Advanced Programmable Interrupt Controller

by Mike Rieker

Though I had originally wrote my APIC code over a year ago, I've been playing with it recently, so I thought I'd write about it. This is not a complete treatment, but it contains stuff that the docs for the chips don't seem to tell.

There are basically two things here to consider.

1. Built into all recent x86 CPU chips (Pent Pro and up) is a thing called a Local APIC. It is addressed at physical addresses FEE00xxx. Actually, that is the default, it can be moved by programming the MSR that holds it base address.

   It has many fun things in it. The big thing is that you can interrupt other CPU's in a multiprocessor system. But if you just have a uniprocessor, there are useful things for it, too.

   The Local APIC is described in Chapter 7 of Volume 3 of the Intel processor books.

2. Some motherboards have an IO APIC on them. This is usually only found on multiprocessor boards. Functionally, it replaces the 8259's. You must essentially shut off the 8259's and turn on the IO APIC to use it.

   The IO APIC is typically located at physical address FEC00000, but may be moved by programming the north/southbridge chipset.

   The Intel chip number is 82093 and you can get the doc for it off of the Intel website.

What the Local APIC Is

As stated above, the Local APIC (LAPIC) is a circuit that is part of the CPU chip. It contains these basic elements:

1. A mechanism for generating interrupts
2. A mechanism for accepting interrupts
3. A timer

If you have a multiprocessor system, the APIC's are wired together so they can communicate. So the LAPIC on CPU 0 can communicate with the LAPIC on CPU 1, etc.

What the IO APIC Is

This is a separate chip that is wired to the Local APIC's so it can forward interrupts on to the CPU chips. It is programmed similar to the 8259's but has more flexibility.

It is wired to the same bus as the Local APIC's so it can communicate with them.

Fun things to do with a Local APIC in a Uniprocessor

(this stuff also applies to multiprocessors, too)

One thing the LAPIC can help with is the following problem:

An IRQ-type interrupt routine wishes to wake a sleeping thread, but this IRQ interrupt may be nested several levels inside other IRQ interrupts, so it cannot simply switch stacks as those outer interrupt routines would not complete until the old thread is re-woken.

So we have to somehow switch out of the current thread and switch into the thread to be woken. A way the LAPIC can help us is to tell it to interrupt this same CPU, but only when there are no IRQ-type interrupt handlers active.

I call this a 'software' interrupt because the operating system software initiated the interrupt. It is programmed into the LAPIC to be at a priority lower than any IRQ-type interrupt.

So now if some IRQ-type routine wants to wake a thread, it makes the necessary changes to the datastructures, then triggers a software interrupt to itself. Then, when all IRQ-type interrupt handlers have returned out, the LAPIC is now able to interrupt.It interrupts out of the currently executing thread and switches to the thread that was just woken. Very neat.

Without the LAPIC, your interrupt routine has to set a flag in memory somewhere that each IRET has to check for. So each IRET checks this flag and checks to see if it is the 'last' IRET. It is more efficient to let the LAPIC do this testing for you.

So now we have to make this software LAPIC interrupt have a lower priority than IRQ interrupts. We do this by studying how the LAPIC assigns priority to interrupts. This is a bit lame but it works ok. The priority is based on the vector number we choose for the interrupt. Interrupt vectors are numbered 0x00 through 0xFF in Intel CPUs. The LAPIC assigns a priority based on the first of the two hex digits and ignores the second digit. Thus, any interrupts using vectors 0x50 through 0x5F have the same priority. So if you block something at priority 0x52, you block all interrupts in the range 0x50 through 0x5F.

Now the CPU itself uses vectors in the range 0x00..0x1F for exceptions, so we don't want to use those for LAPIC interrupts. This means we can use a vector numbered 0x20 or 0x2F or somewhere in that range. We will have to redirect the IRQ interrupts to vectors 0x30..0x3F or something even higher if necessary, by re-programming the 8295's. Now we can block software interrupts without blocking IRQ interrupts.

The LAPIC's priority can be set by writing the LAPIC's TSKPRI (task priority) register. So if you want to block all interrupts through level 0x2F, just write a 0x20 (or 0x2B, etc) into the TSKPRI and you have blocked those interrupts.

Now the LAPIC is not really connected to the 8259's. You cannot block 8259 generated interrupts with the LAPIC. Likewise, being in an IRQ-type interrupt handler does not block any LAPIC interrupts. So we have to manually block/unblock the softints at the beginning of our IRQ handler. Just push the LAPIC's TSKPRI register, set it to 0x20 and handle your IRQ interrupt as usual. When done, pop the saved LAPIC's TSKPRI then IRET.

So the IRQ interrupt handler looks something like:

```
entry:                          # cpu hw ints inhibited on entry
  push LAPIC's TSKPRI           # save previous TSKPRI, either 00 or 20
  movl $0x20, LAPIC's TSKPRI    # make sure softint delivery inhibited now
  sti                           # let cpu process higher priority interrupts
  push general registers        # save registers used by interrupt handlers
```

```
  process interrupt           # process the IRQ interrupt level
  pop general registers       # restore clobbered registers
  cli                         # prevent nesting until we have unwound stack
  send EOI to the 8259's      # let 8259's deliver this IRQ level again after iret
  pop LAPIC's TSKPRI          # restore TSKPRI, possibly allowing softints after iret
  iret
```

The software interrupt handler looks something like:

```
entry:                        # cpu hardware interrupt delivery enabled
  movl $0x20,LAPIC's TSKPRI   # prevent nesting of software interrupts
  send EOI to LAPIC           # allow local APIC to queue another interrupt
  save general registers
  switch stack to highest priority thread
  restore general registers
  cli                         # prevent nesting until we have unwound stack
  movl $0,LAPIC's TSKPRI      # allow local APIC to deliver softints after the iret
  iret
```

Something else the Local APIC is good for in a Uniprocessor:

Another use for the LAPIC is that it has a built in timer. So you can set the timer for any interval and it will generate an interrupt.It basically has a 32-bit counter that runs at the bus speed, like 100MHz. So what I use it for is the quantum reschedule interrupt. If a single thread uses the CPU for a whole tenth of a second, this timer interrupts it and lowers the thread's priority. Then if there is an equal priority thread waiting, it switches to it.

This is particularly attractive in a multiprocessor system, as there is one such timer per CPU chip, so you have one quantum timer per executing thread to work with.

The quantum timer interrupt handler is very similar to the regular softint handler, except it decrements the current thread's priority. So I put the quantum timer on a vector in the 0x20..0x2F range right next to the regular softint vector. Thus writing 0x20 to the TSKPRI register blocks both softints and quantum ints.

Now I couldn't resist the temptation to muck it up by folding in my normal timer queue stuff as I can get basically 10nS resolution on my timer requests. So I have it designed such that whichever CPU queued the next-to-expire timer request, gets its LAPIC taken over to service the timer queue, until another CPU comes along with an even earlier timer request, or the thread's quantum is going to run out earlier than the timer request. So my current timer handler is actually a bit more complicated than the one shown above.

Multiprocessor uses for the Local APIC

Having the Local APIC is essential in a multiprocessor. Without it, the motherboard would have to provide some equivalent functionality.

One function fits in nicely with the 'softint' stuff described for the uniprocessor. We just generalize it a bit. In a general multi-processor OS, an IRQ interrupt handler just needs to 'wack' *some* CPU to handle the newly woken thread, not necessarily its own. With the LAPIC, all it has to do is tell its LAPIC to send a message to the current lowest priority LAPIC that it is connected to, possibly including itself.

Something unique to multiprocessor systems is the following situation:

1. Two or more processors are mapped to the same process at the same time, presumably running different threads in that process

2. One of the CPU's (threads) unmaps a memory region and marks the corresponding pagetable entries as being 'not-present'
3. The CPU executes INVLPG instructions to wipe out any left over cached copies of the pagetable entries it has
4. Now that CPU has to tell the other CPU to invalidate any cached pagetable entries it has. It can do this with the LAPIC's. It sends an interrupt out to tell the other CPU's to invalidate their pagetable entries at that virtual address.

I use a very high priority interrupt for this purpose, even higher than any IRQ interrupts. This is because I have the originating CPU wait for the target CPU's to acknowledge that they have processed the invalidate before continuing on. So I want the target CPU's to process the interrupt immediately.

Another fun thing to do with LAPIC's in a multiprocessor:

You have implemented a kernel debugger. One CPU hits a breakpoint and traps to the kernel debugger. The other CPU's are still merrily churning away doing who-knows-what. So you can interrupt them (my OS sends them an NMI) so they will also call the debugger. This way you will have stable datastructures to examine and you will also be able to see just what the other CPU's were doing at the time of the breakpoint.

What the IOAPIC is good for

The IOAPIC is generally only used in multiprocessor systems, and is not typically found on uniprocessor motherboards.

The main advantage it has over 8259's is that it can distribute the IRQ type interrupts to various CPU's. The 8259's will only deliver the interrupt to the boot CPU, whilst ignoring the others.

Another small advantage it has is that it can route the PCI interrupts to separate vectors and not overlap the IRQ's. My OS does not take advantage of this, but you may want to take on this challenge.

The IOAPIC has a table of 24 64-bit registers. This was extreme overkill, IMO, as less than 32 bits are used from each. But anyway, that's the way it is. Also, the registers are accessed indirectly, there is an 'address' register and a 'data' register. Fortunately, like the 8259 and unlike the LAPIC, the IOAPIC can be programmed at boot time then left alone for the rest of system operation.

Each of the 24 registers has an interrupt source that is hardwired from the motherboard design. Standard designs follow this convention:

- IRQ's 0..15 : IOAPIC registers 0..15
- PCI A..D : IOAPIC registers 16..19

I do not know what they use 20..23 for. It is possible that 23 is used for SMI, but I haven't ever needed to use it, so I haven't research it. This is left as an excercise to the reader.

The trick to programming these things is setting the polarity and trigger mode. For the IRQ 0..15 interrupts, set it to 'edge triggered, active high'mode (both <13> and <15> zero). For the PCI A..D interrupts, set it to 'level triggered, active low' (both <13> and <15> one). At least that's what works on my motherboard.

Now here's where you can get creative. Each of the interrupts can be programmed with any of the 224 vectors the CPU is capable of (we don't use 0x00..0x1F, just 0x20..0xFF). Like the LAPIC's, the IOAPIC's will assign a priority to the interrupt based on the vector you give it. And also like the LAPIC's, the dumb things

only use the 4 top bits of the vector number to distinguish priority, and ignore the low 4 bits. So we have to compromise and group 2 interrupts per level. So you want to assign the lowest priority interrupts to the lowest vector numbers.

Conventionally, from highest to lowest priority, the IRQ's go 0,1,2,8,9,10,11,12,13,14,15,3,4,5,6,7. You can assign the vectors to the IRQ's in that order or any order you wish. Whichever IRQ you give the highest vector number to will be the highest priority, etc. I just stuck with the conventional assignment as I can't think of any reason not to.

So we have 20 vectors to assign, 16 for the IRQ's and 4 for the PCI's. As I said, in my OS I map the PCI interrupts onto the IRQ's like the BIOS set up, you may want to do different. So all I really need is 16 vectors.I want to leave some at the bottom (for softint stuff) and some at the top (for high-priority LAPIC interrupts).

So I can get away with using vector numbers 0x74, 0x7C, 0x84, 0x8C, ... 0xE4, 0xEC. I assign these to the usual IRQ priorities 7 (lowest), 6, 5, 4, ..., 1, 0. This means I program IOAPIC interrupt register 0 with the vector for IRQ 0 which I want to be 0xEC, as IRQ 0 is the highest priority. I program IOAPIC interrupt register 1 with 0xE4 as IRQ 1 is the next highest priority. And so on, until I have programmed the first 16 of the IOAPIC's interrupt registers.

I seem to get useless interrupts on the IRQ 2 line, so I set this table entry to be disabled.

Now for the truly exciting part, programming the next four interrupt registers that correspond to PCI interrupts A..D. I chickened out here. It would be nice to program these completely independent of the IRQ interrupts. Problem is, almost all controller cards report that they use INT-A to send their interrupt. So the motherboards scramble the actual vectors around, so INT-A on slot 10 might actually come in as INT-B to the IOAPIC, and INT-A on slot 9 might come in as INT-C to the IOAPIC. Only the BIOS (or mobo tech doc) knows the answer to this puzzle.

It would have been nice if the PCI BIOS would have provided a call to retrieve this nightmare but it doesn't. So what I do is go out to the northbridge/southbridge chipset and retrieve the PCI-to-IRQ mapping that the BIOS has established. In the PIIX4 chipset, this can be retrieved from the longword at offset 0x60 of device id 0x7110 (vendor id 0x8086). Byte 0 of this long tells me what IRQ it is going to forward PCI-A interrupts to, so I program the IOAPIC to do the same thing by setting the IOAPIC interrupt register 16 to interrupt to the same vector used by the IRQ. So if byte 0 has a 5, it means it has set up controllers connected to PCI-A interrupts to think they are IRQ 5's, so I put vector 0x84 in the IOAPIC's interrupt register 16. Likewise, byte 1 of the long tells me what IRQ it has set INT-B cards to, so I write the vector number I set aside for that IRQ for the IOAPIC's interrupt register 17. Same for bytes 2 and 3. If the BIOS didn't find any PCI cards using a given PCI interrupt line, it sets the byte to 0x80 indicating that it is disabled.

Finally, I simply disable entries 20 through 23.

So what I end up with is:

IOAPIC int reg	Vector
0	0xEC
1	0xE4
2	disabled
3	0x94
4	0x8C
5	0x84
6	0x7C
7	0x74
8	0xD4
9	0xCC

```
      10     0xC4
      11     0xBC
      12     0xB4
      13     0xAC
      14     0xA4
      15     0x9C
      16     depends on what BIOS set up for PCI-A.
             For example if  5, use same as for IRQ  5, 0x84
      17     depends on what BIOS set up for PCI-B.
             For example if  9, use same as for IRQ  9, 0xCC
      18     depends on what BIOS set up for PCI-C.
             For example if 10, use same as for IRQ 10, 0xC4
      19     depends on what BIOS set up for PCI-D.
             For example if 12, use same as for IRQ 11, 0xBC
20..23     disabled
```

Now if that doesn't make your head hurt, not only can you tell the IOAPIC what vector to use to process the interrupt, you can tell it which CPU to send the interrupt to. Here, I take the easy way out. I tell it to send it to the CPU that is currently at the lowest priority. By priority, we mean the higher of what's in its LAPIC TSKPRI register or whatever interrupt it is currently servicing.

One more little thing. Since these interrupt registers are 64 bits, it takes two writes to set them. So what I do is disable the entry by writing a disable value (0x10000) to the low half, then I write the desination info to the top half, then finally set the low half to what I want.

Oh did I say one more thing? There's more. Some motherboards have a thingy that you must write to redirect the interrupts from the 8259's to the IOAPIC. This is done by writing 0x70 to port 0x22 and 0x01 to port 0x23. Also, you may want to disable the 8259's completely. I do this by writing 0xFF to ports 0x21 and 0xA1. So I disable the 8259's first, then program the IOAPIC, then write the ICMR (0x22,0x23) ports to accept IOAPIC interrupts.

From this point on, the IOAPIC is programmed and you shouldn't have to access it again.

When the IOAPIC receives an interrupt from a controller card, it forwards the interrupt out to the LAPIC's (as they are all wired together). When an LAPIC is able to accept the interrupt (ie, its TSKPRI is lower than the vector and it is not currently servicing the same or higher level, and the CPU itself has interrupt delivery enabled), the LAPIC will signal an interrupt to the CPU and the CPU will interrupt through the corresponding interrupt vector that was programmed into the IOAPIC.

Using the programming table above, to inhibit delivery of IRQ 7 and below in a particular CPU, just write 0x8C to the LAPIC's TSKPRI register. This blocks delivery of all APIC generated interrupts in that CPU with vectors numbered up through 0x8F. Other CPU's are free to accept those interrupts, though (assuming their corresponding TSKPRI register is not blocking them). So if you want to block interrupts on ALL CPU's for a given level, you will have to set a spinlock after setting the TSKPRI register. And your interrupt routine will also have to lock that same spinlock. So if another CPU gets that IRQ 7 interrupt while you have the spinlock, it will wait for you to release the spinlock before continuing. While not truly blocking the interrupt on other CPU's, it has the same basic effect.

Misc

Physical/logical id/destination

When an multiprocessor system starts, the hardware loads a physical id number into each CPU's LAPIC circuit. This is how you can tell one CPU from another in the system. You can also reprogram this number if

you wish, but generally there's no need to. It's sufficient that each LAPIC have a unique physical id.

So if you set an IO or Local APIC interrupt register to target a particular physical id, then it will go to that CPU. This may be fine for some instances, but there are cases where we want to interrupt a given set of CPU's. Well we could send an interrupt to each one individually. But that is inefficient and there is a better way. Enter the 'logical destination'. This is a bitmask thing. If a sender has a bit set in its target mask that a destination has in its logical address mask, that target is eligible to receive the interrupt. So what I do is set the following logical addresses up:

- physical id 0 gets logical id 0x01
- physical id 1 gets logical id 0x02
- physical id 2 gets logical id 0x04
- physical id 3 gets logical id 0x08
- ...
- physical id 7 gets logical id 0x80

This scheme limits me to 8 CPU's (as the logical destination mask is only 8 bits), but if you have more CPU's then you can double them up (eg, 0&8 get mask 0x01, 2&9 get mask 0x02, etc). The APIC's also support a clustering mode, but I had no need for it so I didn't study it.

Assuming we have less than 9 CPU's, we can easily select the targets by setting the appropriate bits in our interrupt mask. Like for the IOAPIC destinations, I set all the bits so any CPU's I have are elegible for the interrupt, and I tell the IOAPIC to select the lowest priority amongst all those CPU's. When I have to invalidate a shared pagetable entry, I can select only those CPU's that are also currently using that same pagetable and leave the others alone.

Local APIC initialization

The Local APIC contains stuff that has to be initialzed. It may be that the BIOS sets it for the boot processor so it will behave normally, but you will need to initialize it for any other processors, so you might as well do it for all so they come out the same.

Here's what I put in it:

- task priority (FEE00080) = 0x20 to inhibit softint delivery
- timer interrupt vector (FEE00320) = 0x10000 to disable timer interrupts
- performance counter interrupt (FEE00340) = 0x10000 to disable performance counter interrupts
- local interrupt 0 (FEE00350) = 0x08700 to enable normal external interrupts
- local interrupt 1 (FEE00360) = 0x00400 to enable normal NMI processing
- error interrupt (FEE00370) = 0x10000 to disable error interrupts
- spurious interrupt (FEE000F0) = 0x0010F to enable the APIC and set spurious vector to 15

Then after it is enabled, I set the local interrupt 0 and 1 again to the same thing, just to be sure the reset didn't keep them shut off.

That's another thing the book doesn't quite say. Local interrupt 0 is for interrupts on the CPU's normal interrupt pin. Local interrupt 1 is for interrupts on the CPU's NMI pin. Maybe it's in there somewhere, but I thought I state it here in case it's not or you can't find it.

Finally, I program the logical destination address from the physical id, puquing (fr) if the physical id is .ge. 8.

A little gotcha

To tell the LAPIC to generate an interrupt, you have to write two registers. In my OS (and probably all that use the thing), interrupt routines are allowed to write the LAPIC registers. So you have to disable the CPU's interrupts while writing the two LAPIC registers or you will have a mess:

```
pushfl
cli
movl target_mask, LAPIC's ICREG 1
                   << bad news if an interrupt broke
                   << in here and changed ICREG 1
movl interrupt_type, LAPIC's ICREG 0
popfl
```

Summary

The APIC's provide a mechanism that is required in a multiprocessor system necessary for interprocessor communication. They also contain convienient features for uniprocessor systems as well.

If you want to see my APIC routines, they are in http://www.o3one.org/sources/oz_hw_smproc_486.s
It is in GNU assembler format.

Mike's home page can be found at http://www.o3one.org/