3.kernel 重新加载GDT中断及异常处理

GDT（Global Descriptor Table，全局描述符表）

内存分段管理：在保护模式下，内存通过“段选择子 + 偏移”的方式寻址，GDT存储每个段的描述信息。
定义代码段、数据段、系统段（如TSS）的访问权限（如特权级、可读/可写/可执行）。
在实模式和保护模式切换时，GDT必须正确设置。
GDT是一个数组，每一项称为**段描述符（Segment Descriptor）**，每个描述符占 **8字节（64位）**
2.loader实现 > GDT

lgdt加载GDT

在GDT设置好以后，需要通过lgdt()来将该表的信息加载到GDTR寄存器中。GDTR是一个48位的寄存器，包含limit和base两部分，如下图所示。

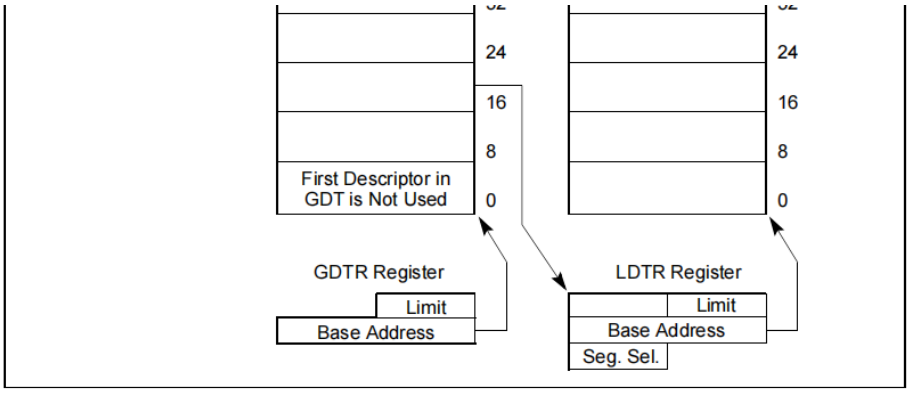


Figure 3-10. Global and Local Descriptor Tables

Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register (see Section 2.4, “Memory-Management Registers”). The base address of the GDT should be aligned on an eight-byte boundary to yield the best processor performance. The limit value for the GDT is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The first descriptor in the GDT is not used by the processor. A segment selector to this “null descriptor” does not

其中base address是32位，limit是16位。
limit保存的GDT表项的字节大小(64KB)，当其值为0时，表示其字节数为1；当GDT表项的数量为N时，limit的值设置成8 * N - 1。

> 不准确的代码： lgdt((uint32_t)gdt_table, sizeof(gdt_table));

整个GDT表最大为64KB，而每个表项为8字节，所以表项数最大为8K个(info register 0x7fff)。
在重新设置后，需要使用jmp $选择子, $offset跳转。

```
    // 重新加载GDT
    jmp $KERNEL_SELECTOR_CS, $gdt_reload

gdt_reload:
    mov $KERNEL_SELECTOR_DS, %ax        // 16为数据段选择子
    mov %ax, %ds
    mov %ax, %ss
    mov %ax, %es
    mov %ax, %fs
    mov %ax, %gs

```

之所以要重新跳转，是为了使CS寄存器重新加载新GDT表中相应的信息。如下图所求，CS寄存器除了保存选择子信息外，还会在内部自动保存从GDT表中加载的基地址、界限、属性等相关信息。当使用jmp $选择子, $offset进行跳转后，CS将从新表中加载新的信息，这样就避免了仍然使用原来的信息。所以，最好是重新加载，以使用新的信息。

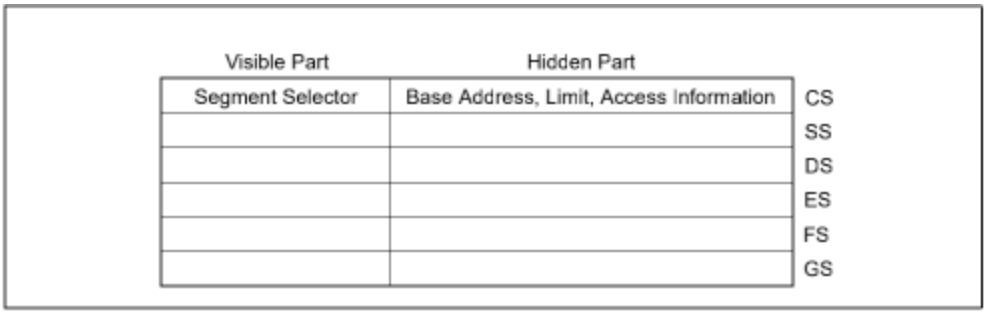


Figure 3-7. Segment Registers

当然，在目前的工程中，由于新表中代码段相关的信息和旧表中的相同，所以不重新加载也不影响程序的运行。

段选择子（Segment Selector）

访问GDT时，CPU使用 段选择子（16位，通过段寄存器）：

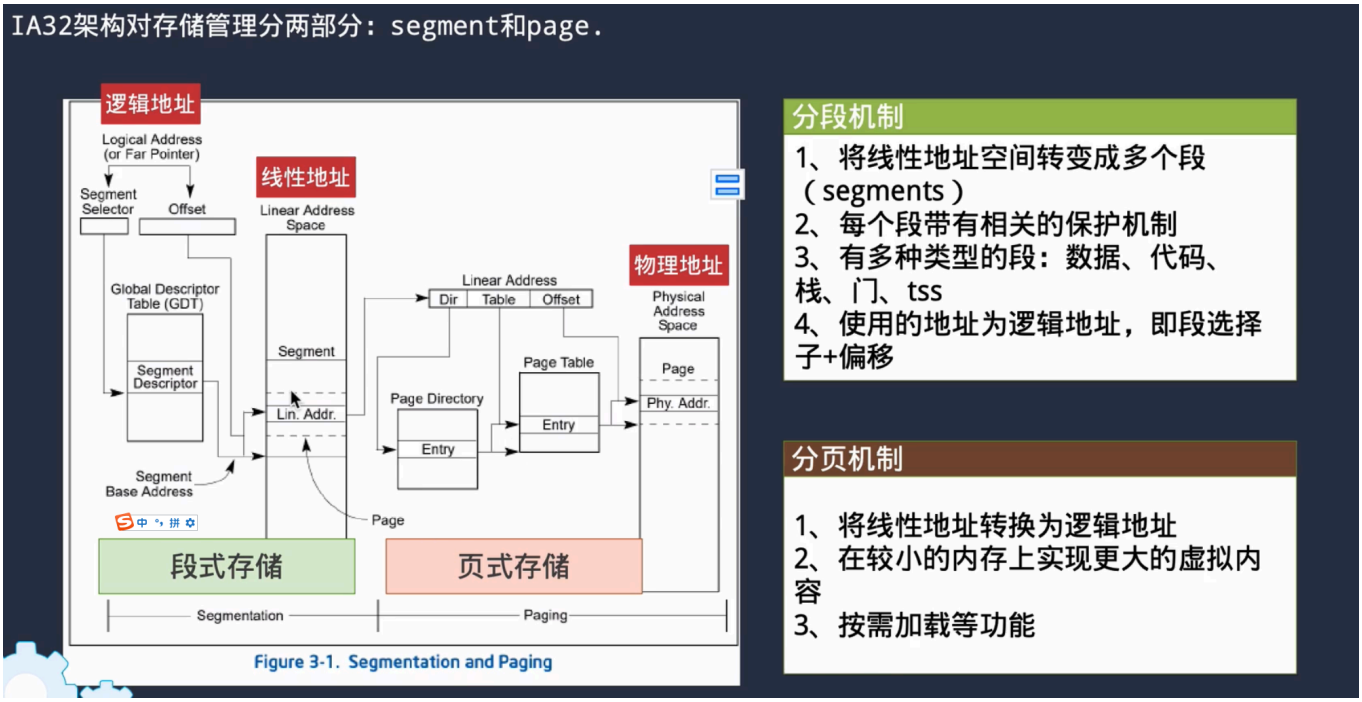
```
| 15-3 | 2 | 1-0 |
|------|---|-----|
| 索引 |TI | RPL |
```

- **索引（Index）**：GDT项的编号（0~8191，因为GDT最多8192项）。
- **TI（Table Indicator）**：
  - `0` = 使用GDT
  - `1` = 使用LDT（局部描述符表）
- **RPL（Requested Privilege Level）**：请求特权级（0~3)

IA32内存管理机制概览

IA32将内存管理分为两部分：段式存储和页式存储，目前我们只考虑了段式存储。

- 逻辑地址：使用段选择子+偏移表示的地址
- 线性地址：将段+偏移进行转换后的地址，也是我们程序中所用的地址。如果分页机制未开启，则线性地址=物理地址。
- 物理地址：实际物理内存中的地址



平坦模型

IA32中段式存储比较复杂，使用的是基础平坦模型，其特点如下：

- 只用了两个段：代码段和数据段
- 段起始地址均为0，大小为4GB。即不管实际有没有那么大的内存空间，都设置成那么大。访问不存在的区域会产生保护异常。

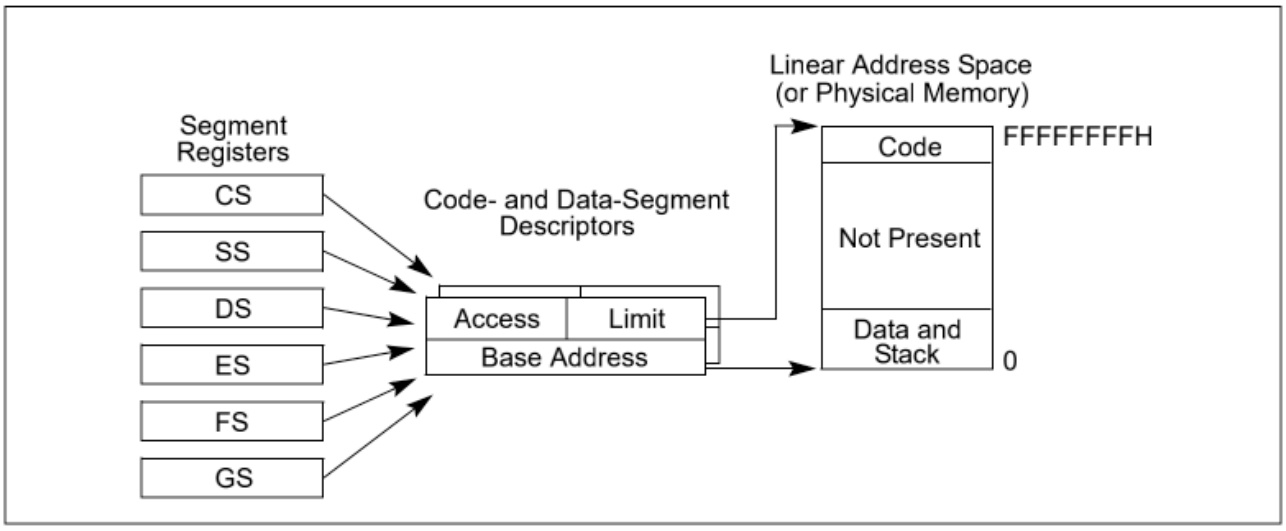


Figure 3-2. Flat Model

即采用上述模型中，我们不使用limit界限检查，也不使用段的基地址功能，这样程序处理会更为简单。

逻辑地址到线性地址的转换

在程序进行内存访问时，会进行逻辑地址的转换，转换到线性地址（暂不考虑分页机制)，转换过程如下：

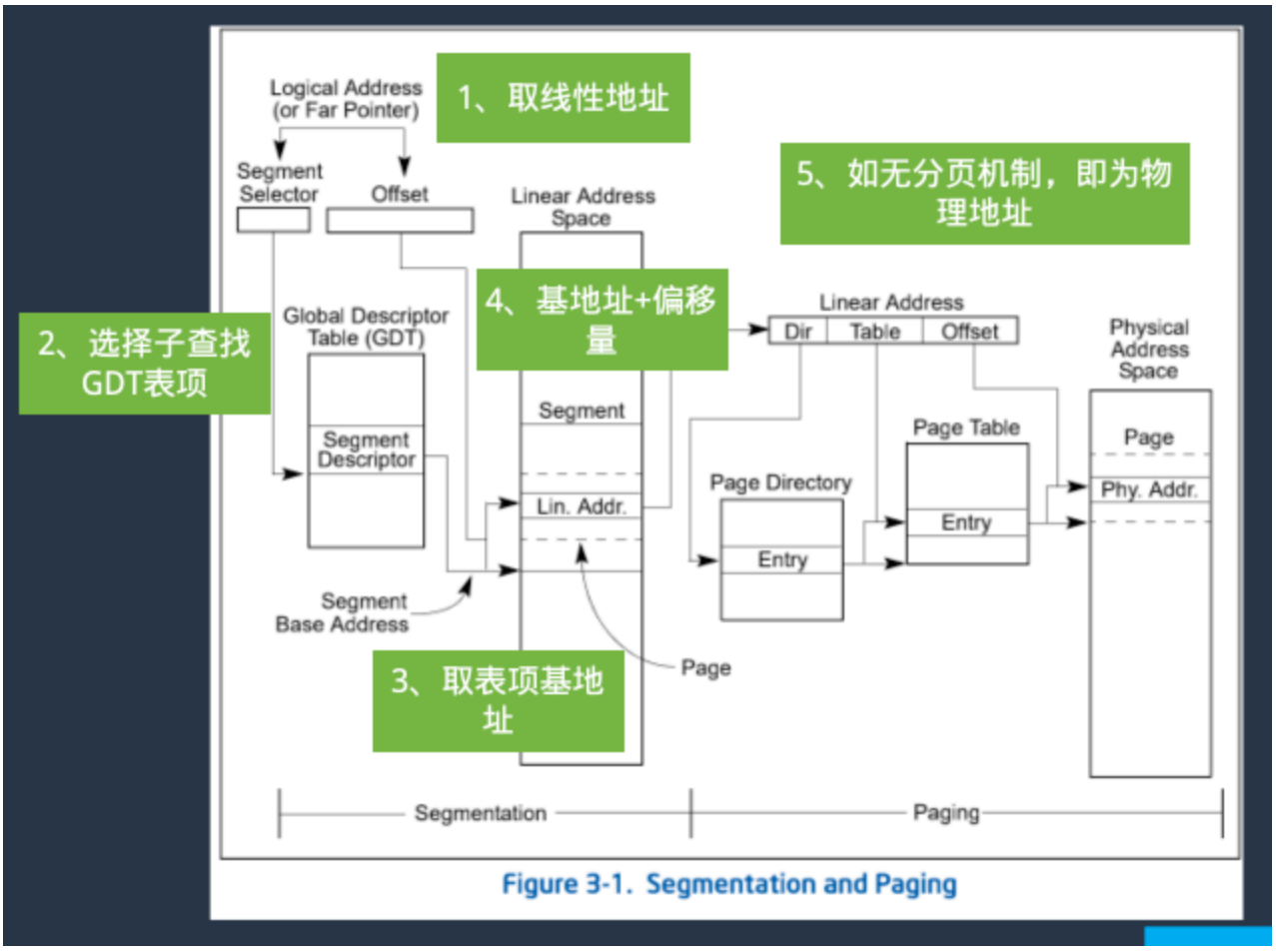


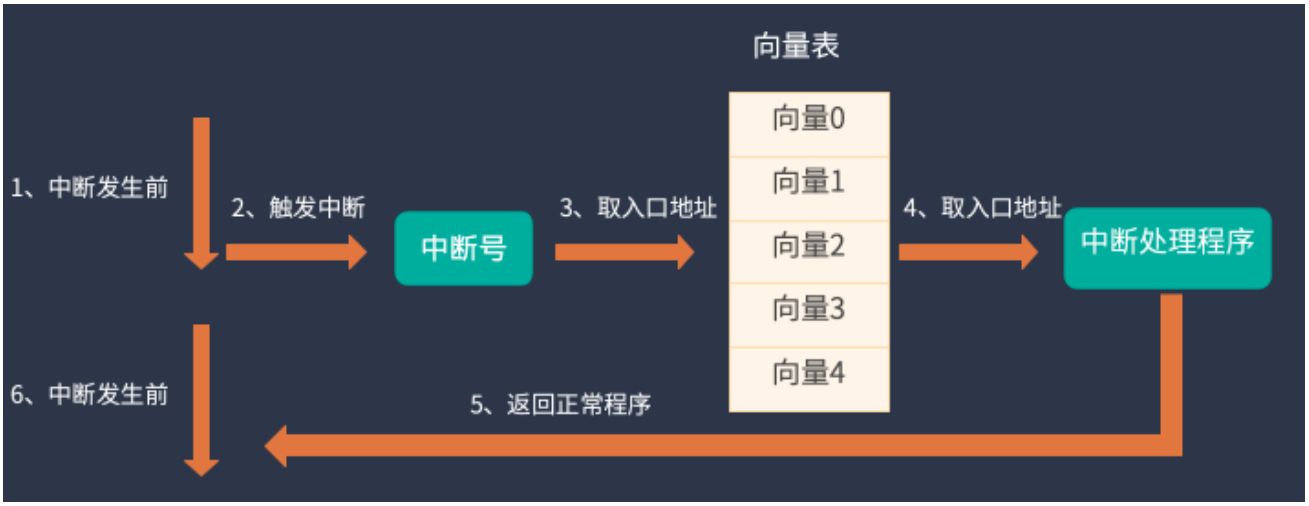
Figure 3-1. Segmentation and Paging

中断和异常

- 异常：由于 CPU 内部事件所引起的中断，如程序出错(非法指令、地址越界、除0异常)。
- 中断：由于外部设备事件所引起的中断，如通常的磁盘中断、打印机中断等通常与现行指令无关

处理流程

1. 打断当前正在运行的程序
2. 从向量表中取出处理程序的首地址，跳转到首地址运行；有些CPU中，向量表中存储的是跳转指令，则跳转到该指令处运行
3. 执行完程序程序后，继续从之前中断地位置运行。



IDT（Interrupt Descriptor Table，中断描述符表）

进入保护模式后，需要使用新的中断描述符表，即IDT表。由IDTR寄存器。寄存器指向。

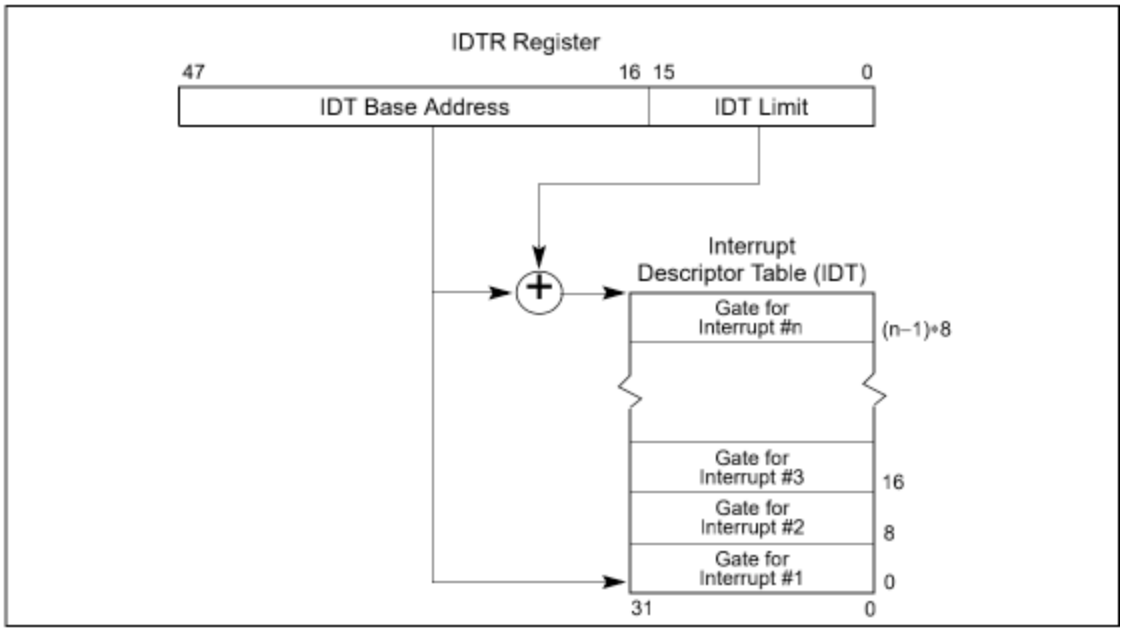


Figure 6-1. Relationship of the IDTR and IDT

IDT是一个 **描述符数组**，每个描述符（称为 **门描述符，Gate Descriptor**）占 **8字节（64位）**，包含：

- **中断处理程序的入口地址**（段选择子 + 偏移）。
- **门的类型**（任务门、中断门、陷阱门）。
- **特权级（DPL）**，控制哪些代码可以触发该中断。

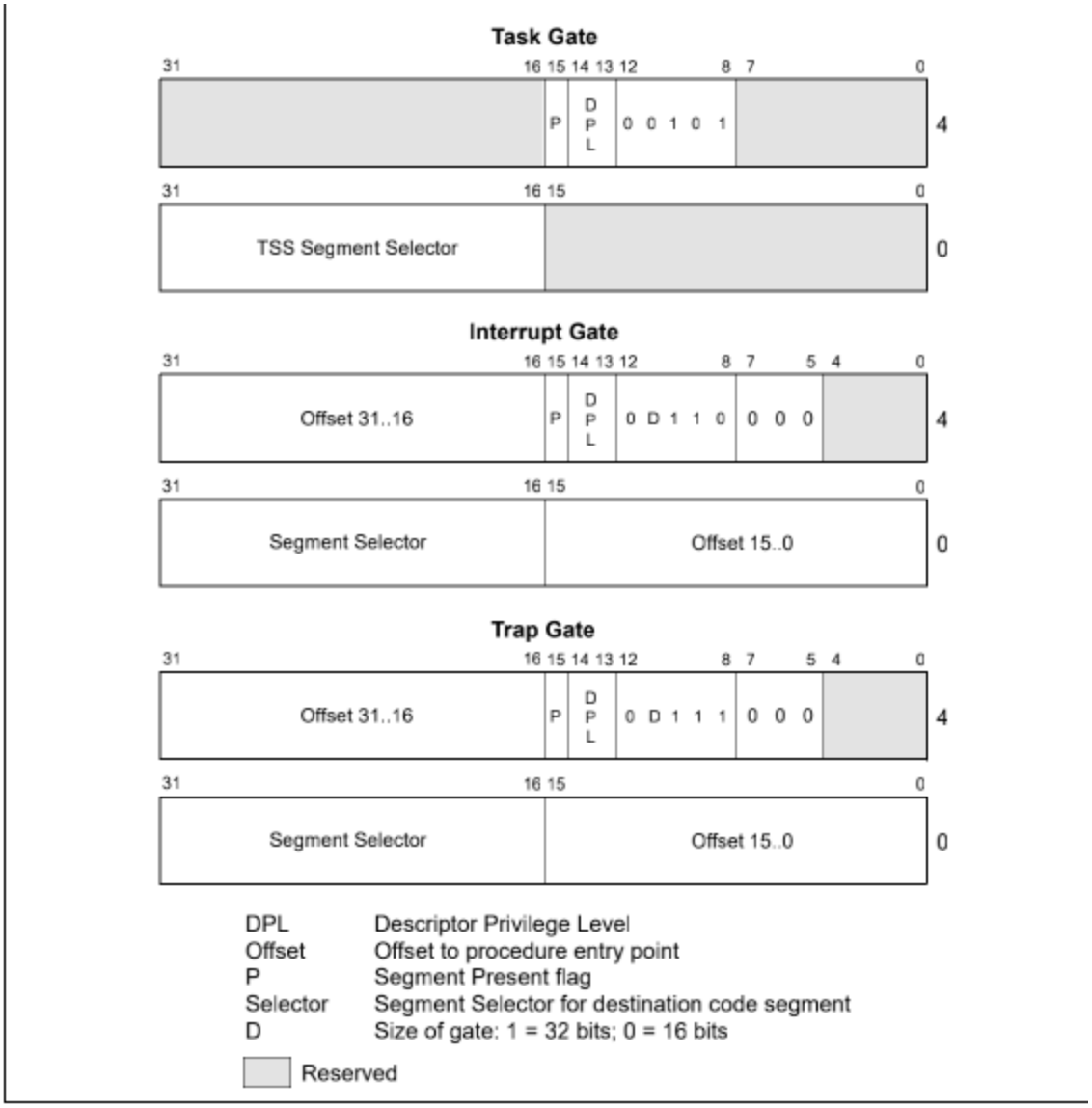
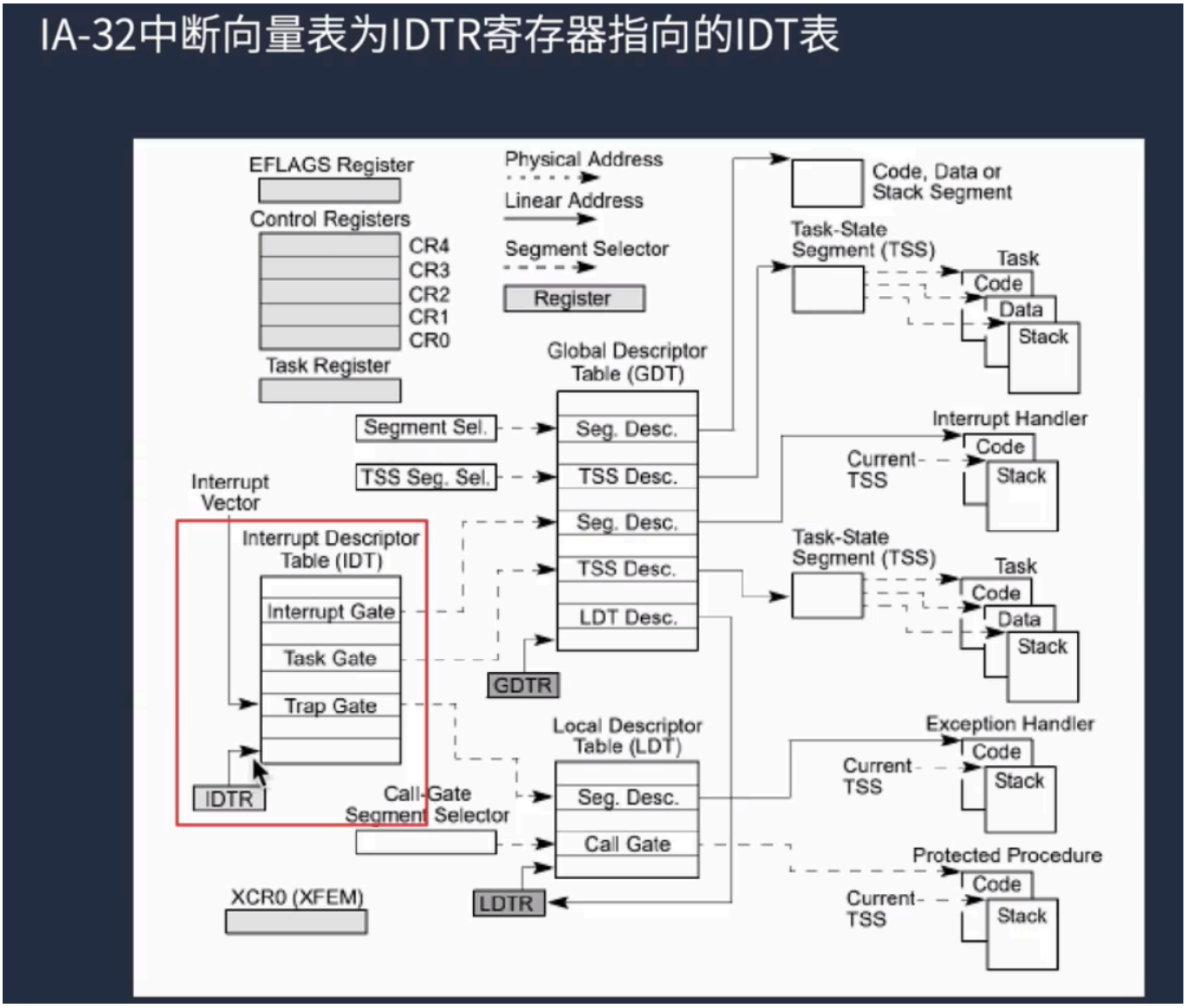


Figure 6-2. IDT Gate Descriptors

在配置IDT表项时，需要将IDT表项中的segment selector设置为KERNEL_SELECTOR_CS，偏移量设置为函数的入口地址即可。

中断处理流程

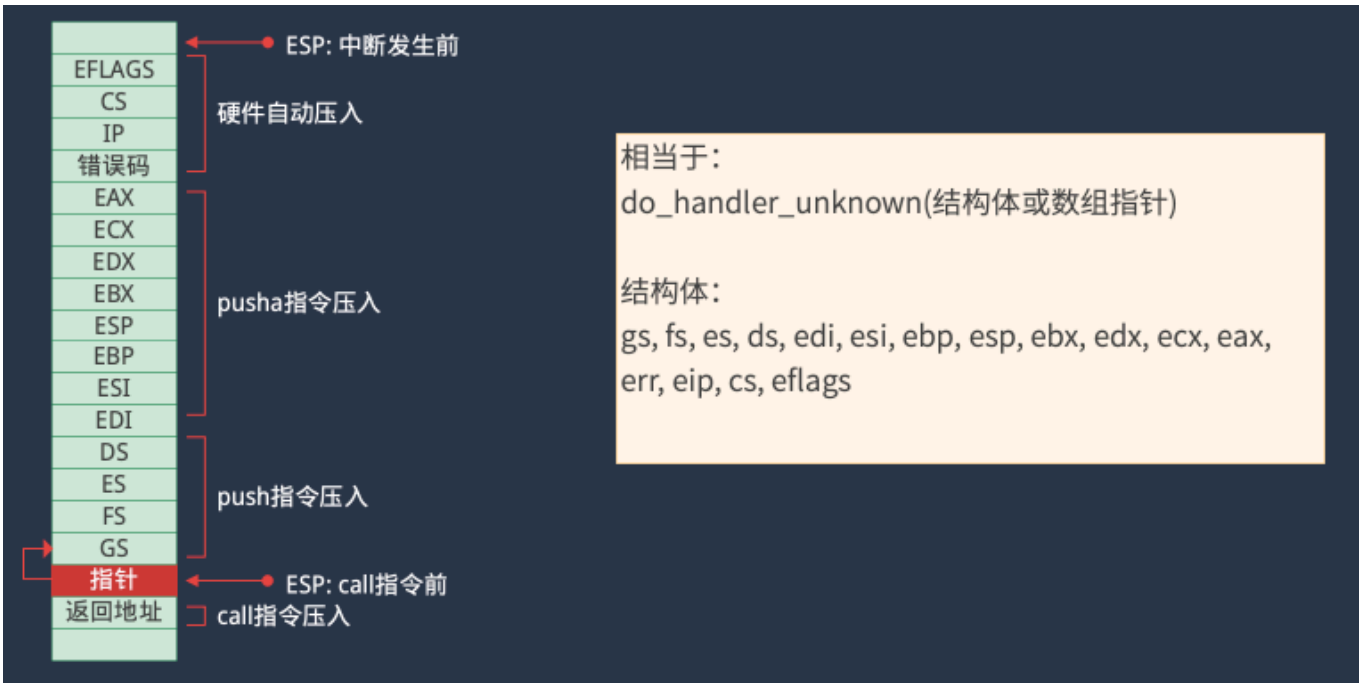


1. **CPU收到中断号 `N`**（如 `IRQ0=0x20`、`#PF=0x0E`）。
2. **检查 `IDTR`**，找到IDT的基址。
3. **计算描述符位置：** `IDT_Entry = IDTR.base + N * 8`（因为每个描述符8字节）。
4. **检查权限**（CPL ≤ DPL，否则触发 `#GP` 异常）。
5. **跳转到处理程序：**
   - 如果是 **中断门**，CPU自动执行 `CLI`（关中断）。
   - 用选择子从GDT表中查找段的首地址
   - 将段首地址+IDT表项中的偏移量，生成处理程序的首地址
6. **执行ISR（Interrupt Service Routine）**。
7. **返回**（`IRET` 指令恢复现场）。

中断现场保护以及解析异常栈信息

将所有的寄存器进行压栈保护，压入方法为中断发生时，CPU会自动保存一部分，其余在中断程序程序中，通过push指令压入一部分。

```
.macro exception_handler name num with_error_code
        .extern do_handler_\name
        .global exception_handler_\name
    exception_handler_\name:
        // 如果没有错误码，压入一个缺省值
        // 这样堆栈就和有错误码的情形一样了
        .if \with_error_code == 0
            push $0
        .endif

        // 压入异常号
        push $\num

        // 保存所有寄存器
        pushal
        push %ds
        push %es
        push %fs
        push %gs

        // 调用中断处理函数
        push %esp

        call do_handler_\name

        add $(1*4), %esp        // 丢掉esp

        // 恢复保存的寄存器
        pop %gs
        pop %fs
        pop %es
        pop %ds
        popal

        // 跳过压入的异常号和错误码
        add $(2*4), %esp
        iret
.endm
```



利用栈帧的特性，函数调用时会先压入参数。
压入栈中的这些寄存器，相当于是传递给它的参数。难点在于压入栈中的寄存器数量太多。
使用了一个结构体exception_frame_t映射栈中这些寄存器的值，给do_handler_unknown()传递了一个指向所有压入的数据的exception_frame_t类型的指针（在call之前push %esp，此时的esp就是指向exception_frame_t类型的指针），从而能方便地直接从结构体中读取所有这些寄存器的值。

x86异常类型表

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

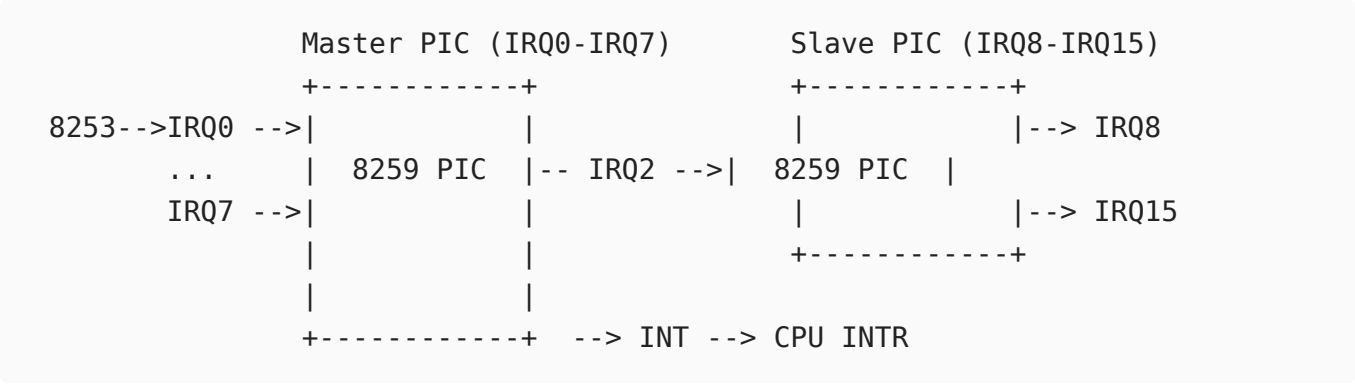
初始化中断控制器

8259功能和结构

8259 PIC（可编程中断控制器）是x86架构中用于 **管理和扩展外部硬件中断** 的关键芯片。
负责接收来自设备（如键盘、定时器、硬盘）的中断请求（IRQ），并按优先级传递给CPU。

x86 CPU只有1个`INTR`引脚，8259 PIC提供 **8个IRQ引脚**（可级联扩展至64个）。

- **优先级管理**：决定哪个中断优先响应（IRQ0最高，IRQ7最低）。
- **中断屏蔽**：允许屏蔽特定IRQ。
- **中断向量号映射**：将IRQ映射到CPU的中断号（如IRQ0 → `0x20`）。



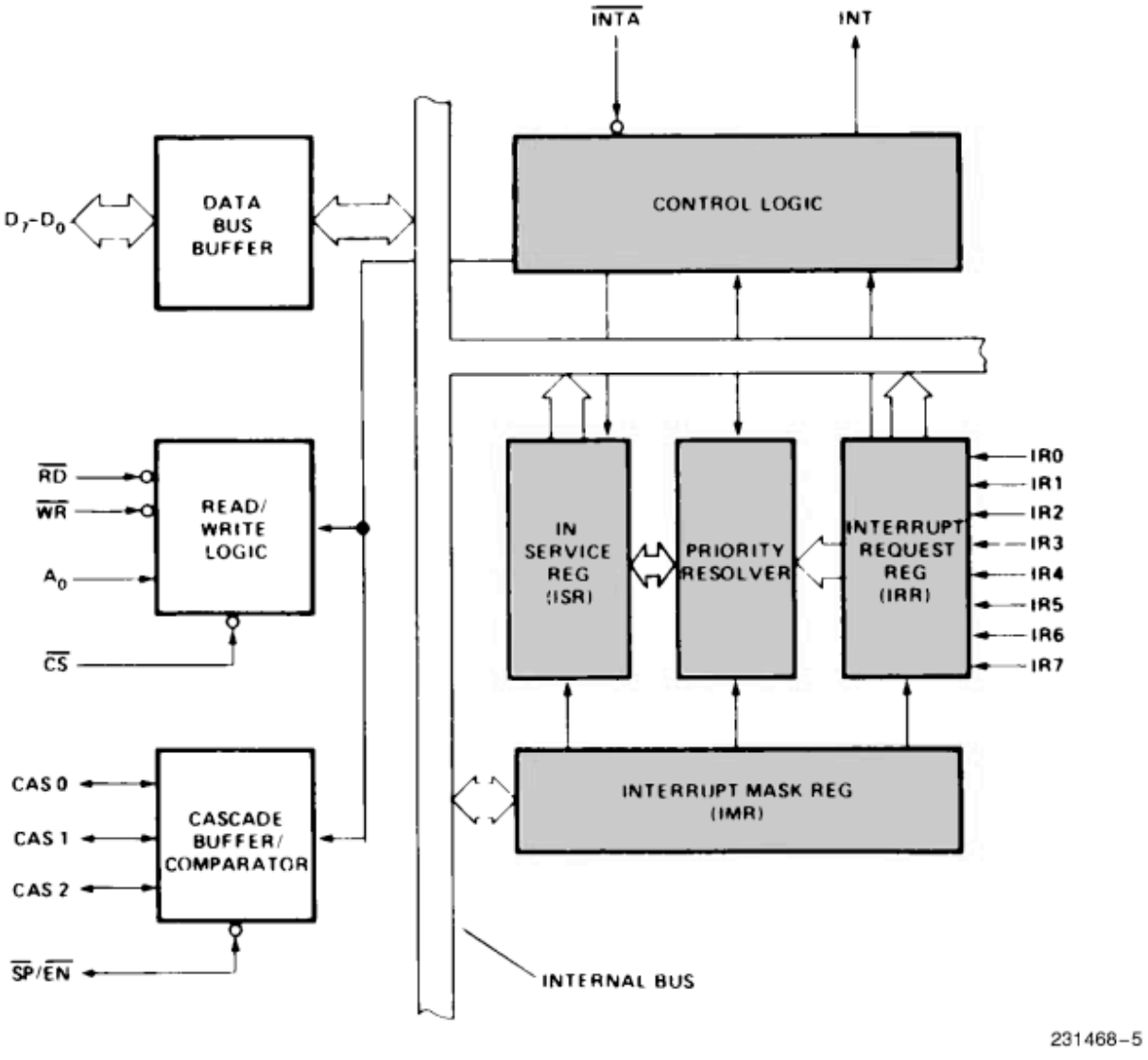


Figure 4a. 8259A Block Diagram

8259的工作模式较为复杂，只做了非常简单的配置，不考虑中断嵌套、优先级等问题。下：

- 主片：边缘触发，级联、起始中断序号为0x20，IRQ2上有从片，普通全嵌套、非缓冲、非自动结束、8086模式
- 从片：边缘触发，级联、起始中断序号为0x28，连接到主片的IRQ2上，普通全嵌套、非缓冲、非自动结束、8086模式

中断处理流程

1. **设备触发IRQ**（如键盘按下→IRQ1）。
2. **8259 PIC接收中断**，检查是否被屏蔽（OCW1）。
3. **PIC向CPU发送INT信号**，CPU响应后，PIC发送 **中断向量号**（如IRQ1 → `0x21`）。
4. **CPU查IDT**，跳转到对应的ISR。
5. **ISR处理完成后，发送EOI**（`0x20` 到PIC端口）。

6. **PIC允许后续中断。**

8259 PIC是传统设计，现代x86 CPU使用 **APIC（Advanced PIC）**：

启动定时器并打开中断

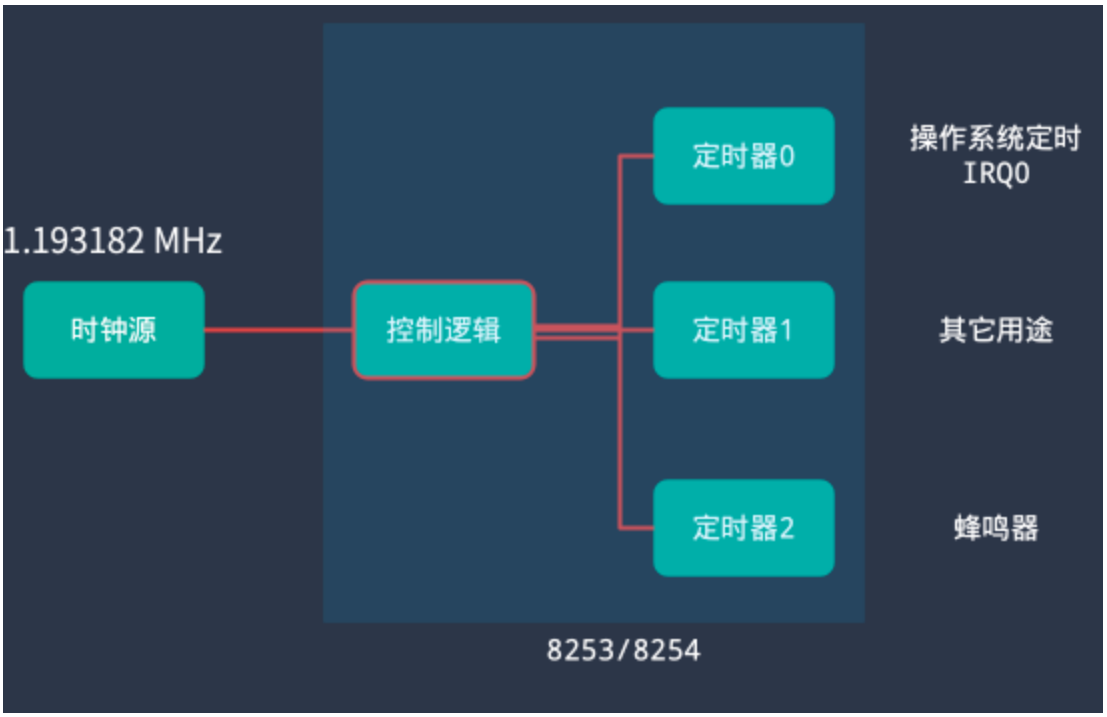
8253 PIT（Programmable Interval Timer）详解

4包含 **3个独立的16位计数器**（Counter 0-2），每个计数器可单独编程。
系统时钟：通过IRQ0提供定时中断

计数器端口地址

计数器	I/O端口（主PIT）	用途
0	`0x40`	系统定时器（连接IRQ0）
1	`0x41`	DRAM刷新（现代系统已弃用）
2	`0x42`	扬声器控制
控制字寄存器	`0x43`	配置计数器的工作模式

只需要将定时器设置成自动周期性触发中断即可



中断使能开关

中断控制受两部分控制，首先是8259中断控制器控制着外设的中断信号是否允许到达CPU，这项功能由其内部寄存器IMR控制。该控制器为8位，其中bit0控制该芯片的IRQ0，bit1控制IRQ1，依此类推。

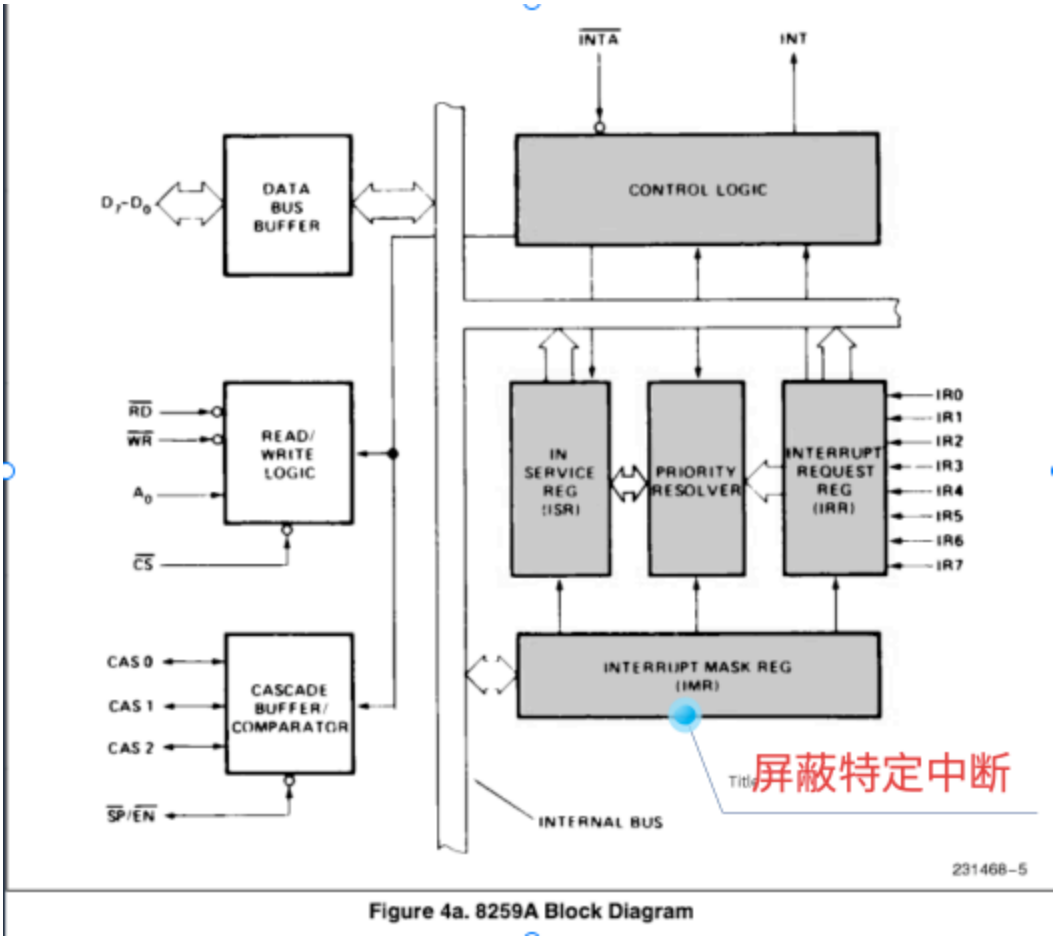


Figure 4a. 8259A Block Diagram

另外，cpu内部的EFLAGS标志寄存器中的IF位控制中CPU是否响应着任意来自8259的中断信号。如果为1，则允许，如果为0则禁止。