Heinrich Heine
Universität
Düsseldorf

Bachelor's Thesis

Interrupt Handling using the x86 APIC

submitted by

Christoph Urlacher

from Ratingen

Department Operating Systems
Prof. Dr. Michael Schöttner
Heinrich Heine University Düsseldorf

March 8th, 2023

Primary Reviewer:	Prof. Dr. Michael Schöttner
Secondary Reviewer:	Prof. Dr. Stefan Conrad
Supervisor:	M.Sc. Fabian Ruhland

Contents

Acronyms

ACPI advanced configuration and power interface.

AML acpi machine language.

AP application processor.

APIC advanced programmable interrupt controller.

APR arbitration priority register.

BSP bootstrap processor.

CPUID cpu identification.

EOI end of interrupt.

ESR error status register.

GDT global descriptor table.

GSI global system interrupt.

ICH intel input/output controller hub.

ICR interrupt command register.

IDT interrupt descriptor table.

IDTR interrupt descriptor table register.

IMCR interrupt mode control register.

IMR interrupt mask register.

INTI interrupt input.

IPI inter-processor interrupt.

IRQ interrupt request.

IRR interrupt request register.

ISA industry standard architecture.

ISR in-service register.

LVT local vector table.

MADT multiple apic descriptor table.

MMIO memory mapped input/output.

MSI message-signaled interrupt.

MSR model specific register.

NMI non-maskable interrupt.

PIC programmable interrupt controller.

PIT programmable interval timer.

REDTBL redirection table.

SIPI startup inter-processor interrupt.

SMP symmetric multiprocessing.

SVR spurious interrupt vector register.

TPR task-priority register.

TSS task state segment.

Glossary

application processor a processor inside an SMP system, e.g. a CPU core.

bootstrap processor the application processor used to boot an SMP system.

discrete APIC the predecessor of the xApic architecture where the local APIC was not integrated into the CPU core, register access is handled through MMIO.

global system interrupt an abstraction used by ACPI to decouple interrupts from hardware interrupt lines.

I/O APIC a part of the APIC architecture inside the chipset, responsible for receiving external interrupts.

in-service register a register, part of the PIC and local APIC, which keeps track of interrupts that are being serviced.

interrupt command register a register of the local APIC used to issue interprocessor interrupts.

interrupt mode control register a register present in some systems to choose the physically connected interrupt controller.

interrupt override information from ACPI, that describes how interrupt lines correspond to global system interrupts.

interrupt request register a register, part of the PIC and local APIC, which keeps track of received interrupts.

local APIC a part of the APIC architecture inside a CPU core, responsible for receiving local interrupts and communication with the I/O APIC.

local interrupt an CPU internal interrupt handled by the local APIC, like the APIC timer interrupt.

local vector table a set of registers, part of the local APIC, that configure how local interrupts are handled.

message-signaled interrupt an interrupt sent in-band over a PCI-bus.

PIC mode only use the PIC for interrupt handling, like the PC/AT.

spurious interrupt vector register a register of the local APIC which contains the APIC software enable flag and the spurious interrupt vector.

symmetric I/O mode use the I/O APIC in combination with the local APIC for interrupt handling in multiprocessor systems.

task-priority register a register of the local APIC which determines interrupt handling order and priority theshold.

virtual wire mode use the local APIC in combination with the PIC as external interrupt controller.

x2Apic a revision of the APIC architecture, register access is handled through MSRs.

xApic a revision of the APIC architecture, register access is handled through MMIO.

Chapter 1

Introduction

Computer systems need to interact with their surroundings, for instance by reading values from sensors, controlling external appliances or interacting with a user through human interface devices. In each of these scenarios, the system's CPU has to react to "external changes", like a key press or sensor reading. An efficient hardware solution to this problem are "interrupts".

Through the past, interrupt hardware went through different iterations: Intel introduced the "Programmable Interrupt Controller" for the "8085" processor in 1976, with a revised version for the "8086" processor. With modern standards like multicore processors, peripheral extendability, greater flexibility or higher performance, the Programmable Interrupt Controller could no longer meet its requirements.

In this thesis, support for the "Advanced Programmable Interrupt Controller", a modern, multiprocessing capable and widely used interrupt controller architecture, introduced by Intel for the x86 "i486" processor, will be implemented into hhuOS, "A small operating system for learning purposes" [1]. This support will cover a complete replacement of the older Programmable Interrupt Controller, introduction of an alternative timer – a part of the APIC architecture – for scheduling, and utilizing the APIC to boot multiprocessor systems.

The following chapter explains important background concepts, Chapter 3 describes how to use the APIC to handle local and external interrupts in singlecore and multicore systems based on the "IA-32 Architecture Software Developers Manual" [2], in Chapter 4 the implementation and integration into hhuOS are explained, Chapter 5 deals with the testing process of the developed software on emulated and real hardware, and Chapter 6 draws conclusions regarding the previous implementation and future improvements.

Chapter 2

Background

In this chapter, important domain-specific concepts will be explained, to create the necessary foundation to follow Chapter 3 and Chapter 4. Important terms present in the glossary are marked in **bold** on their first occurrence.

2.1 Handling of External Events

There are two different strategies to “listen” for external events: “Polling” and “Interrupts”.

The first strategy works by periodically *polling* a device to check for changes. This could mean reading a register of the keyboard every 50 ms to determine if a key was pressed, or reading a sensor output even when the value remains unchanged. Because every device would have to be polled constantly, no matter if any change actually occurred, this method is computationally inefficient (although easy to implement without extra hardware).

The second strategy are *interrupts*. Instead of the CPU actively “looking” for changes, the devices signal the events to the CPU themselves. Every time the CPU is notified of an external event, it pauses the current code execution and calls a function designated to handle this specific event.

This approach is much more efficient than the polling strategy, because the CPU does not have to waste its processing power to check for events when none are occurring.

2.2 Fundamental Concepts

2.2.1 Interrupt

When a device signals an external event to the CPU, the current code execution gets *interrupted*, to handle the event.

Interrupts can be caused by all sorts of events, like key presses on a keyboard or packets received via a network card. These interrupts from external hardware devices are called *external interrupts*.

Other types of interrupts mentioned in this thesis are **IPIs (inter-processor interrupts)**, **MSIs (message-signaled interrupts)** and **local interrupts**:

- IPIs: Interrupts sent between different processor cores in multiprocessor systems, e.g. to initialize different cores on system startup or synchronize caches.
- MSIs: Interrupts sent in-band[1], e.g. over a PCI-bus[2].
- Local Interrupts: Some specific CPU-internal interrupts, e.g. for performance monitoring.

Some interrupts are essential for CPU operation. These **NMIs (non-maskable interrupts)** always have to be handled and cannot be ignored (except in some special cases, like booting). Hardware error interrupts are a typical example for NMIs.

The existence of NMIs hints that it is possible to ignore regular interrupts. Marking an interrupt as “ignored” is called *masking* an interrupt.

[1] In in-band signaling, control information is sent over the same channel as the data, while out-of-band signaling uses a dedicated control line.

[2] PCI supports MSIs since PCI 2.2 [3, sec. 6.8].

2.2.2 Interrupt Controller

A computer system has to process interrupts from many sources, so it is practical to have a designated *interrupt controller*, that receives interrupts from different devices and forwards them to the CPU. The masking functionality is integrated into the interrupt controller, so masked interrupts do not reach the CPU at all.

The interrupt controller receives interrupts through signals over physical connections (*interrupt lines*) to different devices. These signals can be represented in multiple ways through the level on the interrupt lines:

Trigger mode:

- Edge-triggered: A signal is received when the interrupt line level changes.
- Level-triggered: A signal is received when the interrupt line reaches a certain level.

Pin polarity:

- Active-high: A signal is received when the interrupt line level changes from either low to high (in edge-triggered mode) or reaches a level above a threshold (in level-triggered mode).
- Active-low: A signal is received when the interrupt line level changes from either high to low (in edge-triggered mode) or reaches a level below a threshold (in level-triggered mode).

A signal is represented through a combination of trigger mode and pin polarity, so there are a total of four combinations.

When the interrupt controller receives an interrupt signal, it requests the CPU to handle the specific event: The interrupt controller sends an **IRQ (interrupt request)** to the CPU. Because there are multiple devices that can signal interrupts, an IRQ is usually indexed by its **INTI (interrupt input)** on the interrupt controller: If a keyboard is connected to the interrupt controller on INTI1, the CPU receives an IRQ1.

To signal that an interrupt is being handled by the software, the interrupt controller receives an **EOI (end of interrupt)** notice from the OS.

Information on specific interrupt controllers follows in Section 2.4.

2.2.3 Interrupt Handler

When the CPU receives an IRQ, it pauses its current code execution to handle the interrupt (see Figure 2.1). This is done by executing the *interrupt handler* function, that is registered to the specific interrupt.

During interrupt servicing (execution of the interrupt handler), other interrupts can occur. When the execution of an interrupt handler is paused to handle another interrupt, this is called a *cascaded interrupt*.

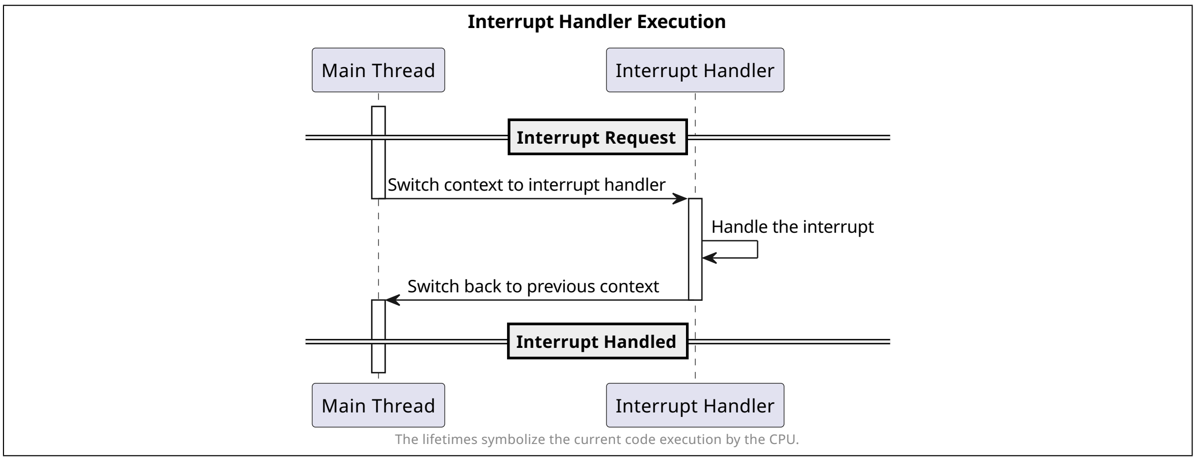

Interrupt Handler Execution
Main Thread
Interrupt Handler
Interrupt Request
Switch context to interrupt handler
Handle the interrupt
Switch back to previous context
Interrupt Handled
Main Thread
Interrupt Handler
The lifetimes symbolize the current code execution by the CPU.

Figure 2.1: Interrupt Handler Execution.

Interrupt handlers have to be registered to different IRQs, so the CPU can locate and execute the specific code that is designated to handle a certain interrupt. The interrupt handler addresses are stored in an *interrupt vector table*: Each address corresponds to an *interrupt vector* number, interrupt vectors 0–31 are reserved for the CPU, interrupt vectors 32–255 are usable for IRQs. In Intel's x86 IA-32 CPU architecture, this table is called the **IDT (interrupt descriptor table)**. The **IDTR (interrupt descriptor table register)** keeps the location of the IDT, it is written by the `lidt` [4] instruction.

2.2.4 Interrupt Trigger Mode

When an edge-triggered IRQ is received, its interrupt handler is called a single time to service the interrupt, which does not require interacting with the device the interrupt originated from. The handler of an edge-triggered IRQ is called every time the interrupt line changes its level (low-to-high or high-to-low, as specified by the pin polarity). This could lead to problems if "glitches" occur on the interrupt line.

An alternative is the level-triggered IRQ: When the interrupt line is above/below a certain level, the interrupt is signaled continuously. Servicing the interrupt thus requires not only notifying the interrupt controller (EOI), but also interaction with the device the interrupt originated from, to deassert the interrupt line. Otherwise, the interrupt handler would be called again after completion.

2.2.5 Spurious Interrupt

When an interrupt "disappears" while the interrupt controller is issuing the IRQ, the interrupt controller sends a *spurious interrupt*. The reason for this could be electrical noise on the interrupt line, masking of an interrupt through software at the same moment the IRQ was dispatched, or incorrectly sent EOIs. Thus, before an interrupt handler is called, the OS has to check the validity of the occurred interrupt and ignore it, if it is deemed spurious.

2.3 Used Technologies

2.3.1 Advanced Configuration and Power Interface

ACPI (advanced configuration and power interface) allows the kernel to gather information about the system hardware during runtime. It also provides interactive functionality for power management, plug and play, hot-swapping or status monitoring. To interact with ACPI, the **AML (acpi machine language)** [5, sec. 16], a small language interpreted by the kernel, has to be used[3].

ACPI defines abstractions for different types of hardware, that are organized in multiple *system description tables*. In this thesis, ACPI 1.0 [5] is used to read information about the system's interrupt hardware configuration, located in the ACPI **MADT (multiple apic descriptor table)** [5, sec. 5.2.8]. The MADT contains information on used interrupt controllers (version, physical memory addresses to access registers, etc.), available CPU cores in multiprocessor systems and specific interrupt configuration (trigger mode and pin polarity).

To allow compatibility to different interrupt controllers, ACPI abstracts external interrupts through **GSIs (global system interrupts)** [5, sec. 5.2.9]. Different interrupt controllers may have the same devices connected to different interrupt lines, maintaining a mapping between actual hardware interrupt lines and GSIs allows the OS to operate on different interrupt controllers[4].

2.3.2 CPU Identification

CPUID (cpu identification) is used to gather information about a system's CPU. The x86 architecture provides the `cpuid` [4] instruction, which allows the software to identify present CPU features at runtime, e.g. what instruction set extensions a processor implements[5].

2.3.3 Symmetric Multiprocessing

SMP (symmetric multiprocessing) is a form of multiprocessing, where a system consists of multiple, homogeneous CPU cores, that all have access to a shared main memory and the I/O devices. SMP systems are controlled by a single OS, that treats all cores as individual, but equal processors. The shared main memory allows every core to work on an arbitrary task, independent of its memory location. Programming for SMP systems requires the usage of multithreading, to allow for an efficient distribution of tasks to multiple CPU cores.

In SMP systems, a single CPU core is active initially, this core is called the **BSP (bootstrap processor)**. Other cores, called **APs (application processors)**, will be initialized

[3] In this thesis, information from ACPI is only read, so AML is not required.

[4] Additional information in Section 3.2.1.

[5] This thesis uses CPUID to determine what APIC feature set is supported, see "APIC" and "x2APIC" features in the CPUID application note [6, sec. 5.1.2].

by this core.

2.3.4 Model Specific Register

A **MSR (model specific register)** is a special control register in an IA-32 CPU, that is not guaranteed to be present in future CPU generations (it is not part of the architectural standard). MSRs that carried over to future generations and are now guaranteed to be included in future IA-32 or Intel 64 CPUs are called “architectural” MSRs. To interact with MSRs, the special instructions `rdmsr` and `wrmsr` are used [4]. These instructions allow atomic read/write operations on MSRs[6].

2.3.5 Memory Mapped I/O

MMIO (memory mapped input/output) is a way for the CPU to perform I/O operations with an external device. Registers of external devices are mapped to the same address space as the main memory, thus a memory address can either be a location in physical memory, or belong to a device’s register. Read and write operations are then performed by reading or writing the address the register is mapped to.

2.3.6 Programmable Interval Timer

The **PIT (programmable interval timer)** [7] is a hardware timer from the original IBM PC. It consists of multiple decrementing counters and can generate a signal (e.g. an interrupt) once 0 is reached. This can be used to control a preemptive[7] scheduler or the integrated PC speaker.

2.4 Intel’s Interrupt Controllers

Because it is logistically difficult to connect every external device directly to the CPU, this task is delegated to a dedicated interrupt controller. Notable interrupt controllers (and the only ones considered in this thesis) are the Intel **PIC (programmable interrupt controller)** and Intel **APIC (advanced programmable interrupt controller)**.

2.4.1 Programmable Interrupt Controller

The Intel “8259A” PIC [8] is the interrupt controller Intel used in the original Intel “8086”[8], the “father” of the x86 processor architecture. A single PIC has eight interrupt lines for external devices, but multiple PICs can be cascaded for a maximum of $8 \cdot 8 = 64$ interrupt lines with nine PICs. The most common PIC configuration

[6] Detailed description in the IA-32 manual [2, sec. 4.2].

[7] A scheduler is called “preemptive”, when it is able to forcefully take away the CPU from the currently working thread, even when the thread did not signal any form of completion.

[8] The 8259 was used in the Intel “8080” and “8085”, the 8259A is a revised version for the Intel 8086

supports 15 IRQs with two PICs (see Figure 2.2): A slave, connected to INTI2 of the master, which is connected to the CPU[9].

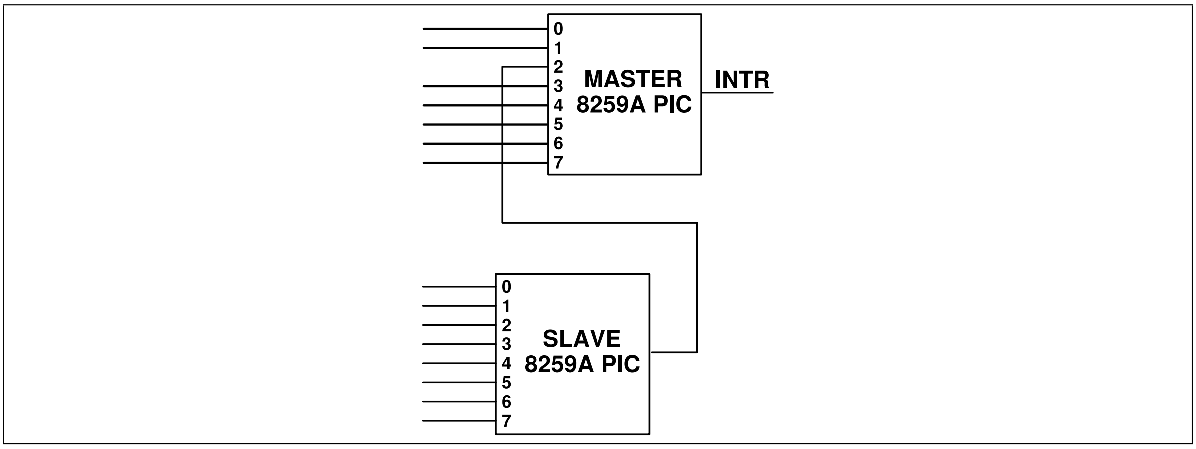


Figure 2.2: The PC/AT PIC Architecture [9, sec. 1.13].

If multiple interrupts get requested simultaneously, the PIC decides which one to service first based on its *interrupt priority*. The PIC IRQ priorities are determined by the used interrupt lines, IRQ0 has the highest priority.
The PIC interrupt handling sequence can be briefly outlined as follows:

1. When the PIC receives an unmasked interrupt from a device (PIC interrupts are edge-triggered with high pin polarity), it sets the corresponding bit in its **IRR (interrupt request register)**[10].

2. The IRQ is dispatched to the CPU.

3. The CPU acknowledges the IRQ.

4. The PIC sets the corresponding bit in its **ISR (in-service register)**[11] and clears the previously set bit in the IRR. If the same interrupt occurs again, it can now be queued a single time in the IRR.

5. When the interrupt is being handled by the OS, it sends an EOI to the PIC, which clears the corresponding bit in the ISR.

With the introduction of multiprocessor systems and devices with an ever-increasing need for large amounts of interrupts (like high-performance networking hardware), the PIC reached its architectural limits:

- Multiple CPU cores require sending IPIs for initialization and coordination, which is not possible using the PIC.

[9] This configuration is called the "PC/AT PIC Architecture", as it was used in the IBM PC/AT [9, sec. 1.13].

[10] Received IRQs that are requesting servicing are marked in the IRR.

[11] IRQs that are being serviced are marked in the ISR. To reduce confusion, interrupt service routines will be denoted as "interrupt handler" instead of "ISR".

- The PIC is hardwired to a single CPU core. It can be inefficient to handle the entire interrupt workload on a single core.
- The IRQ priorities depend on how the devices are physically wired to the PIC.
- The PC/AT PIC architecture only has a very limited number of interrupt lines.
- The PIC does not support MSIs, which allow efficient handling of PCI interrupts.

2.4.2 Advanced Programmable Interrupt Controller

The original Intel "82489DX" **discrete APIC** was introduced with the Intel "i486" processor. It consisted of two parts: A **local APIC**, responsible for receiving special **local interrupts**, and the **I/O APIC**, responsible for receiving external interrupts (and forwarding them to the local APIC). Unlike in modern systems, the i486's local APIC was not integrated into the CPU core (hence "discrete"), see Figure 2.3.
The i486 was superseded by the Intel "P54C"[12], which contained an *integrated APIC* for the first time (see Figure 2.4), which is now the default configuration.
The APIC was designed to fix the shortcomings of the PIC, specifically regarding multiprocessor systems:

- Each CPU core contains its own local APIC. It has the capabilities to handle local interrupts and IPIs, and communicate with the chipset I/O APIC.
- The chipset I/O APIC allows configuration on a per-interrupt basis (priorities, destination, trigger mode or pin polarity are all configurable). Also, it is able to distributes interrupts to different CPU cores, allowing the efficient processing of large amounts of interrupts[13].
- The order in which external devices are connected to the I/O APIC is flexible.
- The APIC architecture supports MSIs.

In comparison to the PIC, the external interrupt handling sequence changed:

1. An external interrupt arrives at the I/O APIC.
2. The I/O APIC redirects the interrupt to the local APIC, which sets the corresponding IRR bit.
3. The local APIC dispatches the IRQ.
4. The CPU acknowledges the IRQ and the local APIC sets the corresponding ISR bit.

[12] The Intel P54C is the successor of the Intel P5, the first Pentium processor.

[13] There are tools that dynamically reprogram the I/O APIC to distribute interrupts to available CPU cores, depending on heuristics of past interrupts (IRQ-balancing).

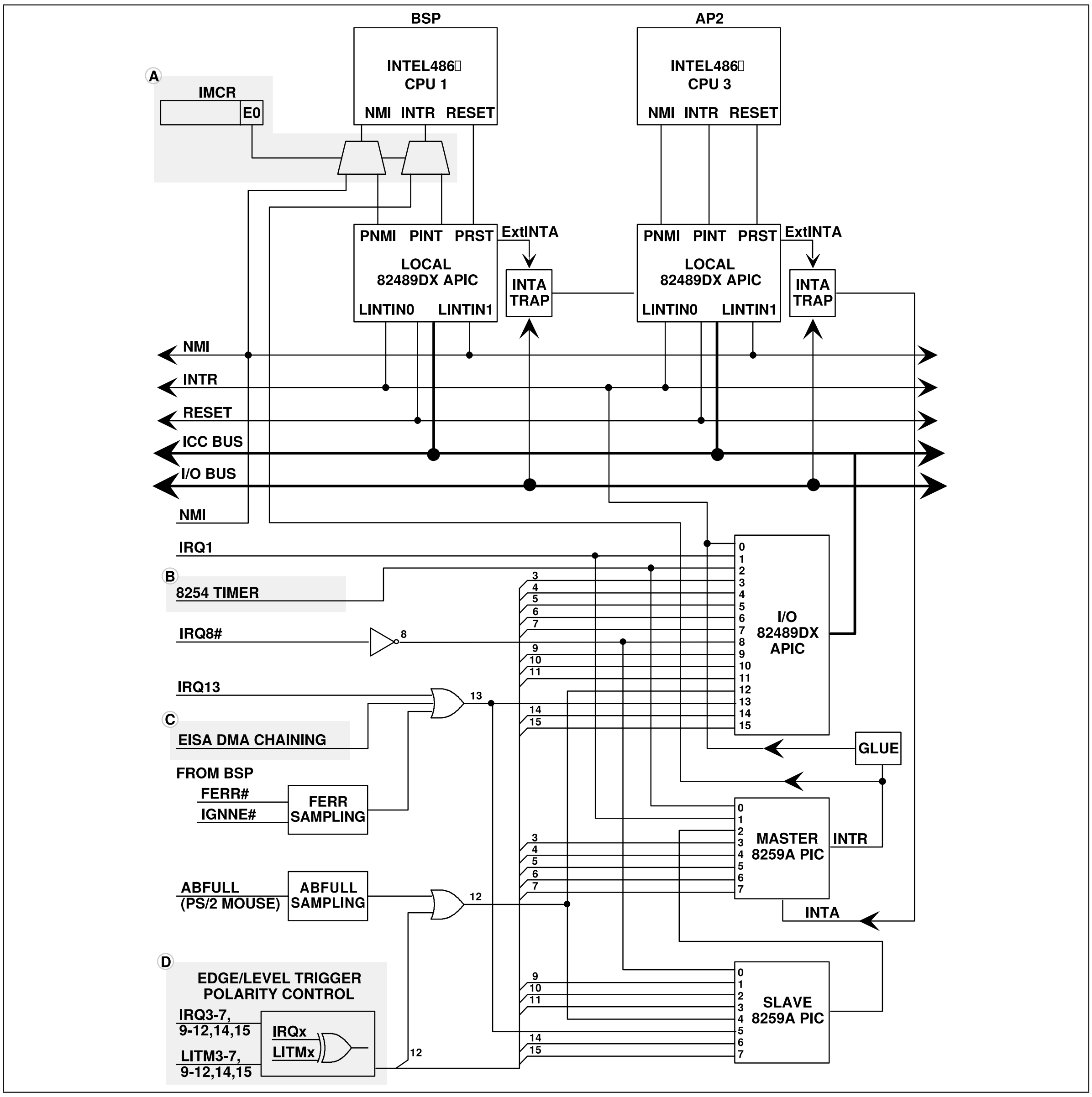


Figure 2.3: The Discrete APIC Architecture [10, sec. 5.1].

5. When the interrupt is being handled by the OS, it sends an EOI to the local APIC[14], which clears the corresponding bit in the ISR. Note how the OS only interacts directly with the local APIC, not the I/O APIC.

To allow splitting the APIC architecture into local APICs of multiple CPU cores and I/O APICs, the individual components communicate over a bus. In the Intel "P6" and "Pentium" processors a dedicated *APIC bus* is used, but since the Intel "Pentium 4" and "Xeon" processors the *system bus* is used instead (see Figure 2.5)[15]. Using the system bus is considered the default in this document.
Because the order in which external devices are connected to the interrupt controller

[14] In case of edge-triggered interrupts. For level-triggered interrupts see Section 3.2.2.

[15] The Intel "Core" microarchitecture is based on the P6 architecture and replaces the Pentium 4's "NetBurst" architecture, but still uses the system bus instead of a discrete bus [2, sec. 3.11.2].

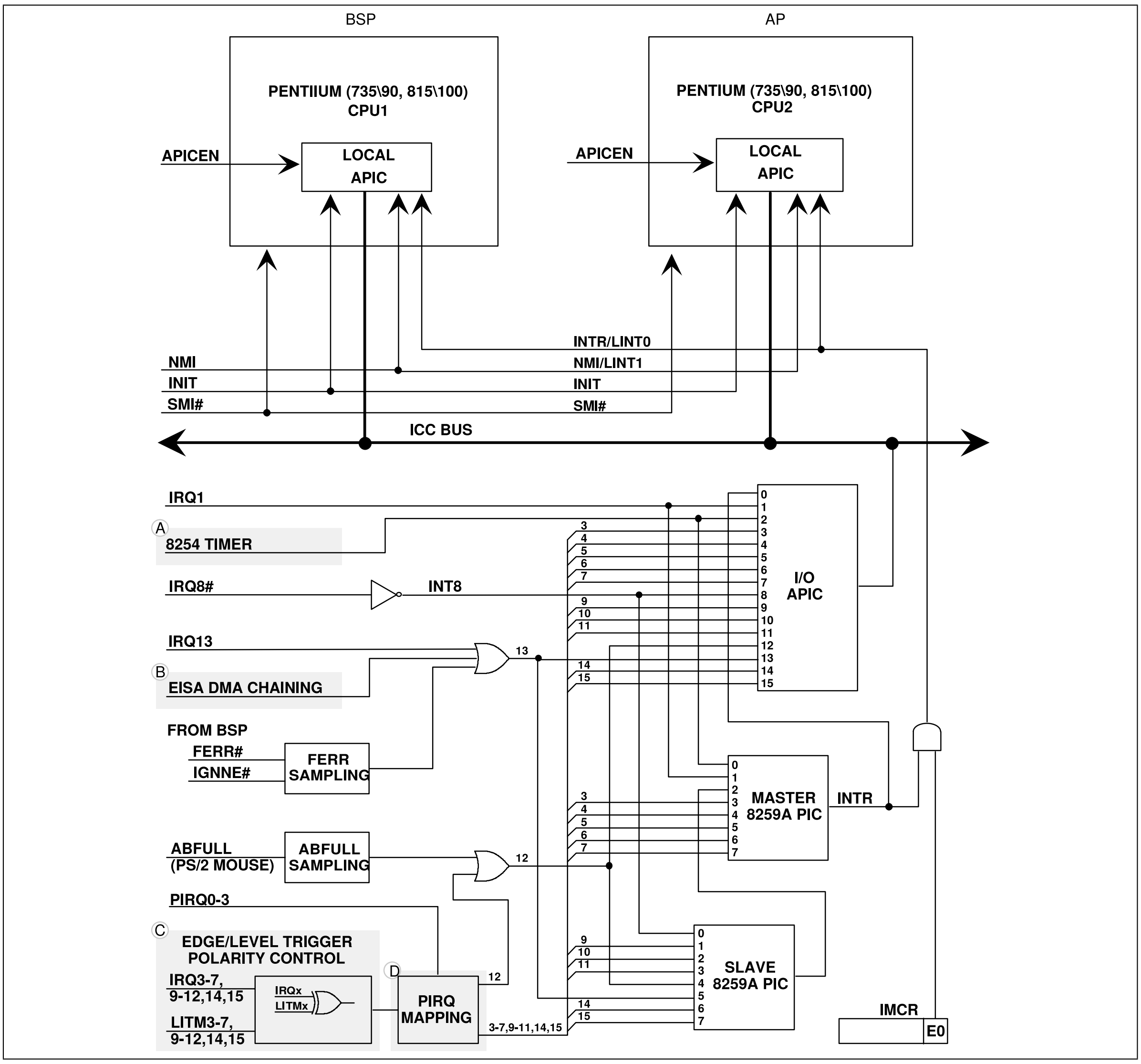


Figure 2.4: The Integrated APIC Architecture [10, sec. 5.2].

is flexible in the APIC architecture, but fixed in the PC/AT PIC architecture, there can be deviations in device-to-IRQ mappings between the PIC and APIC interrupt controllers. To handle this, ACPI provides information about how the PC/AT compatible interrupts (IRQ0 to IRQ15) map to ACPI's GSIs in the MADT, which has to be taken into account by the OS when operating with an APIC. The mapping information for a single interrupt will be called **interrupt override**.

There have been different revisions on the APIC architecture after the original, discrete APIC, notably the **xApic** and **x2Apic** architectures. This thesis is mostly concerned about the older xApic architecture[16]. A notable difference between xApic and x2Apic architectures is the register access: xApic uses 32 bit MMIO based register access, while x2Apic uses an MSR based register access, which allows atomic register access to the 64 bit wide APIC control MSRs [2, sec. 3.11.12].

[16] This does not present a compatibility issue, as the x2Apic architecture is backwards compatible to the xApic architecture.

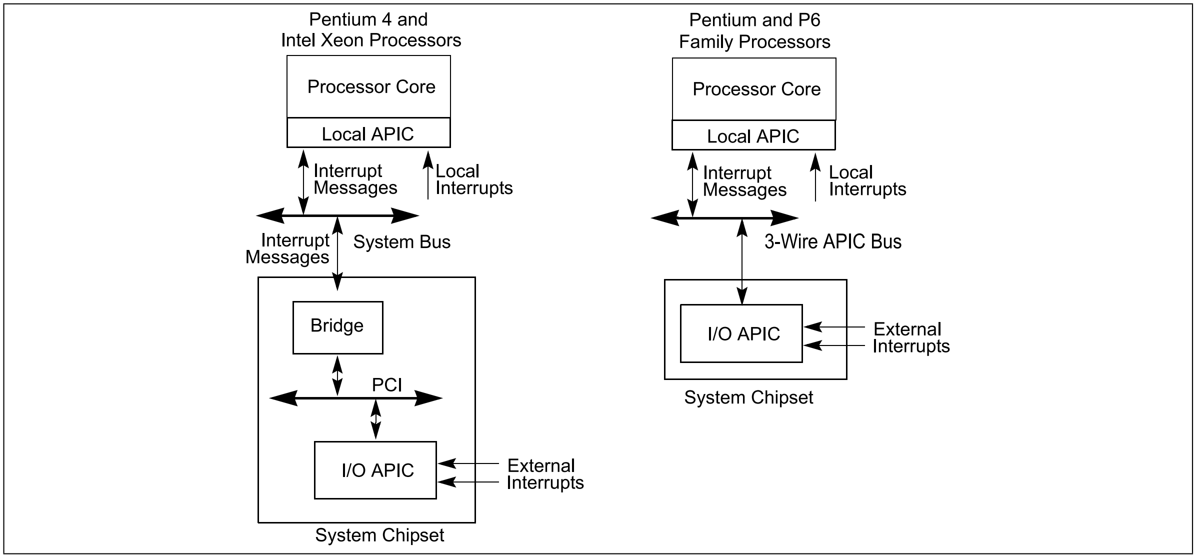


Figure 2.5: System Bus vs APIC Bus [2, sec. 3.11.1].

2.5 PC/AT Compatibility

For compatibility with older computer systems, two cascaded PICs are usually present in current computer systems (see Figure 2.3/Figure 2.4). The local APIC can be initialized to operate with these PICs instead of the I/O APIC, this is called **virtual wire mode** [10, sec. 3.6.2.2]. It is possible to operate with both the PIC and I/O APIC as controllers for external interrupts in *mixed mode* [10, sec. 3.6.2.3], but this is very uncommon. Because the presence of a local APIC usually guarantees the presence of an I/O APIC, this thesis only focuses on interrupt handling with either two PICs with a single processor (**PIC mode** [10, sec. 3.6.2.1]), in case no APIC is available, or the full APIC architecture (local APIC and I/O APIC in **symmetric I/O mode** [10, sec. 3.6.2.3]). Hardware following the MultiProcessor Specification must either implement PIC mode or virtual wire mode for PC/AT compatibility.

To switch from PIC mode (if implemented) to symmetric I/O mode, the **IMCR (interrupt mode control register)**, a special register which controls the physical connection of the PIC to the CPU, has to be configured. In the original discrete APIC architecture, the IMCR can choose if either the PIC or local APIC is connected to the CPU (see Figure 2.3), since the xApic architecture the IMCR only connects or disconnects the PIC to the local APIC's LINT0 INTI (see Figure 2.4). When the IMCR is set to connect the PIC, and the xApic "global enable/disable" flag is unset (see Section 3.1.1), the system is functionally equivalent to an IA-32 CPU without an integrated local APIC [2, sec. 3.11.4.3]. Modern systems (and QEMU) do not support PIC Mode and thus do not have the IMCR. PC/AT compatibility is usually achieved in these systems by connecting two PICs to the first external interrupt input of the I/O APIC (see Figure 2.4).

Specifics on handling PC/AT compatible external interrupts follow in Section 3.2.1.

Chapter 3

Using the APIC

This chapter will detail the general requirements to integrate APIC support into an operating system. Details on the implementation in hhuOS are found in Chapter 4.

3.1 Local APIC

The local APIC block diagram (see Figure 3.1) shows a number of registers that are required for APIC usage[1]:

- **LVT (local vector table)**: Used to configure how local interrupts are handled.
- **SVR (spurious interrupt vector register)**: Contains the software-enable flag and the spurious interrupt vector number.
- **TPR (task-priority register)**: Decides the order in which interrupts are handled and a possible interrupt priority threshold, to ignore low priority interrupts.
- **ICR (interrupt command register)**: Controls the sending of IPIs for starting up APs in SMP enabled systems.
- **APR (arbitration priority register)**: Controls the priority in the APIC arbitration mechanism.

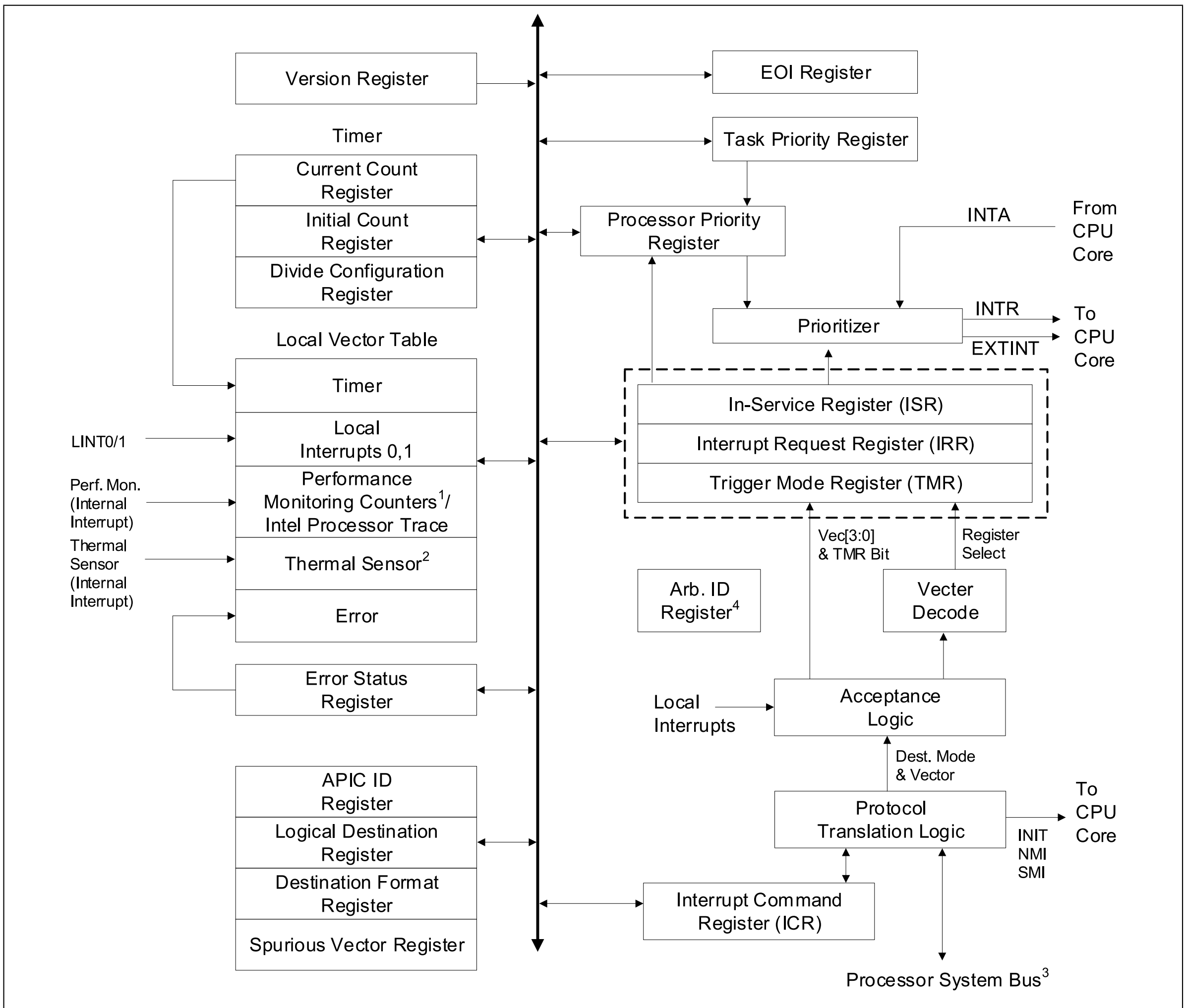
There are multiple registers associated with the LVT, those belong to the different local interrupts the local APIC can handle. Local interrupts this implementation is concerned about are listed below:

- *LINT1*: The local APIC's NMI source.
- *Timer*: Periodic interrupt triggered by the APIC timer.
- *Error*: Triggered by errors in the APIC system (e.g. invalid vector numbers or corrupted messages in internal APIC communication).

The LINT0 interrupt is unlisted, because it is mainly important for virtual wire mode (it can be triggered by external interrupts from the PIC). The performance and thermal monitoring interrupts also remain unused in this implementation.
Besides the local APIC's own registers, the IMCR and IA32_APIC_BASE MSR also require initialization (described in Section 3.1.1).
After system power-up, the local APIC is in the following state [2, sec. 3.11.4.7]:

- IRR, ISR and TPR are reset to `0x00000000`.
- The LVT is reset to `0x00000000`, except for the masking bits (all local interrupts are masked on power-up).
- The SVR is reset to `0x000000FF`.
- The APIC is in xApic mode, even if x2Apic support is present.

[1] Registers like the "Trigger Mode Register", "Logical Destination Register" or "Destination Format Register" remain unused in this thesis.



1. Introduced in P6 family processors.

2. Introduced in the Pentium 4 and Intel Xeon processors.

3. Three-wire APIC bus in P6 family and Pentium processors.

4. Not implemented in Pentium 4 and Intel Xeon processors.

Figure 3.1: The Local APIC Block Diagram [2, sec. 3.11.4.1].

- Only the BSP is enabled, other APs have to be enabled by the BSP's local APIC.

The initialization sequence consists of these steps, all performed before interrupt handling is enabled (see Figure A.1 for the full initialization sequence including all components):

1. Enable symmetric I/O mode.
2. Allocate memory for MMIO register access.
3. Initialize the LVT and NMI.
4. Initialize the SVR.
5. Initialize the TPR.

6. Initialize the APIC timer (if used) and error handler.
7. Synchronize the arbitration IDs.
8. Enable interrupts.

3.1.1 Enabling the Local APIC

Because the system boots in a PC/AT compatible mode (PIC mode or virtual wire mode), `0x01` should be written to the IMCR [10, sec. 3.6.2.1] (in case it exists) to disconnect the PIC from the local APIC's LINT0 INTI (see Figure 2.4). In case the IMCR is not available, all 16 PIC interrupts should be masked.
To set the operating mode, it is first determined which modes are supported by executing the `cpuid` [4] instruction with `eax=1`: If bit 9 of the `edx` register is set, xApic mode is supported, if bit 21 of the `ecx` register is set, x2Apic mode is supported [6, sec. 5.1.2]. If xApic mode is supported (if the local APIC is an integrated APIC), it will be in that mode already. The "global enable/disable" ("EN") bit in the IA32_APIC_BASE MSR (see Figure 3.2/Table B.14) allows disabling xApic mode globally[2]. Enabling x2Apic mode is done by setting the "EXTD" bit (see Figure 3.2/Table B.14) of the IA32_APIC_BASE MSR [2, sec. 3.11.4.3].

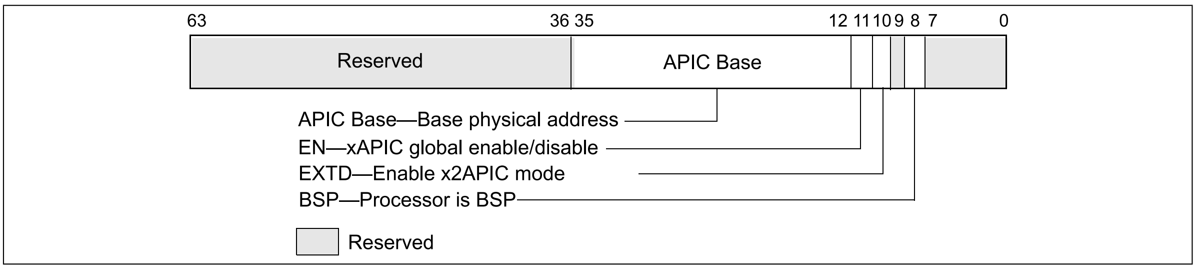


Figure 3.2: The IA32_APIC_BASE MSR [2, sec. 3.11.4.4].

Because QEMU does not support x2Apic mode via its TCG[3], this implementation only uses xApic mode.
Besides the "global enable/disable" flag, the APIC can also be enabled/disabled by using the "software enable/disable" flag in the SVR, see Section 3.1.4.

3.1.2 Accessing Local APIC Registers in xApic Mode

Accessing registers in xApic mode is done via MMIO and requires a single page (4 kB) of memory, mapped to the "APIC Base" address, which can be obtained by reading the IA32_APIC_BASE MSR (see Figure 3.2/Table B.14) or parsing the MADT (see Table B.21). The IA-32 manual specifies a caching strategy of "strong uncachable"[4] [2, sec. 3.11.4.1] for this region.

[2] If the system uses the discrete APIC bus, xApic mode cannot be re-enabled without a system reset [2, sec. 3.11.4.3].

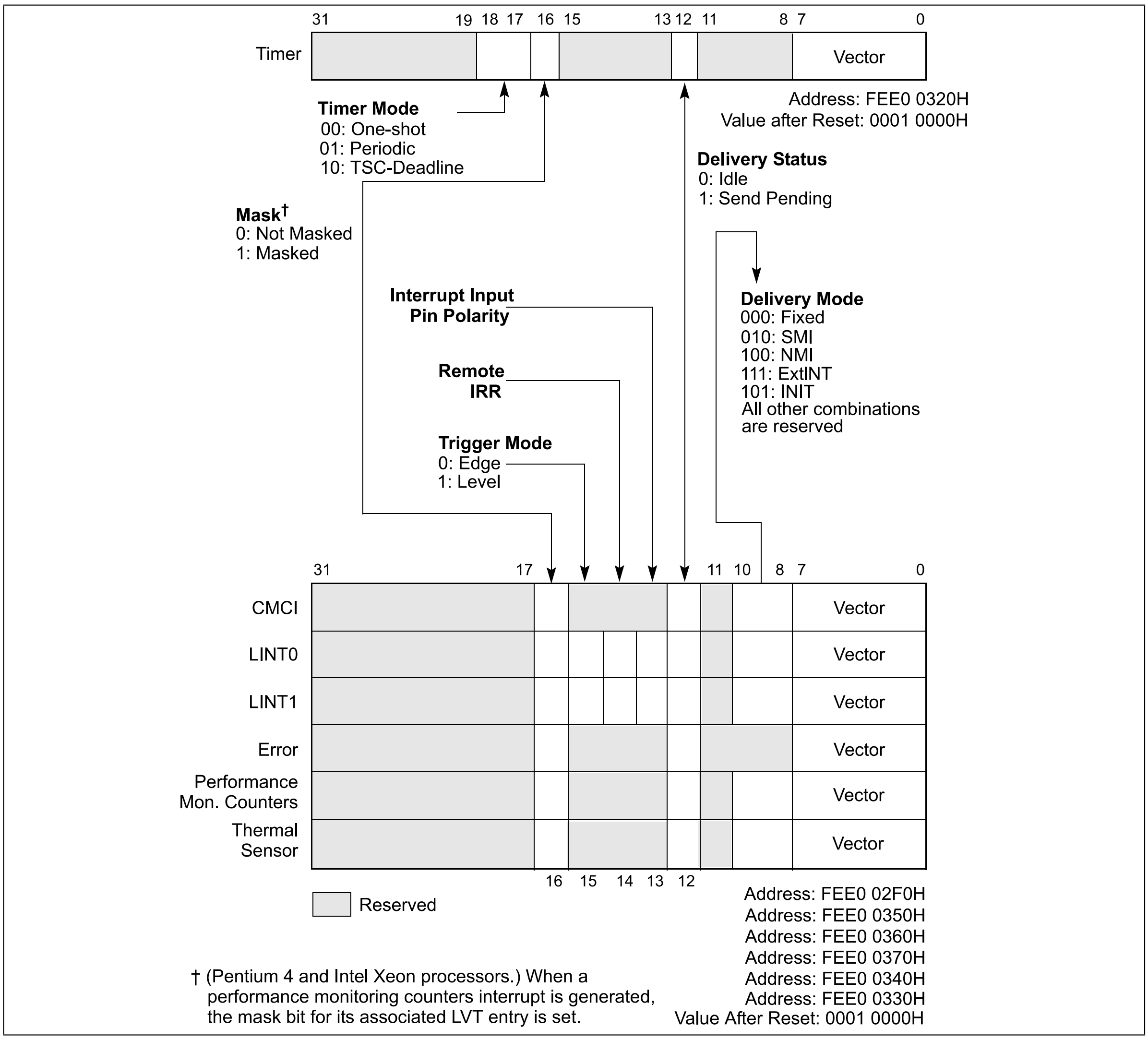
[3] QEMU Tiny Code Generator transforms target instructions to instructions for the host CPU.

[4] See IA-32 manual for information on this caching strategy [2, sec. 3.12.3].

The address offsets (from the base address) for the local APIC registers are listed in the IA-32 manual [2, sec. 3.11.4.1] and in Table B.1.

3.1.3 Handling Local Interrupts

To configure how the local APIC handles the different local interrupts, the LVT registers are written (see Figure 3.3).



† (Pentium 4 and Intel Xeon processors.) When a performance monitoring counters interrupt is generated, the mask bit for its associated LVT entry is set.

Figure 3.3: The Local Vector Table [2, sec. 3.11.5.1].

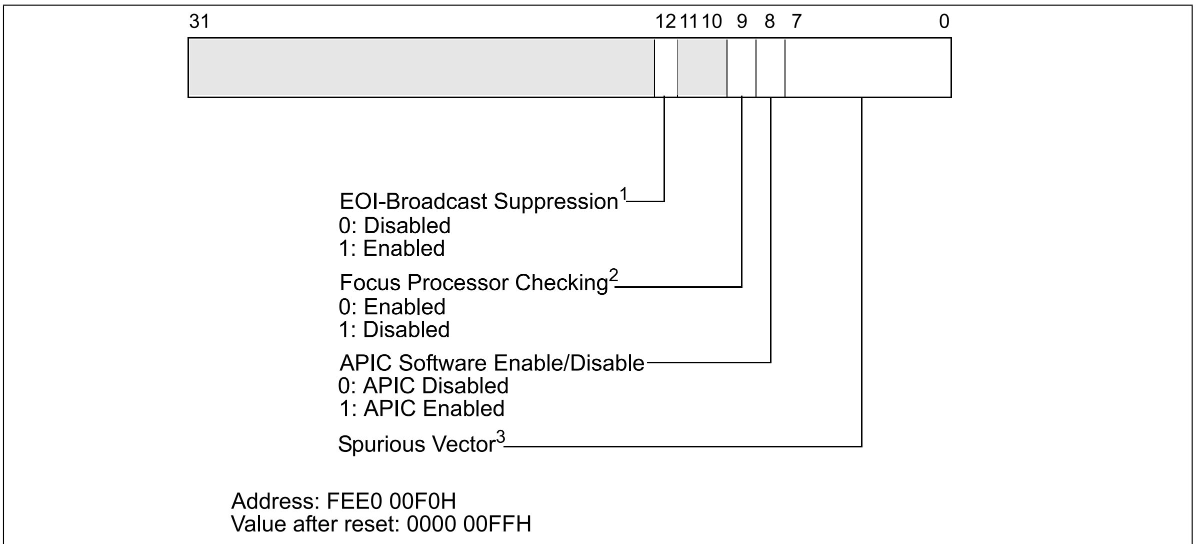
Local interrupts can be configured to use different delivery modes (excerpt) [2, sec. 3.11.5.1]:

- *Fixed*: Simply delivers the interrupt vector stored in the LVT register.
- *NMI*: Configures this interrupt as non-maskable, this ignores the stored vector number.
- *ExtINT*: The interrupt is treated as an external interrupt (instead of local interrupt), used e.g. in virtual wire mode for LINT0.

Initially, all local interrupts are initialized to PC/AT defaults: Masked, edge-triggered, active-high and with *fixed* delivery mode. Most importantly, the vector fields (bits 0:7 of each LVT register) are set to their corresponding interrupt vector.
After default-initializing the local interrupts, LINT1 has to be configured separately (see Table B.11), because it is the local APIC's NMI source[5]. The delivery mode is set to *NMI* and the interrupt vector to `0x00`. This information is also provided by ACPI in the MADT (see Table B.28). Other local interrupts (APIC timer and the error interrupt) will be configured later (see Section 3.1.6 and Section 3.1.7).

3.1.4 Allowing Interrupt Processing

To complete a minimal local APIC initialization, the "software enable/disable" flag and the spurious interrupt vector (see Figure 3.4/Table B.6), are set. It makes sense to choose `0xFF` for the spurious interrupt vector, as on P6 and Pentium processors, the lowest four bits must always be set (see Figure 3.4).



Address: FEE0 00F0H
Value after reset: 0000 00FFH

1. Not supported on all processors. See bit 24 of Local APIC Version Register.
2. Not supported in Pentium 4 and Intel Xeon processors.
3. For the P6 family and Pentium processors, bits 0 through 3 are always 1.

Figure 3.4: The Local APIC SVR Register [2, sec. 3.11.9].

Because the APIC's spurious interrupt has a dedicated interrupt vector (unlike the PIC's spurious interrupt), it can be ignored easily by registering a stub interrupt handler for the appropriate vector.
The final step to initialize the BSP's local APIC is to allow the local APIC to receive interrupts of all priorities. This is done by writing `0x00` to the TPR [2, sec. 3.11.8.3] (see Table B.4). By configuring the TPRs of different local APICs to different priori-

[5] In older specifications [10], LINT1 is defined as NMI source (see Figure 2.4). It is possible that this changed with newer architectures, so for increased compatibility this implementation configures the local APIC NMI source as reported by ACPI. It is also possible that ACPI reports information on the NMI source just for future-proofing.

ties or priority classes, distribution of external interrupts to CPUs can be controlled, but this is not used in this thesis.

3.1.5 Local Interrupt EOI

To notify the local APIC that a local interrupt is being handled, its EOI register (see Table B.5) has to be written. Not all local interrupts require EOIs: NMI, SMI, INIT, ExtINT, SIPI, or INIT-Deassert interrupts are excluded [2, sec. 3.11.8.5].
EOIs for external interrupts are also handled by the local APIC, further described in Section 3.2.2.

3.1.6 APIC Timer

The APIC timer is integrated into the local APIC, so it requires initialization of the latter. Like the PIT, the APIC timer can generate periodic interrupts in a specified interval by using a counter, that is initialized with a starting value depending on the desired interval. Because the APIC timer does not tick with a fixed frequency, but at bus frequency, the initial counter has to be determined at runtime by using an external time source. In addition to the counter register, the APIC timer interval is influenced by a divider: Instead of decrementing the counter at every bus clock, it will be decremented every n-th bus clock, where n is the divider. This is useful to allow for long intervals (with decreased precision), that would require a larger counter register otherwise.
The APIC timer supports three different timer modes, that can be set in the timer's LVT register (see Figure 3.3):

1. *Oneshot*: Trigger exactly one interrupt when the counter reaches zero.
2. *Periodic*: Trigger an interrupt each time the counter reaches zero, on zero the counter reloads its initial value.
3. *TSC-Deadline*: Trigger exactly one interrupt at an absolute time.

This implementation uses the APIC timer in periodic mode, to trigger the scheduler preemption. Initialization requires the following steps (order recommended on OSDev [11]):

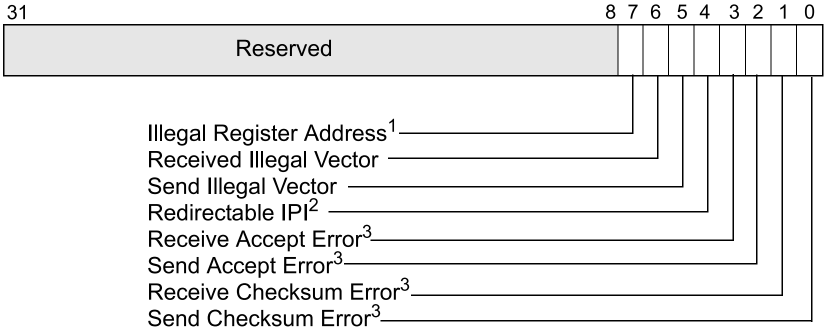
1. Measure the timer frequency with an external time source.
2. Configuration of the timer's divider register (see Table B.13).
3. Setting the timer mode to periodic (see Table B.9).
4. Initializing the counter register (see Table B.12), depending on the measured timer frequency and the desired interval. Setting the initial counter activates the timer.

In this implementation, the APIC timer is calibrated by counting the amount of ticks in one millisecond using oneshot mode. The measured amount of timer ticks can then be used to calculate the required counter for an arbitrary millisecond interval, although very large intervals could require the use of a larger divider, while very small intervals (in micro- or nanosecond scale) could require the opposite, to provide the necessary precision. For this approach it is important that the timer is initialized with the same divider that was used during calibration.

To use the timer, an interrupt handler has to be registered to its interrupt vector.

3.1.7 APIC Error Interrupt

Errors can occur e.g. when the local APIC receives an invalid vector number, or an APIC message gets corrupted on the system bus. To handle these cases, the local APIC provides the local error interrupt, whose interrupt handler can read the error status from the local APIC's **ESR (error status register)** (see Figure 3.5/Table B.7) and take appropriate action.



Address: FEE0 0280H
Value after reset: 0H

NOTES:

1. Used only by Intel Core, Pentium 4, Intel Xeon, and P6 family processors; reserved on the Pentium processor.
2. Used only by some Intel Core and Intel Xeon processors; reserved on other processors.
3. Used only by the P6 family and Pentium processors; reserved on Intel Core, Pentium 4 and Intel Xeon processors.

Figure 3.5: Error Status Register [2, sec. 3.11.5.3].

The ESR is a "write/read" register: Before reading a value from the ESR, it has to be written, which updates the ESR's contents to the error status since the last write. Writing the ESR also arms the local error interrupt, which does not happen automatically [2, sec. 3.11.5.3].

Enabling the local error interrupt is now as simple as enabling it in the local APIC's LVT and registering an interrupt handler for the appropriate vector.

3.2 I/O APIC

To fully replace the PIC and handle external interrupts using the APIC, the I/O APIC has to be initialized by setting its **REDTBL (redirection table)** registers [12, sec. 9.5.8] (see Table B.20). Like the local APIC's LVT, the REDTBL allows configuration of interrupt vectors, masking bits, interrupt delivery modes, pin polarities and trigger modes (see Section 3.1.3).

Additionally, for external interrupts a destination and destination mode can be specified. This is required because the I/O APIC is able to forward external interrupts to different local APICs over the system bus (see Figure 2.4). SMP systems use this mechanism to distribute external interrupts to different CPU cores for performance benefits. Because this implementation's focus is not on SMP, all external interrupts are default initialized to *physical* destination mode[6] [2, sec. 3.11.6.2.1] and are sent to the BSP for servicing, by using the BSP's local APIC ID as the destination. The other fields are set to ISA bus defaults[7], with *fixed* delivery mode, masked, and the corresponding interrupt vector.

The I/O APIC does not have to be enabled explicitly: If the local APIC is enabled and the REDTBL is initialized correctly, external interrupts will be redirected to the local APIC and handled by the CPU.

Unlike the local APIC's registers, the REDTBL registers are accessed indirectly: Two registers, the "Index" and "Data" register [12, sec. 9.5.1], are mapped to the main memory and can be used analogous to the local APIC's registers. The MMIO base address can be parsed from the MADT (see Table B.24). Writing an offset to the index register exposes an indirectly accessible I/O APIC register through the data register. This indirect addressing scheme is useful, because the number of external interrupts an I/O APIC supports, and in turn the number of REDTBL registers, can vary[8].

It is possible that one or multiple of the I/O APIC's interrupt inputs act as an NMI source. If this is the case is reported in the MADT (see Table B.27), so when necessary, the corresponding REDTBL entries are initialized like the local APIC's NMI source (see Section 3.1.3), and using these interrupt inputs for external interrupts is forbidden.

3.2.1 Interrupt Overrides

In every PC/AT compatible system, external devices are hardwired to the PIC in the same order. Because this is not the case for the I/O APIC, the INTI used by each PC/AT compatible interrupt has to be determined by the OS at runtime, by using ACPI. ACPI provides "Interrupt Source Override" structures [5, sec. 5.2.8.3.1] inside

[6] The alternative is *logical* destination mode, which allows addressing individual or clusters of local APIC's in a larger volume of processors [2, sec. 3.11.6.2.2].

[7] Edge-triggered, active-high.

[8] Intel's consumer **ICHs (intel input/output controller hubs)** always support a fixed amount of 24 external interrupts though [12, sec. 9.5.7].

the MADT (see Table B.25) for each PC/AT compatible interrupt that is mapped differently to the I/O APIC than to the PIC. In addition to the interrupt input mapping, these structures also allow customizing the pin polarity and trigger mode of PC/AT compatible interrupts.

This information does not only apply to the REDTBL initialization, but it has to be taken into account every time an action is performed on a PC/AT compatible interrupt, like masking or unmasking: If `IRQ0` (PIT) should be unmasked, it has to be determined what GSI (or in other words, I/O APIC interrupt input) it belongs to. In many systems `IRQ0` is mapped to `GSI2`, because the PC/AT compatible PICs are connected to `GSI0` (see Figure 2.4). Thus, to allow the PIT interrupt in those systems, the REDTBL entry belonging to `GSI2` instead of `GSI0` has to be written.

3.2.2 External Interrupt EOI

Notifying the I/O APIC that an external interrupt has been handled differs depending on the interrupt trigger mode: Edge-triggered external interrupts are completed by writing the local APIC's EOI register (see Section 3.1.5)[9]. Level-triggered interrupts are treated separately: Upon registering a level-triggered external interrupt, the I/O APIC sets an internal "Remote IRR" bit in the corresponding REDTBL entry [12, sec. 9.5.8] (see Table B.20). This is required to not redirect the level-triggered interrupt to the local APIC multiple times, while the INTI is asserted.

There are three possible ways to signal completion of a level-triggered external interrupt to clear the remote IRR bit:

1. Using the local APIC's EOI broadcasting feature: If EOI broadcasting is enabled, writing the local APIC's EOI register also triggers EOIs for each I/O APIC (for the appropriate interrupt), which clears the remote IRR bit.

2. Sending a directed EOI to an I/O APIC: I/O APICs with versions greater than `0x20` include an I/O EOI register [12, sec. 9.5.5]. Writing the vector number of the handled interrupt to this register clears the remote IRR bit.

3. Simulating a directed EOI for I/O APICs with versions smaller than `0x20`: Temporarily masking and setting a completed interrupt as edge-triggered clears the remote IRR bit [13, io_apic.c].

Because the first option is the only one supported by all APIC versions, it is used in this implementation[10].

At this point, after initializing the local and I/O APIC for the BSP, the APIC system is fully usable. External interrupts now have to be enabled/disabled by writing the "masked" bit in these interrupts' REDTBL entries, interrupt handler completion is signaled by writing the local APIC's EOI register, and spurious interrupts are detected by using the local APIC's spurious interrupt vector.

[9] Because external interrupts are forwarded to the local APIC, the local APIC is responsible for tracking them in its IRR and ISR.

[10] Disabling EOI broadcasting is not supported by all local APICs [2, sec. 3.11.8.5].

3.2.3 Multiple I/O APICs

Most consumer hardware, for example all IA processors [2] and ICH hubs [12], only provide a single I/O APIC, although technically multiple I/O APICs are supported by the "MultiProcessor Specification" [10, sec. 3.6.8]. If ACPI reports multiple I/O APICs (by supplying multiple MADT I/O APIC structures, see Table B.24), the previously described initialization has to be performed for each I/O APIC individually. Additionally, the I/O APIC's ID, also reported by ACPI, has to be written to the corresponding I/O APIC's ID register (see Table B.18), because this register is always initialized to zero [12, sec. 9.5.6].

Using a variable number of I/O APICs requires determining the target I/O APIC for each operation that concerns a GSI, like masking or unmasking. For this reason, ACPI provides the "GSI Base" [5, sec. 5.2.8.2] for each available I/O APIC, the number of GSIs a single I/O APIC can handle can be determined by reading the I/O APIC's version register [12, sec. 9.5.7] (see Table B.19)[11].

For many I/O APICs, replacing the local APIC's EOI broadcasting with a directed external EOI could be useful, because EOIs are broadcast to each I/O APIC.

3.3 Symmetric Multiprocessing

Like single-core systems, SMP systems boot using only a single core, the BSP. By using the APIC's capabilities to send IPIs between cores, additional APs can be put into startup state and booted for system use.

To determine the number of usable processors, the MADT is parsed (see Table B.22). Note, that some processors may be reported as disabled, those may not be used by the OS (see Table B.23).

3.3.1 Inter-Processor Interrupts

Issuing IPIs works by writing the local APIC's ICR (see Figure 3.6/Table B.8). It allows specifying IPI type, destination (analogous to REDTBL destinations, see Section 3.2) and vector.

Depending on the APIC architecture, two different IPIs are required: The *INIT IPI* for systems using a discrete APIC, and the **SIPI (startup inter-processor interrupt)** for systems using the xApic or x2Apic architectures:

- The INIT IPI causes an AP to reset its state and start executing at the address specified at its system reset vector. If paired with a system warm-reset, the AP can be instructed to start executing the AP boot sequence by writing the appropriate address to the warm-reset vector [10, sec. B.4.1].
- Since the xApic architecture, the SIPI is used for AP startup: It causes the AP to start executing code in real mode, at a page specified in the IPIs interrupt

[11] This approach was previously used in this implementation, but removed for simplicity.

vector [10, sec. B.4.2]. By copying the AP boot routine to a page in lower physical memory, and sending the SIPI with the correct page number, an AP can be booted.

To wait until the IPI is sent, the ICR's delivery status bit can be polled.

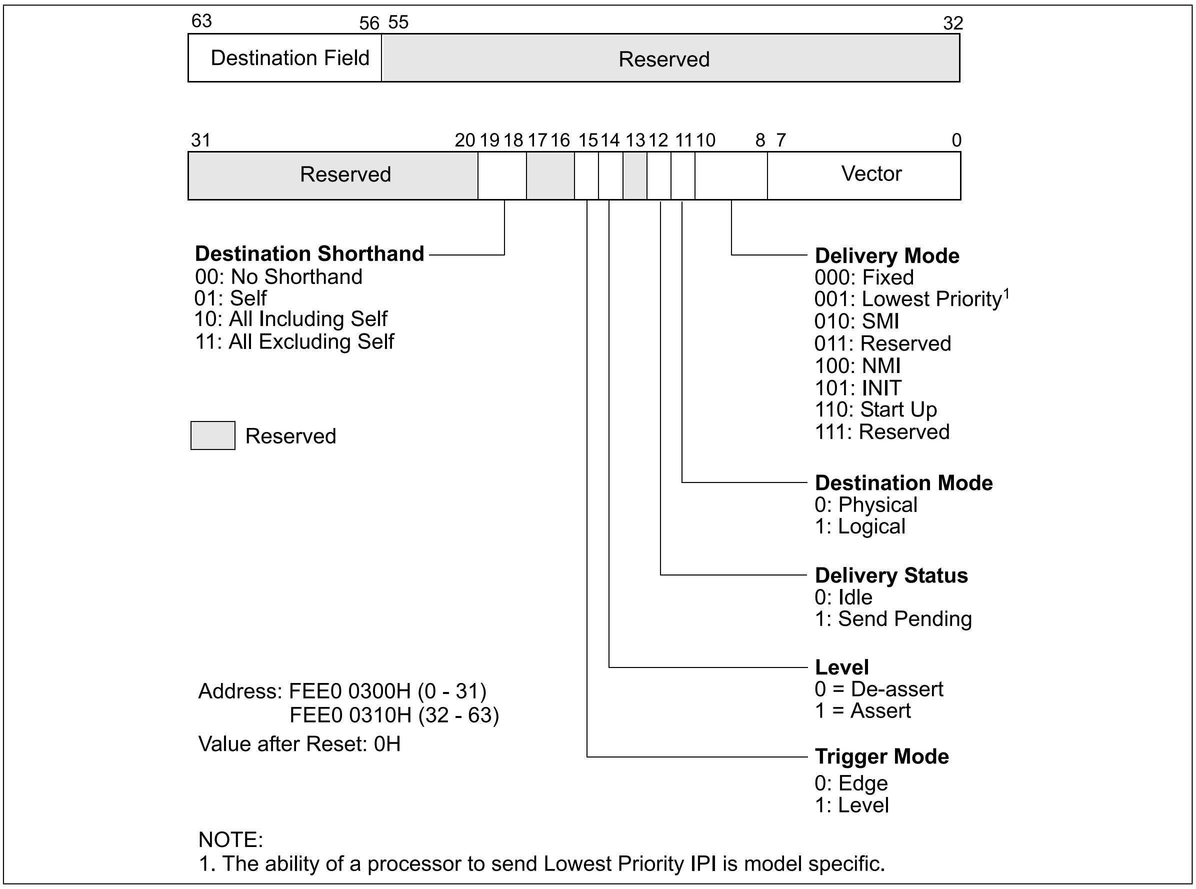


Figure 3.6: Interrupt Command Register [2, sec. 3.11.6.1].

3.3.2 Universal Startup Algorithm

SMP initialization is performed differently on various processors. Intel's "MultiProcessor Specification" defines a "Universal Startup Algorithm" for multiprocessor systems [10, sec. B.4], which can be used to boot SMP systems with either discrete APIC, xApic or x2Apic, as it issues both, INIT IPI and SIPI.

This algorithm has some prerequisites: It is required to copy the AP boot routine (detailed in Section 3.3.3) to lower memory, where the APs will start their execution. Also, the APs need pre-allocated stack memory to call the entry function, and in case of a discrete APIC that uses the INIT IPI, the system needs to be configured for a warm-reset (by writing `0xAH` to the CMOS shutdown status byte, located at `0xF` [10, sec. B.4]), because the INIT IPI does not support supplying the address where AP execution should begin, unlike the SIPI. The warm-reset vector (a 32 bit field, located at physical address `40:67` [10, sec. B.4]) needs to be set to the physical address the AP startup routine was copied to. Additionally, the entire AP startup procedure has to be performed with all sources of interrupts disabled, which offers a small challenge, since some timings need to be taken into account[12].

The usage of delays in the algorithm is quite specific, but the specification provides no further information on the importance of these timings or required precision. The algorithm allowed for successful startup of additional APs when tested in QEMU (with and without KVM) and on certain real hardware, although for different processors or emulators (like Bochs), different timings might be required [14, lapic.c].

After preparation, the universal startup algorithm is now performed as follows, for each AP sequentially (see Figure A.2 for the full SMP startup sequence):

1. Assert and de-assert the level-triggered INIT IPI.
2. Delay for 10 ms.
3. Send the SIPI.
4. Delay for 200 µs.
5. Send the SIPI again.
6. Delay for 200 µs again.
7. Wait until the AP has signaled boot completion, then continue to the next.

If the system uses a discrete APIC, the APs will reach the boot routine by starting execution at the location specified in the warm-reset vector, if the system uses the xApic or x2Apic architecture, the APs will reach the boot routine because its location was specified in the SIPI.

Signaling boot completion from the APs entry function can be done by using a global bitmap variable, where the n-th bit indicates the running state of the n-th processor. This variable does not have to be synchronized across APs, because the startup is performed sequentially.

3.3.3 Application Processor Boot Routine

After executing the “INIT-SIPI-SIPI” sequence, the targeted AP will start executing its boot routine in real mode. The general steps required are similar to those required when booting a single-core system, but since the BSP in SMP systems is already fully operational at this point, much can be recycled. The AP boot routine this implementation uses can be roughly described as follows:

1. Load a temporary **GDT (global descriptor table)**, used for switching to protected mode.

[12] This implementation uses the PIT’s mode 0 on channel 0 for timekeeping, see Section 4.3.5.

2. Enable protected mode by writing `cr0`.
3. Far jump to switch to protected mode and reload the code-segment register, set up the other segments manually.
4. Load the `cr3`, `cr0` and `cr4` values used by the BSP to enable paging (in that order).
5. Load the IDT used by the BSP.
6. Determine the AP's APIC ID by using CPUID.
7. Load the GDT and **TSS (task state segment)** prepared for this AP.
8. Load the stack prepared for this AP.
9. Call the (C++) AP entry function.

The APIC ID is used to determine which GDT and stack were prepared for a certain AP. It is necessary for each AP to have its own GDT, because each processor needs its own TSS for hardware context switching, e.g. when interrupt-based system calls are used on all CPUs.
Because it is relocated into lower physical memory, this code has to be position independent. For this reason, absolute physical addresses have to be used when jumping, loading the IDTR and GDTR, or referencing variables. Also, any variables required during boot have to be available after relocation, this can be achieved by locating them inside the "TEXT" section of the routine, so they stay with the rest of the instructions when copying. These variables have to be initialized during runtime, before the routine is copied.

3.3.4 Application Processor Post-Boot Routine

In the entry function, called at the end of the boot routine, the AP signals boot completion as described in Section 3.3.2 and initializes its local APIC by repeating the necessary steps from Section 3.1.3, Section 3.1.4, Section 3.1.6 and Section 3.1.7[13].
Because multiple local APICs are present and active in the system now, the possibility arises that a certain local APIC receives multiple messages from different local APICs at a similar time. To decide the order of handling these messages, an arbitration mechanism based on the local APIC's ID is used [2, sec. 3.11.7]. To make sure the arbitration priority matches the local APIC's ID, the ARPs can be synchronized by issuing an INIT-level-deassert IPI[14].

[13] MMIO memory does not have to be allocated again, as all local APICs use the same memory region in this implementation. Also, the initial value for the APIC timer's counter can be reused, if already calibrated.

[14] This is not supported on Pentium 4 and Xeon processors.

Chapter 4

Implementation

This chapter describes the implementation of the APIC interrupt controller for hhuOS and the source code produced during this thesis. Code examples are simplified and do not necessarily match the actual implementation completely, to allow for self-contained examples.

4.1 Design Decisions and Scope

The APIC interrupt architecture is split into multiple hardware components and tasks: The (potentially multiple) local APICs, the (usually single) I/O APIC and the APIC timer (part of each local APIC). Furthermore, the APIC system needs to interact with its memory mapped registers and the hhuOS ACPI subsystem, to gather information about the CPU topology and interrupt overrides. Also, the OS should be able to interact with the APIC system in a simple and easy manner, without needing to know all of its individual parts.
To keep the whole system structured and simple, the implementation is split into the following main components (see Figure 4.1):

- `LocalApic`: Provides functionality to interact with the local APIC (masking and unmasking, register access, etc.).
- `IoApic`: Provides functionality to interact with the I/O APIC (masking and unmasking, register access, etc.)
- `ApicTimer`: Provides functionality to calibrate the APIC timer and handle its interrupts.
- `Apic`: Condenses all the functionality above and exposes it to other parts of the OS.

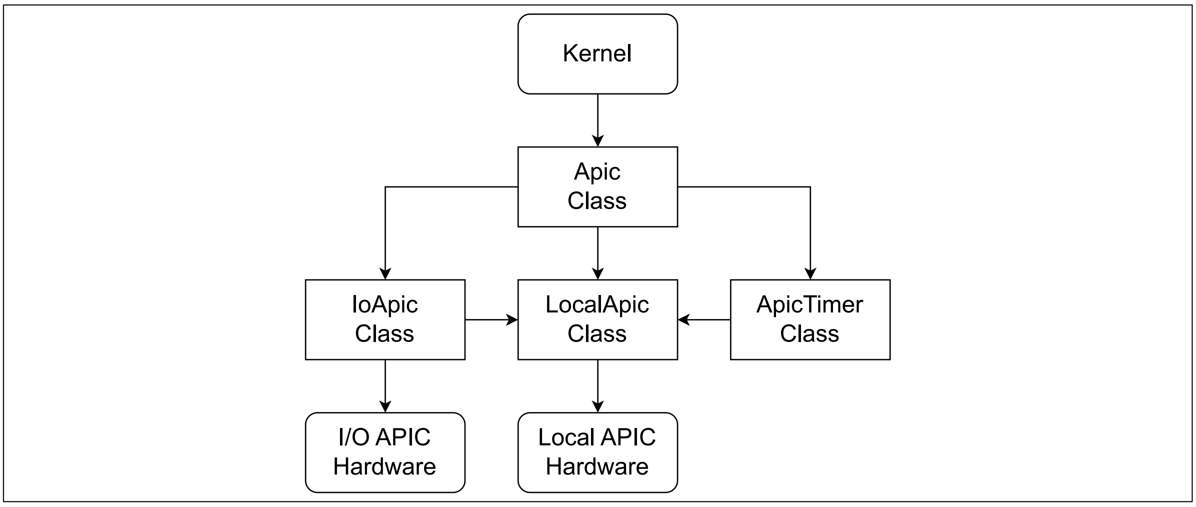


Figure 4.1: Caller Hierarchy of the Main Components.

This implementation is targeted to support systems with a single I/O APIC[1], because consumer hardware typically only uses a single one, and so does QEMU emulation. General information on implementing multiple I/O APIC support can be found in Section 3.2.3.

[1] Operation with more than one I/O APIC is described further in the "MultiProcessor Specification" [10, sec. 3.6.8].

With the introduction of ACPI's GSIs to the OS, new types are introduced to clearly differentiate different representations of interrupts and prevent unintentional conversions:

- `GlobalSystemInterrupt`: ACPI's interrupt abstraction, detached from the hardware interrupt lines.
- `InterruptRequest`: Represents an **ISA (industry standard architecture)** IRQ, allowing the OS to address interrupts by the name of the device that triggers it. When the APIC is used, interrupt overrides map IRQs to GSIs.
- `InterruptVector`: Represents an interrupt's vector number, as used by the `InterruptDispatcher`. The dispatcher maps interrupt vectors to interrupt handlers.

Both BIOS and UEFI are supported by hhuOS, but the hhuOS ACPI subsystem is currently only available with BIOS[2], so when hhuOS is booted using UEFI, APIC support cannot be enabled. Also, the APIC can handle MSIs, but they are not included in this implementation, as hhuOS currently does not utilize them.
SMP systems are partially supported: The APs are initialized, but only to a busy-looping state, as hhuOS currently is a single-core OS and lacks some required infrastructure. All interrupts are handled using the BSP.
A summary of features that are outside the scope of this thesis:

- Operation with a discrete APIC or x2Apic.
- Interrupts with logical destinations (flat/clustered) or custom priorities.
- Returning from APIC operation to PIC mode[3].
- Relocation of a local APIC's MMIO memory region[4].
- Distributing external interrupts to different APs in SMP enabled systems.
- Usage of the system's performance counter or monitoring interrupts.
- Meaningful treatment of APIC errors.
- Handling of MSIs.

To be able to easily extend an APIC implementation for single-core systems to SMP systems, some things are taken into account:

- SMP systems need to manage multiple `LocalApic` and `ApicTimer` instances. This is handled by the `Apic` class.

[2] State from 11/02/2023.

[3] This would be theoretically possible with single-core hardware, but probably useless.

[4] Relocation is possible by writing a new physical APIC base address to the IA32_APIC_BASE MSR [2, sec. 3.11.4.5].

- Initialization of the different components can no longer happen at the same “location”: The local APICs and APIC timers of additional APs have to be initialized by the APs themselves, because the BSP can not access an AP’s registers.
- APs are only allowed to access instances of APIC classes that belong to them.
- Interrupt handlers that get called on multiple APs may need to take the current processor into account (for example the APIC timer interrupt handler).
- Register access has to be synchronized, if it is performed in multiple operations on the same address space.

4.2 Integration into HhuOS

4.2.1 Interrupt Handling in HhuOS

In hhuOS, external interrupts are handled in two stages:

1. After an IRQ is sent by an interrupt controller, the CPU looks up the interrupt handler address in the IDT. In hhuOS, every IDT entry contains the address of the `dispatch` function, which is invoked with the vector number of the interrupt.
2. The `dispatch` function determines which interrupt handler will be called, based on the supplied vector number. In hhuOS, interrupt handlers are implemented through the `InterruptHandler` interface, that provides the `trigger` function, which contains the interrupt handling routine. To allow the `dispatch` function to find the desired interrupt handler, it has to be registered to a vector number by the OS beforehand. This process is handled by the `plugin` function of the interrupt handler interface, which uses the interrupt dispatcher’s `assign` function to register itself to the correct vector number. HhuOS supports assigning multiple interrupt handlers to a single interrupt vector and cascading interrupts.

To prevent the need of interacting with a specific interrupt controller implementation (e.g. `Pic` class) or the dispatcher, a system service (the `InterruptService`) is implemented to expose this functionality to other parts of the OS (it allows e.g. registering interrupt handlers or masking and unmasking interrupts).
Currently, hhuOS utilizes the PIT to manage the global system time, which in turn is used to trigger hhuOS’ preemptive round-robin scheduler (the `Scheduler` class). The PIT and other devices are initialized before the system entry point, in the `System::initializeSystem` function.
Devices that provide interrupts implement the `InterruptHandler` interface:

4.1 The InterruptHandler Interface (InterruptHandler.h) [1] C++

```cpp
// Excerpt from the "InterruptHandler" interface
class InterruptHandler {
public:
    virtual void plugin() = 0;                             // Register the handler
    virtual void trigger(const InterruptFrame &frame) = 0; // Handle an interrupt
};
```

These interrupt handlers have to be registered to the appropriate interrupt vector:

4.2 Assigning an Interrupt Handler (InterruptDispatcher.cpp) [1] C++

```cpp
// Excerpt from the "assign" function
void InterruptDispatcher::assign(InterruptVector vec, InterruptHandler &handler) {
    if (handlers[vec] == nullptr) {
        // Make space for multiple possible interrupt handlers
        handlers[vec] = new Util::ArrayList<InterruptHandler *>;
    }

    handlers[vec]->add(&handler); // Register a handler to a vector
}
```

Multiple interrupt handlers can be registered to the same interrupt vector. For every interrupt that arrives at a CPU, the first-stage interrupt handler (registered in the IDT for every vector) is called, which in turn calls the device-specific handlers:

4.3 Triggering an Interrupt Handler (InterruptDispatcher.cpp) [1] C++

```cpp
// Excerpt from the "dispatch" function
void InterruptDispatcher::dispatch(InterruptVector vec) {
    interruptService.sendEndOfInterrupt(vec); // Signal interrupt servicing
    asm volatile("sti");                      // Allow cascaded interrupts

    auto *handlerList = handlers[vec];
    for (uint32_t i = 0; i < handlerList->size(); i++) {
        handlerList->get(i)->trigger(); // Call registered interrupt handlers
    }

    asm volatile("cli");
}
```

The PIT interrupt handler manages the system time and the scheduler:

4.4 The PIT Interrupt Handler (Pit.cpp) [1] C++

```cpp
// Excerpt from the "Pit" interrupt handler
void Pit::trigger() {
    time.addNanoseconds(timerInterval); // Increase system time

    // Trigger preemption
    if (time.toMilliseconds() % yieldInterval == 0) {
        System::getService<SchedulerService>().yield(); // Trigger preemption
    }
}
```

4.2.2 Necessary Modifications to HhuOS

To integrate the APIC implementation into hhuOS, some preexisting components of its interrupt infrastructure (see Section 4.2) have to be modified:

1. The `InterruptService` has to forward calls to the `Apic` class instead of the `Pic` class, depending on if the APIC system is enabled – to keep the option of running the OS on hardware that does not support the APIC (see Listing 4.5).

2. The `Pit` interrupt handler may no longer trigger the scheduler preemption if the APIC timer is enabled (see Listing 4.6).

3. The `System::initializeSystem` function needs to enable the APIC system if it is available (see Listing 4.7).

4. The existing devices using interrupts need to use the new `InterruptVector` and `InterruptRequest` enums.

4.5 Using the Correct Interrupt Controller (InterruptService.cpp) C++

```cpp
void InterruptService::allowHardwareInterrupt(InterruptRequest interrupt) {
    if (Apic::isEnabled()) {
        Apic::allow(interrupt);
    } else {
        Pic::allow(interrupt);
    }
}
```

4.6 Disabling PIT Preemption (Pit.cpp) C++

```cpp
void Pit::trigger() {
    time.addNanoseconds(timerInterval); // Increase system time

    // Don't use PIT for scheduling when the APIC timer is enabled
    if (Apic::isEnabled()) {
        return;
    }

    if (time.toMilliseconds() % yieldInterval == 0) {
        System::getService<SchedulerService>().yield(); // Trigger preemption
    }
}
```

4.7 Enabling the APIC System (System.cpp) C++

```cpp
// Excerpt from the "initializeSystem" function
void System::initializeSystem() {
    if (Apic::isSupported()) {
        Apic::enable();

        if (Apic::isSmpSupported()) {
            Apic::startupSmp();
        }
    }

    Cpu::enableInterrupts();
}
```

4.2.3 Public Interface of the APIC Subsystem

To interact with the APIC subsystem from the “outside”, the `Apic` class is used, which exposes the necessary functionality:

4.8 The APIC Public Interface (Apic.h) C++

```cpp
// Excerpt from the Apic class definition
class Apic {
public:
    // Enabling the APIC subsystem
    static bool      isSupported();
    static bool      isEnabled();
    static void      enable();

    // Enabling SMP
    static bool      isSmpSupported();
    static void      startupSmp();

    // Controlling the current core's local APIC
    static void      initializeCurrentLocalApic();
    static uint8_t   getCpuCount();
    static LocalApic &getCurrentLocalApic();
    static void      enableCurrentErrorHandler();

    // Controlling the current core's APIC timer
    static bool      isCurrentTimerRunning();
    static void      startCurrentTimer();
    static ApicTimer &getCurrentTimer();

    // Controlling the current core's interrupts
    static void      allow(InterruptRequest interruptRequest);
    static void      forbid(InterruptRequest interruptRequest);
    static bool      status(InterruptRequest interruptRequest);
    static void      sendEndOfInterrupt(InterruptVector vector);
    static bool      isLocalInterrupt(InterruptVector vector);
    static bool      isExternalInterrupt(InterruptVector vector);

    // Tracking interrupt statistics
    static void      mountVirtualFilesystemNodes();
    static void      countInterrupt(InterruptVector vector);
}
```

Enabling the APIC subsystem is done by the `enable` function (see Figure A.1).

4.3 Local APIC

4.3.1 Disabling PIC Mode

Setting the IMCR[5] using hhuOS' `IoPort` class:

4.9 Disabling PIC Mode (LocalApic.cpp) C++

```cpp
void LocalApic::disablePicMode() {
    IoPort registerSelectorPort = IoPort(0x22);
    IoPort registerDataPort = IoPort(0x23);

    registerSelectorPort.writeByte(0x70); // IMCR address is 0x70
    registerDataPort.writeByte(0x01);     // 0x01 disconnects PIC
}
```

[5] Writing the IMCR is detailed in the "MultiProcessor Specification" [10, sec. 3.6.2.1].

4.3.2 Accessing Local APIC Registers in xApic Mode

The xApic register access requires a single page of strong uncachable memory, but since this requires setting attributes in the “Page Attribute Table”[6], this implementation only uses hhuOS’ `mapIO` function, which maps a physical address to the kernel heap, with hhuOS’ “CACHE_DISABLE” flag set in the kernel page table:

4.10 Allocating the Local APIC’s MMIO Region (LocalApic.cpp) C++

```cpp
uint32_t allocateMMIORegion() {
    // The apicBaseAddress is obtained via ACPI
    auto &memoryService = System::getService<MemoryService>();
    void *virtAddressPtr = memoryService.mapIO(apicBaseAddress, Util::PAGESIZE);

    // Account for possible misalignment, as mapIO returns a page-aligned pointer
    uint32_t pageOffset = apicBaseAddress % Util::PAGESIZE;
    return reinterpret_cast<uint32_t>(virtAddressPtr) + pageOffset;
}
```

A register can now be written as follows:

4.11 Writing a Local APIC MMIO Register (LocalApic.cpp) C++

```cpp
void LocalApic::writeDoubleWord(uint32_t reg, uint32_t val) {
    // Use volatile to prevent compiletime caching and code elimination
    *reinterpret_cast<volatile uint32_t *>(virtAddress + reg) = val;
}
```

To reduce the usage of manual bit-shifting and -masking, this implementation provides structures for some commonly used registers, that implement conversion operators to the register format[7]:

4.12 The MSREntry Structure (LocalApicRegisters.h) C++

```cpp
struct BaseMSREntry {
    bool     isBSP;
    bool     isX2Apic;
    bool     isXApic;
    uint32_t baseField;

    explicit BaseMSREntry(uint64_t registerValue);
    explicit operator uint64_t() const;
};
```

4.13 The SVREntry Structure (LocalApicRegisters.h) C++

```cpp
struct SVREntry {
    InterruptVector vector;
    bool            isSWEnabled;
    bool            hasFocusProcessorChecking;
    bool            suppressEoiBroadcasting;

    explicit SVREntry(uint32_t registerValue);
    explicit operator uint32_t() const;
};
```

[6] See `https://docs.kernel.org/x86/pat.html` (visited on 12/02/2023).

[7] The usual approach of “adding” the different required flags together to obtain the desired register content was intentionally not used for the sake of expressiveness: This approach does not highlight which flags are available and which flags are *not* chosen.

4.14 The LVTEntry Structure (LocalApicRegisters.h) C++

```cpp
struct LVTEntry {
    InterruptVector vector;
    DeliveryMode    deliveryMode;
    DeliveryStatus  deliveryStatus;
    PinPolarity     pinPolarity;
    TriggerMode     triggerMode;
    bool            isMasked;
    TimerMode       timerMode;

    explicit LVTEntry(uint32_t registerValue);
    explicit operator uint32_t() const;
};
```

4.15 The ICREntry Structure (LocalApicRegisters.h) C++

```cpp
struct ICREntry {
    InterruptVector      vector;
    DeliveryMode         deliveryMode;
    DestinationMode      destinationMode;
    DeliveryStatus       deliveryStatus;
    Level                level;
    TriggerMode          triggerMode;
    DestinationShorthand destinationShorthand;
    uint8_t              destination;

    explicit ICREntry(uint64_t registerValue);
    explicit operator uint64_t() const;
};
```

These can be used in combination with some convenience functions:

4.16 Writing the IA32_APIC_BASE MSR (LocalApic.cpp) C++

```cpp
void LocalApic::writeBaseMSR(const MSREntry &msrEntry) {
    ModelSpecificRegister baseMSR = ModelSpecificRegister(0x1B);
    baseMSR.writeQuadWord(static_cast<uint64_t>(msrEntry)); // Atomic write
}
```

4.17 Writing the SVR (LocalApic.cpp) C++

```cpp
void LocalApic::writeSVR(const SVREntry &svrEntry) {
    writeDoubleWord(0xF0, static_cast<uint32_t>(svrEntry));
}
```

4.18 Writing the LVT (LocalApic.cpp) C++

```cpp
void LocalApic::writeLVT(uint32_t reg, const LVTEntry &lvtEntry) {
    writeDoubleWord(reg, static_cast<uint32_t>(lvtEntry));
}
```

4.19 Writing the ICR (LocalApic.cpp) C++

```cpp
void LocalApic::writeICR(const ICREntry &icrEntry) {
    auto val = static_cast<uint64_t>(icrEntry);
    icrLock.acquire(); // Synchronized in case of multiple APs
    writeDoubleWord(0x310, val >> 32);
    writeDoubleWord(0x300, val & 0xFFFFFFFF); // Writing the low DW sends the IPI
    icrLock.release();
}
```

4.3.3 Initializing the LVT

The interrupt vectors are set as defined by the `InterruptVector` enum. This implementation configures the LVT by using the `LVTEntry` struct (see Listing 4.14):

4.20 Configuring the Local Error Interrupt (LocalApic.cpp) C++

```cpp
// Excerpt from the initializeLVT function
void LocalApic::initializeLVT() {
    LVTEntry lvtEntry{};
    lvtEntry.deliveryMode = LVTEntry::DeliveryMode::FIXED;
    lvtEntry.pinPolarity  = LVTEntry::PinPolarity::HIGH;
    lvtEntry.triggerMode  = LVTEntry::TriggerMode::EDGE;
    lvtEntry.isMasked     = true;
    lvtEntry.vector       = InterruptVector::ERROR;
    writeLVT(ERROR, lvtEntry);
}
```

This process is repeated for each local interrupt, with variations for the NMI and timer interrupts.

4.3.4 Handling the Spurious Interrupt

This implementation sets the SVR by using the `SVREntry` struct (see Listing 4.13):

4.21 Setting the Spurious Interrupt Vector (LocalApic.cpp) C++

```cpp
// Excerpt from the initialize function
void LocalApic::initialize() {
    SVREntry svrEntry{};
    svrEntry.vector      = InterruptVector::SPURIOUS;
    svrEntry.isSWEnabled = true; // Keep the APIC software enabled
    writeSVR(svrEntry);
}
```

Because hhuOS uses a two-stage interrupt handling approach (described in Section 4.2.1), the spurious interrupt does not receive its own interrupt handler. Instead, it is ignored in the `dispatch` function, hhuOS' "first-stage" interrupt handler:

4.22 Checking for Spurious Interrupts (InterruptDispatcher.cpp) C++

```cpp
// Excerpt from the "checkSpuriousInterrupt" function
bool InterruptService::checkSpuriousInterrupt(InterruptVector interrupt) {
    return interrupt == InterruptVector::SPURIOUS;
}
```

4.23 Ignoring Spurious Interrupts (InterruptDispatcher.cpp) C++

```cpp
// Excerpt from the "dispatch" function
void InterruptDispatcher::dispatch(InterruptVector vec) {
    if (interruptService.checkSpuriousInterrupt(slot)) {
        spuriousCounterWrapper.inc();
        return; // Early return to skip the calling of any handlers
    }
}
```

4.3.5 Using the APIC Timer

The timer frequency is determined by counting the ticks in one millisecond, using the PIT as calibration source:

4.24 Calibrating the APIC Timer (ApicTimer.cpp) C++

```cpp
// Excerpt from the ApicTimer::calibrate function
uint32_t ApicTimer::calibrate() {
    // Start the timer with a large counter
    LocalApic::writeDoubleWord(LocalApic::TIMER_INITIAL, 0xFFFFFFFF);

    // Wait a little
    Pit::earlyDelay(10'000);

    // Calculate how often the timer ticked in one millisecond
    return (0xFFFFFFFF - LocalApic::readDoubleWord(LocalApic::TIMER_CURRENT)) / 10;
}
```

This calibration is performed before the interrupts get enabled, so it is not possible to use hhuOS' `TimeService` for the delay. Instead, the `PIT::earlyDelay` function is used, which configures the PIT for mode “Interrupt on Terminal Count” on channel zero and polls the channel's output status [7]:

4.25 Microsecond Delay without Interrupts (Pit.cpp) C++

```cpp
// Excerpt from the "earlyDelay" function
void Pit::earlyDelay(uint16_t us) {
    uint32_t counter = (static_cast<double>(BASE_FREQUENCY) / 1'000'000) * us;

    controlPort.writeByte(0b110000); // Channel 0, mode 0
    dataPort0.writeByte(static_cast<uint8_t>(counter & 0xFF));        // Low byte
    dataPort0.writeByte(static_cast<uint8_t>((counter >> 8) & 0xFF)); // High byte

    do {
        controlPort.writeByte(0b11100010);        // Readback channel 0
    } while (!(dataPort0.readByte() & (1 << 7))); // Bit 7 is the output pin state
}
```

Furthermore, the calibration is only performed once, even if multiple APIC timers are used. This has to be taken into account if multiple timers with different dividers are to be used.

To handle the APIC timer interrupt on multiple cores, n `ApicTimer` instances are registered to the appropriate interrupt vector, where n is the number of CPUs. Because this means, that each APIC timer interrupt on any CPU core triggers all n interrupt handlers, the handler has to determine if it belongs to the calling CPU. The handler belonging to the BSP's APIC timer also triggers the scheduler preemption (instead of the PIT):

4.26 Handling the APIC Timer Interrupt (ApicTimer.cpp) C++

```cpp
// Excerpt from the ApicTimer interrupt handler
void ApicTimer::trigger(const InterruptFrame &frame) {
    if (cpuId != LocalApic::getId()) {
        // Abort if the handler doesn't belong to the current CPU
        return;
    }

    // Increase the "core-local" time
    time.addNanoseconds(timerInterval * 1'000'000); // Interval is in milliseconds

    // Only the BSP may continue
    if (cpuId != 0) {
        return;
    }

    // BSP triggers preemption
    if (time.toMilliseconds() % yieldInterval == 0) {
        System::getService<SchedulerService>().yield();
    }
}
```

4.3.6 Handling Local APIC Errors

The error interrupt handler obtains the ESR's contents by writing to it first:

4.27 The Local APIC Error Interrupt Handler (ApicErrorHandler.cpp) C++

```cpp
void ApicErrorHandler::trigger() {
    // Writing the ESR updates its contents and arms the interrupt again
    LocalApic::writeDoubleWord(LocalApic::ESR, 0);
    uint32_t errors = LocalApic::readDoubleWord(LocalApic::ESR);

    log.error("APIC error on core [%d]: [0x%x]!", LocalApic::getId(), errors);
}
```

Because every CPU core can only access its own local APIC's registers, a single instance of this interrupt handler can be used for all AP's in the system.

4.4 I/O APIC

4.4.1 Accessing I/O APIC Registers

The I/O APIC's indirect register access requires two MMIO registers, that can be written similar to the local APIC's registers:

4.28 Writing an I/O APIC MMIO Register (IoApic.h) C++

```cpp
template<typename T>
void IoApic::writeMMIORegister(uint32_t reg, T val) {
    *reinterpret_cast<volatile T *>(mmioAddress + reg) = val;
}
```

Using the "Index" and "Data" registers to access the I/O APIC's indirect registers:

4.29 Writing an I/O APIC Indirect Register (IoApic.cpp) C++

```cpp
void IoApic::writeIndirectRegister(uint32_t reg, uint32_t val) {
    writeMMIORegister<uint8_t>(0x00, reg);  // Write the index register
    writeMMIORegister<uint32_t>(0x10, val); // Write the data register
}
```

To reduce the manual bit-shifting and -masking, the same approach is used as for the local APIC:

4.30 The REDTBLEntry Structure (IoApicRegisters.h) C++

```cpp
struct REDTBLEntry {
    InterruptVector vector;
    DeliveryMode    deliveryMode;
    DestinationMode destinationMode;
    DeliveryStatus  deliveryStatus;
    PinPolarity     pinPolarity;
    TriggerMode     triggerMode;
    bool            isMasked;
    uint8_t         destination;

    explicit REDTBLEntry(uint64_t registerValue);
    explicit operator uint64_t() const;
};
```

4.31 Writing the REDTBL (IoApic.cpp) C++

```cpp
void IoApic::writeREDTBL(GlobalSystemInterrupt gsi, const REDTBLEntry &redtbl) {
    auto val = static_cast<uint64_t>(redtbl);
    redtblLock.acquire(); // Synchronized in case of multiple APs
    writeIndirectRegister(0x10 + 2 * gsi, val & 0xFFFFFFFF); // Low DW
    writeIndirectRegister(0x10 + 2 * gsi + 1, val >> 32);    // High DW
    redtblLock.release();
}
```

4.4.2 Interrupt Overrides

This implementation represents an interrupt override using the following structure:

4.32 The External Interrupt Override Structure (IoApic.h) C++

```cpp
struct IrqOverride {
    InterruptRequest         source;
    GlobalSystemInterrupt    target;
    REDTBLEntry::PinPolarity polarity;
    REDTBLEntry::TriggerMode trigger;
};
```

During initialization, the overrides are used to set the correct interrupt vectors, polarities and trigger modes for each REDTBL entry. The `REDTBLEntry` struct (see Listing 4.30) is used to write the REDTBL:

4.33 Initializing the REDTBL (IoApic.cpp) C++

```cpp
// Excerpt from the IoAPic::initializeREDTBL function
void IoApic::initializeREDTBL() {
    // GSI2 belongs to the PIT in many systems
    auto gsi = GlobalSystemInterrupt(2);

    REDTBLEntry redtblEntry{};
    redtblEntry.deliveryMode    = REDTBLEntry::DeliveryMode::FIXED;
    redtblEntry.destinationMode = REDTBLEntry::DestinationMode::PHYSICAL;
    redtblEntry.isMasked        = true;
    redtblEntry.destination     = LocalApic::getId(); // Redirect to BSP

    IrqOverride *override = getOverride(gsi);
    if (override != nullptr) {
        // Apply any information provided by an interrupt override
        redtblEntry.vector      = override->source + 32;
        redtblEntry.pinPolarity = override->polarity;
        redtblEntry.triggerMode = override->trigger;
    } else {
        // Apply PC/AT compatible ISA bus defaults
        redtblEntry.vector      = gsi + 32;
        redtblEntry.pinPolarity = REDTBLEntry::PinPolarity::HIGH;
        redtblEntry.triggerMode = REDTBLEntry::TriggerMode::EDGE;
    }

    writeREDTBL(gsi, redtblEntry);
}
```

During regular operation, the overrides are used to determine the correct REDTBL entries for e.g. unmasking an interrupt:

4.34 Unmasking an IRQ (Apic.cpp) C++

```cpp
void Apic::allow(InterruptRequest interruptRequest) {
    IoApic::IrqOverride *override = IoApic::getOverride(interruptRequest);
    if (override == nullptr) {
        // If no override is specified, the IRQ is identity mapped to the GSI
        IoApic::allow(static_cast<GlobalSystemInterrupt>(interruptRequest));
    } else {
        // If an override is specified, lookup which GSI the IRQ is mapped to
        IoApic::allow(override->target);
    }
}
```

To convey this deviation between GSIs and PC/AT compatible IRQs very clearly, the public `Apic::allow` function accepts an `InterruptRequest` as an argument (that allows addressing the interrupt by name), while the internal `IoApic::allow` function only accepts a `GlobalSystemInterrupt` as argument:

4.35 Unmasking an IRQ Internally (IoApic.cpp) C++

```cpp
void IoApic::allow(GlobalSystemInterrupt gsi) {
    REDTBLEntry redtblEntry = readREDTBL(gsi);
    redtblEntry.isMasked = false;
    writeREDTBL(gsi, redtblEntry);
}
```

The internal `IoApic::allow` function is hidden (`private`) from the OS, and gets only called by the exposed `Apic::allow` function. This prevents accidentally setting the wrong REDTBL entry by not taking possible interrupt overrides into account.

The same principle is applied to the other operations concerning GSIs.

4.5 Symmetric Multiprocessing

An overview of the complete SMP startup process can be found in Figure A.2.

4.5.1 Issuing Inter-Processor Interrupts

To issue an IPI, the ICR is written by using the `ICREntry` struct (see Listing 4.15):

4.36 Issuing an INIT IPI (LocalApic.cpp) C++

```cpp
void LocalApic::issueINIT(uint8_t cpuId, ICREntry::Level level) {
    ICREntry icrEntry{};
    icrEntry.vector                = 0;
    icrEntry.deliveryMode          = ICREntry::DeliveryMode::INIT;
    icrEntry.destinationMode       = ICREntry::DestinationMode::PHYSICAL;
    icrEntry.level                 = level; // ASSERT or DEASSERT
    icrEntry.triggerMode           = ICREntry::TriggerMode::LEVEL;
    icrEntry.destinationShorthand = ICREntry::DestinationShorthand::NO;
    icrEntry.destination           = cpuId;
    writeICR(icrEntry); // Writing the ICR issues IPI
}
```

4.37 Issuing a SIPI (LocalApic.cpp) C++

```cpp
void LocalApic::issueSIPI(uint8_t cpuId, uint32_t page) {
    ICREntry icrEntry{};
    icrEntry.vector                = page;
    icrEntry.deliveryMode          = ICREntry::DeliveryMode::STARTUP;
    icrEntry.destinationMode       = ICREntry::DestinationMode::PHYSICAL;
    icrEntry.level                 = ICREntry::Level::ASSERT;
    icrEntry.triggerMode           = ICREntry::TriggerMode::EDGE;
    icrEntry.destinationShorthand = ICREntry::DestinationShorthand::NO;
    icrEntry.destination           = cpuId;
    writeICR(icrEntry); // Writing the ICR issues an IPI
}
```

It is important to note that the SIPI does not receive the startup address, but the page number. For this reason, the startup routine must be relocated starting at a page boundary. The `page` argument can be determined by removing the page offset from the linear address, which is done by right-shifting the address by the offset length (12 bit for 4 kB pages). For the physical memory location used in this implementation (`0x8000`) this yields page number `8`.

4.5.2 Preparing Symmetric Multiprocessing Startup

Before executing the "Universal Startup Algorithm", the boot code has to be relocated to physical lower memory. The memory region used for this copy has to be identity-mapped to the virtual kernel address space, so the effective addresses do not change after enabling paging in protected mode. To keep the required variables available to the startup code, these are located in the routines "TEXT" section:

4.38 The Boot Routine's Variables (smp_boot.asm) nasm

```nasm
; Export the variables to allow initialization by C++ code
global boot_ap_idtr
global boot_ap_cr0
global boot_ap_cr3
global boot_ap_cr4
global boot_ap_gdts
global boot_ap_stacks
global boot_ap_entry

[SECTION .text]
align 8
bits 16
boot_ap:
; [...] boot_ap 16-bit

; The variables initialized during runtime
align 8
boot_ap_idtr:
    dw 0x0
    dd 0x0
align 8
boot_ap_cr0:
    dd 0x0
align 8
boot_ap_cr3:
    dd 0x0
align 8
boot_ap_cr4:
    dd 0x0
align 8
boot_ap_gdts:
    dd 0x0
align 8
boot_ap_stacks:
    dd 0x0
align 8
boot_ap_entry:
    dd 0x0

bits 32
align 8
boot_ap_32:
; [...] boot_ap_32 32-bit
```

The variables are initialized during runtime:

4.39 Preparing the Boot Routine's Variables (ApicSmp.cpp) C++

```cpp
// Recycle values from the BSP
asm volatile("sidt %0" : "=m"(boot_ap_idtr));
asm volatile("mov %%cr0, %%eax;" : "=a"(boot_ap_cr0));
asm volatile("mov %%cr3, %%eax;" : "=a"(boot_ap_cr3));
asm volatile("mov %%cr4, %%eax;" : "=a"(boot_ap_cr4));

// Set the address of the pre-allocated GDTs, stacks and the entry function
boot_ap_gdts = reinterpret_cast<uint32_t>(apGdts);
boot_ap_stacks = reinterpret_cast<uint32_t>(apStacks);
boot_ap_entry = reinterpret_cast<uint32_t>(&smpEntry);
```

This approach was taken from SerenityOS [15, APIC.cpp]:

Now, the initialized startup routine can be copied to `0x8000`:

4.40 Relocating the Boot Routine (ApicSmp.cpp) C++

```cpp
// Allocate physical memory for copying the startup routine
auto &memoryService = System::getService<MemoryService>();
void *startupCodeMemory = memoryService.mapIO(0x8000, Util::PAGESIZE);

// Identity map the allocated physical memory to the kernel address space
memoryService.unmap(reinterpret_cast<uint32_t>(startupCodeMemory));
memoryService.mapPhysicalAddress(0x8000, 0x8000, Kernel::Paging::PRESENT
                                               | Kernel::Paging::READ_WRITE);

// Copy the startup routine and prepared variables to the identity mapped page
auto startupCode = Util::Address<uint32_t>(reinterpret_cast<uint32_t>(&boot_ap));
auto destination = Util::Address<uint32_t>(0x8000);
destination.copyRange(startupCode, boot_ap_size);
```

The `boot_ap` function is the entry of the startup routine, it is described further in Section 4.5.4.

4.5.3 Universal Startup Algorithm

The “INIT-SIPI-SIPI” sequence, or “Universal Startup Algorithm” is performed by issuing IPIs as described in Section 4.5.1 and using the PIT as time source (see Section 4.3.5):

4.41 The Universal Startup Algorithm (ApicSmp.cpp) C++

```cpp
// Excerpt from the Universal Startup Algorithm
for (uint8_t cpu = 0; cpu < cpuCount; ++cpu) {
    if (cpu == LocalApic::getId()) { continue; } // Skip the BSP

    LocalApic::clearErrors();
    LocalApic::sendInitIpi(cpu, ICREntry::Level::ASSERT);
    LocalApic::waitForIpiDispatch();
    LocalApic::sendInitIpi(cpu, ICREntry::Level::DEASSERT);
    LocalApic::waitForIpiDispatch();
    Pit::earlyDelay(10'000); // 10 milliseconds
    for (uint8_t i = 0; i < 2; ++i) {
        LocalApic::clearErrors();
        LocalApic::sendStartupIpi(cpu, apStartupAddress);
        LocalApic::waitForIpiDispatch();
        Pit::earlyDelay(200); // 200 microseconds
    }

    while (!(runningAPs & (1 << cpu))) {} // Wait until the AP is running
}
```

Boot completion is signaled in the AP’s entry function:

4.42 Signaling AP Boot Completion (smp_entry.cpp) C++

```cpp
// Excerpt from the smpEntry function
void smpEntry(uint8_t cpuId) {
    runningAPs |= (1 << cpuId); // Mark that this AP is running
}
```

4.5.4 Application Processor Boot Routine

Because like the BSP, every AP is initialized in real mode, it has to be switched to protected mode first:

4.43 Preparing the Switch to Protected Mode (smp_boot.asm) nasm

```nasm
; This section has to be compiled for 16-bit real mode
[SECTION .text]
bits 16
boot_ap:
    ; Disable interrupts
    cli

    ; Enable A20 address line
    in al, 0x92
    or al, 2
    out 0x92, al

    ; Load the temporary GDT required for the far jump into protected mode.
    lgdt [tmp_gdt_desc - boot_ap + 0x8000]
```

Then, by enabling protected mode, setting the segment registers and far-jumping to the 32 bit code segment, the switch is performed:

4.44 Switching from Real Mode to Protected Mode (smp_boot.asm) nasm

```nasm
; Continuing boot_ap:
    ; Enable Protected Mode, executed from an identity mapped page.
    mov eax, cr0
    or al, 0x1 ; Set PE bit
    mov cr0, eax

    ; Setup the protected mode segments
    mov ax, 0x10
    mov ds, ax ; Data segment register
    mov es, ax ; Extra segment register
    mov ss, ax ; Stack segment register
    mov fs, ax ; General purpose segment register
    mov gs, ax ; General purpose segment register

    ; Far jump to protected mode, set code segment register
    jmp dword 0x8:boot_ap_32 - boot_ap + 0x8000
```

In 32 bit protected mode, paging is enabled and interrupt processing is prepared by reusing the control register values and the IDT from the BSP:

4.45 Reusing Values from the BSP (smp_boot.asm) nasm

```nasm
; This section has to be compiled for 32-bit protected mode
bits 32
boot_ap_32:
    ; Set cr3, cr0 and cr4 to the BSP's values for paging
    mov eax, [boot_ap_cr3 - boot_ap + 0x8000]
    mov cr3, eax
    mov eax, [boot_ap_cr0 - boot_ap + 0x8000]
    mov cr0, eax
    mov eax, [boot_ap_cr4 - boot_ap + 0x8000]
    mov cr4, eax

    ; Load the system IDT
    lidt [boot_ap_idtr - boot_ap + 0x8000]
```

Finally, to call the AP entry function, the AP requires its own stack. If hardware context switching is to be used, the AP additionally requires its own GDT and TSS:

4.46 Calling the Entry Function (smp_boot.asm) nasm

```nasm
; Continuing boot_ap_32:
    ; Get the local APIC ID of this AP, to locate GDT and stack
    mov eax, 0x1
    cpuid
    shr ebx, 0x18
    mov edi, ebx ; Now the ID is in EDI

    ; Load the AP's prepared GDT and TSS
    mov ebx, [boot_ap_gdts - boot_ap + 0x8000]
    mov eax, [ebx + edi * 0x4] ; Select the GDT
    lgdt [eax]
    mov ax, 0x28
    ltr ax

    ; Load the correct stack for this AP
    mov ebx, [boot_ap_stacks - boot_ap + 0x8000]
    mov esp, [ebx + edi * 0x4] ; Select the stack
    add esp, 0x1000 ; Stack starts at the bottom
    mov ebp, esp

    ; Call the entry function smpEntry(uint8_t cpuid)
    push edi
    call [boot_ap_entry - boot_ap + 0x8000]
```

4.5.5 Application Processor Post-Boot Routine

When the AP has booted, it initializes its own APIC, including the APIC timer and error handler:

4.47 Initializing the Core’s Local APIC (LocalApic.cpp) C++

```cpp
// Excerpt from the smpEntry function
Apic::initializeCurrentLocalApic();
Apic::enableCurrentErrorHandler();
Apic::startCurrentTimer();
```

Because this added another possible recipient to internal APIC messages, the arbitration IDs are synchronized by issuing an “INIT-level-deassert IPI” using the `ICREntry` struct (see Listing 4.15):

4.48 Synchronizing the Arbitration IDs (LocalApic.cpp) C++

```cpp
void LocalApic::synchronizeArbitrationIds() {
    ICREntry icrEntry{};
    icrEntry.vector                 = 0;
    icrEntry.deliveryMode           = ICREntry::DeliveryMode::INIT;
    icrEntry.destinationMode        = ICREntry::DestinationMode::PHYSICAL;
    icrEntry.level                  = ICREntry::Level::DEASSERT;
    icrEntry.triggerMode            = ICREntry::TriggerMode::LEVEL;
    icrEntry.destinationShorthand = ICREntry::DestinationShorthand::ALL;
    icrEntry.destination            = 0;
    writeICR(icrEntry);
    waitForIpiDispatch();
}
```

Because hhuOS’ paging is not designed for multiple processors, the booted AP remains in a busy loop and does not enable its interrupts.

Chapter 5

Testing

Common techniques for testing software include component-based tests, like “unit tests”, and “end-to-end tests”, where the complete functionality of a software system with all its parts is tested. As unit testing is mostly suitable for independent slices of a system’s application logic, it is not very useful for testing low-level software designed to run directly on hardware devices.
This chapter deals with the process and results of testing hhuOS with the APIC implementation developed during this thesis.

5.1 Methodology

This application can be tested by running the entire hhuOS operating system on both emulated and real hardware, and monitoring its state of operation[1].
This can be done in multiple ways:

- When running hhuOS in an emulated environment, the current machine state can be inspected directly[2].
- To test the functionality of an interrupt controller specifically, stub interrupt handlers can be used to verify that a specific interrupt (like an IPI) has been registered.
- By attaching a debugger (very simple for emulated hardware), it can be observed that e.g. the timer interrupt is correctly increasing the timestamp for each running CPU.
- By exposing some of the APIC's internal data through hhuOS' virtual file system, end-to-end verification can be performed from the system itself, as debugging and monitoring is significantly more difficult when running on real hardware.
- Because the interrupt controller is a vital part for many components of an operating system, observing a successful boot is already a significant indicator for a correct implementation.

5.2 Results

QEMU was used as the main development platform, the implemented features were tested by using the QEMU monitor. Specifically, QEMU provides the current register state of all local and I/O APICs, by attaching a debugger to the running operating system correct behavior was observed on the emulated hardware level:

- Working local interrupts – verified by observing the local APIC's IRR.
- Propagation of interrupts to the CPU – verified by observing the local APIC's ISR.
- A working local APIC timer – verified by observing the APIC timer's registers.

Although hhuOS is developed mainly for learning purposes, every OS' core task remains to be the management of computer hardware. For this reason, this implementation was additionally tested on a "ThinkPad X60s" with an Intel "Core 2 Duo"

[1] There is almost no logic that can be tested isolated or hardware independently.

[2] QEMU offers the "QEMU monitor" to query information about specific hardware components, like the local or I/O APIC (`info lapic` and `info pic`).

processor. By providing internal status data through the virtual file system, implemented features (including booting additional APs) were verified on this very specific set of hardware.
More specifically, information about detected and enabled local APICs and the I/O APIC, register values from the BSP's LVT and the REDTBL, the contents of the PIC's **IMR (interrupt mask register)**, and the number of occurred interrupts by core (similar to `/proc/interrupts` in Linux [13]) are exposed on the path `/device/apic/`. This way, the following things could be verified on real hardware:

- The APIC is indeed used instead of the PIC – verified by observing the PIC's IMR.
- Successful MMIO for local and I/O APIC – verified by observing the LVT and REDTBL.
- Devices "plugging in" to the APIC instead of the PIC – verified by observing the LVT and REDTBL.
- Interrupt handlers are called correctly – verified by using the keyboard.
- Working startup of additional APs – verified by checking the state of the AP's local APICs and observing the system log.
- Handling interrupts on another AP – verified by redirecting the keyboard interrupt to another processor and observing the number of occurred keyboard interrupts on this processor[3].

It should be noted, that e.g. the REDTBL observation requires the same MMIO registers as the functionality that is to be verified, which would allow for wrong conclusions if the MMIO mechanism was faulty. It is very unlikely though that all the correct values are provided accidentally.

[3] Interestingly, this was easier to verify on real hardware than in QEMU, because QEMU usually instantly crashes when interrupts are enabled on an AP beside the BSP. Also, redirecting the keyboard interrupt in QEMU significantly changes the behavior of any key press and renders the keyboard unusable. This is expected, as hhuOS' paging is not designed for multiple processors. On the ThinkPad X60s though, it was possible (surprisingly) to use the redirected keyboard interrupt to `cat` the `/device/apic/irqs` file to observe the interrupt statistics, which showed the keyboard interrupts indeed arriving at a different core than the BSP. After a certain time, the system crashes on the ThinkPad as well.

Chapter 6

Conclusion

In this thesis, support for the APIC was integrated into hhuOS. The implementation supports usage of the local APIC in combination with the I/O APIC for regular interrupt handling without the PIC. Also, the APIC's included timer is used to trigger hhuOS' scheduler, and it is demonstrated how to initialize a multiprocessor system using the APIC's IPI capabilities. All of this is implemented with real hardware in mind, so ACPI is used to gather system information during runtime and adapt the initialization and usage accordingly.

6.1 Comparing PIC and APIC Implementations

Handling interrupts using the PIC is simple, as seen in hhuOS' PIC implementation. Initialization and usage can be performed by using a total of only four different registers, two per PIC[1].

In comparison, the code complexity required to use the APIC is very high. The most obvious reason is its significantly increased set of features: Local interrupts are special to the APIC, also the APIC system is made up of multiple components that require internal communication and individual initialization. Additionally, the APIC supports advanced processor topologies with large numbers of cores and offers increased flexibility by allowing to configure individual interrupts' vectors, destinations, signals and priorities, which results in additional setup.

Another source of complexity that is not present with the PIC is the APIC's variability: With the PC/AT architecture, the entire hardware interrupt configuration is known before boot. This is not the case when using the APIC, as the number of interrupt controllers or even the number and order of connected devices is only known while the system is running. Parsing this information from ACPI tables and allowing the implementation to adapt to different hardware configurations, all while maintaining PC/AT compatibility, increases the amount of code by a large margin.

In general, all of this effort results in a much more powerful and future-proof system: It is not limited to specific hardware configurations, allows scaling to a large amount of interrupts or processors, and is required for multiprocessor machines, which equals almost all modern computer systems.

6.2 Future Improvements

6.2.1 Dependence on ACPI

The APIC system requires a supplier of system information during operation, but this must not necessarily be ACPI. Systems following Intel's "MultiProcessor Specification" [10] can acquire the required information by parsing configuration tables similar to ACPI's system description tables, but provided in accordance to the MultiProcessor Specification. This would increase the number of systems supported by this APIC implementation, but the general compatibility improvement is difficult to quantize, as single-core systems are still supported by the old PIC implementation[2].

Alternatively, systems that do not support ACPI could be supported partially by utilizing the local APIC's virtual wire mode [10, sec. 3.6.2.2]. The reliance on information provided in the MADT stems mostly from using the I/O APIC as the external interrupt controller. By using the local APIC for its local interrupt and

[1] Omitting the infrastructure that is required for both, PIC and APIC, like the IDT or dispatcher.

[2] The "MultiProcessor Specification" was (pre-) released in 1993 [16], the first ACPI release was in 1996 [17]. Multiprocessor systems between these years could be affected.

multiprocessor functionality, but keeping the PC/AT compatible PIC as the external interrupt controller, multiprocessor systems without ACPI could be supported.

6.2.2 Discrete APIC and x2Apic

This implementation only supports the xApic architecture, as it is convenient to emulate and the x2Apic architecture is fully backwards compatible [2, sec. 3.11.12]. The original, discrete local APIC (Intel 82489DX) is not supported explicitly, which reduces this implementation's compatibility to older systems[3]. Supporting the x2Apic architecture does not increase compatibility, but using the x2Apic's MSR-based atomic register access is beneficial in multiprocessor systems. Newer ACPI versions provide separate x2Apic specific structures in the MADT [18, sec. 5.2.12.12], so supporting x2Apic also requires supporting modern ACPI versions[4].

6.2.3 PCI Devices

The APIC is capable of handling MSIs, but this is not implemented in this thesis. To support PCI devices using the APIC instead of the PIC, either support for MSIs has to be implemented, or ACPI has to be used to determine the corresponding I/O APIC interrupt inputs. This is rather complicated, as it requires interacting with ACPI using AML [18, sec. 6.2.13], which needs an according interpreter[5].

6.2.4 Multiprocessor Systems

This implementation demonstrates AP startup in SMP systems by using the APIC's IPI mechanism, but processors are only initialized to a spin-loop. Because the APIC is a prerequisite for more in-depth SMP support, this implementation enables more substantial improvements in this area, like distributing tasks to different CPU cores and using the cores' local APIC timers to manage core-local schedulers.

Another possible area of improvement is the execution of the interrupt handlers: At the moment, each I/O APIC redirects every received interrupt to the BSP's local APIC. Distributing interrupts to different CPU cores could improve interrupt handling performance, especially with frequent interrupts, like from network devices. This redirection could be implemented statically for each interrupt or based on priorities, but could also happen dynamically ("IRQ-balancing" [20]), based on interrupt workload measurements.

[3] Although it is possible that no or only very few modifications are necessary, as this implementation does not utilize many of the xApic's exclusive features [2, sec. 3.23.27].

[4] Currently (on 27/02/2023), hhuOS supports ACPI 1.0b [5].

[5] There exist modular solutions that can be ported to the target OS [19].

6.2.5 UEFI Support

Currently, hhuOS' ACPI system only works with BIOS, so when booting an UEFI system, this implementation is disabled, because it requires ACPI. By supporting ACPI on UEFI systems, this implementation could also be used for those systems, which would improve compatibility with modern platforms.

Appendices

Appendix A

Figures

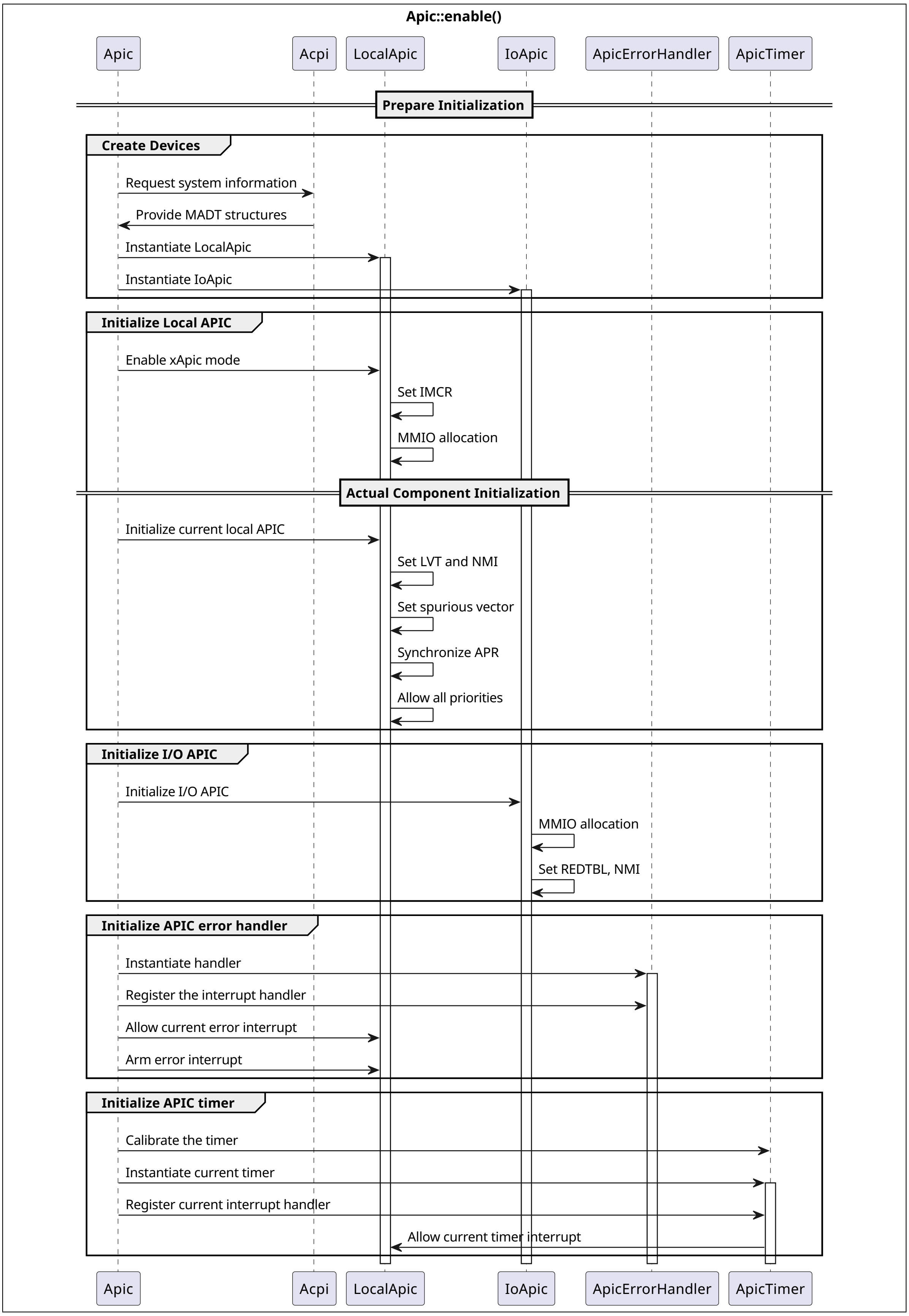
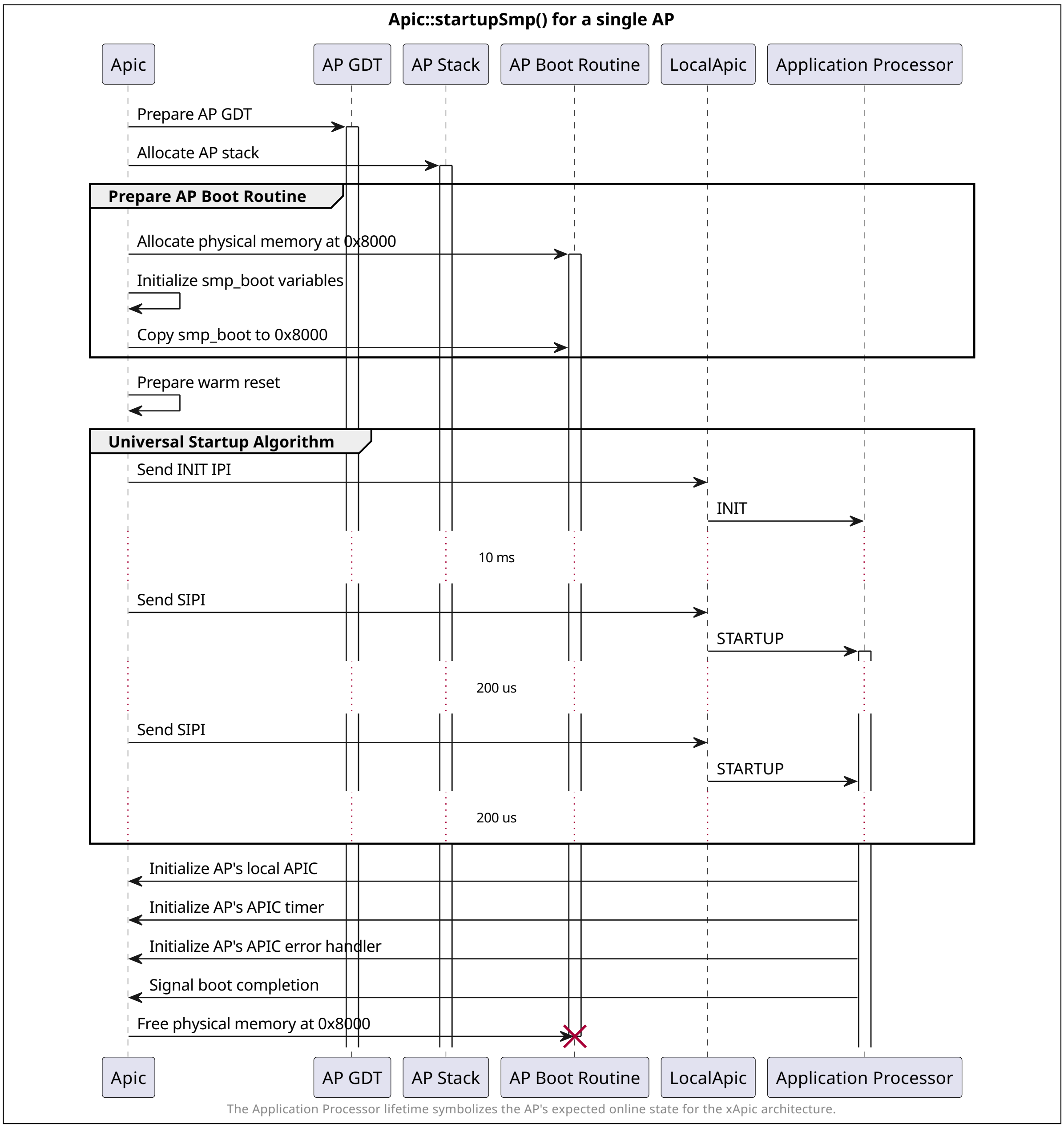


Figure A.1: Enabling the APIC Subsystem.



The Application Processor lifetime symbolizes the AP's expected online state for the xApic architecture.

Figure A.2: Starting SMP Operation.

Note that this diagram is slightly misleading, because the application processor runs in parallel and is not susceptible to delays on the BSP. Initialization of the AP's local APIC follows the sequence described by Figure A.1 (starting at "Actual Component Initialization", excluding the I/O APIC, error handler instantiation and timer calibration).

Appendix B

Tables

This section lists all the registers and structures required to follow Chapter 4.

B.1 Local APIC Registers

Register Name	Memory Offset
Local APIC ID Register	0x20
Local APIC Version Register	0x30
Task Priority Register	0x80
EOI Register	0xB0
Spurious Interrupt Vector Register	0xF0
Error Status Register	0x280
Interrupt Command Register[0:31]	0x300
Interrupt Command Register[32:63]	0x310
LVT Timer Register	0x320
LVT LINT1 Register	0x360
LVT Error Register	0x370
Timer Initial Count Register	0x380
Timer Divide Configuration Register	0x3E0

Table B.1: Local APIC Registers used in this Implementation [2, sec. 3.11.4.1].

Bit Number	Description
0:23	Reserved
24:31	Local APIC ID

Table B.2: Local APIC ID Register (xApic since Pentium 4) [2, sec. 3.11.4.6].

Bit Number	Description
0:7	Local APIC Version
8:15	Reserved
16:23	Max LVT Entry
24	EOI Broadcast Suppression Support
25:31	Reserved

Table B.3: Local APIC Version Register [2, sec. 3.11.4.8].

Bit Number	Description
0:3	Task-Priority Subclass
4:7	Task-Priority Class
8:31	Reserved

Table B.4: Task Priority Register [2, sec. 3.11.8.3.1].

Bit Number	Description
0:31	Send EOI Signal

Table B.5: Local APIC EOI Register [2, sec. 3.11.8.5].

Bit Number	Description
0:7	Spurious Interrupt Vector
8	APIC Software Enable/Disable
9	Focus Processor Checking
10:11	Reserved
12	EOI Broadcast Suppression
13:31	Reserved

Table B.6: Spurious Interrupt Vector Register [2, sec. 3.11.9].

Bit Number	Description
0:4	Reserved
5	Send Illegal Vector
6	Receive Illegal Vector
7	Illegal Register Access
8:31	Reserved

Table B.7: Error Status Register (Pentium 4) [2, sec. 3.11.5.3].

Bit Number	Description
0:7	Interrupt Vector
8:10	Delivery Mode
11	Destination Mode
12	Delivery Status
13	Reserved
14	Level
15	Trigger Mode
16:17	Reserved
18:19	Destination Shorthand
20:55	Reserved
56:63	Destination Field

Table B.8: Interrupt Command Register [2, sec. 3.11.6.1].

Bit Number	Description
0:7	Interrupt Vector
8:11	Reserved
12	Delivery Status
13:15	Reserved
16	Masked
17:18	Timer Mode
19:31	Reserved

Table B.9: LVT Timer Register [2, sec. 3.11.5.1].

Bit Number	Description
0:7	Interrupt Vector
8:11	Reserved
12	Delivery Status
13:15	Reserved
16	Masked
17:31	Reserved

Table B.10: LVT Error Register [2, sec. 3.11.5.1].

Bit Number	Description
0:7	Interrupt Vector
8:10	Delivery Mode
11	Reserved
12	Delivery Status
13	Pin Polarity
14	Remote IRR
15	Pin Polarity
16	Masked
17:31	Reserved

Table B.11: LVT LINT1 Register [2, sec. 3.11.5.1].

Bit Number	Description
0:31	Initial Count

Table B.12: Timer Initial Count Register [2, sec. 3.11.5.4].

Bit Number	Description
0:1	Divider
2	Reserved
3	Divider
4:31	Reserved

Table B.13: Timer Divide Configuration Register [2, sec. 3.11.5.4].

Bit Number	Description
0:7	Reserved
8	BSP Flag
9	Reserved
10	Enable x2Apic
11	Enable xApic
12:35	APIC Base Address
36:63	Reserved

Table B.14: IA32_ APIC_BASE MSR [2, sec. 3.11.12.1].

B.2 I/O APIC Registers

Register Name	Memory Offset
Index Register	0x00
Data Register	0x10
Register Name	**Index Offset**
I/O APIC ID Register	0x00
I/O APIC Version Register	0x01
Redirection Table	0x10:0x3F

Table B.15: I/O APIC Registers used in this Implementation [12, sec. 9.5].

Bit Number	Description
0:7	Indirect Register Index

Table B.16: I/O APIC Index Register [12, sec. 9.5.2].

Bit Number	Description
0:31	Indirect Register Data

Table B.17: I/O APIC Data Register [12, sec. 9.5.3].

Bit Number	Description
0:14	Reserved
15	Scratchpad Bit
16:23	Reserved
24:27	I/O APIC ID
28:31	Reserved

Table B.18: I/O APIC ID Register [12, sec. 9.5.6].

Bit Number	Description
0:7	I/O APIC Version
8:14	Reserved
15	PRQ
16:23	Maximum Redirection Entries
24:31	Reserved

Table B.19: I/O APIC Version Register [12, sec. 9.5.7].

Bit Number	Description
0:7	Interrupt Vector
8:10	Delivery Mode
11	Destination Mode
12	Delivery Status
13	Pin Polarity
14	Remote IRR
15	Trigger Mode
16	Masked
17:47	Reserved
48:55	Extended Destination ID
56:63	Destination

Table B.20: I/O APIC REDTBL Register [12, sec. 9.5.8].

B.3 System Description Tables

Byte Number	Description
0:35	MADT Header
36:39	Local APIC Base Address
40:43	Local APIC Flags
44:	List of APIC Structures

Table B.21: ACPI MADT [5, sec. 5.2.8].

Byte Number	Description
0:1	APIC Structure Header
2	ACPI Processor ID
3	APIC ID
4:7	Local APIC Flags (see Table B.23)

Table B.22: MADT Processor Local APIC Structure [5, sec. 5.2.8.1].

Bit Number	Description
0	Enabled
1:31	Reserved

Table B.23: Local APIC Flags [5, sec. 5.2.8.1].

Byte Number	Description
0:1	APIC Structure Header
2	I/O APIC ID
3	Reserved
4:7	I/O APIC Base Address
8:11	I/O APIC GSI Base

Table B.24: MADT I/O APIC Structure [5, sec. 5.2.8.2].

Byte Number	Description
0:1	APIC Structure Header
2	Bus
3	Source
4:7	GSI
8:9	INTI Flags (see Table B.26)

Table B.25: MADT Interrupt Source Override Structure [5, sec. 5.2.8.3.1].

Bit Number	Description
0:1	Pin Polarity
2:3	Trigger Mode
4:11	Reserved

Table B.26: INTI Flags [5, sec. 5.2.8.3.1].

Byte Number	Description
0:1	APIC Structure Header
2:3	INTI Flags (see Table B.26)
4:7	GSI

Table B.27: MADT I/O APIC NMI Source [5, sec. 5.2.8.3.2].

Byte Number	Description
0:1	APIC Structure Header
2	ACPI Processor ID
3:4	INTI Flags (see Table B.26)
5	Local APIC INTI

Table B.28: MADT Local APIC NMI Source [5, sec. 5.2.8.3.3].

Bibliography

[1] B. Akguel, C. Gesse, F. Ruhland, F. Krakowski, M. Schoettner, *et al.*, *hhuOS - A small operating system*, GitHub, Branch "experimental". [Online]. Available: `https://github.com/hhuOS/hhuOS/tree/experimental` (visited on 02/08/2023).

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, Dec. 2022. [Online]. Available: `https://cdrdv2.intel.com/v1/dl/getContent/671200` (visited on 02/08/2023).

[3] *PCI 2.2 Local Bus Specification*, Rev 2.2, PCI Special Interest Group, Dec. 1998. [Online]. Available: `https://www.ics.uci.edu/~harris/ics216/pci/PCI_22.pdf` (visited on 02/08/2023).

[4] F. Cloutier, *x86 and amd64 instruction reference*, Sep. 2022. [Online]. Available: `https://www.felixcloutier.com/x86/index.html` (visited on 02/10/2023).

[5] *Advanced Configuration and Power Interface Specification*, Rev 1.0b, UEFI Forum, Feb. 1999. [Online]. Available: `http://uefi.org/sites/default/files/resources/ACPI_1_Errata_B.pdf` (visited on 02/08/2023).

[6] *Intel Processor Identification and the CPUID Instruction*, Intel Corporation, May 2012. [Online]. Available: `https://www.scss.tcd.ie/~jones/CS4021/processor-identification-cpuid-instruction-note.pdf` (visited on 02/08/2023).

[7] *Intel 8254 Datasheet*, Rev 5, Intel Corporation, Sep. 1993. [Online]. Available: `https://www.scs.stanford.edu/10wi-cs140/pintos/specs/8254.pdf` (visited on 02/26/2023).

[8] *Intel 8259A Datasheet*, Rev 3, Intel Corporation, Jun. 1978. [Online]. Available: `https://pdos.csail.mit.edu/6.828/2017/readings/hardware/8259A.pdf` (visited on 02/08/2023).

[9] *PC/AT Technical Reference*, IBM, Mar. 1986. [Online]. Available: `http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/pc/at/6183355_PC_AT_Technical_Reference_Mar86.pdf` (visited on 02/08/2023).

[10] *Intel MultiProcessor Specification*, Rev 1.4, Intel Corporation, May 1997. [Online]. Available: `https://web.archive.org/web/20121002210153/http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf` (visited on 02/08/2023).

[11] *OSDev*. [Online]. Available: https://wiki.osdev.org/ (visited on 02/10/2023).

[12] *Intel ICH5 Datasheet*, Rev 001, Intel Corporation, Apr. 2003. [Online]. Available: https://www.intel.com/content/dam/doc/datasheet/82801eb-82801er-io-controller-hub-datasheet.pdf (visited on 02/08/2023).

[13] L. Torvalds *et al.*, *Linux 6.1.11*, Feb. 2023. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v6.1.11 (visited on 02/10/2023).

[14] F. Kaashoek, R. Morris, and R. Cox, *xv6 OS*. [Online]. Available: https://github.com/mit-pdos/xv6-public (visited on 02/10/2023).

[15] A. Kling *et al.*, *SerenityOS*. [Online]. Available: https://github.com/SerenityOS/serenity (visited on 02/10/2023).

[16] *Intel MultiProcessor Specification*, Rev 1.0, Intel Corporation, Oct. 1993.

[17] *Advanced Configuration and Power Interface Specification*, Rev 1.0, UEFI Forum, Dec. 1996. [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_1.pdf (visited on 02/21/2023).

[18] *Advanced Configuration and Power Interface Specification*, Rev 6.5, UEFI Forum, Aug. 2022. [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf (visited on 02/08/2023).

[19] *ACPICA*, Intel Corporation. [Online]. Available: https://acpica.org/ (visited on 02/23/2023).

[20] N. Horman, P. J. Waskiewicz, A. Arapov, *et al.*, *Irqbalance*. [Online]. Available: https://github.com/Irqbalance/irqbalance (visited on 02/23/2023).

List of Listings

List of Tables

List of Figures

Statutory Declaration

I hereby state that I have written this Bachelor's Thesis independently and that I have not used any sources or aids other than those declared. All passages taken from the literature have been marked as such. This thesis has not yet been submitted to any examination authority in the same or a similar form.

Düsseldorf, March 8th, 2023

C. Urlache

Christoph Urlacher