版本：A

**DSP F28335中断系统**

|  |  |
| --- | --- |
| 编制： | Lizy |
| 校对： | - |
| 2023.4.15 | |

更改记录

|  |  |
| --- | --- |
| **发布日期** | **更改描述** |
| 20230415 | 第一版发布 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 中断系统说明

CPU在进程正常的程序处理的时候，有时候会被要求处理更高需求级别的任务，因此不得不**中断当前**任务进程，进入中断服务程序。而在处理完这些额外的任务之后，还需要回到之前的任务，因此就需要在进入中断程序之前**必须保存现场**，以确保在主要任务被打断并完成中断程序之后，能够准确地回到之前的任务节点。

## 中断分类

### 中断请求分类

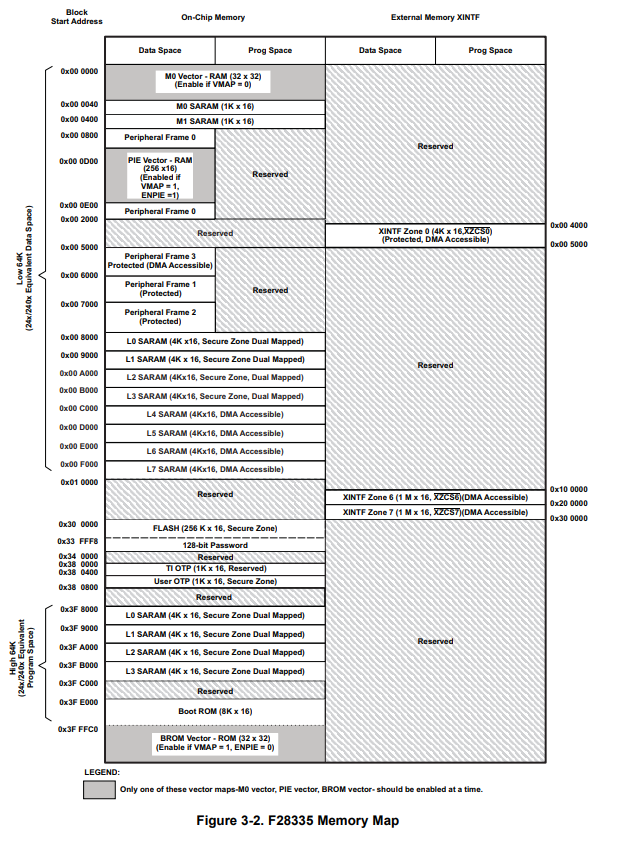
* 可屏蔽中断：可通过软件的方法屏蔽或使能中断；大部分硬件中断是可屏蔽中断；
* 不可屏蔽中断：不可用软件屏蔽的中断。中断一旦发生，就会强制停止CPU进程，进入中断程序，比如复位和NMI；

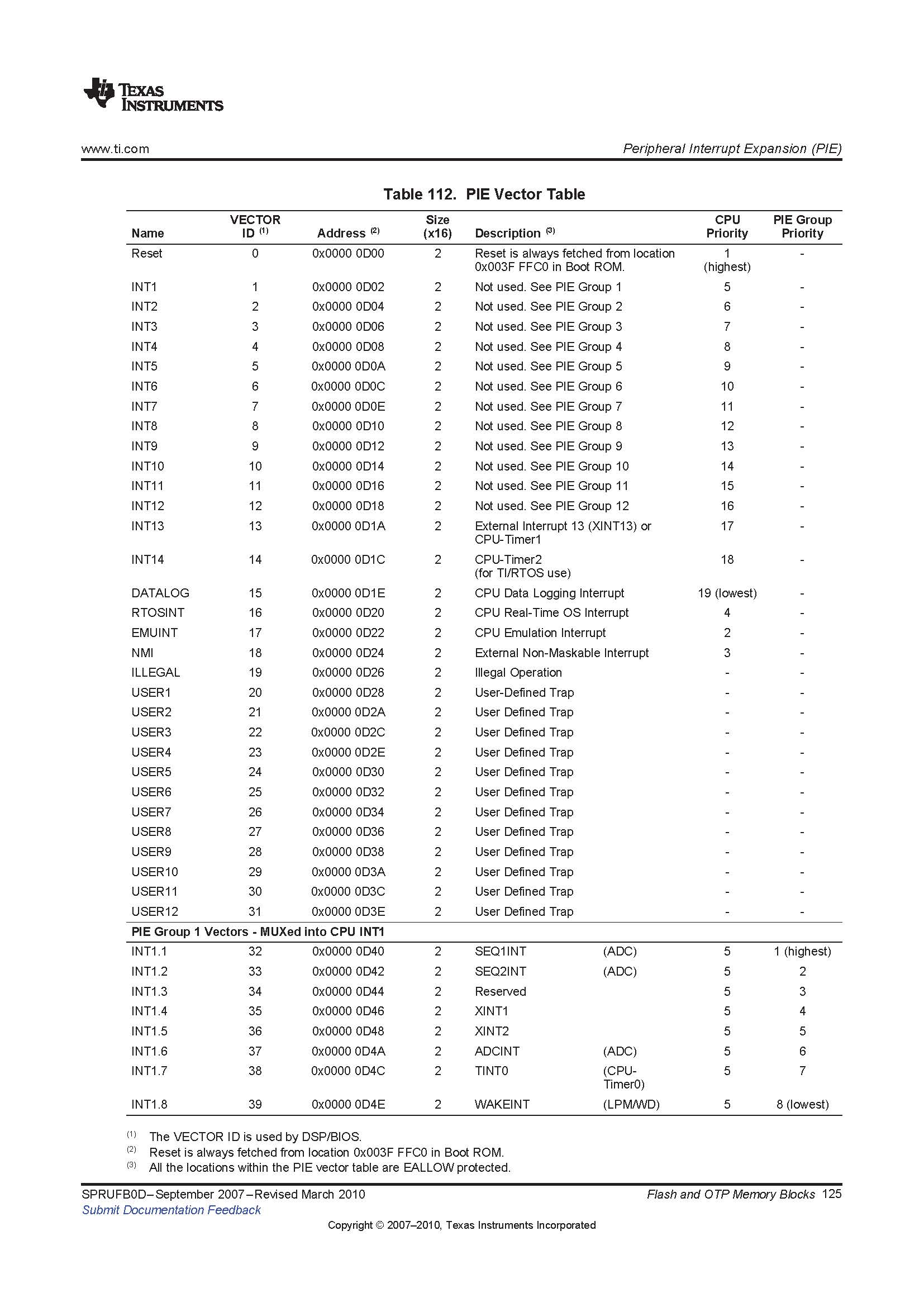
### 中断源分类

* 片内部中断源：PWM、CAP、定时器等；
* 片外部中断源：外部中断输入引脚XINT1、XINT2等引入的信号；

### 中断向量表和优先级

对于F28335,其中断向量表根据配置可以定位到3个不同的地方。当复位是VMAP =1 ,ENPIE=0。中断向量表定位在Boot ROM中的最后32\*32个存储单元中。当设置VMAP=0时,向量表定位在0x000000开始的32\*32的存储单元中。当ENPIE =1,VMAP=1时，中断向量表定位到PF0区域中。





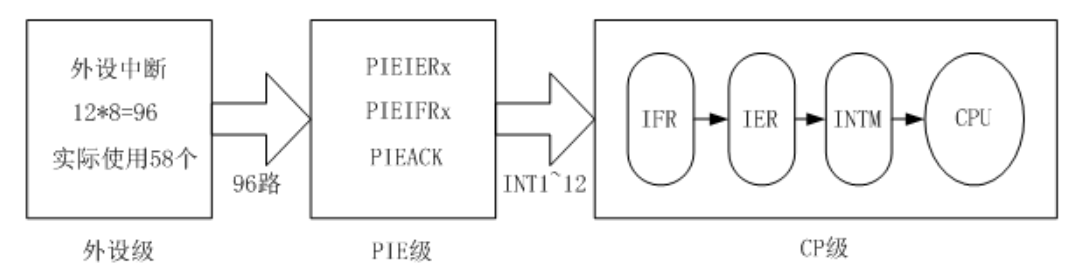
从图中可得：

* CPU级中断有19个；
* CPU级中断的优先级Reset优先级最高，但是系统复位时复位向量总是从0x3fffc0取。
* CPU可屏蔽中断优先级顺序（从高到低）：RTOSINT,INT1-INT14,DLOGINT。INT1-INT14是通用中断，RTOSINT和DLOGINT是用于仿真的。这些中断的控制是由寄存器IFR，IER和DBGIER控制的。IFR是中断申请标志寄存器，IER是中断使能寄存器，DBGIER是中断允许寄存器（在实时仿真模式下才使用）；ST1中的第0位，INTM控制整个中断的屏蔽，INTM=1是所有中断都被屏蔽（关中断DINT）；
* INT1-INT12接PIE模块拓展了中断系统，INTx.1-INTx.8中断优先级依次降低。

## 中断结构

F28335有很多的外设资源，这些外设资源有可能会同时发布额外任务给CPU，换句话说F28335的中断源有很多，这些中断源想要得到CPU的响应就必须将中断信号传递给CPU。可是F28335的中断线数是有限的，这个时候PIE模块来分配中断资源了。

F28335的中断采用三级中断机制，分为外设级、PIE级和CPU级。对于某个集体的外设中断请求，任意一级的不许可，CPU最终都不会响应外设中断。以定时器0的周期中断为例介绍F28335的三级中断。





常见的中断源

从图中可以看出：

* RST是硬件中断，没有应用软件的中断服务程序，而是DSP启动过程；
* GPIO口可以配置为XINT外部中断或NMI不可屏蔽中断，通过中断选择寄存器（GPIOXINTnSEL或GPIOXNMISEL）和配置寄存器（XINTnCR或XNMICR）来设置；

# 三级式中断系统

## 外设级

假设在程序执行的过程中，某个外设产生了一个中断事件，那么在这个外设的**某个寄存器**中与该中断事件相关的中断标志位（IF = Interrupt Flag）被置为1。此时，如果该中断相应的中断使能位（IE = Interrupt Enable）已经被置位，也就是值为1，该外设就会像PIE控制器发出一个中断请求；相反，如果中断事件产生了，相应的中断标志位IF也已经被置位了，但是没有使能相应中断，那么外设就不会向PIE控制器提出中断请求。值得一提的是，这时候虽然外设不会向PIE控制器提出中断请求，但是相应的中断标志位会一直保持状态，直到用程序清除为止。当然，在中断标志位保持置位状态的时候，一旦该中断被使能，那么外设会立即向PIE发出中断请求。

注意，不管在什么情况下，外设寄存器的中断标志位**IF都必须手动清除**（SCI、SPI除外）。

如：CpuTimer0Regs.TCR.bit.TIF = 1; //清除定时器0的中断标志位

外设级在编程时需要手动的有：

1. **外设中断的使能**：需要将与该中断相关的外设寄存器中的中断使能位IE置1；
2. 外设中断的屏蔽：需要将与该中断相关的外设寄存器中的中断屏蔽位置0；
3. **外设中断标志位的清除**：需要将与该中断相关的外设寄存器中的中断标志IF置1清除；

## PIE级

当外设产生中断事件，相关中断标志位IF置位，中断使能位IE使能后，外设就会给PIE提出中断请求。

PIE控制器将96个外设和外部引脚的中断进行了分组，每8个中断为1组，一共12组，分别是PIE1-PIE12。每组中断被多路汇集进入了1个CPU中断，如WAKEINT、TINT0、ADCINT、XINT2、XINT1、SEQ2INT、SEQ1INT这7中断都是PIE1组内，这些中断也都汇集到了CPU中断的INT1，同样的，PIE2组的中断都汇集到了CPU中断的INT2、、、、PIE12组的中断都汇集到了CPU中断的INT12。

和外设级类似，PIE控制器中的每一组都会有一个中断标志寄存器PIEIFRx和一个中断使能寄存器PIEIERx，x = 1，2……12。每个寄存器的低8位对应8个外设中断，高8位保留。

如：PieCtrlRegs.PIEIER1.bit.INTx7 = 1; //使能PIE级第1组第7个（定时器0）中断使能位

由于PIE控制器是多路复用，每一组内有许多不同的外设中断共用一个CPU中断，但是**每一组在同一个时间内只能有一个中断被响应**。首先，PIE**组内**的各个中断也是有优先级的，位置在前面的中断优先级高于位置在后面的中断，这样，如果同时有多个中断提出请求的话，PIE先处理优先级高的，后处理优先级低的。同时，PIE控制器处理每组有PIEIFR和PIEIER寄存器外，还有一个PIE中断应答寄存器PIEACK，它的低12位对应着12个组，即PIE1-PIE12，也就是INT1-INT12，高位保留。这些位的状态表示了PIE能否响应组内的中断。如CPU定时器0的周期中断被响应了，则PIEACK的第0位（对应PIE1,即INT1）就会被置位，并且一直保持直到手动清除这个标志。当CPU在响应T0INT的时候，PIEACK的第0位一直是1，这样如果**PIE1组内**发生了其他的外设中断，则暂时不会被PIE控制器响应并发送给CPU，必须等到PIEACK的第0位被复位之后，如果该中断还存在，那么PIE控制器会立即把中断请求送给CPU。所以，每个外设中断被响应后，一定要对PIEACK的相关位进行手动复位，以使得PIE控制器能够响应同组内的其他中断。

如：PieCtrlRegs.PIEACK.bit.ACK1= 1; //清除PIE第1组INT1的应答位

因此，当外设中断向PIE提出中断请求后，PIE中断标志寄存器PIEIFRx的相关标志位被置位，这时候如果相应的PIEIERx相应的中断使能位被置位，PIEACK相应位的值为0时，PIE控制器便会将该外设中断请求交给CPU；如果相应的PIEIERx相关中断使能位未被置位，就是没有使能，或者PIEACK响应位为1，就是PIE控制器正在处理同组的其他中断，PIE控制器暂时都不会响应外设的中断请求。

PIE级在编程时需要手动的有：

PIE中断使能：需要使能某个外设中断，就得将其相应组的使能寄存器PIEIERx的响应位进行置位；

PIE的中断屏蔽：这是和使能相反的操作；

PIE应答寄存器PIEACK相关位的清除，以使得CPU能够响应同组内的其他中断；

将PIE级的中断和外设级的中断相比发现，外设中断的中断标志位xxxIF是需要手动清除的，而PIE级的中断标志位PIEIFRx都是自动清除的。但是PIE级多了一个PIAEACK寄存器，它相当于一个关卡，同一时间只能放一个中断过去，只有等这个中断被响应完成后，再给关卡一个放行命令后，才能让同组的下一个中断过去，被CPU响应。

## CPU级

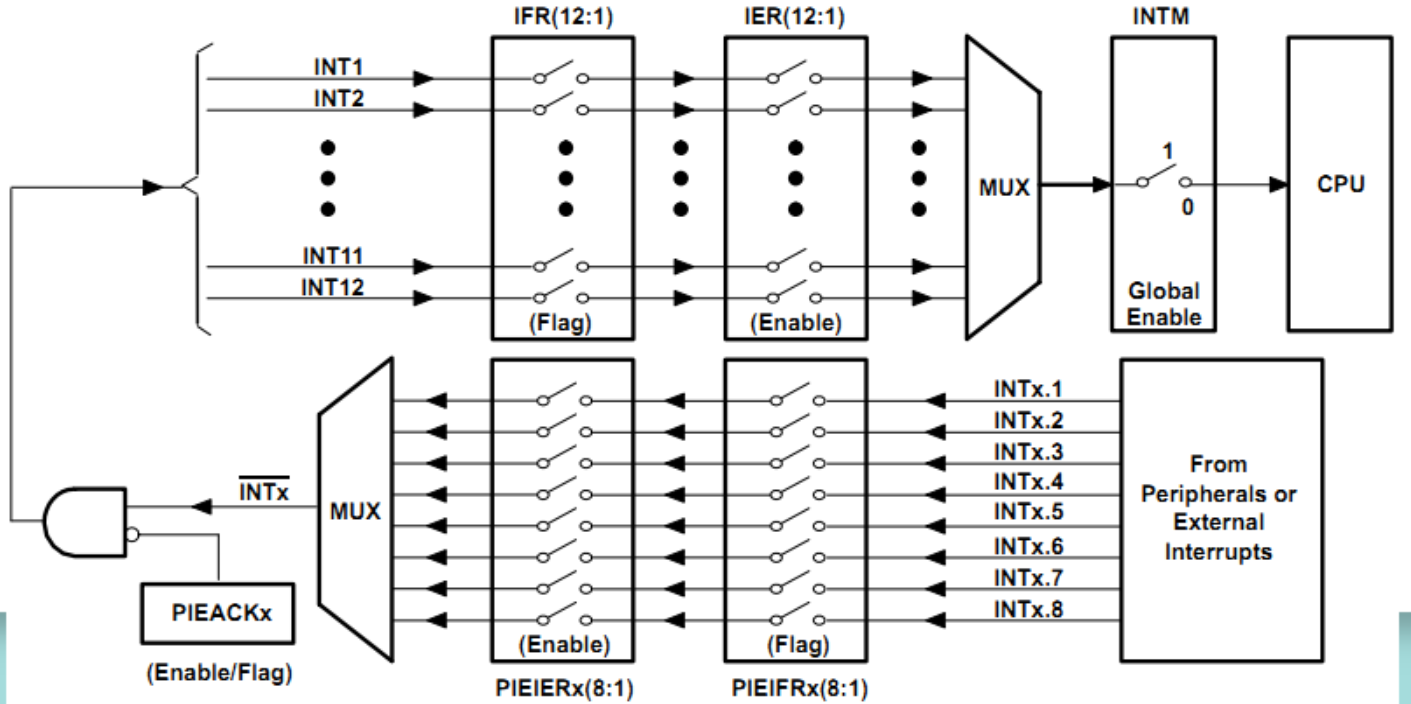
和前两级类似，CPU级也有中断标志寄存器IFR和中断使能寄存器IER。当某个外设中断请求通过PIE发送到CPU时，CPU中断标志寄存器IFR中相应的中断标志位INTx就会被置位。例如，当CPU定时器0的周期中断T0INT发送到CPU时，IFR的第0位INT1就会被置位，然后该状态就会被锁存在寄存器IFR中。这时候，CPU不会马上去执行相应的中断，而是检查IER寄存器中的相关位的使能情况和CPU寄存器**ST1中全局中断屏蔽寄存器INTM**的使能情况。如果IER中的相关位被置位，并且INTM的值为0，则中断就会被CPU相应。在CPU定时器0的周期中断的例子中，当IER的第0为INT1被置位，INTM的值为0（EINT），则CPU就会响应定时器0的周期中断T0INT。

CPU接收到了中断请求，并发现可以去响应的时候，就得暂停正在执行的程序，转而去响应中断服务程序，但此时，它必须得做些准备工作，以便执行完中断服务程序后回来还能找到原来的执行的地方和原来的状态。CPU会先将相应的IFR位进行清除，EALLOW也被清除，INTM被置位（使能全局中断屏蔽，禁止中断），就是不能响应其他中断了（**进入中断服务程序之前，硬件做的**），等于CPU向其他中断发出通知，现在正忙，没有时间处理别的请求了，得等到处理完手上的中断后才能再来处理。然后，CPU会存储返回地址并自动保存相关信息，如将正在处理的数据放入堆栈等等，做好这些准备工作后，CPU会从PIE向量表中取出对应的中断向量ISR，从而转去执行中断服务程序。

所以**F28335默认禁止中断嵌套**。

可以看出，CPU级中断标志的置位和清除与PIE级相同，都是自动完成（硬件）的。

下图是F28335的三级中断图：



# 实现步骤

1. 在外设初始化函数中使能外设中断；

void InitCpuTimer( )

{

……

CpuTimer0Regs.TCR.bit.TIE = 1; //使能CPU定时器0的周期中断

……

}

1. 在主函数中主要是初始化外设，使能PIE和CPU中断等。

void main()

{

......

......

//初始化CPU定时器0

InitCpuTimer();

//禁止和清除所有CPU中断标志及使能

DINT;

IFR = 0X0000;

IER = 0X0000;

//初始化PIE中断，清除PIE中断标志及使能

InitPieCtrl();

//初始化中断向量表

InitPieVectTable();

EALLOW;

PieVectTable.TINT0 = &timer0\_isr; //更新CPU定时器0的中断向量表(中断服务程序入口地址)

EDIS;

//使能PIE中断

PieCtrlRegs.PIEIER1.bit.INTx7 = 1; //使能PIE级的CPU定时器0的中断

//使能CPU中断

IER |= M\_INT1; //开中断INT1

EINT; //使能全局中断

ERTM; //使能实时中断

}

1. 在**中断服务程序**中必须手动清除**外设中断标志位IF**和复位PIE应答寄存器的**PIEACK**相关位，使得CPU能够响应PIE控制器同组内的其他中断。

interrupt void timer0\_isr()

{

......

......

CpuTimer0Regs.TCR.bit.TIF = 1; //清除定时器中断标志位

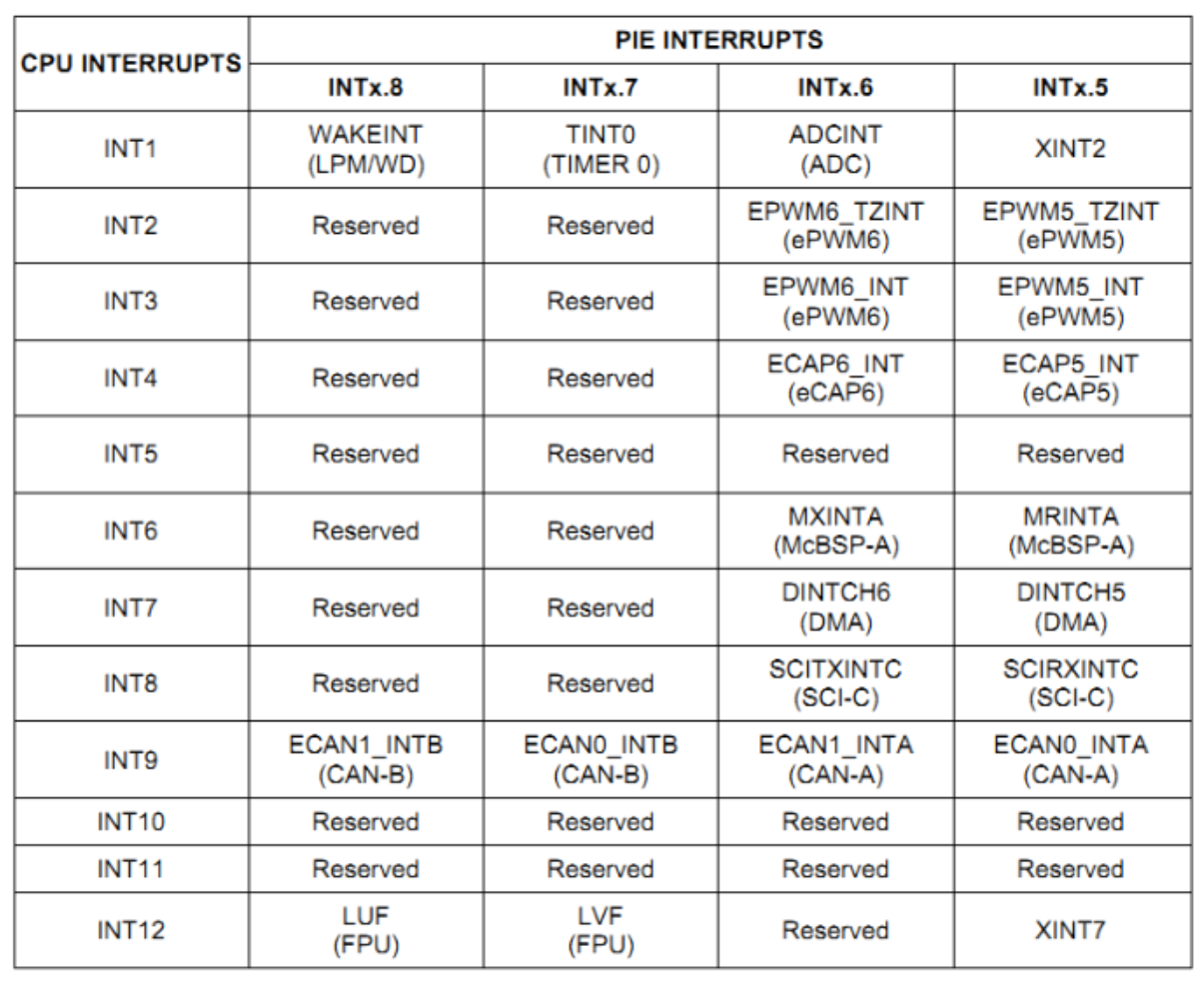
CpuTimer0Regs.TCR.bit.TRB = 1; //重装定时器

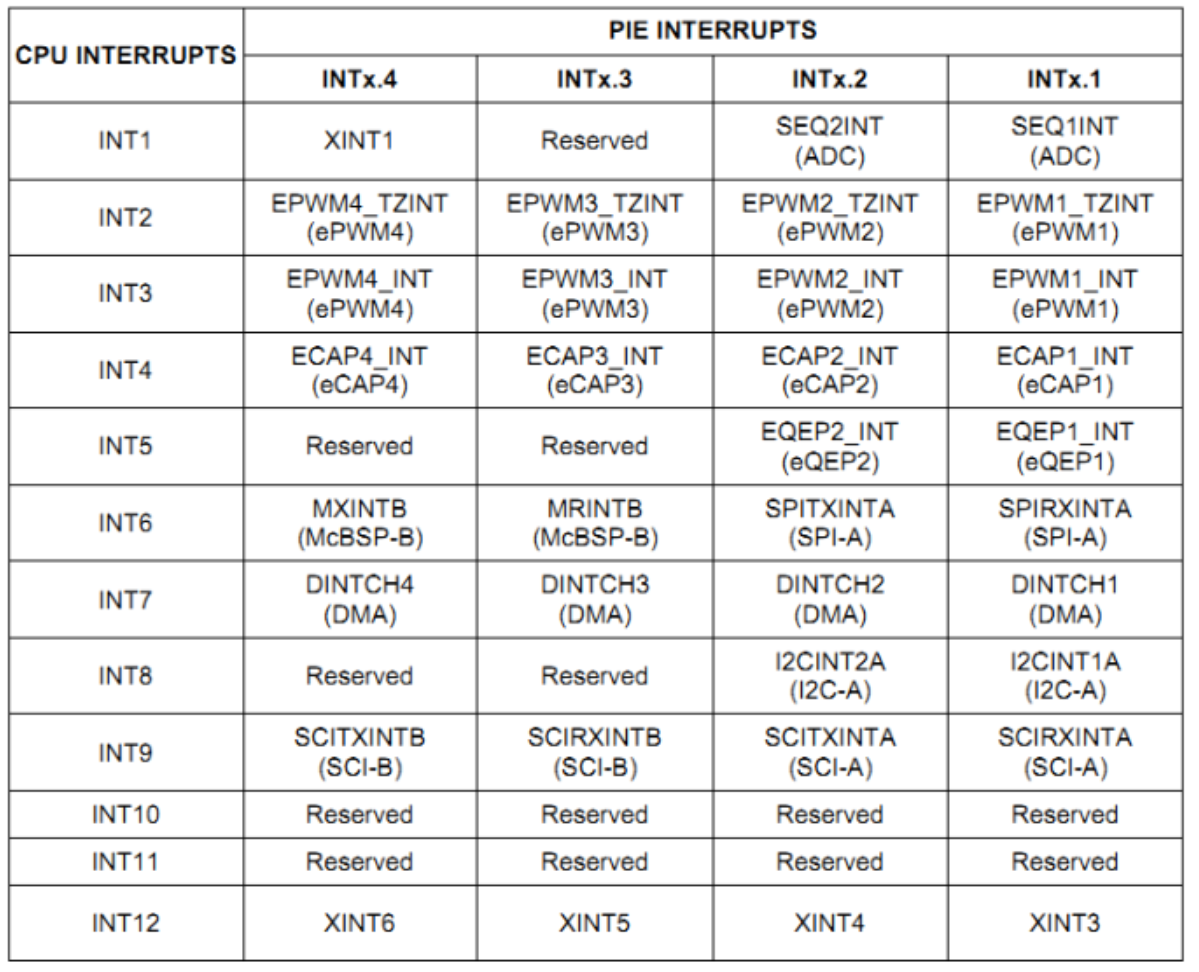
PieCtrlRegs.PIEACK.bit.ACK1 = 1; //清除PIEACK以响应同组其他中断

}

注：中断服务程序必须用关键字interrupt标识；

# 中断向量表





图中如TINT0是外设定时器0的中断，XINTx是通过GPIO口的外部中断，可以通过GPIO XONTn中断选择寄存器（GPIOXINTnSEL）来配置，通过配置寄存器（XINTnCR）来**使能**外设GPIO中断；

# 中断嵌套

TI默认情况下，**中断不嵌套**。可以通过IER和PIIERx寄存器的软件控制来嵌套中断并对其进行优先级排序。

关于中断嵌套总结：

* **TI默认情况**下，**中断不嵌套**，即**单重中断**：在进入中断服务程序之前通过硬件的隐指令关闭了中断（相当于DINT），中断服务程序**结束前再打开中断**（相当于EINT）。这样一个中断要是没执行完，尽管又来一个优先级更高的中断，CPU也不会被中断去执行更高优先级的中断；
* 如果在**中断服务程序内**使能全局中断（EINT），那么如果有新的中断来，CPU都会去执行新的中断，无论其优先级的高低（只要有就去响应），这就是**多重中断**；实现了中断嵌套，但是这种嵌套是不分优先级的嵌套；
* 因为F28335中**全局中断屏蔽INTM**是寄存器**ST1中的一个位**，所以不能单独使能高优先级的中断，只能全开或全关，所以中断嵌套过程中**全局中断屏蔽寄存器**；

# 软、硬中断

## 硬件中断

硬件中断是由**与系统相连的外设**(比如网卡 硬盘 键盘等)自动产生的。每个设备或设备集都有他自己的IRQ(中断请求), 基于IRQ, CPU可以将相应的请求分发到相应的硬件驱动上(注: 硬件驱动通常是内核中的一个**子程序**, 而不是一个独立的进程)。比如当网卡受到一个数据包的时候, 就会发出一个中断。

对于**时钟中断**, 内核调度代码会将当前正在运行的代码挂起, 从而让其他代码来运行。它的存在时为了让**调度代码**(或称为调度器)可以调度多任务。

## 软中断

软中断是一种需要内核为正在运行的进程去做一些事情(通常为I/O)的请求，如系统调用。通常软中断是对一些I/O的请求。

软中断仅与内核相联系, 而内核主要负责对需要运行的任何其他进程进行调度。软中断不会直接中断CPU, 也只有当前正在运行的代码(或进程)才会产生软中断。

有一个特殊的软中断是Yield调用, 它的作用是请求内核调度器去查看是否有一些其他的进程可以运行。

## 硬件中断和软中断的区别

1. 硬件中断是由**外设**引发的, 软中断是**执行中断指令**产生的；
2. 硬件中断的中断号是由中断控制器提供的, 软中断的中断号由指令直接指出, 无需使用中断控制器；
3. 硬件中断是可屏蔽的, 软中断不可屏蔽；
4. 硬件中断处理程序要确保它能**快速地完成任务,** 这样程序执行时才不会等待较长时间, 称为**上半部**；
5. 软中断处理硬中断未完成的工作, 是一种推后执行的机制, 属于下半部；

