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. You can find all of the APICs on a system (both local and IO APICs) by parsing the MADT.

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 Chapter 10 of the Intel System Programming Guide, Vol 3A Part 1 (https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf) .

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://web.archive.org/web/20070112195752/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). This address is the same for each local APIC that exists in a configuration, meaning you are only able to directly access the registers of the local APIC of the core that your code is currently executing on. Note that there is a MSR that specifies the actual APIC base (only available on CPUs with family >5). The MADT contains the local APIC base and on 64-bit systems it may also contain a field specifying a 64-bit base address override which you ought to use instead. 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. You should then avoid using these interrupt vectors for other purposes. 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 check_apic() {
   uint32_t eax, edx;
   cpuid(1, &eax, &edx);
   return edx & CPUID_FEAT_EDX_APIC;
}

/* Set the physical address for local APIC registers */
void cpu_set_apic_base(uintptr_t apic) {
   uint32_t edx = 0;
   uint32_t eax = (apic & 0xfffff000) | 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 cpu_get_apic_base() {
   uint32_t eax, edx;
   cpuGetMSR(IA32_APIC_BASE_MSR, &eax, &edx);

#ifdef __PHYSICAL_MEMORY_EXTENSION__
   return (eax & 0xfffff000) | ((edx & 0x0f) << 32);
#else
   return (eax & 0xfffff000);
#endif
}

void enable_apic() {
    /* Hardware enable the Local APIC if it wasn't enabled */
    cpu_set_apic_base(cpu_get_apic_base());

    /* Set the Spurious Interrupt Vector Register bit 8 to start receiving interrupts */
    write_reg(0xF0, ReadRegister(0xF0) | 0x100);
}
```

Local APIC and x86 SMM Attacks

The APIC was introduced to the core Intel processor architecture skeleton in Intel's 82489DX discrete chip (https://4donline.ihs.com/images/VipMasterIC/IC/INTL/INTLD047/INTLD047-2-1259.pdf?hkey=EF798316E3902B6ED9A73243A3159BB0) in a similar time period as System Management Mode was introduced to operating systems. In original architecture, the APIC could not be mapped to memory, and it wasn't until later changes that it became mappable.

As System Management Mode's memory (SMRAM) is given a protected range of memory (which can vary from system to system), it is possible to map the APIC memory location into the SMRAM. The result of this is that SMM memory is pushed outside its protected range and exposed to lesser-privileged permission rings. Using this method, attackers can leverage their permissions using System Management Mode, which is protected from all rings above -2.

In newer generation Intel processors (starting with the Intel Atom (https://en.wikipedia.org/wiki/Intel_Atom) in 2013), this has been taken into account. An undocumented check is performed against the System Management Range Registers when the APIC is relocated to memory. This check ensures that the APIC does not overlap with the SMRAM. **However**, this relies on the SMRR to be configured correctly. Otherwise, this mitigation will not work properly and attackers will still be able to user the attack.

Local APIC registers

The local APIC registers are memory mapped to an address that can be found in the MP/MADT tables. Make sure you map these to virtual memory if you are using paging. Each register is 32 bits long, and expects to be written and read as a 32 bit integer. Although each register is 4 bytes, they are all aligned on a 16 byte boundary.

List of local APIC registers (TODO: Add descriptions for all registers):

Offset	Register name	Read/Write permissions
000h - 010h	Reserved	
020h	LAPIC ID Register	Read/Write

030h	LAPIC Version Register	Read only
040h - 070h	Reserved	
080h	Task Priority Register (TPR)	Read/Write
090h	Arbitration Priority Register (APR)	Read only
0A0h	Processor Priority Register (PPR)	Read only
0B0h	EOI register	Write only
0C0h	Remote Read Register (RRD)	Read only
0D0h	Logical Destination Register	Read/Write
0E0h	Destination Format Register	Read/Write
0F0h	Spurious Interrupt Vector Register	Read/Write
100h - 170h	In-Service Register (ISR)	Read only
180h - 1F0h	Trigger Mode Register (TMR)	Read only
200h - 270h	Interrupt Request Register (IRR)	Read only
280h	Error Status Register	Read only
290h - 2E0h	Reserved	
2F0h	LVT Corrected Machine Check Interrupt (CMCI) Register	Read/Write
300h - 310h	Interrupt Command Register (ICR)	Read/Write
320h	LVT Timer Register	Read/Write
330h	LVT Thermal Sensor Register	Read/Write
340h	LVT Performance Monitoring Counters Register	Read/Write
350h	LVT LINT0 Register	Read/Write
360h	LVT LINT1 Register	Read/Write
370h	LVT Error Register	Read/Write
380h	Initial Count Register (for Timer)	Read/Write
390h	Current Count Register (for Timer)	Read only
3A0h - 3D0h	Reserved	
3E0h	Divide Configuration Register (for Timer)	Read/Write
3F0h	Reserved	

EOI Register

Write to the register with offset 0xB0 using the value 0 to signal an end of interrupt. A non-zero value causes a general protection fault.

Local Vector Table Registers

There are some special interrupts that the processor and LAPIC can generate themselves. While external interrupts are configured in the I/O APIC, these interrupts must be configured using registers in the LAPIC. The most interesting registers are: 0x320 = lapic timer, 0x350 = lint0, 0x360 = lint1. See the Intel SDM vol 3 for more info.

Register format:

Bits 0-7	The vector number
Bits 8-11 (reserved for timer)	100b if NMI
Bit 12	Set if interrupt pending.
Bit 13 (reserved for timer)	Polarity, set is low triggered

Bit 14 (reserved for timer)	Remote IRR
Bit 15 (reserved for timer)	trigger mode, set is level triggered
Bit 16	Set to mask
Bits 17-31	Reserved

Spurious Interrupt Vector Register

The offset is 0xF0. The low byte contains the number of the spurious interrupt. As noted above, you should probably set this to 0xFF. To enable the APIC, set bit 8 (or 0x100) of this register. If bit 12 is set then EOI messages will not be broadcast. All the other bits are currently reserved.

Interrupt Command Register

The interrupt command register is made of two 32-bit registers; one at 0x300 and the other at 0x310. It is used for sending interrupts to different processors. The interrupt is issued when 0x300 is written to, but not when 0x310 is written to. Thus, to send an interrupt command one should first write to 0x310, then to 0x300. At 0x310 there is one field at bits 24-27, which is local APIC ID of the target processor (for a physical destination mode). Here is how 0x300 is structured:

Bits 0-7	The vector number, or starting page number for SIPIs
Bits 8-10	The destination mode. 0 is normal, 1 is lowest priority, 2 is SMI, 4 is NMI, 5 can be INIT or INIT level de-assert, 6 is a SIPI.
Bit 11	The destination mode. Clear for a physical destination, or set for a logical destination. If the bit is clear, then the destination field in 0x310 is treated normally.
Bit 12	Delivery status. Cleared when the interrupt has been accepted by the target. You should usually wait until this bit clears after sending an interrupt.
Bit 13	Reserved
Bit 14	Clear for INIT level de-assert, otherwise set.
Bit 15	Set for INIT level de-assert, otherwise clear.
Bits 18-19	Destination type. If this is > 0 then the destination field in 0x310 is ignored. 1 will always send the interrupt to itself, 2 will send it to all processors, and 3 will send it to all processors aside from the current one. It is best to avoid using modes 1, 2 and 3, and stick with 0.
Bits 20-31	Reserved

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 4 byte 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.

IO APIC Registers

Using the methods described above, the following registers are accessible.

0x00	Get/set the IO APIC's id in bits 24-27. All other bits are reserved.
0x01	Get the version in bits 0-7. Get the maximum amount of redirection entries in bits 16-23. All other bits are reserved. Read only.
0x02	Get the arbitration priority in bits 24-27. All other bits are reserved. Read only.
0x10 to 0x3F	Contains a list of redirection entries. They can be read from and written to. Each entries uses two addresses, e.g. 0x12 and 0x13.

Here is what a redirection entry looks like.

Bits 0-7	Interrupt vector. Allowed values are from 0x10 to 0xFE.
Bits 8-10	Type of delivery mode. 0 = Normal, 1 = Low priority, 2 = System management interrupt, 4 = Non maskable interrupt, 5 = INIT, 7 = External. All others are reserved.
Bit 11	Destination mode. Affects how the destination field is read, 0 is physical mode, 1 is logical. If the Destination Mode of this entry is Physical Mode, bits 56-59 contain an APIC ID.
Bit 12	Set if this interrupt is going to be sent, but the APIC is busy. Read only.
Bit 13	Polarity of the interrupt. 0 = High is active, 1 = Low is active.
Bit 14	Used for level triggered interrupts only to show if a local APIC has received the interrupt (= 1), or has sent an EOI (= 0). Read only.
Bit 15	Trigger mode. 0 = Edge sensitive, 1 = Level sensitive.
Bit 16	Interrupt mask. Stops the interrupt from reaching the processor if set.
Bits 17-55	Reserved.
Bits 56-63	Destination field. If the destination mode bit was clear, then the lower 4 bits contain the bit APIC ID to sent the interrupt to. If the bit was set, the upper 4 bits also contain a set of processors. (See below)

For more information, check out chapter 3 of the I/O APIC datasheet (http://web.archive.org/web/20161130153145/http://download.intel.com/design/chipsets/datashts/29056601.pdf) .

The redirection table allows you to choose which external interrupts are sent to which processors and with which interrupt vectors. When choosing the processors you should consider: spreading out the workload between the processors, avoiding processors in a low-power state, and avoiding throttled processors. When choosing the

interrupt vectors you should remember that interrupts 0x00 to 0x1F are reserved for internal processor exceptions, the interrupts you remapped the PIC to may receive spurious interrupts, that 0xFF is probably where you put the APIC spurious interrupt, and that the upper 4 bits of an interrupt vector indicate its priority.

Logical Destination Mode

Logical destination mode uses an 8-bit logical APIC ID, contained in the LDR (logical destination register, unique to each APIC). All APICs compare their local ID to the destination code sent with the interrupt. This allows to target a group of processors by programming them with the same logical APIC ID.

The LDR is formatted as follows

Bits 0-23	Reserved.		
Bits 24-31	Flat model	Bitmap of target processors (bit identifies single processor; supports a maximum of 8 local APIC units)	
	Cluster model	Bits 24-27	Local APIC address (identifies the specific processor in a group)
		Bits 28-31	Cluster address (identifies a group of processors)

The DFR (destination format register) specifies Flat or Cluster model and is structured as follows

Bits 0-27	Reserved.
Bits 28-31	Model (1111b = Flat model, 0000b = Cluster model)

'Don't use cluster mode addressing, especially "hierarchical cluster mode". AFAIK it was intended for large NUMA systems, where there's a "node controller" for each NUMA domain that forwarded interrupts to CPUs within that NUMA domain (with a seperate APIC bus for each NUMA domain). Unless your chipset has these "node controllers" (or "cluster managers" as Intel calls them) it won't work, and no modern computers have them (AFAIK there are only a few obscure Pentium III/P6 NUMA systems that ever did). You want to use "flat model" for normal SMP and for most NUMA systems (including AMD's).' (Brendan (http://forum.osdev.org/viewtopic.php?f=1&t=14808&start=17))

More info can be found in "Pentium Processor System Architecture. Chapter 15: The APIC" (https://books.google.nl/books?id=TVzjEZg1--YC&printsec=frontcover)

SIPI Sequence

The INIT-SIPI sequence is done by the operating system after the firmware already gives the first INIT signal. This example code from the [Silcos Kernel http://wiki.osdev.org/Silcos_Kernel]

```
void APIC::wakeupSequence(U32 apicId, U8 pvect)
{
                ICRHigh hreg = {
                                .destField = apicId
                };

                ICRLow lreg(DeliveryMode::INIT, Level::Deassert, TriggerMode::Edge);

                xAPICDriver::write(APIC_REGISTER_ICR_HIGH, hreg.value);
                xAPICDriver::write(APIC_REGISTER_ICR_LOW, lreg.value);

                lreg.vectorNo = pvect;
                lreg.delvMode = DeliveryMode::StartUp;

                Dbg("APBoot: Wakeup sequence following...");

                xAPICDriver::write(APIC_REGISTER_ICR_HIGH, hreg.value);
                xAPICDriver::write(APIC_REGISTER_ICR_LOW, lreg.value);
```

```
}

// NOTE: ICRLow and ICRHigh are types in the Silcos kernel. If your code uses direct bit
// manipulations you must replace some code with bit operations.
```

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/)
- Volume 3A:System Programming Guide, Chapter 10.4 for further reading about the LAPIC (http://www.intel.com/products/processor/manuals/)
- Advanced Programmable Interrupt Controller by Mike Rieker (http://web.archive.org/web/20140308064246/http://www.osdever.net/tutorials/pdf/apic.pdf)
- "The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs". Microsoft. 18 September 2010 (http://web.archive.org/web/20100918084750/http://www.microsoft.com/whdc/archive/apic.mspx)
- Pentium Processor System Architecture (https://books.google.nl/books?id=TVzjEZg1--YC&printsec=frontcover)

Categories: Interrupts | Time | Multiprocessing
