The MSI Driver Guide HOWTO

Authors: Tom L Nguyen
Martine Silbermann
Matthew Wilcox

About this guide

This guide describes the basics of Message Signaled Interrupts (MSIs), the advantages of using MSI over traditional interrupt mechanisms, how to change your driver to use MSI or MSI-X and some basic diagnostics to try if a device doesn't support MSIs.

What are MSIs?

A Message Signaled Interrupt is a write from the device to a special address which causes an interrupt to be received by the CPU.

The MSI capability was first specified in PCI 2.2 and was later enhanced in PCI 3.0 to allow each interrupt to be masked individually. The MSI-X capability was also introduced with PCI 3.0. It supports more interrupts per device than MSI and allows interrupts to be independently configured.

Devices may support both MSI and MSI-X, but only one can be enabled at a time.

Why use MSIs?

There are three reasons why using MSIs can give an advantage over traditional pin-based interrupts.

Pin-based PCI interrupts are often shared amongst several devices. To support this, the kernel must call each interrupt handler associated with an interrupt, which leads to reduced performance for the system as a whole. MSIs are never shared, so this problem cannot arise.

When a device writes data to memory, then raises a pin-based interrupt, it is possible that the interrupt may arrive before all the data has arrived in memory (this becomes more likely with devices behind PCI-PCI bridges). In order to ensure that all the data has arrived in memory, the interrupt handler must read a register on the device which raised the interrupt. PCI transaction ordering rules require that all the data arrive in memory before the value may be returned from the register. Using MSIs avoids this problem as the interrupt-generating write cannot pass the data writes, so by the time the interrupt is raised, the driver knows that all the data has arrived in memory.

PCI devices can only support a single pin-based interrupt per function. Often drivers have to query the device to find out what event has occurred, slowing down interrupt handling for the common case. With MSIs, a device can support more interrupts, allowing each interrupt to be specialised to a different purpose. One possible design gives infrequent conditions (such as errors) their own interrupt which allows the driver to handle the normal interrupt handling path more efficiently. Other possible designs include giving one interrupt to each packet queue in a network card or each port in a storage controller.

How to use MSIs

PCI devices are initialised to use pin-based interrupts. The device driver has to set up the device to use MSI or MSI-X. Not all machines support MSIs correctly, and for those machines, the APIs described below will simply fail and the device will continue to use pin-based interrupts.

Include kernel support for MSIs

To support MSI or MSI-X, the kernel must be built with the CONFIG_PCI_MSI option enabled. This option is only available on some architectures, and it may depend on some other options also being set. For example, on x86, you must also enable X86_UP_APIC or SMP in order to see the CONFIG_PCI_MSI option.

Using MSI

Most of the hard work is done for the driver in the PCI layer. The driver simply has to request that the PCI layer set up the MSI capability for this device.

To automatically use MSI or MSI-X interrupt vectors, use the following function:

```
int pci_alloc_irq_vectors(struct pci_dev *dev, unsigned int min_vecs,
              unsigned int max_vecs, unsigned int flags);
```

which allocates up to max_vecs interrupt vectors for a PCI device. It returns the number of vectors allocated or a negative error. If the device has a requirements for a minimum number of vectors the driver can pass a min_vecs argument set to this limit, and the PCI core will return -ENOSPC if it can't meet the minimum number of vectors.

The flags argument is used to specify which type of interrupt can be used by the device and the driver (PCI_IRQ_LEGACY, PCI_IRQ_MSI, PCI_IRQ_MSIX). A convenient short-hand (PCI_IRQ_ALL_TYPES) is also available to ask for any possible kind of interrupt. If the PCI_IRQ_AFFINITY flag is set, pci_alloc_irq_vectors() will spread the interrupts around the available CPUs.

To get the Linux IRQ numbers passed to request_irq() and free_irq() and the vectors, use the following function:

```
int pci_irq_vector(struct pci_dev *dev, unsigned int nr);
```

Any allocated resources should be freed before removing the device using the following function:

```
void pci_free_irq_vectors(struct pci_dev *dev);
```

If a device supports both MSI-X and MSI capabilities, this API will use the MSI-X facilities in preference to the MSI facilities. MSI-X supports any number of interrupts between 1 and 2048. In contrast, MSI is restricted to a maximum of 32 interrupts (and must be a power of two). In addition, the MSI interrupt vectors must be allocated consecutively, so the system might not be able to allocate as many vectors for MSI as it could for MSI-X. On some platforms, MSI interrupts must all be targeted at the same set of CPUs whereas MSI-X interrupts can all be targeted at different CPUs.

If a device supports neither MSI-X or MSI it will fall back to a single legacy IRQ vector.

The typical usage of MSI or MSI-X interrupts is to allocate as many vectors as possible, likely up to the limit supported by the device. If nvec is larger than the number supported by the device it will automatically be capped to the supported limit, so there is no need to query the number of vectors supported beforehand:

```
nvec = pci_alloc_irq_vectors(pdev, 1, nvec, PCI_IRQ_ALL_TYPES)
if (nvec < 0)
        goto out_err;
```

If a driver is unable or unwilling to deal with a variable number of MSI interrupts it can request a particular number of interrupts by passing that number to pci_alloc_irq_vectors() function as both 'min_vecs' and 'max_vecs' parameters:

```
ret = pci_alloc_irq_vectors(pdev, nvec, nvec, PCI_IRQ_ALL_TYPES);
if (ret < 0)
        goto out_err;
```

The most notorious example of the request type described above is enabling the single MSI mode for a device. It could be done by passing two 1s as 'min_vecs' and 'max_vecs':

```
ret = pci_alloc_irq_vectors(pdev, 1, 1, PCI_IRQ_ALL_TYPES);
if (ret < 0)
        goto out_err;
```

Some devices might not support using legacy line interrupts, in which case the driver can specify that only MSI or MSI-X is acceptable:

```
nvec = pci_alloc_irq_vectors(pdev, 1, nvec, PCI_IRQ_MSI | PCI_IRQ_MSIX);
if (nvec < 0)
        goto out_err;
```

Legacy APIs

The following old APIs to enable and disable MSI or MSI-X interrupts should not be used in new code:

```
pci_enable_msi()              /* deprecated */
pci_disable_msi()             /* deprecated */
pci_enable_msix_range()       /* deprecated */
pci_enable_msix_exact()       /* deprecated */
pci_disable_msix()            /* deprecated */
```

Additionally there are APIs to provide the number of supported MSI or MSI-X vectors: pci_msi_vec_count() and pci_msix_vec_count(). In general these should be avoided in favor of letting pci_alloc_irq_vectors() cap the number of vectors. If you have a legitimate special use case for the count of vectors we might have to revisit that decision and add a pci_nr_irq_vectors() helper that handles MSI and MSI-X transparently.

Considerations when using MSIs

Spinlocks

Most device drivers have a per-device spinlock which is taken in the interrupt handler. With pin-based interrupts or a single MSI, it is not necessary to disable interrupts (Linux guarantees the same interrupt will not be re-entered). If a device uses multiple interrupts, the driver must disable interrupts while the lock is held. If the device sends a different interrupt, the driver will deadlock trying to recursively acquire the spinlock. Such deadlocks can be avoided by using spin_lock_irqsave() or spin_lock_irq() which disable local interrupts and acquire the lock (see Documentation/kernel-hacking/locking.rst).

How to tell whether MSI/MSI-X is enabled on a device

Using 'lspci -v' (as root) may show some devices with "MSI", "Message Signalled Interrupts" or "MSI-X" capabilities. Each of these capabilities has an 'Enable' flag which is followed with either "+" (enabled) or "-" (disabled).

MSI quirks

Several PCI chipsets or devices are known not to support MSIs. The PCI stack provides three ways to disable MSIs:

1. globally
2. on all devices behind a specific bridge
3. on a single device

Disabling MSIs globally

Some host chipsets simply don't support MSIs properly. If we're lucky, the manufacturer knows this and has indicated it in the ACPI FADT table. In this case, Linux automatically disables MSIs. Some boards don't include this information in the table and so we have to detect them ourselves. The complete list of these is found near the quirk_disable_all_msi() function in drivers/pci/quirks.c.

If you have a board which has problems with MSIs, you can pass pci=nomsi on the kernel command line to disable MSIs on all devices. It would be in your best interests to report the problem to linux-pci@vger.kernel.org including a full 'lspci -v' so we can add the quirks to the kernel.

Disabling MSIs below a bridge

Some PCI bridges are not able to route MSIs between busses properly. In this case, MSIs must be disabled on all devices behind the bridge.

Some bridges allow you to enable MSIs by changing some bits in their PCI configuration space (especially the Hypertransport chipsets such as the nVidia nForce and Serverworks HT2000). As with host chipsets, Linux mostly knows about them and automatically enables MSIs if it can. If you have a bridge unknown to Linux, you can enable MSIs in configuration space using whatever method you know works, then enable MSIs on that bridge by doing:

```
echo 1 > /sys/bus/pci/devices/$bridge/msi_bus
```

where $bridge is the PCI address of the bridge you've enabled (eg 0000:00:0e.0).

To disable MSIs, echo 0 instead of 1. Changing this value should be done with caution as it could break interrupt handling for all devices below this bridge.

Again, please notify linux-pci@vger.kernel.org of any bridges that need special handling.

Disabling MSIs on a single device

Some devices are known to have faulty MSI implementations. Usually this is handled in the individual device driver, but occasionally it's necessary to handle this with a quirk. Some drivers have an option to disable use of MSI. While this is a convenient workaround for the driver author, it is not good practice, and should not be emulated.

Finding why MSIs are disabled on a device

From the above three sections, you can see that there are many reasons why MSIs may not be enabled for a given device. Your first step should be to examine your dmesg carefully to determine whether MSIs are enabled for your machine. You should also check your .config to be sure you have enabled CONFIG_PCI_MSI.

Then, 'lspci -t' gives the list of bridges above a device. Reading */sys/bus/pci/devices/*/msi_bus* will tell you whether MSIs are enabled (1) or disabled (0). If 0 is found in any of the msi_bus files belonging to bridges between the PCI root and the device, MSIs are disabled.

It is also worth checking the device driver to see whether it supports MSIs. For example, it may contain calls to pci_alloc_irq_vectors() with the PCI_IRQ_MSI or PCI_IRQ_MSIX flags.