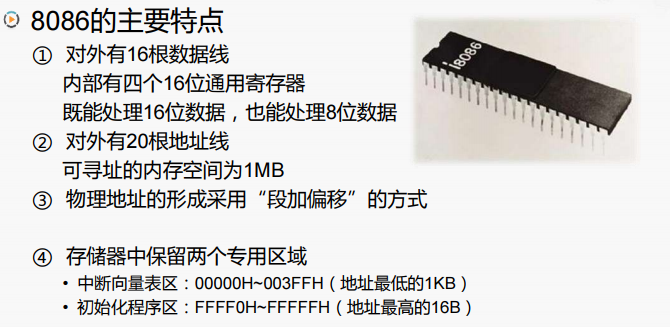
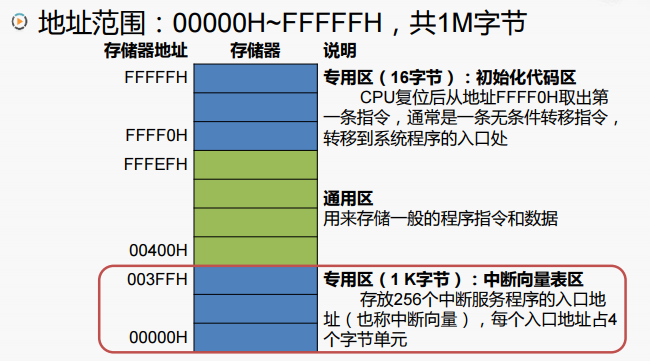
**中断与异常**

**一、中断向量表**

**1、8086中的中断向量表**

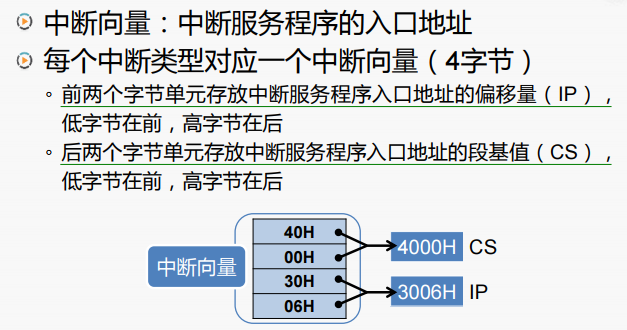


8086实模式下的存储器地址空间：



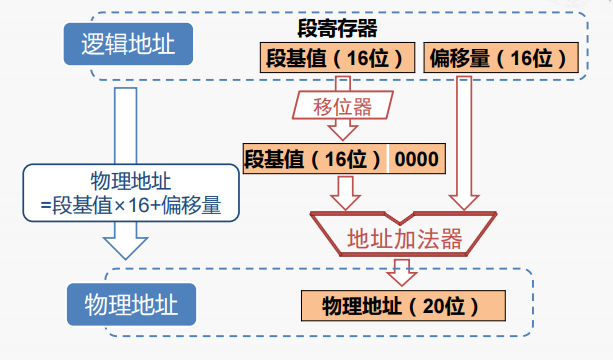
其中由于8086中的内存地址线共有20条，而8086CPU机器字长为16位，所以为了得到内存地址必须采用段加偏移的方式获得内存地址。且内存中每一个内存单元只有8bits

什么是中断向量？

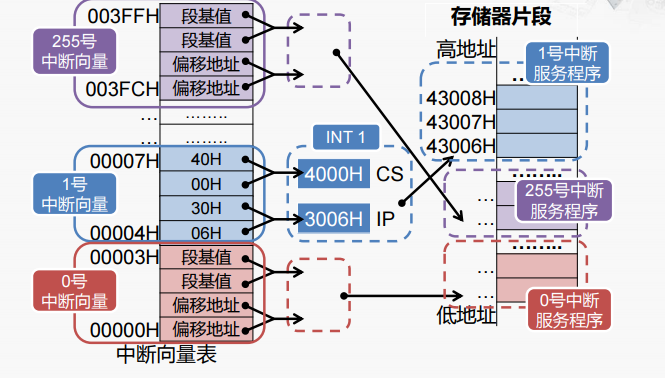


其中中断向量中的段基址存放在CS代码段寄存器中，偏移量存放在指令指针寄存器中，上述两个寄存器都是16位的（与8086的机器字长相同）

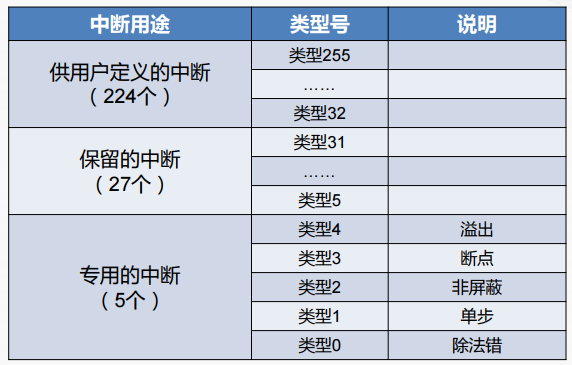
CPU通过如下段加偏移的方式将逻辑地址（逻辑地址就是段基址：偏移量的组合形式。例如上面的中断向量的逻辑地址就是： 4000H：3006H；注意8086的内存存放数据使用的是小端对齐）转化为中断服务程序的物理地址：



在8086中，中断向量表整齐的存放在内存地址为0起始区域当中，总共占有1KB，共有256个中断向量，每一个中断向量含有4个字节，当中断发生的时候，CPU使用中断类型码（0~255）去查找具体的中断向量。



不过值得注意的是，与中断向量依次整齐的存放在内存中不同，中断服务程序的存放到底放在通用区的哪并没有明确的要求。



此为8086的中断向量表

综上我们可以知道8086发生中断时CPU是如何访问中断服务程序的：

1、CPU发生中断，获得中断类型码，将中断类型码\*4获得中断向量在内存中的地址

2、访问中断向量表中的中断向量获得中断服务程序的段基地址和偏移量，并将上面两个内容存放到CS代码段寄存器和IP指令指针寄存器中

3、CPU根据CS和IP中的内容（逻辑地址）计算出中断服务程序的入口地址，然后在下一个时钟周期从该地址取指执行中断服务程序

中断向量表是如何存放进内存当中呢？

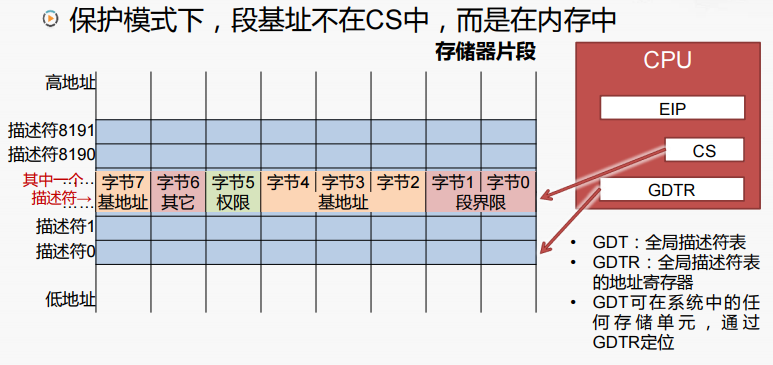
CPU运行在实模式下，CPU复位之后就会去1MB的内存地址空间最高16字节去取第一条指令，然后这个地址就会被南北桥芯片引导到BIOS芯片，CPU执行BIOS芯片当中的指令对主板上的各个设备进行基本的配置，其中一项工作就是在主存地址为0的地方开始去构建出中断向量表，所谓构建中断向量表就是在主存中最低的1KB的地方填写那些中断向量。构建好中断向量表以及准备中断服务程序后，CPU遇到中断时就可以访问主存中的中断向量表并调用中断服务程序。

**2、IA-32中的中断向量表**

在IA-32的保护模式下，由于IP指令指针寄存器被拓展为32位，而32位已经能够和内存地址空间一一对应。而且此时段基地址也不在CS寄存器当中，而是存放在内存当中

在IA-32中，内存中存放着一张描述符表，这张表共有8192个表项，每一个表项（被称为描述符）由8个字节构成，这8个字节的第2、3、4、7字节为基地址（这个地址对应了实模式下保存在CS中的段基地址）；字节0、1为段界限表明了这个段共有多长，字节5是权限表明了这个段是否可读/写。由于CS寄存器仍然只有16位，共可寻址2^16个存储单元，而由于每一个描述符共有8个字节，所以CS共可寻址2^13（即8192）个描述符。

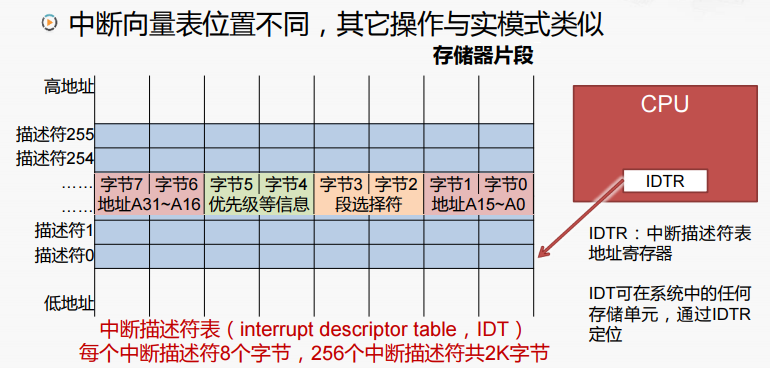
由于在IA-32中，描述符表并不是从内存为0的地方开始存放，所以另需一个寄存器来保存描述符表的起始地址，即有了全局描述符表地址寄存器GDTR。



那么GDTR中的内容又是从何而来的呢？

因为x86CPU在启动时都会先进入实模式，在实模式下，就会在内存的某一个地方先把描述符表先填好，然后再将这张表的起始地址填到GDTR中（GDTR在CPU内部）。

所以在保护模式下，CPU每一次访问主存都需要先用CS与GDTR中的内容得到一个地址，用这个地址访问存储器，然后取出描述符，再把描述符中的4个字节的及地址提取出来，然后和EIP指令指针寄存器中的内容组合，得到要访问存储器中的地址最后访问主存取得指令或数据。由于保护模式下，每一次访问存储器都需要这个过程，所以访问中断向量表也是如此。



与实模式不同，IA-32的中断向量表的位置也发生了改变，在保护模式下，中断向量表不在存放在起始地址为0开始的那块区域，而是存放在内存的任意位置，而且他的名字也发生了变化，称为中断描述符表IDT。IDT共有256个描述符表项，共占用2K字节，每一个描述表项共有8个字节，其中字节0、1、6、7为32位的地址，字节2、3位段选择符，当发生中断时CPU还是根据中断类型码查找中断描述符表，由于起始地址不是0，所以CPU必须先知道中断描述符表的起始地址，这个地址存放在中断描述符表地址寄存器IDTR当中（这个地址也需要系统初始化软件在建立中断描述符表后填写进IDTR）。

此时CPU根据IDTR和中断类型码（IDTR+中断类型码\*8）得到中断描述符的地址，然后把这个描述符取出后，将段选择符16位放入到CS中，而地址32位放入到EIP指令指针寄存器当中。如果在实模式下，CPU会在下一个时钟周期从新的地址开始取指，而在保护模式下，会用CS和GDTR配合去内存中找出对应的段基地址，然后和EIP组合才能得到对应的地址（而这个地址才是我们需要调用中断服务程序的入口地址），这样下一步才会从这个地址取回指令执行中断服务程序。

综上我们可以知道在IA-32中发生中断时我们是如何找到终端服务子程序的入口地址的：

1、产生中断获得中断类型码，由中断类型码\*8+IDTR得到地址访问中断描述符表

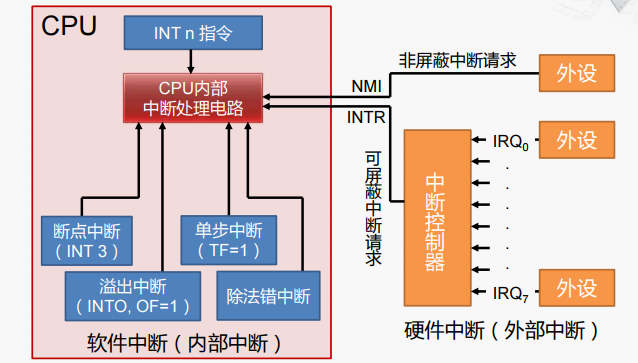
2、从中断描述符表中取出中断描述符，将8个字节当中的32位地址---->EIP指令指针寄存器，16位段选择符------>CS代码段寄存器

3、使用CS+GDTR全局描述符地址寄存器得到的地址访问全局描述符表，获得32位段基地址

4、使用段基地址+EIP指令指针寄存器当中的内容组合得到最终的中断服务程序的入口地址

**二、中断处理过程**

中断的检测电路大致组成（在内部有内部中断检测电路，在外部有外部中断检测电路，而且外部中断又可分为非屏蔽中断和可屏蔽中断）



中断处理过程：

**1、中断请求**

对于外部中断CPU在当前指令最后一个时钟周期查询中断请求信号

**2、中断响应**

获得中断类型码（对于内部中断和NMI中断不需要从数据总线上读取中断类型码，但是可屏蔽中断INTR需要，这个中断类型码由中断控制器提供）并将状态标志寄存器FLAGS进行压栈保护

**3、关中断**

CPU关闭中断响应，即不在接受其他外部中断请求（清除中断标志位IF和单步跟踪标志位TF）

**4、保护断点**

将发生中断处的指令地址CS：IP压入堆栈，以使中断处理完后能正确返回

**5、识别中断源**

CPU识别中断的来源，确定中断向量号，从而找到相应的中断服务程序的入口地址

**6、保护现场**

将断点处各寄存器中的内容压入堆栈进行保护

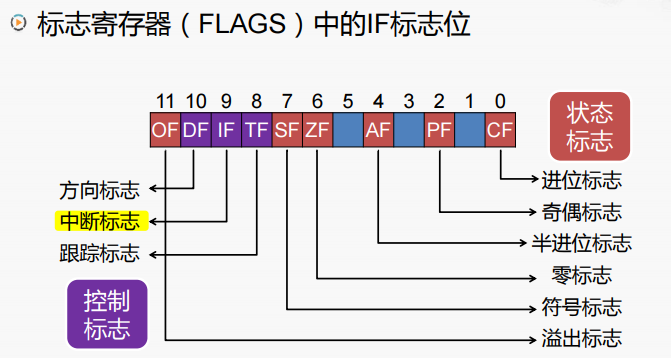
**7、执行中断服务程序**

转到中断服务程序入口开始执行，可在适当时刻重新开放中断，以便允许响应较高优先级的外部中断

**8、恢复现场并返回**

把保护现场时压入堆栈的信息弹回到原寄存器（恢复中断），然后执行中断返回指令，从而返回主程序继续运行

上面步骤中涉及到的相关硬件内容和软件内容：  
·标志寄存器FLAGS



其中中断标志位IF控制对可屏蔽中断的响应，但不对非屏蔽中断和内部中断起作用：

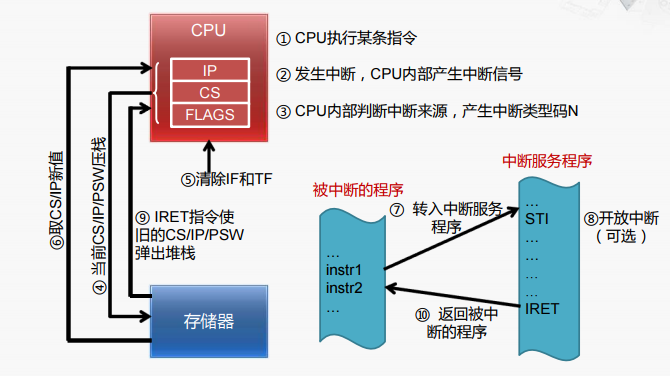
·若IF=1，这允许CPU响应可屏蔽中断请求，否则不允许CPU新颖可屏蔽中断请求

·可以使用相关指令将IF标志位进行设置：STI——将中断标志IF置1；CLI——将中断标志位IF清零

·中断返回指令IRET

它会在中断返回前，将从栈顶弹出3个字，分别送入IP、CS和标志寄存器FLAGS中；它放在中断服务程序的末尾，按中断调用时逆序恢复现场，返回程序发生中断处继续执行

中断处理过程实例（内部中断）



在上面的处理过程中，一般前面３项（关中断、保护断点、识别中断源）由处理中断的硬件电路完成；而后面３项（保护现场、执行中断服务程序、恢复现场并返回）通常由软件（中断服务程序）完成

**三、内部中断的分类**

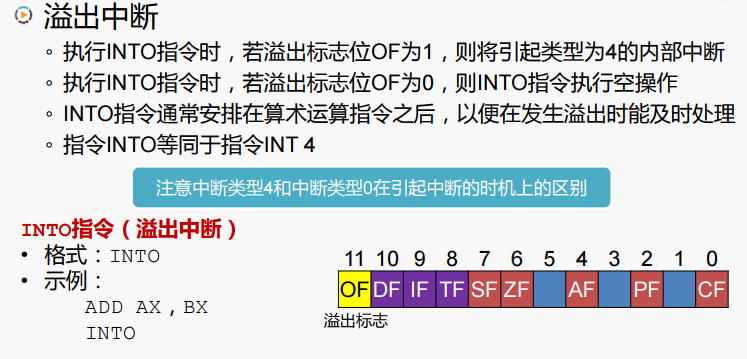
X86内部中断可以分为类型0：除法错误中断；类型1：单步中断；类型3：断点中断；类型4：溢出中断



1、除法错误中断：

在执行除法指令后，若所得的商超出了目标寄存器所能表示的范围，比如用数值0作除数，则CPU立即产生一个类型0中断

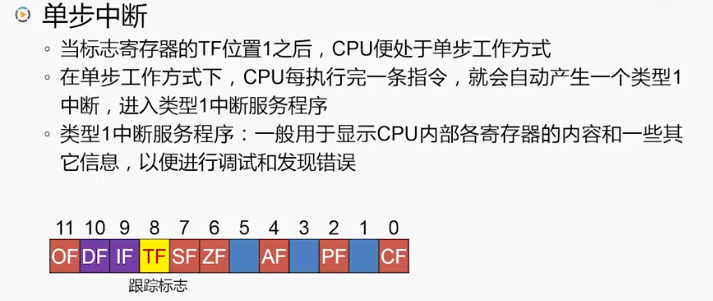
2、溢出中断：



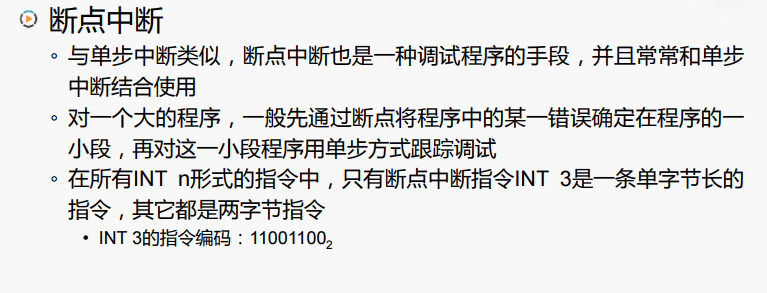
中断类型4和中断类型0在中断的时机上有区别，虽然它们都在检查运算中出现的异常情况，但是中断类型0实在除法指令执行后立即发生的，而中断类型4则是要在编程时加入INTO进行主动检查，因为很多时候这样的加法溢出不需要处理，若果每一个都要引发中断反而会影响程序的性能，所以指令系统设置时将是否需要检查溢出错误的情况交给程序员去进行判断

与上面的除法错误中断和溢出中断不同，中断类型1单步中断和中断类型3断点中断都是用于调试程序时使用的

3、单步中断：



4、断点中断：



断点中断的使用过程：

1）设置断点：

·用断点中断指令INT 3代替用户程序的原有指令

·保存用户程序的原有指令

2）发生断点：

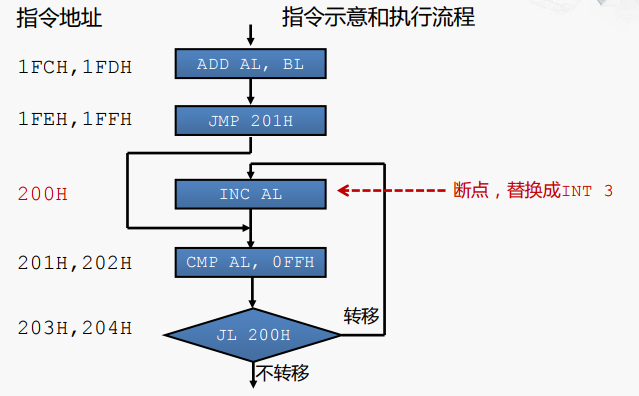
·用户程序运行到断点时，执行INT 3，进入断点中断服务程序，通常显示CPU各寄存器的值等

3）恢复执行：

·断点中断服务程序返回前，恢复用户程序的原有指令，并将IP指令指针寄存器的值-1（CS和IP的作用起到的就是MIPS中的PC程序计数器的作用，因为INT 3这条指令代替了原有的指令并作出了执行，使得IP寄存器必然自动的+1，为了在结束断点中断的时恢复执行原有指令，必须将IP寄存器-1去执行原有程序中的那一条被替换的指令）

·中断返回后，CPU从断点处继续执行

具体过程如图



那么为什么我们会将断点中断的指令INT 3设置为单字节长度？这是因为x86指令中最短的指令长度就是一个字节，比如INC自增指令，将断点中断的指令设置为单字节使得在中断时替换原有指令后不会影响到下一条指令的指令。比如图中的INC指令就是单字节指令，如果设置断点中断的指令为2个字节，那么由于当前原有指令的空间有限只有一个字节，使得若要放入两个字节的断点中断指令必然会影响到下一条指令

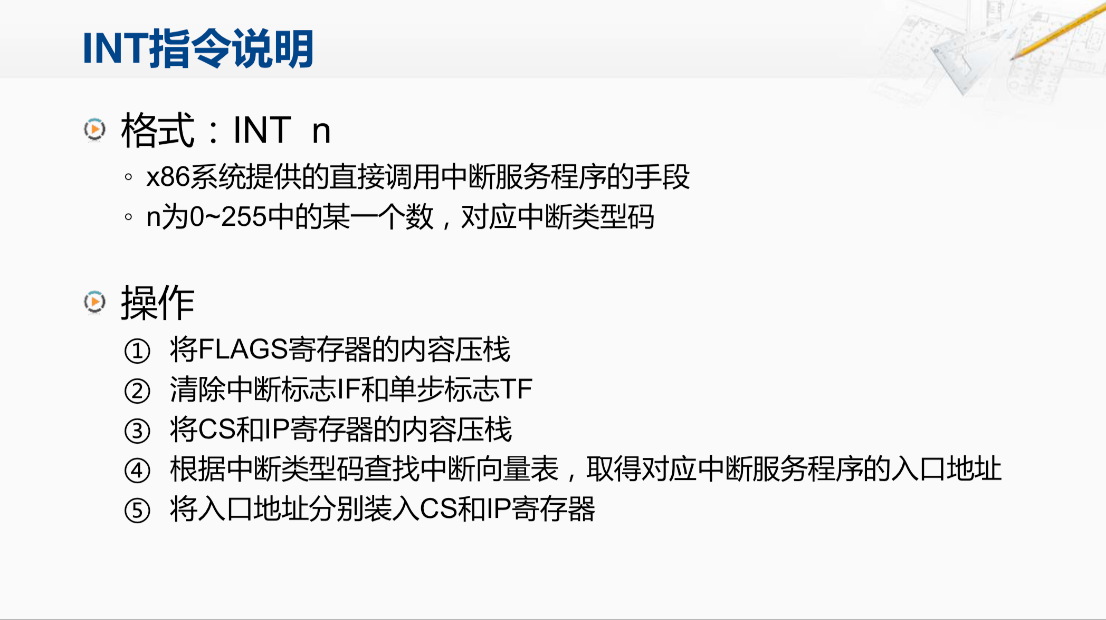
**四、基于中断的功能调用**

没有发生中断却需要调用中断服务程序的两种使用场景：

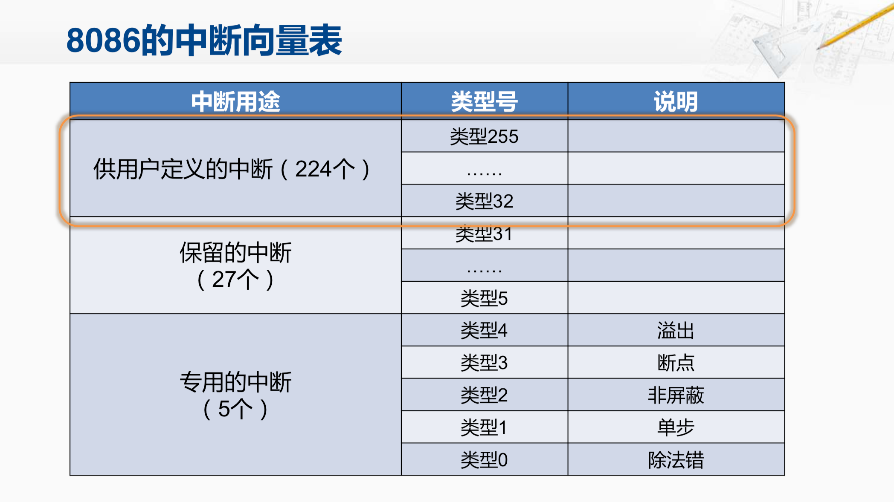
·CPU中的一些专用的中断，比如说前面指出的中断类型1（单步中断）和中断类型3（断点中断），它们就是需要通过调用指令的方式来实现

·还有一种就是供用户定义的中断，这种中断实质上并不是“中断或者是异常”，它们的目的就是通过中断的机制来实现某种特定的功能，例如BIOS中断和DOS中断

1、主动触发中断的指令：INT n

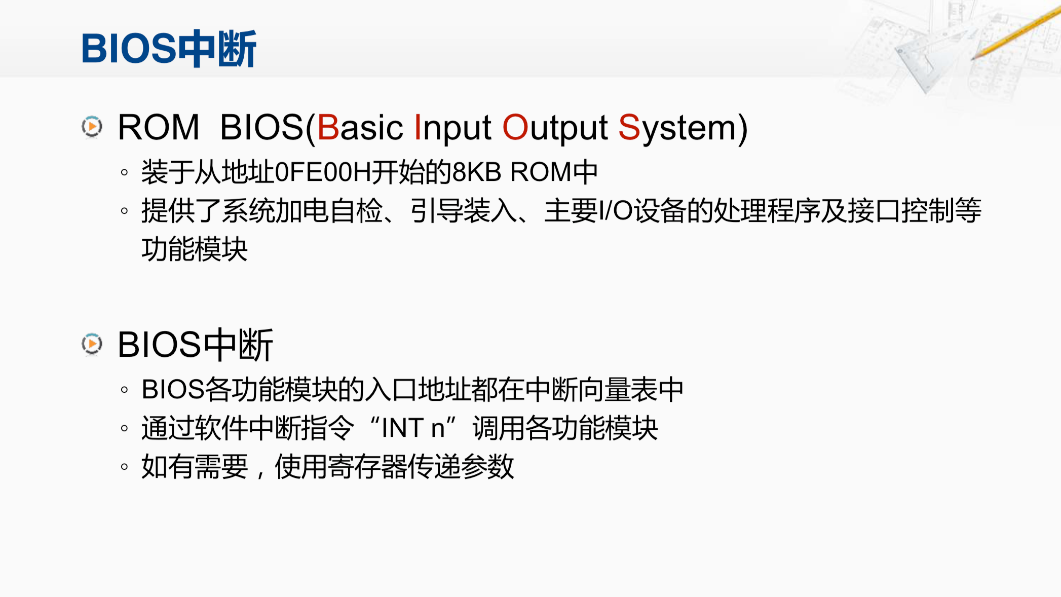


当中断服务程序的入口地址装入到CS：IP寄存器后，下一步CPU就会到中断服务程序的入口取出下一条指令执行中断服务程序，这些操作与被动触发的专用中断指令并没有什么区别。但是主动触发中断指令可以在不满足触发中断的条件下，去主动调用中断服务程序，这一条指令的一般意义在于它为用户定义的中断提供了中断机制，使得它们的功能性调用（某些特定性功能）可以通过中断的形式来完成



2、BIOS中断

BIOS是基本输入输出系统，存放在主板上的ROM当中。在寄存器刚通电或者复位时，CPU就会从BIOS ROM中取出第一条指令开始执行。在BIOS中会提供系统加电自检和主要输入输出设备处理程序等功能模块（模块就是具有独立功能的程序），这些功能不仅在系统初始化时需要使用，同时在后面用户使用计算机的过程中还会使用到。所以BIOS的设计者就将这些功能进行封装，形成多个独立的功能模块，然后将这些独立功能模块的入口地址存放在中断向量表（供用户定义的中断区域）中。这样在我们想使用这些功能的时候可以通过主动调用中断指令“INT n”来调用相应的中断服务程序从而实现BIOS芯片底层编写好的功能。同时对于某一些BIOS中断可以通过将某些数值写入到相关的寄存器中，然后使用BIOS中断，从而达到向功能模块传递参数的目的

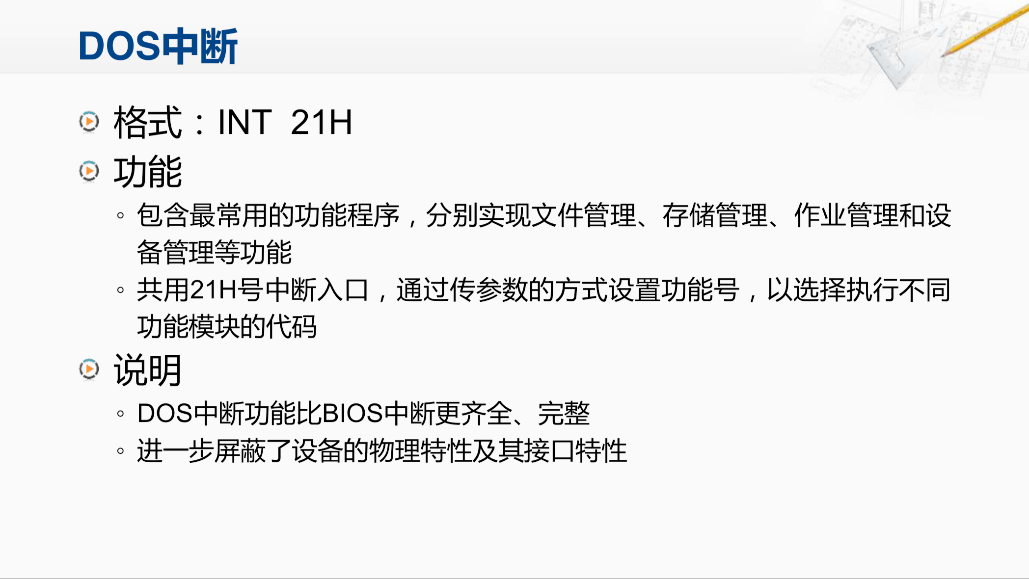


示例：

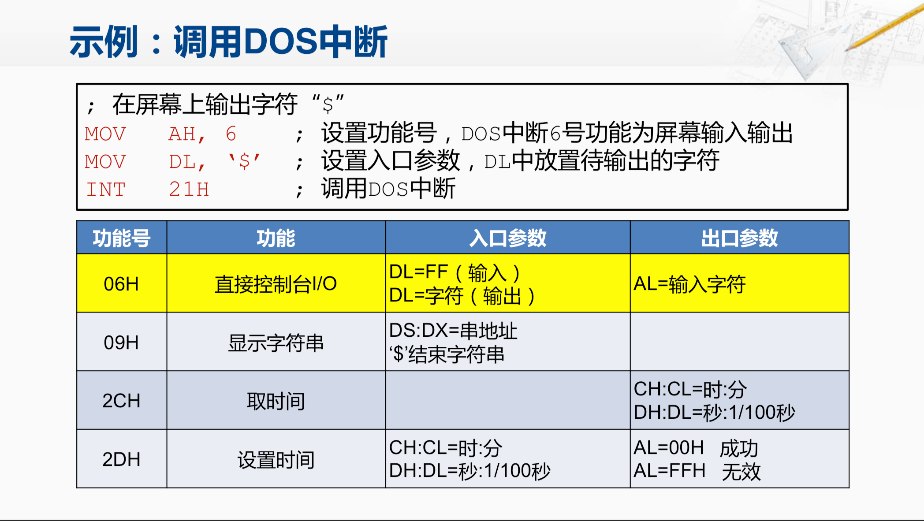


3、DOS中断

DOS中断只有一个中断类型号21H，但是它提供了极其丰富的功能，同时通过向寄存器写入相应值的方式可以达到传参的目的或者完成调用具体什么功能的目的



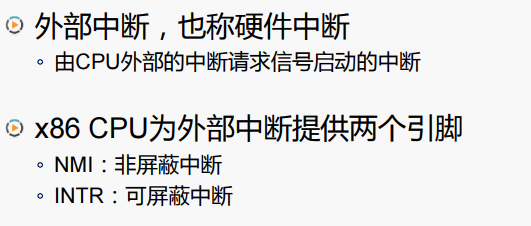
示例：



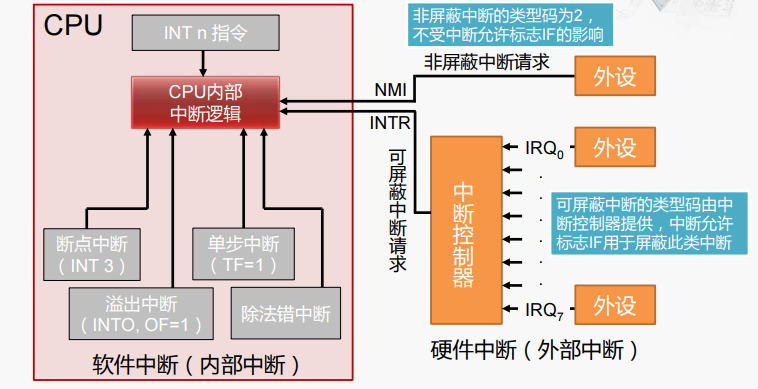
综上可以知道：基于中断实现的功能调用，无论是BIOS中断还是DOS中断其本质并不是计算机运行当中发生了异常的情况，而是利用了现有的中断这种机制来实现一些系统函数，代码的调用，以便向高层的软件屏蔽底层硬件的细节，从而提高编程的便利性、正确性和可移植性

**五、外部中断**

**1、外部中断的概念**：



**2、外部中断的连接示意**：

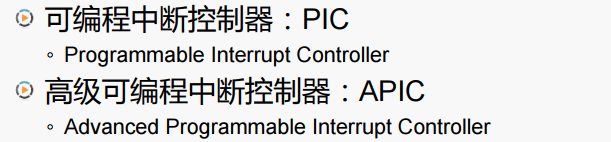


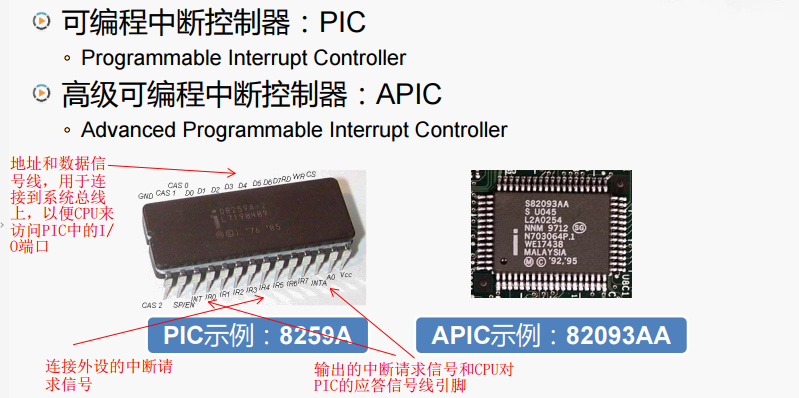
NMI非屏蔽中断一般会连接到系统当中非常重要、不希望被屏蔽的中断请求信号，比如在某些系统当中把电池没电的信号连接到NMI上，非屏蔽中断不受中断标志位影响，即使CPU将IF标志位设置为0关闭对外部中断的响应，CPU仍然会响应NMI的中断请求。这样CPU就可以调用中断服务程序在断电之前将一些重要的信息保存到硬盘当中。当然不同的系统可能会连接不同的中断请求信号到NMI，但仍然是一些非常重要不处理会导致严重错误的事件。

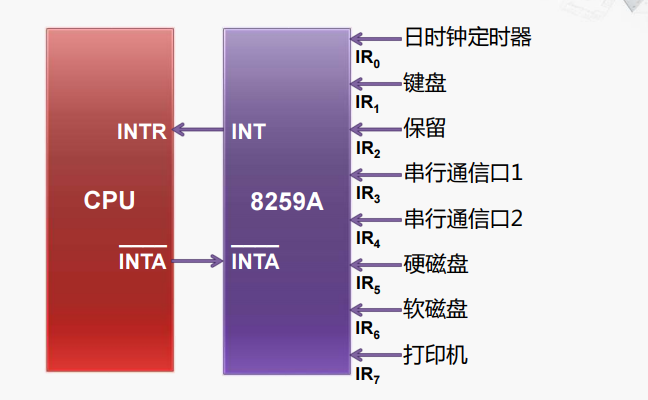
对于一般的外设的中断请求都会连接到可屏蔽中断INTR上，由于INTR信号线只有一根所以需要一个转换电路——中断控制器（它将多个外设的中断请求信号IRQi输入然后输出一根线连接到CPU的INTR引脚）。中断控制器实际上也是一个I/O接口，内含I/O端口，CPU可以访问这些端口（即寄存器），对中断控制器进行配置，如这些外设的中断请求哪一个优先级比较高、哪一些中断请求比较低，或者在这些中断请求中屏蔽一部分。

由于这些中断控制器可以通过程序配置，所以也叫做可编程中断控制器。

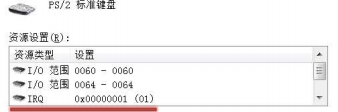
**3、中断控制器**：



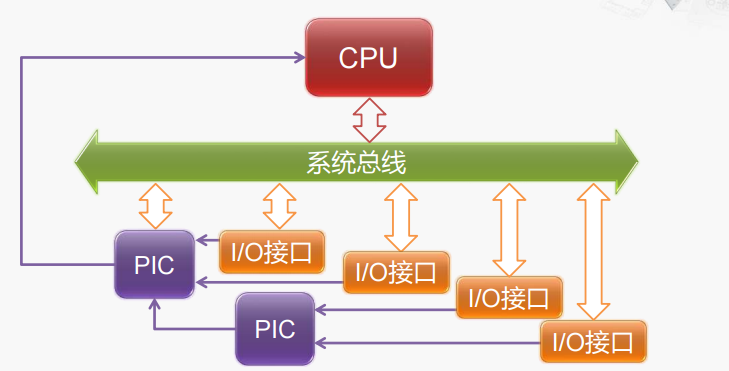




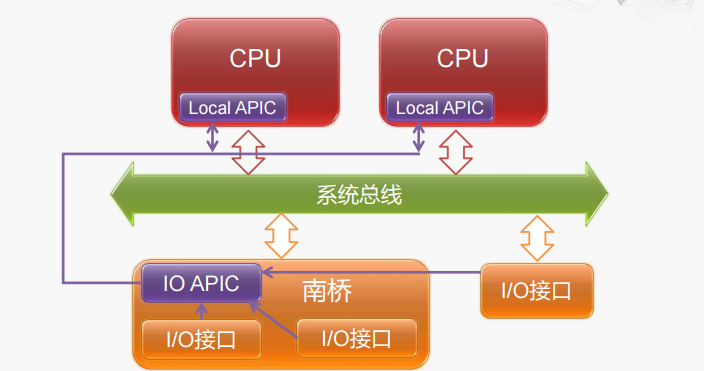
上面的是中断控制器与CPU的相连方式，至今这种连接规范还有一定的影响，比如在自己的笔记本的键盘的中断请求就是通过IRQ1与中断控制器相连



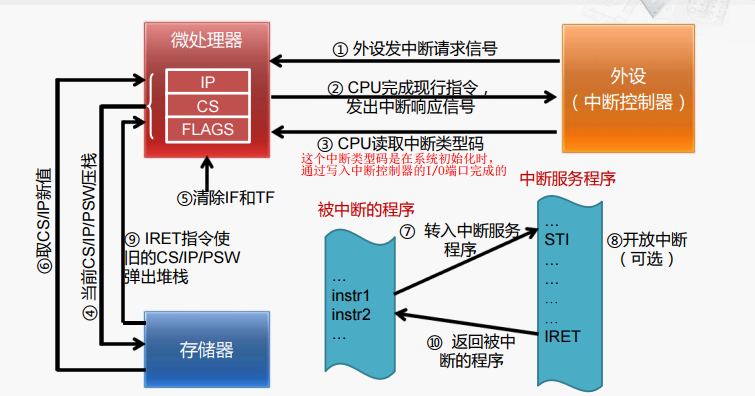
在早期的中断控制实现的时候由于外设的数量可能超过中断控制器PIC的输入引脚数，所以在PIC中会保留一位引脚输出用于串联多个PIC实现超过多于PIC可容纳的输入数量的I/O接口的中断请求信号于PIC相连。



在现代的计算机当中，由于I/O接口很多集成在南桥当中，所以I/O接口的中断请求信号实际在南桥的内部，因此在南桥内也会设置一个中断控制器，如APIC，它的输入不仅包含南桥内部的I/O接口的中断请求信号，而且还会包含南桥外部的I/O接口的中断请求信号，然后输出的中断请求信号连接到CPU中。由于现代计算机中往往会有多个CPU，所以在每一个CPU内部还会有一个中断控制器，CPU不仅需要接收中断请求信号，还要发出中断请求信号给别的CPU用以跟别的CPU进行交互（比如两个CPU进行协同工作，当一个CPU完成了自己的工作后会以中断请求信号的方式通知另一个CPU去完成剩余的工作）。



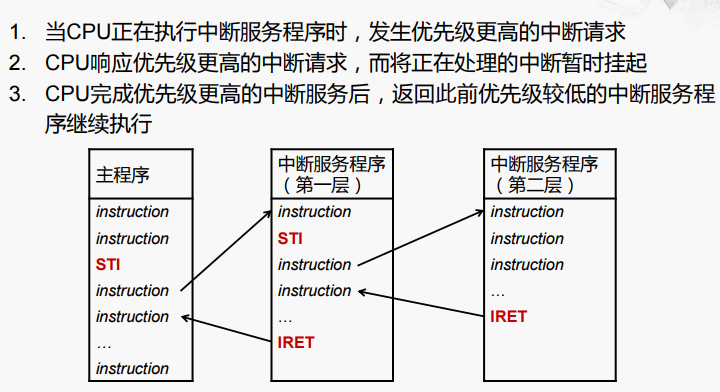
**４、可屏蔽中断的处理过程**：



注意：内部中断以及NMI中断不需要从数据总线上读取中断类型码，而INTR需要，而且该中断类型码由发出中断请求的接口电路提供。

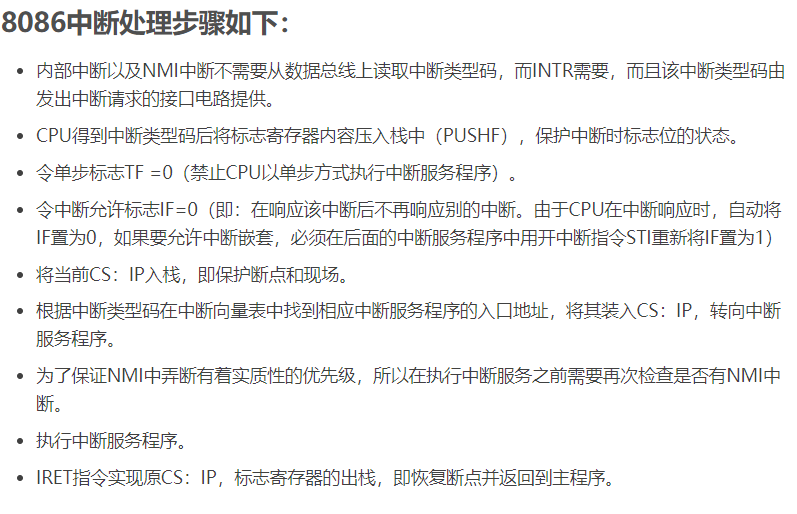
其过程仍然与内部中断的处理过程相同（中断请求——>中断响应——>关中断——＞保护断点——＞识别中断源——>保护现场——>执行中断服务程序——>恢复现场——>中断返回），只不过在处理外部中断的时候有可能会有更高优先级的外部中断进行请求（前提是在执行外部中断服务程序的时候开放中断），进而导致CPU去执行优先级更高的中断服务程序。

中断嵌套：



附上CSDN上网文中的步骤：

１）第一个人的观点：



２）第二个观点：

