1. PCI Engineering Change Notice – MSI-X

TITLE:	MSI-X
DATE:	June 10, 2003
AFFECTED DOCUMENT(S):	PCI Local Bus Specification, Revision 2.3 & 3.0 Draft
SPONSOR:	Joe Cowan; Hewlett-Packard Company

1.1. Summary of the Functional Changes

Changes are to the PCI Local Bus Specification Revision 2.3, released March 29, 2002. The changes planned for MSI in PCI 2.3 will also be integrated into the draft PCI 3.0 specification, which has already undergone membership review.

Extend the current MSI functionality to support a larger number of MSI vectors, plus a separate and independent Message Address and Message Data for each MSI vector. Allow more flexibility when SW allocates fewer MSI vectors than requested by HW. Enable per-vector masking capability.

Compared to the MSI-X ECN that underwent membership review in September 2002, this version has the following notable changes: (1) allows the MSI-X Table to be placed into general purpose read/write memory on a device, (2) collects the per-vector Pending Bits into separate Pending Bit Array (PBA), (3) allows different MSI-X vectors to have different Upper Message Address values, (4) restricts the programming model to permit only full DWORD or QWORD transactions to access the MSI-X Table and PBA, (5) adds a new MSI capability structure with 32-bit Message Address and per-vector masking, and (6) adds Function Masking capability.

Compared to the MSI-X ECN that underwent membership review in February 2003, this version primarily has only clarifications and additional implementation notes. There is only one semantic change, that of changing the MSI-X Capability ID from 0Dh to 11h.

1.2. Benefits

1. An advanced device can deliver interrupts to any processor in an SMP platform, even when the number of processors exceeds 32 (the current MSI vector limit).
2. By supporting separate and independent Message Address/Data for each MSI vector, an advanced device can target interrupts to different processors in an SMP without relying on a re-vectoring table in the chip set.

1.3. Assessment of the Impact

Defines an optional new "MSI-X" capability structure. New devices can implement both the old and new MSI capability structures. Old SW can use the old MSI capability with complete backward compatibility. See attached pages, which show all required changes to the MSI section in Chapter 6.

1.4. Analysis of the Hardware Implications

New devices that require the extended MSI capabilities can implement the new functionality. Requires new MSI-X capability structure, MSI-X Table, Pending Bit Array, and associated control logic. Can leverage existing MSI implementation logic. Does not require use of any reserved pins, nor define any new commands.

1.5. Analysis of the Software Implications

New devices can support both old and new MSI capabilities. Old SW can continue to use old MSI capability for full backward compatibility. Requires new SW to use new MSI-X capabilities.

1.6. Additional Description and Rationale

There are at least two classes of upcoming adapters that require more flexibility in delivering interrupts over what MSI provides today. First there's the class that supports many (i.e., hundreds or thousands of) request and completion queues, with granularity down to a per-process level or finer. On SMP systems that support process affinity, it's of great benefit for a completion queue that's owned by a given process to have its associated completion interrupts delivered to the processor that typically runs that process. To support this well, an adapter function needs to support at least as many MSI vectors as there are processors in the SMP system, so that each processor can have at least one dedicated MSI vector by the adapter function. SMPs supporting 64 processors are becoming common today, and much larger ones are envisioned for the future. The current MSI limit of 32 vectors per adapter function isn't adequate to support current and future large SMP systems for this class of adapters.

InfiniBand Architecture (IBA) Host Channel Adapters (HCAs) are a good example of this adapter class. IBA HCAs typically support many thousands of completion queues (CQs). With apps doing OS-bypass with IBA messaging and/or remote DMA, each CQ is typically owned by a single process. When a process allocates a CQ, it's reasonable on an OS that supports process affinity for the driver to assign a specific MSI vector to that CQ based upon OS policy.

Another class of adapters benefiting from extended MSI functionality are those whose external links support multiple Quality-of-Service (QoS) levels, such as Ethernet NICs with special support for 802.1p. An adapter function from this class may not need a large number of distinct MSI vectors (maybe only 4 or 8), but it's important for the different MSI vectors to be directed to different processors in an SMP system so that different processors can be assigned to handling different QoS levels.

While current MSIs can be directed to different processors by appropriate re-vectoring logic in chipsets, not all chipsets implement such re-vectoring logic. Some chipsets deliver interrupts by merely "forwarding" MSIs from a PCI bus over to the processor "front-side bus" or equivalent. Having all MSIs associated with a single adapter function use the same message address means that all of the function's MSI vectors on such systems will go to the same processor. A lack of control over the full message data value can be limiting as well. Allowing different MSI vectors for the same adapter function each to have an arbitrary address and arbitrary data allows an adapter to target interrupts to different processors on such systems. Moreover, removing the dependence on non-standardized re-vectoring logic in chipsets makes it more worthwhile for adapter vendors to implement support for multiple MSI vectors per function, and OSVs to implement support for features such as interrupt assignment based upon processor affinity.

Details of MSI-X ECN Changes

For details of the changes (with changes highlighted) see attached “Section 6.8 - Message Signaled Interrupts”, as it would appear in “Chapter 6 - Configuration Space” for the Conventional PCI 2.3 Specification.

Chapter 6
Configuration Space

6.8. Message Signaled Interrupts

Message Signaled Interrupts (MSI) is an optional feature that enables a device function to request service by writing a system-specified ~~message~~ data value to a system-specified address (using a PCI DWORD memory write transaction). ~~The transaction address specifies the message destination and the transaction data specifies the message.~~ System software initializes the message ~~destination~~ address and message data (from here on referred to as the "vector") during device configuration, allocating one or more ~~non-shared message~~vectors to each MSI capable function.

Since the target of the transaction cannot distinguish between an MSI write transaction and any other write transaction, all transaction termination conditions are supported. Therefore, an MSI write transaction can be terminated with a Retry, Master-Abort, Target-Abort, or normal completion (refer to Section 3.3.3.2.).

It is recommended that devices implement interrupt pins to provide compatibility in systems that do not support MSI (devices default to interrupt pins). However, it is expected that the need for interrupt pins will diminish over time. Devices that do not support interrupt pins due to pin constraints (rely on polling for device service) may implement messages to increase performance without adding additional pins. Therefore, system configuration software must not assume that a message capable device has an interrupt pin.

Interrupt latency (the time from interrupt signaling to interrupt servicing) is system dependent. Consistent with current interrupt architectures, message signaled interrupts do not provide interrupt latency time guarantees.

MSI-X defines a separate optional extension to basic MSI functionality. Compared to MSI, MSI-X supports a larger maximum number of vectors per function, the ability for software to control aliasing when fewer vectors are allocated than requested, plus the ability for each vector to use an independent address and data value, specified by a table that resides in Memory Space. However, most of the other characteristics of MSI-X are identical to those of MSI.

MSI and MSI-X each support per-vector masking. Per-vector masking is an optional extension to MSI, and a standard feature with MSI-X. A function that supports the per-vector masking extension to MSI is still backward compatible with system software that is unaware of the extension. MSI-X also supports a Function Mask bit, which when set masks all of the vectors associated with a function.

Per-vector masking is managed through a *Mask* and *Pending* bit pair per MSI vector or MSI-X Table entry. An MSI vector is masked when its associated Mask bit is set. An MSI-X vector is masked when its associated MSI-X Table entry Mask bit or the MSI-X Function Mask bit is set. While a vector is masked, the function is prohibited from sending the associated message, and the function must set the associated Pending bit whenever the function would otherwise send the message. When software unmasks a vector whose associated Pending bit is set, the function must schedule sending the associated message, and clear the Pending bit as soon as the message has been sent.

A function is permitted to implement both MSI and MSI-X, but system software is prohibited from enabling both at the same time. If system software enables both at the same time, the result is undefined.

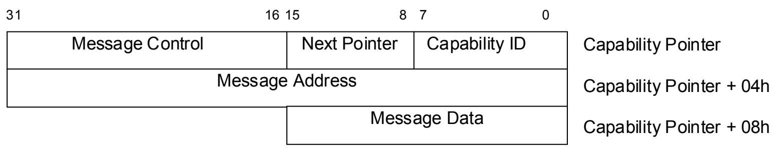
For the sake of software backward compatibility, MSI and MSI-X use separate and independent capability structures. On functions that support both MSI and MSI-X, system software that supports only MSI can still enable and use MSI without any modification. MSI functionality is managed exclusively through the MSI Capability Structure, and MSI-X functionality is managed exclusively through the MSI-X Capability Structure.

6.8.1. ~~Message~~ MSI Capability Structure

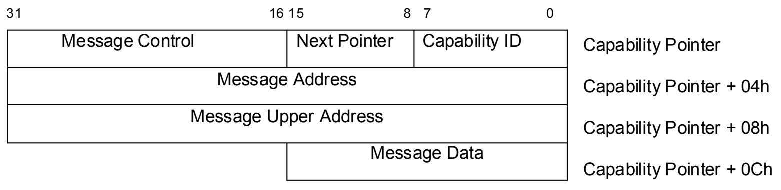
The capabilities mechanism (refer to Section 6.7.) is used to identify and configure an MSI or MSI-X capable device. The MSI capability structure is described in the current section. The MSI-X capability structure is described in Section 6.8.2.

The ~~message~~ MSI capability structure is illustrated in Figure 6-1. Each device function that supports MSI (in a multi-function device) must implement its own MSI capability structure. More ~~then~~ than one MSI capability structure per function is prohibited, but a function is permitted to have both an MSI and an MSI-X capability structure.

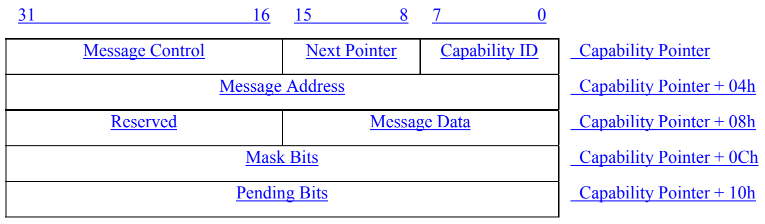
Capability Structure for 32-bit Message Address



Capability Structure for 64-bit Message Address



Capability Structure for 32-bit Message Address and Per-vector Masking



Capability Structure for 64-bit Message Address and Per-vector Masking

31 16	15 8	7 0	
Message Control	Next Pointer	Capability ID	Capability Pointer
Message Address			Capability Pointer + 04h
Message Upper Address			Capability Pointer + 08h
Reserved	Message Data		Capability Pointer + 0Ch
Mask Bits			Capability Pointer + 10h
Pending Bits			Capability Pointer + 14h

Figure 6-1: ~~Message Signaled Interrupt~~MSI Capability Structures

To request service, an MSI function writes the contents of the Message Data register to the address specified by the contents of the Message Address register (and, optionally, the Message Upper Address register for a 64-bit message address). A read of the address specified by the contents of the Message Address register produces undefined results.

A function supporting MSI implements one of four MSI Capability Structure layouts illustrated in Figure 6-1, depending upon which optional features are supported. If a function supports 64-bit addressing (DAC) when acting as a master, the function is required to implement 64-bit addressing.

~~The capability structure for a 32-bit message address (illustrated in Figure 6-1) is implemented if the function supports a 32-bit message address. The capability structure for a 64-bit message address (illustrated in Figure 6-1) is implemented if the function supports a 64-bit message address. If a device supports MSI and the device supports 64-bit addressing (DAC) when acting as a master, the device is required to implement the 64-bit message address structure.~~

The message control register indicates the function's capabilities and provides system software control over MSI.

Each field is further described in the following ~~sub-~~sections. Reserved registers and bits always return 0 when read and write operations have no effect. Read-only registers return valid data when read and write operations have no effect.

6.8.1.1. Capability ID for MSI

7::0	CAP_ID	The value of 05h in this field identifies the function as ~~message signaled interrupt~~being MSI capable. This field is read only.

6.8.1.2. Next Pointer for MSI

7::0	NXT_PTR	Pointer to the next item in the capabilities list. Must be NULL for the final item in the list. This field is read only.

6.8.1.3. Message Control for MSI

This register provides system software control over MSI. After reset, MSI is disabled. If MSI and MSI-X are both disabled, ~~(bit 0 is cleared) and~~ the function requests servicing via its **INTx#** pin (if supported). System software can enable MSI by setting bit 0 of this register. System software is permitted to modify the Message Control register's read/write bits and fields. A device driver is not permitted to modify the Message Control register's read/write bits and fields.

Bits	**Field**	**Description**
15::~~08~~09	Reserved	Always returns 0 on a read and a write operation has no effect.
8	Per-vector masking capable	If 1, the function supports MSI per-vector masking. If 0, the function does not support MSI per-vector masking. This bit is read only.
7	64 bit address capable	If 1, the function is capable of ~~generating~~ sending a 64-bit message address. If 0, the function is not capable of ~~generating~~ sending a 64-bit message address. This bit is read only.
6::4	Multiple Message Enable	software writes to this field to indicate the number of allocated ~~message~~vectors (equal to or less than the number of requested ~~message~~vectors). The number of allocated ~~message~~vectors is aligned to a power of two. If a function requests four ~~message~~vectors (indicated by a Multiple Message Capable encoding of "010"), system software can allocate either four, two, or one ~~message~~vector by writing a "010", "001, or "000" to this field, respectively. When MSI is enabled, a ~~device~~ function will be allocated at least 1 ~~message~~vector. The encoding is defined as: **Encoding # of ~~message~~vectors allocated** 000 1 001 2 010 4 011 8 100 16 101 32 110 Reserved 111 Reserved This field's state after reset is "000". This field is read/write.

Bits	Field	Description
3::1	Multiple Message Capable	System software reads this field to determine the number of requested ~~message~~vectors. The number of requested ~~message~~vectors must be aligned to a power of two (if a function requires three ~~message~~vectors, it requests four by initializing this field to "010"). The encoding is defined as: **Encoding # of ~~message~~vectors requested** 000 1 001 2 010 4 011 8 100 16 101 32 110 Reserved 111 Reserved This field is read only.
0	MSI Enable	If 1 and the MSI-X Enable bit in the MSI-X Message Control register (see Section 6.8.2.3) is 0, the function is permitted to use MSI to request service and is prohibited from using its **INTx#** pin (if implemented; see 6.2.4 Interrupt pin register). System configuration software sets this bit to enable MSI. A device driver is prohibited from writing this bit to mask a function's service request. Refer to Section 6.2.2 for control of INTx#. If 0, the function is prohibited from using MSI to request service. This bit's state after reset is 0 (MSI is disabled). This bit is read/write.

6.8.1.4. Message Address for MSI

Bits	Field	Description
31::02	Message Address	System-specified message address. If the Message Enable bit (bit 0 of the Message Control register) is set, the contents of this register specify the DWORD-aligned address (**AD[31::02])** for the MSI memory write transaction. **AD[1::0]** are driven to zero during the address phase. This field is read/write.
01::00	Reserved	Always returns 0 on read. Write operations have no effect.

6.8.1.5. Message Upper Address for MSI (Optional)

Bits	Field	Description
31::00	Message Upper Address	System-specified message upper address. This register is optional and is implemented only if the ~~device~~function supports a 64-bit message address (bit 7 in Message Control register set)[40]. If the Message Enable bit (bit 0 of the Message Control register) is set, the contents of this register (if non-zero) specify the upper 32-bits of a 64-bit message address (**AD[63::32]**). If the contents of this register are zero, the ~~device~~function uses the 32 bit address specified by the message address register. This field is read/write.

[40] This register is required when the device supports 64-bit addressing (DAC) when acting as a master.

6.8.1.6. Message Data for MSI

Bits	Field	Description
15::00	Message Data	System-specified message data. ~~Each MSI function is allocated up to 32 unique messages.~~ ~~System architecture specifies the number of unique messages supported by the system.~~ If the Message Enable bit (bit 0 of the Message Control register) is set, the message data is driven onto the lower word (**AD[15::00]**) of the memory write transaction's data phase. **AD[31::16]** are driven to zero during the memory write transaction's data phase. **C/BE[3::0]#** are asserted during the data phase of the memory write transaction. The Multiple Message Enable field (bits 6-4 of the Message Control register) defines the number of low order message data bits the function is permitted to modify to generate its system software allocated ~~message~~vectors. For example, a Multiple Message Enable encoding of "010" indicates the function has been allocated four ~~message~~vectors and is permitted to modify message data bits 1 and 0 (a function modifies the lower message data bits to generate the allocated number of ~~message~~vectors). If the Multiple Message Enable field is "000", the function is not permitted to modify the message data. This field is read/write.

6.8.1.7. Mask Bits for MSI (Optional)

The Mask Bits and Pending Bits registers enable software to disable or defer message sending on a per-vector basis.

MSI vectors are numbered 0 through N-1, where N is the number of vectors allocated by software. Each vector is associated with a correspondingly numbered bit in the Mask Bits and Pending Bits registers.

The Multiple Message Capable field indicates how many vectors (with associated Mask and Pending bits) are implemented. All unimplemented Mask and Pending bits are reserved.

After reset, the state of all implemented Mask and Pending bits is 0 (no vectors are masked and no messages are pending).

Bits	Field	Description
31::00	Mask Bits	For each Mask bit that is set, the function is prohibited from sending the associated message. This field is read/write.

6.8.1.8. Pending Bits for MSI (Optional)

Bits	Field	Description
31::00	Pending Bits	For each Pending bit that is set, the function has a pending associated message. This field is read only.

6.8.2. MSI-X Capability & Table Structures

The MSI-X capability structure is illustrated in Figure 6-2. More than one MSI-X capability structure per function is prohibited, but a function is permitted to have both an MSI and an MSI-X capability structure.

In contrast to the MSI capability structure, which directly contains all of the control/status information for the function's vectors, the MSI-X capability structure instead points to an MSI-X Table structure and a MSI-X Pending Bit Array (PBA) structure, each residing in Memory Space.

Each structure is mapped by a Base Address register (BAR) belonging to the function, located beginning at 10h in Configuration Space. A BAR Indicator register (BIR) indicates which BAR, and a QWORD-aligned Offset indicates where the structure begins relative to the base address associated with the BAR. The BAR is permitted to be either 32-bit or 64-bit, but must map Memory Space. A function is permitted to map both structures with the same BAR, or to map each structure with a different BAR.

The MSI-X Table structure, illustrated in Figure 6-3, typically contains multiple entries, each consisting of several fields: Message Address, Message Upper Address, Message Data, and Vector Control. Each entry is capable of specifying a unique vector.

The Pending Bit Array (PBA) structure, illustrated in Figure 6-4, contains the function's Pending Bits, one per Table entry, organized as a packed array of bits within QWORDs. The last QWORD will not necessarily be fully populated.

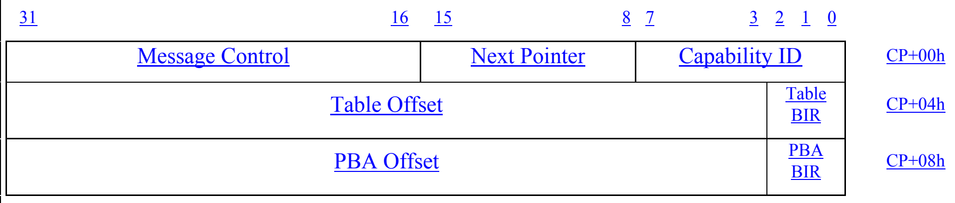


Figure 6-2: MSI-X Capability Structure

DWORD 3	DWORD 2	DWORD 1	DWORD 0		
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 0	Base
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 1	Base+1*16
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 2	Base+2*16
…	…	…	…	…	…
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry (N-1)	Base+(N-1)*16

Figure 6-3: MSI-X Table Structure

63 62 61 … … 2 1 0		
Pending Bits 0 through 63	QWORD 0	Base
Pending Bits 64 through 127	QWORD 1	Base+1*8
…	…	…
Pending Bits ((N-1) div 64)*64 through N-1	QWORD ((N-1) div 64)	Base+((N-1) div 64)*8

Figure 6-4: MSI-X PBA Structure

To request service using a given MSI-X Table entry, a function performs a DWORD memory write transaction using the contents of the Message Data field entry for data, the contents of the Message Upper Address field for the upper 32 bits of address, and the contents of the Message Address field entry for the lower 32 bits of address. A memory read transaction from the address targeted by the MSI-X message produces undefined results.

If a Base Address register that maps address space for the MSI-X Table or MSI-X PBA also maps other usable address space that is not associated with MSI-X structures, locations (e.g., for CSRs) used in the other address space must not share any naturally aligned 4 KB address range with one where either MSI-X structure resides. This allows system software where applicable to use different processor attributes for MSI-X structures and the other address space. (Some processor architectures do not support having different processor attributes associated with the same naturally aligned 4 KB physical address range.) The MSI-X Table and MSI-X PBA are permitted to co-reside within a naturally aligned 4 KB address range, though they must not overlap with each other.

Implementation Note: Dedicated BARs and Address Range Isolation

To enable system software to map MSI-X structures onto different processor pages for improved access control, it is recommended that a function dedicate separate Base Address registers for the MSI-X Table and MSI-X PBA, or else provide more than the minimum required isolation with address ranges.

If dedicated separate Base Address registers is not feasible, it is recommended that a function dedicate a single Base Address register for the MSI-X Table and MSI-X PBA.

If a dedicated Base Address register is not feasible, it is recommended that a function isolate the MSI-X structures from the non-MSI-X structures with aligned 8 KB ranges rather than the mandatory aligned 4 KB ranges.

For example, if a Base Address register needs to map 2 KB for an MSI-X Table containing 128 entries, 16 bytes for an MSI-X PBA containing 128 bits, and 64 bytes for registers not related to MSI-X, the following is an *acceptable* implementation. The Base Address register requests 8 KB of total address space, maps the first 64 bytes for the non MSI-X registers, maps the MSI-X Table beginning at an offset of 4 KB, and maps the MSI-X PBA beginning at an offset of 6 KB.

A *preferable* implementation for a shared Base Address register is for it to request 16 KB of total address space, map the first 64 bytes for the non MSI-X registers, map the MSI-X Table beginning at an offset of 8 KB, and map the MSI-X PBA beginning at an offset of 12 KB.

Implementation Note: MSI-X Memory Space Structures in Read/Write Memory

The MSI-X Table and MSI-X PBA structures are defined such that they can reside in general purpose read/write memory on a device, for ease of implementation and added flexibility. To achieve this, none of the contained fields are required to be read-only, and there are also restrictions on transaction alignment and sizes.

For all accesses to MSI-X Table and MSI-X PBA fields, software must use aligned full DWORD or aligned full QWORD transactions; otherwise, the result is undefined.

MSI-X Table entries and Pending bits are each numbered 0 through N-1, where N-1 is indicated by the Table Size field in the MSI-X Message Control register. For a given arbitrary MSI-X Table entry *K*, its starting address can be calculated with the formula:

entry starting address = Table base + K*16

For the associated Pending bit K, its address for QWORD access and bit number within that QWORD can be calculated with the formulas:

QWORD address = PBA base + (K div[41] 64)*8

QWORD bit# = K mod[42] 64

Software that chooses to read Pending bit K with DWORD accesses can use these formulas:

DWORD address = PBA base + (K div 32)*4

DWORD bit# = K mod 32

Each field in the MSI-X capability, Table, and PBA structures is further described in the following sections. Within the MSI-X capability structure, reserved registers and bits always return 0 when read, and write operations have no effect. Read-only registers return valid data when read, and write operations have no effect. Within the MSI-X Table and PBA structures, reserved fields have special rules.

6.8.2.1. Capability ID for MSI-X

7::0	CAP_ID	The value of 11h in this field identifies the function as being MSI-X capable. This field is read only.

[41] Div is an integer divide with truncation.

[42] Mod is the remainder from an integer divide.

6.8.2.2. Next Pointer for MSI-X

7::0	NXT_PTR	Pointer to the next item in the capabilities list. Must be NULL for the final item in the list. This field is read only.

6.8.2.3. Message Control for MSI-X

After reset, MSI-X is disabled. If MSI and MSI-X are both disabled, the function requests servicing via its **INTx#** pin (if supported). System software can enable MSI-X by setting bit 15 of this register. System software is permitted to modify the Message Control register's read/write bits and fields. A device driver is not permitted to modify the Message Control register's read/write bits and fields.

Bits	**Field**	**Description**
15	MSI-X Enable	If 1 and the MSI Enable bit in the MSI Message Control register (see Section 6.8.1.3) is 0, the function is permitted to use MSI-X to request service and is prohibited from using its **INTx#** pin (if implemented; see 6.2.4 Interrupt pin register). System configuration software sets this bit to enable MSI-X. A device driver is prohibited from writing this bit to mask a function's service request. If 0, the function is prohibited from using MSI-X to request service. This bit's state after reset is 0 (MSI-X is disabled). This bit is read/write.
14	Function Mask	If 1, all of the vectors associated with the function are masked, regardless of their per-vector Mask bit states. If 0, each vector's Mask bit determines whether the vector is masked or not. Setting or clearing the MSI-X Function Mask bit has no effect on the state of the per-vector Mask bits. This bit's state after reset is 0 (unmasked). This bit is read/write.
13::11	Reserved	Always returns 0 on a read and a write operation has no effect.
10::00	Table Size	System software reads this field to determine the MSI-X Table Size N, which is encoded as N-1. For example, a returned value of "00000000011" indicates a table size of 4. This field is read only.

6.8.2.4. Table Offset / Table BIR for MSI-X

Bits	**Field**	**Description**
31::3	Table Offset	Used as an offset from the address contained by one of the function's Base Address registers to point to the base of the MSI-X Table. The lower 3 Table BIR bits are masked off (set to zero) by software to form a 32-bit QWORD-aligned offset. This field is read only.
2::0	Table BIR	Indicates which one of a function's Base Address registers, located beginning at 10h in Configuration Space, is used to map the function's MSI-X Table into Memory Space. **BIR Value** — **Base Address register** 0 — 10h 1 — 14h 2 — 18h 3 — 1Ch 4 — 20h 5 — 24h 6 — Reserved 7 — Reserved For a 64-bit Base Address register, the Table BIR indicates the lower DWORD. With PCI-to-PCI bridges, BIR values 2 through 5 are also reserved. This field is read only.

6.8.2.5. PBA Offset / PBA BIR for MSI-X

Bits	Field	Description
31::3	PBA Offset	Used as an offset from the address contained by one of the function's Base Address registers to point to the base of the MSI-X PBA. The lower 3 PBA BIR bits are masked off (set to zero) by software to form a 32-bit QWORD-aligned offset. This field is read only.
2::0	PBA BIR	Indicates which one of a function's Base Address registers, located beginning at 10h in Configuration Space, is used to map the function's MSI-X PBA into Memory Space. The PBA BIR value definitions are identical to those for the MSI-X Table BIR. This field is read only.

6.8.2.6. Message Address for MSI-X Table Entries

Bits	Field	Description
31::02	Message Address	System-specified message lower address. For MSI-X messages, the contents of this field from an MSI-X Table entry specifies the lower portion of the DWORD-aligned address (**AD[31::02]**) for the memory write transaction. This field is read/write.
01:00	Message Address	For proper DWORD alignment, software must always write zeroes to these two bits; otherwise the result is undefined. The state of these bits after reset must be 0. These bits are permitted to be read only or read/write.

6.8.2.7. Message Upper Address for MSI-X Table Entries

Bits	Field	Description
31::00	Message Upper Address	System-specified message upper address bits. If this field is zero, Single Address Cycle (SAC) messages are used. If this field is non-zero, Dual Address Cycle (DAC) messages are used. This field is read/write.

6.8.2.8. Message Data for MSI-X Table Entries

Bits	**Field**	**Description**
31::00	Message Data	System-specified message data. For MSI-X messages, the contents of this field from an MSI-X Table entry specifies the data driven on **AD[31::00]** during the memory write transaction's data phase. **C/BE[3::0]#** are asserted during the data phase of the memory write transaction. In contrast to message data used for MSI messages, the low-order message data bits in MSI-X messages are **not** modified by the function. This field is read/write.

6.8.2.9. Vector Control for MSI-X Table Entries

Bits	**Field**	**Description**
31::01	Reserved	After reset, the state of these bits must be 0. However, for potential future use, software must preserve the value of these reserved bits when modifying the value of other Vector Control bits. If software modifies the value of these reserved bits, the result is undefined.
00	Mask Bit	When this bit is set, the function is prohibited from sending a message using this MSI-X Table entry. However, any other MSI-X Table entries programmed with the same vector will still be capable of sending an equivalent message unless they are also masked. This bit's state after reset is 1 (entry is masked). This bit is read/write.

6.8.2.10. Pending Bits for MSI-X PBA Entries

Bits	Field	Description
63::00	Pending Bits	For each Pending Bit that is set, the function has a pending message for the associated MSI-X Table entry. Pending bits that have no associated MSI-X Table entry are reserved. After reset, the state of reserved Pending bits must be 0. Software should never write, and should only read Pending Bits. If software writes to Pending Bits, the result is undefined. Each Pending Bit's state after reset is 0 (no message pending). These bits are permitted to be read only or read/write.

6.8.3. MSI and MSI-X Operation

At configuration time, system software traverses the function's capability list. If a capability ID of 05h is found, the function implements MSI. If a capability ID of 11h is found, the function implements MSI-X. A given function is permitted to implement MSI alone, MSI-X alone, both, or neither. Within a device, different functions are permitted to implement different sets of these interrupt mechanisms, and system software manages each function's interrupt mechanisms independently.

6.8.3.1. MSI Configuration

In this section, all register and field references are in the context of the MSI capability structure.

System software reads the ~~MSI capability structure's~~ Message Control register to determine the function's MSI capabilities.

System software reads the Multiple Message Capable field (bits 3-1 of the Message Control register) to determine the number of requested ~~message~~vectors. MSI supports a maximum of 32 vectors per function. System software writes to the Multiple Message Enable field (bits 6-4 of the Message Control register) to allocate either all or a subset of the requested ~~message~~vectors. For example, a function can request four ~~message~~vectors and be allocated either four, two, or one ~~message~~vector. The number of ~~message~~vectors requested and allocated ~~are~~ is aligned to a power of two (a function that requires three ~~message~~vectors must request four).

If the Per-vector Masking Capable bit (bit 8 of the Message Control register) is set, and system software supports per-vector masking, system software may mask one or more vectors by writing to the Mask Bits register.

If the 64-bit Address Capable bit (bit 7 of the Message Control register) is set, system software initializes the MSI capability structure's Message Address register (specifying the lower 32 bits of the message address) and the Message Upper Address register (specifying the upper 32 bits of the message address) with a system-specified message

~~destination~~ address. System software may program the Message Upper Address register to zero so that the function ~~generates~~ uses a 32-bit address for the MSI write transaction. If this bit is clear, system software initializes the MSI capability structure's Message Address register (specifying a 32-bit message address) with a system specified message ~~destination~~ address.

System software initializes the MSI capability structure's Message Data register with a system specified ~~message~~data value. Care must be taken to initialize only the Message Data register (i.e., a 2-byte value) and not modify the upper two bytes of that DWORD location.

6.8.3.2. MSI-X Configuration

In this section, all register and field references are in the context of the MSI-X capability, MSI-X Table, and MSI-X PBA structures.

System software allocates address space for the function's standard set of Base Address registers and sets the registers accordingly. One of the function's Base Address registers includes address space for the MSI-X Table, though the system software that allocates address space does not need to be aware of which Base Address register this is, or the fact the address space is used for the MSI-X Table. The same or another Base Address register includes address space for the MSI-X PBA, and the same point regarding system software applies.

Depending upon system software policy, system software, device driver software, or each at different times or environments may configure a function's MSI-X capability and table structures with suitable vectors. For example, a booting environment will likely require only a single vector, whereas a normal OS environment for running applications may benefit from multiple vectors if the function supports an MSI-X Table with multiple entries. For the remainder of this section, "software" refers to either system software or device driver software.

Software reads the Table Size field from the Message Control register to determine the MSI-X Table size. The field encodes the number of table entries as N-1, so software must add 1 to the value read from the field to calculate the number of table entries N. MSI-X supports a maximum table size of 2048 entries.

Software calculates the base address of the MSI-X Table by reading the 32-bit value from the Table Offset / Table BIR register, masking off the lower 3 Table BIR bits, and adding the remaining QWORD-aligned 32-bit Table offset to the address taken from the Base Address register indicated by the Table BIR. Software calculates the base address of the MSI-X PBA using the same process with the PBA Offset / PBA BIR register.

For each MSI-X Table entry that will be used, software fills in the Message Address field, Message Upper Address field, Message Data field, and Vector Control Field. Software must not modify the Address or Data fields of an entry while it is unmasked. Refer to Section 6.8.3.5 for details.

Implementation Note: Special Considerations for QWORD Accesses

Software is permitted to fill in MSI-X Table entry DWORD fields individually with DWORD writes, or software in certain cases is permitted to fill in appropriate pairs of DWORDs with a single QWORD write. Specifically, software is always permitted to fill in the Message Address and Message Upper Address fields with a single QWORD write. If a given entry is currently masked (via its Mask bit or the Function Mask bit), software is permitted to fill in the Message Data and Vector Control fields with a single QWORD write, taking advantage of the fact the Message Data field is guaranteed to become visible to hardware no later than the Vector Control field. However, if software wishes to mask a currently unmasked entry (without setting the Function Mask bit), software must set the entry's Mask bit using a DWORD write to the Vector Control field, since performing a QWORD write to the Message Data and Vector Control fields might result in the Message Data field being modified before the Mask bit in the Vector Control field becomes set.

For potential use by future specifications, the Reserved bits in the Vector Control field must have their values after reset preserved by software. If software does not preserve their values, the result is undefined.

For each MSI-X Table entry that software chooses not to configure for generating messages, software can simply leave the entry in its default state of being masked.

Software is permitted to configure multiple MSI-X Table entries with the same vector, and this may indeed be necessary when fewer vectors are allocated than requested.

Implementation Note: Handling MSI-X Vector Shortages

For the case where fewer vectors are allocated to a function than desired, software-controlled aliasing as enabled by MSI-X is one approach for handling the situation. For example, if a function supports 5 queues, each with an associated MSI-X table entry, but only 3 vectors are allocated, the function could be designed for software still to configure all 5 table entries, assigning one or more vectors to multiple table entries. Software could assign the 3 vectors {A,B,C} to the 5 entries as ABCCC, ABBCC, ABCBA, or other similar combinations.

Alternatively, the function could be designed for software to configure it (using a device-specific mechanism) to use only 3 queues and 3 MSI-X table entries. Software could assign the 3 vectors {A,B,C} to the 5 entries as ABC--, A-B-C, A--CB, or other similar combinations.

6.8.3.3. Enabling Operation

To maintain backward compatibility, the MSI Enable bit in the MSI Message Control register and the MSI-X Enable bit in the MSI-X Message Control register ~~(bit 0 of the Message Control register) is~~ are each cleared after reset (MSI ~~is~~ and MSI-X are both disabled). System configuration software sets ~~this bit~~one of these bits to enable either MSI or MSI-X, but never both simultaneously. Behavior is undefined if both MSI and MSI-X are enabled simultaneously. A device driver is prohibited from writing this bit to mask a function's service request. ~~Once~~ While enabled for MSI or MSI-X operation, a function is prohibited from using its **INTx#** pin (if implemented) to request service (MSI, MSI-X, and **INTx#** are mutually exclusive).

6.8.3.4. ~~Generating~~Sending Messages

Once MSI or MSI-X is enabled (the appropriate bit ~~0 of~~ in one of the Message Control ~~R~~registers is set), and one or more vectors is unmasked, the function is permitted to ~~may~~ send messages. To send a message, a function does a DWORD memory write to the appropriate message address with the appropriate message data.~~address specified by the contents of the Message Address register (and optionally the Message Upper Address register for a 64-bit message address).~~

~~The~~ For MSI, the DWORD that is written is made up of the value in the MSI Message Data register in the lower two bytes and zeroes in the upper two bytes.

~~If~~ For MSI, if the Multiple Message Enable field (bits 6-4 of the MSI Message Control register) is non-zero, the ~~device~~ function is permitted to modify the low order bits of the message data to generate multiple ~~message~~vectors. For example, a Multiple Message Enable encoding of “010” indicates the function is permitted to modify message data bits 1 and 0 to generate up to four unique ~~message~~vectors. If the Multiple Message Enable field is “000”, the function is not permitted to modify the message data.

For MSI-X, the MSI-X Table contains at least one entry for every allocated vector, and the 32-bit Message Data field value from a selected table entry is used in the message **without** any modification to the low-order bits by the function.

How a function uses multiple ~~message~~vectors (when allocated) is device dependent. A function must handle being allocated fewer ~~less message~~vectors than requested.

6.8.3.5. Per-vector Masking and Function Masking

Per-vector masking is an optional feature with MSI, and a standard feature in MSI-X.

Function Masking is a standard feature in MSI-X. When the MSI-X Function Mask bit is set, all of the function’s entries must behave as being masked, regardless of the per-entry Mask bit states. Function Masking is not supported in MSI, but software can readily achieve a similar effect by setting all MSI Mask bits using a single DWORD write.

“Per-vector masking” in MSI-X is controlled by a Mask bit in each MSI-X Table entry. While more accurately termed “per-entry masking”, masking an MSI-X Table entry is still referred to as “vector masking” so similar descriptions can be used for both MSI and MSI-X. However, since software is permitted to program the same vector (a unique Address/Data pair) into multiple MSI-X table entries, all such entries must be masked in order to guarantee the function won’t send a message using that Address/Data pair.

For MSI and MSI-X, while a vector is masked, the function is prohibited from sending the associated message, and the function must set the associated Pending bit whenever the function would otherwise send the message. When software unmasks a vector whose associated Pending bit is set, the function must schedule sending the associated message, and clear the Pending bit as soon as the message has been sent. Note that clearing the MSI-X Function Mask bit may result in many messages needing to be sent.

If a masked vector has its Pending bit set, and the associated underlying interrupt events are somehow satisfied (usually by software though the exact manner is function-specific), the function must clear the Pending bit, to avoid sending a spurious interrupt message later when software unmasks the vector. However, if a subsequent interrupt event occurs while the vector is still masked, the function must again set the Pending bit.

Software is permitted to mask one or more vectors indefinitely, and service their associated interrupt events strictly based on polling their Pending bits. A function must set and clear its Pending bits as necessary to support this “pure polling” mode of operation.

For MSI-X, a function is permitted to cache Address and Data values from unmasked MSI-X Table entries. However, anytime software unmasks a currently masked MSI-X Table entry either by clearing its Mask bit or by clearing the Function Mask bit, the function must update any Address or Data values that it cached from that entry. If software changes the Address or Data value of an entry while the entry is unmasked, the result is undefined.

6.8.3.6. Hardware/Software Synchronization

If a ~~device~~function ~~generates~~sends messages with ~~signals~~ the same ~~message~~ vector ~~many~~ multiple times before being acknowledged by software, only one message is guaranteed to be serviced. If all messages must be serviced, a device driver handshake is required. In other words, once a function ~~signals~~sends ~~Message~~ Vector A, it cannot ~~signal~~send ~~Message~~ Vector A again until it is explicitly enabled to do so by its device driver (provided all messages must be serviced). If some messages can be lost, a device driver handshake is not required. For functions that support multiple ~~message~~vectors, a function can ~~signal~~ send multiple unique ~~message~~vectors and is guaranteed that each unique message will be serviced. For example, a ~~device~~function can ~~signal~~send ~~Message~~ Vector A followed by ~~Message~~ Vector B without any device driver handshake (both ~~Message~~ Vector A and ~~Message~~ Vector B will be serviced).

Implementation Note: Servicing MSI and MSI-X Interrupts

When system software allocates fewer MSI or MSI-X vectors to a function than it requests, multiple interrupt sources within the function, each desiring a unique vector, may be required to share a single vector. Without proper handshakes between hardware and software, hardware may send fewer messages than software expects, or hardware may send what software considers to be extraneous messages.

A rather sophisticated but resource-intensive approach is to associate a dedicated event queue with each allocated vector, with producer and consumer pointers for managing each event queue. Such event queues typically reside in host memory. The function acts as the producer and software acts as the consumer. Multiple interrupt sources within a function may be assigned to each event queue as necessary. Each time an interrupt source needs to signal an interrupt, the function places an entry on the appropriate event queue (assuming there's room), updates a copy of the producer pointer (typically in host memory), and sends an interrupt message with the associated vector when necessary to notify software that the event queue needs servicing. The interrupt service routine for a given event queue processes all entries it finds on its event queue, as indicated by the producer pointer. Each event queue entry identifies the interrupt source and possibly additional information about the nature of the event. The use of event queues and producer/consumer pointers can be used to guarantee that interrupt events won't get dropped when multiple interrupt sources are forced to share a vector. There's no need for additional handshaking between sending multiple messages associated with the same event queue, to guarantee that every message gets serviced. In fact, various standard techniques for "interrupt coalescing" can be used to avoid sending a separate message for every event that occurs, particularly during heavy bursts of events.

In more modest implementations, the hardware design of a function's MSI or MSI-X logic sends a message any time a falling edge would have occurred on the **INTx#** pin if MSI or MSI-X had not been enabled. For example, consider a scenario in which two interrupt events (possibly from distinct interrupt sources within a function) occur in rapid succession. The first event causes a message to be sent. Before the interrupt service routine has had an opportunity to service the first event, the second event occurs. In this case, only one message is sent, because the first event is still active at the time the second event occurs (a hardware **INTx#** pin signal would have had only one falling edge).

One handshake approach for implementations like the above is to use standard per-vector masking, and allow multiple interrupt sources to be associated with each vector. A given vector's interrupt service routine sets the vector's Mask bit before it services any associated interrupting events and clears the Mask bit after it has serviced all the events it knows about. (This could be any number of events.) Any occurrence of a new event while the Mask bit is set results in the Pending bit being set. If one or more associated events are still pending at the time the vector's Mask bit is cleared, the function immediately sends another message.

A handshake approach for MSI functions that do not implement per-vector masking is for a vector's interrupt service routine to re-inspect all of the associated interrupt events after clearing what is presumed to be the last pending interrupt event. If another event is found to be active, it is serviced in the same interrupt service routine invocation, and the complete re-inspection is repeated until no pending events are found. This ensures that if an additional interrupting event occurs before a previous interrupt event is cleared, whereby the function does *not* send an additional interrupt message, that the new event is serviced as part of the current interrupt service routine invocation.

This alternative has the potential side effect of one vector's interrupt service routine processing an interrupting event that has already generated a new interrupt message. The interrupt service routine invocation resulting from the new message may find no pending interrupt events. Such occurrences are sometimes referred to as spurious interrupts, and software using this approach must be prepared to tolerate them.

~~An MSI is by definition a non-shared interrupt that enforces data consistency (ensures the interrupt service routine accesses the most recent data). The system guarantees that any data written by the device prior to sending the MSI has reached its final destination before the interrupt service routine accesses that data. Therefore, a device driver is not required to read its device before servicing its MSI.~~An MSI or MSI-X message, by virtue of being a posted memory write (PMW) transaction, is prohibited by PCI ordering rules from passing PMW transactions sent earlier by the function. The system must guarantee that an interrupt service routine invoked as a result of a given message will observe any updates performed by PMW transactions arriving prior to that message. Thus, the interrupt service routine of a device driver is not required to read from a device register in order to ensure data consistency with previous PMW transactions. However, if multiple MSI-X Table entries share the same vector, the interrupt service routine may need to read from some device specific register to determine which interrupt sources need servicing.

6.8.3.7. ~~MSI~~ Message Transaction Termination

The target of an MSI or MSI-X write transaction cannot distinguish between it and any other memory write transaction. The termination requirements for an MSI or MSI-X write transaction are the same as for any other memory write transaction except as noted below.

If the MSI or MSI-X write transaction is terminated with a Master-Abort or a Target-Abort, the master that originated the MSI or MSI-X memory write transaction is required to report the error by asserting **SERR#** (if bit 8 in the Command register is set) and to set the appropriate bits in the Status register (refer to Section 3.7.4.2.). An MSI or MSI-X memory write transaction is ignored by the target if it is terminated with a Master-Abort or Target-Abort.

Refer to the *PCI-to-PCI Bridge Architecture Specification, Revision* 1.1 (Section 6, Error Support) for PCI-to-PCI bridge **SERR#** generation in response to error conditions from posted memory writes on the destination bus.

Note that **SERR#** generation in an MSI-enabled environment containing PCI-to-PCI bridges requires the **SERR#** reporting enable bits in all devices in the MSI message path to be set. For PCI-to-PCI bridges specifically, refer to Section 6.2.2 and *PCI-to-PCI Bridge Architecture Specification, Revision* 1.1, Sections 3.2.4.3 and 3.2.5.17.).

If the MSI or MSI-X write transaction results in a data parity error, the master that originated the MSI or MSI-X write transaction is required to assert **SERR#** (if bit 8 in the Command register is set) and to set the appropriated bits in the Status register (refer to Section 3.7.4.).

6.8.2.2. ~~MSI~~ Message Transaction Reception and Ordering Requirements

As with all memory write transactions, the device that includes the target of the interrupt message (the interrupt receiver) is required to complete all interrupt message transactions as a target without requiring other transactions to complete first as a master. (Refer to Section 3.3.3.3.4. In general, this means that the message receiver must complete the interrupt message transaction independent of when the CPU services the interrupt. For example, each time the interrupt receiver receives an interrupt message, it could set a bit in an internal register indicating that this message had been received and then complete the transaction on the bus. The appropriate interrupt service routine would later be dispatched because this bit was set. The message receiver would not be allowed to delay the completion of the interrupt message on the bus pending acknowledgement from the processor that the interrupt was being serviced. Such dependencies can lead to deadlock when multiple devices ~~generate~~ send interrupt messages simultaneously.

Although interrupt messages remain strictly ordered throughout the PCI bus hierarchy, the order of receipt of the interrupt messages does not guarantee any order in which the interrupts will be serviced. Since the message receiver must complete all interrupt message transactions without regard to when the interrupt was actually serviced, the message receiver will generally not maintain any information about the order in which the interrupts were received. This is true both of interrupt messages received from different devices, and multiple messages received from the same device. If a device requires one interrupt message to be serviced before another, then the device must not send the second interrupt message until the first one has been serviced.