APIC

From OSDev Wiki

APIC ("Advanced Programmable Interrupt Controller") is the updated Intel standard for the older PIC. It is used in multiprocessor systems and is an integral part of all recent Intel (and compatible) processors. The APIC is used for sophisticated interrupt redirection, and for sending interrupts between processors. These things weren't possible using the older PIC specification.

Contents

Detection

The CPUID.01h:EDX[bit 9] flag specifies whether a CPU has a built-in local APIC.

Local APIC and IO-APIC

In an APIC-based system, each CPU is made of a "core" and a "local APIC". The local APIC is responsible for handling cpu-specific interrupt configuration. Among other things, it contains the *Local Vector Table (LVT)* that translates events such as "internal clock" and other "local" interrupt sources into a interrupt vector (e.g. LocalINT1 pin could be raising an NMI exception by storing "2" in the corresponding entry of the LVT).

More information about the local APIC can be found in the "System Programming Guide" of current Intel processors.

In addition, there is an I/O APIC (e.g. intel 82093AA) that is part of the chipset and

provides multi-processor interrupt management, incorporating both static and dynamic symmetric interrupt distribution across all processors. In systems with multiple I/O subsystems, each subsystem can have its own set of interrupts.

Each interrupt pin is individually programmable as either edge or level triggered. The interrupt vector and interrupt steering information can be specified per interrupt. An indirect register accessing scheme optimizes the memory space needed to access the I/O APIC's internal registers. To increase system flexibility when assigning memory space usage, the I/O APIC's two-register memory space is relocatable, but defaults to 0xFEC00000.

The Intel standards for the APIC can be found on the Intel site under the category "Multiprocessor Specification", or simply this PDF file (http://developer.intel.com/design/pentium/datashts/24201606.pdf) .

Inter-Processor Interrupts

Inter-Processor Interrupts (IPIs) are generated by a local APIC and can be used as basic signaling for scheduling coordination, multi-processor bootstrapping, etc.

Local APIC configuration

The local APIC is enabled at boot-time and can be disabled by clearing bit 11 of the IA32_APIC_BASE Model Specific Register (MSR) (see example below, this only works on CPUs with family >5, as the Pentium does not have such MSR). The CPU then receives its interrupts directly from a 8259-compatible PIC. The Intel Software Developer's Manual, however states that, once you have disabled the local APIC through IA32_APIC_BASE you can't enable it anymore until a complete reset. The I/O APIC can also be configured to run in legacy mode so that it emulates an 8259 device.

The local APIC's registers are memory-mapped in physical page FEE00xxx (as seen in table 8-1 of Intel P4 SPG). Note that there is a MSR that specifies the actual APIC base (only available on CPUs with family >5). You can choose to leave the Local APIC base just where you find it, or to move it at your pleasure. **Note:** I don't think you can move it any further than the 4th Gb.

To enable the Local APIC to receive interrupts it is necessary to configure the "Spurious Interrupt Vector Register". The correct value for this field is the IRQ number that you want to map the spurious interrupts to within the lowest 8 bits, and the 8th bit set to 1 to actually enable the APIC (see the specification for more details). You should choose an interrupt number that has its lowest 4 bits set and is above 32 (as you might guess); easiest is to use 0xFF. This is important on some older processors because the lowest 4 bits for this value must be set to 1 on these.

Disable the 8259 PIC properly. This is nearly as important as setting up the APIC. You do this in two steps: masking all interrupts and remapping the IRQs. Masking all interrupts disables them in the PIC. Remapping is what you probably already did when you used the PIC: you want interrupt requests to start at 32 instead of 0 to avoid conflicts with the exceptions. This is necessary because even though you masked all interrupts on the PIC, it could still give out spurious interrupts which will then be misinterpreted from your kernel as exceptions.

Here are some code examples on setting up the APIC:

```
#define IA32_APIC_BASE_MSR 0x1B
#define IA32_APIC_BASE_MSR_BSP 0x100 // Processor is a BSP
#define IA32_APIC_BASE_MSR_ENABLE 0x800

/** returns a 'true' value if the CPU supports APIC
 *  and if the local APIC hasn't been disabled in MSRs
 *  note that this requires CPUID to be supported.
 */
bool cpuHasAPIC()
{
   uint32_t eax, edx;
   cpuid(1, &eax, &edx);
   return edx & CPUID_FLAG_APIC;
}

/* Set the physical address for local APIC registers */
void cpuSetAPICBase(uintptr_t apic)
{
   uint32_t edx = 0;
   uint32_t eax = (apic & 0xfffff100) | IA32_APIC_BASE_MSR_ENABLE;

#ifdef __PHYSICAL_MEMORY_EXTENSION__
   edx = (apic >> 32) & 0x0f;
#endif

   cpuSetMSR(IA32_APIC_BASE_MSR, eax, edx);
}

/**
 * Get the physical address of the APIC registers page
 * make sure you map it to virtual memory ;)
 */
uintptr_t cpuGetAPICBase()
{
```

```
    uint32_t eax, edx;
    cpuGetMSR(IA32_APIC_BASE_MSR, &eax, &edx);

#ifdef __PHYSICAL_MEMORY_EXTENSION__
    return (eax & 0xfffff100) | ((edx & 0x0f) << 32);
#else
    return (eax & 0xfffff100);
#endif
}

void enableAPIC()
{
    /* Hardware enable the Local APIC if it wasn't enabled */
    cpuSetAPICBase(cpuGetAPICBase());

    /* Set the Spourious Interrupt Vector Register bit 8 to start r
    WriteRegister(0xF0, ReadRegister(0xF0) | 0x100);
}
```

IO APIC Configuration

The IO APIC uses two registers for most of its operation - an address register at IOAPICBASE+0 and a data register at IOAPICBASE+0x10. All accesses must be done on dword boundaries. The address register uses the bottom 8 bits for register select. Here is some example code that illustrates this:

```
uint32_t cpuReadIoApic(void *ioapicaddr, uint32_t reg)
{
    uint32_t volatile *ioapic = (uint32_t volatile *)ioapicaddr;
    ioapic[0] = (reg & 0xff);
    return ioapic[4];
}

void cpuWriteIoApic(void *ioapicaddr, uint32_t reg, uint32_t value)
{
    uint32_t volatile *ioapic = (uint32_t volatile *)ioapicaddr;
    ioapic[0] = (reg & 0xff);
    ioapic[4] = value;
}
```

Note the use of the volatile keyword. This prevents a compiler like Visual C from reordering or optimizing away the memory accesses, which would be a Bad Thing™. The volatile keyword is put before the '*' sign. It means that the *value*

pointed to is volatile, not the pointer itself.

See Also

Articles

- 8259 PIC
- IOAPIC
- APIC timer

Threads

- APIC timer (http://www.osdev.org/phpBB2/viewtopic.php?t=10686)
- Mapping the I/O APIC (http://www.osdev.org/phpBB2/viewtopic.php?t=11529)
- Brendan gives some general info on the APIC and implementing it. (http://www.osdev.org/phpBB2/viewtopic.php?p=107868#107868)

External Links

- original I/O APIC specification/datasheet (http://www.intel.com/design/chipsets/datashts/290566.htm)
- updated I/O APIC specification/datasheet (http://developer.intel.com/design/chipsets/specupdt/290710.htm)
- Volume 3A:System Programming Guide, Part 1,manuals has a chapter on the APIC (http://www.intel.com/products/processor/manuals/)
- Advanced Programmable Interrupt Controller by Mike Rieker (http://www.osdever.net/tutorials/pdf/apic.pdf)
- "The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs". Microsoft. 7 January 2010 (http://msdn.microsoft.com/en-us/windows/hardware/gg462964.aspx)

Categories: Interrupts | Time | Multiprocessing
