PCI Interrupts for x86 Machines under FreeBSD

John H. Baldwin
Yahoo!, Inc.
Atlanta, GA 30327
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

An important element in computers with multiple autonomous devices is the ability of a device to notify the CPU that it needs attention via an interrupt. The OS visible mechanics of interrupts for PCI devices is quite convoluted, especially on x86 PC systems. This paper will cover the various ways that PCI INTx interrupts have been implemented on x86 as well as the methods used by the system BIOS to communicate the implementation to operating systems. It will also cover the newer Message Signaled Interrupts that address some of the limitations of INTx interrupts. Finally, the paper will provide an overview of FreeBSD's implementation of both INTx and MSI interrupts on the x86 platform.

1 Introduction

Interrupts are an important part of device support in modern machines. They allow system software to use event-driven algorithms instead of polling. This in turn can allow for much more efficient use of system resources such as CPU time, especially in systems with multiple devices operating concurrently.

One of the tasks an operating system must perform when handling device interrupts is to determine which device needs attention when an interrupt is asserted. This mapping can be done several different ways. Some systems may include this mapping information in a static table hard-coded into the operating system or firmware. Other systems may use dynamic mappings with programmable hardware, and still other systems may use a combination.

PCI allows devices to assert interrupts in two very different styles. The first style consists of dedicated sideband signals and is known as Legacy INTx interrupts. The second style consists of special memory writes that are sent over the data bus just like DMA operations and is known as Message Signaled Interrupts (MSI). MSI interrupts are only available on newer devices as it is a recent addition to the PCI standards.

First, however, a brief segue into nomenclature. In PCI, the term "device" actually refers to a piece of hardware that contains from one to eight "functions". A typical x86 chipset may include several USB controllers as functions of a single PCI device for example. A PCI expansion slot is a PCI device, and a single expansion card may contain multiple functions. However, from the operating system's perspective, each function on a PCI device is a logical operating system device. For example, on both Darwin and FreeBSD, each function of a multi-function PCI device receives its own device object and can be served by different drivers. To attempt to avoid confusion, the rest of this paper will use the term "slot" for a PCI device and will only use "device" to refer to a logical operating system device.

2 Legacy PCI INTx Interrupts

As mentioned above, PCI INTx interrupts are implemented using side-band signals. Each PCI slot has four interrupt signals: `INTA#`, `INTB#`, `INTC#`, and `INTD#`. Each PCI function may use one of the interrupt signals associated with its slot. The function indicates which signal, if any, that it uses in one of the standard config space registers.

PCI interrupt lines are level-triggered and assert when the signal is pulled low. This allows multiple interrupt signals to share a single physical line.

3 Interrupts on x86 CPUs

Exceptions, Non-Maskable Interrupts (NMIs), Inter-Processor Interrupts (IPIs), and device interrupts all use the same interrupt mechanism on x86 CPUs. The operating system provides an Interrupt Descriptor Table (IDT) to the CPU that contains an array of handlers. When an interrupt is asserted, the CPU determines the associated IDT index, or vector. It can do this by asking an external interrupt controller for the IDT vector. If the interrupt was triggered by a message, the message will contain the IDT vector. Exceptions and NMIs are assigned static IDT vectors. Once the CPU has the IDT vector, it uses that vector as an index into the IDT. It then triggers the handler for that IDT slot. Thus, for an operating system to handle a device interrupt on an x86 CPU, it must know which device driver handlers are associated with a given IDT vector.

Thus, at one end we have a PCI interrupt line being signaled by a PCI function that needs attention. At the other end we have a CPU receiving an IDT vector. In the middle is one of those "and then a miracle occurs" mysteries. In this case the mystery hardware in the middle is known collectively as interrupt controllers.

4 Interrupt Controllers (Stuff in the Middle)

Interrupt controllers on the x86 platform are responsible for receiving interrupt signals from devices, mapping the signal to an IDT vector, and then interrupting one or more CPUs in the system with the IDT vector. The x86 platform has some extra quirks, however. First, due to limitations with the original interrupt controllers and busses used on the PC-AT, separate programmable interrupt routers were added to the platform to sit in between PCI interrupt signals and the PC-AT interrupt controllers. Second, as the x86 platform continued to progress, an entirely new set of interrupt controllers, known as Advanced Programmable Interrupt Controllers (APICs), were introduced. In order to preserve backwards compatibility, systems with APICs also still include the PC-AT interrupt controllers and either system can be used to handle interrupts on modern systems. It is even possible to use both at the same time though doing so is discouraged.

4.1 8259A Master and Slave PICs (PC-AT)

The original PC-AT included two 8259A Programmable Interrupt Controllers (PICs) chained together. As with many other aspects of the PC-AT, this setup became part of the de facto standard for the x86 platform. Each 8259A has eight interrupt input signals that are assigned to eight consecutive IDT vectors. Two PICs thus yields sixteen total interrupt inputs. On the PC-AT these signals were known as ISA IRQs 0 through 15. However, the second 8259A (known as the slave PIC) is connected to the third input pin on the first 8259A (known as the master PIC). Thus, ISA IRQ 2 is not available for device interrupts, and there are really only fifteen interrupt inputs available for device interrupts.

The original PC-AT also used an ISA bus for its devices. ISA interrupts are edge triggered and are asserted by having the device raise the signal from low to high. This inhibits sharing of ISA interrupts by multiple devices, so each ISA device requires a dedicated interrupt input on the 8259As. All PC-AT compatible systems included an ISA timer which used IRQ 0, a keyboard controller that used IRQ 1, and a real time clock which used IRQ 8. The optional float-

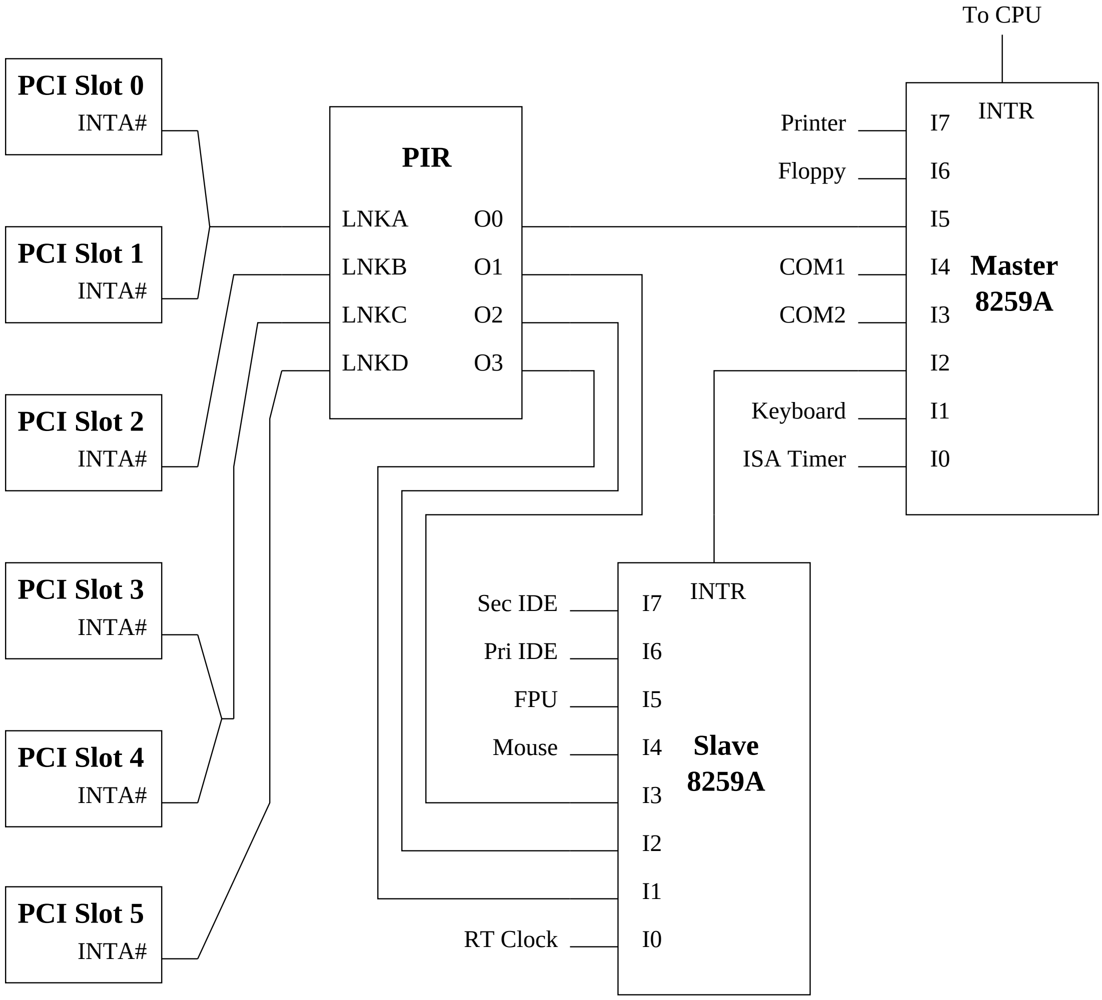

Figure 1: PCI Interrupts Routed via a Programmable Interrupt Router to 8259As

ing point co-processor used IRQ 13 if it was present. When PCI was introduced, most PC-AT compatible systems included two serial ports which used IRQs 3 and 4, a floppy controller which used IRQ 6, a line printer port which used IRQ 7, a PS/2 mouse port which used IRQ 12, and two IDE controllers which used IRQs 14 and 15. This left only IRQs 5, 9, 10, and 11 for PCI interrupts to use.

To make matters even more complicated many systems included other ISA devices such as additional serial or printer ports or sound cards. Each of these additional ISA devices also required a dedicated IRQ. Sound cards, for example, often used IRQ 5. Thus, the set of IRQs available for use by PCI interrupts was not fixed. Rather, simply adding or removing an ISA peripheral could change the set across reboots. To deal with this complication, programmable interrupt routers were added.

4.2 Programmable Interrupt Routers (PCI Link Devices)

A programmable interrupt router is used to route PCI interrupt signals to interrupt inputs on another interrupt controller. A router contains several input signals and output signals. Each output signal is hooked up to an input on an interrupt controller. Each input signal can be routed to one of the output signals. Multiple input signals can be routed to the same output signal. Multiple PCI interrupt signals can be routed to a single input signal.

System software, such as the BIOS or operating system, is responsible for programming the interrupt router. Programming the router consists of routing each input signal that is in use to an output signal. For example, a PC-AT compatible system might have the output pins from the programmable interrupt router hooked up to the interrupt lines for IRQs 3, 4, 5, 7, 9, 10, and 11 on the 8259A PICs. If IRQs 3, 4, and 7 are in use by ISA devices, then each input pin on the interrupt router must be routed to one of the IRQs 5, 9, 10, or 11. Note that routing a single input pin (also known as a PCI Link Device) to an IRQ routes all of the PCI interrupts connected to that input pin to the IRQ. That is, one can't arbitrarily route individual PCI interrupts at will. Instead, one can only route a group of PCI interrupts that are connected to an input pin. This grouping is set in the hardware and cannot be altered by system software. Also, if a system has more input pins than available IRQs, then at least some of the input pins will be routed to the same IRQ resulting in the union of all the PCI interrupts on those input pins sharing the same IRQ.

An example of using a programmable interrupt router in combination with the 8259A PICs can be found in Figure 1. On the left are six PCI slots. The `INTA#` pin from each slot is connected to one of the input pins on the interrupt router, which is labeled "PIR." Notice that some of the PCI interrupts are wired to the same input pin on the interrupt router. For example, the first two interrupts are both connected to the `LNKA` input pin. The interrupt router also has four output pins that are connected to input pins on the two 8259A controllers corresponding to IRQs 5, 9, 10, and 11. Suppose that `LNKA` is routed to `O0`, `LNKB` is routed to `O1`, `LNKC` is routed to `O2`, and `LNKD` is routed to `O0`. Then the interrupts for PCI slots 0, 1, and 5 would be routed to IRQ 5. The interrupt for PCI slot 2 would be routed to IRQ 11, and the interrupts for PCI slots 3 and 4 would be routed to IRQ 10.

4.3 I/O APICs

Many x86 systems, including most recent systems, include a second set of interrupt controllers known as APICs. In these systems, each CPU includes a local APIC which receives interrupt messages and uses them to assert interrupts on the CPU. The chipset includes one or more I/O APICs which are responsible for converting device interrupt signals into messages that are delivered to one or more local APICs.

One of the biggest differences between the 8259A PICs and I/O APICs is that the pins on I/O APICs are completely independent. With the 8259A PICs, the eight input pins are mapped to eight consecutive IDT vectors, and all of the interrupts are sent to the same CPU. I/O APICs, on the other hand, do not have any controller-wide settings. Instead, each pin is programmed independently. Each pin is assigned its own IDT vector by the operating system and can be mapped to one or more CPUs. I/O APICs can also contain a variable number of pins. Typically an I/O APIC contains 16, 24, or 32 input pins.

PCI interrupt signals are routed to I/O APIC interrupt pins in several different ways. The earliest APIC systems include a single I/O APIC with 16 input pins that simply replicate the functionality of the two 8259A PICs. In these systems, the 16 input pins are used for the 16 ISA IRQs, and PCI interrupts are routed onto the ISA IRQs using a programmable interrupt router. Most APIC systems, however, use dedicated I/O APIC input pins for PCI interrupt signals. In these systems, the first 16 pins on the first I/O APIC are used for the 16 ISA IRQs. PCI interrupt signals are connected to other input pins on the first I/O APIC (if it contains more than 16 input pins) as well as input pins on additional I/O APICs (if present). Some very recent systems have begun routing some of the PCI interrupt signals to a programmable interrupt router whose output pins are connected to a set of I/O APIC input pins that are dedicated to PCI interrupts.

Figure 2 contains an example of a system with a 24-pin I/O APIC where the PCI in-

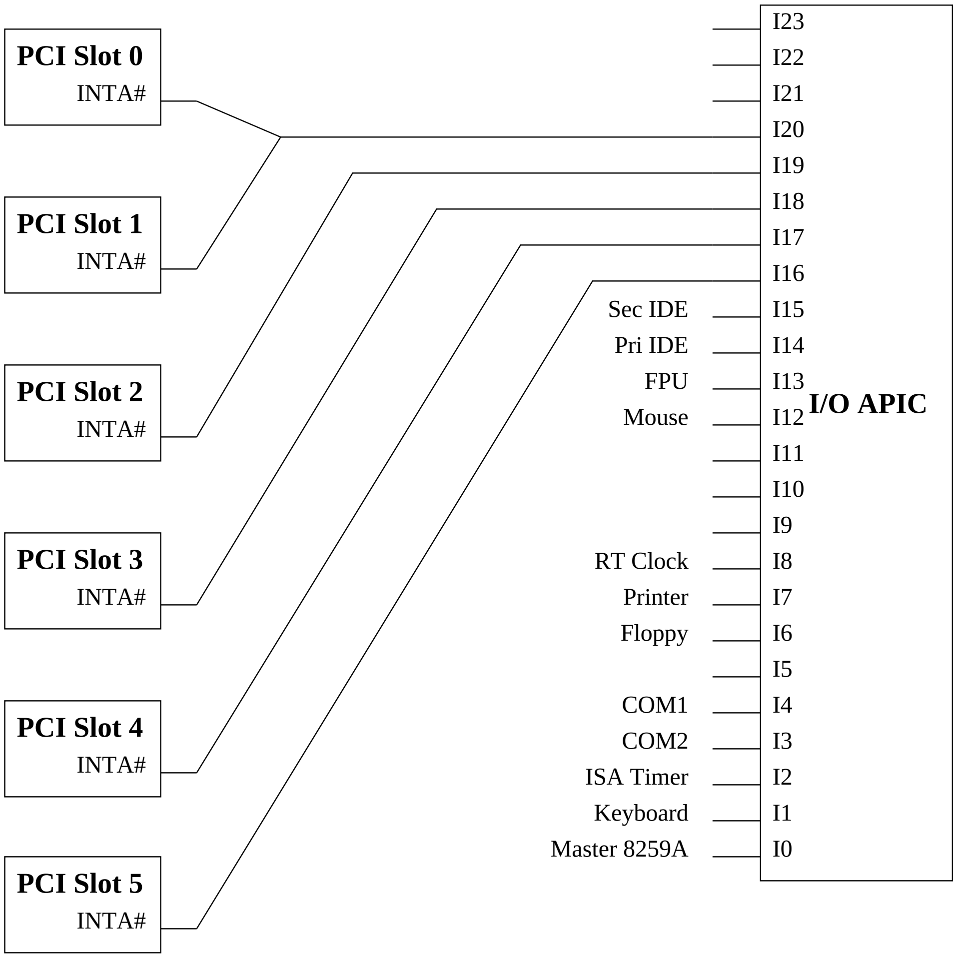

Figure 2: PCI Interrupts Routed Directly to an I/O APIC

terrupts are connected directly to input pins on the I/O APIC. As with the previous figure, there are six PCI slots on the left, and the `INTA#` pin from each slot is connected to one of the input pins on the I/O APIC. A system may still have shared PCI interrupts even when using APICs if multiple interrupt lines are connected to the same I/O APIC input pin. For example, the first two interrupts for PCI slots 0 and 1 are both connected to `I20` input pin. The other interrupts are all routed to their own input pin: PCI slots 2, 3, 4, and 5 are routed to input pins `I19`, `I18`, `I17`, and `I16`, respectively.

5 PCI Interrupt Routing (Navigating the Maze)

PCI interrupt routing consists of figuring out which platform-specific interrupt is asserted when a given PCI interrupt signal is asserted. On x86 machines, this consists of figuring out which input pin on an interrupt controller is asserted when a given PCI interrupt signal is asserted. This can include a detour through a programmable interrupt router in between the PCI interrupt signal and the interrupt controller.

5.1 PCI-PCI Bridge Swizzle

Most PCI interrupt routing information is platform-specific; however, the PCI specification does define a specific case where interrupt routing is platform independent. Specifically, if an add-in card contains its own PCI-PCI bridge, then the interrupt pins are for the PCI slots behind the PCI-PCI bridge are mapped onto the interrupt pins on the upstream side of the PCI-PCI bridge.

If you assign the values 0 - 3 to `INTA#` - `INTD#` then the mapping can be described as:

$$pin_{parent} = (slot_{child} + pin_{child}) \bmod 4$$

where pin_{parent} is the upstream interrupt pin on the PCI-PCI bridge and $slot_{child}$ and pin_{child} are the PCI slot and pin, respectively, of the interrupt signal being routed across the bridge. Thus, `INTA#` of slot 0 is mapped to `INTA#` on the bridge. For slot 1, `INTA#` is mapped to `INTB#` on the bridge, and `INTD#` is mapped to `INTA#` on the bridge. A full table of the mapping can be found in Table 24-13 of *PCI System Architecture* [5].

Although the PCI specification only defines this routing for PCI-PCI bridges in add-in cards, some x86 systems also use this mapping for PCI slots behind PCI-PCI bridges that are not in add-in cards but are part of the main chassis. The rule there appears to be that if no other routing information is provided by the BIOS for a given PCI bus behind a PCI-PCI bridge, the above mapping should be used to route the interrupts across the PCI-PCI bridge to the parent PCI bus.

5.2 Routing via $PIR

The first PCI interrupt routing table provided by the x86 BIOS is the $PIR table, so named for it's four character signature: "$PIR". This table describes how PCI interrupt signals are connected to input pins on a programmable interrupt router. In addition, it provides details that can be used by the operating system to either program the interrupt router directly or to ask the PCI BIOS to route individual links. Note that the $PIR table only knows about ISA IRQs, so it cannot in general be used with APIC. An exception to this rule is that some early APIC systems only route ISA IRQs via their sole I/O APIC and still use an programmable interrupt router and a $PIR table to route PCI interrupts onto ISA IRQs. Details on the format and layout of the $PIR table can be found in pages 233 - 238 of *PCI System Architecture* [5].

5.2.1 The $PIR Table

The main body of the $PIR table consists of a variable-sized array of slot entries. Each slot entry contains details about a single PCI slot such as whether or not the slot is an embedded device in the main chassis or whether the slot represents a physical slot on the motherboard. Each slot entry also contains an array of four pin entries containing the routing information for `INTA#`, `INTB#`, `INTC#`, and `INTD#` for the slot. Each pin entry contains a byte holding a link identifier and a bitmap of valid ISA IRQs for this pin entry. Since each slot entry always contains pin entries for all four pins, but not all slots on embedded devices use all four pins, a pin can be marked as disconnected by using a link identifier of zero.

Each non-zero link identifier refers to a specific input pin on the programmable interrupt router. Thus, all of the (slot, pin) entries that have the same link identifier are all physically connected to the same input pin and will always share the same interrupt. Also, routing a link to an IRQ routes all of the (slot, pin) entries connected to that link to that IRQ.

Routing a PCI interrupt to an ISA IRQ using $PIR is a mostly straightforward process. First, the bus, slot, and pin of the PCI interrupt to be routed are used to lookup a (slot, pin) entry in the $PIR table. The link for the PCI interrupt is then taken from the (slot, pin) entry. If the link is already routed to an IRQ, then the routing is complete. Otherwise, an IRQ must be chosen and the link must be routed to the IRQ.

Once the operating system has picked an IRQ for an unrouted link, there are two ways the link can be routed to that IRQ. First, the operating system can manually program the interrupt router to route the link. Second, the PCI BIOS provides a BIOS call to route an individual link to a specific IRQ.

To aid with the first approach, the $PIR table includes the PCI location (bus, slot, and function) and device ID of the interrupt router. The device ID can be used to determine an interrupt router driver. For some routers, for example, the programming model is that the link identifier in the (slot, pin) entries is the offset of a single byte register and that the IRQ to be used is written to the register to route the link. This approach requires the operating system to maintain several different interrupt router drivers, and new routers aren't supported until a new driver is written or an existing driver updated.

5.2.2 FreeBSD's $PIR Implementation

To support $PIR routing, FreeBSD maintains data about each link that appears in the $PIR table. During boot, the kernel walks the $PIR table to build a list of links and allocate per-link data such as whether or not the link is routed, and which IRQ it is routed to. The kernel also determines if each link has already been routed by the BIOS (and if so, to which IRQ). To do this, it walks each slot entry in the $PIR table. It then examines the intpin and intline PCI config registers for each function at the PCI bus and slot specified in the slot entry. If a function has a valid intline config register, then the kernel assumes that the BIOS has routed the link associated with this function to the IRQ in the intline register. It determines the link by using the intpin register to index the pin entry in the $PIR slot entry. The kernel also builds the bitmap of valid IRQs for each link during this table walk. In theory, all of the (slot, pin) entries that refer to the same link should have the same bitmap of valid IRQs. In practice, this is not always the case. As a result, FreeBSD uses the intersection of the various bitmaps (via a bitwise AND) to determine the bitmap of valid IRQs for a given link.

The trickiest part of $PIR routing is figuring out which IRQ to use for a link that was not routed by the BIOS. FreeBSD uses a rather conservative approach that favors reliability at the expense of possibly forcing more interrupt sharing than is strictly necessary. If the link has only one valid IRQ in its bitmap, then that IRQ is used. However, if there is more than one valid IRQ, the process is slightly more complex. When the kernel does its initial scan of the $PIR table to build the per-link data, it also builds a bitmap of IRQs that the BIOS used to route links. Then when it comes time to pick an IRQ for an unrouted link, the kernel will try to use one of the IRQs that the BIOS used as those IRQs are known to work. If none of the IRQs used by the BIOS are valid for the link, then the kernel tries to use one of the IRQs in a system-wide bitmap of PCI-only IRQs in the $PIR header. If the kernel is still not able to find an IRQ at this point, it tries to pick an IRQ from a static bitmap of IRQs.

FreeBSD just uses the PCI BIOS call to route a link to an IRQ to avoid the maintenance overhead of interrupt router drivers. While some older systems may not include the BIOS call, these older systems also will typically route all the links to IRQs during boot (or provide a BIOS option to do so). Thus, for these systems the ability to route links is not critical.

An example of the link table portion of the output from `pirtool` can be find in Figure 3. This machine contains at least six different links: 0x60, 0x61, 0x62, 0x63, 0x68, and 0x6b. The first entry indicates that the `INTA#` pin for slot 2 on bus 0 is connected to the 0x60 link, for example. This machine only has one expansion slot, and the four intpins in that slot are routed to links 0x62, 0x63, 0x60, and 0x61.

FreeBSD allows the user to tweak the $PIR routing via loader tunables. First, the bitmap of IRQs used as the last resort when choosing an IRQ for an unrouted IRQ can be set via `hw.pci.irq_override_mask`. Probably more useful, however, is the ability to specify the

Entry	Location	Bus	Device	Pin	Link	IRQs
0	embedded	0	2	A	0x60	3 4 5 6 10 11 14 15
1	embedded	1	0	A	0x60	3 4 5 6 10 11 14 15
1	embedded	1	0	B	0x61	3 4 5 6 10 11 14 15
1	embedded	1	0	C	0x62	3 4 5 6 10 11 14 15
1	embedded	1	0	D	0x63	3 4 5 6 10 11 14 15
2	embedded	0	29	A	0x60	3 4 5 6 10 11 14 15
2	embedded	0	29	B	0x63	3 4 5 6 10 11 14 15
2	embedded	0	29	C	0x62	3 4 5 6 10 11 14 15
2	embedded	0	29	D	0x6b	3 4 5 6 10 11 14 15
3	embedded	0	31	A	0x62	3 4 5 6 10 11 14 15
4	embedded	4	13	A	0x61	3 4 5 6 10 11 14 15
5	embedded	2	4	A	0x60	3 4 5 6 10 11 14 15
6	embedded	4	3	A	0x68	3 4 5 6 10 11 14 15
7	slot 1	3	7	A	0x62	3 4 5 6 10 11 14 15
7	slot 1	3	7	B	0x63	3 4 5 6 10 11 14 15
7	slot 1	3	7	C	0x60	3 4 5 6 10 11 14 15
7	slot 1	3	7	D	0x61	3 4 5 6 10 11 14 15

Figure 3: $PIR Link Table from `pirtool` on a Dell SC 1425

IRQ for individual links. The tunable for this is `hw.pci.link.`*`link`*`.irq` where *link* is the link to route, and the value of the tunable is the IRQ to route the link to. The *link* is specified as a hexadecimal value with a leading "0x" identical to the output from `pirtool` as well as all kernel output containing $PIR link identifiers. For example, to route link 0x60 to IRQ 11, set the loader tunable `hw.pci.link.0x60.irq=11`. Any IRQs used in user-specified routings are included in the mask of IRQs used by the BIOS when choosing IRQs for unrouted links. Thus, if a BIOS routes only some links and routes them all to IRQ 10, but IRQ 11 is known to be free for PCI interrupts, one can force the system to route one or more of the unrouted links to IRQ 11 to split some of the interrupt load simply by routing a single link to IRQ 11 using a tunable.

5.3 Routing via the MP Table

When APIC was introduced on x86, a new table was introduced to describe the routing of PCI interrupts to I/O APIC input pins. This table is known as the MP Table, and its format is described in the Multiprocessor Specification [4]. The MP Table contains several different types of entries including entries enumerating processors and I/O APICs. It also contains entries describing all device interrupts that are connected to I/O APIC input pins, and these entries are used for routing PCI interrupts.

Each interrupt entry in the MP Table contains a device interrupt as the source, and an I/O APIC input pin as the destination. The source interrupt contains a bus type (such as ISA or PCI), a bus ID, and an IRQ value. For PCI interrupts, the bus ID is the PCI bus number of the slot the interrupt belongs to. The IRQ value contains both the PCI slot and intpin. The lower two bits contain the intpin (0 means `INTA#`, etc.), and bits 2 through 6 contain the slot. The destination is specified as an APIC ID of an I/O APIC and an input pin number.

Given this, routing a PCI interrupt using the MP Table is fairly simple. The kernel simply walks the MP Table until it finds an entry whose source interrupt matches the PCI bus, slot, and intpin. It then uses the destination I/O APIC input pin as the PCI interrupt's destination.

An example of the interrupt entries output from `mptable(8)` can be find in Figure 4. This system contains three I/O APICs with APIC IDs of 8, 9, and 10. The ISA IRQs

```
I/O Ints:       Type    Polarity    Trigger     Bus ID    IRQ    APIC ID PIN#
                ExtINT  active-hi        edge        5      0          8    0
                INT       conforms    conforms       5      1          8    1
                INT       conforms    conforms       5      0          8    2
                INT       conforms    conforms       5      3          8    3
                INT       conforms    conforms       5      4          8    4
                INT       conforms    conforms       5      7          8    7
                INT       conforms    conforms       5      8          8    8
                INT       conforms    conforms       5      9          8    9
                INT       conforms    conforms       5     12          8   12
                INT       conforms    conforms       0    2:A          8   16
                INT       conforms    conforms       0   29:A          8   16
                INT       conforms    conforms       0   29:B          8   19
                INT       conforms    conforms       0   29:D          8   23
                INT       conforms    conforms       0   31:A          8   18
                INT       conforms    conforms       4   13:A          8   17
                INT       conforms    conforms       4    3:A          8   20
                INT       conforms    conforms       2    4:A          9    0
                INT       conforms    conforms       3    7:A         10    2
                INT       conforms    conforms       3    7:B         10    3
                INT       conforms    conforms       3    7:C         10    0
                INT       conforms    conforms       3    7:D         10    1
```

Figure 4: MP Table Interrupt Entries from `mptable(8)` on a Dell SC 1425

are connected to the first sixteen pins on the first I/O APIC. The PCI interrupts for PCI busses 0 and 4 are connected to other pins on the first I/O APIC. The only PCI interrupt on PCI bus 2 is connected to the first pin on the second I/O APIC, and the PCI interrupts on PCI bus 3 are connected to the first four pins on the third I/O APIC.

5.4 Routing via ACPI

The Advanced Configuration and Power Interface (ACPI) was developed in part to provide a unified interface to configuration management for x86 machines [1]. Both the $PIR table and the MP Table are among the configuration methods and tables merged into the ACPI umbrella. Thus, ACPI provides a unified interface to PCI interrupt routing on x86 that replaces both $PIR and the MP Table. There are four main components to ACPI's interrupt routing support: global system interrupts (GSIs), the global `_PIC` method, PCI interrupt link devices, and per-PCI bus `_PRT` methods.

5.4.1 Global System Interrupts

ACPI uses a cookie system to "name" interrupts known as Global System Interrupts. Each interrupt controller input pin is assigned a GSI using a fairly simple scheme. For the 8259A case, the GSIs map directly to ISA IRQs. Thus, IRQ 0 is GSI 0, etc. The APIC case is slightly more complicated, but still simple. Each I/O APIC is assigned a base GSI by the BIOS. Each input pin on the I/O APIC is mapped to a GSI number by adding the pin number (zero-based) to the base GSI. Thus, if an I/O APIC has a base GSI of N, pin 0 on that I/O APIC has a GSI of N, pin 1 has a GSI of N + 1, etc. The I/O APIC with a base GSI of 0 maps the ISA IRQs onto its first 16 input pins. Thus, the ISA IRQs are effectively always mapped 1:1 onto GSIs. More details about GSIs can be found in Section 5.2.11 of the ACPI 2.0c spec [2].

5.4.2 The `_PIC` Method

The global `_PIC` method is a global ACPI function that the operating system calls during boot to inform ACPI which set of interrupt hardware it plans to use. It accepts a single input parameter which indicates the interrupt mode to use. For x86 platforms, the parameter has two possible values: 0 indicates PIC mode (that is, 8259As), and 1 indicates APIC mode. The mode defaults to PIC mode on boot. Typically, the implementation of `_PIC` saves the interrupt mode in a global variable that is used by subsequent calls to `_PRT` methods to determine which routing tables to return. Figure 5 contains a sample implementation of `_PIC`. More details can be found in Section 5.8.1 of the ACPI 2.0c spec [2].

5.4.3 PCI Interrupt Link Devices

Each input pin on a programmable interrupt router is represented as a device in the ACPI namespace. These devices are called PCI interrupt link devices and are identified with a PnP ID of `PNP0C0F`. The destination of the link is set via an IRQ resource in the device's resource list. It is managed using the `_CRS`, `_PRS`, and `_SRS` methods just like the IRQ resource is managed for other ACPI devices such as built-in serial ports on the LPC bus.

FreeBSD manages PCI interrupt link devices in much the same way it handles links in the $PIR table. One difference is that ACPI provides a standard way to query the link for its current setting via the `_CRS` method. In practice, however, `_CRS` is not always implemented, so the PCI interrupt link driver does examine the PCI intline config register of any devices that use a link to determine which IRQ, if any, the BIOS routed the link to. Another difference is that ACPI allows links to be turned off completely via the `_DIS` method. In fact, FreeBSD turns off all links during boot and explicitly turns them on only when they are needed to route interrupts.

Routing a link to an ISA IRQ follows much the same process for both PCI interrupt link devices and $PIR links. One difference is that all PCI interrupt link devices are disabled during boot, so all of them must be routed to an IRQ on first use. However, the PCI interrupt link device will simply route the link to the IRQ assigned by the BIOS if the BIOS assigned one during boot and that IRQ is still valid (the IRQ might not be valid if a link device is used in APIC mode and has a different set of valid GSIs in APIC mode than in PIC mode). The PCI interrupt link driver builds a mask of known-good ISA IRQs from the IRQs used by the BIOS similar to the $PIR driver. However, ACPI doesn't provide a global mask of PCI-only IRQs similar to the one found in the $PIR table header. The PCI interrupt link driver also does not include a static mask of possible ISA IRQs or honor the `hw.pci.irq_override_mask` sysctl. On the other hand, the ACPI System Control Interrupt (SCI) is always shareable with PCI interrupts, so it is added to the mask of known-good ISA IRQs when in PIC mode. Thus, there is always at least one IRQ in that mask when operating in PIC mode, removing the need for both the global mask of PCI-only IRQs and the static mask of possible ISA IRQs.

Routing a link to a non-ISA GSI is fairly simple. In that case, there aren't any complications with trying to determine which of the possible interrupts really are usable. Instead, all of the possible GSIs are usable, and a simple weighting scheme is used to balance the links across the available GSIs.

As with the $PIR code, the PCI interrupt link driver allows the user to override the routing for individual links via tunables. The format of the tunable is `hw.pci.link.`*`link`*`.irq` where *link* is the link to route, and the value of the tunable is the GSI to route the link to. The *link* is specified as the last component of the PCI interrupt link devices path in the ACPI namespace. For example, to route the PCI interrupt link device `_SB_.PCI0.ISA0.LNKA` to ISA IRQ 11, set the loader tunable `hw.pci.link.LNKA.irq=11`. Unlike the $PIR code, any IRQs used in user-specified routings are *not* included in the mask of known-good ISA IRQs used by the BIOS.

```
Scope (\)
{
    Name (PICF, 0x00)
    Method (_PIC, 1, NotSerialized)
    {
        Store (Arg0, PICF)
    }
}
```

Figure 5: ACPI _PIC method from a Dell SC 1425

5.4.4 _PRT Methods

The meat of PCI interrupt routing via ACPI is the _PRT method. Each PCI bus that ACPI provides interrupt routing information for appears as a device in the ACPI namespace. Each of these devices contains a _PRT method that returns an array of objects describing the interrupt routing for slots on that PCI bus.

Each object contains four members that describe the routing for a single PCI interrupt. The first member is an ACPI PCI address using the same format as _ADR. Thus, the upper four bytes contain the slot, and the lower four bytes contain the function. Since the PCI function is not part of a PCI interrupt's address, it is always specified as a wildcard value of `0xFFFF` and should be ignored by the operating system. The second member is a single byte indicating the intpin. A value of 0 specifies `INTA#`, 1 specifies `INTB#`, etc.

The third and fourth members define the destination of the interrupt. If the third value is either zero or an empty string, then the fourth value is a GSI and the PCI interrupt is hard-wired to that GSI. This type of mapping is just like the MP Table I/O interrupt entries which map PCI interrupts to specific I/O APIC input pins. If the third value is not empty, then it is the name of a PCI interrupt link device in the ACPI namespace, and the fourth value is a resource index. The resource index indicates which of the resources of the PCI interrupt link device the interrupt is connected to. In practice, though, some BIOSes bogusly include a name of a link device even in entries that are routed via a hardwired GSI, and no systems to date use a resource index other than zero. Thus, as a workaround for the busted systems, FreeBSD ignores the third value and assumes the fourth value is a hardwired GSI if the fourth value is not zero. The _PRT method is described in further detail in Section 6.2.8 of the ACPI 2.0c spec [2].

Figure 6 contains a portion of an example _PRT. Specifically, it includes the first entry in the table. This corresponds to the PCI interrupt for PCI bus 3, slot 7, `INTA#` and can be compared with the routing for this same interrupt in Figures 3 and 4. First, note there are actually two routing tables declared as constants: `PIC3` is used for PIC mode, and `APC3` is used for APIC mode. The _PRT method checks the value of the `PICF` global variable to determine which table to return to the operating system. Recall that `PICF` is set in the _PIC method in Figure 5. For PIC mode, the interrupt is routed to the `LNKC` device, which is an ACPI PCI interrupt link device corresponding to link 0x62 from the $PIR table. For APIC mode, the interrupt is routed to GSI 66. For this machine, ACPI assigns a base GSI of 64 to the I/O APIC with an APIC ID of 10. Thus, GSI 66 corresponds to pin 2 on that I/O APIC which matches the routing in the MP Table for this PCI interrupt.

Note that ACPI allows PCI interrupt link devices to be specified in _PRT entries in APIC mode. Thus, ACPI allows the BIOS to communicate an interrupt routing where a PCI interrupt signal is connected to an input pin on a programmable interrupt router whose output pins are connected to arbitrary input pins on one or more I/O APICs. Neither the $PIR table or MP Table allow for this configuration since the $PIR table assumes all IRQ values are less than 16, and the MP Table assumes all PCI interrupts are mapped to

```
Device (PXHB)
{
    ...
    Name (PIC3, Package (0x04)
    {
        Package (0x04)
        {
            0x0007FFFF,
            0x00,
            LNKC,
            0x00
        },
        ...
    })
    Name (APC3, Package (0x04)
    {
        Package (0x04)
        {
            0x0007FFFF,
            0x00,
            0x00,
            0x42
        },
        ...
    })
    Method (_PRT, 0, NotSerialized)
    {
        If (LNot (PICF))
        {
            Store (PIC3, Local0)
        }
        Else
        {
            Store (APC3, Local0)
        }

        Return (Local0)
    }
}
```

Figure 6: Excerpt of ACPI _PRT method from a Dell SC 1425 for PCI3:7:INTA#

hardwired I/O APIC input pins.

6 FreeBSD's INTx Implementation

FreeBSD's PCI interrupt routing code attempts to provide a machine independent framework that machine dependent code can hook into where necessary. First, FreeBSD uses cookie values defined by machine dependent code for `SYS_RES_IRQ` resources. This provides a way to handle interrupts in machine independent code and interfaces. Second, when the PCI bus needs to route an interrupt it passes the request up the device tree until it reaches a level where the request can be handled.

All interrupt resources in FreeBSD drivers are managed as `SYS_RES_IRQ` resources. When a driver wants to use an interrupt, it allocates a `SYS_RES_IRQ` resource in much the same way it allocates memory or I/O space. The driver can then attach an interrupt handler to that resource. When a PCI device attempts to allocate a INTx interrupt, the PCI bus first routes it to an IRQ value that is used to create the `SYS_RES_IRQ` resource. It does this by asking its parent device, which is either a Host-PCI or PCI-PCI bridge, to look up the IRQ for the given PCI interrupt. The different interrupt routing algorithms are then implemented in different drivers for Host-PCI and PCI-PCI bridge drivers.

The simplest PCI bridge driver is the PCI-PCI bridge driver. This driver's interrupt routing routine implements the swizzle defined in Section 5.1 by calculating the corresponding slot and pin on the upstream side of the bridge and passing the request up to the PCI bridge driver for the upstream PCI bus. Thus, routing requests for interrupts on busses that are not part of the main chassis will bubble up through the device tree until they hit a bridge for a PCI bus that is part of the main chassis.

Interrupt routing for PCI busses that are part of the main chassis is handled by machine dependent PCI bridge drivers. For example, if ACPI is enabled, then ACPI will probe and attach to all the PCI bridges in the ACPI namespace. When an interrupt routing request reaches a PCI bridge with an ACPI driver, it will use the `_PRT` for the corresponding PCI bus to determine the GSI for the PCI interrupt. It then maps the GSI to a `SYS_RES_IRQ` cookie value which it returns. Thus, the machine dependent code is responsible for mapping platform-specific interrupts to `SYS_RES_IRQ` cookies in the PCI bridge drivers. Then in the top-level root, or nexus, devices in the device tree, the machine dependent code is responsible for mapping the `SYS_RES_IRQ` resources back to the platform-specific interrupts.

FreeBSD does allow the user to override the IRQ for any given PCI interrupt via a tunable. The format for this tunable is `hw.pci`*`bus`*`.`*`slot`*`.INT`*`pin`*`.irq` where *bus* is the PCI bus number, *slot* is the PCI slot number, and *pin* is the intpin (`A`, `B`, `C`, or `D`). The value of the tunable is the IRQ to use for the specified PCI interrupt. This tunable should only be used as a last resort when there aren't more specific tunables (such as the PCI link tunables) available. One instance in which this tunable is useful is correcting hard-wired routing to I/O APIC intpins due to a broken MP Table or `_PRT` entry. For example, to route the PCI interrupt for bus 0, slot 16, `INTA#` to IRQ 24, set the loader tunable `hw.pci0.16.INTA.irq=24`.

6.1 IRQs are Yummy Cookies

For the x86 platforms, FreeBSD models the mapping of IRQ values to platform interrupts on the Global System Interrupts approach from ACPI. In fact, when using ACPI FreeBSD uses the GSI values directly as IRQs. FreeBSD also always maps IRQ values 0 through 15 to the sixteen ISA IRQs. The only remaining case is when using the MP Table to enumerate APICs and route interrupts. For this case, the MP Table code simulates the GSI approach by assigning suitable base IRQ values to each I/O APIC similar to th base GSI values used by ACPI. The MP Table code calculates the base IRQs by adding the number of input pins on each I/O APIC to the base IRQ of the current I/O APIC to determine the base IRQ of the next I/O APIC. Thus, if you have a system with three I/O APICs where the first two I/O APICs have 24 pins and the third I/O APIC has 16 pins, the first I/O APIC would be assigned IRQs 0-23, the second I/O APIC would be assigned IRQs 24-47, and the last I/O APIC would be assigned IRQs 48-63.

The x86 platforms use a global array indexed by the IRQ value to map the IRQs to platform interrupts. Each entry in the array is a pointer to an interrupt source object. Interrupt source objects consist of a `struct intsrc` which contains a pointer to a group of function pointers in a `struct pic`. One can think of `struct intsrc` and `struct pic` as abstract base classes. Each interrupt controller driver provides its own extended versions of `struct pic` and `struct intsrc`. The extended versions contain the base structure as the first member and add driver-specific data after that. For example, the I/O APIC code defines a `struct ioapic` which extends `struct pic`. Each instance of `struct ioapic` contains functions for managing I/O APIC input pins in its method table. It also defines a `struct ioapic_intsrc` which extends `struct intsrc` to add I/O APIC-specific data such as which I/O APIC input pin an interrupt source represents. The interrupt controller drivers determine the IRQ values for each interrupt source object. Thus, they must ensure the IRQ properly matches up with the IRQ value used for any PCI interrupts routed to that interrupt source.

6.2 IDT Vectors on x86

Once the operating system has mapped a PCI interrupt to an interrupt source, the only remaining step for x86 platforms is mapping the interrupt source to an IDT vector. IDT vectors range from 0 to 255, and IDT vectors 0-31 are reserved for CPU faults and exceptions and NMIs. In addition, FreeBSD uses vectors 240-255 for IPIs, vector 239 for the local APIC timer interrupt, and vector 128 for system calls. That leaves vectors 32-127 and 129-238 for device interrupts.

The 8259As each require 8 contiguous IDT vectors. They each can also interrupt the CPU even when all input pins are masked if a spurious interrupt occurs. Thus, vectors 32-47 are reserved for the 8259As, even when APICs are used instead of the 8259As.

The rest of the device interrupts are allocated on an as-needed basis to active interrupt sources. For example, I/O APIC input pins allocate an IDT vector the first time an interrupt handler is registered. Most I/O APIC input pins are never used, so this strategy avoids reserving IDT vectors for interrupt sources that will never trigger.

7 PCI Message Signaled Interrupts

Legacy PCI INTx interrupts work, but they have several limitations. First, each PCI function is only allowed a single interrupt. Second, PCI INTx interrupts use a separate signal from the address and data lines used for PCI data transactions.

The single interrupt restriction can be a bottleneck for high-performance devices. For example, on some Ethernet adapters, the transmit and receive units run in parallel, but a single interrupt forces the driver to process events from the two units serially. Another case where a device can benefit from multiple handlers is a device that generates interrupts for specific performance-critical events very often while also generating interrupts for other events less often. An Ethernet device can be an example of this as well. A busy Ethernet device will generate several receive and transmit completion interrupts while handling traffic, and it can also generate other interrupts for events like link status changes. If the device were able to split out the receive and transmit completion events into dedicated interrupts, those interrupt handlers could be smaller and faster allowing for less overhead for those events.

Using a separate signal from the normal

address and data lines for PCI INTx interrupts raises several issues. First, on many x86 systems this requires separate physical traces on the motherboard to connect the signals to interrupt controller input pins. Second, the platform and operating system have to work together to route the interrupts. The largest issue, however, is that by using separate signals, the interrupt may be raised on the CPU before all of the effects of the event that triggered the interrupt are visible to the CPU. As a result, all PCI device driver interrupt handlers must begin with a read from a register on the PCI device. This read will not be completed until any pending transactions in between the CPU and the PCI device complete, and thus guarantees that all the effects of the event that triggered the interrupt will be visible to the CPU. This adds extra latency and work to the interrupt handler even if the handler doesn't need to read a PCI register. For example, if an Ethernet device had dedicated interrupt handlers for receive and transmit completion events, those handlers only need to walk the descriptor rings in RAM in the common case. Forcing them to start with a dummy read would just add overhead and latency.

Starting with PCI 2.2, a new interrupt mechanism known as Message Signaled Interrupts (MSI) was introduced to address these concerns. With MSI, each PCI function can have one or more interrupt messages. Each message has associated address and data registers whose values are assigned by the operating system. When a PCI function asserts an interrupt using MSI, it performs a PCI write operation that writes the value of the data register to the address specified in the address register. The platform must ensure that something is listening for writes to the addresses used by MSI messages and translate them into interrupt requests to one or more CPUs.

The format of the message address and data fields is platform-specific. For x86 platforms, the message data contains the IDT vector to trigger when the interrupt occurs. Thus, MSI interrupts are able to bypass the entire interrupt routing maze, and the operating system can directly link MSI interrupts to IDT vectors.

8 FreeBSD's MSI Implementation

FreeBSD implements MSI messages as `SYS_RES_IRQ` interrupts similar to the legacy INTx interrupts. The driver visible differences include different resource IDs (legacy INTx interrupt is rid 0, MSI messages start at rid 1) for `SYS_RES_IRQ` resources and new APIs for allocating and releasing MSI messages. Behind the scenes, the PCI bus driver is responsible for programming the various MSI registers. It also allocates IRQs to map MSI messages onto via requests to the parent bridge. These requests pass up through the various PCI bridge drivers (very similar to how PCI interrupt routing passes up through PCI bridges) until it finds a device that can allocate IRQs for MSI messages. Similar requests are forwarded up the device tree to release MSI IRQs no longer in use and to compute the address and data register values for an MSI IRQ.

For the x86 platforms, the PCI bridge requests bubble up through the device tree until they arrive at the `nexus0` device. This device's driver proxies the requests over to the x86 MSI code. The x86 MSI code uses interrupt source objects to manage IRQs for MSI messages. It provides a single `struct pic` shared by all MSI interrupt sources. The MSI interrupt sources are created on the fly when a request is made by a driver to allocate MSI messages. Each MSI interrupt source is assigned an IRQ value in the range 256 - 383. These IRQ values are used to avoid conflicting with the IRQs used for legacy INTx interrupts which use a range of 0 - 255. Once an MSI interrupt source is created, it is never destroyed, but it may be reused by a different device if it is released by a driver and another driver makes a subsequent allocation request. When an MSI interrupt source is allocated, it is assigned an IDT vector. If the MSI interrupt source is released, it frees the IDT vector back to the system. If a driver requests multiple MSI messages, care must be taken to ensure that the group of MSI messages use an aligned, contiguous range of IDT vectors. An extension to MSI known as MSI-X removes this limitation since it provides for separate address and data registers for each message.

9 Conclusion

Correctly implementing support for PCI interrupts requires knowledge of both PCI and the underlying platform. The x86 platform, in particular, requires a non-trivial amount of effort to properly support. Hopefully MSI will reduce the complexity of supporting PCI interrupts on future platforms as well as future variations of the x86 platform.

10 Acknowledgments

The PCI interrupt code in FreeBSD has been designed and implemented by several different people. Mike Smith and Warner Losh provided the initial support for PCI interrupt routing including support for using the $PIR routing table. Mike and Warner also conceived the idea of PCI busses asking their parent bridge driver to route interrupts. Mike also implemented the initial support for routing interrupts via the ACPI `_PRT` tables. Mitsuru Iwasaki and Nate Lawson implemented the first pci link management driver.

The work I have done on PCI interrupts was also greatly aided by others. Enhancements to the existing PCI routing code were sponsored by The Weather Channel, while the MSI code was sponsored by Yahoo!. Both companies sponsored my time and provided hardware for testing.

My biggest debt is owed to Mike Smith, however. Several years ago Mike sat down with me and explained exactly what the interrupt routing information in the MP Table meant and why the existing code in FreeBSD 4.x needed to be improved. He also encouraged me to work on ACPI and PCI interrupt routing general. On an even broader scale, Mike mentored me as a I transitioned from working on FreeBSD documentation to hacking on the source to the FreeBSD kernel including my work on SMP, ACPI, and PCI support.

11 Availability

All of the code for managing PCI interrupts is available in the FreeBSD source tree [3]. The code is split into roughly two parts. First, the code to manage interrupt routing and MSI is contained in the kernel. Second, the code for various utilities to dump interrupt routing tables is contained in userland sources. All of the interrupt routing code and related utilities can be found in FreeBSD 5.2 and later. The MSI code as well as the update to `pciconf` to display PCI capabilities was added to FreeBSD after FreeBSD 6.2 and will be present in both FreeBSD 6.3 and FreeBSD 7.0.

The PCI-PCI bridge driver is found in `src/sys/dev/pci/pci_pci.c`. Among other things, this driver contains the code to use the PCI-PCI bridge swizzle to route interrupts across PCI-PCI bridges in add-in cards.

The $PIR interrupt routing code is found in `src/sys/i386/include/pc/bios.h` and `src/sys/i386/pci/pci_pir.c`. The source for the `pirtool` utility is found in the `src/tools/tools/pirtool` directory.

The MP Table interrupt routing code is present for both amd64 and i386. For amd64 it is found in `src/sys/amd64/include/mptable.h`, `src/sys/amd64/amd64/mptable.c`, and `src/sys/amd64/amd64/mptable_pci.c`. The i386 code is found in the same files but under `src/sys/i386`. The source for the `mptable(8)` utility is found in the `src/usr.sbin/mptable` directory.

The ACPI interrupt routing code is all found in the `src/sys/dev/acpica` directory. The `acpi_pci_link.c` file contains the device driver for PCI interrupt link devices. The `acpi_pcib.c` file contains the code to parse `_PRT` tables. It will call the PCI interrupt link code when a PCI interrupt is attached to a link device. The source for the `acpidump` utility is found in the `src/usr.sbin/acpi/acpidump` utility.

The machine independent code for MSI support is found in `src/sys/dev/pci/pci.c` and `src/sys/dev/pci/pci_pci.c`. The MSI support for amd64 and i386 is found in `src/sys/amd64/amd64/msi.c` and `src/sys/i386/i386/msi.c`, respectively. The source for the `pciconf` utility is found in the `src/usr.sbin/pciconf` directory, with the code to parse PCI capabilities in `cap.c`.

References

[1] ACPI - Advanced Configuration and Power Interface, http://www.acpi.info

[2] Advanced Configuration and Power Interface Revision 2.0c, http://www.acpi.info/DOWNLOADS/ACPIspec-2-0c.pdf

[3] FreeBSD Project, http://www.FreeBSD.org

[4] Multiprocessor Specification v1.4, http://developer.intel.com/design/pentium/datashts/24201606.pdf

[5] Tom Shanley and Don Anderson, *PCI System Architecture, Fourth Edition*, Mindshare, Inc. (1999).