# 硬件抽象层

操作系统作为一种最底层的系统软件，所以在操作系统中很多地方，它所直接面对和操纵的是底层硬件，譬如芯片、设备控制器界面、特殊寄存器以及会涉及保护模式的数据结构，如段描述符、选择子、页目录和页表等等。针对这些硬件相关的操作时需要用到较多高级语言无法替代的特殊汇编指令，为了避免在代码中使用过多汇编语言，导致降低了代码的可读性，我们为此引入了硬件抽象层对这类汇编指令、控制芯片、保护模式中数据结构封装和抽象。

硬件抽象层是Unix V6++设计过程中添加的新的模块，相对于Unix V6并中没有对应的行为而言。设计该模块的目的是为了抽象操作系统中与体系结构相关的部分，在硬件与内核之间起着承上启下的作用。对下面的硬件而言，硬件抽象层封装对底层硬件操作的细节，对位于其上层的操作系统内核来说，硬件抽象层则向上提供了定义明确的接口，内核通过这些规定接口调用底层硬件的功能，具有意义的名字较汇编语言更可以清晰描述执行的操作，代码可读性较强。

硬件抽象层主要提供三部分功能：第一部分是对汇编语言的抽象；第二部分是对中断控制芯片，时钟芯片等控制芯片的抽象；第三部分是对象化I386体系结构中出现的数据结构。

## 汇编指令的封装

操作系统开发过程中存在某些必须用汇编指令，而无法用高级语言替代的地方。主要是一些涉及到芯片或设备的底层操作需要大量用到汇编代码，在高级语言中并没有直接对这些底层操作的支持，例如用于I/O端口输入输出操作系统的IN、OUT指令以及一些芯片的特权指令例如CLI、STI、LGDT、LIDT等等。

大多数高级语言提供了在高级语言代码中插入内联汇编语言的方法。gcc编译器支持使用\_\_asm\_\_关键字在代码中添加汇编代码。因为内联汇编是编译器提供的语法标准，但是与C++原有语法有较大的不同，而且内联汇编语法比较复杂，在C++代码中插入内联汇编会感觉比较突兀，给代码阅读带来较大的困难。出于提高代码可读性的目的，考虑使用函数或宏对内联汇编代码进行封装，在使用汇编时可以使用函数调用的风格执行汇编代码。

在Unix V6++中，主要有两个类与若干宏定义完成了汇编代码的封装，如图4.1所示。



#### 图3.1 汇编代码抽象类与宏定义

### X86Assembly类

X86Assembly类中抽象了一部分特殊的系统指令，选择的标准是这些特殊指令在我们的系统中被使用到，并且它们无法由高级语言代码经编译后生成。例如开关中断，加载GDT，IDT，TR寄存器等指令被封装在该类中。如果不对其封装则会导致需要直接在代码中嵌入内联汇编语句。在Unix V6++的实现中，使用静态函数的做法封装系统指令，放在X86Assembly类中(代码3.1)。

class X86Assembly

{

public:

//允许中断

static inline void STI()

{

\_\_asm\_\_ \_\_volatile\_\_("sti");

}

//屏蔽中断

static inline void CLI()

{

\_\_asm\_\_ \_\_volatile\_\_("cli");

}

//加载GDT指令

static inline void LGDT(unsigned short gdtr[3])

{

\_\_asm\_\_ \_\_volatile\_\_("lgdt %0"::"m" (\*gdtr));

}

};

#### 代码3.1 汇编指令的函数封装

由此对特殊系统指令的使用便转化为对静态函数的调用操作了，虽然有稍许效率上的损失，但大大增强了代码的易读性(代码3.2)。

X86Assembly::CLI();

X86Assembly::LGDT( (unsigned short\*)(&gdtr) );

#### 代码3.2 系统指令的调用

### IOPort类

IOPort类单独将对硬件端口的操作封装为一个类，该类在系统初始化模块，驱动程序中被大量的使用。本质上来说，IOPort类是对于输入输出IN、OUT指令的封装，in、OUT指令用于CPU和外设控制寄存器进行数据交换，IN指令是从外设的寄存器中读入数据到CPU中，OUT指令则是将CPU中的数据输出到外设的寄存器。根据每次交换数据宽度的不同，可以区分出每次交换一个字节(Byte)、一个字(Word)和一个双字(DWord)三种情况，分别使用进行不同宽度数据交换的in/out指令如：inb、inw、inl以及outb、outw、outl，为此需要对这三种情况区别对待。

class IOPort

{

public:

//从指定端口port读入8比特数据，放入data中作为返回值

static inline unsigned char InByte(unsigned short port)

{

unsigned char data;

\_\_asm\_\_ \_\_volatile\_\_("inb %%dx, %%al" : "=a" (data) : "d" (port) );

return data;

}

//从指定端口port读入16比特数据，放入data中作为返回值

static inline unsigned short InWord(unsigned short port)

{

unsigned short data;

\_\_asm\_\_ \_\_volatile\_\_("inw %%dx, %%ax" : "=a" (data) : "d" (port) );

return data;

}

//从指定端口port读入32比特数据，放入data中作为返回值

static inline unsigned int InDWord(unsigned short port)

{

unsigned int data;

\_\_asm\_\_ \_\_volatile\_\_("inl %%dx, %%eax" : "=a" (data) : "d" (port) );

return data;

}

}

#### 代码3.3 IOPort类封装三种的in指令

上面的代码给出了对in指令的三种不同数据宽度的函数封装。本质上，三种数据宽度的实现方式基本一样，都是根据传入参数port，读入该端口地址的数据到al/ax/eax寄存器，其实也就是局部变量data中，然后将data作为返回值传给调用者。

使用函数名分别为InByte、InWord、InDWord，这三个函数的参数都是unsigned short类型的16位端口地址；而差异体现在函数的返回值类型上，分别使用unsigned char、unsigned short和unsigned int进行一个字节(Byte)、一个字(Word)和一个双字(DWord)的数据交换。

//从指定端口port读入8比特数据

static inline unsigned char InByte(unsigned short port)

//从指定端口port读入16比特数据

static inline unsigned short InWord(unsigned short port)

//从指定端口port读入32比特数据

static inline unsigned int InDWord(unsigned short port)

#### 代码3.4 in指令封装的函数签名

对out指令的封装也是用同样的实现方法，这里仅给出三种不同数据宽度的out指令的封装函数签名。三个函数都接受port和data两个参数，返回值为void类型，区别在于data参数的数据类型不同(代码3.5)。

//将8比特数据data，写入到指定端口port中

static inline void OutByte(unsigned short port, unsigned char data)

//将16比特数据data，写入到指定端口port中

static inline void OutWord(unsigned short port, unsigned short data)

//将32比特数据data，写入到指定端口port中

static inline void OutDWord(unsigned short port, unsigned int data)

#### 代码3.5 out指令封装的函数签名

## 控制芯片的抽象

在操作系统初始化时，需要初始化8259A中断控制芯片，8254时钟芯片，在Unix V6++的实现中，将这些芯片抽象化为静态类，封装了对这些芯片进行的初始化与设置操作。使用上节介绍的IOPort类对I/O端口进行读写。

对芯片的封装将设置操作抽象为函数，而不是I/O端口的一系列读写操作，隐藏了真正设置的细节。用具体的名字表达操作的思路，使代码执行的思路更为清晰。例如，使用如下的代码 Chip8259A::IrqEnable(Chip8259A::IRQ\_TIMER)操作8259A芯片将会比直接读写8259A芯片控制寄存器的端口有更好的可读性。如图3.2所示，是这两个类的类图：



#### 图3.2 控制芯片类

其中Chip8259A::IrqEnable()与Chip8259A::IrqDisable()分别用于设置或清除中断屏蔽寄存器的某个值。8259A与8254芯片提供的功能远多余这两个类所暴露的接口，但在Unix V6++中，仅使用到了图中的这些功能。

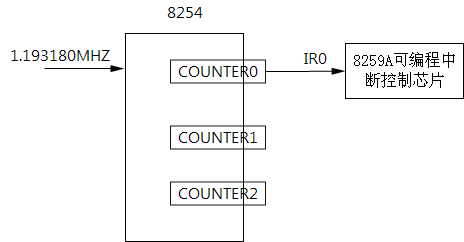
### Chip8254类

Chip8254类是对8253/8254可编程定时芯片(Programmable Interval Timer)进行的封装，隐藏对该芯片的操作设置细节，仅抽象出Chip8254::Init()接口进行时钟频率的初始化和设置，让内核通过该接口操作该芯片。

###### 8253/8254可编程定时芯片

首先，需要了解8253/8254可编程定时芯片的功能以及如何对其进行编程操作，这是我们能够针对它进行封装的必要前提。

在PC机中，该芯片的主要功能是用于产生连续的，固定时间间隔的周期性脉冲，该脉冲就是时钟中断的来源。一块8254芯片内部有3个完全独立的计数器，分别是COUNTER0, COUNTER1, COUNTER2，它们每个都可以工作在不同计数方式下。在Unix V6++中，我们只用到COUNTER0，它的输出引脚连接到可编程中断控制器8259A 芯片的IR0中断请求端，产生固定间隔的时钟中断。8254芯片的连接示意图如下所示(图3.3)。



#### 图3.3 8254芯片的连接示意图

在Unix V6++的内核初始化阶段，通过对8254芯片的编程可以设置时钟中断的间隔频率。对该芯片的编程可分为两个步骤：第一步，对芯片控制字(Control Word)进行编程，该Control Word结构如下所示(表3.1)。

|  |  |  |
| --- | --- | --- |
| 位 | 名称 | 含义 |
| 7 | SC1 | 计数器选择位(SC1,SC0)：00 = Counter 0; 01 = Counter 1;  10 = Counter 2; 11 = read-back命令; |
| 6 | SC0 |
| 5 | RW1 | 计数器读写操作方式：00 = 计数器锁存命令; 01 = 只读/写低字节;  10 =只读/写高字节; 11 = 先读/写低字节，再读/写高字节; |
| 4 | RW0 |
| 3 | M2 | M2 M1 M0选择计数器工作方式，Mode 0 – Mode 5共6种方式，用000 – 101表示; |
| 2 | M1 |
| 1 | M0 |
| 0 | BCD | 0表示二进制计数，1表示BCD码计数 |

#### 表3.1 8254芯片Control Word结构

第二步，写入计数值，写完后计数器立即开始计数，即产生时钟中断。关于计数值如何确定的问题，8254芯片的输入频率是1193180Hz，用于时钟中断通常选用计数方式3(Mode 3，即M2 M1 M0 = 011)。假设设置计数值为100，那么芯片输出的每秒1193180 / 100次时钟中断。

该芯片在PC机中占用I/O端口地址为0x40 ~ 0x43，分别对应COUNTER0, COUNTER1, COUNTER2和 Control Word。Unix V6++中仅用到COUNTER0，因而只需要用到端口地址0x40和0x43。

###### Chip8254类的实现

具体到Chip8254类的实现上来说，Chip8254类中仅抽象出Chip8254::Init()一个静态函数用于初始化和设置时钟频率，虽然该芯片的能力不止于此，但对于我们的内核来说已经足够用了。

void Chip8254::Init(int Ticks /\*= 60\*/)

{

if(Ticks <= 0)

return;

//向控制字端口43H，写入命令值

IOPort::OutByte(CTRLWRD\_PORT, CTRLCMD\_SEL0 | CTRLCMD\_MODE3 | CTRLCMD\_RW );

//向计数器0写入计数值，先写低8位

IOPort::OutByte(CNT0\_PORT, (INPUT\_FREQ / Ticks) % 256 );

//然后写计数值的高8位，端口地址不变

IOPort::OutByte(CNT0\_PORT, (INPUT\_FREQ / Ticks) / 256 );

}

#### 代码3.6 Chip8254::Init()函数的实现

Chip8254::Init()函数的实现中使用了很多常量的定义，如CTRLWRD\_PORT、CNT0\_PORT、CTRLCMD\_SEL0等，这些都是依据8254芯片的硬件规范使用有意义的名称定义的常量，这样可以降低编中可能的错误。这些常量的定义在下面给出，常量的定义是有选择的，譬如我们只用到COUNTER0和Control Word的端口地址0x40和0x43，因而没有定义关于COUNTER1和COUNTER2的端口地址(代码3.7)：

const unsigned int INPUT\_FREQ = 1193180; /\*芯片输入频率为1.193180MHz\*/

const unsigned short CNT0\_PORT = 0x40; /\*COUNTER 0的端口地址40H\*/

const unsigned short CTRLWRD\_PORT = 0x43; /\*控制字端口地址43H\*/

const unsigned char CTRLCMD\_SEL0 = 0x00; /\*选择计数器0\*/

const unsigned char CTRLCMD\_MODE3 = 0x06; /\*计数模式: Mode 3\*/

const unsigned char CTRLCMD\_RW = 0x30; /\*计数值读写: 先低字节,后高字节\*/

#### 代码3.7 8254芯片相关常量定义

在Chip8254::Init()函数的实现中，第一步写入Control Word，对应端口地址CTRLWRD\_PORT = 0x43，根据Control Word的定义，它的值由三部分构成：CTRLCMD\_SEL0表示选择计数器0，对应Control Word的Bit (7 ~ 6)为0；然后计数方式选择Mode3，需要设置Control Word的Bit (3 ~ 1)为011，因此定义CTRLCMD\_MODE3值为0x60；最后计数值读写方式为先低字节,后高字节，需设置Control Word的Bit (5 ~ 4)为11，因此CTRLCMD\_RW值定义为0x30。通过按位或操作“|”将三者相结合，便是Chip8254::Init()函数中(CTRLCMD\_SEL0 | CTRLCMD\_MODE3 | CTRLCMD\_RW)的由来，三者按位或运算之后实际写入Control Word的结果是0x36。

第二步则是先端口地址0x40的计数器0写入计数值，按照先低字节，后高字节的顺序。初始化为每秒60次时钟中断，所以写入计数值1193180 / 60即可。至此，8254时钟芯片初始化完成，以每秒60次的频率产生时钟中断，如果需要改变每秒产生时钟中断的次数，也只需要再次调用Chip8254::Init()函数，传递要设置的次数作为参数即可。

### Chip8259A类

Chip8259A类是对8259A可编程中断控制芯片(Programmable Interrupt Controller)进行的封装，该类对内核隐藏初始化和设置细节，通过抽象出Chip8259A::Init()进行对芯片的初始化，以及Chip8259A::IrqEnable()与Chip8259A::IrqDisable()两个函数用于设置或清除中断屏蔽寄存器的某个比特位，允许相应中断请求进入。

###### 8259A可编程中断控制芯片

8259A是系统中硬件中断的管理者。它帮助外设转达对CPU提出的中断请求。8259A作为外设和CPU之间的中间层，外设提出中断请求都是发送给8259A，由它来通知CPU中断请求的进入。

8259A可编程中断控制芯片包含一系列内部寄存器，我们通过in/out指令设置这些寄存器来初始化和控制8259A芯片的行为。该芯片可供编程设置的寄存器分为两类：

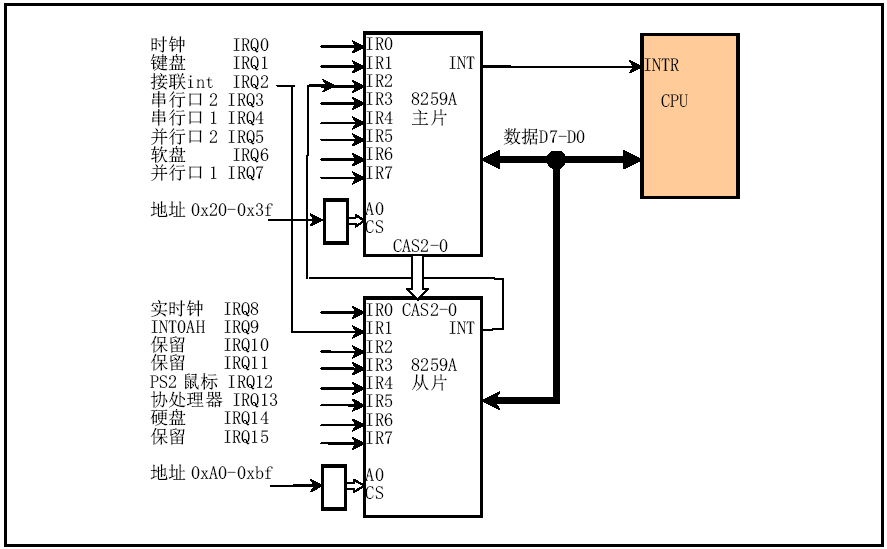
1. **Initialization Command Words (ICW)**：用于初始化8259A工作模式的命令字，须在8259A开始正常工作，即接收中断之前写入。
2. **Operation Command Words (OCW)**：在8259A的工作期间任何时候，OCW用来监测8259A的状态和初始化设定的工作模式。

后面会具体解释不同ICW和OCW的作用以及如何对它们编程设置。

###### PC中的8259A中断管理机构

8259A存在的必要性是因为它可以管理8个中断源，对应中断请求输入引脚(IR0~IR7)，可以接收来自不同外设提出的中断请求；而CPU不行，因为CPU用于中断的只有INTR，/INTA和NMI引脚，如果各种外设的中断请求引脚都与CPU的INTR相连接，CPU收到一个中断时，它将无从分辨究竟是谁发出的。

PC机中采用了主从2片级联的8259A(Master和Slave 8259A)，共可管理15级中断向量，两片8259A的IR0~IR7引脚连接到了各种外设的中断请求引脚上(图3.4)。



#### 图3.4 PC系统中的8259A中断管理机构

外设将中断请求发送至8259A，然后由8259A通知CPU有中断进入，CPU响应来自8259A的中断信号之后需要知道具体是哪个设备发出的中断请求，这就需要由8259A芯片将中断向量号告诉CPU，CPU以中断向量号为索引查找中断描述符表，取得中断处理程序的入口地址，转而执行该中断处理程序。**换言之，8259A中断管理机构就是从硬件上建立外设中断请求与中断向量号的对应关系。**

###### 对8259A芯片编程

系统中Master-8259A的端口地址为0x20和0x21， Slave-8259A的端口地址为0xA0和0xA1。ICW寄存器一共4个，ICW1 ~ ICW4。初始化阶段只涉及针对ICW的编程，对OCW的操作需要在芯片初始化完成，进入正常工作状态时进行。

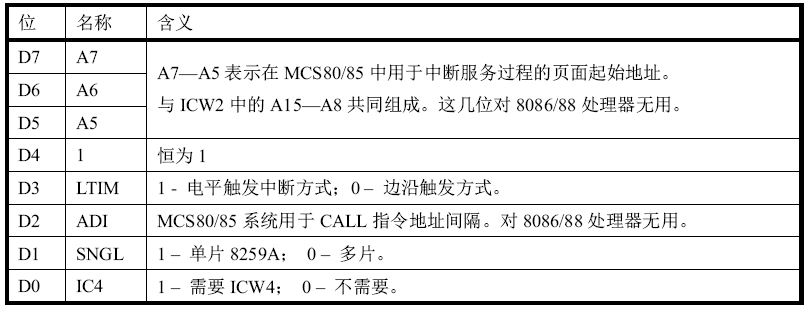
对8259A初始化ICW必须遵照一定的规则，即按照特定的顺序来设置，不可随意打乱。其规则如下：

* 要求对ICW1和ICW2这两个命令字的设置是一定要的。
* 设置主片ICW1的端口地址使用0x20，设置主片ICW2的端口地址0x21。
* 设置从片ICW1的端口地址使用0xA0，设置从片ICW2的端口地址0xA1。
* 如果系统中多片8259A硬件上是级联的，则需要对ICW3初始化，且紧跟在ICW2初始化之后，使用与ICW2相同端口地址(主片0x21，从片0xA1)。
* 对于是否需要初始化ICW4根据先前ICW1中Bit(0)的取值，为1则要初始化ICW4；反之则不要。

 对系统中任何一片8259A都要遵照ICW1~ICW4的顺序来初始化。不可以只对Master的ICW1~ICW4初始化，然后对Slave的ICW3~ICW4初始化，跳过了Slave的ICW1~ICW2，或者认为Slave的ICW1~ICW2会自动获取Master的ICW1~ICW2，往里面初始化写全0，这都是不行的。

从ICW1~ICW4的初始化顺序前面已经详细说明了。那么究竟应该向ICW写入什么样的值才可以让它正常工作呢？

1. **对ICW1编程**



#### 表3.2 命令字ICW1格式

由表3.2可知，ICW1是一个8位寄存器，其中D7~D5比特位跟Intel CPU无关，将这3个比特位写0即可。D4位要求恒为1。D3位取0，表示采用边沿触发方式。D2为也是无关位，取0。D1为0，表示系统中多片级联。D0为1，表示需要对ICW4初始化。**结论是：最后需要写入ICW1的值为 (0001 0001)B，或者0x11。**主片和从片都应是这个值。

1. **对ICW2编程**

对ICW2的初始化用于确定该8259A芯片的8个中断输入引脚(IR0~IR7)对应中断向量号范围是多少。所谓中断向量号范围包含8个中断向量号，分配给连接到8259A芯片IR0~IR7引脚的8个外设产生的中断，即每种外设产生中断都会分配得到一个中断向量号。

Intel CPU支持最多256个中断，包括软件的和硬件的中断。每个中断是由一个0 ~ 255之间的数字来标识，这个数字就是中断向量号。在Intel CPU中，第0 ~ 31号中断(前32个，即0x00~0x1f)由Intel固定设定或保留用。用于处理CPU执行指令时产生的异常。从32 ~ 255(0x20 ~ 0xff)号中断由用户自行分配。

我们需要为PC系统中两片8259A支持的15级中断，每种中断分配一个中断向量号，就只能从32 ~ 255(0x20 ~ 0xff)这个范围内分配。

关于对ICW2如何赋值的问题。ICW2的值用于设置8259A送出的中断向量号的高5位。有了高5位的值，8259A自动为IR0~IR7进入的中断信号添加后缀，范围从(000~111)B，组成8位的中断向量号。其中来自IR0的中断加后缀000，来自IR7的中断加后缀111。

也就是说，这高5位的值确定的是这片8259A对应的中断向量号范围的基数。假定初始化ICW2的值为(0010 0xxx)B，低3位写入0或1都会被忽略。那么连接到这片8259A的外设分配到的中断向量号为：

0020 0**xxx**

或

0010 0**000** = 0x20 IR0对应中断

到

0010 0**111** = 0x27 IR7对应中断

这正是Unix V6++对系统中Master的0 ~ 7级中断请求分配的中断向量号，内核初始化时对Master的ICW2写入值为0x20。而对Slave的ICW2写入值为0x28，使得从片8 ~ 15级中断请求分配到中断向量号为0x28 ~ 0x2f。当然也可以选取其它的中断向量号范围，譬如主片0x30 ~ 0x37，从片0x38 ~ 0x3f(或者从片0x40 ~ 0x47，不用紧接着主片的范围之后)。

对于外设来说，中断向量号也不是随意分配给它的，对于连接到同一片8259A上的外设说，它们分配得到的中断向量号是一个基数+各自所在的中断请求引脚(IR0~IR7)。一旦给定Master的IR0(时钟中断)分配中断向量号为0x20，那么Master的IR1(键盘中断)获得的中断向量号必定是0x21。而中断向量号作为CPU查找IDT的索引值，也就意味着时钟中断的处理程序入口将占据IDT中第0x20项，而键盘中断的将紧随其后占据IDT中第0x21项的位置，也就是说，软件的设置要和硬件相吻合，才可正常发挥作用。

1. **对ICW3编程**

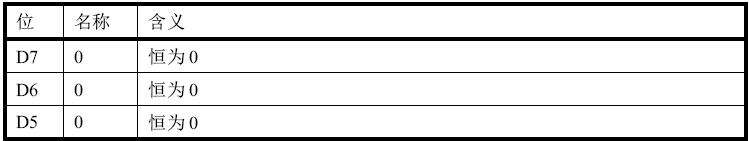
当系统工作在级联方式下，需要对ICW3进行初始化，并且Master和Slave都要进行ICW3的初始化。ICW3表示Slave的中断输出级联到Master的IR0~IR7中的哪一个。Master和Slave需要分别讨论。

* 对于Slave来说，其ICW3中的低三位存放该Slave连接到Master的IR引脚编号。D2 ~ D0这3个比特的取值范围从000 ~ 111分别对应Slave连接到IR0~IR7之一的某个引脚。Slave的ICW3的高5位必须全部为0。
* 对于Master来说，其ICW3中任一比特位为1，则表示对应IR7~IR0引脚上连接的是一片Slave 8259A，而非一般外设的中断引脚。

系统中的Slave中断输出引脚级联到Master的IR2。则应该在Slave的ICW3中应该存放0000 0010(最低3个比特位值为010，表示连接至Master的IR2引脚)；而在Master的ICW3中将对应D2为置1，其余为0，所以值为0000 0100。因此结论是：Slave的ICW3写入0000 0**010** = 0x02；Master的ICW3写入0000 0**1**00 = 0x04。

1. **对ICW4编程**

设置ICW1的IC4比特位意味着需要初始化ICW4，并且紧跟在ICW3之后。





#### 表3.3 命令字ICW4格式

在Unix V6++中，无论对于Master和Slave的初始化，存入ICW4的值都是0x01。表示SFNM=0 普通全嵌套模式；BUF=0，M/S=0非缓冲；AEOI=0非自动结束方式(需要在中断服务程序结尾向8259A发送一个中断结束命令，复位中断服务寄存器中的比特位)；PM=1用于Intel CPU。

1. **对OCW1编程**

OCW1用于控制8259A芯片有选择地屏蔽和接收来自不同引脚的外设中断。OCW1是一个8位的寄存器，其中某位为1表示屏蔽对应中断请求，为0则允许对应的中断请求。对Master和Slave的OCW1设置需分别通过端口地址0x21和0xA1。

1. **对OCW2编程**

OCW2用于发送EOI命令，至于该命令字的其余比特位功能则并没有用到。EOI是指当CPU完成当前中断的处理，向8259A芯片发送EOI告知处理结束，以允许后续中断进入。对Master和Slave的OCW2设置需分别通过端口地址0x20和0xA0。

###### Chip8259A类的实现

Chip8259A::Init()函数在系统启动时执行8259A芯片的初始化工作，借助由IOPort类封装的端口操作。从下面的代码可以看出，初始化确定了主、从片对应的中断向量号范围是0x20 ~ 0x27以及0x28 ~ 0x2f；级联情况是主片的IR2引脚连接了从片；确定了主、从片的中断模式，以及屏蔽所有中断，让中断屏蔽位处于一个已知状态，这在保护模式下中断处理函数准备就绪之前是必须的(代码3.8)。

void Chip8259A::Init()

{

// 编程ICW1确定主、从片的工作模式

IOPort::OutByte(MASTER\_IO\_PORT\_1, 0x11);

IOPort::OutByte(SLAVE\_IO\_PORT\_1, 0x11);

//编程ICW2确定主、从片对应的中断向量号

IOPort::OutByte(MASTER\_IO\_PORT\_2, MASTER\_IRQ\_START);

IOPort::OutByte(SLAVE\_IO\_PORT\_2, SLAVE\_IRQ\_START);

//编程ICW3确定主、从片级联的引脚情况

/\*TO MASTER: 主片的IR2引脚连接了从片,所以IR2对应比特位为1 。 \*/

IOPort::OutByte(MASTER\_IO\_PORT\_2, 1 << IRQ\_SLAVE); /\*0x04 = 0000 0100 b\*/

/\*TO SLAVE: 从片连接到主片的IR2引脚, 所以要使得低三位是 010\*/

IOPort::OutByte(SLAVE\_IO\_PORT\_2, IRQ\_SLAVE);

//编程ICW4确定主、从片的中断执行模式

//普通全嵌套模式、正常EOI、Intel X86模式

IOPort::OutByte(MASTER\_IO\_PORT\_2, 0x01);

IOPort::OutByte(SLAVE\_IO\_PORT\_2, 0x01);

/\*初始化之后先屏蔽所有中断,写OCW1寄存器,

目地是让中断屏蔽位处于一个已知状态下\*/

IOPort::OutByte(MASTER\_IO\_PORT\_2, MASK\_ALL);

IOPort::OutByte(SLAVE\_IO\_PORT\_2, MASK\_ALL);

}

#### 代码3.8 Chip8259A::Init()函数的实现

同样，对于代码中出现的许多常量，我们在下面给出相关定义，其中包括主、从8259A芯片的端口地址、起始中断向量号以及时钟、键盘、磁盘对应的中断请求号(代码3.9)。

/\*主片(Master)的IO端口地址\*/

static const unsigned short MASTER\_IO\_PORT\_1 = 0x20;

static const unsigned short MASTER\_IO\_PORT\_2 = 0x21;

/\*从片(Slave)的IO端口地址\*/

static const unsigned short SLAVE\_IO\_PORT\_1 = 0xA0;

static const unsigned short SLAVE\_IO\_PORT\_2 = 0xA1;

static const unsigned char MASTER\_IRQ\_START = 0x20; /\*主片的起始中断向量号\*/

static const unsigned char MASK\_ALL = 0xFF; /\*屏蔽状态字, 屏蔽所有的中断请求\*/

static const unsigned int IRQ\_TIMER = 0; /\* 时钟中断(IRQ0)发送到IR0引脚\*/

static const unsigned int IRQ\_KBD = 1; /\*键盘中断(IRQ1)发送到IR1引脚\*/

static const unsigned int IRQ\_SLAVE = 2; /\*Slave发出的中断,发送到主片的IR2\*/

static const unsigned int IRQ\_IDE = 14; /\*硬盘中断(IRQ14)发送到从片的IR6\*/

#### 代码3.9 Chip8259A类相关常量定义

此外，Chip8259A类还包含了Chip8259A::IrqEnable()与Chip8259A::IrqDisable()两个函数接口，用于设置或清除某个中断屏蔽位来控制是否允许相关中断请求进入。这两个函数都是通过对OCW1命令字编程实现允许或屏蔽中断功能的。

Chip8259A::IrqEnable()为开启中断函数，通过设置8259A的中断屏蔽寄存器中相应的中断屏蔽位清0，从而允许某个特定外设的中断。输入参数是该外设的中断请求号，如IRQ\_TIMER = 0、IRQ\_KBD = 1等常量值。

Chip8259A::IrqDisable()为屏蔽中断函数，执行与Chip8259A::IrqEnable()正好相反的功能。通过将相应的中断屏蔽位置1，从而屏蔽来自某个特定外设的中断。输入参数同样是中断请求号。

在使用静态函数实现对8259A中断控制芯片初始化、允许和屏蔽特定中断请求的封装之后，我们就可以用函数调用来控制芯片的行为了。需要注意的是，Chip8259A::IrqEnable()和Chip8259A::IrqDisable()函数每次只能设置或清除一个中断屏蔽位，也就是允许或屏蔽一种中断(代码3.10)。

Chip8259A::Init(); /\* 初始化8259A中断控制芯片 \*/

Chip8259A::IrqEnable(Chip8259A::IRQ\_TIMER); /\* 允许时钟中断进入 \*/

Chip8259A::IrqDisable(Chip8259A::IRQ\_TIMER); /\* 屏蔽时钟中断\*/

/\* 允许磁盘中断，需开启从片到主片的级联，即要同时Enable IRQ\_IDE 和IRQ\_SLAVE \*/

Chip8259A::IrqEnable(Chip8259A::IRQ\_IDE);

Chip8259A::IrqEnable(Chip8259A::IRQ\_SLAVE);

#### 代码3.10 Chip8259A类成员函数的调用

最后要说的是这样封装会带来意料之外的好处。在开发过程某些环节，可能会要单独屏蔽时钟中断后观察程序运行效果，有了IrqEnable ()和IrqDisable()函数只需简单调用一下即可，而不必同时在多处插入一连串端口操作指令，那样既麻烦又容易导致错误。

## I386体系结构抽象

I386体系结构的主要是将保护模式下I386体系结构的内存数据结构对象化，保护模式下的系统，需要初始化必要的内存数据结构，包括GDT表，IDT表，页表等。

### 全局描述符表GDT的抽象

GDT表是由8字节大小的不同类型的段描述符组成的，在物理内存中连续排列。GDT类包含了一个512字节大小的SegmentDescriptor类型的结构体数组(图3.5)。该结构体为8字节大小，使用gcc编译器关键字\_\_attribute\_\_((packed))修饰，使的该结构体中各字段之间不会插入“空隙”，严格保证GDT表中每个SegmentDescriptor的大小8字节。



#### 图3.5 全局段描述符表GDT表的抽象

struct SegmentDescriptor

{

unsigned short m\_Low16BitsSegmentLimit : 16;

unsigned short m\_Low16BitsBaseAddress : 16;

……

public:

void SetBaseAddress(unsigned int baseAddress);

void SetSegmentLimit(unsigned int segmentLimit);

}\_\_attribute\_\_((packed));

#### 代码3.11 SegmentDescriptor结构体定义

SegmentDescriptor结构体的定义中还包含了用于设置段描述符基地址的SetBaseAddress()函数和设置段限长的SetSegmentLimit()函数。之所以提供这两个接口是因为在段描述符的基地址和段限长在Intel规定的描述符8字节的格式中被分割成了多段，让调用者函数负责分割基地址和段限长并正确设置描述符的各个字段容易引入错误，因此让该结构体自身提供设置基地址和段限长的函数接口。

### 中断描述符表IDT的抽象

出于和抽象GDT一样的目的，Unix V6++中采用同样的方式对中断描述符表IDT进行抽象(图3.6)。



#### 图3.6中断描述符表IDT的抽象

主要区别在于存储在IDT表中的数据项为门描述符（GateDescriptor）。在IDT类中，根据门描述符类型不同，使用不同的函数设置中断处理函数。如中断处理函数设置需要使用SetInterruptGate()，该函数将会设置相应的Type数据项并将相应的处理函数handler保存在在门描述符中(代码3.12)。

class IDT

{

public:

void SetInterruptGate(int Number, unsigned int Handler);

void SetTrapGate(int Number, unsigned int Handler);

private:

GateDescriptor m\_Descriptor[256]; /\* 256\*8 Bytes \*/

}

#### 代码3.12 IDT类的实现

IDT::SetInterruptGate()函数用于根据中断号、中断处理程序设置IDT表中对应的中断门描述符，其中Number参数为中断号，Handler为对应中断处理程序的入口地址。同样的，IDT::SetTrapGate ()函数用于根据中断号、中断处理程序设置IDT表中对应的陷入门描述符。

### 页目录与页表的抽象

页目录表与页表都是分配在物理内存页上的数组，页目录表中存放页目录表项PageDirectoryEntry页目录表中存放页表项PageTableEntry。



#### 图3.7 PageDirectory与PageTable类结构

这两个数据结构并无特殊的成员函数，与SegmentDescriptor结构体的定义一样，PageDirectoryEntry和PageTableEntry都使用gcc编译器关键字\_\_attribute\_\_((packed))修饰，严格保证每个表项大小4字节。