第6章　中断系统的 C 语言程序设计

6.1 中断系统

SPCE061A 单片机中断系统，可以提供 14 个中断源，具有两个中断优先级，可实现两级中断嵌套功能。用户可以用关中断指令（或复位）屏蔽所有的中断请求，也可以用开中断指令使 CPU 接受中断申请。每一个中断源可以用软件独立控制为开或关中断状态，但中断级别不可用软件设置。

6.1.1 中断源

SPCE061A 单片机的中断系统有 14 个中断源分为两个定时器溢出中断、两个外部中断、一个串行口中断、一个触键唤醒中断、7 个时基信号中断、PWM 音频输出中断。如下表 6.1。

表6.1 中断源列表

中断源	中断优先级	中断向量	保留字
Fosc/1024 溢出信号 PWM INT	FIQ/IRQ0	FFF8H/FFF6H	_FIQ/_IRQ0
TimerA 溢出信号	FIQ /IRQ1	FFF9H/FFF6H	_FIQ/_IRQ1
TimerB 溢出信号	FIQ /IRQ2	FFFAH/FFF6H	_FIQ/_IRQ2
外部时钟源输入 信号 EXT2	IRQ3	FFFBH	_IRQ3
外部时钟源输入 信号 EXT1			
触键唤醒信号			
4096Hz 时基信号	IRQ4	FFFCH	_IRQ4
2048Hz 时基信号			
1024Hz 时基信号			
4Hz 时基信号	IRQ5	FFFDH	_IRQ5
2Hz 时基信号			
频选信号 TMB1	IRQ6	FFFEH	_IRQ6
频选信号 TMB2			
UART 传输中断	IRQ7	FFFFH	_IRQ7
BREAK	软中断		

从表中可以看到每个中断入口地址对应多个中断源，因此在中断服务程序中需通过查询中断请求位来判断是那个中断源请求的中断。

定时器溢出中断源

定时器溢出中断由 SPCE061A 内部定时器中断源产生，故它们属于内部中断；在 SPCE061A 内部有两个 16 位定时器/计数器，定时器 TimerA/TimerB 在定时脉冲作用下从预置数单元开始加 1 计数，当计数达“0xFFFF”时可以自动向 CPU 提出溢出中断请求，以表明定时器 TimerA 或 TimerB 的定时时间已到。定时器 TimerA/TimerB 的定时时间可由用户通过程序设定，以便 CPU 在定时器溢出中断服务程序内进行计时。另外，SPCE061A 单片机的定时器时钟源很丰富，从高频到低频都有，因此，根据定时时间长短可以选择不同的时钟源，定时器 A 的时钟源比定时器 B 多，定时器 B 无低频时钟源。在中断一览表中也可以看出，定时器 A 的中断，IRQ 和 FIQ 中都有，方便开发人员的使用。详见第 5 章**定时器/计数器**内容。定时器溢出中断通常用于需要进行定时控制的场合。

外部中断源

SPCE061A 单片机有两个外部中断，分别为 EXT1 和 EXT2，两个外部输入脚分别为 B 口的 IOB2 和 IOB3 的复用脚。EXT1（IOB2）和 EXT2（IOB3）两条外部中断请求输入线，用于输入两个外部中断源的中断请求信号，并允许外部中断以负跳沿触发方式来输入中断请求信号。如图 6.1。

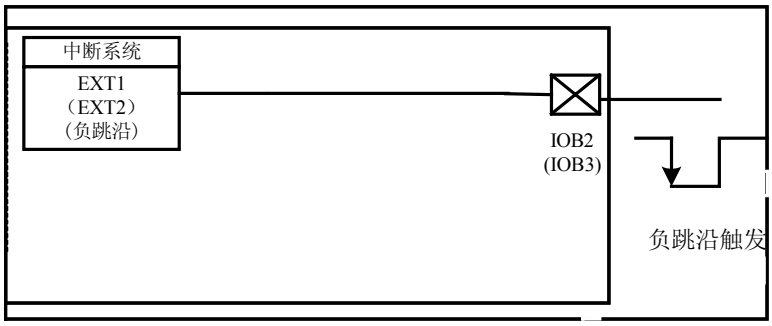

图6.1 外部中断结构

另外，SPCE061A 单片机在 IOB2 和 IOB4 之间以及 IOB3 和 IOB5 之间分别加入两个反馈电路，可以外接 RC 振荡器，做外部定时中断使用。这部分内容请参照第四章系统设置部分。

串行口中断源

串行口中断由 SPCE061A 内部串行口中断源产生，故也是一种内部中断。串行口中断分为串行口发送中断和串行口接收中断两种，但其中断向量是一个，因此，进入串行中断服务程序时，也需要判断是接收中断还是发送中断。在串行口进行发送/接收完一组串行数据时，串行口电路自动使串行口控制寄存器 P_UART_Command2 中的 TXReady 和 RXReady 中断标志位置位。并自动向 CPU 发出串行口中断请求，CPU 响应串行口中断后便立即转入串行口中断服务程序执行。因此，只要在串行中断服务程序中安排一段对 P_UART_Command2 对 TXReady 和 RXReady 中断标志位状态的判断程序，便可区分串行口发生了接收中断请求还是发送中断。当然，串行传输中，既可以使用中断方式收发数据也可使用查询方式来收发数据。在 SPCE061A 中串行口为 B 口的 IOB7（RXD）和 IOB10（TXD）两个复用脚。

触键唤醒中断源

当系统给出睡眠命令时，CPU 便关闭 PLL 倍频电路，停止 CPU 时钟工作而使系统进入睡眠状态，在睡眠过程中，通过 IOA 口低 8 位接的键盘就可以给出唤醒信号，将系统从睡眠状态转到工作状态。与此同时，产生一个 IRQ3 中断请求。进入键唤醒中断，CPU 继续执行下一个程序指令。一般来讲，中断系统提供的中断源 IRQ1～IRQ6 均可作为系统的唤醒源来用，做定时唤醒系统。

若以触键作为唤醒源，其功能通过并行 A 口的 IOA0～IOA7 及中断源 IRQ3_KEY 的设置来实现。

时基信号中断源

时基信号发生器的输入信号来自实时时钟 32768Hz；输出有通过选频逻辑的 TMB1、TMB2 信号和直接从时基计数器溢出而来的各种实时时基信号。当开启时基信号中断后，有时基信号到来，发出时基信号中断申请，CPU 查询到有中断请求后，允许中断并置位 P_INT_Ctrl 中相应的中断请求位，在中断服务程序中通过测试 P_INT_Ctrl 来确定是那个频率时基信号产生的中断，可以通过时基信号来定时控制。

6.1.2 中断优先级和中断入口地址

SPCE061A单片机中，FIQ的优先级高于IRQ的优先级，在IRQ中断中IRQ1的中断优先级高于IRQ2，IRQ2的中断优先级高于IRQ3，按照IRQ的序号，序号越高则中断优先级越低，UART的中断优先级最低。在IRQ中断中，只是中断查询有先后，不能进行中断嵌套。同中断向量内的中断源中断优先级相同，看软件想先处理哪个就处理哪个。因此，也可以说，同中断向量内的中断源中断优先级由软件决定。

表 6.2 按中断优先级关系从上到下列出了 SPCE061A 的中断优先级和入口地址。

表6.2　　中断优先级和中断向量

中断向量[注]	中断优先级别
FFF7H（复位向量）	RESET
FFF6H	FIQ
FFF8H	IRQ0
FFF9H	IRQ1
FFFAH	IRQ2
FFFBH	IRQ3
FFFCH	IRQ4
FFFDH	IRQ5
FFFEH	IRQ6
FFFFH	UART IRQ

6.2 中断控制

6.2.1 中断控制的寄存器

SPCE061A 单片机有多个中断源，为了使每个中断源都能独立地被开放和屏蔽，

以便用户能灵活使用，它在每个中断信号的通道中设置了一个中断屏蔽触发器，只有该触发器无效，它所对应的中断请求信号才能进入 CPU，即此类型中断开放。否则即使其对应的中断请求标志位置“1”，CPU 也不会响应中断，即此类型的中断被屏蔽。同时 CPU 内还设置了一个中断允许触发器，它控制 CPU 能否响应中断。

SPCE061A 对中断源的开放和屏蔽，以及每个中断源是否被允许中断，都受中断允许寄存器 P_INT_Ctrl 和 P_INT_Clear 及 P_INT_Ctrl_New 控制和一些中断控制指令。

中断控制单元 P_INT_Ctrl（读/写）（7010H）

P_INT_Ctrl 控制单元具有可读和可写的属性，其读写时的意义是不同的。

表6.3 中断控制单元 P_INT_Ctrl

b7	b6	b5	b4	b3	b2	b1	b0
IRQ3_KEY	IRQ4_4KHz	IRQ4_2KHz	IRQ4_1KHz	IRQ5_4Hz	IRQ5_2Hz	IRQ6_TMB1	IRQ6_TMB2

b15	b14	b13	b12	b11	b10	b9	b8
FIQ_Fosc/1024	IRQ0_Fosc/1024	FIQ_TMA	IRQ1_TMA	FIQ_TMB	IRQ2_TMB	IRQ3_EXT2	IRQ3_EXT1

当写中断控制单元中的某位为“1”时，即允许该位所代表的中断被开放，并关闭屏蔽中断触发器，此时当有该中断申请时，CPU 会响应。如果该位被置 0 则禁止该位所代表的中断。即使有中断申请，CPU 也不会响应。

当读取中断控制单元时，其主要作为中断标志。因为 P_INT_Ctrl 每一位均代表一个中断，当 CPU 响应某中断时，便将该中断标志置“1”，可以通过读取该寄存器来确定 CPU 响应的中断。

清除中断标志控制单元 P_INT_Clear（写）（7011H）

清除中断标志控制单元主要用于清除中断控制标志位，当 CPU 响应中断后，会将中断标志置位为“1”，当进入中断服务程序后，要将其控制标志清零，否则 CPU 总是执行该中断。

表6.4 清除中断标志控制单元 P_INT_Clear

b7	B6	b5	b4	b3	b2	b1	b0
IRQ3_KEY	IRQ4_4KHz	IRQ4_2KHz	IRQ4_1KHz	IRQ5_4Hz	IRQ5_2Hz	IRQ6_TMB1	IRQ6_TMB2

b15	b14	b13	b12	b11	b10	b9	b8
FIQ_Fosc/1024	IRQ0_Fosc/1024	FIQ_TMA	IRQ1_TMA	FIQ_TMB	IRQ2_TMB	IRQ3_EXT2	IRQ3_EXT1

因为 P_INT_Clear 寄存器的每一位均对应一个中断，所以如果想清除某个中断状态标志，只要将该寄存器中对应的中断位置 1 即可清除该中断状态标志位。该寄存器只有写的属性，读该寄存器是无任何意义的。

激活和屏蔽中断控制单元 P_INT_Ctrl_New(读/写)($702DH)

该单元用于激活和屏蔽中断。

表6.5 激活和屏蔽中断控制单元 P_INT_Ctrl_New

b0	b1	b2	b3	b4	b5	b6	b7
IRQ6	IRQ6	IRQ5	IRQ5	IRQ4	IRQ4	IRQ4	IRQ3

b8	b9	b10	b11	b12	b13	b14	b15
IRQ3	IRQ3	IRQ2	FIQ	IRQ1	FIQ	IRQ0	FIQ

当写该控制单元时，与写 P_INT_Ctrl 功能相似。

读该控制单元时，只作为了解激活那一中断的功能使用。与其写入值是一致的。

中断控制配置端口

将本小节的内容总结一下，就得到了这个中断控制配置端口表。

表6.6 中断控制配置端口表

P_INT_Ctrl_New(写)	P_INT_Ctrl (读)	P_INT_Clear(写)	功能
1	—	—	允许中断/唤醒功能
0	—	—	屏蔽中断/唤醒功能，但不清除 P_INT_Ctrl (读)单元相应的中断标志位
—	1	—	有中断事件发生
—	0	—	没有中断事件发生
—	—	1	清除中断事件
—	—	0	不改变中断源的状态

6.2.2 中断响应过程

从中断请求发生到被响应,从中断响应到转向执行中断服务程序,完成中断所要求的操作任务,是一个复杂的过程。整个过程都是在 CPU 的控制下有序进行的,下面按顺序叙述 SPCE061A 单片机中断响应过程。

1．中断查询

SPCE061A 的设计思想是把所有的中断请求都汇集到 P_INT_Ctrl 和 P_UART_Command2（该寄存器用于检测串行传输中断标志位）寄存器中。其中外中断是使用采样的方法将中断请求锁定在 P_INT_Ctrl 寄存器的相应标志位中，而其它的中断请求由于都发生在芯片的内部，可以直接去置位 P_INT_Ctrl 和 P_UART_Command2 中各自的中断请求标志,不存在采样的问题,所谓查询就是由 CPU 测试 P_INT_Ctrl 和 P_UART_Command2 中各标志位的状态，以确定有没有中断请求发生以及是哪一个中断请求。中断请求汇集使中断查询变得简单，因为只需对两寄存器查询即可。

SPCE061A 中断查询发生在每一个指令周期结束后，按中断优先级顺序对中断请求进行查询，即先查询高级中断后，再查询低级中断，即先查询 FIQ 再查询 IRQ，同级中断按 IRQ0→IRQ1→IRQ2→IRQ3→IRQ4→IRQ5→IRQ6→UART 的顺序查询。如果查询到有标志位为“1”，则表明有中断请求发生。因为中断请求是随机的发生的，CPU 无法预先得知，因此在程序执行过程中，中断查询要在每个指令结束后不停的进行。

2．中断响应

中断响应就是 CPU 对中断源提出的中断请求的接受，是在中断查询后进行的，当

查询到有效的中断请求时，紧接着就进行中断响应。中断响应的主要内容可以理解为是硬件自动生成一条调用指令，其格式为 CALL addR16，这里的 addR16 就是存储器中断区中相应中断入口地址。在 SPCE061A 单片机中，这些入口地址已经由系统设定，例如，对于时基信号 2Hz 中断的响应，产生的调用指令为“ CALL 0xFFFD ”。生成 CALL 指令后，紧接着就由 CPU 执行，首先将程序计数器 PC 的内容压入堆栈，然后，再将 SR 压入堆栈，以保护断点，再将中断入口地址装入 PC，使程序执行转向相应的中断区入口地址，调用中断服务程序。

中断响应是有条件的，并不是查询到所有中断请求都能被立即响应。当 CPU 正处在为一个同级或高级的中断服务时，中断响应被封锁。因为当一个中断被响应时，要求把对应的优先级触发器置位，封锁低级和同级中断。

中断响应是需要时间的。中断响应的时间应该是从中断信号出现到 CPU 响应的时间与 CPU 响应中断信号到进入中断服务程序的时间之和。首先中断信号出现，CPU 查询到后，再执行下一条指令结束后去响应中断，这个时间可以根据指令周期长短来确定；指令周期最长为 182 个时钟周期，原因是累乘加指令需要的时间最长为 182 个时钟周期；其次 CPU 响应中断后，到 CPU 执行中断服务程序又需要 8 个时钟，原因是需要堆栈 PC 指针和 SR 寄存器及将中断向量赋值给 PC 及跳转到中断服务程序，这些操作共需要 8 个时钟周期。因此,SPCE061A 从中断信号出现到进入中断服务最长需要 190 个时钟周期。当然，如果出现有同级或高级中断正在响应或服务中须等待的时候，那么响应时间是无法计算的。

在一般应用情况下，中断响应时间的长短通常无需考虑。只有在精确定时应用场合，才需要知道中断响应时间，以保证定时的精确控制。

3．中断请求的撤销

中断响应后，P_INT_Ctrl 和 P_UART_Command2 中的中断请求标志应及时清除。否则就意味着中断请求仍然存在，弄不好就会造成中断的重复查询和响应，因此就存在一个中断请求的撤销问题。在 SPCE061A 中断中，中断撤销只是将标志位清“0”的问题。SPCE061A 中断除 UART 中断外，所有的中断均需软件清除标志位，即将 P_INT_Ctrl 中相应的中断位清零。即可将中断请求撤销。而 UART 中断，则是硬件自动清零，不需要软件操作。如当接收到数据后，P_UART_Command2 中的接收标志位自动置“1”，进入 UART 中断，在 UART 中断中读出数据，P_UART_Command2 相应的中断标志位自动清零。

4．中断服务流程

SPCE061A 单片机的中断服务流程如图 6.2 所示。

1.中断入口

所谓中断的入口即中断的入口地址，每个中断源都有自己的入口地址，这一点在上文已经提到。中断入口地址表如表 6.2 所示。当 CPU 响应中断后，就是通过中断入口地址进入中断服务程序。

2.关中断和开中断

在一个中断执行过程中又可能有新的中断请求，但对于重要的中断必须执行到底，不允许被其它的中断所嵌套。如 FIQ 中断，对此，可以采用关闭中断的方法来解决，如在 IRQ 中断中不允许 FIQ 中断嵌套，就可以在 IRQ 中断中关闭中断，当中断服务程序执行结束后，再打开中断，去响应 FIQ 中断。即在现场保护之前先关闭中断系统，彻底屏蔽其它中断请求，待中断处理完成后再打开中断系统。

还有一种情况是中断处理可以被打扰，但现场的保护和恢复不允许打扰，以免现场被破坏，为此应在现场保护和现场恢复的前后进行开关中断。这样做的结果是除现场保护和现场恢复的片刻外，仍然保持着系统中断嵌套功能，对于 SPCE061A 单片机中断的开和关可通过中断控制指令来控制 IRQ 和 FIQ 中断，如果想实现单个中断源的控制，可以通过 P_INT_Ctrl 控制寄存器来置位和清位来打开或关闭某个中断。

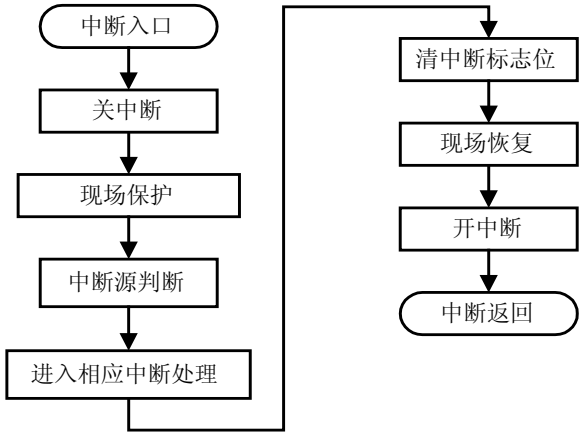

图6.2 中断服务流程图

3．现场保护和现场恢复

所谓现场是指中断时刻单片机中存储单元中的数据状态。为了使中断服务的执行不破坏这些数据或状态，以免在中断返回后影响主程序的运行，因此要把它们送入堆栈中保存起来，这就是现场保护。现场保护一定要位于中断处理程序的前面。中断服务结束后，在返回主程序前，则需要把保存的现场内容从堆栈中弹出，以恢复那些存储单元的原有内容，这就是现场恢复。现场恢复一定要位于中断处理程序的后面。

4．中断源判断

因为 SPCE061A 中断源多于中断入口地址，所以当 CPU 响应中断后，经中断入口地址进入中断服务程序，通过读 P_INT_Ctrl 可判断出产生中断请求的中断源。

5.中断处理

中断处理是中断服务程序的核心内容，是中断的具体目的。

6.清中断标志位

因为 CPU 是根据中断标志位来判断并进行响应中断的，除串口中断外，所有的中断标志位不是靠硬件清除，而是软件清除的，所以在中断服务程序中，必须将中断标志清除，否则 CPU 总是会响应该中断的。

7.中断返回

中断服务程序最后一条指令必须是中断返回指令 RETI，当 CPU 执行这条指令时，从堆栈中弹出断点 PC、及 SR，恢复断点重新执行被中断的程序。

6.3 中断控制的相关 C 函数

SPCE061.lib 中提供了相应的 API 函数如下所示：

函数原型

```
void    Set_INT_Ctrl(unsigned int);
```

void　Set_INT_Mask(unsigned int);

功能说明　Set INT Enable

用法

Set_INT_Ctrl(INT_Information);

Set_INT_Mask(INT_Information);

参数

INT_Information

函数原型

unsigned int Get_INT_Mask(void);

功能说明　Get INT Enabling Information

用法

INT_Information = Get_INT_Mask();

返回值

INT_Information

函数原型

unsigned int　Get_INT_Ctrl(void);

功能说明　Get INT Flag

用法

INT_Flag = Get_INT_Ctrl();

返回值

INT_Flag

函数原型

void　INT_Clear(unsigned int);

功能说明　Clear INT Flag

用法

INT_Clear(INT_Flag);

参数

INT_Flag

INT Enabling Information →

INT Flag →

C_IRQ6_TMB2	→	Timer B IRQ6	(b0)
C_IRQ6_TMB1	→	Timer A IRQ6	(b1)
C_IRQ5_2Hz	→	IRQ5 2 Hz	(b2)
C_IRQ5_4Hz	→	IRQ5 4 Hz	(b3)
C_IRQ4_1KHz	→	1024Hz IRQ4	(b4)
C_IRQ4_2KHz	→	2048Hz IRQ4	(b5)
C_IRQ4_4KHz	→	4096Hz IRQ4	(b6)
C_IRQ3_KEY	→	Key Change IRQ3	(b7)
C_IRQ3_EXT1	→	Ext1 IRQ3	(b8)
C_IRQ3_EXT2	→	Ext2 IRQ3	(b9)

C_IRQ2_TMB	→	Timer B IRQ2	(b10)
C_FIQ_TMB	→	Timer B FIQ	(b11)
C_IRQ1_TMA	→	Timer A IRQ1	(b12)
C_FIQ_TMA	→	Timer A FIQ	(b13)
C_IRQ0_PWM	→	PWM IRQ0	(b14)
C_FIQ_PWM	→	PWM FIQ	(b15)

函数原型

```
void FIQ_ON(void);
void FIQ_OFF(void);
void IRQ_ON(void);
void IRQ_OFF(void);
void INT_FIQ(void);
void INT_IRQ(void);
void INT_FIQ_IRQ(void);
void INT_OFF(void);
```

功能说明　INT On/Off Function

用法

FIQ_ON();	→	FIQ ON
FIQ_OFF();	→	FIQ OFF
IRQ_ON();	→	IRQ ON
IRQ_OFF();	→	IRQ OFF
INT_FIQ();	→	FIQ ON，IRQ OFF
INT_IRQ();	→	FIQ OFF，IRQ ON
INT_FIQ_IRQ();	→	FIQ ON，IRQ ON
INT_OFF();	→	FIQ OFF，IRQ OFF

6.4 中断系统的应用实例

6.4.1 单中断源的应用

例 6.1 定时器中断

要求：利用定时器 A 定时 10ms，在 A 口的 IOA0 脚输出周期 20ms 的方波。

分析：采用定时器 A 定时，首先解决使用那种中断方式，是 FIQ 中断方式，还是 IRQ 中断方式。当然在这个例子中，采用哪一种中断都可以，这里我们采用 IRQ 中断，即开中断时需打开 IRQ1 中断即定时器 A 中断；即将 P_INT_Ctrl 的 IRQ1_TMA 置位。其次考虑定时 10ms 的问题；确定定时 10ms，首先考虑采用的时钟源（P_TimerA_Ctrl），这里采用 8kHz 的时钟源 A，时钟源 B 为 1。当然选用其他频率也可以定时 10ms；接下来确定定时器 A 的预置数(P_TimerA_Data)是多少?

P_TimerA_Data = 0xFFFF – (Source A & Source B Frequency) / Desired Frequency = 0xFFFF-8k/0.1k = 0xFFFF-80 = 0xFFAF。

程序代码如下，读者可以看看和上一章利用 TimerA 产生方波的例子有什么区别。

主程序如下：

```
#include "SPCE061.H"
main()
{
    *P_IOA_Dir=0xFFFF;
    *P_IOA_Attrib=0xFFFF;
    *P_IOA_Data=0xFFFF;

    *P_TimerA_Ctrl=C_SourceA_8192Hz+C_SourceB_1;   //TimerA:8192Hz
    *P_TimerA_Data=0xFFAF;                         //10 ms

    *P_INT_Ctrl=C_IRQ1_TMA;
    __asm("INT IRQ");

    while(1)
    *P_Watchdog_Clear = C_WDTCLR;
}
```

中断服务程序如下：

```
#include "SPCE061.H"
unsigned int uiOutput=0xFFFF;
void IRQ1(void) __attribute__ ((ISR));
void IRQ1(void)
{
    uiOutput ^= 0xFFFF;                    //reverse
    *P_IOA_Data=uiOutput;                  //output rectangle
    *P_INT_Clear=C_IRQ1_TMA;                  //clear  INT  flag
}
```

例 6.2 时基中断

要求：定时 0.5s，使 A 口的 8 个二极管闪烁。

分析：定时 0.5s 采用哪个时基信号比较方便呢？SPCE061A 单片机时基信号频率丰富，有 2Hz、4Hz、8Hz、16Hz、32Hz、64Hz、128Hz、256Hz、512Hz、1024H、2048Hz、4096Hz 等多种频率。我们可以很明显的看出 2Hz 时基信号中断是最方便的。只要触发 2Hz 的时基信号中断，就可以达到 0.5s 的定时目的。

下面附有本例的程序代码，读者仔细和上一例比较一下，看看区别在什么地方。

主程序代码如下：

```
#include "SPCE061.H"
main()
{
    asm("INT  OFF");
                                           //output
    *P_IOA_Dir=0x00FF;
    *P_IOA_Attrib=0x00FF;
```

```
    *P_IOA_Data=0x0000;

    *P_INT_Ctrl=C_IRQ5_2Hz;                    //Setup  interrupt

    asm("INT  IRQ");

    while(1)
        *P_Watchdog_Clear  =  C_WDTCLR;
}
```

中断服务子程序代码如下：

```
#include "SPCE061.H"
unsigned int g_uiOutput=0x0000;
void  IRQ5(void)  __attribute__  ((ISR));
void  IRQ5(void)
{
    if(*P_INT_Ctrl&0x0004)
    {    //IRQ5_2Hz
        *P_IOA_Data=g_uiOutput;
        g_uiOutput^=  0xffff;
        *P_INT_Clear=0x0004;
    }
    else
    {    //IRQ5_4Hz
        *P_INT_Clear=0x0008;
    }
}
```

例 6.3 触键唤醒中断

触键唤醒中断源主要是在系统进入睡眠状态后，通过 A 口低八位的按键来唤醒系统时钟。同时进入触键唤醒中断。恢复睡眠时的 PC 指针。

要求：使系统平时处于睡眠状态，可通过触键唤醒，唤醒后逐个点亮 8 个指示灯，然后熄灭 8 个指示灯，继续进入睡眠状态。

分析：首先考虑如何使系统进入睡眠状态，可以通过设置时钟系统控制寄存器（P_SystemClock）的 B4 位使系统进入睡眠状态。其次因为只有 A 口的低 8 位具有唤醒功能，所以按键必须接 A 口低 8 位。8 个 LED 接 A 口高 8 位。

主程序代码如下：

```
#include "SPCE061.H"
main()
{
    unsigned int uiTem;
    __asm("INT OFF");
```

```
    *P_IOA_Dir = 0xff00;                //IOA0~IOA7:input;IOA8~IOA15:output
    *P_IOA_Attrib = 0xff00;
    *P_IOA_Data = 0x0000;

    *P_INT_Ctrl = C_IRQ3_KEY;           //Open IRQ3_Key interrupt
    __asm("INT IRQ");

    uiTem = *P_IOA_Latch;               //Latch IOA0~IOA7

    while(1)
    {
        *P_IOA_Data = 0xff00;           //Turn off LED
        *P_SystemClock = C_Sleep;       //Sleep;Wakeup when Key change
    }
}
```

中断服务子程序如下：

```
#include "SPCE061.H"
unsigned int uiOutput,uiDelay;
void IRQ3(void) __attribute__ ((ISR));
void IRQ3(void)
{
    if(*P_INT_Ctrl&0x0100)
    {   //IRQ3_Ext1
        *P_INT_Clear=0x0100;
    }
    else if(*P_INT_Ctrl&0x0200)
    {   //IRQ3_Ext2
        *P_INT_Clear=0x0200;
    }
    else
    {   //IRQ3_KeyWakeUp
        for(uiOutput=0x7fff;uiOutput>0x007f;uiOutput>>=1)
        {
            *P_IOA_Data=uiOutput;
            uiDelay=0x7fff;
            while(uiDelay--);
            *P_Watchdog_Clear = 0x0001;
        }
        *P_INT_Clear=0x0080;
    }
}
```

例 6.4 外部中断

SPCE061A 有两个外部中断，为负跳沿（或者称为下降沿）触发。可以使用形成反馈电路定时来触发外部中断，也可以不使用反馈电路，通过给 IOB2 或 IOB3 外部中断触发信号，进入外部中断

要求：在外部中断中控制连接在 A 口低八位的 8 个 LED

分析：首先考虑使用外部中断 1 还是外部中断 2，此处两个外部中断都可以。只是初始化时略有不同，选择外部中断 1，初始化 IOB2 为带上拉电阻的输入端口，选择外部中断 2，初始化 IOB3 为带上拉电阻的输入端口位高阻输入。此例选择外部中断 1。

A 口低八位接 LED，IOB2 接键盘。程序运行后，按键一次，LED 亮，再按一次，LED 灭，如此可反复操作。

主程序代码如下：

```
#include "SPCE061.H"
main()
{
    *P_IOA_Dir=0xff;                    //IOA0~IOA7 output
    *P_IOA_Attrib=0xff;
    *P_IOA_Data = 0x00;

    *P_IOB_Dir=0x00;                    //IOB2,input
    *P_IOB_Attrib=0x00;
    *P_IOB_Data=0x04;
    *P_INT_Ctrl=C_IRQ3_EXT1;            //Open IRQ3_EXT1 interrupt
    __asm("INT IRQ");
    while(1)
    {
        *P_Watchdog_Clear = C_WDTCLR;
    }
}
```

中断服务子程序代码如下：

```
#include "SPCE061.H"
unsigned int g_uiOutput=0;
void IRQ3(void) __attribute__ ((ISR));
void IRQ3(void)
{
    unsigned int uiDelay;
    if(*P_INT_Ctrl&C_IRQ3_EXT1)
    {    //IRQ3_Ext1
         *P_IOA_Data=(g_uiOutput^=0xffff);
         uiDelay=0x7fff;
         while(uiDelay--);
         *P_INT_Clear=C_IRQ3_EXT1;
    }
    else if(*P_INT_Ctrl&C_IRQ3_EXT2)
```

```
    {    //IRQ3_Ext2
         *P_INT_Clear=C_IRQ3_EXT2;
    }
    else
    {    //IRQ3_KeyWakeUp
         *P_INT_Clear=C_IRQ3_KEY;
    }
}
```

6.4.2 多中断源应用

在单片机软件开发中，使用多个中断源的机会有很多。在 SPCE061A 单片机中多中断源的使用有两种方式，一种是同个中断入口中断源的使用；另一种是不同中断入口的多个中断源的使用。

例 6.5 同中断向量的多个中断源使用

要求：IRQ6 中断有两个中断源 IRQ6_TMB1 和 IRQ6_TMB2，利用两个中断源分别控制 4 个发光二极管，分别为 1s 和 0.5s 闪烁。

分析：IRQ6_TMB1 有多种选择 8，16，32，64Hz 选择，可以选择其中任何频率均可做 0.5s 定时。此处我们选择 64Hz；同样 IRQ6_TMB2 有 128，256，512，1024Hz 选择，每种频率都可以达到定时 1s，此处选择 128Hz。

本例的主程序的代码如下：

```
#include "SPCE061.H"
main()
{
    *P_IOA_Dir=0xffff;                                  //IOA0~IOA3 initial: output
    *P_IOA_Attrib=0xffff;
    *P_IOA_Data=0x0000;

    *P_IOB_Dir=0xffff;                                  //IOB0~IOB3 initial: output
    *P_IOB_Attrib=0xffff;
    *P_IOB_Data=0x0000;

    *P_TimeBase_Setup=C_TMB1_64Hz|C_TMB2_128Hz;//Select TimeBase
    *P_INT_Ctrl=C_IRQ6_TMB1|C_IRQ6_TMB2;          //Open interrupt
    __asm("int irq");
    while(1)
        *P_Watchdog_Clear=0x0001;
}
```

中断服务程序代码如下：

```
#include "SPCE061.H"
unsigned int g_uiTime1,g_uiTime2;
```

```
void IRQ6(void) __attribute__ ((ISR));
void IRQ6(void)
{
    if(*P_INT_Ctrl&0x0001)
    {   //TMB2
        g_uiTime2+=1;
        if(g_uiTime2<=64)                   //g_uiTime2=64----0.5 second
            *P_IOB_Data=0x0000;
        else if(g_uiTime2>128)              //g_uiTime2=128-----1 second
            g_uiTime2=0;
        else
            *P_IOB_Data=0x000f;
        *P_INT_Clear=C_IRQ6_TMB2;
    }
    else
    {   //TMB1
        g_uiTime1+=1;
        if(g_uiTime1<=64)                   //g_uiTime1=64----1 second
            *P_IOA_Data=0x0000;
        else if(g_uiTime1>128)              //g_uiTime1=128---2 second
            g_uiTime1=0;
        else
            *P_IOA_Data=0x000f;
        *P_INT_Clear=C_IRQ6_TMB1;
    }
}
```

例 6.6 不同中断入口的中断源使用

要求：用不同中断入口的中断源，利用分别控制 4 个发光二极管，分别为 1s 和 0.5s 闪烁。

分析： 可以采用定时器中断也可以采用时基中断。此例利用的是 0.5s 定时使用的是 IRQ2 中的定时器 B；1s 定时利用的是 IRQ4 中的 IRQ_1kHz 中断。

本例的主程序的代码如下：

```
#include "SPCE061.H"
#define  TIMER_DATA_FOR_4KHZ   (0xffff - 2048)  //0.5 second
main()
{
    *P_IOA_Dir=0xffff;                          //IOA0~IOA4 initial: output
    *P_IOA_Attrib=0xffff;
    *P_IOA_Data=0x00ff;

    *P_IOB_Dir=0xffff;                          //IOB0~IOB4 initial: output
    *P_IOB_Attrib=0xffff;
    *P_IOB_Data=0x00ff;
```

```
    *P_TimerB_Data=TIMER_DATA_FOR_4KHZ;     //TimerB setup
    *P_TimerB_Ctrl=C_SourceA_4096Hz;

    *P_INT_Ctrl=C_IRQ4_1KHz|C_IRQ2_TMB;       //Open interrupt
    __asm("int irq");
    while(1)
        *P_Watchdog_Clear=0x0001;
}
```

中断服务子程序代码如下：

```
#include "SPCE061.H"
unsigned int g_uiIOA_LED=0xff,g_uiIOB_LED=0xff,g_uiClockCnt=0;
void IRQ2(void) __attribute__ ((ISR));
void IRQ2(void)
{
    *P_IOB_Data=g_uiIOB_LED;
    g_uiIOB_LED ^= 0xffff;
    *P_INT_Clear=0x0400;
}
void IRQ4(void) __attribute__ ((ISR));
void IRQ4(void)
{
    if(*P_INT_Ctrl&C_IRQ4_1KHz)
    {
        if(g_uiClockCnt<1024)
            g_uiClockCnt++;
        else
        {
            *P_IOA_Data=g_uiIOA_LED;
            g_uiIOA_LED ^= 0xffff;
            g_uiClockCnt = 0x0000;
            *P_INT_Clear=C_IRQ4_1KHz;
        }
    }
    if(*P_INT_Ctrl&C_IRQ4_2KHz)
    {
        *P_INT_Clear=C_IRQ4_2KHz;
    }
    if(*P_INT_Ctrl&C_IRQ4_4KHz)
    {
        *P_INT_Clear=C_IRQ4_4KHz;
    }
}
```