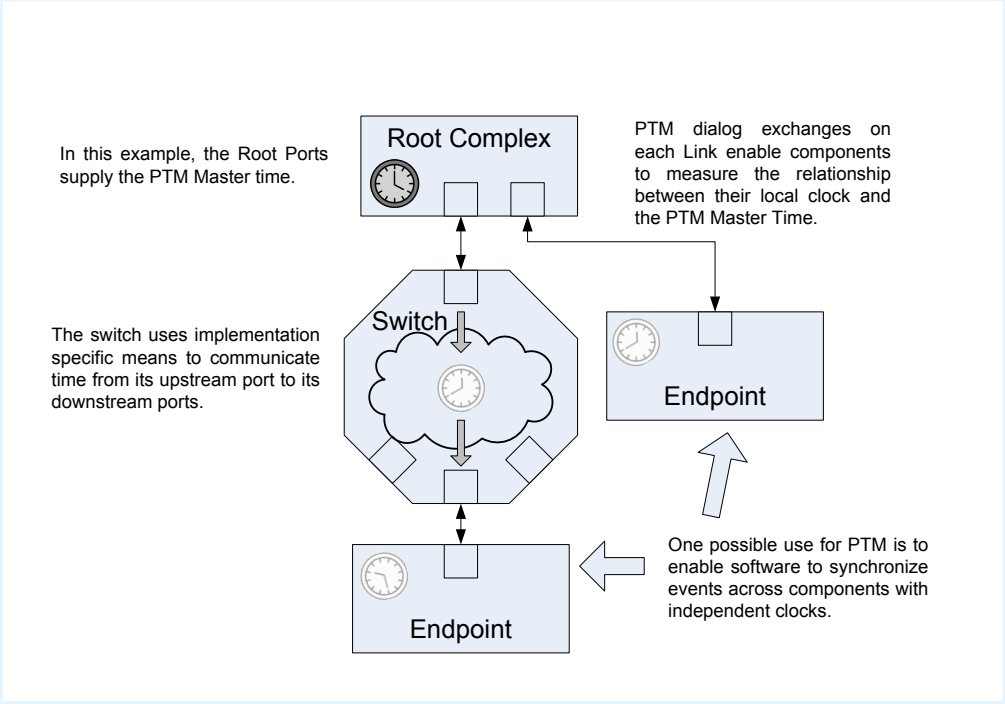

Figure 6-25 Precision Time Measurement Example

Figure 6-25 illustrates a simple device hierarchy employing PTM. Each Upstream Port initiates PTM dialogs to establish the relationship between its local time and the PTM Master Time provided by the Root Port.

In this example, the Switch initiates PTM dialogs on its Upstream Port to obtain the PTM Master Time for use in fulfilling PTM Request Message received at its Downstream Ports. This Switch employs implementation specific means to communicate the PTM Master Time from its Upstream Port to its Downstream Ports.

PTM capable components can make their PTM context available for inspection by software, enabling software to translate timing information between local times and PTM Master Time. In turn, this capability enables software to coordinate events across multiple components with very fine precision.

Similarly, it is strongly recommended that platforms implementing PTM also make the PTM Master Time available to software.

6.22.3 Configuration and Operational Requirements

Software must not have the PTM Enable bit Set in the PTM Control register on a Function associated with an Upstream Port unless the associated Downstream Port on the Link already has the PTM Enable bit Set in its associated PTM Control register.

PTM support by a Function is indicated by the presence of a PTM Extended Capability structure. It is not required that all Endpoints in a hierarchy support PTM, and it is not required for software to enable PTM in all Endpoints that do support it.

If a PTM Message is received by a Port that does not support PTM, or by a Downstream Port when the PTM Enable bit is clear, the Message must be treated as an Unsupported Request. This is a reported error associated with the Receiving Port (see Section 6.2). A Properly formed PTM Response received by an Upstream Ports that support PTM, but for which the PTM Enable bit is clear, must be silently discarded.

As observed through PTM, the PTM Master Time must satisfy the following behavioral requirements:

- Time values must be monotonic, and strictly increasing.
- The perceived granularity must be no greater than the value reported in the Local Clock Granularity field of the PTM Capability register.
- The perceived time must start no later than when the PTM Root processes its first PTM Request Message.

Referring to Figure 6-24 , the following rules define timestamp capture:

- A PTM Requester must update its stored t1 timestamp when transmitting a PTM Request Message, even if that transmission is a replay.
- A PTM Responder must update its stored t2 timestamp when receiving a PTM Request Message, even if received TLP is a duplicate.
- A PTM Responder must update its stored t3 timestamp when transmitting a PTM Response or ResponseD Message, even if that transmission is a replay.
- A PTM Requester must update its stored t4 timestamp when receiving a PTM Response Message, even if received TLP is a duplicate.
    - Timestamps must be based on the STP Symbol or Token that frames the TLP, as if observing the first bit of that Symbol or Token at the Port's pins. Typically this will require an implementation specific adjustment to compensate for the inability to directly measure the time at the actual pins, as the time will commonly be measured at some internal point in the Rx or Tx path. The accuracy and consistency of this measurement are not bounded by this specification, but it is strongly recommended that the highest practical level of accuracy and consistency be achieved.

6.22.3.1 PTM Requester Role

- Support for the PTM Requester role is indicated by setting the PTM Requester Capable bit in the PTM Capability register.
- PTM Requesters are permitted to request PTM Master Time only when PTM is enabled. The mechanism for directing a PTM Requester to issue such a request is implementation specific.
    - Upstream Ports obtain PTM Master Time via PTM dialogs as described in Section 2.2.8.10 .
    - The mechanism by which RCiEPs request PTM Master Time is implementation specific.

- Once having issued a PTM Request Message, the Upstream Port must not issue another PTM Request Message prior to the receipt of a PTM Response Message, PTM ResponseD Message, Reset, or the passage of 100 µs without a corresponding PTM Message from the Downstream Port.
- Upon receiving a PTM Response, the Upstream Port must wait at least 1 µs before issuing another PTM Request Message.
- For Multi-Function Devices (MFDs) containing multiple PTM Requesters, the Upstream Port associated with that MFD must issue a single PTM dialog during each PTM context refresh. PTM Requesters within the MFD maintain their individual PTM contexts using this one, Device-wide PTM dialog. The mechanism for refreshing multiple PTM contexts from one PTM dialog is implementation specific.
- It is strongly recommended that an Upstream Port invalidate its internal PTM context when any of the following occur. If ePTM is supported, then an Upstream Port must invalidate its internal PTM context when any of the following occur:
  - A PTM Request is replayed.
  - A duplicate PTM ResponseD TLP is received.
  - The relationship between PTM Master Time and the Upstream Port's local time changes, as determined by implementation specific criteria. For example, this may occur as a result of a transition to a non-D0 state or due to accumulated PPM drift.

  These events are grouped under the label "Local Time Invalidation Event" in Figure 6-26 .
- If ePTM is supported, an Upstream Port, upon replaying a PTM TLP, must invalidate it's PTM context until two successive PTM dialogs have been completed successfully and without replays.

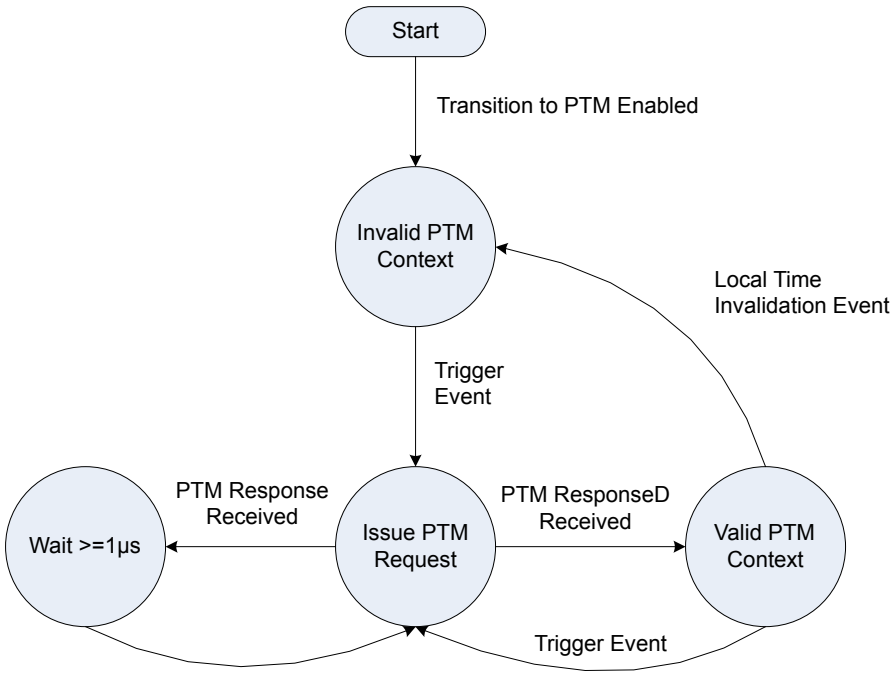

Figure 6-26 PTM Requester Operation

IMPLEMENTATION NOTE

PTM Invalidation on the Reception of duplicate TLPs

Duplicate TLPs are detected and discarded in the Data Link Layer, whereas PTM messages are identified in the Transaction Layer. In some implementations it may be difficult or excessively complicated to distinguish a duplicate PTM TLP from other duplicate TLPs.

Because Upstream Ports are permitted to invalidate their internal PTM context for implementation-specific criteria, a PTM Requester is allowed to invalidate its internal PTM context upon the reception of any duplicate TLP in addition to any duplicate PTM TLP. Similarly, if ePTM is supported, then a PTM Responder is allowed to invalidate its historical timestamps (t2 - t3) upon the reception of any duplicate TLP.

6.22.3.2 PTM Responder Role

- Support for the PTM Responder role is indicated by setting the PTM Responder Capable bit in the PTM Capability register.
- Switches and Root Complexes are permitted to implement the PTM Responder Role.
  - A PTM capable Switch, when enabled for PTM by setting the PTM Enable bit in the PTM Control register associated with the Switch Upstream Port, must respond to all PTM Request Messages received at any of its Downstream Ports.
  - The mechanism by which Root Complexes communicate PTM Master Time to RCiEPs is implementation specific.
- PTM Responders must populate PTM ResponseD Messages as follows (refer to Figure 6-24 and the accompanying implementation note):
  - The PTM Master Time field is a 64-bit value containing the value of PTM Master Time at the receipt of the PTM Request Message for the current PTM Dialog. In Figure 6-24 , for the 2nd PTM dialog, this is the PTM Master Time at time t2'.
  - The Propagation Delay field is a 32-bit value containing the interval between the receipt of the PTM Request Message and the transmission of the PTM Response Message for the previous PTM dialog. In Figure 6-24 , for the 2nd PTM dialog, this is the time interval between t2 and t3 captured during the 1st PTM dialog.
  - The unit of measurement for both fields is one ns.
  - A PTM Responder with multiple Downstream Ports must populate all PTM ResponseD Messages with values from a single PTM Root across all its PTM Ports Downstream ports.
- Switch Downstream Ports and Root Ports acting as PTM Responders must respond to each PTM Request Message received at their Downstream Ports with either PTM Response or PTM ResponseD according to the following rules:
  - A PTM Responder must not send a PTM Response or PTM ResponseD Message without first receiving a PTM Request Message.
  - Upon receipt of a PTM Request Message, a PTM Responder must attempt to issue a PTM Response or PTM ResponseD Message within 10 µs.
  - A PTM Responder must issue PTM Response when the Downstream Port does not have valid historical timestamps (t3 - t2) with which to fulfill a PTM Request Message.

    - If ePTM is supported, a PTM Responder must invalidate its historical timestamps (t3 - t2) immediately upon replaying any PTM Response or PTM ResponseD. A PTM Responder must invalidate its historical timestamps (t3 - t2) after receiving any duplicate PTM Request.
- A PTM Responder must issue PTM ResponseD when it has stored copies of the values required to populate the PTM ResponseD Message: historical timestamps (t3 - t2) and the PTM Master Time at the receipt of the most recent PTM Request Message (time t2').
- A PTM Responder is permitted to issue PTM Response when it has stored copies of the historical timestamps (t3 - t2) but must request the PTM Master Time from elsewhere. In this case, it is permitted to issue PTM Response messages in response to PTM Request Messages while it retrieves the PTM Master Time if that retrieval is expected to take more than 10 µs.
- The perceived granularity of the historical timestamps and PTM Master Time values transmitted by a PTM Responder must not exceed that reported in the Local Clock Granularity field of the PTM Capability register.

6.22.3.3 PTM Time Source Role - Rules Specific to Switches

In addition to the requirements listed above for the PTM Requester and PTM Responder Roles, Switches must follow these requirements:

- When the Upstream Port is associated with a Multi-Function Device, only a single Function associated with that Upstream Port is permitted to implement the PTM Extended Capability structure. For Switches, all PTM functionality associated with the Switch must be controlled through that structure. It is not required that the Function implementing the PTM Extended Capability structure be the Switch Upstream Port Function.
- The PTM Extended Capability structure for a Switch must indicate support for both the PTM Requester and PTM Responder roles.
- The PTM Extended Capability in the Upstream Port controls all Switches in that Upstream Port.
- A Switch is permitted to act as a PTM Root, or to issue PTM Requests on its Upstream Port to obtain the PTM Master Time for use in fulfilling PTM Requests received at its Downstream Ports. In the latter case the Switch must account for any internal delays within the Switch.
- A Switch is permitted to maintain a local PTM context for use in fulfilling PTM Requests received on its Downstream Ports.
- A Switch which is not acting as a PTM Root must invalidate its local context no more than 10 ms from the last PTM dialog on its Upstream Port. The Switch must then refresh its local PTM context prior to issuing further PTM ResponseD Messages on its Downstream Ports. This requirement for periodic refreshes is optional if it is guaranteed by implementation-specific means that the Switch's local clock is phase locked with PTM Master Time.
- Any Switch implementing a local clock for the purpose of maintaining a local PTM context must report the granularity of this clock as defined in the PTM Capabilities structure (Section 7.9.16).

IMPLEMENTATION NOTE

PTM Timestamp Capture Mechanisms

PTM uses services from both the Data Link and Transaction Layers. Accuracy requires that time measurements be taken as close to the Physical Layer as possible. Conversely, the messaging protocol itself properly belongs to the Transaction Layer. The PTM message protocol applies to a single Link, where the Upstream Port is the requester and the Downstream Port is the responder.

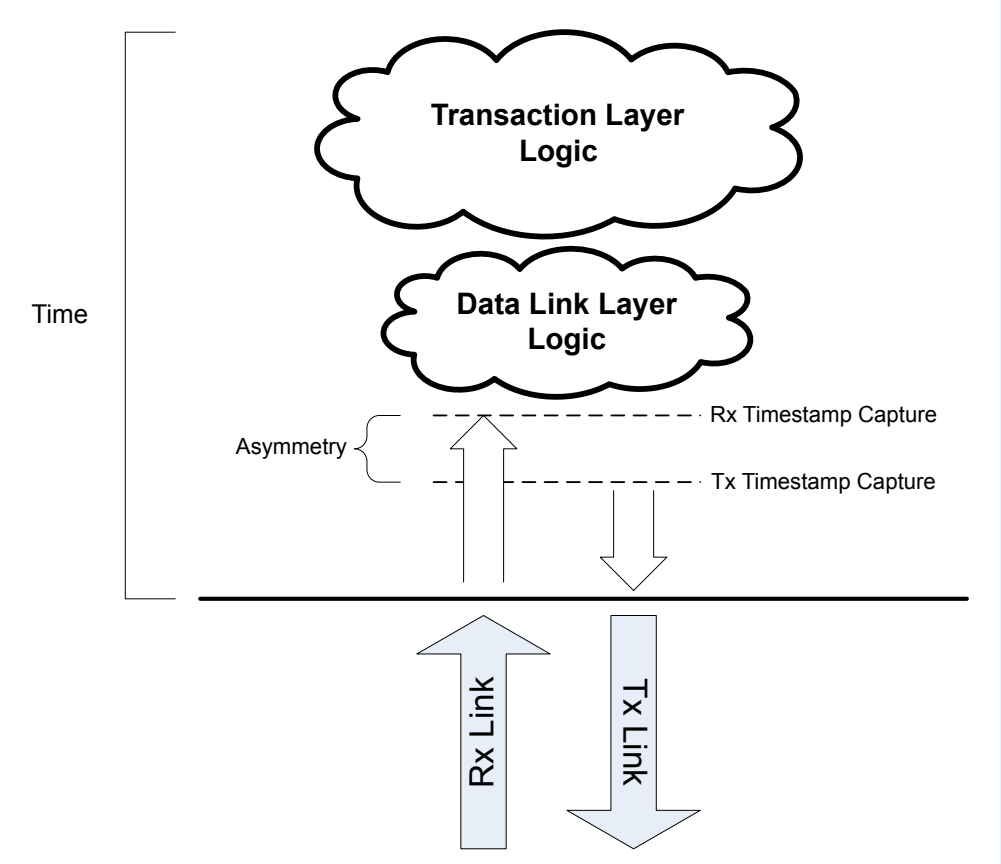

Figure 6-27 PTM Timestamp Capture Example

Figure 6-27 illustrates how to select suitable timestamp capture points. For some implementations, the logic within the Transaction Layer and Data Link Layers is non-deterministic. Implementation details and current conditions have considerable impact on exactly when a particular packet may encounter any particular processing step. This makes it effectively impossible to capture any timestamp that accurately records the time of a particular physical event if timestamps are captured in the higher layers.

6.23 Readiness Notifications (RN)

Readiness Notifications (RN) is intended to reduce the time software needs to wait before issuing Configuration Requests to a Device or Function following DRS Events or FRS Events. RN includes both the Device Readiness Status (DRS) and

Function Readiness Status (FRS) mechanisms. These mechanisms provide a direct indication of Configuration-Readiness (see Terms and Acronyms entry for “Configuration-Ready”). When used, DRS and FRS allow an improved behaviour over the CRS mechanism, and eliminate its associated periodic polling time of up to 1 second following a reset.

It is permitted that system software/firmware provide mechanisms that supersede the FRS and/or DRS mechanisms, however such software/firmware mechanisms are outside the scope of this specification.

IMPLEMENTATION NOTE
Optimizing Configuration Readiness

It is strongly recommended that implementers of system firmware/software avoid unnecessary delays wherever possible. It is strongly recommended that hardware be designed to eliminate or minimize required delays, and to make full use of the mechanisms provided in this specification and related specifications to communicate what, if any, delays are required. Hardware implementers should appropriately document implementation behavior to enable system firmware/software to implement optimal behaviors.

Even with good documentation, some cases may at first appear problematic - for example, how can system firmware benefit from the Device Readiness Status (DRS) mechanism, when it is necessary to read from Root Port Configuration space to do so? In such cases, platform specific knowledge is required, i.e. that the Root Port supports Immediate Readiness.

6.23.1 Device Readiness Status (DRS)

When implemented, DRS must be used to indicate when a Device is Configuration-Ready following any of the following Device-level occurrences, which are subsequently referred to as “DRS Events”:

- Exit from Cold Reset
- Exit from Warm Reset, Hot Reset, Loopback, or Disabled
- Exit from L2/L3 Ready
- Any other scenario where the Port transitions from DL_Down to DL_Up status.

The DRS Message protocol requirements include the following:

- There is no enable or disable mechanism for DRS. For Downstream Ports that support DRS, the DRS Supported bit in the Link Capabilities 2 register must be Set. For Upstream Ports that support DRS, it is strongly recommended that the DRS Supported bit in the Link Capabilities 2 register be Set. It is expressly permitted for Upstream Ports to send DRS Messages even when the DRS Supported bit is Clear.
- A DRS Message must be transmitted by a DRS-capable Upstream Port following every DL_Down to DL_Up transition when all non-VF Functions on the Logical Bus associated with that Upstream Port become ready.
  - A Type 0 Function is ready when it is Configuration-Ready.
  - A Type 1 Function that is a Switch Upstream Port is ready when it is Configuration-Ready and all Functions on its secondary bus are Configuration-Ready.
  - A Type 1 Function that is not a Switch Upstream Port is ready when the Function itself is Configuration-Ready.
- After a Device transmits a DRS Message, non-VF Functions indicated as Configuration-Ready by that DRS Message must not return Completions with CRS unless a subsequent DRS Event occurs.

Additional requirements relating to Switches implementing DRS include:

- Must support DRS functionality in all Ports
- Implementation at each Downstream Port of the DRS Signaling Control field.
- For any physically-integrated Device that appears beneath a Switch Downstream Port, the DRS sent by the Switch does not indicate Configuration Readiness for that Device
  - For such a Device, implementation and use of DRS is independent of the Switch

Additional requirements for Root Ports and Switch Downstream Ports include:

Implementation of the DRS Message Received bit, which indicates receipt of a DRS Message

IMPLEMENTATION NOTE

DRS Messages and ACS Source Validation

Functions are permitted to transmit DRS Messages before they have been assigned a Bus Number. Such messages will have a Requester ID with a Bus Number of 00h. If the Downstream Port has ACS Source Validation enabled, these Messages (see Section 6.12.1.1) will likely be detected as an ACS Violation error.

6.23.2 Function Readiness Status (FRS)

When implemented, FRS must be used to indicate a specific Function as being Configuration-Ready following any of the following Function-level occurrences, which are subsequently referred to as “FRS Events”:

- Function Level Reset (FLR)
- Completion of $D3_{Hot}$ to D0 transition
- Setting or Clearing of VF Enable in a PF (SR-IOV)

The FRS Message protocol requirements include the following:

- The Requester ID of the FRS Message must indicate the Function that has changed readiness status (see Section 2.2.8.6.4)
- The FRS Reason field in the FRS Message must indicate why that Function changed readiness status
- After a Function transmits an FRS Message, the indicated Function(s) must not return Completions with CRS unless a subsequent DRS Event or FRS Event occurs

Additional requirements for Switches implementing FRS include:

- Must support FRS functionality in the Upstream Port and all Downstream Ports
- The ability to transmit FRS Messages Upstream when required by the FRS protocol

Additional requirements for Physical Functions (PFs) include:

- The ability to transmit FRS Message Upstream when the VF Enable or VF Disable process completes

Additional requirements for Root Ports and Root Complex Event Collectors implementing FRS include:

- Must implement the FRS Queuing Extended Capability (see Section 7.8.9)

6.23.3 FRS Queuing

Root Ports and Root Complex Event Collectors that support FRS must implement the FRS Queuing Extended Capability (see Section 7.8.9).

For a Root Port, the FRS Message Queue contains FRS Messages received by the Root Port or generated by the Root Port.

For a Root Complex Event Collector, the FRS Message Queue contains FRS Messages generated by RCiEPs associated with the Root Complex Event Collector (see Section 7.9.10) or generated by the Root Complex Event Collector.

The FRS Message Queue must satisfy the following requirements:

- The FRS Message Queue must be empty following Reset.
- For a Root Port, the FRS Message Queue must be emptied when the Link goes to DL_Down.
- FRS Messages must be queued in the order received.
- If the FRS Message Queue is not full at the time an FRS Message is received or is internally generated, that FRS Message must be entered in the queue and the FRS Message Received bit must set to 1b.
- If the FRS Message Queue is full at the time an FRS Message is received or is internally generated, that FRS Message must be discarded and the FRS Message Overflow bit must be set to 1b. The pre-existing FRS Message Queue must be preserved.
- The oldest FRS Message must be visible in the FRS Message Queue register (see Section 7.8.9.4).
- Writing the FRS Message Queue register must remove the oldest element from the queue.
- When either FRS Message Received or FRS Message Overflow transitions from 0b to 1b, an interrupt must be generated if enabled.

6.24 Enhanced Allocation

The Enhanced Allocation (EA) Capability is an optional Capability that allows the allocation of I/O, Memory and Bus Number resources in ways not possible with the BAR and Base/Limit mechanisms in the Type 0 and Type 1 Configuration Headers.

It is only permitted to apply EA to certain functions, based on the hierarchal structure of the functions as seen in PCI configuration space, and based on certain aspects of how functions exist within a platform environment (see Figure 6-28).

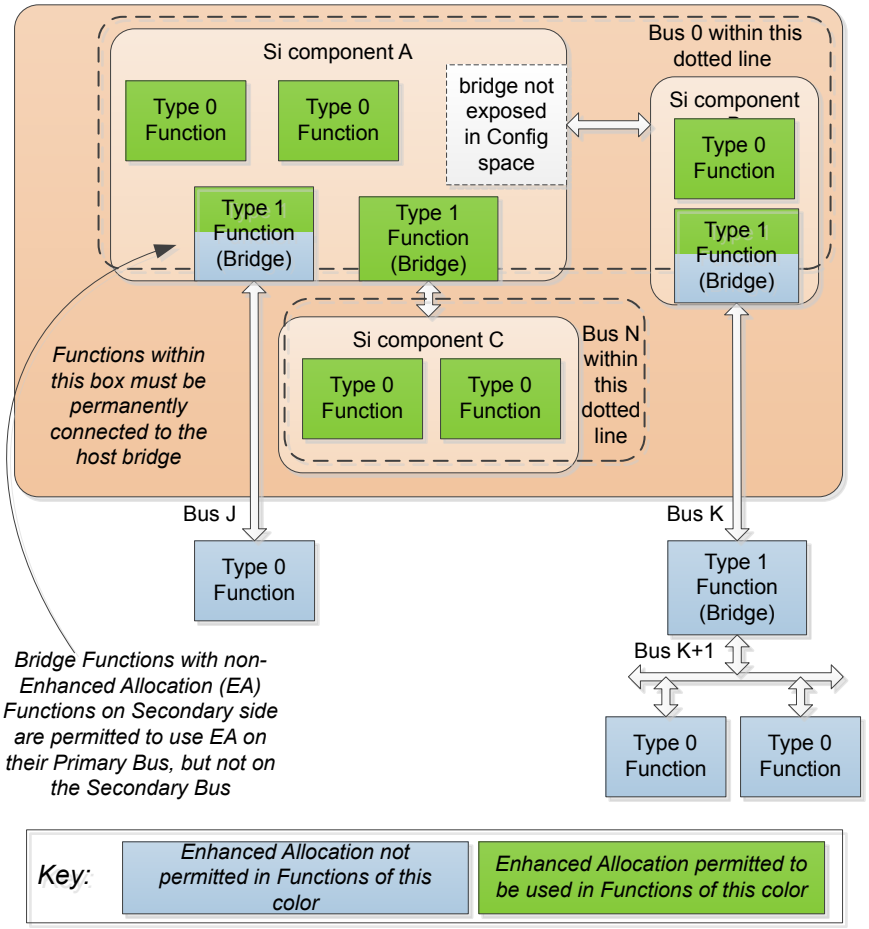

Figure 6-28 Example Illustrating Application of Enhanced Allocation

Only functions that are permanently connected to the host bridge are permitted to use EA. A bridge function (i.e., any function with a Type 1 Configuration Header), is permitted to use EA for both its Primary Side and Secondary Side if and only if the function(s) behind the bridge are also permanently connected (below one or more bridges) to the host bridge, as shown for “Si component C” in Figure 6-28 .

A bridge function is permitted to use EA only for its Primary Side if the function(s) behind the bridge are not permanently connected to the bridge, as with the bridges above Bus J and Bus K in Figure 6-28 , and in this case the non-EA resource allocation mechanisms in the Type 1 Header for Bus numbers, MMIO ranges and I/O ranges are used for the Secondary side of the bridge. System software must ensure that the allocated Bus numbers are within the range indicated in the Fixed Secondary Bus Number and Fixed Subordinate Bus Number registers of the EA capability. System software must ensure that the allocated MMIO and I/O ranges are within ranges indicated with the corresponding Properties in the EA capability for resources to be allocated behind the bridge. For Bus numbers, MMIO and I/O ranges behind the bridge, hardware is permitted to indicate overlapping ranges in multiple bridge functions, however, in such cases, system software must ensure that the ranges actually assigned are non-overlapping.

Functions that rely exclusively on EA for I/O and Memory address allocation must hardwire all bits of all BARs in the PCI Header to 0. Such Functions must be clearly documented as relying on EA for correct operation, and platform integrators must ensure that only EA-aware firmware/software are used with such Functions.

When a Function allocates resources using EA and indicates that a resource range is associated with an equivalent BAR number, the Function must not request resources through the equivalent BAR, which must be indicated by hardwiring all bits of the equivalent BAR to 0.

For a bridge function that is permitted to implement EA based on the rules above, it is permitted, but not required, for the bridge function to use EA mechanisms to indicate resource ranges that are located behind the bridge Function. In the example shown in in Figure 6-28 , the bridge above Bus N is permitted to use EA mechanisms to indicate the resources used by the two functions in "Si component C", or that bridge is permitted to not indicate the resources used by the two functions in "Si component C". System firmware/software must comprehend that such bridge functions are not required to indicate inclusively all resources behind the bridge, and as a result system firmware/software must make a complete search of all functions behind the bridge to comprehend the resources used by those functions.

A Function with an Expansion ROM is permitted to use the existing mechanism or the EA mechanism, but is not permitted to support both. If a Function uses the EA mechanism (EA entry with BEI of 8), the Expansion ROM Base Address and Expansion ROM Enable fields must be hardwired to 0 (see Section 7.5.1.2.4). The Enable bit of the EA entry is equivalent to the Expansion ROM Enable bit. If a Function uses Expansion ROM Base Address Register mechanism, no EA entry with a BEI of 8 is permitted. In both cases, Expansion ROM Validation, if supported, uses the Expansion ROM Validation Status and Expansion ROM Validation Details fields (see Section 7.5.1.2.4).

The requirements for enabling and/or disabling the decode of I/O and/or Memory ranges are unchanged by EA, including but not limited to the Memory Space and I/O Space enable bits in the Command register.

Any resource allocated using EA must not overlap with any other resource allocated using EA, except as permitted above for identifying permitted address ranges for resources behind a bridge.

6.25 *Emergency Power Reduction State*

Emergency Power Reduction State is an optional mechanism to request that Functions quickly reduce their power consumption. Emergency Power Reduction is a fail-safe mechanism intended to be used to prevent system damage and is not intended to provide normal dynamic power management.

If a Function implements Emergency Power Reduction State, it must also implement the Power Budgeting extended capability and must report Power Budgeting values for this state (see Section 7.8.1). Devices that are integrated on the system board are not required to implement the Power Budgeting extended capability, but if they do so, they must meet the preceding requirement.

Functions enter and exit this state based on either autonomously or via external requests. External requests may be either following a signaling protocol defined in an applicable form factor specification, or by a vendor-specific method. Table 6-15 defines how the Emergency Power Reduction Supported and Emergency Power Reduction Initialization Required fields determine the mechanisms that are allowed to trigger entry and exit from this state (see Section 7.5.3.15).

Table 6-15 Emergency Power Reduction Supported Values

Emergency Power Reduction Supported	Emergency Power Reduction Initialization Required	Entry/Exit Permitted by		
		Form Factor Mechanism	Vendor Specific Mechanism(s)	Autonomous Mechanisms
00b	0	No	Yes	Yes
	1	No	No	No
01b	Any	No	Yes	Yes
10b	Any	Yes	Yes	Yes

Emergency Power Reduction Supported	Emergency Power Reduction Initialization Required	Entry/Exit Permitted by		
		Form Factor Mechanism	Vendor Specific Mechanism(s)	Autonomous Mechanisms
11b	Reserved			

Functions may indicate that they require re-initialization on exit from this state:

- If the Emergency Power Reduction Initialization Required bit is Clear (see Section 7.5.3.15):
  - On entry to this state, the Function either operates normally (perhaps with reduced performance), or enters a device specific “power reduction dormant state”. The Upstream Port of the Device remains operating. Outstanding requests initiated by or directed to the Function must complete normally.
  - On exit from this state, the Function operates normally (perhaps resuming normal performance). Functions that entered a “power reduction dormant state” exit that state. In either case, no software intervention is required.
- If the Emergency Power Reduction Initialization Required bit is Set (see Section 7.5.3.15):
  - On entry to this state, the Function ceases normal operation. The Upstream Port of the associated Device is permitted to enter DL_Down.
    - If the Upstream Port remains in DL_Up, outstanding requests directed to or initiated by the Function must complete normally.
    - If the Upstream Port enters DL_Down, outstanding request behavior is defined in Section 2.9.1 . This transition may result in a Surprise Down error.
    - Sticky bits must be preserved in this state.
  - On exit from this state, software intervention is required to resume normal operation. The mechanism used to indicate to software when this is required is outside the scope of this specification (e.g., a device specific interrupt). If the Upstream Port entered DL_Down, all Functions of the Device are reset and a full reconfiguration is required (see Section 2.9.2).

The following rules apply to the Emergency Power Reduction State:

- A Device supports Emergency Power Reduction State if at least one Function in the Upstream Port indicates support (i.e., Emergency Power Reduction Supported is non-zero).
- Emergency Power Reduction State is associated with a Device. All Functions in a Device that support it enter and exit this state at the same time.
- Functions where the Emergency Power Reduction Supported field is 00b are not affected by the Emergency Power Reduction State of the Device as long as the Upstream Port remains in DL_Up. The Emergency Power Reduction Detected bit is RsvdZ.
- Functions where the Emergency Power Reduction Supported field is 01b or 10b:
  - Set the Emergency Power Reduction Detected bit when the Device enters Emergency Power Reduction State.
  - Clear the Emergency Power Reduction Detected bit when requested if the Device has exited the Emergency Power Reduction State.
- For Switches, Downstream Switch Ports enter and exit Emergency Power Reduction State at the same time as the associated Upstream Switch Port. The corresponding fields in Configuration Space are reserved for Downstream Switch Ports.
- For SR-IOV Devices, VFs enter and exit Emergency Power Reduction State at the same time as their PF. The corresponding fields in Configuration Space are reserved for VFs.

- Encoding 10b shall not be used unless the associated form factor specification defines a mechanism for requesting Emergency Power Reduction.
- It is strongly recommended that the Emergency Power Reduction Supported field be initialized by hardware or firmware within the Function prior to initial device enumeration. This initialization is permitted to be deferred to device driver load when this is not practical (e.g., when there is no firmware ROM).

IMPLEMENTATION NOTE

Diagnostic Checking of Emergency Power Reduction Detected

The Emergency Power Reduction Detected bit permits system software to detect that Emergency Power Reduction State was entered, even momentarily. The Emergency Power Reduction Request bit can be used by software to request entry. Normally, software would use a system specific method to enter the Emergency Power Reduction State using external mechanisms.

IMPLEMENTATION NOTE

Emergency Power Reduction State: Example Add-in Card

Figure 6-29 shows an example multi-Device add-in card supporting Emergency Power Reduction. Note that Device C does not support the Emergency Power Reduction State. Device C might be a Switch that fans out to Devices A and B.

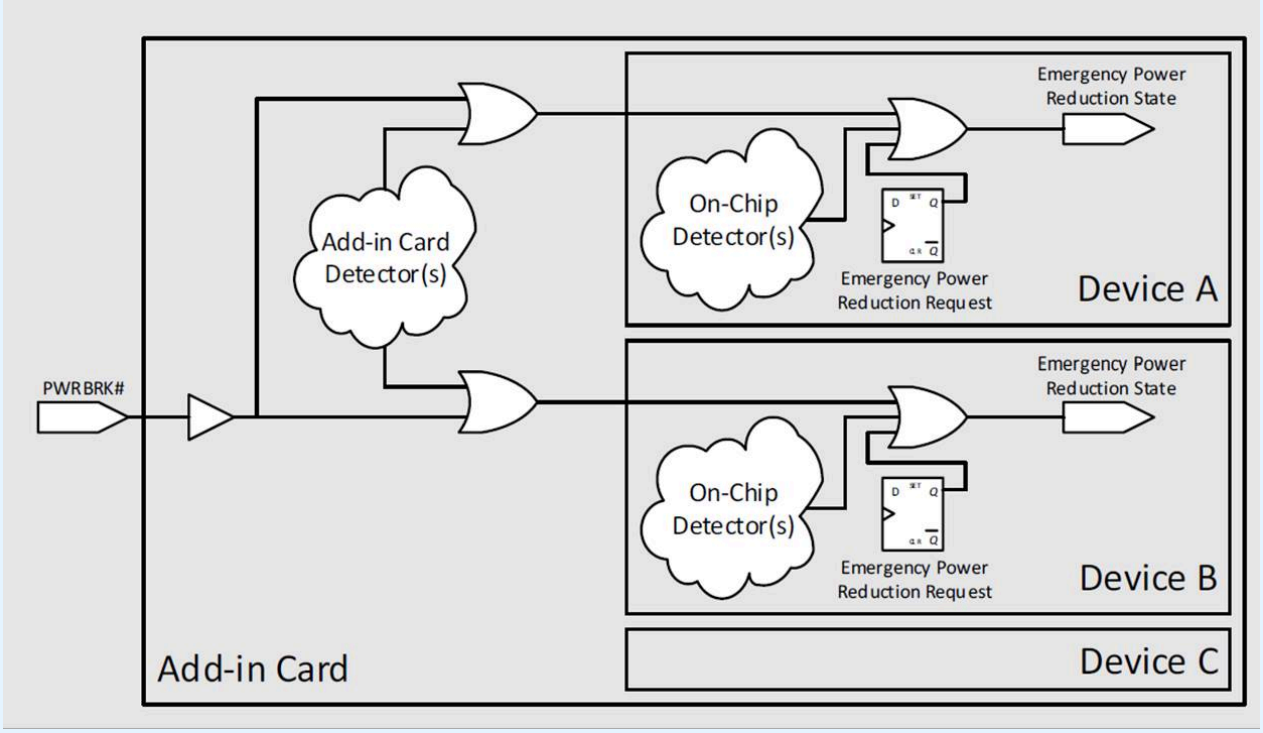

Figure 6-29 Emergency Power Reduction State: Example Add-in Card

6.26 *Hierarchy ID Message*

When software initializes a PCI Hierarchy, it assigns unique Bus and Device numbers to each component so that every Function in the Hierarchy has a unique Routing ID within that Hierarchy. To ensure that Routing IDs are unique in large systems that contain more than one Hierarchy and in clustered systems that contain multiple Hierarchies, additional information is required to augment the Routing ID to produce a unique number. Functions can be uniquely identified by the combination of:

- Unique Identifier for the System (or Root Complex)
- Unique Identifier for the Hierarchy within that Root Complex
- Routing ID within that Hierarchy

The Hierarchy ID Message (see Section 2.2.8.6.5) is used to provide the additional information needed for a Function to uniquely identify itself in a multi-hierarchy platform.

Hierarchy ID Messages are generated by a Downstream Port upon software request. Received messages at an Upstream Port are reported in the Hierarchy ID Extended Capability (see Section 7.9.18).

Hierarchy ID Messages are a PCI-SIG-Defined Type 1 VDM. Hierarchy ID Messages can safely be sent at any time and components that do not comprehend them will silently ignore them.

Hierarchy ID Messages typically are sent from a Downstream Port at the top of the Hierarchy (e.g., a Root Port). In systems where the Root Port does not support Hierarchy ID Messages, Hierarchy ID Messages can be sent from Switch Downstream Ports.

The Hierarchy ID Message is intended for use by software, firmware, and/or hardware. When using the Hierarchy ID Message, all bits of the Hierarchy ID, System GUID, System GUID Authority ID fields must be compared, without regard to any internal structure. How this information is used is outside the scope of this specification.

Layout of the Hierarchy ID Message is shown in Figure 2-33 . Fields in the Hierarchy ID Message are as follows:

Hierarchy ID contains the Segment Group Number associated with this Hierarchy (as defined by the *PCI Firmware Specification*). This field can be used in conjunction with the Routing ID to uniquely identify a Function within a System. The value 0000h indicates the default (or only) Hierarchy of the Root Complex. Non-zero values indicate additional Hierarchies.

System GUID[143:0], in conjunction with System GUID Authority ID, provides a globally unique identification for a System.

System GUID[143:136] is byte 14 in the Hierarchy ID Message.
System GUID[135:128] is byte 15 in the Hierarchy ID Message.
System GUID[127:120] is byte 16 in the Hierarchy ID Message.
System GUID[119:112] is byte 17 in the Hierarchy ID Message.
System GUID[111:104] is byte 18 in the Hierarchy ID Message.
System GUID[103:96] is byte 19 in the Hierarchy ID Message.
System GUID[95:88] is byte 20 in the Hierarchy ID Message.
System GUID[87:80] is byte 21 in the Hierarchy ID Message.
System GUID[79:72] is byte 22 in the Hierarchy ID Message.
System GUID[71:64] is byte 23 in the Hierarchy ID Message.
System GUID[63:56] is byte 24 in the Hierarchy ID Message.
System GUID[55:48] is byte 25 in the Hierarchy ID Message.
System GUID[47:40] is byte 26 in the Hierarchy ID Message.
System GUID[39:32] is byte 27 in the Hierarchy ID Message.
System GUID[31:24] is byte 28 in the Hierarchy ID Message.
System GUID[23:16] is byte 29 in the Hierarchy ID Message.
System GUID[15:8] is byte 30 in the Hierarchy ID Message.
System GUID[7:0] is byte 31 in the Hierarchy ID Message.

System GUID Authority ID identifies the mechanism used to ensure that the System GUID is globally unique. The mechanism for choosing which Authority ID to use for a given system is implementation specific. The defined values are shown in Table 6-16 .

Table 6-16 System GUID Authority ID Encoding

Authority ID	Description
00h	**None** - System GUID[143:0] is not meaningful.

Authority ID	Description
	System GUID[143:0] must be 0.
01h	**Timestamp** - System GUID[63:0] contains a timestamp associated with the particular system. Encoding is a Unix 64 bit time (number of seconds since midnight UTC January 1, 1970). The mechanism of choosing the timestamp to represent a system is implementation specific. System GUID[143:64] must be 0.
02h	**IEEE EUI-48** - System GUID[47:0] contains a 48 bit Extended Unique Identifier (EUI-48) associated with the particular system. Encoding is defined by the IEEE. See [EUI-48] for details. EUI-48 values are frequently used as network interface MAC addresses. The mechanism of choosing the EUI-48 value to represent a system is implementation specific. System GUID[143:48] must be 0.
03h	**IEEE EUI-64** - System GUID[63:0] contains a 64 bit Extended Unique Identifier (EUI-64) associated with the particular system. Encoding is defined by the IEEE. See [EUI-64] for details. The mechanism of choosing the EUI-64 value to represent a system is implementation specific. System GUID[143:64] must be 0.
04h	**RFC-4122 UUID** - System GUID[127:0] contain a UUID as defined by the IETF in [RFC-4122]. This definition is technically equivalent to [ITU-T-Rec.-X.667] or [ISO-IEC-9834-8]. The mechanism of choosing the UUID value to represent a system is implementation specific. System GUID[143:128] must be 0
05h	**IPv6 Address** - System GUID[127:0] contains the unique IPv6 address of one of the network interfaces of the system. The mechanism of choosing the IPv6 value to represent a system is implementation specific. System GUID[143:128] must be 0.
06h to 7Fh	**Reserved** - System GUID[143:0] contains a unique value. The mechanism used to ensure uniqueness is outside the scope of this specification.
80h to FFh	**PCI-SIG Vendor Specific** - System GUID Authority ID values 80h to FFh are reserved for PCI-SIG vendor-specific usage. System GUID[143:128] contains a PCI-SIG assigned Vendor ID. System GUID[127:0] contain a unique number assigned by that vendor. The mechanism used for assigning numbers is implementation specific. One possible mechanism would be to use the serial number assigned to the system. The mechanism used to choose between these System GUID Authority IDs is implementation specific. One usage would be to allow a vendor to define up to 128 distinct 128-bit System GUID schemes.

IMPLEMENTATION NOTE

System GUID Consistency and Stability

To support the purpose of System GUID, software should ensure that a single system uses identical System GUID and System GUID Authority ID values everywhere.

Implementers should carefully consider their stability requirements for the System GUID value. For example, some use cases may require that the value not change when the system is rebooted. In those cases, a mechanism that picks the EUI-48 value associated with the first Ethernet MAC address discovered might be problematic if the result changes due to hardware failure, system reconfiguration, or variations/parallelism in the discovery algorithm.

IMPLEMENTATION NOTE

Hierarchy ID vs. Device Serial Number

The Device Serial Number mechanism can also be used to uniquely identify a component (see Section 7.9.3). Device Serial Number may be a more expensive solution to this problem if it involves a ROM associated with each component.

IMPLEMENTATION NOTE
Virtual Functions and Hierarchy ID

The Hierarchy ID capability can be emulated by the Virtualization Intermediary (VI). Doing so provides VF software access to this Hierarchy ID information.

When VF hardware needs access to this information, the VF should implement the Hierarchy ID capability. This provides access to both VF software and hardware.

In some situations, the VF should get the same information as the PF. In other situations, particularly those involving migration of Virtual Machines, it may be appropriate to present the VF with Hierarchy ID information that differs from the associated PF and from other VFs associated with that PF.

The following mechanisms are supported:

	VF Hierarchy ID Capability	Hierarchy ID VF Configurable	Hierarchy ID Writeable	VF Software has access	VF Hardware has access	VF Hierarchy Data / GUID
1	Not Present	n/a	n/a	No	No	Not Emulated
2				Yes	No	Emulated
3	Present	0b	0b	Yes	Yes	Same as PF
4		1b	0b	Yes	Yes	Same as PF
5		1b	1b	Yes	Yes	Configured by VI

In mechanism 1, the the Virtualization Intermediary does not emulate the capability. VF software and hardware have no access.

In mechanism 2, the Virtualization Intermediary emulates the capability and returns whatever Hierarchy ID information is desired. VF software has access. VF hardware does not have access.

In mechanisms 3 and 4, VF information is the same as the PF and is automatically filled in from received Hierarchy ID messages. Both VF hardware and software have access.

In mechanism 5, VF information is configured by software (probably the VI). Both VF hardware and software have access.

6.27 Flattening Portal Bridge (FPB)

6.27.1 Introduction

The Flattening Portal Bridge (FPB) is an optional mechanism which can be used to improve the scalability and runtime reallocation of Routing IDs and Memory Space resources.

For non-ARI Functions associated with an Upstream Port, the Routing ID consists of a 3-bit Function Number portion, which is determined by the construction of the Upstream Port hardware, and a 13-bit Bus Number and Device number portion, determined by the Downstream Port above the Upstream port.

For ARI Functions associated with an Upstream Port, the Routing ID consists of an 8-bit Function Number portion, and only the 8-bit Bus Number portion is determined by the Downstream Port above the Upstream port.

A bridge that implements the FPB Capability can itself also be referred to as an FPB. The FPB Capability can be applied to any logical bridge, as illustrated in Figure 6-30 .

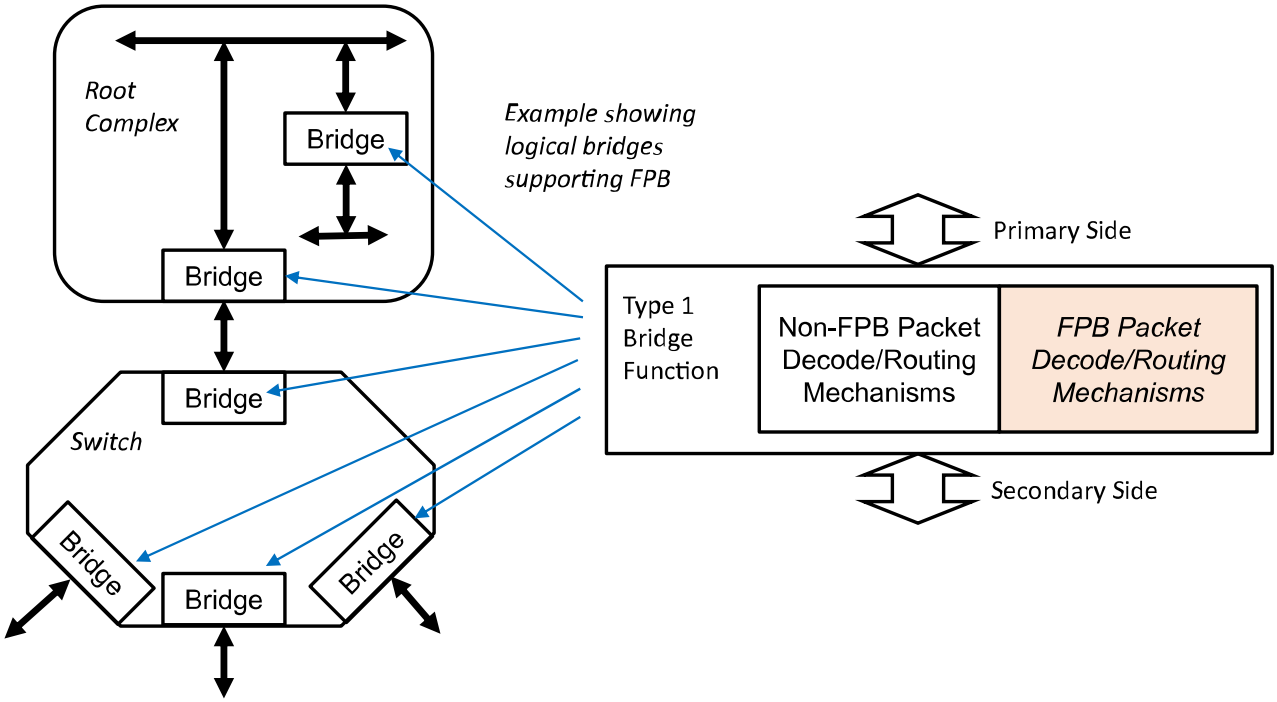

Figure 6-30 FPB High Level Diagram and Example Topology

FPB changes the way Bus Numbers are consumed by Switches to reduce waste, by “flattening” the way Bus Numbers are used inside of Switches and by Downstream Ports (see Figure 6-31).

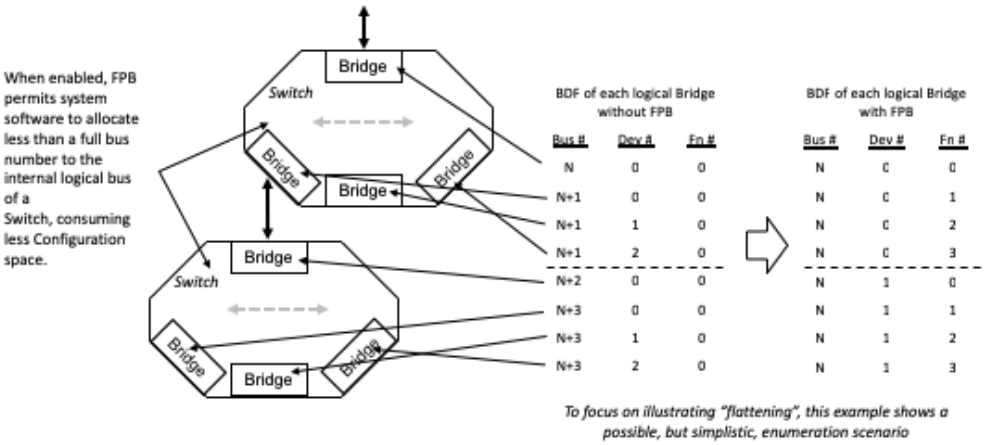

Figure 6-31 Example Illustrating "Flattening" of a Switch

FPB defines mechanisms for system software to allocate Routing IDs and Memory Space resources in non-contiguous ranges, enabling system software to assign pools of these resources from which it can allocate "bins" to Functions below the FPB. This is done using a bit vector where each bit when Set assigns a corresponding range of resources to the Secondary Side of the bridge (see Figure 6-32).

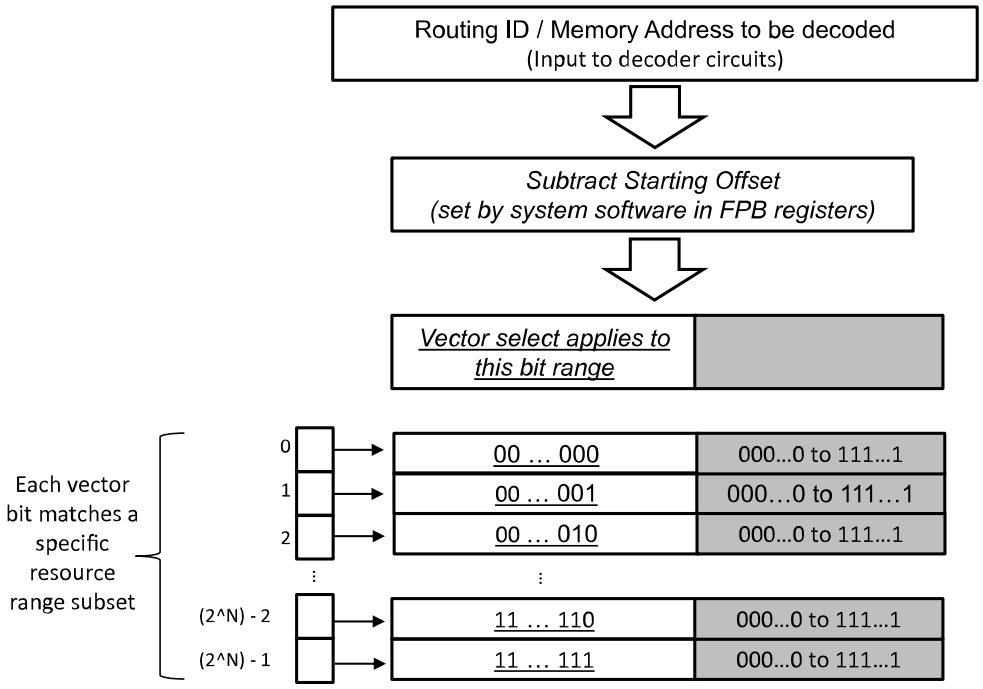

Figure 6-32 Vector Mechanism for Address Range Decoding

This allows system software to assign Routing IDs and/or Memory Space resources required by a device hot-add without having to rebalance other, already assigned resource ranges, and to return to the pool resources freed, for example by a hot remove event.

FPB is defined to allow both the non-FPB and FPB mechanisms to operate simultaneously, such that, for example, it is possible for system firmware/software to implement a policy where the non-FPB mechanisms continue to be used in parts of the system where the FPB mechanisms are not required (see Figure 6-33). In this figure, the decode logic is assumed to provide a '1' output when a given TLP is decoded as being associated with the bridge's Secondary Side. The non-FPB decode mechanisms apply as without FPB, so for example only the Bus Number portion (bits 15:8) of a Routing ID is tested by the non-FPB decode logic when evaluating an ID routed TLP.

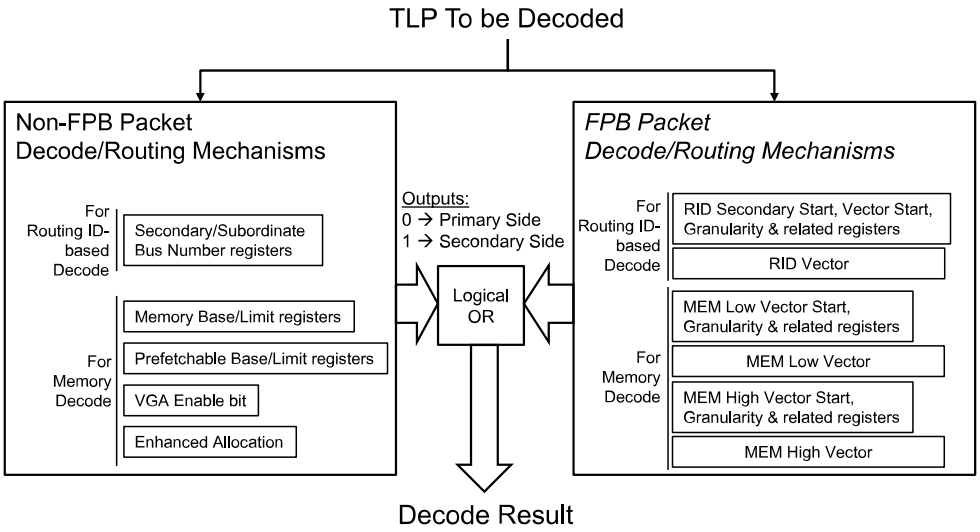

Figure 6-33 Relationship between FPB and non-FPB Decode Mechanisms

It is important to recognize that, although FPB adds additional ways for a specific bridge to decode a given TLP, FPB does not change anything about the fundamental ways that bridges operate within the Switch and Root Complex architectural structures. FPB uses the same architectural concepts to provide management mechanisms for three different resource types:

1. Routing IDs
2. Memory below 4 GB (“MEM Low”)
3. Memory above 4 GB (“MEM High”)

A hardware implementation of FPB is permitted to support any combination of these three mechanisms. For each mechanism, FPB uses a bit-vector to indicate, for a specific subset range of the selected resource type, if resources within that range are associated with the Primary or Secondary side of the FPB. Hardware implementations are permitted to implement a small range of sizes for these vectors, and system firmware/software is enabled to make the most effective use of the available vector by selecting an initial offset at which the vector is applied, and a granularity for the individual bits within the vector to indicate the size of the resource range to which the bits in a given vector apply.

6.27.2 Hardware and Software Requirements

The following rules apply when any of the FPB mechanisms are used:

- If system software violates any of the rules concerning FPB, the hardware behavior is undefined.
- It is permitted to implement FPB in any PCI bridge (Type 1) Function, and every Function that implements FPB must implement the FPB Capability (see Section 7.8.10).
- If a Switch implements FPB then the Upstream Port and all Downstream Ports of the Switch must implement FPB.
- Software is permitted to enable FPB at some Switch Ports and not others.
- A Root Complex is permitted to implement FPB on some Root Ports but not on others.

- A Type 1 Function is permitted to implement the FPB mechanisms applying to any one, two or three of these elemental mechanisms:
  - Routing IDs (RID)
  - Memory below 4 GB (“MEM Low”)
  - Memory above 4 GB (“MEM High”)
- System software is permitted to enable any combination (including all or none) of the elemental mechanisms supported by a specific FPB.
- The error handling and reporting mechanisms, except where explicitly modified in this section, are unaffected by FPB.
- Following any reset of the FPB Function, the FPB hardware must Clear all bits in all implemented vectors.
- Once enabled (through the FPB RID Decode Mechanism Enable, FPB MEM Low Decode Mechanism Enable, and/or FPB MEM High Decode Mechanism Enable bits), if system software subsequently disables an FPB mechanism, the values of the entries in the associated vector are undefined, and if system software subsequently re-enables that FPB mechanism the FPB hardware must Clear all bits in the associated vector.
- If an FPB is implemented with the No_Soft_Reset bit Clear, when that FPB is cycled through D0→ $D3_{Hot}$→D0, then all FPB mechanisms must be disabled, and the FPB must Clear all bits in all implemented vectors.
- If an FPB is implemented with the No_Soft_Reset bit Set, when that FPB is cycled through D0→ $D3_{Hot}$→D0, then all FPB configuration state must not change, and the entries in the FPB vectors must be retained by hardware.
- Hardware is not required to perform any type of bounds checking on FPB calculations, and system software must ensure that the FPB parameters are correctly programmed
  - It is explicitly permitted for system software to program Vector Start values that cause the higher order bits of the corresponding vector to surpass the resource range associated with a given FPB, but in these cases system software must ensure that those higher order bits of the vector are Clear.
  - Examples of errors that system software must avoid include duplication of resource allocation, combinations of start offsets with set vector bits that could create “wrap-around” or bounds errors

The following rules apply to the FPB Routing ID (RID) mechanism:

- FPB hardware must consider a specific range of RIDs to be associated with the Secondary side of the FPB if the Bus Number portion falls within the Bus Number range indicated by the values programmed in the Secondary and Subordinate Bus Number registers logically OR’d with the value programmed into the corresponding entry in the FPB RID Vector.
- If it is intended to use only the FPB RID mechanism for BDF decoding, then system software must ensure that both the Secondary and Subordinate Bus Number registers are 0.
- System software must ensure that the FPB routing mechanisms are configured such that Configuration Requests targeting Functions Secondary side of the FPB will be routed by the FPB from the Primary to Secondary side of the FPB.

When ARI is not enabled, the FPB RID mechanism can be applied with different granularities, programmable by system software through the FPB RID Vector Granularity field in the FPB RID Vector Control 1 Register. Figure 6-34 illustrates the relationships between the layout of RIDs and the supported granularities. The reader may find it helpful to refer to this figure when considering the requirements defined below and in the definition of the Flattening Portal Bridge (FPB) Capability (see Section 7.8.6).

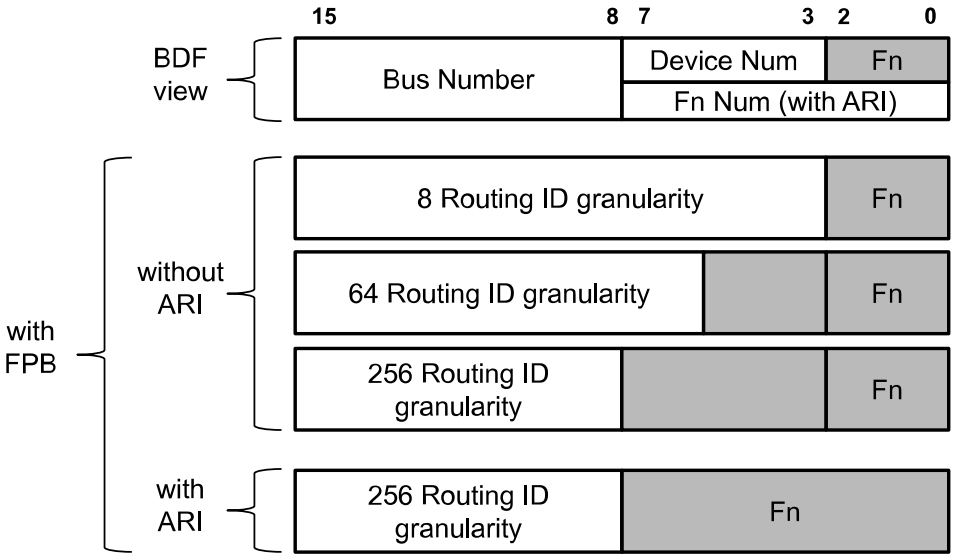

Figure 6-34 Routing IDs (RIDs) and Supported Granularities

- System software must program the FPB RID Vector Granularity and FPB RID Vector Start fields in the FPB RID Vector Control 1 Register per the constraints described in the descriptions of those fields.
- For all FPBs other than those associated with Upstream Ports of Switches:
  - When ARI Forwarding is not supported, or when the ARI Forwarding Enable bit in the Device Control 2 Register is Clear, FPB hardware must convert a received on the Primary side of the FPB to a Type 0 Configuration Request on the Secondary side of the FPB when bits 15:3 of the Routing ID of the Type 1 Configuration Request matches the value in the RID Secondary Start field in the FPB RID Vector Control 2 Register, and system software must configure the FPB accordingly.
  - When the ARI Forwarding Enable bit in the Device Control 2 Register is Set, FPB hardware must convert a Type 1 Configuration Request received on the Primary side of the FPB to a Type 0 Configuration Request on the Secondary side of the FPB when the Bus Number portion of the Routing ID of the Type 1 Configuration Request matches the value in the Bus Number address (bits 15:8 only) of the Secondary Start field in the FPB RID Vector Control 2 Register, and system software must configure the FPB accordingly.
- For FPBs associated with Upstream Ports of Switches only, when the FPB RID Decode Mechanism Enable bit is Set, FPB hardware must use the FPB Num Sec Dev field of the FPB Capabilities register to indicate the quantity of Device Numbers associated with the Secondary Side of the Upstream Port bridge, which must be used by the FPB in addition to the RID Secondary Start field in the FPB RID Vector Control 2 Register to determine when a Configuration Request received on the Primary side of the FPB targets one of the Downstream Ports of the Switch, determining in effect when such a Request must be converted form a Type 1 Configuration Request to a Type 0 Configuration Request, and system software must configure the FPB appropriately.
  - System software configuring FPB must comprehend that the logical internal structure of a Switch will change depending on the value of the FPB RID Decode Mechanism Enable bit in the Upstream Port of a Switch.
  - Downstream Ports must use their corresponding RID values, and their Requester IDs and Completer IDs, as determined by the Upstream Port's FPB Num Sec Dev and RID Secondary Start values
  - All implemented Functions in the range determined by the Switch Upstream Port Function's RID Secondary Start and FPB Num Sec Dev must be Switch Downstream Ports associated with that

Switch Upstream Port; System Software is required to scan all Functions in this range to determine which are Implemented.

- It is strongly recommended that System Software assign the RID Secondary Start such that the Bus and Device Numbers are not the same as for the Switch Upstream Port; otherwise, the resulting hardware behavior is undefined.

- FPBs must implement bridge mapping for INTx virtual wires (see Section 2.2.8.1)
- Hardware and software must apply this algorithm (or the logical equivalent) to determine which entry in the FPB RID Vector applies to a given Routing ID (RID) address:
  - IF the RID is below the value of FPB RID Vector Start, then the RID is out of range (below the start) and so cannot be associated with the Secondary side of the bridge, ELSE
  - calculate the offset within the vector by first subtracting the value of FPB RID Vector Start, then dividing this according to the value of FPB RID Vector Granularity to determine the bit index within the vector.
  - IF the bit index value is greater than the length indicated by FPB RID Vector Size Supported, then the RID is out of range (beyond the top of the range covered by the vector) and so cannot be associated with the Secondary side of the bridge, ELSE
  - if the bit value within the vector at the calculated bit index location is 1b, THEN the RID address is associated with the Secondary side of the bridge, ELSE the RID address is associated with the Primary side of the bridge.

The following rules apply to the FPB MEM Low mechanism:

The FPB MEM Low mechanism can be applied with different granularities, programmable by system software through the FPB MEM Low Vector Granularity field in the FPB MEM Low Vector Control Register. Figure 6-35 illustrates the relationships between the layout of addresses in the memory address space below 4 GB to which the FPB MEM Low mechanism applies. The reader may find it helpful to refer to this figure when considering the requirements defined below and in the definition of the Flattening Portal Bridge (FPB) Capability (see Section 7.8.10).

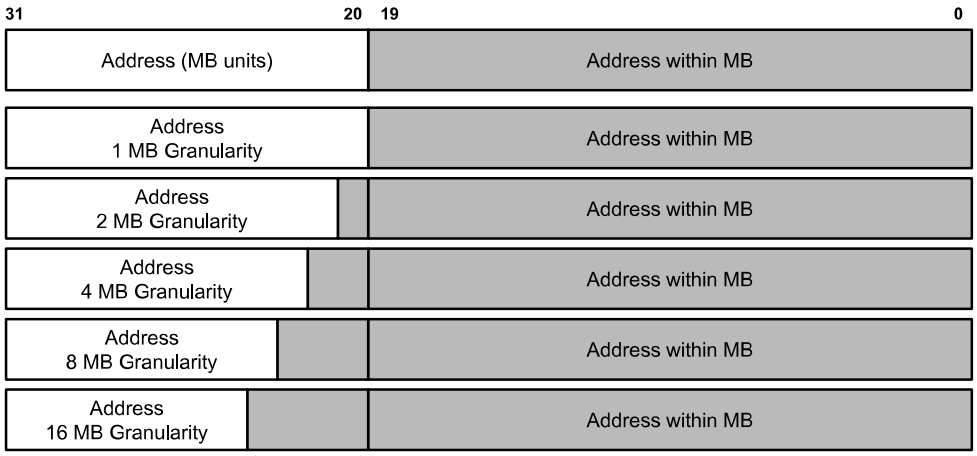

Figure 6-35 Addresses in Memory Below 4 GB and Effect of Granularity

- System software must program the FPB MEM Low Vector Granularity and FPB MEM Low Vector Start fields in the FPB MEM Low Vector Control Register per the constraints described in the descriptions of those fields.
- FPB hardware must consider a specific Memory address to be associated with the Secondary side of the FPB if that Memory address falls within any of the ranges indicated by the values programmed in other bridge

Memory decode registers (enumerated below) logically OR'd with the value programmed into the corresponding entry in the FPB MEM Low Vector. Other bridge Memory decode registers include:

- Memory Base/Limit registers
- Prefetchable Base/Limit registers
- VGA Enable bit in the Bridge Control register
- Enhanced Allocation (EA) Capability (if supported)
- FPB MEM High mechanism (if supported and enabled)

- Hardware and software must apply this algorithm (or the logical equivalent) to determine which entry in the FPB MEM Low Vector applies to a given Memory address:
  - If the Memory address is below the value of FPB MEM Low Vector Start, then the Memory address is out of range (below) and so is not associated with the Secondary side of the bridge by means of this mechanism, else
  - calculate the offset within the vector by first subtracting the value of FPB MEM Low Vector Start, then dividing this according to the value of FPB MEM Low Vector Granularity to determine the bit index within the vector.
  - If the bit index value is greater than the length indicated by FPB MEM Low Vector Size Supported, then the Memory address is out of range (above) and so is not associated with the Secondary side of the bridge by means of this mechanism, else
  - if the bit value within the vector at the calculated bit index location is 1b, then the Memory address is associated with the Secondary side of the bridge, else the Memory address is associated with the Primary side of the bridge.

The following rules apply to the FPB MEM High mechanism:

- System software must program the FPB MEM High Vector Granularity and FPB MEM High Vector Start Lower fields in the FPB MEM High Vector Control 1 Register per the constraints described in the descriptions of those fields.
- FPB hardware must consider a specific Memory address to be associated with the Secondary side of the FPB if that Memory address falls within any of the ranges indicated by the values programmed in other bridge Memory decode registers (enumerated below) logically OR'd with the value programmed into the corresponding entry in the FPB MEM High Vector. Other bridge Memory decode registers include:
  - Memory Base/Limit registers
  - Prefetchable Base/Limit registers
  - VGA Enable bit in the Bridge Control register
  - Enhanced Allocation (EA) Capability (if supported)
  - FPB MEM Low mechanism (if supported and enabled)
- Hardware and software must apply this algorithm to determine which entry in the FPB MEM High Vector applies to a given Memory address:
  - If the Memory address is below the value of FPB MEM High Vector Start Upper/FPB MEM High Vector Start Lower, then the Memory address is out of range (below) and so is not associated with the Secondary side of the bridge by means of this mechanism, else
  - calculate the offset within the vector by first subtracting the value of FPB MEM High Vector Start Upper/FPB MEM High Vector Start Lower, then dividing this according to the value of FPB MEM High Vector Granularity to determine the bit index within the vector.

- If the bit index value is greater than the length indicated by FPB MEM High Vector Size Supported, then the Memory address is out of range (above) and so is not associated with the Secondary side of the bridge by means of this mechanism, else
- if the bit value within the vector at the calculated bit index location is 1b, then the Memory address is associated with the Secondary side of the bridge, else the Memory address is associated with the Primary side of the bridge.

IMPLEMENTATION NOTE
FPB Address Decoding

FPB uses a bit vector mechanism to decode ranges of Routing IDs, and Memory Addresses above and below 4 GB. A bridge supporting FPB contains the following for each resource type/range where it supports the use of FPB:

- A Bit vector
- A Start Address
- A Granularity

These are used by the bridge to determine if a given address is part of the range decoded by FPB as associated with the secondary side of the bridge. An address that is determined not to be associated with the secondary side of the bridge using either or both of the non-FPB decode mechanisms and the FPB decode mechanisms is (by default) associated with the primary side of the bridge. Here, when we use the term “associated” we mean, for example, that the bridge will apply the following handling to TLPs:

- Associated with Primary, Received at Primary → Unsupported Request (UR)
- Associated with Primary, Received at Secondary → Forward upstream
- Associated with Secondary, Received at Primary → Forward downstream
- Associated with Secondary, Received at Secondary → Unsupported Request (UR)

In FPB, every bit in the vector represents a range of resources, where the size of that range is determined by the selected granularity. If a bit in the vector is Set, it indicates that TLPs addressed to an address within the corresponding range are to be associated with the secondary side of the bridge. The specific range of resources each bit represents is dependent on the index of that bit, and the values in the Start Address & Granularity. The Start Address indicates the lowest address described by the bit vector. The Granularity indicates the size of the region that is represented by each bit. Each successive bit in the vector applies to the subsequent range, increasing with each bit according to the Granularity.

For example, consider a bridge using FPB to describe a MEM Low range. FPB MEM Low Vector Start has been set to FC0h, indicating that the range described by the bit vector starts at address FC00 0000. FPB MEM Low Vector Granularity has been set to 0000b, indicating that each bit represents a 1 MB range.

From these values we can determine that bit 0 of the vector represents a 1MB range starting at FC000 0000 (FC00 0000-FC0F FFFF), bit 1 represents FC10 0000-FC1F FFFF, etc.

Bits in the vector that are set to 0 indicate that the range is not included in the range described by FPB. In the above example, If bit 0 is Clear, packets addressed to anywhere between FC00 0000 and FC0F FFFF should not be routed to the secondary bus of the bridge due to FPB.

IMPLEMENTATION NOTE
Hardware and Software Considerations for FPB

FPB is intended to address a class of issues with PCI/PCIe architecture that relate to resource allocation inefficiency. These issues can be categorized as “static” or “dynamic” use case scenarios, where static use cases refer to scenarios where resources are allocated at system boot and then typically not changed again, and dynamic use cases refer to scenarios where run-time resource rebalancing (e.g. allocation of new resources, freeing of resources no longer needed) is required, due to hot add/remove, or by other needs.

In the Static cases there are limits on the size of hierarchies and number of Endpoints due to the use of additional Bus Numbers and the lack of use of Device Numbers caused by the PCI/PCIe architectural definition for Switches and Downstream Ports. FPB addresses this class of problems by “flattening” the use of Routing IDs (RIDs) so that Switches and Downstream Ports are able to make more efficient use of the available RIDs.

For the Dynamic cases, without FPB, the “best known method” to avoid rebalancing has been to reserve large ranges of Bus Numbers and Memory Space in the bridge above the relevant Port or Endpoint such that hopefully any future needs can be satisfied within the pre-allocated ranges. This leads to potentially unused allocations, which makes the Routing ID issues worse, and in a resource constrained platform this approach is difficult to implement, even for relatively simple cases, where, for example, one might have an add-in card implementing a single Endpoint replaced by another add-in card that has a Switch and two Endpoints, so that although an initial allocation of just one Bus would have been sufficient, the initial allocation breaks immediately with the new add-in card.

For Memory Space the pre-allocation approach is problematic when hot-plugged Endpoints may require the allocation of Memory Space below 4 GB, which by its nature is a limited resource, which is quickly used up by pre-allocation of even relatively small amounts, and for which pre-allocation is unattractive because of the multiple system elements placing demands on system address space allocation below 4 GB.

FPB includes mechanisms to enable discontinuous resource range allocation/reallocation for both Requester IDs and Memory Space. The intent is to allow system software the ability to maintain resource “pools” which can be allocated (and freed back to) at run-time, without disrupting other operations in progress as is required with rebalancing.

To support the run time use of FPB by system software, FPB hardware implementations should avoid introducing stalls or other types of disruptions to transactions in flight, including during the times that system software is modifying the state of the FPB hardware. It is not, however, expected that hardware will attempt to identify cases where system software erroneously modifies the FPB configuration in a way that does affect transactions in flight. Just as with the non-FPB mechanisms, it is the responsibility of system software to ensure that system operation is not corrupted due to a reconfiguration operation.

It is not explicitly required that system firmware/software perform the enabling and/or disabling of FPB mechanisms in a particular sequence, however care should be taken to implement resource allocation operations in a hierarchy such that the hardware and software elements of the system are not corrupted or caused to fail.

6.28 Vital Product Data (VPD)

Vital Product Data (VPD) is the information that uniquely defines items such as the hardware, software, and microcode elements of a system. The VPD provides the system with information on various FRUs (Field Replaceable Unit) including Part Number, Serial Number, and other detailed information. VPD also provides a mechanism for storing information such as performance and failure data on the device being monitored. The objective, from a system point of view, is to collect this information by reading it from the hardware, software, and microcode components.

Support of VPD within add-in cards is optional depending on the manufacturer. Though support of VPD is optional, add-in card manufacturers are encouraged to provide VPD due to its inherent benefits for the add-in card, system manufacturers, and for Plug and Play.

The mechanism for accessing VPD is documented in Section 7.9.19 .

VPD for PCI Express is unchanged from the definition in the [PCI-3.0]. That definition, in turn, was based on earlier versions of the [PCI] as well as the [PLUG-PLAY-ISA-1.0a].

Vital Product Data is made up of Small and Large Resource Data Types.

Table 6-17 Small Resource Data Type Tag Bit Definitions

Offset	Field Name		
Byte 0	Value = 0xxx xyyyb		
	Bit 7	Small Resource Type	0b
	Bits 6:3	Small Item Name	xxxx
	Bits 2:0	Length in bytes	yy
Bytes 1 to n	Actual information		

Table 6-18 Large Resource Data Type Tag Bit Definitions

Offset	Field Name		
Byte 0	Value = 1xxx xxxxb		
	Bit 7	Large Resource Type	1b
	Bits 6:0	Large Item Name	xxxxxxx
Byte 1	Length in bytes of data items bits[7:0] (lsb)		
Byte 2	Length in bytes of data items bits[15:8] (msb)		
Bytes 3 to n	Actual data items		

The first VPD tag is the Identifier String (02h) and provides the product name of the device.

One VPD-R (10h) tag is used as a header for the read-only keywords. The VPD-R list (including tag and length) must checksum to zero. Attempts to write the read-only data will be executed as a no-op.

One VPD-W (11h) tag is used as a header for the read-write keywords. The storage component containing the read/write data is a non-volatile device that will retain the data when powered off.

The last tag must be the End Tag (0Fh).

A small example of the resource data type tags used in a typical VPD is shown in Table 6-19 .

Table 6-19 Resource Data Type Flags for a Typical VPD

TAG 02h	Identifier String	Large Resource Data Type
TAG 10h	VPD-R list containing one or more VPD keywords	Large Resource Data Type
TAG 11h	VPD-W list containing one or more VPD keywords	Large Resource Data Type
TAG 0Fh	End Tag	Small Resource Data Type

6.28.1 VPD Format

Information fields within a VPD resource type consist of a three-byte header followed by some amount of data (see Figure 6-36). The three-byte header contains a two-byte keyword and a one-byte length. A keyword is a two-character (ASCII) mnemonic that uniquely identifies the information in the field. The last byte of the header is binary and represents the length value (in bytes) of the data that follows.

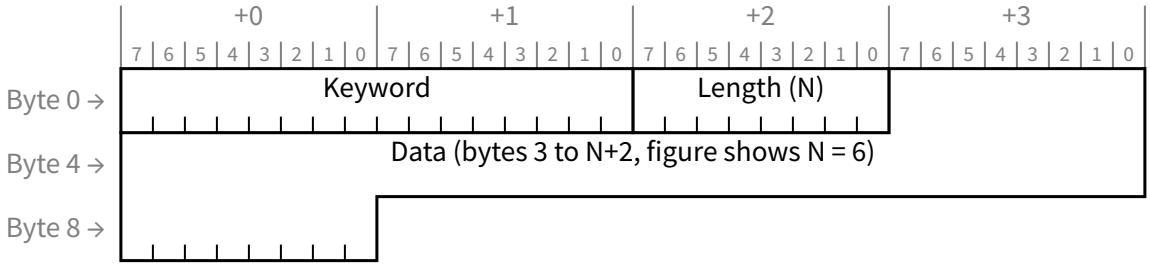

Figure 6-36 VPD Format

VPD keywords are listed in two categories: read-only fields and read/write fields. Unless otherwise noted, keyword data fields are provided as ASCII characters. Use of ASCII allows keyword data to be moved across different enterprise computer systems without translation difficulty.

An example of the “add-in card serial number” VPD item is as follows:

Table 6-20 Example of Add-in Serial Card Number

Byte 0	53h “S”	Keyword: SN
Byte 1	4Eh “N”	
Byte 3	08h	Length: 8
Byte 4	30h “0”	Data: “00000194”
Byte 5	30h “0”	
Byte 6	30h “0”	
Byte 7	30h “0”	
Byte 8	30h “0”	
Byte 9	31h “1”	
Byte 10	39h “9”	
Byte 11	34h “4”	

6.28.2 VPD Definitions

This section describes the current VPD large and small resource data tags plus the VPD keywords. This list may be enhanced at any time. Companies wishing to define a new keyword should contact the PCISIG. All unspecified values are reserved for SIG assignment.

6.28.2.1 VPD Large and Small Resource Data Tags

VPD is contained in four types of Large and Small Resource Data Tags. The following tags and VPD keyword fields may be provided in PCI devices.

Table 6-21 VPD Large and Small Resource Data Tags

Tag	Description
Large resource type Identifier String Tag (02h)	This tag is the first item in the VPD storage component. It contains the name of the add-in card in alphanumeric characters.
Large resource type VPD-R Tag (10h)	This tag contains the read only VPD keywords for an add-in card.
Large resource type VPD-W Tag (11h)	This tag contains the read/write VPD keywords for an add-in card.
Small resource type End Tag (0Fh)	This tag identifies the end of VPD in the storage component.

6.28.2.2 Read-Only Fields

Table 6-22 VPD Read-Only Fields

Keyword	Name	Description
PN	Add-in Card Part Number	This keyword is provided as an extension to the Device ID (or Subsystem ID) in the Configuration Space header in Figure 6-36 .
EC	Engineering Change Level of the Add-in Card	The characters are alphanumeric and represent the engineering change level for this add-in card.
FG	Fabric Geography	Reserved for legacy use by [PICMG] specifications.
LC	Location	Reserved for legacy use by [PICMG] specifications.
MN	Manufacture ID	This keyword is provided as an extension to the Vendor ID (or Subsystem Vendor ID) in the Configuration Space header in Figure 6-36 . This allows vendors the flexibility to identify an additional level of detail pertaining to the sourcing of this device.
PG	PCI Geography	Reserved for legacy use by [PICMG] specifications.
SN	Serial Number	The characters are alphanumeric and represent the unique add-in card Serial Number.

Keyword	Name	Description
TR	Thermal Reporting	This keyword provides a standard interface for reporting four fields: ***AFI Level***, ***MaxTherm***, ***DTherm***, and ***MaxAmbient***. The data area for this field is four bytes long. This data is encoded as a 4-byte binary value in little endian order (byte 0 contains bits 7:0). This value contains the four fields as follows: AFI Level bits [3:0], MaxTherm bits [7:4], DTherm bits [11:8], and MaxAmbient bits [19:12] are placed in bits 19:0. Bits 31:20 are Reserved and must be set to 000h. Field description is provided within the [ECN-CEM-THERMAL] to the [CEM-3.0]. This keyword is intended to be used only in designs based on that form factor specification. Note that due to the character nature of the VPD encoding mechanism, this binary value is permitted to start on any byte boundary within the VPD.
Vx	Vendor Specific	This is a vendor specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through 9 or A through Z.
CP	Extended Capability	This field allows a new capability to be identified in the VPD area. Since dynamic control/status cannot be placed in VPD, the data for this field identifies where, in the device's memory or I/O address space, the control/status registers for the capability can be found. Location of the control/status registers is identified by providing the index (a value between 0 and 5) of the Base Address register that defines the address range that contains the registers, and the offset within that Base Address register range where the control/status registers reside. The data area for this field is four bytes long. The first byte contains the ID of the extended capability. The second byte contains the index (zero based) of the Base Address register used. The next two bytes contain the offset (in little-endian order) within that address range where the control/status registers defined for that capability reside.
RV	Checksum and Reserved	The first byte of this item is a checksum byte. The checksum is correct if the sum of all bytes in VPD (from VPD address 0 up to and including this byte) is zero. The remainder of this item is reserved space (as needed) to identify the last byte of read-only space. The read-write area does not have a checksum. This field is required.

6.28.2.3 Read/Write Fields

Table 6-23 VPD Read/Write Fields

Keyword	Name	Description
Vx	Vendor Specific	This is a vendor specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through 9 or A through Z.
Yx	System Specific	This is a system specific item and the characters are alphanumeric. The second character (x) of the keyword can be 0 through 9 or B through Z.
YA	Asset Tag Identifier	This is a system specific item and the characters are alphanumeric. This keyword contains the system asset identifier provided by the system owner.
RW	Remaining Read/ Write Area	This descriptor is used to identify the unused portion of the read/write space. The product vendor initializes this parameter based on the size of the read/write space or the space remaining following the Vx VPD items. One or more of the Vx, Yx, and RW items are required.

6.28.2.4 VPD Example

The following is an example of a typical VPD.

Table 6-24 VPD Example

Offset	Item Value
0	Large Resource Type ID String Tag (02h) 82h “Product Name”
1	Length 0021h
3	Data “ABCD Super-Fast Widget Controller”
36	Large Resource Type VPD-R Tag (10h) 90h
37	Length 0059h
39	VPD Keyword “PN”
41	Length 08h
42	Data “6181682A”
50	VPD Keyword “EC”
52	Length 0Ah
53	Data “4950262536”
63	VPD Keyword “SN”
65	Length 08h
66	Data “00000194”
74	VPD Keyword “MN”
76	Length 04h
77	Data “1037”
81	VPD Keyword “RV”
83	Length 2Ch
84	Data Checksum
85	Data Reserved (00h)
128	Large Resource Type VPD-W Tag (11h) 91h
129	Length 007Ch
131	VPD Keyword “V1”
133	Length 05h
134	Data “65A01”
139	VPD Keyword “Y1”
141	Length 0Dh
142	Data “Error Code 26”
155	VPD Keyword “RW”

Offset	Item Value
157	Length 61h
158	Data Reserved (00h)
255	Small Resource Type End Tag (0Fh) 78h

6.29 Native PCIe Enclosure Management

NPEM is an optional PCIe Extended Capability that provides mechanisms for enclosure management. This mechanism is designed to provide management for enclosures containing PCIe SSDs that is consistent with the established capabilities in the storage ecosystem.

This section defines the architectural aspects of the mechanism. The NPEM extended capability is defined in Section 7.9.20 .

An enclosure is any platform, box, rack, or set of boxes that contain one or more PCIe SSDs. The NPEM capability provides storage related enclosure control (e.g., status LED control) for a PCIe SSD. The NPEM capability may reside in a Downstream port, or an Endpoint (i.e., the PCIe SSD). Figure 6-37 shows an example configuration with a single Downstream Port containing the NPEM capability and vendor specific logic to control the associated LEDs.

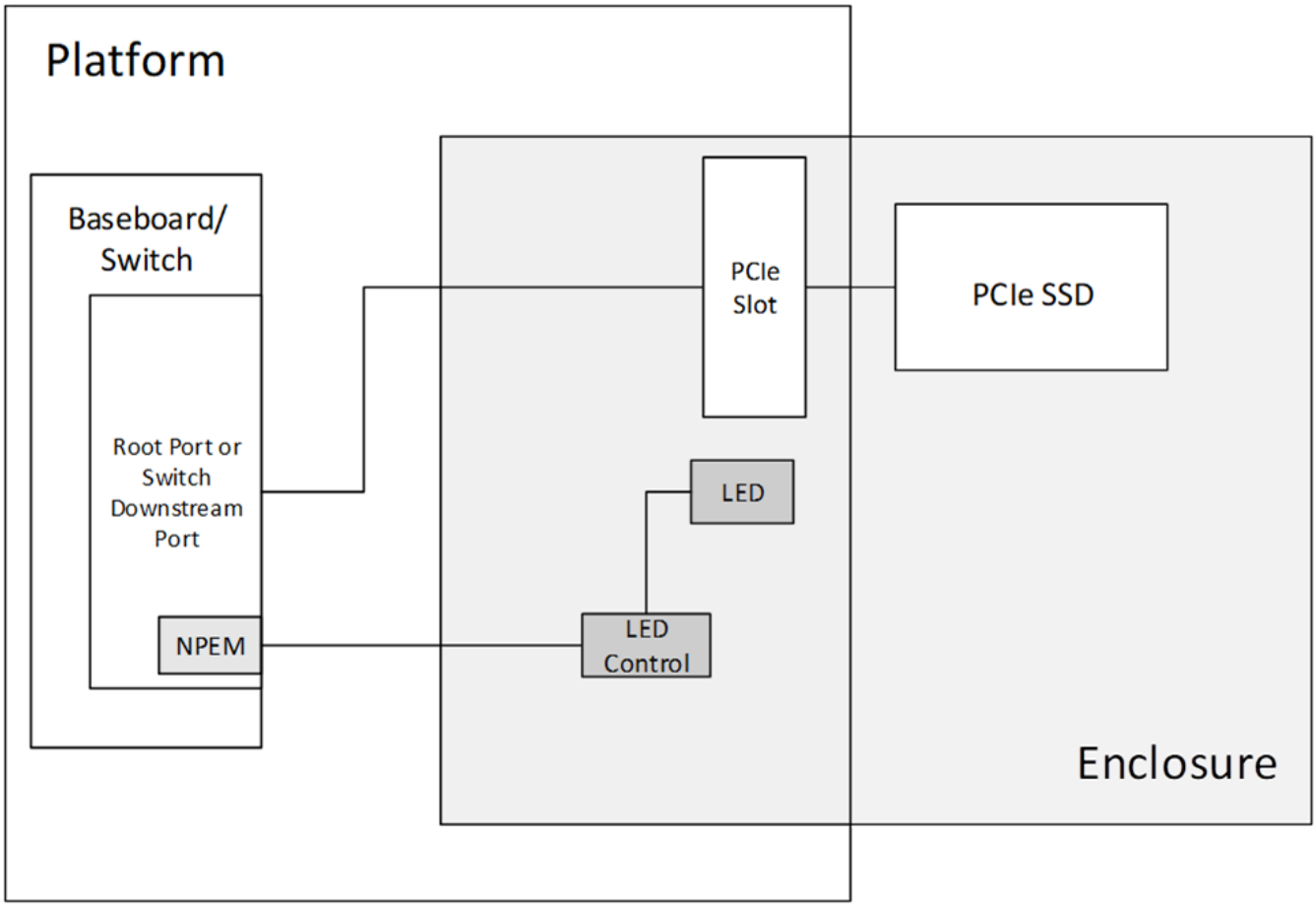

Figure 6-37 Example NPEM Configuration using a Downstream Port

Figure 6-38 shows an example configuration with the NPEM capability located in the Upstream Port (in this case, the SSD function).

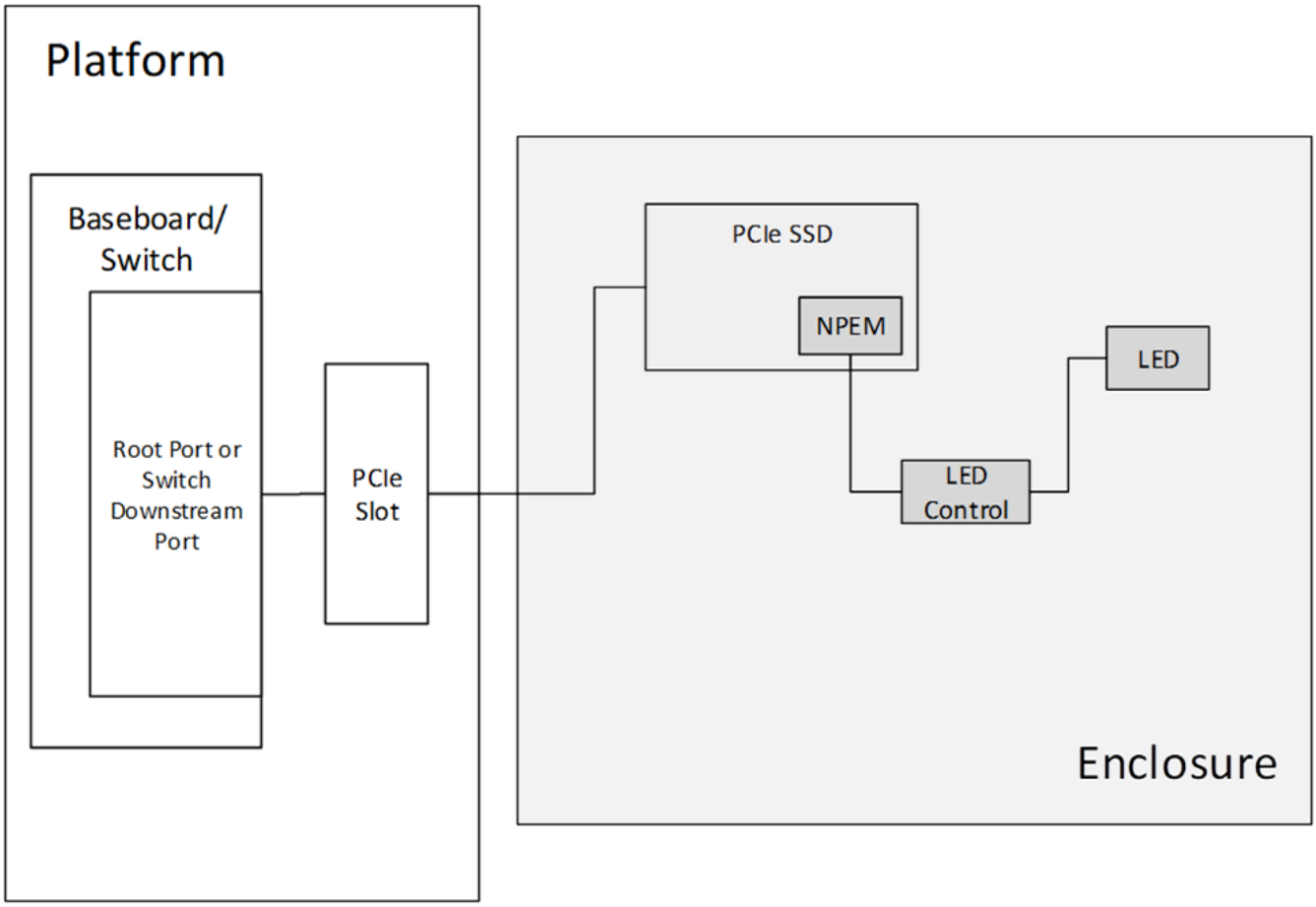

Figure 6-38 Example NPEM Configuration using an Upstream Port

Software issues an NPEM command by writing to the NPEM Control register to change the indications associated with an SSD. NPEM Command is a single write to the NPEM Control register that changes the state of zero or more bits. NPEM indicates a successful completion to software using the command completed mechanism. Figure 6-39 shows the overall flow.

This specification defines the software interface provided by the NPEM capability. The Port to enclosure interface, enclosure, enclosure to LED interface, number of LEDs per SSD, and associated LED blink patterns are all outside the scope of this specification.

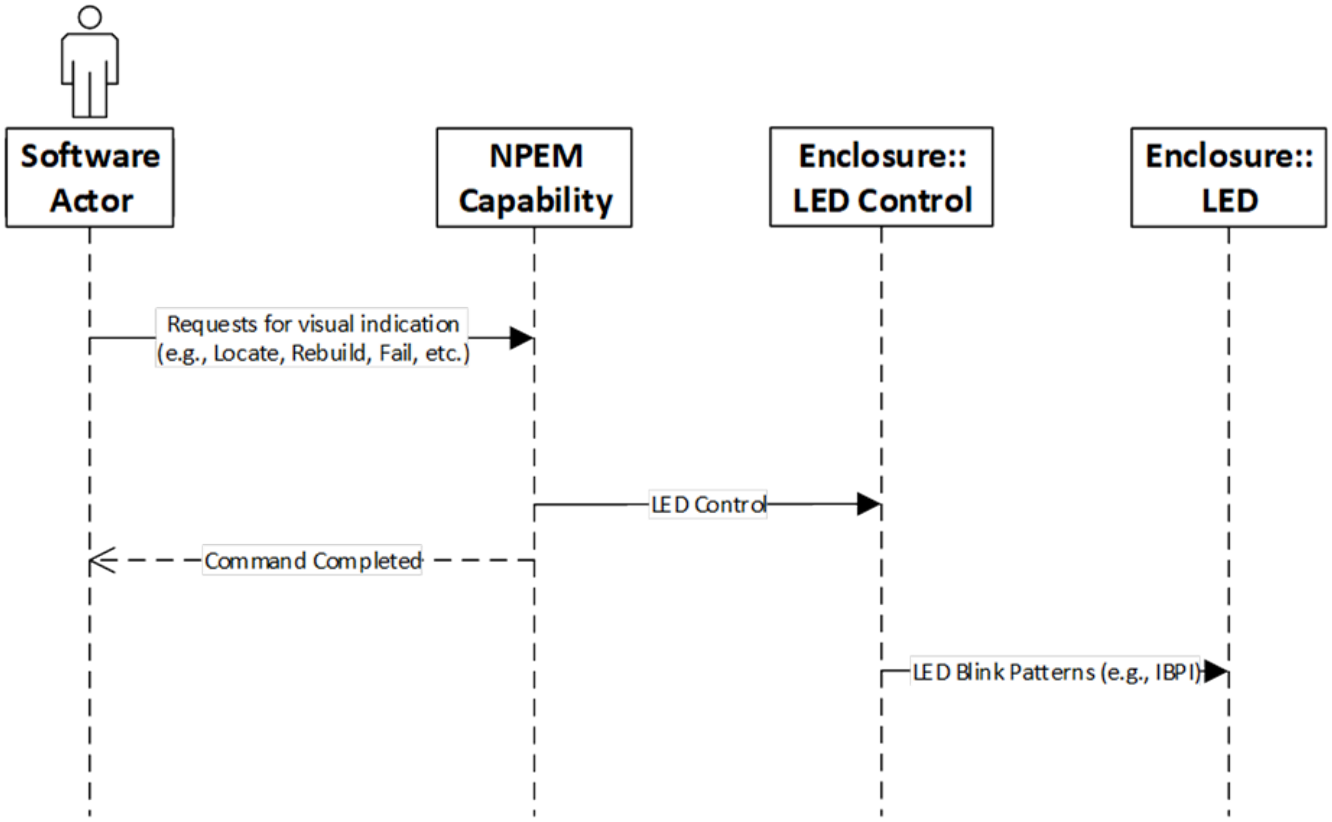

Figure 6-39 NPEM Command Flow

NPEM provides a mechanism for system software to issue a reset to the LED control element within the enclosure by means of the NPEM Reset mechanism, which is independent of the PCIe Link itself. The NPEM command completed mechanism also applies to NPEM Reset.

Storage system admin or software controls the indications for various device states through the NPEM capability.

IMPLEMENTATION NOTE

NPEM States

Table 6-25 shows an example of NPEM states and a possible meaning that some enclosures may assign to the architected NPEM states.

Table 6-25 NPEM States

NPEM State	Actor	Definition
OK	System Admin or Storage Software	OK state may mean the drive is functioning normally. This state may implicitly mean that an SSD is present, powered on, and working normally as seen by the software. A more granular indication of drive not physically present or present but not powered up are both outside the scope of this specification.
Locate	System Admin	Locate state may mean the specific drive is being identified by an admin.
Fail	Storage Software	Fail state may mean the drive is not functioning properly
Rebuild	Storage Software	Rebuild state may mean this drive is part of a multi-drive storage volume/array that is rebuilding or reconstructing data from redundancy on to this specific drive.
PFA	Storage Software	PFA stands for Predicted Failure Analysis. This state may mean the drive is still functioning normally but predicted to fail soon.
Hot Spare	Storage Software	Hot Spare state may mean this drive is marked to be automatically used as a replacement for a failed drive and contents of the failed drive may be rebuilt on this drive.
In A Critical Array	Storage Software	In A Critical Array state may mean the drive is part of a multi-drive storage array and that array is degraded.
In A Failed Array	Storage Software	NPEM In A Failed Array state may mean the drive is part of a multi-drive storage array and that array is failed.
Invalid Device Type	Storage Software	Invalid Device Type state may mean the drive is not the right type for the connector (e.g., An enclosure supports SAS and NVMe drives and this drive state indicates that a SAS drive is plugged into an NVMe slot).
Disabled	Storage Software	Disabled state may mean the drive in this slot is disabled. A removal of this drive from the slot may be safe. The power from this slot may be removed.

IMPLEMENTATION NOTE

Software Polling of NPEM Command Completed

Different NPEM implementations may vary widely in how long they take to complete NPEM commands, from instantaneous to tens of ms. To avoid or minimize software polling overheads, it is recommended that software implement one or both of the following optimizations.

Instead of software writing a command and then immediately polling for completion, it is recommended that software reverse this order. When ready to write a new command, software first polls for completion of the previous command, and then writes the new command. This enables overlapped operation, often completely hiding the time it takes hardware to execute an NPEM command. To enable this polling model, software must initialize the hardware following a reset by writing a no-op command in order to have hardware generate the first NPEM command completion.

For the case where an NPEM command has not completed when software polls the bit, it is recommended that software not continuously "spin" on polling the bit, but rather poll under interrupt at a reduced rate; for example at 10 ms intervals.

6.30 Conventional PCI Advanced Features Operation

For Conventional PCI devices integrated into a Root Complex, the Conventional PCI Advanced Features Capability (AF) provides mechanisms for using advanced features originally developed for PCI Express.

- The Function Level Reset (INITIATE_FLR) mechanism enables software to quiesce and reset hardware with Function-level granularity.

  FLR applies on a per Function basis. Only the targeted Function is affected by the FLR operation.

- The Transactions Pending (TP) mechanism is used to indicate that the Function has issued one or more non-posted transactions (including Delayed Transactions) which have not been completed.

The FLR and TP mechanisms defined here are strictly for Conventional PCI devices integrated into a Root Complex where the implementation permits non-posted transactions for a given Conventional PCI Function to complete even if the value of the Bus Master Enable bit in its Command Register is 0b. Implementations that do not meet this requirement must not implement the FLR and TP mechanisms.

FLR modifies the Function state as follows:

Function registers and Function-specific state machines must be set to their initialization values as specified in this document, except for the following bits, which must not be modified: Fast Back-to-Back Transactions Enable, Cache Line Size, Latency Timer, Interrupt Line, PME_En, PME_Status.

Note that the controls that enable the Function to initiate bus transactions are cleared, including the Bus Master Enable bit in the Command Register, the MSI Enable bit in the MSI Capability Structure, and the like, effectively causing the Function to become quiescent.

After an FLR has been initiated, the Function must complete the FLR within 100 ms. If software initiates an FLR when the Transactions Pending bit is 1b, then software must not initialize the Function until allowing adequate time to achieve reasonable certainty that any outstanding transactions will have completed. The Transactions Pending bit must be clear upon completion of the FLR.

FLR modifies Function state not described by this specification (in addition to state that is described by this specification), and so the following criteria must be applied using Function- specific knowledge to evaluate the Function's behavior in response to an FLR:

- The Function must not give the appearance of an initialized adapter with an active host on any external interfaces controlled by that Function. The steps needed to terminate activity on external interfaces are outside of the scope of this specification.
  - For example, a network adapter must not respond to queries that would require adapter initialization by the host system or interaction with an active host system, but is permitted to perform actions that it is designed to perform without requiring host initialization or interaction. If the network adapter includes multiple Functions that operate on the same external network interface, this rule affects only those aspects associated with the particular Function reset by FLR.
- The Function must not retain within itself software readable state that potentially includes secret information associated with any preceding use of the Function. Main host memory assigned to the Function must not be modified by the Function.
  - For example, a Function with internal memory readable directly or indirectly by host software must clear or randomize that memory.
- The Function must return to a state such that normal configuration of the Function's PCI interface will cause it to be useable by drivers normally associated with the Function

When an FLR is initiated, the targeted Function must behave as follows:

- The Function must complete normally the configuration write that initiated the FLR operation and then initiate the FLR.
- While an FLR is in progress:
  - The Function must not respond to any request on the bus (i.e. requests targeting the Function will Master Abort).

The Transactions Pending (TP) bit indicates that the Function has issued one or more non-posted transactions which have not been completed. This field may be used by software to determine when a Function has become quiescent.

IMPLEMENTATION NOTE

Avoiding Issues with Pending Transactions

An FLR causes a Function to lose track of any pending (outstanding non-posted) transactions. Depending upon the specific implementation of the RC-integrated PCI Function, if software issues an FLR while there are pending transactions, there is a possibility for data corruption as described in the "Avoiding Data Corruption From Stale Completions" Implementation Note.

To avoid potential issues with Root Complex implementations where Stale Completions are possible or a Discard Timer is present, it is recommended that software use an algorithm similar to the following:

1. Software that's performing the FLR synchronizes with other software that might potentially access the Function directly, and ensures that such accesses will not occur during this algorithm.
2. Software clears the entire Command register, disabling the Function from mastering any new transactions.
3. Software polls the Transactions Pending bit in the AF Status Register either until it's clear or until it's been long enough to achieve reasonable certainty that any remaining outstanding Transactions will never complete. On many systems, the Transactions Pending bit will usually clear within a few milliseconds, so software might choose to poll during this initial period using a tight software loop. On rare cases when the Transactions Pending bit doesn't clear by this time, software will need to poll for a longer system-specific period (potentially seconds), so software might choose to conduct this polling using a timer-based interrupt polling mechanism.
4. Software initiates the FLR.
5. Software waits 100 ms.
6. Software reconfigures the Function and enables it for normal operation.

Software Initialization and Configuration

7.

The PCI Express Configuration model supports two Configuration Space access mechanisms:

- PCI-compatible Configuration Access Mechanism (CAM) (see Section 7.2.1)
- PCI Express Enhanced Configuration Access Mechanism (ECAM) (see Section 7.2.2)

The PCI-compatible mechanism supports 100% binary compatibility with Conventional PCI or later aware operating systems and their corresponding bus enumeration and configuration software.

The enhanced mechanism is provided to increase the size of available Configuration Space and to optimize access mechanisms.

7.1 Configuration Topology

To maintain compatibility with PCI software configuration mechanisms, all PCI Express elements have a PCI-compatible Configuration Space. Each PCI Express Link originates from a logical PCI-PCI Bridge and is mapped into Configuration Space as the secondary bus of this Bridge. The Root Port is a PCI-PCI Bridge structure that originates a PCI Express Link from a PCI Express Root Complex (see Figure 7-1).

A PCI Express Switch not using FPB Routing ID mechanisms is represented by multiple PCI-PCI Bridge structures connecting PCI Express Links to an internal logical PCI bus (see Figure 7-2). The Switch Upstream Port is a PCI-PCI Bridge; the secondary bus of this Bridge represents the Switch's internal routing logic. Switch Downstream Ports are PCI-PCI Bridges bridging from the internal bus to buses representing the Downstream PCI Express Links from a PCI Express Switch. Only the PCI-PCI Bridges representing the Switch Downstream Ports may appear on the internal bus. Endpoints, represented by Type 0 Configuration Space Headers, are not permitted to appear on the internal bus.

A PCI Express Endpoint is mapped into Configuration Space as a single Function in a Device, which might contain multiple Functions or just that Function. PCI Express Endpoints and Legacy Endpoints are required to appear within one of the Hierarchy Domains originated by the Root Complex, meaning that they appear in Configuration Space in a tree that has a Root Port as its head. Root Complex Integrated Endpoints (RCiEPs) and Root Complex Event Collectors do not appear within one of the Hierarchy Domains originated by the Root Complex. These appear in Configuration Space as peers of the Root Ports.

Unless otherwise specified, requirements in the Configuration Space definition for a device apply to single Function devices as well as to each Function individually of a Multi-Function Device.

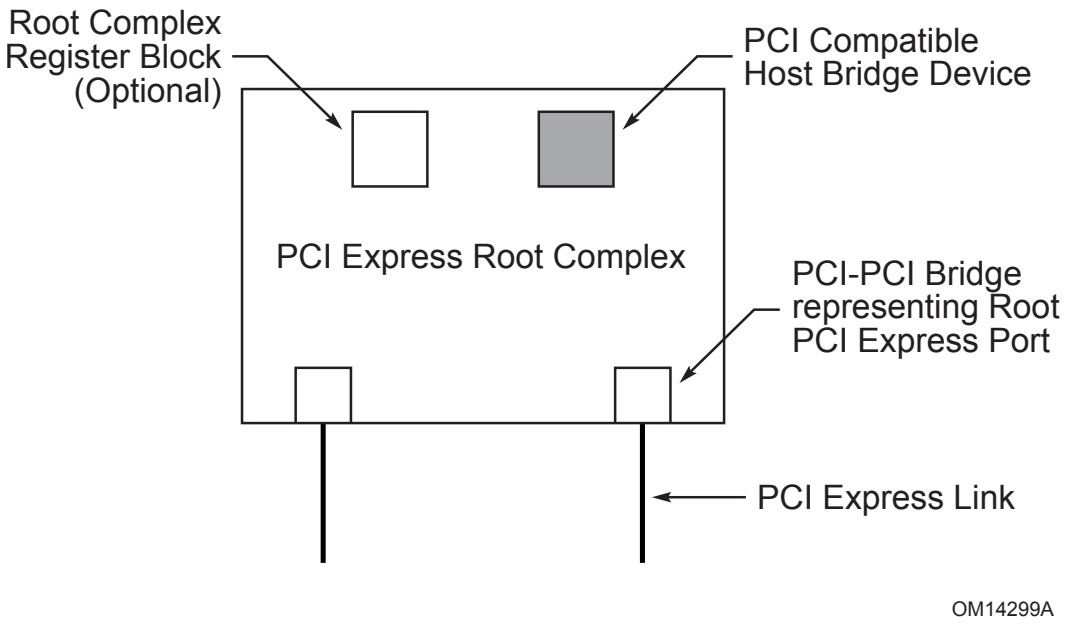

Figure 7-1 PCI Express Root Complex Device Mapping

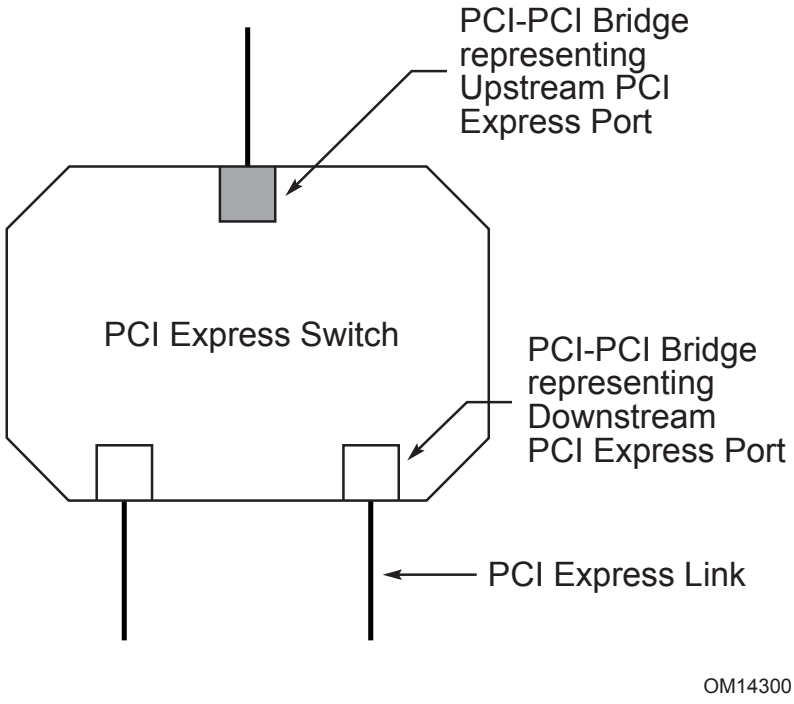

Figure 7-2 PCI Express Switch Device Mapping[130]

130. Future PCI Express Switches may be implemented as a single Switch device component (without the PCI-PCI Bridges) that is not limited by legacy compatibility requirements imposed by existing PCI software.

7.2 PCI Express Configuration Mechanisms

PCI Express extends the Configuration Space to 4096 bytes per Function as compared to 256 bytes allowed by [PCI]. PCI Express Configuration Space is divided into a PCI-compatible region, which consists of the first 256 bytes of a Function's Configuration Space, and a PCI Express Extended Configuration Space which consists of the remaining Configuration Space (see Figure 7-3). The PCI-compatible Configuration Space can be accessed using either the mechanism defined in Section 7.2.1 or Section 7.2.2 . Accesses made using either access mechanism are equivalent. The PCI Express Extended Configuration Space can only be accessed by using the ECAM mechanism defined in Section 7.2.2 .[131]

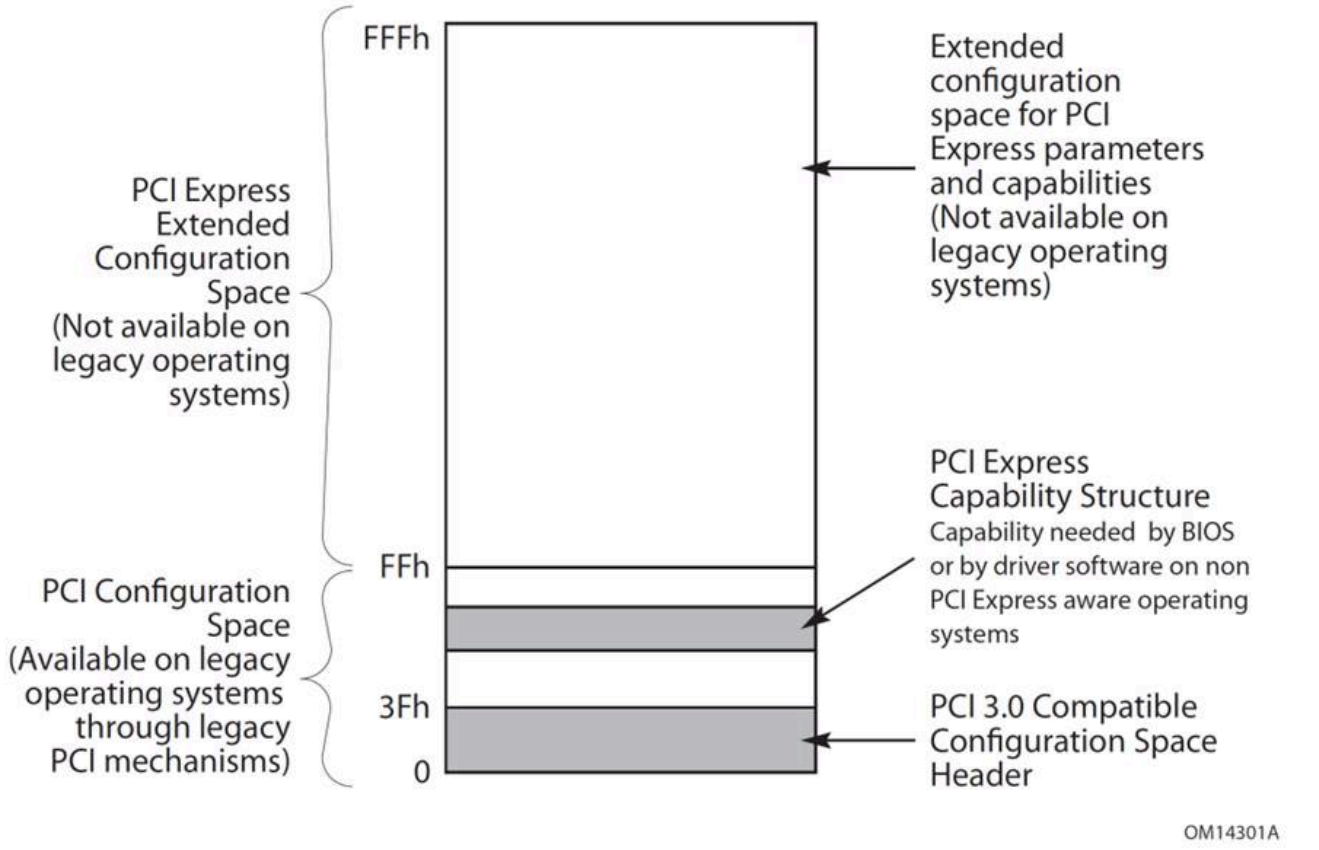

Figure 7-3 PCI Express Configuration Space Layout

7.2.1 PCI-compatible Configuration Mechanism

The PCI-compatible PCI Express Configuration Mechanism supports the PCI Configuration Space programming model defined in the [PCI]. By adhering to this model, systems incorporating PCI Express interfaces remain compliant with conventional PCI bus enumeration and configuration software.

In the same manner as PCI device Functions, PCI Express device Functions are required to provide a Configuration Space for software-driven initialization and configuration. Except for the differences described in this chapter, the PCI Express

131. The the mechanism defined in Section 7.2.1 and the ECAM mechanism defined in Section 7.2.2 and operate independently from each other; there is no implied ordering between the two.

Configuration Space headers are organized to correspond with the format and behavior defined in the [PCI] (Section 6.1).

The PCI-compatible Configuration Access Mechanism uses the same Request format as the ECAM. For PCI-compatible Configuration Requests, the Extended Register Address field must be all zeros.

7.2.2 PCI Express Enhanced Configuration Access Mechanism (ECAM)

For systems that are PC-compatible, or that do not implement a processor-architecture-specific firmware interface standard that allows access to the Configuration Space, the ***Enhanced Configuration Access Mechanism*** (***ECAM***) is required as defined in this section.

For systems that implement a processor-architecture-specific firmware interface standard that allows access to the Configuration Space, the operating system uses the standard firmware interface, and the hardware access mechanism defined in this section is not required. For example, for systems that are compliant with [DIG64], the operating system uses the SAL firmware service to access the Configuration Space.

In all systems, device drivers are encouraged to use the application programming interface (API) provided by the operating system to access the Configuration Space of its device and not directly use the hardware mechanism.

The ECAM utilizes a flat memory-mapped address space to access device configuration registers. In this case, the memory address determines the configuration register accessed and the memory data updates (for a write) or returns the contents of (for a read) the addressed register. The mapping from memory address space to PCI Express Configuration Space address is defined in Table 7-1 .

The size and base address for the range of memory addresses mapped to the Configuration Space are determined by the design of the host bridge and the firmware. They are reported by the firmware to the operating system in an implementation-specific manner. The size of the range is determined by the number of bits that the host bridge maps to the Bus Number field in the configuration address. In Table 7-1 , this number of bits is represented as n, where $1 \le n \le 8$. A host bridge that maps n memory address bits to the Bus Number field supports Bus Numbers from 0 to 2^n -1, inclusive, and the base address of the range is aligned to a $2^{(n+20)}$-byte memory address boundary. Any bits in the Bus Number field that are not mapped from memory address bits must be Clear.

For example, if a system maps three memory address bits to the Bus Number field, the following are all true:

- $n = 3$.
- Address bits A[63:23] are used for the base address, which is aligned to a 2^{23}-byte (8 MB) boundary.
- Address bits A[22:20] are mapped to bits [2:0] in the Bus Number field.
- Bits [7:3] in the Bus Number field are set to Clear.
- The system is capable of addressing Bus Numbers between 0 and 7, inclusive.

A minimum of one memory address bit ($n = 1$) must be mapped to the Bus Number field. Systems are encouraged to map additional memory address bits to the Bus Number field as needed to support a larger number of buses. Systems that support more than 4 GB of memory addresses are encouraged to map eight bits of memory address ($n = 8$) to the Bus Number field. Note that in systems that include multiple host bridges with different ranges of Bus Numbers assigned to each host bridge, the highest Bus Number for the system is limited by the number of bits mapped by the host bridge to which the highest bus number is assigned. In such a system, the highest Bus Number assigned to a particular host bridge would be greater, in most cases, than the number of buses assigned to that host bridge. In other words, for each host bridge, the number of bits mapped to the Bus Number field, n, must be large enough that the highest Bus Number assigned to each particular bridge must be less than or equal to 2^n -1 for that bridge.

In some processor architectures, it is possible to generate memory accesses that cannot be expressed in a single Configuration Request, for example due to crossing a DW aligned boundary, or because a locked access is used. A Root Complex implementation is not required to support the translation to Configuration Requests of such accesses.

Table 7-1 Enhanced Configuration Address Mapping

Memory Address[132]	PCI Express Configuration Space
A[(20+*n*-1):20]	Bus Number $1 \le n \le 8$
A[19:15]	Device Number
A[14:12]	Function Number
A[11:8]	Extended Register Number
A[7:2]	Register Number
A[1:0]	Along with size of the access, used to generate Byte Enables

Note: for Requests targeting Extended Functions in an ARI Device, A[19:12] represents the (8-bit) Function Number, which replaces the (5-bit) Device Number and (3-bit) Function Number fields above.

The system hardware must provide a method for the system software to guarantee that a write transaction using the ECAM is completed by the completer before system software execution continues.

132. This address refers to the byte-level address from a software point of view.

IMPLEMENTATION NOTE

Ordering Considerations for the Enhanced Configuration Access Mechanism

The ECAM converts memory transactions from the host CPU into Configuration Requests on the PCI Express fabric. This conversion potentially creates ordering problems for the software, because writes to memory addresses are typically posted transactions but writes to Configuration Space are not posted on the PCI Express fabric.

Generally, software does not know when a posted transaction is completed by the completer. In those cases in which the software must know that a posted transaction is completed by the completer, one technique commonly used by the software is to read the location that was just written. For systems that follow the PCI ordering rules throughout, the read transaction will not complete until the posted write is complete. However, since the PCI ordering rules allow non-posted write and read transactions to be reordered with respect to each other, the CPU must wait for a non-posted write to complete on the PCI Express fabric to be guaranteed that the transaction is completed by the completer.

As an example, software may wish to configure a device Function's Base Address register by writing to the device using the ECAM, and then read a location in the memory-mapped range described by this Base Address register. If the software were to issue the memory-mapped read before the ECAM write was completed, it would be possible for the memory-mapped read to be re-ordered and arrive at the device before the Configuration Write Request, thus causing unpredictable results.

To avoid this problem, processor and host bridge implementations must ensure that a method exists for the software to determine when the write using the ECAM is completed by the completer.

This method may simply be that the processor itself recognizes a memory range dedicated for mapping ECAM accesses as unique, and treats accesses to this range in the same manner that it would treat other accesses that generate non-posted writes on the PCI Express fabric, i.e., that the transaction is not posted from the processor's viewpoint. An alternative mechanism is for the host bridge (rather than the processor) to recognize the memory-mapped Configuration Space accesses and not to indicate to the processor that this write has been accepted until the non-posted Configuration Transaction has completed on the PCI Express fabric. A third alternative would be for the processor and host bridge to post the memory-mapped write to the ECAM and for the host bridge to provide a separate register that the software can read to determine when the Configuration Write Request has completed on the PCI Express fabric. Other alternatives are also possible.

IMPLEMENTATION NOTE

Generating Configuration Requests

Because Root Complex implementations are not required to support the generation of Configuration Requests from accesses that cross DW boundaries, or that use locked semantics, software should take care not to cause the generation of such accesses when using the memory-mapped ECAM unless it is known that the Root Complex implementation being used will support the translation.

7.2.2.1 Host Bridge Requirements

For those systems that implement the ECAM, the PCI Express Host Bridge is required to translate the memory-mapped PCI Express Configuration Space accesses from the host processor to PCI Express configuration transactions. The use of Host Bridge PCI class code is Reserved for backwards compatibility; Host Bridge Configuration Space is opaque to standard PCI Express software and may be implemented in an implementation specific manner that is compatible with PCI Host Bridge Type 0 Configuration Space. A PCI Express Host Bridge is not required to signal errors through a Root Complex Event Collector. This support is optional for PCI Express Host Bridges.

7.2.2.2 PCI Express Device Requirements

Devices must support an additional 4 bits for decoding configuration register access, i.e., they must decode the Extended Register Address[3:0] field of the Configuration Request header.

IMPLEMENTATION NOTE

Device-Specific Registers in Configuration Space

It is strongly recommended that PCI Express devices place no registers in Configuration Space other than those in headers or Capability structures architected by applicable PCI specifications.

Device-specific registers that have legitimate reasons to be placed in Configuration Space (e.g., they need to be accessible before Memory Space is allocated) should be placed in a Vendor-Specific Capability structure (Section 7.9.4), a Vendor-Specific Extended Capability structure (Section 7.9.5 , or Section 7.9.6).

Device-specific registers accessed in the run-time environment by drivers should be placed in Memory Space that is allocated by one or more Base Address registers. Even though PCI-compatible or PCI Express Extended Configuration Space may have adequate room for run-time device-specific registers, placing them there is highly discouraged for the following reasons:

- Not all Operating Systems permit drivers to access Configuration Space directly.
- Some platforms provide access to Configuration Space only via firmware calls, which typically have substantially lower performance compared to mechanisms for accessing Memory Space.
- Even on platforms that provide direct access to a memory-mapped PCI Express Enhanced Configuration Mechanism, performance for accessing Configuration Space will typically be significantly lower than for accessing Memory Space since:
  - Configuration Reads and Writes must usually be DWORD or smaller in size,
  - Configuration Writes are usually not posted by the platform, and
  - Some platforms support only one outstanding Configuration Write at a time.

IMPLEMENTATION NOTE

Configuration Space Read Side Effects

During a read access, any observable interaction that occurs besides the desired value being returned is called a read side effect. System software that has no specific knowledge of the Function being accessed may issue read requests to anywhere within the Function's Configuration Space. It is highly undesirable that any such access has any read side effects. No such side effects are required in any of the Configuration Space registers defined in this specification. It is strongly recommended that any implementation of those registers, as well as any vendor-defined Configuration Space registers, be free of any read side effects.

7.2.3 *Root Complex Register Block (RCRB)*

A Root Port or RCiEP may be associated with an optional 4096-byte block of memory mapped registers referred to as the Root Complex Register Block (RCRB). These registers are used in a manner similar to Configuration Space and can include PCI Express Extended Capabilities and other implementation specific registers that apply to the Root Complex. The structure of the RCRB is described in Section 7.6.2 .

Multiple Root Ports or internal devices are permitted to be associated with the same RCRB. The RCRB memory-mapped registers must not reside in the same address space as the memory-mapped Configuration Space or Memory Space.

A Root Complex implementation is not required to support accesses to an RCRB that cross DWORD aligned boundaries or accesses that use locked semantics.

IMPLEMENTATION NOTE

Accessing Root Complex Register Block

Because Root Complex implementations are not required to support accesses to a RCRB that cross DW boundaries, or that use locked semantics, software should take care not to cause the generation of such accesses when accessing a RCRB unless the Root Complex will support the access.

7.3 Configuration Transaction Rules

7.3.1 Device Number

With non-ARI Devices, PCI Express components are restricted to implementing a single Device Number on their primary interface (Upstream Port), but are permitted to implement up to eight independent Functions within that Device Number. Each internal Function is selected based on decoded address information that is provided as part of the address portion of Configuration Request packets.

Except when FPB Routing ID mechanisms are used (see Section 6.27), Downstream Ports that do not have ARI Forwarding enabled must associate only Device 0 with the device attached to the Logical Bus representing the Link from the Port. Configuration Requests targeting the Bus Number associated with a Link specifying Device Number 0 are delivered to the device attached to the Link; Configuration Requests specifying all other Device Numbers (1-31) must be terminated by the Switch Downstream Port or the Root Port with an Unsupported Request Completion Status (equivalent to Master Abort in PCI).

Non-ARI Devices must capture their assigned Device Number as discussed in Section 2.2.6.2 . Non-ARI Devices must respond to all Type 0 Configuration Read Requests, regardless of the Device Number specified in the Request.

Switches, and components wishing to incorporate more than eight Functions at their Upstream Port, are permitted to implement one or more “virtual switches” represented by multiple Type 1 Configuration Space Headers (PCI-PCI Bridge) as illustrated in Figure 7-2 . These virtual switches serve to allow fan-out beyond eight Functions. FPB provides a “flattening” mechanism that, when enabled, causes the virtual bridges of the Downstream Ports to appear in configuration space at RID addresses following the RID of the Upstream Port (see Section 6.27). Since Switch Downstream Ports are permitted to appear on any Device Number, in this case all address information fields (Bus, Device, and Function Numbers) must be completely decoded to access the correct register. Any Configuration Request targeting an unimplemented Bus, Device, or Function must return a Completion with Unsupported Request Completion Status.

With an ARI Device, its Device Number is implied to be 0 rather than specified by a field within an ID. The traditional 5-bit Device Number and 3-bit Function Number fields in its associated Routing IDs, Requester IDs, and Completer IDs are interpreted as a single 8-bit Function Number. See Section 6.13 . Any Type 0 Configuration Request targeting an unimplemented Function in an ARI Device must be handled as an Unsupported Request.

If an ARI Downstream Port has ARI Forwarding enabled, the logic that determines when to turn a Type 1 Configuration Request into a Type 0 Configuration Request no longer enforces a restriction on the traditional Device Number field being 0.

The following section provides details of the Configuration Space addressing mechanism.

7.3.2 Configuration Transaction Addressing

PCI Express Configuration Requests use the following addressing fields:

- Bus Number - PCI Express maps logical PCI Bus Numbers onto PCI Express Links such that PCI-compatible configuration software views the Configuration Space of a PCI Express Hierarchy as a PCI hierarchy including multiple bus segments.
- Device Number - Device Number association is discussed in Section 7.3.1 and in Section 6.27 . When an ARI Device is targeted and the Downstream Port immediately above it is enabled for ARI Forwarding, the Device Number is implied to be 0, and the traditional Device Number field is used instead as part of an 8-bit Function Number field. See Section 6.13 .
- Function Number - PCI Express also supports Multi-Function Devices using the same discovery mechanism as PCI. A Multi-Function Device must fully decode the Function Number field. It is strongly recommended that a single-Function Device also fully decode the Function Number field. With ARI Devices, discovery and enumeration of Extended Functions require ARI-aware software. See Section 6.13 .
- Extended Register Number and Register Number - Specify the Configuration Space address of the register being accessed (concatenated such that the Extended Register Number forms the more significant bits).

7.3.3 Configuration Request Routing Rules

For Endpoints, the following rules apply:

- If Configuration Request Type is 1,
  - Follow the rules for handling Unsupported Requests
- If Configuration Request Type is 0,
  - Determine if the Request addresses a valid local Configuration Space of an implemented Function

        - If so, process the Request
        - If not, follow rules for handling Unsupported Requests

For Root Ports, Switches, and PCI Express-PCI Bridges, the following rules apply:

- Propagation of Configuration Requests from Downstream to Upstream as well as peer-to-peer are not supported
    - Configuration Requests are initiated only by the Host Bridge, including those passed through the SFI CAM mechanism
- If Configuration Request Type is 0,
    - Determine if the Request addresses a valid local Configuration Space of an implemented Function
        - If so, process the Request
        - If not, follow rules for handling Unsupported Requests
- If Configuration Request Type is 1, apply the following tests, in sequence, to the Bus Number and Device Number fields:
    - If in the case of a PCI Express-PCI Bridge, equal to the Bus Number assigned to secondary PCI bus or, in the case of a Switch or Root Complex, equal to the Bus Number and decoded Device Numbers assigned to one of the Root (Root Complex) or Downstream Ports (Switch), or if required based on the FPB Routing ID mechanism,
        - Transform the Request to Type 0 by changing the value in the Type[4:0] field of the Request (see Table 2-3) - all other fields of the Request remain unchanged
        - Forward the Request to that Downstream Port (or PCI bus, in the case of a PCI Express-PCI Bridge)
    - If not equal to the Bus Number of any of Downstream Ports or secondary PCI bus, but in the range of Bus Numbers assigned to either a Downstream Port or a secondary PCI bus, or if required based on the FPB Routing ID mechanism,
        - Forward the Request to that Downstream Port interface without modification
    - Else (none of the above)
        - The Request is invalid - follow the rules for handling Unsupported Requests
- PCI Express-PCI Bridges must terminate as Unsupported Requests any Configuration Requests for which the Extended Register Address field is non-zero that are directed towards a PCI bus that does not support Extended Configuration Space.

Note: This type of access is a consequence of a programming error.

Additional rule specific to Root Complexes:

- Configuration Requests addressing Bus Numbers assigned to devices within the Root Complex are processed by the Root Complex
    - The assignment of Bus Numbers to the devices within a Root Complex may be done in an implementation specific way.

For all types of devices:

Configuration Reads and Writes to unimplemented registers are not considered to be errors. Unless errors defined elsewhere in this specification are detected and need to be reported, such Requests must return a Completion with Successful Completion status, with reads returning a data value of all 0's and writes discarding the write data without effect.

All other Configuration Space addressing fields are decoded as described elsewhere in this specification.

7.3.4 PCI Special Cycles

PCI Special Cycles (see the [PCI] for details) are not directly supported by PCI Express. PCI Special Cycles may be directed to PCI bus segments behind PCI Express-PCI Bridges using Type 1 configuration cycles as described in the [PCI].

7.4 Configuration Register Types

Configuration register fields are assigned one of the attributes described in Table 7-2 . All PCI Express components, with the exception of the Root Complex and system-integrated devices, initialize register fields to specified default values. Root Complexes and system-integrated devices initialize register fields as required by the firmware for a particular system implementation.

Table 7-2 Register and Register Bit-Field Types

Register Attribute	Description
HwInit	**Hardware Initialized** - Register bits are permitted, as an implementation option, to he hard-coded, initialized by system/device firmware, or initialized by hardware mechanisms such as pin strapping or nonvolatile storage.[133] Initialization by system firmware is permitted only for system-integrated devices. Bits must be fixed in value and read-only after initialization. After Initialization, values are only permitted to change following Conventional Reset (see Section 6.6.1) and subsequent re-initialization. HwInit register bits are not modified by an FLR.
RO	**Read-only** - Register bits are read-only and cannot be altered by software. Where explicitly defined, these bits are used to reflect changing hardware state, and as a result bit values can be observed to change at run time.[134] Register bit default values and bits that cannot change value at run time, are permitted to be hard-coded, initialized by system/ device firmware, or initialized by hardware mechanisms such as pin strapping or nonvolatile storage. Initialization by system firmware is permitted only for system-integrated devices. If the optional feature that would Set the bits is not implemented, the bits are hardwired to 0b.
RW	**Read-Write** - Register bits are read-write and are permitted to be either Set or Cleared by software to the desired state. If the optional feature that is associated with the bits is not implemented, the bits are permitted to be hardwired to 0b.
RW1C	**Write-1-to-clear status** - Register bits indicate status when read. A Set bit indicates a status event which is Cleared by writing a 1b. Writing a 0b to RW1C bits has no effect. If the optional feature that would Set the bit is not implemented, the bit is read-only and hardwired to 0b.
ROS	**Sticky - Read-only** - Register bits are read-only and cannot be altered by software. If the optional feature that would Set the bit is not implemented, the bit is hardwired to 0b. Bits are neither initialized nor modified by hot reset or FLR.[135] Where noted, devices that consume auxiliary power must preserve sticky register bit values when auxiliary power consumption (via either Aux Power PM Enable or PME_En) is enabled. In these cases, register bits are neither initialized nor modified by hot, warm, or cold reset (see Section 6.6).

133. For historical reasons, readers may observe inconsistencies in this document in the use of HwInit and RO. As this document is revised we will attempt to ensure that new definitions conform to the definitions given here.
134. For historical reasons, readers may observe inconsistencies in this document in the use of HwInit and RO. As this document is revised we will attempt to ensure that new definitions conform to the definitions given here.
135. Bits/fields with the “Sticky” attribute must be implemented such that no Function-specific software or firmware is required to maintain the observed state of the bit/field. Particularly for power management scenarios, it is permitted, but not recommended, to use Function-specific software or firmware to restore the correct values, provided this is done before the system hardware or system software could observe incorrect values. How this could be done is outside the scope of this document.

Register Attribute	Description
RWS	**Sticky - Read-Write** - Register bits are read-write and are Set or Cleared by software to the desired state. Bits are neither initialized nor modified by hot reset or FLR.[136] If the optional feature that is associated with the bits is not implemented, the bits are permitted to be hardwired to 0b. Where noted, devices that consume auxiliary power must preserve sticky register bit values when auxiliary power consumption (via either Aux Power PM Enable or PME_En) is enabled. In these cases, register bits are neither initialized nor modified by hot, warm, or cold reset (see Section 6.6).
RW1CS	**Sticky - Write-1-to-clear status** - Register bits indicate status when read. A Set bit indicates a status event which is Cleared by writing a 1b. Writing a 0b to RW1CS bits has no effect. If the optional feature that would Set the bit is not implemented, the bit is read-only and hardwired to 0b. Bits are neither initialized nor modified by hot reset or FLR.[137] Where noted, devices that consume auxiliary power must preserve sticky register bit values when auxiliary power consumption (via either Aux Power PM Enable or PME_En) is enabled. In these cases, register bits are neither initialized nor modified by hot, warm, or cold reset (see Section 6.6).
RsvdP	**Reserved and Preserved** - Reserved for future RW implementations. Register bits are read-only and must return zero when read. Software must preserve the value read for writes to bits.
RsvdZ	**Reserved and Zero** - Reserved for future RW1C implementations. Register bits are read-only and must return zero when read. Software must use 0b for writes to bits.

7.5 PCI and PCIe Capabilities Required by the Base Spec for all Ports

The following registers and capabilities are required by this specification in all Functions.

7.5.1 PCI-Compatible Configuration Registers

The first 256 bytes of a Function's Configuration Space form the PCI-compatible region. This region completely aliases the conventional PCI Configuration Space of the Function. Legacy PCI devices can also be accessed with the ECAM without requiring any modifications to the device hardware or device driver software.

Layout of the Configuration Space and format of individual configuration registers are depicted following the little-endian convention.

7.5.1.1 Type 0/1 Common Configuration Space

Figure 7-4 details allocation for common register fields of Type 0 and Type 1 Configuration Space Headers for PCI Express Device Functions. Fields labeled Type Specific vary between different Configuration Space header types.

136. Bits/fields with the "Sticky" attribute must be implemented such that no Function-specific software or firmware is required to maintain the observed state of the bit/field. Particularly for power management scenarios, it is permitted, but not recommended, to use Function-specific software or firmware to restore the correct values, provided this is done before the system hardware or system software could observe incorrect values. How this could be done is outside the scope of this document.
137. Bits/fields with the "Sticky" attribute must be implemented such that no Function-specific software or firmware is required to maintain the observed state of the bit/field. Particularly for power management scenarios, it is permitted, but not recommended, to use Function-specific software or firmware to restore the correct values, provided this is done before the system hardware or system software could observe incorrect values. How this could be done is outside the scope of this document.

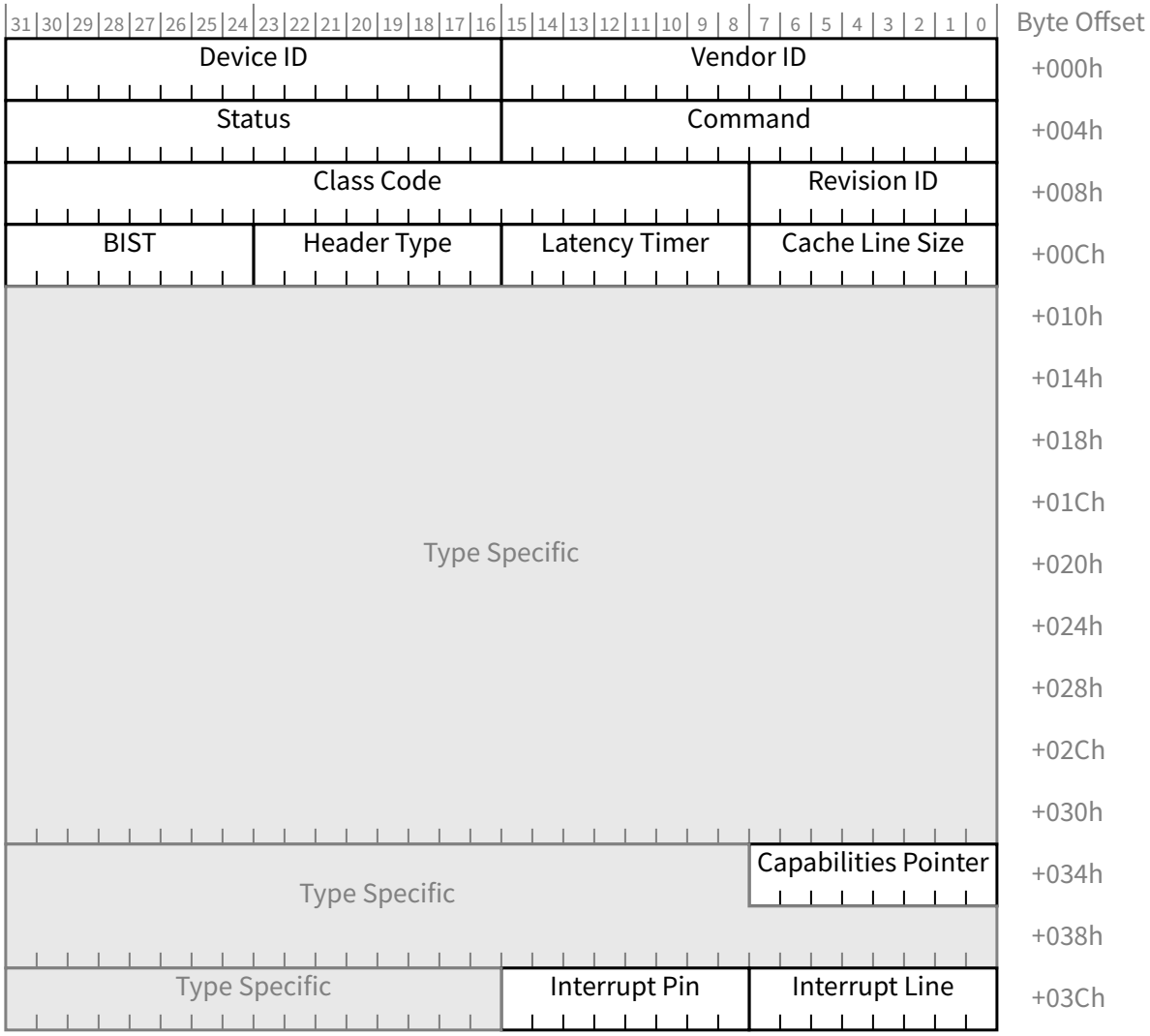

Figure 7-4 Common Configuration Space Header

These registers are defined for both Type 0 and Type 1 Configuration Space Headers. The PCI Express-specific interpretation of these registers is defined in this section.

7.5.1.1.1 *Vendor ID Register* (Offset 00h)

The Vendor ID register is HwInit and the value in this register identifies the manufacturer of the Function. In keeping with PCI-SIG procedures, valid vendor identifiers must be allocated by the PCI-SIG to ensure uniqueness. Each vendor must have at least one Vendor ID. It is recommended that software read the Vendor ID register to determine if a Function is present, where a value of FFFFh indicates that no Function is present.

7.5.1.1.2 *Device ID Register* (Offset 02h)

The Device ID register is HwInit and the value in this register identifies the particular Function. The Device ID must be allocated by the vendor. The Device ID, in conjunction with the Vendor ID and Revision ID, are used as one mechanism for software to determine which driver should be loaded. The vendor must ensure that the chosen values do not result in the use of an incompatible device driver.

7.5.1.1.3 *Command Register* (Offset 04h)

Table 7-3 defines the Command Register and the layout of the register is shown in Figure 7-5 . Individual bits in the Command Register may or may not be implemented depending on the feature set supported by the Function. For PCI Express to PCI/PCI-X Bridges, refer to the [PCIe-to-PCI-PCI-X-Bridge] for requirements for this register.

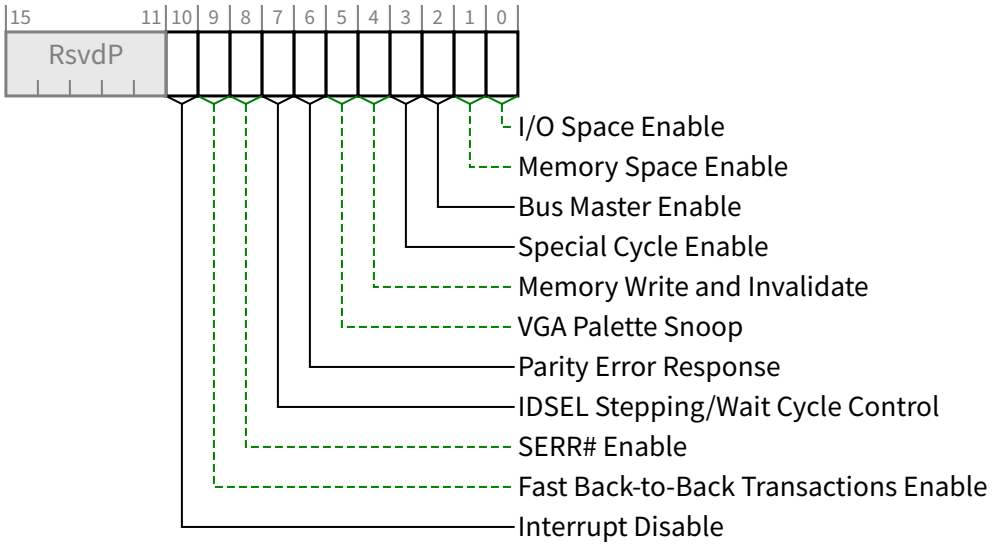

Figure 7-5 Command Register

Table 7-3 Command Register

Bit Location	Register Description	Attributes
0	***I/O Space Enable*** - Controls a Function's response to I/O Space accesses. When this bit is Clear, all received I/O accesses are caused to be handled as Unsupported Requests. When this bit is Set, the Function is enabled to decode the address and further process I/O Space accesses. For a Function with a Type 1 Configuration Space Header, this bit controls the response to I/O Space accesses received on its Primary Side. Default value of this bit is 0b. This bit is permitted to be hardwired to 0b if a Function does not support I/O Space accesses.	RW
1	***Memory Space Enable*** - Controls a Function's response to Memory Space accesses. When this bit is Clear, all received Memory Space accesses are caused to be handled as Unsupported Requests. When this bit is Set, the Function is enabled to decode the address and further process Memory Space	RW

Bit Location	Register Description	Attributes
	accesses. For a Function with a Type 1 Configuration Space Header, this bit controls the response to Memory Space accesses received on its Primary Side. Default value of this bit is 0b. This bit is permitted to be hardwired to 0b if a Function does not support Memory Space accesses.	
2	***Bus Master Enable*** - Controls the ability of a Function to issue Memory[138] and I/O Read/Write Requests, and the ability of a Port to forward Memory and I/O Read/Write Requests in the Upstream direction • **Functions with a Type 0 Configuration Space Header:** When this bit is Set, the Function is allowed to issue Memory or I/O Requests. When this bit is Clear, the Function is not allowed to issue any Memory or I/O Requests. Note that as MSI/MSI-X interrupt Messages are in-band memory writes, setting the Bus Master Enable bit to 0b disables MSI/MSI-X interrupt Messages as well. Requests other than Memory or I/O Requests are not controlled by this bit. Default value of this bit is 0b. This bit is hardwired to 0b if a Function does not generate Memory or I/O Requests. • **Functions with a Type 1 Configurations Space Header:** This bit controls forwarding of Memory or I/O Requests by a Port in the Upstream direction. When this bit is 0b, Memory and I/O Requests received at a Root Port or the Downstream side of a Switch Port must be handled as Unsupported Requests (UR), and for Non-Posted Requests a Completion with UR Completion Status must be returned. This bit does not affect forwarding of Completions in either the Upstream or Downstream direction. The forwarding of Requests other than Memory or I/O Requests is not controlled by this bit. Default value of this bit is 0b.	RW
3	***Special Cycle Enable*** - This bit was originally described in the [PCI]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
4	***Memory Write and Invalidate*** - This bit was originally described in the [PCI] and the [PCI-to-PCI-Bridge]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b. For PCI Express to PCI/PCI-X Bridges, refer to the [PCIe-to-PCI-PCI-X-Bridge] for requirements for this register.	RO
5	***VGA Palette Snoop*** - This bit was originally described in the [PCI] and the [PCI-to-PCI-Bridge]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
6	***Parity Error Response*** - See Section 7.5.1.1.14 . This bit controls the logging of poisoned TLPs in the Master Data Parity Error bit in the Status Register. An RCiEP that is not associated with a Root Complex Event Collector is permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RW
7	***IDSEL Stepping/Wait Cycle Control*** - This bit was originally described in the [PCI]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
8	***SERR# Enable*** - See Section 7.5.1.1.14 .	RW

138. The AtomicOp Requester Enable bit in the Device Control 2 register must also be Set in order for an AtomicOp Requester to initiate AtomicOp Requests, which are Memory Requests.

Bit Location	Register Description	Attributes
	When Set, this bit enables reporting upstream of Non-fatal and Fatal errors detected by the Function. Note that errors are reported if enabled either through this bit or through the PCI Express specific bits in the Device Control Register (see Section 7.5.3.4). In addition, for Functions with Type 1 Configuration Space Headers, this bit controls transmission by the primary interface of ERR_NONFATAL and ERR_FATAL error Messages forwarded from the secondary interface. This bit does not affect the transmission of forwarded ERR_COR messages. An RCiEP that is not associated with a Root Complex Event Collector is permitted to hardwire this bit to 0b. Default value of this bit is 0b.	
9	***Fast Back-to-Back Transactions Enable*** - This bit was originally described in the [PCI]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
10	***Interrupt Disable*** - Controls the ability of a Function to generate INTx emulation interrupts. When Set, Functions are prevented from asserting INTx interrupts. Any INTx emulation interrupts already asserted by the Function must be deasserted when this bit is Set. As described in Section 2.2.8.1 , INTx interrupts use virtual wires that must, if asserted, be deasserted using the appropriate Deassert_INTx message(s) when this bit is Set. Only the INTx virtual wire interrupt(s) associated with the Function(s) for which this bit is Set are affected. For Functions with a Type 0 Configuration Space Header that generate INTx interrupts, this bit is required. For Functions with a Type 0 Configuration Space Header that do not generate INTx interrupts, this bit is optional. If not implemented, this bit must be hardwired to 0b. For Functions with a Type 1 Configuration Space Header that generate INTx interrupts on their own behalf, this bit is required. This bit has no effect on interrupts forwarded from the secondary side. For Functions with a Type 1 Configuration Space Header that do not generate INTx interrupts on their own behalf this bit is optional. If not implemented, this bit must be hardwired to 0b. Default value of this bit is 0b.	RW

7.5.1.1.4 *Status Register* (Offset 06h)

Table 7-4 defines the Status Register and the layout of the register is shown in Figure 7-6 . Functions may not need to implement all bits, depending on the feature set supported by the Function. For PCI Express to PCI/PCI-X Bridges, refer to the [PCIe-to-PCI-PCI-X-Bridge] for requirements for this register.

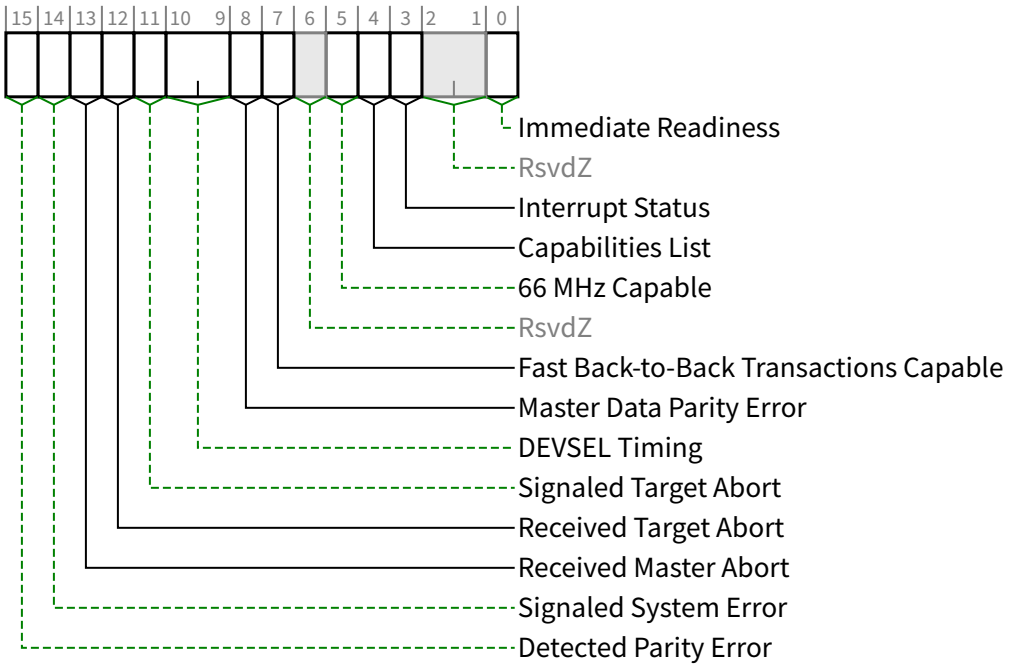

Figure 7-6 Status Register

Table 7-4 Status Register

Bit Location	Register Description	Attributes
0	***Immediate Readiness*** - This optional bit, when Set, indicates the Function is guaranteed to be ready to successfully complete valid Configuration Requests at any time. It is permitted for this indication to be based on implementation-specific knowledge of how long it takes the host to become ready to issue Configuration Requests. When this bit is Set, for accesses to this Function, software is exempt from all requirements to delay configuration accesses following any type of reset, including but not limited to the timing requirements defined in Section 6.6 . How this guarantee is established is beyond the scope of this document. It is permitted that system software/firmware provide mechanisms that supersede the indication provided by this bit, however such software/firmware mechanisms are outside the scope of this specification.	RO
3	***Interrupt Status*** - When Set, indicates that an INTx emulation interrupt is pending internally in the Function. Note that INTx emulation interrupts forwarded by Functions with a Type 1 Configuration Space Header from the secondary side are not reflected in this bit. Setting the Interrupt Disable bit has no effect on the state of this bit. Functions that do not generate INTx interrupts are permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RO

Bit Location	Register Description	Attributes
4	***Capabilities List*** - Indicates the presence of an Extended Capability list item. Since all PCI Express device Functions are required to implement the PCI Express Capability structure, this bit must be hardwired to 1b.	RO
5	***66 MHz Capable*** - This bit was originally described in the [PCI]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
7	***Fast Back-to-Back Transactions Capable*** - This bit was originally described in the [PCI]. Its functionality does not apply to PCI Express and the bit must be hardwired to 0b.	RO
8	***Master Data Parity Error*** - See Section 7.5.1.1.14 This bit is Set by a Function with a Type 0 Configuration Space Header if the Parity Error Response bit in the Command Register is 1b and either of the following two conditions occurs: • Function receives a Poisoned Completion • Function transmits a Poisoned Request This bit is Set by a Function with a Type 1 Configuration Space Header if the Parity Error Response bit in the Command Register is 1b and either of the following two conditions occurs: • Port receives a Poisoned Completion going Downstream • Port transmits a Poisoned Request Upstream If the Parity Error Response bit is 0b, this bit is never Set. Default value of this bit is 0b.	RW1C
10:9	***DEVSEL Timing*** - This field was originally described in the [PCI]. Its functionality does not apply to PCI Express and the field must be hardwired to 00b.	RO
11	***Signaled Target Abort*** - See Section 7.5.1.1.14 . This bit is Set when a Function completes a Posted or Non-Posted Request as a Completer Abort error. This applies to a Function with a Type 1 Configuration Space Header when the Completer Abort was generated by its Primary Side. Functions with a Type 0 Configuration Space Header that do not signal Completer Abort are permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RW1C
12	***Received Target Abort*** - See Section 7.5.1.1.14 . This bit is Set when a Requester receives a Completion with Completer Abort Completion Status. On a Function with a Type 1 Configuration Space Header, the bit is Set when the Completer Abort is received by its Primary Side. Functions with a Type 0 Configuration Space Header that do not make Non-Posted Requests on their own behalf are permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RW1C
13	***Received Master Abort*** - See Section 7.5.1.1.14 . This bit is Set when a Requester receives a Completion with Unsupported Request Completion Status. On a Function with a Type 1 Configuration Space Header, the bit is Set when the Unsupported Request is received by its Primary Side. Functions with a Type 0 Configuration Space Header that do not make Non-Posted Requests on their own behalf are permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RW1C

Bit Location	Register Description	Attributes
14	***Signaled System Error*** - See Section 7.5.1.1.14 . This bit is Set when a Function sends an ERR_FATAL or ERR_NONFATAL Message, and the SERR# Enable bit in the Command Register is 1b. Functions with a Type 0 Configuration Space Header that do not send ERR_FATAL or ERR_NONFATAL Messages are permitted to hardwire this bit to 0b. Default value of this bit is 0b.	RW1C
15	***Detected Parity Error*** - See Section 7.5.1.1.14 . This bit is Set by a Function whenever it receives a Poisoned TLP, regardless of the state the Parity Error Response bit in the Command Register. On a Function with a Type 1 Configuration Space Header, the bit is Set when the Poisoned TLP is received by its Primary Side. Default value of this bit is 0b.	RW1C

7.5.1.1.5 *Revision ID Register* (Offset 08h)

The Revision ID Register is HwInit and the value in this register specifies a Function specific revision identifier. The value is chosen by the vendor. Zero is an acceptable value. The Device ID, in conjunction with the Vendor ID and Revision ID, are used as one mechanism for software to determine which driver should be loaded. The vendor must ensure that the chosen values do not result in the use of an incompatible device driver.

7.5.1.1.6 *Class Code Register* (Offset 09h)

The Class Code Register is read-only and is used to identify the generic operation of the Function and, in some cases, a specific register level programming interface. The register layout is shown in Figure 7-7 and described in Table 7-5 . Encodings for base class, sub-class, and programming interface are provided in the [PCI-Code-and-ID]. All unspecified encodings are Reserved.

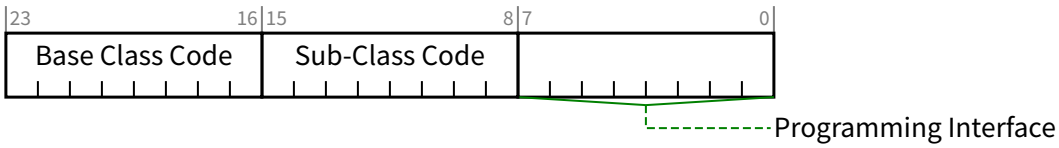

Figure 7-7 Class Code Register

Table 7-5 Class Code Register

Bit Location	Register Description	Attributes
7:0	***Programming Interface*** - This field identifies a specific register-level programming interface (if any) so that device independent software can interact with the Function. Encodings for this field are provided in the [PCI-Code-and-ID]. All unspecified encodings are Reserved.	RO
15:8	***Sub-Class Code*** - Specifies a base class sub-class, which identifies more specifically the operation of the Function.	RO

Bit Location	Register Description	Attributes
	Encodings for sub-class are provided in the [PCI-Code-and-ID]. All unspecified encodings are Reserved.	
23:16	***Base Class Code*** - A code that broadly classifies the type of operation the Function performs. Encodings for base class, are provided in the [PCI-Code-and-ID]. All unspecified encodings are Reserved.	RO

7.5.1.1.7 *Cache Line Size Register* (Offset 0Ch)

The Cache Line Size register is programmed by the system firmware or the operating system to system cache line size. However, note that legacy PCI-compatible software may not always be able to program this register correctly especially in the case of Hot-Plug devices. This read-write register is implemented for legacy compatibility purposes but has no effect on any PCI Express device behavior. For PCI Express to PCI/PCI-X Bridges, refer to the [PCIe-to-PCI-PCI-X-Bridge] for requirements for this register. The default value of this register is 00h.

7.5.1.1.8 *Latency Timer Register* (Offset 0Dh)

This register is also referred to as Primary Latency Timer for Type 1 Configuration Space Header Functions. The Latency Timer was originally described in the [PCI] and the [PCI-to-PCI-Bridge]. Its functionality does not apply to PCI Express. This register must be hardwired to 00h.

7.5.1.1.9 *Header Type Register* (Offset 0Eh)

This register identifies the layout of the second part of the predefined header (beginning at byte 10h in Configuration Space) and also whether or not the Device might contain multiple Functions. The register layout is shown in Figure 7-8 and Table 7-6 describes the bits in the register.

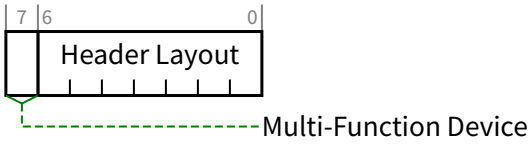

Figure 7-8 Header Type Register

Table 7-6 Header Type Register

Bit Location	Register Description	Attributes
6:0	***Header Layout*** - This field identifies the layout of the second part of the predefined header. For Functions that implement a Type 0 Configuration Space Header the encoding 000 0000b must be used. For Functions that implement a Type 1 Configuration Space Header the encoding 000 0001b must be used.	RO

Bit Location	Register Description	Attributes
	The encoding 000 0010b is Reserved. This encoding was originally described in the [PC-Card] and is used in previous versions of the programming model. Careful consideration should be given to any attempt to repurpose it. All other encodings are Reserved.	
7	***Multi-Function Device*** - When Set, indicates that the Device may contain multiple Functions, but not necessarily. Software is permitted to probe for Functions other than Function 0. When Clear, software must not probe for Functions other than Function 0 unless explicitly indicated by another mechanism, such as an ARI or SR-IOV Extended Capability structure. Except where stated otherwise, it is recommended that this bit be Set if there are multiple Functions, and Clear if there is only one Function.	RO

7.5.1.1.10 *BIST Register* (Offset 0Fh)

This register is used for control and status of BIST. Functions that do not support BIST must hardwire the register to 00h. A Function whose BIST is invoked must not prevent normal operation of the PCI Express Link. Table 7-7 describes the bits in the register and Figure 7-9 shows the register layout.

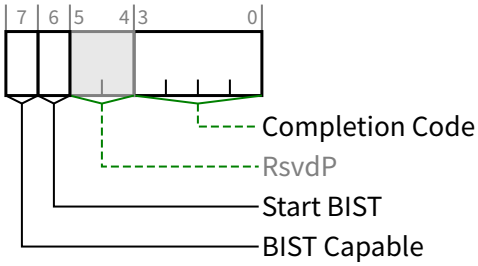

Figure 7-9 BIST Register

Table 7-7 BIST Register

Bit Location	Register Description	Attributes
3:0	***Completion Code*** - This field encodes the status of the most recent test. A value of 0000b means that the Function has passed its test. Non-zero values mean the Function failed. Function-specific failure codes can be encoded in the non-zero values. This field's value is only meaningful when BIST Capable is Set and Start BIST is Clear. Default value of this field is 0000b. This field must be hardwired to 0000b if BIST Capable is Clear.	RO
6	***Start BIST*** - If BIST Capable is Set, Set this bit to invoke BIST. The Function resets the bit when BIST is complete. Software is permitted to fail the device if this bit is not Clear (BIST is not complete) 2 seconds after it had been Set. Writing this bit to 0b has no effect. This bit must be hardwired to 0b if BIST Capable is Clear.	RW/RO (see description)

Bit Location	Register Description	Attributes
7	***BIST Capable*** - When Set, this bit indicates that the Function supports BIST. When Clear, the Function does not support BIST.	HwInit

7.5.1.1.11 *Capabilities Pointer* (Offset 34h)

This register is used to point to a linked list of capabilities implemented by this Function. Since all PCI Express Functions are required to implement both the PCI Power Management Capability and the PCI Express Capability structure, these structures must be included somewhere in the linked list; this register may point to either of these Capability Structures or to an optional Capability Structure implemented by the Function. The bottom two bits are Reserved and must be set to 00b. Software must mask these bits off before using this register as a pointer in Configuration Space to the first entry of a linked list of new capabilities.

7.5.1.1.12 *Interrupt Line Register* (Offset 3Ch)

The Interrupt Line register communicates interrupt line routing information. The register is read/write and must be implemented by any Function that uses an interrupt pin (see following description). Values in this register are programmed by system software and are system architecture specific. The Function itself does not use this value; rather the value in this register is used by device drivers and operating systems.

7.5.1.1.13 *Interrupt Pin Register* (Offset 3Dh)

The Interrupt Pin register is a read-only register that identifies the legacy interrupt Message(s) the Function uses (see Section 6.1 for further details). Valid values are 01h, 02h, 03h, and 04h that map to legacy interrupt Messages for INTA, INTB, INTC, and INTD respectively. A value of 00h indicates that the Function uses no legacy interrupt Message(s). The values 05h through FFh are Reserved.

PCI Express defines one legacy interrupt Message for a single Function device and up to four legacy interrupt Messages for a Multi-Function Device. For a single Function device, only INTA may be used.

Any Function on a Multi-Function Device can use any of the INTx Messages. If a device implements a single legacy interrupt Message, it must be INTA; if it implements two legacy interrupt Messages, they must be INTA and INTB; and so forth. For a Multi-Function Device, all Functions may use the same INTx Message or each may have its own (up to a maximum of four Functions) or any combination thereof. A single Function can never generate an interrupt request on more than one INTx Message.

7.5.1.1.14 Error Registers

The Error Control/Status register bits in the Command and Status registers (see Section 7.5.1.1.3 and Section 7.5.1.1.4 respectively) and the Bridge Control and Secondary Status registers of Type 1 Configuration Space Header Functions (see Section 7.5.1.3.10 and Section 7.5.1.3.7 respectively) control PCI-compatible error reporting for both PCI and PCI Express device Functions. Mapping of PCI Express errors onto PCI errors is also discussed in Section 6.2.7.1 . In addition to the PCI-compatible error control and status, PCI Express error reporting may be controlled separately from PCI device Functions through the PCI Express Capability structure described in Section 7.5.3 . The PCI-compatible error control and status register fields do not have any effect on PCI Express error reporting enabled through the PCI Express Capability structure. PCI Express device Functions may implement optional advanced error reporting as described in Section 7.8.4 .

For PCI Express Root Ports represented by a Type 1 Configuration Space Header:

- The primary side Error Control/Status registers apply to errors detected on the internal logic associated with the Root Complex.
- The secondary side Error Control/Status registers apply to errors detected on the Link originating from the Root Port.

For PCI Express Switch Upstream Ports represented by a Type 1 Configuration Space Header:

- The primary side Error Control/Status registers apply to errors detected on the Upstream Link of the Switch.
- The secondary side Error Control/Status registers apply to errors detected on the internal logic of the Switch.

For PCI Express Switch Downstream Ports represented by a Type 1 Configuration Space Header:

- The primary side Error Control/Status registers apply to errors detected on the internal logic of the Switch.
- The secondary side Error Control/Status registers apply to errors detected on the Downstream Link originating from the Switch Port.

7.5.1.2 Type 0 Configuration Space Header

Figure 7-10 details allocation for register fields of Type 0 Configuration Space Header for PCI Express device Functions.

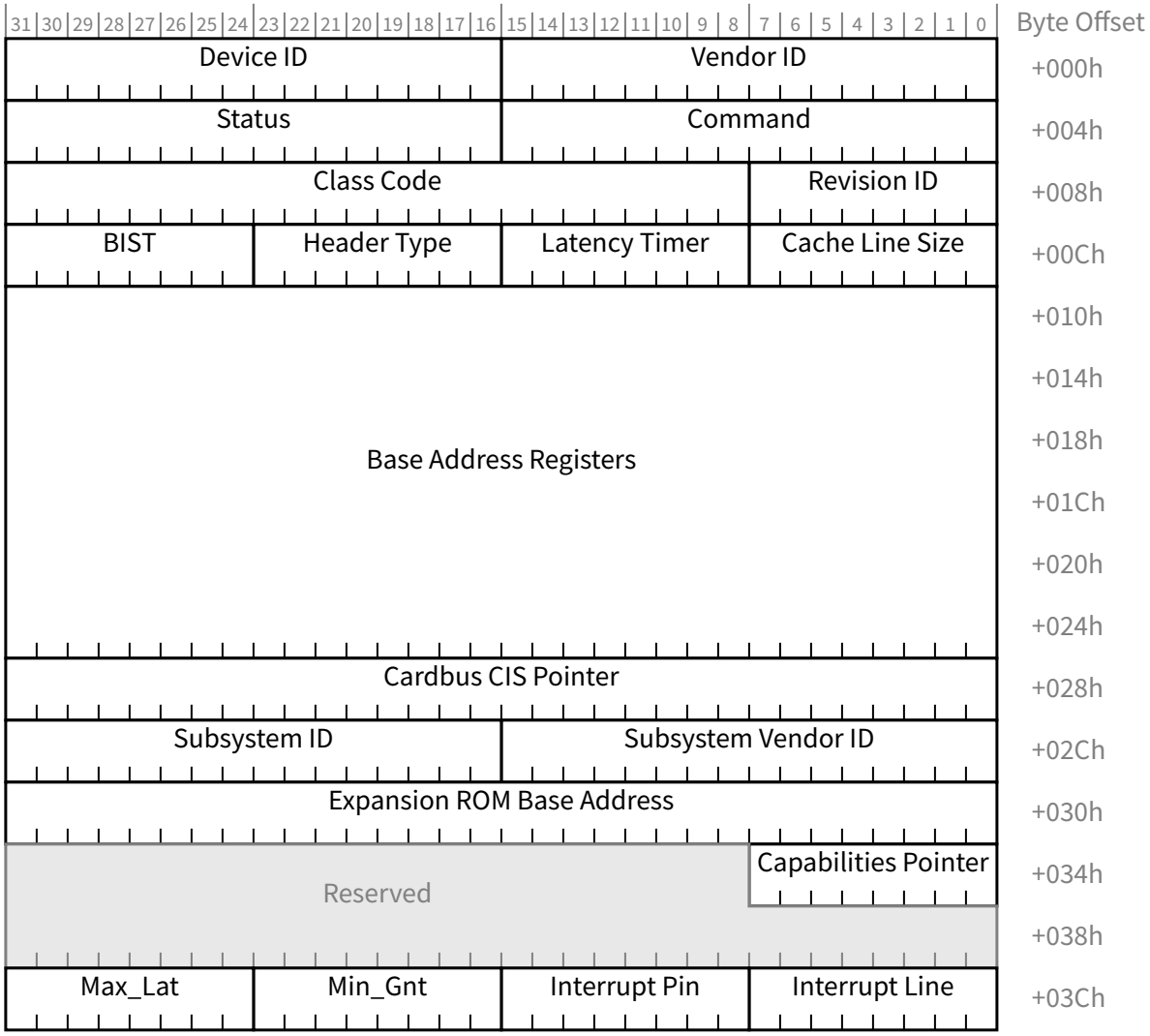

Figure 7-10 Type 0 Configuration Space Header

Section 7.5.1.1 details the PCI Express-specific registers that are valid for all Configuration Space Header types. The PCI Express-specific interpretation of registers specific to Type 0 Configuration Space Header is defined in this section.

7.5.1.2.1 *Base Address Registers* (Offset 10h - 24h)

System software must build a consistent address map before booting the machine to an operating system. This means it has to determine how much memory is in the system, and how much address space the Functions in the system require. After determining this information, system software can map the Functions into reasonable locations and proceed with system boot. In order to do this mapping in a device-independent manner, the base registers for this mapping are placed in the predefined header portion of Configuration Space. It is strongly recommended that power-up firmware/software also support the optional Enhanced Configuration Access Mechanism (ECAM).

Bit 0 in all Base Address registers is read-only and used to determine whether the register maps into Memory or I/O Space. Base Address registers that map to Memory Space must return a 0b in bit 0 (see Figure 7-11). Base Address registers that map to I/O Space must return a 1b in bit 0 (see Figure 7-12).

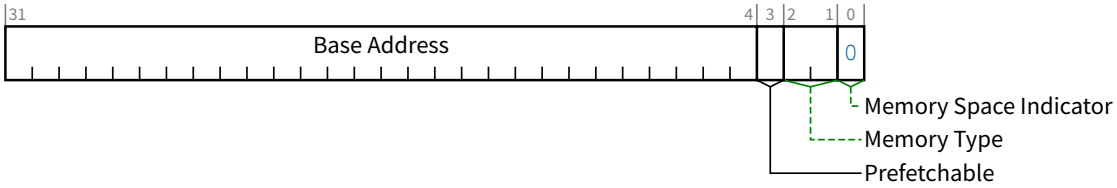

Figure 7-11 Base Address Register for Memory

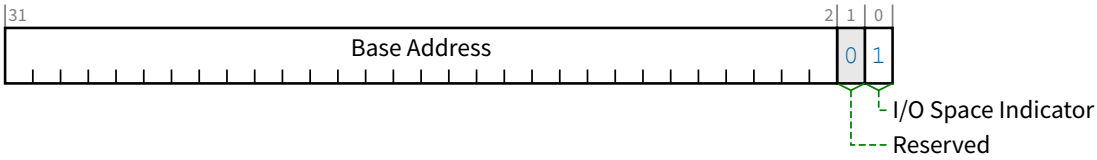

Figure 7-12 Base Address Register for I/O

Base Address registers that map into I/O Space are always 32 bits wide with bit 0 hardwired to 1b. Bit 1 is Reserved and must return 0b on reads and the other bits are used to map the Function into I/O Space.

Base Address registers that map into Memory Space can be 32 bits or 64 bits wide (to support mapping into a 64-bit address space) with bit 0 hardwired to 0b. For Memory Base Address registers, bits 2 and 1 have an encoded meaning as shown in Table 7-8 . Bit 3 should be set to 1b if the data is prefetchable and set to 0b otherwise. A Function is permitted to mark a range as prefetchable if there are no side effects on reads, the Function returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range[139] without causing errors. Bits 3-0 are read-only.

Table 7-8 Memory Base Address Register Bits 2:1 Encoding

Bits 2:1(b)	Meaning
00	Base register is 32 bits wide and can be mapped anywhere in the 32 address bit Memory Space.
01	Reserved[140]
10	Base register is 64 bits wide and can be mapped anywhere in the 64 address bit Memory Space.
11	Reserved

The number of upper bits that a Function actually implements depends on how much of the address space the Function will respond to. A 32-bit Base Address register can be implemented to support a single memory size that is a power of 2 from 16 bytes to 2 GB. A Function that wants a 1 MB memory address space (using a 32-bit Base Address register) would

139. Any device that has a range that behaves like normal memory should mark the range as prefetchable. A linear frame buffer in a graphics device is an example of a range that should be marked prefetchable.
140. The encoding to support memory space below 1 MB was supported in an earlier version of the PCI Local Bus Specification. System software should recognize this encoding and handle it appropriately.

build the top 12 bits of the address register, hardwiring the other bits to 0's. The attributes for some of the bits in the BAR are affected by the Resizable BAR Capability, if it is implemented.

Power-up software can determine how much address space the Function requires by writing a value of all 1's to the register and then reading the value back. The Function will return 0's in all don't-care address bits, effectively specifying the address space required. Unimplemented Base Address registers are hardwired to zero.

This design implies that all address spaces used are a power of two in size and are naturally aligned. Functions are free to consume more address space than required, but decoding down to a 4 KB space for memory is suggested for Functions that need less than that amount. For instance, a Function that has 64 bytes of registers to be mapped into Memory Space may consume up to 4 KB of address space in order to minimize the number of bits in the address decoder. Functions that do consume more address space than they use are not required to respond to the unused portion of that address space if the Function's programming model never accesses the unused space. The Function is permitted to return Unsupported Request for accesses targetting the unused locations. Functions that map control functions into I/O Space must not consume more than 256 bytes per I/O Base Address register or per each entry in the Enhanced Allocation Capability. The upper 16 bits of the I/O Base Address register may be hardwired to zero for Functions intended for 16-bit I/O systems, such as PC compatibles. However, a full 32-bit decode of I/O addresses must still be done.

IMPLEMENTATION NOTE

Sizing a 32-bit Base Address Register Example

Decode (I/O or memory) of the appropriate address space is disabled via the Command Register before sizing a Base Address register. Software saves the original value of the Base Address register, writes a value of all 1's to the register, then reads it back. Size calculation can be done from the 32 bit value read by first clearing encoding information bits (bits 1:0 for I/O, bits 3:0 for memory), inverting all 32 bits (logical NOT), then incrementing by 1. The resultant 32-bit value is the memory/I/O range size decoded by the register. Note that the upper 16 bits of the result is ignored if the Base Address register is for I/O and bits 31:16 returned zero upon read. The original value in the Base Address register is restored before re-enabling decode in the Command Register of the Function.

64-bit (memory) Base Address registers can be handled the same, except that the second 32 bit register is considered an extension of the first (i.e., bits 63:32). Software writes a value of all 1's to both registers, reads them back, and combines the result into a 64-bit value. Size calculation is done on the 64-bit value.

A Type 0 Configuration Space Header has six DWORD locations allocated for Base Address registers starting at offset 10h in Configuration Space. A Type 1 Configurations Space Header has only two DWORD locations. A Function may use any of the locations to implement Base Address registers. An implemented 64-bit Base Address register consumes two consecutive DWORD locations. Software looking for implemented Base Address registers must start at offset 10h and continue upwards through offset 24h. A typical Function requires one memory range for its control functions. Some graphics Functions use two ranges, one for control functions and another for a frame buffer. A Function that wants to map control functions into both memory and I/O Spaces at the same time must implement two Base Address registers (one memory and one I/O). The driver for that Function might only use one space in which case the other space will be unused. Functions are recommended to always map control functions into Memory Space.

A PCI Express Function requesting Memory Space through a BAR must set the BAR's Prefetchable bit unless the range contains locations with read side effects or locations in which the Function does not tolerate write merging. It is strongly encouraged that resources mapped into Memory Space be designed as prefetchable whenever possible. PCI Express Functions other than Legacy Endpoints must support 64-bit addressing for any Base Address register that requests prefetchable Memory Space. The minimum Memory Space address range requested by a BAR is 128 bytes. The attributes for some of the bits in the BAR are affected by the Resizable BAR Capability, if it is implemented.

IMPLEMENTATION NOTE

Additional Guidance on the Prefetchable Bit in Memory Space BARs

PCI Express adapters with Memory Space BARs that request a large amount of non-prefetchable Memory Space (e.g., over 64 MB) may cause shortages of that Space on certain scalable platforms, since many platforms support a total of only 1 GB or less of non-prefetchable Memory Space. This may limit the number of such adapters that can be supported on those platforms. For this reason, it is especially encouraged for BARs requesting large amounts of Memory Space to have their Prefetchable bit Set, since prefetchable Memory Space is more bountiful on most scalable platforms.

While a Memory Space BAR is required to have its Prefetchable bit Set if none of the locations within its range have read side effects and all of the locations tolerate write merging, there are system configurations where having the Prefetchable bit Set will still allow correct operation even if those conditions are not met. For those cases, it may make sense for the adapter to have the Prefetchable bit Set in certain candidate BARs so that the system can map those BARs into prefetchable Memory Space in order to avoid non-prefetchable Memory Space shortages.

On PCI Express systems that meet the criteria enumerated below, setting the Prefetchable bit in a candidate BAR will still permit correct operation even if the BAR's range includes some locations that have read side-effects or cannot tolerate write merging. This is primarily due to the fact that PCI Express Memory Reads always contain an explicit length, and PCI Express Switches never prefetch or do byte merging. Generally only 64-bit BARs are good candidates, since only Legacy Endpoints are permitted to set the Prefetchable bit in 32-bit BARs, and most scalable platforms map all 32-bit Memory BARs into non-prefetchable Memory Space regardless of the Prefetchable bit value.

Here are criteria that are sufficient to guarantee correctness for a given candidate BAR:

- The entire path from the host to the adapter is over PCI Express.
- No conventional PCI or PCI-X devices do peer-to-peer reads to the range mapped by the BAR.
- The PCI Express Host Bridge does no byte merging. (This is believed to be true on most platforms.)
- Any locations with read side-effects are never the target of Memory Reads with the TH bit Set. See Section 2.2.5 .
- The range mapped by the BAR is never the target of a speculative Memory Read, either Host initiated or peer-to-peer.

The above criteria are a simplified set that are sufficient to guarantee correctness. Other less restrictive but more complex criteria may also guarantee correctness, but are outside the scope of this specification.

7.5.1.2.2 *Cardbus CIS Pointer Register* (Offset 28h)

This register was originally described in the [PC-Card]. Its functionality does not apply to PCI Express. It must be read-only and hardwired to 0000 0000h.

7.5.1.2.3 *Subsystem Vendor ID Register*/*Subsystem ID Register* (Offset 2Ch/2Eh)

The Subsystem Vendor ID and Subsystem ID registers are used to uniquely identify the adapter or subsystem where the PCI Express component resides. They provide a mechanism for vendors to distinguish their products from one another even though the assemblies may have the same PCI Express component on them (and, therefore, the same Vendor ID and Device ID).

Implementation of these registers is required for all Functions except those that have a Base Class 06h with Sub Class 00h-04h (00h, 01h, 02h, 03h, 04h), or a Base Class 08h with Sub Class 00h-03h (00h, 01h, 02h, 03h). Subsystem Vendor IDs can be obtained from the PCI SIG and are used to identify the vendor of the adapter, motherboard, or subsystem[141] . A Subsystem Vendor ID (SVID) must be a Vendor ID assigned by the PCI-SIG to the vendor of the subsystem. In keeping with PCI-SIG procedures, valid vendor identifiers must be obtained from the PCI-SIG to ensure uniqueness.

Values for the Subsystem ID are vendor assigned. Subsystem ID values, in conjunction with the Subsystem Vendor ID, form a unique identifier for the PCI product. Subsystem ID and Device ID values are distinct and unrelated to each other, and software should not assume any relationship between them.

Values in these registers must be loaded before the Function becomes Configuration-Ready. How these values are loaded is not specified but could be done during the manufacturing process or loaded from external logic (e.g., strapping options, serial ROMs, etc.). These values must not be loaded using Expansion ROM software because Expansion ROM software is not guaranteed to be run during POST in all systems.

If a device is designed to be used exclusively on the system board, the system vendor may use system specific software to initialize these registers after each power-on.

IMPLEMENTATION NOTE

Subsystem Vendor ID and Subsystem ID

The Subsystem Vendor ID and Subsystem ID fields, taken together, allow software to uniquely identify a PCI circuit board product. Vendors should therefore not reuse Subsystem ID values across multiple product types that share a common Subsystem Vendor ID. It is acceptable, although not preferred, to reuse the Subsystem ID value on products with the same Subsystem Vendor ID in cases where the products are in the same family and generation, and differ only in capacity or performance. Note also that it is acceptable for vendors to use multiple unique Subsystem ID values over time for a single product type, such as to indicate some internal difference such as component selection.

7.5.1.2.4 *Expansion ROM Base Address Register* (Offset 30h)

Some Functions, especially those that are intended for use on add-in cards, require local EPROMs for Expansion ROM (refer to the [PCI-Firmware] for a definition of ROM contents). This register is defined to handle the base address and size information for this Expansion ROM. The register layout is shown in Figure 7-13 and Table 7-9 describes the bits in the register.

141. A company requires only one Vendor ID. That value can be used in either the Vendor ID register of Configuration Space (e.g., offset 00h) or the Subsystem Vendor ID register of Configuration Space (e.g., offset 2Ch). It is used in the Vendor ID register (e.g., offset 00h) if the company built the silicon. It is used in the Subsystem Vendor ID register (e.g., offset 2Ch) if the company built the assembly. If a company builds both the silicon and the assembly, then the same value would be used in both registers.