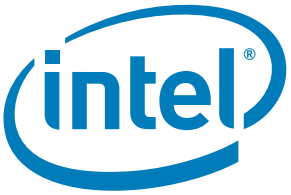


# Assigning Interrupts to Processor Cores using an Intel® 82575/82576 or 82598/82599 Ethernet Controller

*September 2009*

319935-002

# Contents

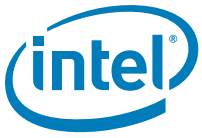


# Revision History

| Rev | Rev Date | Description |
|---|---|---|
| 2.1 | Sept 2009 | Added 82576 and 82599 applicability. |
| 2.0 | Feb 2009 | Initial public release. |



*Note: This page intentionally left blank.*



# 1.0 IP Forwarding and Pinning Queues to Processors

Multicore processors and the newest Ethernet adapters (including the 82575, 82576, 82598, and 82599) allow TCP forwarding flows to be optimized by assigning execution flows to individual cores. By default, Linux automatically assigns interrupts to processor cores. Two methods currently exist for automatically assigning the interrupts, an in-kernel IRQ balancer and the IRQ balance daemon in user space. Both offer tradeoffs that might lower CPU usage but do not maximize the IP forwarding rates. Optimal throughput can be obtained by manually pinning the queues of the Ethernet adapter to specific processor cores.

For IP forwarding, a transmit/receive queue pair should use the same processor core and reduce any cache synchronization between different cores. This can be performed by assigning transmit and receive interrupts to specific cores. Starting with Linux kernel 2.6.27, multiple queues can be used on the 82575, 82576, 82598, and 82599. Additionally, multiple transmit queues were enabled in Extended Messaging Signaled Interrupts (MSI-X). MSI-X supports a larger number of interrupts that can be used, allowing for finer-grained control and targeting of the interrupts to specific CPUs.

*Note:* The cores on one socket of a dual-socket system can be assigned non-consecutive numbers. For example, on a dual-socket quad-core system Socket 0 has CPU 0, 2, 4, 6 and Socket 1 has CPU 1, 3, 5, 7 on x86_64, and Socket 0 has CPU 0, 1, 2, 3 and Socket 1 has CPU 4, 5, 6 7 on x86_32.

## 1.1 Assumptions

The following hardware and software are assumed in this document's IP forwarding example:

- x86_64 SMP system
- Intel Ethernet adapter using 82575EB,82576EB, 82598EB,or 82599ES Ethernet controllers
- Installed Fedora Core 11 x86_64 with development packages
- Kernel version 2.6.29.6-217.2.16.fc11.x86_64 (Use a kernel >2.6.27 for MSI-X and multiple transmit and receive queues. At the time of writing, most stable Linux distributions are using earlier kernels. Some Linux distributions backport patches to earlier kernels; check with your distribution to see if an earlier kernel has backported changes for the previously mentioned features.)
- Network connection

## 1.2 Build a New Kernel

*Note:* These instructions were summarized from

http://fedoraproject.org/wiki/Docs/CustomKernel.

Build the kernel as a user with "sudo" privileges.

1. Install prerequisite packages *rpmdevtools* and *yum-utils*.

   ```
   sudo yum install yum-utils rpmdevtools
   ```

2. Prepare an RPM package building environment in a home directory.

   ```
   rpmdev-setuptree
   ```

3. Download the kernel source rpm file.

   ```
   yumdownloader --source kernel
   ```

4. Install build dependencies for the kernel source:

   ```
   sudo yum-builddep kernel-2.6.29.6-217.2.16.fc11.x86_64
   ```

5. Install the kernel into the local *rpmbuild* directories. Note that kernel-2.6.29.6-217.2.16.fc11.src.rpm installs into ~/rpmbuild/SOURCES and ~/rpmbuild/SPECS.

   ```
   rpm -Uvh kernel-2.6.29.6-217.2.16.fc11.x86_64.src.rpm
   ```

Ignore any messages similar to the following:

```
warning: user mockbuild does not exist - using root

warning: group mockbuild does not exist - using root
```

6. Prepare the kernel source tree. The source is in the ~/rpmbuild/BUILD/kernel-2.6.29/linux-2.6.29.x86_64.

   ```
   cd ~/rpmbuild/SPECS

   rpmbuild -bp --target=`uname -m` kernel.spec

   cd ~/rpmbuild/BUILD/kernel-2.6.29/linux-2.6.29.x86_64/
   ```

7. Copy the default configuration file.

   ```
   cp configs/kernel-2.6.29.6-x86_64.config .config

   make oldconfig
   ```

8. Configure the kernel.
   a. Start up the configuration menu:

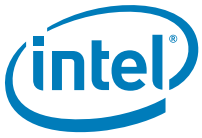
      ```
      make menuconfig
      ```

   b. Make sure you are using the SLUB memory allocator (the unqueued SLAB allocator):

      ```
      General setup -> Choose SLAB allocator -> SLUB allocator (Unqueued Allocator)
      ```

   c. Turn off forced pre-emption:

      ```
      Processor type and features -> Preemption Model -> No Forced Preemption
      (Server)
      ```



d. Turn on NUMA memory allocation (for multiple processors):

```
Processor type and features -> Numa Memory Allocation and Scheduler
Support = Y
```

e. Lower the interrupt frequency. 100 Hz is a typical choice for servers, SMP and NUMA with most processors might show reduced performance when too many timer interrupts are occurring:

```
Processor type and features -> Timer Frequency -> 100 Hz
```

f. Make sure MSI and MSI-X are enabled:

```
Bus options (PCI etc.) -> Message Signaled Interrupts (MSI and MSI-X) = Y
```

g. Turn off Fair Queueing and QoS:

```
Networking support -> Networking Options -> QoS and/or Fair Queueing = N
```

h. Turn off kernel statistics:

```
Kernel hacking -> Collect kernel timers statistics = N
```

i. If possible:

```
Kernel hacking -> Collect scheduler statistics = N
```

9. Add a new comment line to the top of the .config file.

The comment should contain the hardware platform the kernel is built for. This is found with *uname -i*.

For example, an x86_64 machine has the following line added to the top of the config file:

```
# x86_64
```

10. Copy the generated config file to *~/rpmbuild/SOURCES/*:

```
cp .config ~/rpmbuild/SOURCES/config-x86_64
```

11. Edit the kernel.spec file to create unique kernel package names.

```
cd ~/rpmbuild/SPECS
```

12. Edit the *buildid* line in the *kernel.spec* file to generate a unique kernel name. Change the line *# % define buildid .local* to *%define buildid .test*.

Note that there is a space between the "*%*" and "*define*" that needs to be deleted.

13. Use the rpmbuild utility to build the new RPM files.

```
rpmbuild -bb --target=`uname -m` kernel.spec
```

The build process takes some time to complete. If the build completes successfully, the new kernel packages can be found in the *~/rpmbuild/RPMS/x86_64* directory.

## 1.3 Install the New Kernel and Restart the System

1. Install the new kernel with the rpm *-ivh* command. Do not use the *-U* or *--upgrade* options as that deletes your current kernel and it might be necessary to go back to an older version of the kernel if problems occur.

```
rpm -ivh ~/rpmbuild/RPMS/x86_64/kernel- 2.6.29.6-217.2.16.test.fc11.x86_64.rpm
```

2. Reboot.

```
shutdown -r now
```

## 1.4 IGB and IXGBE Drivers

The following sections are specific to the *igb* or the *ixgbe* drivers.

### 1.4.1 IGB

1. Get the *igb* driver from sourceforge, compile and install.

Note that the extra CFLAGS allocate a separate handler for Tx cleanups. (This assumes an Ethernet adapter that uses the igb driver.).

```
wget http://downloads.sourceforge.net/e1000/igb-1.2.44.9.tar.gz

tar xzf igb-1.2.44.9.tar.gz

cd igb-1.2.44.9/src/

sudo make CFLAGS_EXTRA="-DCONFIG_IGB_SEPARATE_TX_HANDLER " install

cd ../..
```

2. Remove the old drivers.

```
rmmod igb
```

3. Add the module to the kernel using MSI-X interrupts and multiple queues.

```
modprobe igb IntMode=3
```

4. Make sure IRQ balance is turned off. This fails if the service isn't running, but the error can be ignored.

```
service irqbalance stop
```

5. Make sure iptables is off. This fails if the service isn't running, but the error can be ignored.

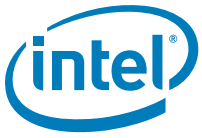
```
service iptables stop
```

6. Make sure the interface is up. The interface must be up for the interrupts to appear in */proc/interrupts*.

Check *ifconfig -all* to see which interfaces are assigned to the Ethernet adapter. This example assumes that the interfaces are *eth1* and *eth2*.

```
ifconfig eth1 up

ifconfig eth2 up
```



7. Assign interrupts to specific processors. Check the number of processor (cores) by checking */proc/cpuinfo*.

```
cat /proc/cpuinfo | grep processor
```

The output looks like:

```
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

8. Check the *proc filesystem* for interrupts.

The following table should appear in */proc/interrupts*:

| | CPU0 | CPU1 | CPU2 | CPU3 | | |
|---|---|---|---|---|---|---|
| 29 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth1-tx-0 |
| 30 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth1-tx-1 |
| 31 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth1-tx-2 |
| 32 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth1-tx-3 |
| 33 | 5 | 0 | 0 | 0 | PCI-MSI-edge | eth1-rx-0 |
| 34 | 5 | 0 | 0 | 0 | PCI-MSI-edge | eth1-rx-1 |
| 35 | 5 | 0 | 0 | 0 | PCI-MSI-edge | eth1-rx-2 |
| 36 | 5 | 0 | 0 | 0 | PCI-MSI-edge | eth1-rx-3 |
| 37 | 2 | 0 | 0 | 0 | PCI-MSI-edge | eth1 |
| 39 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth2-tx-0 |
| 40 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth2-tx-1 |
| 41 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth2-tx-2 |
| 42 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth2-tx-3 |
| 43 | 6 | 0 | 0 | 0 | PCI-MSI-edge | eth2-rx-0 |
| 44 | 6 | 0 | 0 | 0 | PCI-MSI-edge | eth2-rx-1 |
| 45 | 6 | 0 | 0 | 0 | PCI-MSI-edge | eth2-rx-2 |
| 46 | 6 | 0 | 0 | 0 | PCI-MSI-edge | eth2-rx-3 |
| 47 | 2 | 0 | 0 | 0 | PCI-MSI-edge | eth2 |

The interrupts can be assigned to specific processors by assigning a processor to the interrupts. The processor number is converted to a binary mask, so processor 0 is 1, processor 1 is 2, processor 2 is 4, processor 3 is 8 (2^processor_number).

To assign *eth1-tx-0 (29)* to processor 1, echo the converted mask to */proc/irq/29/smp_affinity*.

```
echo 2 > /proc/irq/29/smp_affinity
```

*Note:* The cores on one socket of a dual-socket system are assigned non-consecutive numbers. For example, on a dual-socket quad-core system Socket 0 has CPU 0, 2, 4, 6 and Socket 1 has CPU 1, 3, 5, 7.

### 1.4.2 IXGBE

1. Get the Ethernet driver from sourceforge, compile and install.

The extra CFLAGS disable Large Receive Offload (LRO). LRO increases performance by effectively letting the kernel handle fewer packet headers for the same amount of data but it causes the stack to fragment packets in the IP forwarding/routing case due to the large (effective) MTU size of the received packets.

Force compilation of NAPI (Rx polling mode).

*Note:* The shell variable *EXTRA_INC_DIR_SAVE* should be set to the *ioatdma* source directory when the *ioatdma* driver was installed.

```
wget http://downloads.sourceforge.net/e1000/ixgbe-1.3.31.5.tar.gz

tar xzf ixgbe-1.3.31.5.tar.gz

cd ixgbe-1.3.31.5/src

sudo make "CFLAGS_EXTRA=-DIXGBE_NO_LRO  -DIXGBE_NAPI" install

cd ../..
```

2. Remove the old drivers.

```
rmmod ixgbe
```

3. Add the module to the kernel. If available on the platform, add the module to the kernel using Receive Side Scaling (RSS). RSS=1 sets the descriptor queue count to the number of online processors up to a maximum of 16.

```
modprobe ixgbe RSS=1,1
```

4. Make sure IRQ balance is off. This fails if the service isn't running, but the error can be ignored.

```
service irqbalance stop
```

5. Make sure iptables is off. This fails if the service isn't running, but the error can be ignored.

```
service iptables stop
```

6. Make sure the interface is up. The interface must be up for the interrupts to appear in */proc/interrupts*. This example assumes that the interfaces are *eth1* and *eth2*.

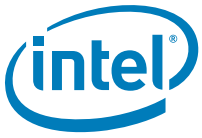
```
ifconfig eth1 up

ifconfig eth2 up
```

7. Reduce the transmit ring to 256 entries and the receive ring to 512 entries.

```
ethtool -G eth1 tx 256

ethtool -G eth1 rx 512
```



8. Assign interrupts to specific processors. Check the number of processor (cores) by checking */proc/cpuinfo*.

```
cat /proc/cpuinfo | grep processor
```

The output looks like:

```
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

9. Check the *proc filesystem* for interrupts.

The following table should appear in */proc/interrupts*:

| | CPU0 | CPU1 | CPU2 | CPU3 | | |
|---|---|---|---|---|---|---|
| 29 | 139 | 0 | 0 | 0 | PCI-MSI-edge | eth1-rx-0 |
| 30 | 21 | 118 | 0 | 0 | PCI-MSI-edge | eth1-rx-1 |
| 31 | 21 | 0 | 118 | 0 | PCI-MSI-edge | eth1-rx-2 |
| 32 | 21 | 0 | 0 | 118 | PCI-MSI-edge | eth1-rx-3 |
| 33 | 139 | 0 | 0 | 0 | PCI-MSI-edge | eth1-tx-0 |
| 34 | 21 | 55 | 0 | 63 | PCI-MSI-edge | eth1-tx-1 |
| 35 | 71 | 0 | 68 | 0 | PCI-MSI-edge | eth1-tx-2 |
| 36 | 21 | 63 | 50 | 5 | PCI-MSI-edge | eth1-tx-3 |
| 37 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth1:lsc |
| 38 | 89 | 0 | 0 | 0 | PCI-MSI-edge | eth2-rx-0 |
| 39 | 21 | 68 | 0 | 0 | PCI-MSI-edge | eth2-rx-1 |
| 40 | 21 | 0 | 68 | 0 | PCI-MSI-edge | eth2-rx-2 |
| 41 | 21 | 0 | 0 | 68 | PCI-MSI-edge | eth2-rx-3 |
| 42 | 89 | 0 | 0 | 0 | PCI-MSI-edge | eth2-tx-0 |
| 43 | 21 | 68 | 0 | 0 | PCI-MSI-edge | eth2-tx-1 |
| 44 | 84 | 0 | 5 | 0 | PCI-MSI-edge | eth2-tx-2 |
| 45 | 21 | 0 | 63 | 5 | PCI-MSI-edge | eth2-tx-3 |
| 46 | 0 | 0 | 0 | 0 | PCI-MSI-edge | eth2:lsc |

The easiest way to assign the interrupts to queues is to use the `set_irq_affinity.sh` script that is included in the latest released ixgbe drivers

The interrupts can be assigned to specific processors by assigning a processor to the interrupts. The processor number is converted to a binary mask, so processor 0 is 1, processor 1 is 2, processor 2 is 4, processor 3 is 8 (2^processor_number).

To assign *eth3:v0-Rx (2290)* to processor 1, echo the converted mask to */proc/irq/2290/smp_affinity*.

```
echo 2 > /proc/irq/2290/smp_affinity
```

*Note:* The cores on one socket of a dual-socket system are assigned non-consecutive numbers. For example, on a dual-socket quad-core system Socket 0 has CPU 0, 2, 4, 6 and Socket 1 has CPU 1, 3, 5, 7.

## 1.5 Initializing IP Forwarding

IP forwarding is initialized by default on reboot.

1. Edit */etc/sysctl.conf* and change the line:

```
net.ipv4.ip_forward = 0
```

to

```
net.ipv4.ip_forward = 1
```

2. Intitialize IP forwarding immediately.

```
sudo sysctl -w net.ipv4.ip_forward=1
```

3. Set rp_filter to zero.

```
net.ipv4.conf.eth0.rp_filter=0
```

and if desired:

```
net.ipv4.conf.all.rp_filter=0
```

# 2.0 Other Optimizations Found From Performance Testing

Testing has shown increased performance with some stack tuning on Linux operating systems. The tuning mainly involves increasing resources for the network stack. This includes increasing the TCP window size.

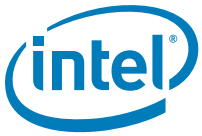
These parameters are configurable using the *sysctl* utility in Linux as indicated in this example:

```
sysctl -w net.core.rmem_default=524287
```

Running *sysctl* without the *-w* argument lists the parameter with its current setting. Though increasing the values can help improve performance, it is necessary to experiment with different values to determine what works best for a given system, workload and traffic type.

Note that these are only suggested starting points.

```
/proc/sys/net/core/rmem_default - default receive window
   (default=124928), suggested change to 524287
/proc/sys/net/core/wmem_default - default send window
   (default=124928), suggested change to 524287
/proc/sys/net/core/rmem_max - maximum receive window
   (default=131071), suggested change to 524287
/proc/sys/net/core/wmem_max - maximum send window
   (default=131071), suggested change to 524287
/proc/sys/net/core/optmem_max - maximum option memory buffers
   (default=20480), suggested change to 524287
```



```
/proc/sys/net/core/netdev_max_backlog -number of unprocessed input packets
before kernel starts dropping them

   (default=1000), suggested change to 300000

/proc/sys/net/ipv4/tcp_rmem - memory reserved for TCP rcv buffers (min default
max)

   (defaults 4096    87380   4194304), suggested change to 10000000 10000000
    10000000

/proc/sys/net/ipv4/tcp_wmem - memory reserved for TCP snd buffers (min default
max)

   (defaults 4096    16384   4194304), suggested change to 10000000 10000000
    10000000

/proc/sys/net/ipv4/tcp_mem - memory reserved for TCP buffers (min default max)

   (defaults 193152  257536  386304), suggested change to 10000000 10000000
   10000000
```



*Note:* This page intentionally left blank.