The MSI Driver Guide HOWTO

Tom L Nguyen tom.l.nguyen@intel.com

10/03/2003

Revised Feb 12, 2004 by Martine Silbermann
email: Martine.Silbermann@hp.com
Revised Jun 25, 2004 by Tom L Nguyen
Revised Jul 9, 2008 by Matthew Wilcox <willy@linux.intel.com>

1. 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.

2. 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.

3. 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 arrives in memory before the value can 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.

4. 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.

4.1 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.

4.2 Using MSI

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

4.2.1 pci_enable_msi

int pci_enable_msi(struct pci_dev *dev)

A successful call will allocate ONE interrupt to the device, regardless
of how many MSIs the device supports. The device will be switched from
pin-based interrupt mode to MSI mode. The dev->irq number is changed
to a new number which represents the message signaled interrupt.
This function should be called before the driver calls request_irq()
since enabling MSIs disables the pin-based IRQ and the driver will not
receive interrupts on the old interrupt.

4.2.2 pci_enable_msi_block

int pci_enable_msi_block(struct pci_dev *dev, int count)

This variation on the above call allows a device driver to request multiple
MSIs. The MSI specification only allows interrupts to be allocated in
powers of two, up to a maximum of 2^5 (32).

If this function returns 0, it has succeeded in allocating at least as many
interrupts as the driver requested (it may have allocated more in order
to satisfy the power-of-two requirement). In this case, the function
enables MSI on this device and updates dev->irq to be the lowest of
the new interrupts assigned to it. The other interrupts assigned to
the device are in the range dev->irq to dev->irq + count - 1.

If this function returns a negative number, it indicates an error and

the driver should not attempt to request any more MSI interrupts for
this device. If this function returns a positive number, it will be
less than 'count' and indicate the number of interrupts that could have
been allocated. In neither case will the irq value have been
updated, nor will the device have been switched into MSI mode.

The device driver must decide what action to take if
pci_enable_msi_block() returns a value less than the number asked for.
Some devices can make use of fewer interrupts than the maximum they
request; in this case the driver should call pci_enable_msi_block()
again. Note that it is not guaranteed to succeed, even when the
'count' has been reduced to the value returned from a previous call to
pci_enable_msi_block(). This is because there are multiple constraints
on the number of vectors that can be allocated; pci_enable_msi_block()
will return as soon as it finds any constraint that doesn't allow the
call to succeed.

4.2.3 pci_disable_msi

void pci_disable_msi(struct pci_dev *dev)

This function should be used to undo the effect of pci_enable_msi() or
pci_enable_msi_block(). Calling it restores dev->irq to the pin-based
interrupt number and frees the previously allocated message signaled
interrupt(s). The interrupt may subsequently be assigned to another
device, so drivers should not cache the value of dev->irq.

A device driver must always call free_irq() on the interrupt(s)
for which it has called request_irq() before calling this function.
Failure to do so will result in a BUG_ON(), the device will be left with
MSI enabled and will leak its vector.

4.3 Using MSI-X

The MSI-X capability is much more flexible than the MSI capability.
It supports up to 2048 interrupts, each of which can be controlled
independently. To support this flexibility, drivers must use an array of
`struct msix_entry':

```
struct msix_entry {
        u16     vector; /* kernel uses to write alloc vector */
        u16     entry; /* driver uses to specify entry */
};
```

This allows for the device to use these interrupts in a sparse fashion;
for example it could use interrupts 3 and 1027 and allocate only a
two-element array. The driver is expected to fill in the 'entry' value
in each element of the array to indicate which entries it wants the kernel
to assign interrupts for. It is invalid to fill in two entries with the
same number.

4.3.1 pci_enable_msix

int pci_enable_msix(struct pci_dev *dev, struct msix_entry *entries, int nvec)

Calling this function asks the PCI subsystem to allocate 'nvec' MSIs.

The 'entries' argument is a pointer to an array of msix_entry structs
which should be at least 'nvec' entries in size. On success, the
function will return 0 and the device will have been switched into
MSI-X interrupt mode. The 'vector' elements in each entry will have
been filled in with the interrupt number. The driver should then call
request_irq() for each 'vector' that it decides to use.

If this function returns a negative number, it indicates an error and
the driver should not attempt to allocate any more MSI-X interrupts for
this device. If it returns a positive number, it indicates the maximum
number of interrupt vectors that could have been allocated. See example
below.

This function, in contrast with pci_enable_msi(), does not adjust
dev->irq. The device will not generate interrupts for this interrupt
number once MSI-X is enabled. The device driver is responsible for
keeping track of the interrupts assigned to the MSI-X vectors so it can
free them again later.

Device drivers should normally call this function once per device
during the initialization phase.

It is ideal if drivers can cope with a variable number of MSI-X interrupts,
there are many reasons why the platform may not be able to provide the
exact number a driver asks for.

A request loop to achieve that might look like:

```
static int foo_driver_enable_msix(struct foo_adapter *adapter, int nvec)
{
        while (nvec >= FOO_DRIVER_MINIMUM_NVEC) {
                rc = pci_enable_msix(adapter->pdev,
                                     adapter->msix_entries, nvec);
                if (rc > 0)
                        nvec = rc;
                else
                        return rc;
        }

        return -ENOSPC;
}
```

4.3.2 pci_disable_msix

void pci_disable_msix(struct pci_dev *dev)

This API should be used to undo the effect of pci_enable_msix(). It frees
the previously allocated message signaled interrupts. The interrupts may
subsequently be assigned to another device, so drivers should not cache
the value of the 'vector' elements over a call to pci_disable_msix().

A device driver must always call free_irq() on the interrupt(s)
for which it has called request_irq() before calling this function.
Failure to do so will result in a BUG_ON(), the device will be left with
MSI enabled and will leak its vector.

4.3.3 The MSI-X Table

The MSI-X capability specifies a BAR and offset within that BAR for the MSI-X Table. This address is mapped by the PCI subsystem, and should not be accessed directly by the device driver. If the driver wishes to mask or unmask an interrupt, it should call disable_irq() / enable_irq().

4.4 Handling devices implementing both MSI and MSI-X capabilities

If a device implements both MSI and MSI-X capabilities, it can run in either MSI mode or MSI-X mode but not both simultaneously. This is a requirement of the PCI spec, and it is enforced by the PCI layer. Calling pci_enable_msi() when MSI-X is already enabled or pci_enable_msix() when MSI is already enabled will result in an error. If a device driver wishes to switch between MSI and MSI-X at runtime, it must first quiesce the device, then switch it back to pin-interrupt mode, before calling pci_enable_msi() or pci_enable_msix() and resuming operation. This is not expected to be a common operation but may be useful for debugging or testing during development.

4.5 Considerations when using MSIs

4.5.1 Choosing between MSI-X and MSI

If your device supports both MSI-X and MSI capabilities, you should use the MSI-X facilities in preference to the MSI facilities. As mentioned above, MSI-X supports any number of interrupts between 1 and 2048. In constrast, 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 may not be able to allocate as many vectors for MSI as it could for MSI-X. On some platforms, MSI interrupts must all be targetted at the same set of CPUs whereas MSI-X interrupts can all be targetted at different CPUs.

4.5.2 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.

There are two solutions. The first is to take the lock with spin_lock_irqsave() or spin_lock_irq() (see Documentation/DocBook/kernel-locking). The second is to specify IRQF_DISABLED to request_irq() so that the kernel runs the entire interrupt routine with interrupts disabled.

If your MSI interrupt routine does not hold the lock for the whole time it is running, the first solution may be best. The second solution is normally preferred as it avoids making two transitions from interrupt disabled to enabled and back again.

4.6 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 will be followed with either "+" (enabled) or "-" (disabled).

5. 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

5.1. 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 will automatically disable 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.

5.2. 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 which Linux doesn't yet know about, 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 can break interrupt handling for all devices below this bridge.

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

5.3. 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 practise, and should not be emulated.

5.4. 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 MSI 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_enable_msi(), pci_enable_msix() or pci_enable_msi_block().