Multiprocessing Support for Hobby OSes Explained

by Ben L. Titzer

The latest version of this tutorial may be found at http://www.redpants.org/docs/mpx86.php

Reference Materials

- Intel Multiprocessing Specification
- Intel Software Developer's Manual Volume 3
- Intel 82093AA I/O APIC Manual

Introduction

Many hobby operating system projects start out with very modest goals of being able to boot off of a floppy and load a kernel written in a high level language like C or C++. Some progress further, to the point that they can manage virtual memory and multiple processes, but very few of these operating systems ever get to the point that they support multi-processing with more than one CPU. The reason for this is a general lack of good information on how to accomplish the necessary steps of detecting and initializing other processors in the system.

The design of a multi-processing operating system must be made very carefully and many situations must be taken into account to avoid race conditions that undermine the stability and correctness of a multi-processing OS. Basic locking primitives are needed that protect kernel data structures from concurrent access in situations that can result in corruption, which inevitably lead to instability in the OS kernel itself. This document touches briefly on locking mechanisms, but does not go deeply into the design decisions of a multi-processing operating system. It is meant for the hobby OS developer that understands virtual memory and multithreading and would like to take their OS project to the next level by beginning to add multiprocessing support.

1 - Multiprocessing in Nutshell

How does multiprocessing work? The most basic simplification is that multiple processors can execute code simultaneously and independent of each other. Instead of one processor in a system, there are more than one, from as few as two up to thousands. These processors can either share the same system memory or have separate private memories that only they can access. There can also be configurations in which processors are "clustered" where there may be many physical memories with several processors each.

Systems that share system memory between all processors so that all processors see the same physical memory are called *UMA* for Uniform Memory Access. They are more often called *SMP* systems for Symmetric Multiprocessing. Systems that have separate, private physical memories are called *NUMA* for Non-uniform Memory Access. SMP architectures are generally used where the number of processors accessing the same physical memory is at most a dozen or a few dozen. This is because of the law of diminishing returns: as each processor is added, it has to compete with the other processors in the system for memory bandwidth, and so the speed increase from adding more processors becomes much less than linear.

NUMA architectures, where there is no central memory for the processors to contend over, offers much greater scalability, often into the thousands of processors. NUMA have the disadvantage of larger memory requirements (because the OS and applications are duplicated in many separate memories) and because coordinating the system's execution requires extra communication overhead. SMP and NUMA each have their specific uses. NUMA is used for systems on the scale of super-computers and on tasks that have a high degree of parallel data that is not interdependent. SMP is more useful in smaller systems that operate on interdependent data, such as a PC workstation or a server.

This document only focuses on one uniform memory access architecture, that of the Intel Pentium family of processors, since the Intel platform is the most common among hobby OSes, and SMP multiprocessing machines with Intel architecture processors are relatively commonplace.

1.1 - Basics of an SMP System

SMP systems share the same physical memory between all the processors in the system. There is one copy of the OS kernel that manages resources such as memory and devices. The OS kernel can schedule processes to run on different CPUs without the need to copy any of the process's state from one part of physical memory to the next. Since all CPUs see identical physical memory, they are all equally capable of running any particular process or interacting with the hardware devices. They are also equally capable of running the OS kernel code.

1.2 - Communication in an SMP System via Shared Memory

Processors in the system can communicate to each other by one of two methods. The first is to communicate by reading and writing from the same addresses in physical memory to signal that some condition has been meant or that one processor should perform some task. An example of two processors communicating by reading and writing the same address in memory is as follows:

processor 1:

```
volatile int *what_to_do = SHARED_ADDRESS; // point to some memory
*what_to_do = DO_NOTHING;          // default to do nothing

// wait for other processor to set *what_to_do
while ( *what_to_do == DO_NOTHING ) ;
switch ( *what_do_do )
{
  ...
}
```

processor 2:

```
volatile int *what_to_do = SHARED_ADDRESS; // point to some memory
*what_to_do = DO_SOMETHING_ELSE;  // notify other processor
```

In this example, processor 1 and processor 2 communicate by reading and writing from address SHARED_ADDRESS, which we assume is some constant, previously agreed upon address. The first processor sets this integer in memory to the constant DO_NOTHING and waits in a loop until that integer becomes any other value. The second processor simply writes a value into that shared memory address which causes the first to break out of the while loop and enter the switch statement. The second processor could tell the first to do one of several possible things based on what value it wrote to SHARED_ADDRESS.

Cache Coherency and SMP

What about processor caches? What if the shared memory is cached in one of one of the processors' caches? This would cause massive problems communicating via shared memory because the memory in question would have to be uncached to ensure that changes made to shared memory by one processor are seen by other processors interested in the same memory range. This problem is solved by a *coherency protocol* implemented in hardware that ensures that changes made by one processor are seen by other processors. The details of this scheme aren't particularly interesting in this document and since they make the processor caches appear transparent to software, they are not discussed further.

1.3 - Communicating Better with Interprocessor Interrupts

The about example is a rather clumsy and particularly inefficient way to communicate to other processors. First, the processor "listening" in the while loop isn't doing anything useful while it is waiting for the other processor to signal it. The other problem with this is that there may in fact be more than two processors in the system (remember that there can be dozens in some SMP machines). If more than one of these processors is listening and one processor tries to signal one to do something, then they all will wake up, not just one.

We can reduce these problems by having the listening processor only check the flag periodically and between checks do something useful, but then the processor is less responsive. We could solve the problem of multiple listening processors with flags for each processor, but the latency and busy polling problems still remain. If you are an intermediate OS developer, chances you understand this problem and know the solution already: interrupts. In multiprocessor systems, communication can be made through *interprocessor interrupts (IPIs)* that allow one processor to send an interrupt to another specified processor or range of processors. The ability to interrupt another processor solves both the latency and polling problems. The processor can be doing useful work, but still stay responsive to interrupts from the other processors in the system.

2 - Intel Multiprocessing Specification

Now that we have discussed the differences between polling and interrupts on SMP systems, it is time to consider the more practical questions of how they work and how to use them. For this purpose, to standardize how Intel processors work in a SMP setting, Intel developed a standard called the *Intel Multiprocessing Specification*, which sets standards for the interface between the BIOS/firmware level and system software (OS) level. It is strongly recommended that you download this manual, as it covers some specifics that are important. You can find it here. This manual was introduced with the 486 line of processors which supported multiprocessing. The 386 processors also supported multiprocessing, but saw almost no use as a multiprocessing platform because there were no standards.

2.1 - The APIC module

The centerpiece of the Intel Multiprocessing specification is the *APIC* device, which stands for Advanced Programmable Interrupt Controller. Even beginning OS developers have probably heard of the PIC (Programmable Interrupt Controller) which delivers IRQs to the processor. The APIC module is similar in function to the PIC, but it accepts and directs interrupts among multiple processors. In Intel multiprocessing systems, there is one *local APIC* module for each processor and at least one *IO APIC* that routes interrupt requests among multiple processors. The local APIC module is built into the processor die itself for Pentium family of processors, but is separate for 486 processors. This local module for 486s was a different model (the 82489DX) and had slightly fewer features than the later modules built into the Pentium line of processors. For that reason they are not discussed, and we focus on multiprocessing with the Pentium and higher line of processors.

The local APIC module serves as the only input of interrupts to the processor. The external PIC and IO APICs send their interrupts to the local APIC of the destination processor and that local APIC interrupts the processor. The APIC can be programmed to mask these interrupts 0-255. However, the APIC cannot mask the exceptions 0-21 which are generated internal to the processor.

Each local APIC module has a unique ID that is initialized by the BIOS, firmware, or hardware. The OS is guaranteed that the local APIC IDs are unqiue. Local APICs are also capable of sending IPIs (inter-processor interrupts) to other processors in the system using the local IDs of the destination. This is primarily how the OS communicates with other processors, by programming the current processor's (whichever processor the OS is running on) local APIC chip to send an IPI to a destination APIC ID.

2.2 - Bootup Sequence

The Specification not only defined the the APIC as the basic building block of multiple processor systems, but it also had to define some standards on booting the system so that multiple processor systems could remain backwards compatible. Some guarantee as to the state of the other processors in the system was needed so that an a uniprocessor OS could function correctly on one processor.

The Multiprocessing specification defines a standard boot sequence that guarantees the OS that the system is in a state ready for multiprocessor detection and initialization. The specification states that in the standard boot sequence the BIOS, hardware or firmware (not the OS) will select one of the processors to be designated the *BSP* or Bootstrap Processor. The selection of which processor is the BSP can be either hardwired to physical location, generated randomly, or selected by some other means. The only restriction the specification enforces is that one and only one processor is selected as the BSP and the other processors, called *AP's* for Application Processors are initialized to Real Mode and put into a halted state. The APs' local APICs are initialized such that they will not service any interrupts. The system is initialized so that all interrupts are directed to the BSP. The BSP then boots normally exactly as if the system was a uniprocessor machine.

2.3 - Multiple Processor Detection

The resulting initialization and loading of the OS in uniprocessor mode should be familiar to even beginning OS developers and is not the aim of this document. What is the aim of this document is the steps the operating system must now take to detect and initialize the APs, which are still in a halted state. In order for the OS to detect the presence of multiple processors, the specification requires that the BIOS or firmware construct two tables in physical memory that describes the configuration of the system, including information about processors, IO APIC modules, irq assignments, busses present in the system, and other useful data for the OS. The OS must find these structures and parse them in order to determine what initialization needs to be done. If the OS does not find these tables, then the OS can assume that the system is not multiprocessor capable and it can continue with uniprocessor initialization. This allows an OS compiled for SMP operation to fall back on default, uniprocessor behavior on a uniprocessor system.

Finding the MP Floating Pointer Structure

The first structure the OS must search for is called the *MP Floating Pointer Structure*. This table contains some information pertaining to the multiprocessing configuration and indicates that the system is multiprocessing compliant. This structure has the following format:

MP Floating Pointer Structure			
Field	**Offset**	**Length**	**Description/Use**

Signature	0	4 bytes	This 4 byte signature is the ASCII string "_MP_" which the OS should use to find this structure.
MPConfig Pointer	4	4 bytes	This is a 4 byte pointer to the MP configuration structure which contains information about the multiprocessor configuration.
Length	8	1 byte	This is a 1 byte value specifying the length of this structure in 16 byte paragraphs. This should be 1.
Version	9	1 byte	This is a 1 byte value specifying the version of the multiprocessing specification. Either 1 denoting version 1.1, or 4 denoting version 1.4.
Checksum	10	1 byte	The sum of all bytes in this floating pointer structure including this checksum byte should be zero.
MP Features 1	11	1 byte	This is a byte containing feature flags.
MP Features 2	12	1 byte	This is a byte containing feature flags. Bit 7 reflects the presence of the ICMR, which is used in configuring the IO APIC.
MP Features 3-5	13	3 bytes	Reserved for future use.

The MP Floating Pointer Structure is in one of three memory areas: (1) The first kilobyte of the Extended BIOS Data Area (EBDA). (2) The last kilobyte of base memory (639-640k). (3) The BIOS ROM address space (0xF0000-0xFFFFF). The OS should search for the ASCII string "_MP_" in these three areas. If the OS finds this structure, this indicates the system is multiprocessing compliant and multiple processor initialization should continue. If this structure is not present in any of these three areas, then uniprocessor initialization should continue.

Parsing the MP Configuration Table

The MP Floating Pointer Structure indicates whether the MP Configuration Table exists by the value in **MP Features 1**. If this byte is zero, then the value in **MPConfig Pointer** is a valid pointer to the physical address of the MP Configuration Table. If **MP Features 1** is non-zero, this indicates that the system is one of the default configurations as described in the Intel Multiprocessing Specification Chapter 5. These default configurations are concisely described in that chapter of the specification and we will not discuss them fully here except to say that these default configurations have only two processors and the local APICs have IDs 0 and 1, among a few other nice properties. If one of these default implementations is specified in the MP Floating Pointer Structure, then the OS need not parse the MP Configuration Table, and can initialize the system based on the information in the specification.

The MP Configuration Table contains information regarding the processors, APICs, and busses in the system. It has a header (called the base table) and a series of variable length entries immediately following it in increasing address. The base table has the following format:

MP Configuration Table			
Field	**Offset**	**Length**	**Description/Use**
Signature	0	4 bytes	This 4 byte signature is the ASCII string "PCMP" which confirms that this table is present.
Base Table Length	4	2 bytes	This 2 byte value represents the length of the base table in bytes, including the header, starting from offset 0.

Specification Revision	6	1 byte	This 1 byte value represents the revision of the specification which the system complies to. A value of 1 indicates version 1.1, a value of 4 indicates version 1.4.
Checksum	7	1 byte	The sum of all bytes in the base table including this checksum and reserved bytes must add to zero.
OEM ID	8	8 bytes	An ASCII string that identifies the manufacturer of the system. This string is not null terminated.
Product ID	16	12 bytes	An ASCII string that identifies the product family of the system. This string is not null terminated.
OEM Table Pointer	28	4 bytes	An optional pointer to an OEM-defined configuration table. If no OEM table is present, this field is zero.
OEM Table Size	32	2 bytes	The size (if it exists) of the OEM table. If the OEM table does not exist, this field is zero.
Entry Count	34	2 bytes	The number of entries following this base header table in memory. This allows software to find the end of the table when parsing the entries.
Address of Local APIC	36	4 bytes	The physical address where each processor's local APIC is mapped. Each processor memory maps its own local APIC into this address range.
Extended Table Length	40	2 bytes	The total size of the extended table (entries) in bytes. If there are no extended entries, this field is zero.
Extended Table Checksum	42	1 byte	A checksum of all the bytes in the extended table. All off the bytes in the extended table must sum to this value. If there are no extended entries, this field is zero.

The MP Configuration Table is immediately followed by **Entry Count** entries that describe the configuration of processors, busses and IO APICs in the system. The first byte of each entry denotes the entry type, e.g. a processor entry or a bus entry. The entries are sorted by entry type in ascending order. The table entry types are summarized as follows:

MP Configuration Table Entries			
Entry Description	**Entry Type Code**	**Length**	**Comments**
Processor	0	20 bytes	An entry describing a processor in the system. One entry per processor.
Bus	1	8 bytes	An entry describing a bus in the system. One entry per bus.
IO APIC	2	8 bytes	An entry describing an IO APIC present in the system. One entry per IO APIC.
IO Interrupt Assignment	3	8 bytes	An entry describing the assignment of an interrupt source to an IO APIC. One per bus interrupt source.
Local Interrupt Assignment	4	8 bytes	An entry describing a local interrupt assignment in the system. One entry per system interrupt source.

Since the entries of the MP Configuration Table are sorted by entry type in ascending order, the first entries will be all the processor entries, followed by all the bus entries, followed by the IO APIC entries, and so on. The OS should parse these entries in order to discover how many processors, how many IO APICs, and other information it will need to initialize the system. The processor entries have the format:

Processor Entry			
Field	**Offset (in bytes:bits)**	**Length**	**Description/Use**
Entry Type	0	1 byte	Since this is a processor entry, this field is set to 0.
Local APIC ID	1	1 byte	This is the unique APIC ID number for the processor.
Local APIC Version	2	1 byte	This is bits 0-7 of the Local APIC version number register.
CPU Enabled Bit	3:0	1 bit	This bit indicates whether the processor is enabled. If this bit is zero, the OS should not attempt to initialize this processor.
CPU Bootstrap Processor Bit	3:1	1 bit	This bit indicates that the processor entry refers to the bootstrap processor if set.
CPU Signature	4	4 bytes	This is the CPU signature as would be returned by the CPUID instruction. If the processor does not support the CPUID instruction, the BIOS fills this value according to the values in the specification.
CPU Feature flags	8	4 bytes	This is the feature flags as would be returned by the CPUID instruction. If the processor does not support the CPUID instruction, the BIOS fills this value according to values in the specification.

Bus entries identify the kinds of buses in the system. The BIOS is responsible for assigning them each a unique ID number. The entries allow the BIOS to communicate to the OS the buses in the system. The format of the entries is described in the Intel Multiprocessing Specification Chapter 5. Because using and initializing buses is beyond the scope of this document, bus entries are not discussed.

The configuration table contains at least one IO APIC entry which provides to the OS the base address for communicating with the IO APIC and its ID. The entry for an IO APIC has the following format:

IO APIC Entry			
Field	**Offset (in bytes:bits)**	**Length**	**Description/Use**
Entry Type	0	1 byte	Since this is an IO APIC entry, this field is set to 2.
IO APIC ID	1	1 byte	This is the ID of this IO APIC.
IO APIC Version	2	1 byte	This is bits 0-7 of the IO APIC's version register.
IO APIC Enabled	3:0	1 bit	This bit indicates whether this IO APIC is enabled. If this bit is zero, the OS should not attempt to access this IO APIC.
IO APIC Address	4	4 bytes	This contains the physical base address where this IO APIC is mapped.

3 - Initializing and Using the local APIC

Now that we are able to detect the processors and IO APICs in a system, it is necessary to initialize and configure the bootstrap processor's local APIC so that it can begin to send interrupts to the other processors in the system. Interprocessor interrupts are the best way to communicate between processors in certain situations, and as we will see, they are used by the bootstrap processor to awaken the other processors in the system.

3.1 - Memory Mappings of APIC Modules

Each local APIC module is memory mapped into the address space of its corresponding processor. They are all mapped to their local processor's address space at the same address so that when a processor accesses this address range it is accessing its own local APIC. However, for an IO APIC, it is mapped into the address space of all processors at the same address so that all processors can address the same IO APIC through the same address range. Multiple IO APICs each have their own address range in which they are mapped, but are, again, mapped globally and accessable from all processors. The address ranges APICs are given as follows:

APIC Memory Mappings		
APIC Type	**Default address**	**Alternate Address**
Local APIC	0xFEE00000	If specified, the value of the **Address of Local APIC** field in the MP Configuration Table.
First IO APIC	0xFEC00000	If specified, the value of the **IO APIC Address** field in the IO APIC entry in the MP Configuration Table.
Additional IO APICs	-	The value of the **IO APIC Address** field in the IO APIC entry in the MP Configuration Table.

3.2 - The Local APIC's Register Set

In order for the OS to begin to communicate with the other processors present in the system, it must first initialize its own local APIC module. The local APIC module is the means by which the local processor can send interrupts to the other processors and is memory mapped into the address space of the processor at the addresses in the previous table. The APIC uses no IO ports and is configured by writing the appropriate settings into the APIC's registers at the correct memory offsets. The registers' offsets are summarized in the following table:

Local APIC Register Addresses		
Offset	**Register Name**	**Software Read/Write**
0x0000h - 0x0010	reserved	-
0x0020h	**Local APIC ID Register**	Read/Write
0x0030h	**Local APIC ID Version Register**	Read only
0x0040h - 0x0070h	reserved	-
0x0080h	**Task Priority Register**	Read/Write
0x0090h	**Arbitration Priority Register**	Read only
0x00A0h	**Processor Priority Register**	Read only
0x00B0h	**EOI Register**	Write only
0x00C0h	reserved	-

0x00D0h	**Logical Destination Register**	Read/Write
0x00E0h	**Destination Format Register**	Bits 0-27 Read only, Bits 28-31 Read/Write
0x00F0h	**Spurious-Interrupt Vector Register**	Bits 0-3 Read only, Bits 4-9 Read/Write
0x0100h - 0x0170	**ISR 0-255**	Read only
0x0180h - 0x01F0h	**TMR 0-255**	Read only
0x0200h - 0x0270h	**IRR 0-255**	Read only
0x0280h	**Error Status Register**	Read only
0x0290h - 0x02F0h	reserved	-
0x0300h	**Interrupt Command Register 0-31**	Read/Write
0x0310h	**Interrupt Command Register 32-63**	Read/Write
0x0320h	**Local Vector Table (Timer)**	Read/Write
0x0330h	reserved	-
0x0340h	**Performance Counter LVT**	Read/Write
0x0350h	**Local Vector Table (LINT0)**	Read/Write
0x0360h	**Local Vector Table (LINT1)**	Read/Write
0x0370h	**Local Vector Table (Error)**	Read/Write
0x0380h	**Initial Count Register for Timer**	Read/Write
0x0390h	**Current Count Register for Timer**	Read only
0x03A0h - 0x03D0h	reserved	-
0x03E0h	**Timer Divide Configuration Register**	Read/Write
0x03F0h	reserved	-

Note that the local APIC's registers are divided into 32 bit words that are aligned on 16 byte boundaries. Registers that are larger than 32 bits are split into multiple 32 bit words, aligned on successive 16 byte boundaries. The Intel Multiprocessing Specification states that all local APIC registers must be accessed with 32 bit reads and writes.

3.3 - Initializing the BSP's Local APIC

In order for the OS to communicate with the other processors in the system, it first must enable and configure its local APIC. Software must first enable the local APIC by setting a bit in a register and programming other registers with vectors to handle bus and inter-processor interrupts.

Spurious-Interrupt Vector Register

The **Spurious-Interrupt Vector Register** contains the bit to enable and disable the local APIC. It also has a field to specify the interrupt vector number to be delivered to the processor in the event of a *spurious interrupt*. This register is 32 bits and has the following format:

32bit Spurious-Interrupt Vector Register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

.	FC	EN	VECTOR	1	1	1	1

- **EN bit** - This allows software to enable or disable the APIC module at any time. Writing a value of 1 to this bit enables the APIC module, and writing a value of 0 disables it.
- **FC Bit** - This bit indicates whether focus checking is enabled for the current processor. A value of 0 indicates focus checking is enabled, and a value of 1 indicates it is disabled. For our purposes, this bit can be ignored.
- **VECTOR** - This field of the Spurious-Interrupt Vector Register specifies which interrupt vector is delivered to the processor in the event of a *spurious interrupt*. Bits 0-3 of this vector field are hard-wired to 1111b, or 15. Bits 4-7 of this field are programmable by software.

A *spurious interrupt* can happen when all pending interrupts are masked or there are no pending interrupts during an internal interrupt acknowledge cycle of the APIC. The APIC module delivers an interrupt vector to its local processor specified by the value in the **VECTOR** field of this register. The processor then transfers control to the interrupt handler in the IDT, at the vector number delivered to it by the APIC. Basically, the **VECTOR** field specifies which interrupt handler to transfer control to in the event of a spurious interrupt. Spurious interrupts happen because of certain interactions within the APIC's hardware itself, and do not reflect any meaningful information. Software can safely ignore these interrupts, and should program this vector to refer to an interrupt handler that ignores the interrupt.

Local APIC Version and Local APIC ID Registers

The **Local APIC Version Register** is a read-only register that the APIC reports its version information to software. It also specifies the maximum number of entries in the **Local Vector Table** (LVT). The **Local APIC ID Register** stores the ID of the local APIC.

Local APIC Version Register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
.								MAXIMUM LVT ENTRY								.								VERSION							

- **Maximum LVT Entry** - Indicates the number of the Maximum LVT entry. For Pentium processors, this number is 3 (4 entries total) and for P6 family, this is 4 (5 entries total).
- **Version** - Indicates the version number of the local APIC module. For 82489DX APICs, this number is 0h. For integrated APICs of the Pentium family and higher, this number is 1h.

Local APIC ID Register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
.				APIC ID				.																							

Local Vector Table

The **Local Vector Table** allows software to program the interrupt vectors that are delivered to the processor in the event of errors, timer events, and LINT0 and LINT1 interrupt inputs. It also allows software to specify status and mode information to the APIC module for the local interrupts.

Local Vector Table

.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timer	.														TP	M	.			DS	.				VECTOR							
LINT0	.															M	TM	RI	IP	DS	.	DMODE			VECTOR							
LINT1	.															M	TM	RI	IP	DS	.	DMODE			VECTOR							
ERROR	.															M	.			DS	.				VECTOR							
PCINT	.															M	.			DS	.	DMODE			VECTOR							

- **Vector**: The interrupt vector number.
- **DMODE (Delivery Mode)**: Defined only for the local interrupts LINT0, LINT1, and PCINT (the performance monitoring counter). It can be one of three defined values:
  - **000 (Fixed)** - Delivers the interrupt to the processor as specified in the corresponding LVT entry.
  - **100 (NMI)** - The interrupt is delivered to the local processor as a NMI (non-maskable interrupt) and the vector information is ignored. The interrupt is treated as an edge-triggered interrupt regardless of how software had programmed it.
  - **111 (ExtINT)** - Delivers the interrupt to the processor as if it had originated in an external controller such as an 8249A PIC. The external controller is expected to supply the vector information. The interrupt is always treated as level trigger, regardless of how the software had programmed the entry.
- **DS (Delivery Status)** - Read only to software. A value of 0 (idle) indicates that there are no pending interrupts for this interrupt or that the previous interrupt from this source has completed. A value of 1 (send pending) indicates that the interrupt transmission has begun but has not yet been completely accepted.
- **IP (Interrupt Polarity)** - Specifies the interrupt polarity of the interrupt source. A value of 0 indicates active high and a value of 1 indicates active low.
- **RI (Remote Interrupt Request Register bit)** - For level triggered interrupts, this bit is set when the APIC module accepts the interrupt and is cleared upon EOI. Undefined for edge triggered interrupts.
- **TM (Trigger Mode)** - When the delivery mode is Fixed, (0) indicates edge-sensitivity and (1) indicates level-sensitivity.
- **M (Mask)** - Indicates whether the interrupt is masked. A value of 1 indicates the interrupt is masked, while 0 indicates the interrupt is unmasked.
- **TP (Timer Periodic Mode)** - Indicates whether the timer interrupt should be fired periodically (1) or only once (0).

3.4 - Issuing Interrupt Commands

The local APIC module has a 64 bit register called the **Interrupt Command Register** that software can use cause the APIC to issue interrupts to other processors. A write to the low 32 bits of the register causes the command specified in the write operation to be issued. The format of the Interrupt Command Register is as follows:

Interrupt Command Register																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
DESTINATION FIELD								.																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
.												DSH		.		TM	LV	.	DS	DM	DMODE			VECTOR							

- **Vector** - Indicates the vector number identifying the interrupt being sent.
- **DMODE (Delivery Mode)** - Specifies how the APICs in the destination field should handle the interrupt being sent. All inter-processor interrupts are treated as edge-triggered, even if programmed otherwise.
  - **000 (Fixed)** - Delivers the interrupt to the processors listed in the destination field according to the information in the ICR.
  - **001 (Lowest Priority)** - Same as fixed mode, except the interrupt is delivered to the processor executing at the lowest priority among the set of processors specified in the destination field.
  - **010 (SMI)** - Only the edge triggered mode is allowed. The vector field must be programmed to 00b.
  - **011 (Reserved)**
  - **100 (NMI)** - Delivers the interrupt as an NMI to all processors listed in the destination. The vector information is ignored.
  - **101 (INIT)** - Delivers the interrupt as an INIT, causing all processors in the destination to assume their INIT state. Vector information is ignored.
  - **101 (INIT Level De-assert)** - (Specified by setting Level to 0 and Trigger Mode to 1). The interrupt is delivered to all processors regardless of the destination field. Causes all the APICs to reset their arbitration IDs to the local APIC IDs.
  - **110 (Startup)** - Sends a Startup message to the processors listed in the destination field. The 8-bit vector information is the physical page number of the address for the processors to begin executing from. This message is not automatically retried, and software may need to retry in the case of failure.
- **DM (Destination Mode)** - Indicates whether the destination field contains a physical (0) or logical (1) address.
- **DS (Delivery Status)** - Indicates idle (0), that there is no activity for this interrupt, or send pending (1), that the transmission has started, but has not yet been completely accepted.
- **LV (Level)** - For the INIT De-assert mode, this is set to 0. For all other delivery modes, this is set to 1.
- **TM (Trigger Mode)** - Used from the INIT De-assert mode only.
- **DSH (Destination Shorthand)** - Indicates whether shorthand notation is being used to specify the destination of the interrupt. If destination shorthand is used, then the destination field is ignored. This field can have the values:
  - **00 (No shorthand)** - Indicates no shorthand is being specified and that the destination field contains the destination.
  - **01 (Self)** - Indicates that the current APIC is the only destination. Useful for self interrupts.
  - **10 (All)** - Broadcasts the message to all APICs, including the processor sending the interrupt.
  - **11 (All excluding Self)** - Broadcasts the message to all APICs, excluding the processor sending the interrupt.
- **Destination Field** - When the destination shorthand field is set to 00 and the destination mode is physical, the destination field (bits 56-59) contains the APIC ID of the destination. When the mode is logical, the interpretation of this field is more complicated. See the Intel SDM Vol 3, Chap 7, for

details.

4 - Application Processor Startup

5 - MP Detection and Initialization Recap

1. The BIOS selects the BSP and begins uniprocessor startup, initializing the APs to Real Mode and halting them.
2. The OS code (either bootstrap or kernel) searches for the **MP Floating Pointer** structure.
3. The OS uses the **MP Floating Pointer** structure to select a default configuration or to find the **MP Configuration Table**.
4. The OS parses the **MP Configuration Table** to determine how many processors and IO APICs are in the system.
5. The OS initializes the bootstrap processor's local APIC.
6. The OS sends **Startup IPIs** to each of the other processors with the address of trampoline code.
7. The trampoline code initializes the AP's to protected mode and enters the OS code to being further initialization.
8. When the AP's have been awakened and initialized, the BSP can initialize the IO APIC into **Symmetric IO mode**, to allow the AP's to begin to handle interrupts.
9. The OS continues further initialization, using locking primitives as necessary.

6 - Locks and IPIs
