第2章 SPCE061A 单片机硬件结构

从第一章中 SPCE061A 的结构图可以看出 SPCE061A 的结构比较简单，在芯片内部集成了 ICE 仿真电路接口、FLASH 程序存储器、SRAM 数据存储器、通用 I/O 端口、定时器/计数器、中断控制、CPU 时钟、模-数转换器 A/D、DAC 输出、通用异步串行输入输出接口、串行输入输出接口、低电压监测/低电压复位等若干部分。各个部分之间存在着直接或间接的联系，在本章中我们将详细的介绍每个部分结构及应用。

2.1 μ'nSP™的内核结构

μ'nSP™的内核如0所示其结构。它由总线、算术逻辑运算单元、寄存器组、中断系统及堆栈等部分组成，右边文字为各部分简要说明。

算术逻辑运算单元 ALU

μ'nSP™的 ALU 在运算能力上很有特色，它不仅能做 16 位基本的算术逻辑运算，也能做带移位操作的 16 位算术逻辑运算,同时还能做用于数字信号处理的 16 位×16 位的乘法运算和内积运算。

1．16 位算术逻辑运算

不失一般性，μ'nSP™与大多数 CPU 类似，提供了基本的算术运算与逻辑操作指令，加、减、比较、取补、异或、或、与、测试、写入、读出等 16 位算术逻辑运算及数据传送操作。

2．带移位操作的 16 位算逻运算

对图 2.1 稍加留意，就会发现μ'nSP™的 ALU 前面串接有一个移位器 SHIFTER，也就是说，操作数在经过 ALU 的算逻操作前可先进行移位处理，然后再经 ALU 完成算逻运算操作。移位包括：算术右移、逻辑左移、逻辑右移、循环左移以及循环右移。所以，μ'nSP™的指令系统里专有一组复合式的‘移位算逻操作’指令；此一条指令完成移位和算术逻辑操作两项功能。程序设计者可利用这些复合式的指令，撰写更精简的程序代码，进而增加程序代码密集度 (Code Density)。在微控制器应用中，如何增加程序代码密集度是非常重要的议题；提高程序代码密集度意味着：减少程序代码的大小，进而减少 ROM 或 FLASH 的需求，以此降低系统成本与增加执行效能。

3．16 位×16 位的乘法运算和内积运算

除了普通的 16 位的算逻运算指令外，μ'nSP™的指令系统还提供处理速度较高的 16 位×16 位的乘法运算指令 Mul 和内积运算指令 Muls。二者都可以用于两个有符号数或一个有符号数与一个无符号数的运算。在 ISA1.1 指令集下，Mul 指令只需花费 12 个时钟周期，Muls 指令花费 10n+6 个时钟周期，其中 n 为乘积求和的项数。例如：“MR=[R2]*[R1] ,4”表示求 4 项乘积的和，Muls 指令只需花费 46（10×4+6=46）个时钟周期。这两条指令为μ'nSP™应用于复杂的数字信号处理运算方面提供了便利的条

件。

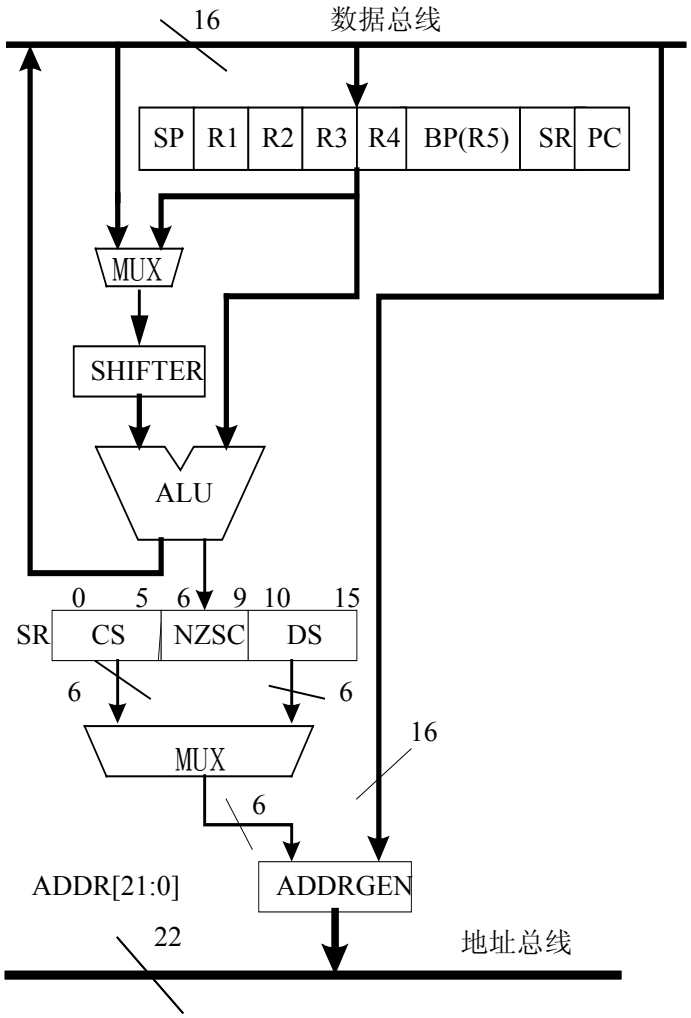


图 2.1　μ'nSP™的内核结构

寄存器组

μ'nSP™的 CPU 寄存器组里有 8 个 16 位寄存器，可分为通用型寄存器和专用型寄存器两大类别。通用型寄存器包括：R1~R4，作为算术逻辑运算的源及目标寄存器。专用型寄存器包括 SP、BP、SR、PC，是与 CPU 特定用途相关的寄存器。

1.　通用型寄存器 R1~R4

通常可分别用于数据运算或传送的源及目标寄存器。而寄存器 R4、R3 配对使用还可组成一个 32 位的乘法结果寄存器 MR；其中 R4 为结果的高字组，R3 为结果的低字组，用于存放乘法运算或内积运算结果。

2.　堆栈指针寄存器 SP

SP 是在 CPU 执行压栈/出栈指令（push/pop）、子程序调用/返回指令（call/retf）以及进入中断服务子程序(ISR，Interrupt Service Routine) 或从 ISR 返回指令（reti）时自动

减少（压栈）或增加（弹栈），以示堆栈指针的移动。堆栈的最大容量范围限制在 2K 字的 RAM 内，即地址为 0x000000~0x0007FF 的存储器范围中。

3.　基址指针寄存器 BP

μ'nSP™提供了一种方便的寻址方式，即变址寻址方式[BP+IM6]；程序设计者可通过它直接存取 ROM 与 RAM 中的各种数据，包括：局部变量（Local Variable）、函数参数（Function Parameter）、返回地址（Return Address）等等；这在 C 语言程序中是特别有用的。BP 除了上述用途外，也可做为通用寄存器 R5 用于数据运算或传送的源及目标寄存器。因此，在本书或程序中，BP 与 R5 是共享的，均代表基址指针寄存器。

4.　段寄存器 SR

有多种功能用途，如图 2.1 所示。SR 中有代码段选择字段(CS)和数据段选择字段(DS)，它们可分别与其它 16 位的寄存器合在一起形成 22 位地址线，用来寻址 4M 字容量的存储器。（注意： SPCE061A 只有 32K 字闪存，占一页存储空间，所以代码段选择字段(CS)和数据段选择字段(DS) 在 SPCE061A 不用）。

算逻运算结果的各标志位 NZSC 亦储存于其中，即 SR 中间的 4 位（B6~B9）。CPU 在执行条件/无条件短跳转指令（JUMP）时需测试这些标志位以控制程序的流向。这些标志位的内容是：

进位标志 C

C=0 时表示运算过程中无进位或有借位产生，而 C=1 表示运算过程中有进位或无借位产生。在无符号数运算中，16 位数可以表示的数值范围是 0x0000~0xFFFF，即 0~65535。如果运算结果大于 65535(0xFFFF)，则标志位 C 被置 1。请注意：标识位 C 一般用于无符号数运算的进、借位判断。

零标志 Z

Z=0 时表示运算结果不为 0，Z=1 时表运算结果为 0。

负标志 N

标志位 N 是用来判断运算结果的最高位（B15）为 0 还是为 1。B15=0 则 N=0；B15=1 则 N=1。

符号标志 S

S=0 时表示运算结果不为负，S=1 时则表示运算结果（在二进制补码的规则下）为负。对于有符号数运算，16 位数所表示的数值范围是为 0x8000~0x7FFF，即-32768~32767。若运算结果小于零，则标志位 S 置 1。有符号数运算的运算结果可能会大于 0x7FFF 或小于 0x8000。比如：0x7FFF+0x7FFF=0xFFFE（65534），运算结果为正（S=0），且无进位（C=0）发生；在此情况下，标志位 N 被置 1（因为最高有效位为 1）。若标志位 N 和 S 不同，则说明有溢出发生，即：S=0，N=1 或 S=1，N=0。例如当为有符号数时,可判断正负。而 JVC(N==S)，JVS(N!=S)则可用来判断 overflow。请注意：N,S 的组合用于有符号数溢出的判断。

这里，需特别提醒注意：在运算操作过程中，若目标寄存器是 PC，则所有标志位均不会受到影响。

总结：

[1]．由于补码可以把有符号数与无符号数的运算统一起来，所以对于同一条加法或减法指令，既可以认为是有符号数运算又可以认为是无符号数运算，只是观察的角度、判断的标准不同而已。

[2]．标识位 C 一般用于无符号数运算的进、借位判断。

[3]．N,S 的组合用于有符号数溢出的判断。

[4]．有符号数的范围为-32768~32767，无符号数的范围为 0~65535。若为有符号数，运算前数的正负应通过标识位‘N’判断；运算后结果的正负应通过标识位‘S’判断。

下面我们举几个例子来分析说明标识位

[例 2.1]: R1=32767，R2=32767，求二者之和。运算后 R1 中的内容为 0xFFFE。

```
R1=32767        //赋值后的标识位为 N=0,Z=0,S=0,C=1；R1=0x7FFF
R2=32767        //赋值后的标识位为 N=0,Z=0,S=0,C=1；R2=0x7FFF
R1+=R2          //运算后的标识位为 N=1,Z=0,S=0,C=0；R1=0xFFFE
```

若作为无符号数看待，此时：‘C’为 0 说明无进位产生；若作为有符号数看待，此时：N!=S 说明计算结果超出有符号数的范围，即产生溢出，另外‘S’为 0 说明运算结果不为负

[例 2.2]: R1=-12345，R2=-1，求二者之和。运算后 R1 中的内容为 0xCFC6。

```
R1=-12345       //赋值后的标识位为 N=1,Z=0,S=0,C=1,此时 R1=0xCfC7
R2=-1           //赋值后的标识位为 N=1,Z=0,S=0,C=1,此时 R2=0xFFFF
R1+=R2          //运算后的标识位为 N=1,Z=0,S=1,C=1,此时 R1=0xCfC6
```

若作为无符号数看待，此时：‘C’为 1 说明有进位产生；若作为有符号数看待，此时：‘N=S’说明无溢出产生，‘S’为 1 说明运算结果为负。

[例 2.3]：R1=32767，R2=-12345，求二者之差。运算后 R1 中的内容为 0xB038。

```
R1=32767        //赋值后的标识位为 N=0,Z=0,S=0,C=1, R1=0x7FFF
R2=-12345       //赋值后的标识位为 N=1,Z=0,S=0,C=1, R2=0xC7C7
R1-=R2          //运算后的标识位为 N=1,Z=0,S=0,C=0, R1=0xB038
```

若作为无符号数看待，此时：‘C’为 0，说明有借位产生；若作为有符号数看待此时‘N!=S’说明有溢出产生；‘S’为 0 说明运算结果为正。

5.　程序计数器 PC（Program Counter）

它的作用与所有微控制器中的 PC 作用均相同，是作为程序的地址指针来控制程序走向的专用寄存器。CPU 每执行完当前指令，都会将 PC 值累加当前指令所要占据的字节数或字数，以指向下一条指令的地址。在μ’nSP™里，16 位的 PC 通常与 SR 寄存器的 CS 选择字段共同组成 22 位的程序代码地址。

数据总线和地址总线

μ’nSP™是 16 位单片机，它具有 16 位数据线和 22 位地址线。由此决定其基本数据类型是 16 位的“word”型，而不是 8 位的“Byte”型；因而每次存储器都是按“word”操作的，22 位地址线最多可寻访 4M 字的存储容量。地址线中的高 6 位 A16~A21 来自段寄存器 SR 中的 6 位代码段（CS： Code Segment）和 6 位数据段（DS： Data Segment）选

择字段，低 16 位 A0~A15 则来自内部寄存器。通常，地址线的高 6 位称为存储器地址的页选，简称页码（Page）；而低 16 位则称为存储器地址的偏移量（Offset）。μ'nSP™通过对段（Segment）的编码来实现存储器页的检索，即是说‘Segment’的含义与‘Page’的含义是等同的。因而，通过 Segment 与 Offset 的配合即可产生 22 位地址线，如上图 2.1 中 ADDRGEN 所示。（注意： SPCE061A 只有 32K 字闪存 FLASH，仅占一页存储空间，所以代码段选择字段(CS)和数据段选择字段(DS) 在 SPCE061A 不用）

2.2 SPCE061A 片内存储器结构

SPCE061A 的片内存储器地址映射如图 2.2 所示。单片机的存储器有 2K 字的 SRAM（包括堆栈区）和 32K 字闪存（FLASH）。

2.2.1 RAM

SPCE061A 有 2K 字的 SRAM(包括堆栈区)，其地址范围从 0x0000 到 0x07FF。前 64 个字，即 0x0000～0x003F 地址范围内可采用 6 位地址直接地址寻址方法，寻访速度为 2 个 CPU 时钟周期；其余 0x0040～0x07FF 地址范围内存储器的寻访速度则为 3 个 CPU 时钟周期。

图 2.2 SPCE061 内存映射表

2.2.2 堆栈

堆栈是在内存 RAM 区专门开辟出来的按照“先进后出”原则进行数据存取的一种工作方式如图 2.2，主要用于子程序调用及返回和中断处理断点的保护及返回。堆栈的最大容量范围限制在 2K 字 RAM 内，即其地址范围从 0X07FF 到 0X0000 的存储器范围中。值得注意的是堆栈的生长方向，SPCE061A 系统复位后，SP 初始化为 0x07FF,每执行 PUSH 指令一次，SP 指针减一。

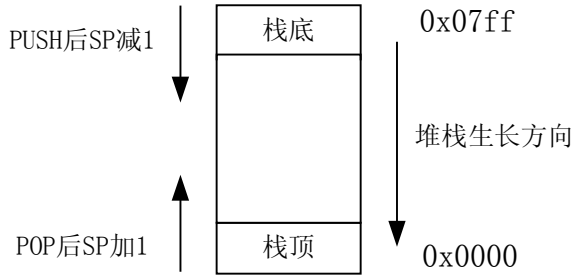


图 2.3　堆栈

2.2.3 闪存 Flash

SPCE061A 是一个用闪存替代掩膜 ROM 的 MTP(多次编程)芯片，具有 32K 字(32K*16bit)闪存容量。用户可用闪存来存储用户程序。为了安全起见，不对用户开放整体擦除功能。

用户必须通过向 P_Flash_Ctrl (写) ($7555H)单元写入 0xAAAA，来激活闪存的存取功能，从而访问闪存。然后，向 P_Flash_Ctrl (写) ($7555H)单元写入 0x5511，来擦除页的内容。写入 0x5533，对闪存编程。这些指令不能被任何其他的操作包括中断、ICE 的单步跟踪动作打断。这是因为闪存控制器必须保证闪存处于编程状态。如果一些其它的进程插入到当前的执行队列里，闪存的状态将发生改变，擦除页和编程的操作不能再继续进行。

此外，为保证程序的正确编写，用户必须在编程之前擦除页的内容。页大小为 0x100。第一页地址范围：0x8000~0x80FF，最后一页的地址范围：0xFF00~0xFFFF。0xFC00~0xFFFF 范围内的地址由系统保留，用户最好不要用本范围内的地址。

32K 字的内嵌式闪存被划分为 128 个页(每个页存储容量为 256 个字)，它们在 CPU 正常运行状态下均可通过程序擦除或写入。全部 32K 字闪存均可在 ICE 工作方式下被编程写入或被擦除。

2.2.3.1 读存储单元操作

在芯片上电以后，芯片就处于读存储单元状态，读存储单元的操作与 SRAM 相同。

2.2.3.2 擦除操作

在对闪存编程操作前，必须对闪存进行擦除操作。由于闪存采用模块分区的阵列结构，使得各个存储模块(页)可以被独立地擦除。当给出的地址是在模块地址范围之内且向命令用户接口写入模块擦除命令时，相应的模块就被擦除。要保证擦除操作的正确完成，必须考虑以下几个参数：

1. 该闪存的内部模块分区结构。
2. 每个模块分区的擦除时间。

2.2.3.3 编程操作

闪存芯片的编程操作是自动字节编程，既可以顺序写入，也可指定地址写入。编程操作时注意芯片的编程时间参数。Flash 程序空间为 0x8000—0xFFFF，Flash 命令用户接口地址为 0x7555 。第一页范围是[0x8000—0x80FF]，最后一页[0xFF00—0xFFFF]。

1. 擦除一页流程是：先给命令用户接口地址 0x7555 里送 0xAAAA，然后再给命令用户接口地址 0x7555 里送 0x5511，再后给要擦除页地址送任意数，约 20ms 即可完成擦除操作，然后可以执行其它操作。例如擦除第 6 页[0x8500—0x85FF]流程如下：(1)0x7555 ←0xAAAA (2) 0x7555 ←0x5511(3) 0x85XX←0xXXXX (其中 X 为任意值)。

2. 写入一个字流程是：先给命令用户接口地址 0x7555 里送 0xAAAA，然后再给命令用户接口地址 0x7555 里送 0x5533，再后给要写入字地址送数据，约 40us 即可完成写入操作，然后可以执行其它操作。例如向 0x8000 单元写入 0xffff 流程如下：（1）0x7555 ←0xAAAA (2) 0x7555 ←0x5533 (3) 0x8000 ←0xFFFF

3. 写多个字流程是：先给命令用户接口地址 0x7555 里送 0xAAAA，然后再给命令用户接口地址 0x7555 里送 0x5544，然后给要写入字首地址送数据，约 40us 即可完成 1 个字写入操作。再给命令用户接口地址 0x7555 里送 0x5544，给要写入字地址送数据，等待 40us 即可，循环操作，即可完成多字的写入。

*上面所提到延时等待是由硬件完成不需要软件延时，闪存的擦写过程如图 2.4 所示，

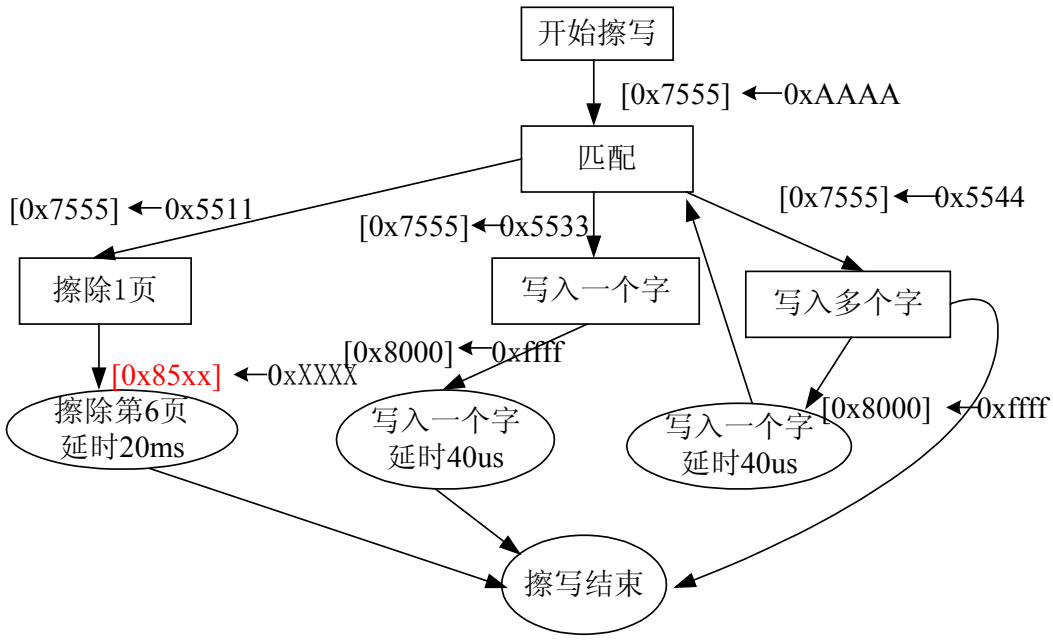


图 2.4　闪存的擦写过程

以上所介绍的擦除及写操作有一定的适应范围，一般情况下用于对数据的擦除。例如，当把一个程序段写入程序存储器后，由于程序的代码占用空间可能比较小（假如为 5K 字）我们就可以在程序代码段后面的空间（5K~32K）存储一些数据，这样就可以对这一段的空间（5K~32K）进行上述的写或擦除操作。前提条件：必须计算出程序代码占用的空间。

[例 2.4]：

```
//=================================================================
//程序名称：FLASH.asm
//描述：擦除、写 1 字、写多字子程序
//=================================================================
.INCLUDE hardware.inc

.DEFINE C_FLASH_SIZE          0x8000    //定义 flash 的空间为 32K 字
.DEFINE C_FLASH_BLOCK_SIZE    0x0100    //定义共分为 256 页
.DEFINE C_FLASH_MATCH         0xAAAA
.DEFINE C_FLASH_PAGE_ERASE    0x5511    //擦除一页控制字
.DEFINE C_FLASH_1WORD_PGM     0x5533    //写一个字的控制字
.DEFINE C_FLASH_SEQUENT_PGM   0x5544    //写多个字的控制字
.CODE
//=================================================================
//函数    名称:    F_FlashWrite1Word()
//描述：写一个字到 FLASH 中
//入口参数：1、被写数据的存储地址 2、被写数据
//无出口
//=================================================================
.public _F_FlashWrite1Word
.DEFINE P_Flash_Ctrl  0x7555
_F_FlashWrite1Word:    .proc
    PUSH BP TO [SP]                    //将 BP 压入栈内
    BP = SP + 1                        //BP 的值变为 SP+1

    R1 = C_FLASH_MATCH                 //先送 AAAA
    [P_Flash_Ctrl] = R1
    R1 = C_FLASH_1WORD_PGM             //再送 5533
    [P_Flash_Ctrl] = R1

    R1 = [BP+3]                        //取存储数据地址
    R2 = [BP+4]                        //取数据
    [R1] = R2                          //把数据写入内存单元中
    POP BP FROM [SP]                   //出栈
    RETF                               //子程序返回
    .ENDP

//=================================================================
//函数名称：    F_FlashWrite()
//描述：顺序写多个字
//入口参数：1、被写数据的起始地址 2、被写数据 3、写数据的数量
//无出口
//=================================================================
```

```
.public _F_FlashWrite
_F_FlashWrite:      .proc                   //子程序的开始
    PUSH BP TO [SP]                         //将 BP 压入栈内
    BP = SP + 1                             //BP 的值变为 SP+1

    R1 = [BP+3]                             //取被写数据的首地址
    R2 = [BP+4]                             //取被写数据
    R3 = [BP+5]                             //写 N 个字

    R4 = C_FLASH_MATCH                      //将 AAAA 送到控制单元
    [P_Flash_Ctrl] = R4
L_FlashWriteLoop:
    R4 = C_FLASH_SEQUENT_PGM                //把 5544 送到控制单元
    [P_Flash_Ctrl] = R4

    R4 = [R2++]                             //继续下一个数的写入
    [R1++] = R4
    R3 -= 1                                 //字计数减一
    JNZ L_FlashWriteLoop                    //不为 0 时写操作继续进行

    [P_Flash_Ctrl] = R3                     //写结束
    POP BP FROM [SP]                        //出栈
    RETF                                    //子程序返回
    .ENDP
//=================================================================//
//函数名称：     F_FlashErase()
//描述：擦除 256 字节
//入口参数:1、擦除页的起始地址
//=================================================================
.public _F_FlashErase
_F_FlashErase:      .proc                   //擦初一页的子程序
    PUSH BP TO [SP]                         //将 bp 压入栈内
    BP = SP + 1

    R1 = C_FLASH_MATCH                      //先将 AAAA 送到控制单元
    [P_Flash_Ctrl] = R1
    R1 = C_FLASH_PAGE_ERASE                 //再将 5511 送到控制单元
    [P_Flash_Ctrl] = R1

    R1 = [BP+3]                             //取擦除页内的地址
    [R1] = R1                               //写入任意值进行擦除
    POP BP FROM [SP]                        //出栈
    RETF                                    //子程序返回
    .ENDP
```

2.3 SPCE061A　输入/输出接口

输入/输出接口（也可简称为 I/O 口）是单片机与外设交换信息的通道。输入端口负责从外界接收检测信号、键盘信号等各种开关量信号。输出端口负责向外界输送由内部电路产生的处理结果、显示信息、控制命令、驱动信号等。μ'nSP™内有并行和串行两种方式的 I/O 口。并行口线路成本较高，但是传输速率也很高；与并行口相比，串行口的传输速率较低但可以节省大量的线路成本。SPCE061A 有两个 16 位通用的并行 I/O 口：A 口和 B 口。这两个口的每一位都可通过编程单独定义成输入或输出口。

A 口的 IOA0~IOA7 用作输入口时具有唤醒功能，即具有输入电平变化引起 CPU 中断功能。在那些用电池供电、追求低能耗的应用场合，可以应用 CPU 的睡眠模式（通过软件设置）以降低功耗，需要时以按键来唤醒 CPU，使其进入工作状态。例如：手持遥控器、电子字典、PDA、计算器、移动电话等。

I/O 端口结构

SPCE061A 提供了位控制结构的 I/O 端口，每一位都可以被单独定义用于输入或输出数据。通常，对某一位的设定包括以下 3 个基本项：数据向量 Data、属性向量 Attribution 和方向控制向量 Direction。3 个端口内每个对应的位组合在一起，形成一个控制字，用来定义相应 I/O 口位的输入输出状态和方式。例如，假设需要 IOA0 是下拉输入管脚，则相应的 Data、Attribution 和 Direction 的值均被置为“0”。如果需要 IOA1 是带唤醒功能的悬浮式输入管脚，则 Data、Attribution 和 Direction 的值被置为“010”。与其它的单片机相比，除了每个 I/O 端口可以单独定义其状态外，每个对应状态下的 I/O 端口性质电路都是内置的，在实际的电路中不需要再次外接。例：设端口 A 口为带下拉电阻的输入口，在连接硬件时无需在片外接下拉电路。

A 口和 B 口的 Data、Attribution 和 Direction 的设定值均在不同的寄存器里，用户在进行 I/O 口设置时要特别注意这一点。I/O 端口的组合控制设置如表 2.1所示：

表2.1 I/O 端口的组合控制设置

Direction	Attribution	Data	功能	是否带唤 唤醒功能	功能描述
0	0	0	下拉*	是**	带下拉电阻的输入管脚
0	0	1	上拉	是**	带上拉电阻的输入管脚
0	1	0	悬浮	是**	悬浮式输入管脚
0	1	1	悬浮	否	悬浮式输入管脚***
1	0	0	高电平输出 (带数据反相器)	否	带数据反相器的高电平输出 (当向数据位写入“0” 时输出“1”)
1	0	1	低电平输出 (带数据反相器)	否	带数据反相器的低电平输出 (当向数据位写入“1”时输出“0”)
1	1	0	低电平输出	否	带数据缓存器的低电平输出 (无数据反相功能)
1	1	1	高电平输出	否	带数据缓存器的高电平输出 (无数据反相功能)

注：

*：口位默认为带下拉电阻的输入管脚；

* *：只有当 IOA [7~0]内位的控制字为 000，001 和 010 时， 相应位才具有唤醒的功能。

***：此种悬浮输入作为 ADC IOA[6~0] 的输入

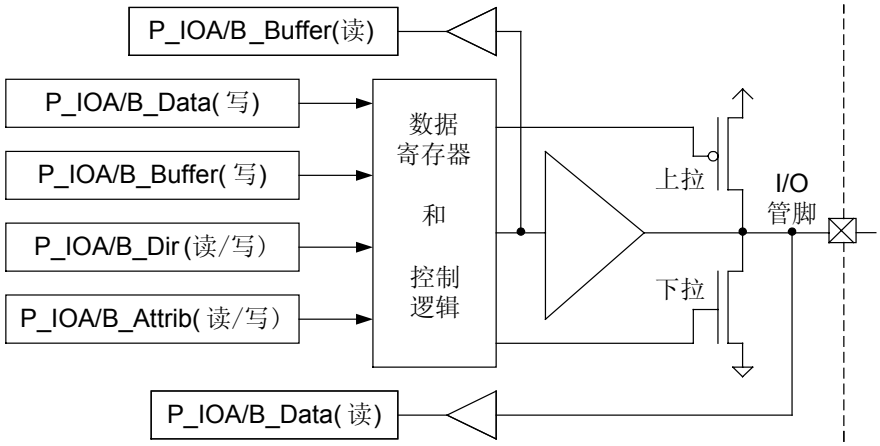


图 2.5　I/O 结构

P_IOA_Data(读/写)(7000H)

A 口的数据单元，用于向 A 口写入或从 A 口读出数据。当 A 口处于输入状态时，读出是读 A 口管脚电平状态； 写入是将数据写入 A 口的数据寄存器。当 A 口处于输出状态时，写入输出数据到 A 口的数据寄存器。

P_IOA_Buffer (读/写) (7001H)

A 口的数据向量单元，用于向数据向量寄存器写入或从该寄存器读出数据。当 A 口处于输入状态时，写入是将 A 口的数据向量写入 A 口的数据寄存器；读出则是从 A 口数据寄存器内读其数值。当 A 口处于输出状态时，写入输出数据到 A 口的数据寄存器。

对输出而言,P_IOA_Data 与 P_IOA_Buffer 是一样的.但对输入而言，P_IOA_Data 读的是 IO 的值，P_IOA_Buffer 读的是 buffer 内的值。假设 IOA[0]作为输出，并去接 LED 阳极 (LED 阴极接地)。若 P_IOA_Data 的 IOA[0]为 1。在某些需要较大驱动能力的 LED 而言,LED 会亮，但 IOA[0]会被拉到一个很低的值。此时从 P_IOA_Data 读回为 0，但 P_IOA_Buffer 则为 1。至于读回的意义是做什么，是方便做其它的 IO 运算。

P_IOA_Dir(读/写)(7002H)

A 口的方向向量单元，用于用来设置 A 口是输入还是输出，该方向控制向量寄存器可以写入或从该寄存器内读出方向控制向量。Dir 位决定了口位的输入/输出方向：即‘0’为输入，‘1’为输出。

P_IOA_Attrib(读/写)(7003H)

A 口的属性向量单元，用于 A 口属性向量的设置。

P_IOA_Latch(读)(7004H)

读该单元以锁存 A 口上的输入数据，用于进入睡眠状态前的触键唤醒功能的启动(参见睡眠/唤醒部分)。

并行 I/O 口的组合控制

方向向量_Dir、属性向量_Attrib 和数据向量_Data 分别代表三个控制口。这三个口中每个对应的位组合在一起，形成一个控制字，来定义相应 I/O 口位的输入/输出状态和方式。

表 2.1具体表示了如何通过对 I/O 口位的方向向量位_Dir、属性向量位_Attrib 以及数据向量位_Data 进行编程，来设定口位的输入/输出状态和方式。

由表 2.1可以得出以下一些结论：

_Dir 位决定了口位的输入/输出方向：即‘0’为输入，‘1’为输出。

_Attrib 位决定了在口位的输入状态下是为悬浮式输入还是非悬浮式输入：即‘0’为带上拉或下拉电阻式输入，而‘1’则为悬浮式输入。在口位的输出状态下则决定其输出是反相的还是同相的；‘0’为反相输出，‘1’则为同相输出。

_Data 位在口位的输入状态下被写入时，与_Attrib 位组合在一起形成输入方式的控制字‘00’、‘01’、‘10’、‘11’，以决定输入口是带唤醒功能的上拉电阻式、下拉电阻式或悬浮式以及不带唤醒功能的悬浮式输入。_Data 位在口位的输出状态下被写入的是输出数据，不过，数据是经过反相器输出还是经过同相缓存器输出要由_Attrib 位来决定。

例如，假设要把 A 口的 B0 定义成下拉电阻式的输入口，则 A 口_Dir、_Attrib 和_Data 向量的三个相应的 B0 应组合设为‘000’。如果想把 A 口的 B1 定义成悬浮式并具有唤醒功能的输入口，只需将_Dir、_Attrib 和_Data 向量中相应的 B1 组合设置为‘010’即可。

A 口的 IOA0~IOA7 作为唤醒源，常用于键盘输入。要激活 IOA0~IOA7 的唤醒功能，必须读 P_IOA_Latch 单元，以此来锁存 IOA0~IOA7 管脚上的键状态。随后，系统才可通过指令进入低功耗的睡眠状态。当有键按下时，IOA0~IOA7 的输入状态将不同于其在进入睡眠前被锁存时的状态，从而引起系统的唤醒。

[例 2.5]:设置端口

设置 IOA[3~0] 为带下拉电阻的输入口，IOA[7~4]为带上拉电阻的输入口，IOA[11~8]为带数据缓存器的高电平输出口，IOA[15~12] 为带数据缓存器的低电平输出口。

```
R1 = 0x0FF0;            //设置 A 口的数据向量，IOA0~IOA3 与 IOA12~IOA15 置高
[P_IOA_Data] = R1;      // IOA4~IOA7 与 IOA8~IOA11 置低
R1 = 0xFF00;            //设置 A 口的属性向量，IOA0~IOA8 置低，IOA12~IOA15
[P_IOA_Attrib] = R1;    //置高
R1 = 0xFF00;            //设置 A 口的属性向量，IOA0~IOA8 置低，IOA12~IOA15
[P_IOA_Dir] = R1;       //置高
```

各对应口位的设置如下表 2.2：

表2.2　I/O 口位

地址	——	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
7002H	Dir	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
7003H	Attrib	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
7000H	Data	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
——	状态	带数据缓存器的低电平输出				带数据缓存器的高电平输出				带上拉电阻的输入				带下拉电阻的输入			

从上面的分析我们可以看出，当作为输入口时所读的数值来自不同的地方，读 P_IOA_Data 是 A 口的管脚上的当前状态，读 P_IOA_Buffer 的值来自数据寄存器（见例 2.6）；当 A 口作为输出口时，数据都是写到 A 口的数据寄存器。对于某些 I/O 端口的应用，这种读写方式可以节省许多用于存放端口数据的 RAM 空间。

[例 2.6]：程序说明：在单步运行程序期间将 A 口的任意管脚接 VDD，通过观察寄存器表中的 R2，R3 的值可观察到二者的不同。

```
.INCLUDE    hadware.inc              //包含头文件
.CODE
.PUBLIC _main                        //主程序
_main:
        R1=0x0000                    //设置 A 口为带下拉电阻的输入口
        [P_IOA_Data]=R1              //设置 A 口的数据向量
        [P_IOA_Attrib]=R1            //设置 A 口的属性向量
        [P_IOA_Dir]=R1               //设置 A 口的方向向量
//*****************此时将 A 口的任意一口接高电平*********************//
        R3=[P_IOA_Data]              //把 P_IOA_Data 中的值送到寄存器 R3
        R2=[P_IOA_Buffer]            //把 P_IOA_Buffer 中的值送到寄存器 R2
WAIT:
    JMP WAIT                         //主程序循环
```

[例 2.7]：

```
.DEFINE    P_IOB_Data       0x7005
.DEFINE    P_IOB_Dir        0x7007
.DEFINE    P_IOB_Attrib     0x7008
```

```
        .DEFINE    P_IOB_Buffer      0x7006
    .CODE
    .PUBLIC _main                    //主程序
     _main:
        R1=0xFFFF                    //设置 B 口为带反向器的低电平输出
        [P_IOB_Dir]=R1               //设置 3 个属性向量
        [P_IOB_Data]=R1
        R1=0x0000
        [P_IOB_ Attrib]=R1

        R1=0x0000                    //向寄存器写入 0x0000
        [P_IOB_Data]=R1
    //**********************此时检测 B 口的电平为高**********************//
        R3=[P_IOB_Buffer]            // 回读 Buffer 的值为 0x0000
    LOOP:
        JMP     LOOP                 //主程序循环
```

P_IOA_Data 与 P_IOA_Buffer 区别：

读取端口

从端口读取数据的两种方式：从 P_IOA_Data 读取数据相当于读取端口管脚的信号。然而，从 P_IOA_Buffer 读取数据相当于从数据寄存器中读取数据。

```
R1 = [P_IOA_Buffer]        //从 A 口数据寄存器中读取数据
R1 = [P_IOA_Data]          //从 A 口管脚读取数据
```

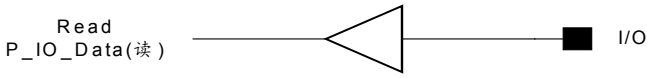
设置端口

这里有两种写入 IOA 的数据方法，向 P_IOA_Data 写入数据等同于向 P_IOA_Buffer 写入数据。

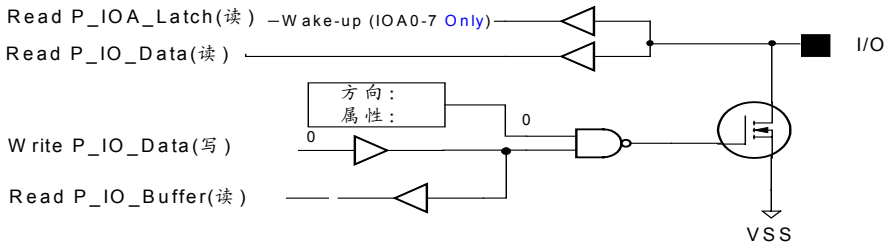
```
R1 = 0x0000;
[P_IOA_Data] = R1 ;              // 写数据到 P_IOA_Data
[P_IOA_Buffer] = R1;             // 写数据到 P_IOA_Buffer
```

详细如图 2.5。

PortA, PortB

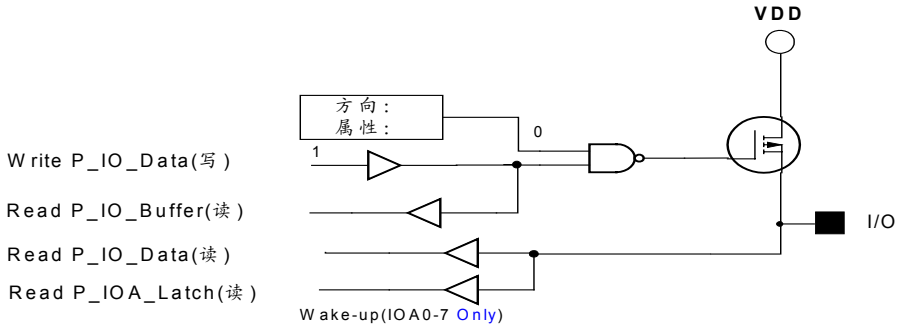


悬浮输入 (Pure)



下拉输入

SPCE061 Typical
233KOhms @ DC2.7V
180KOhms @ DC3.3V



上拉输入

SPCE061 Typical
325KOhms @ DC2.7V
242KOhms @ DC3.3V

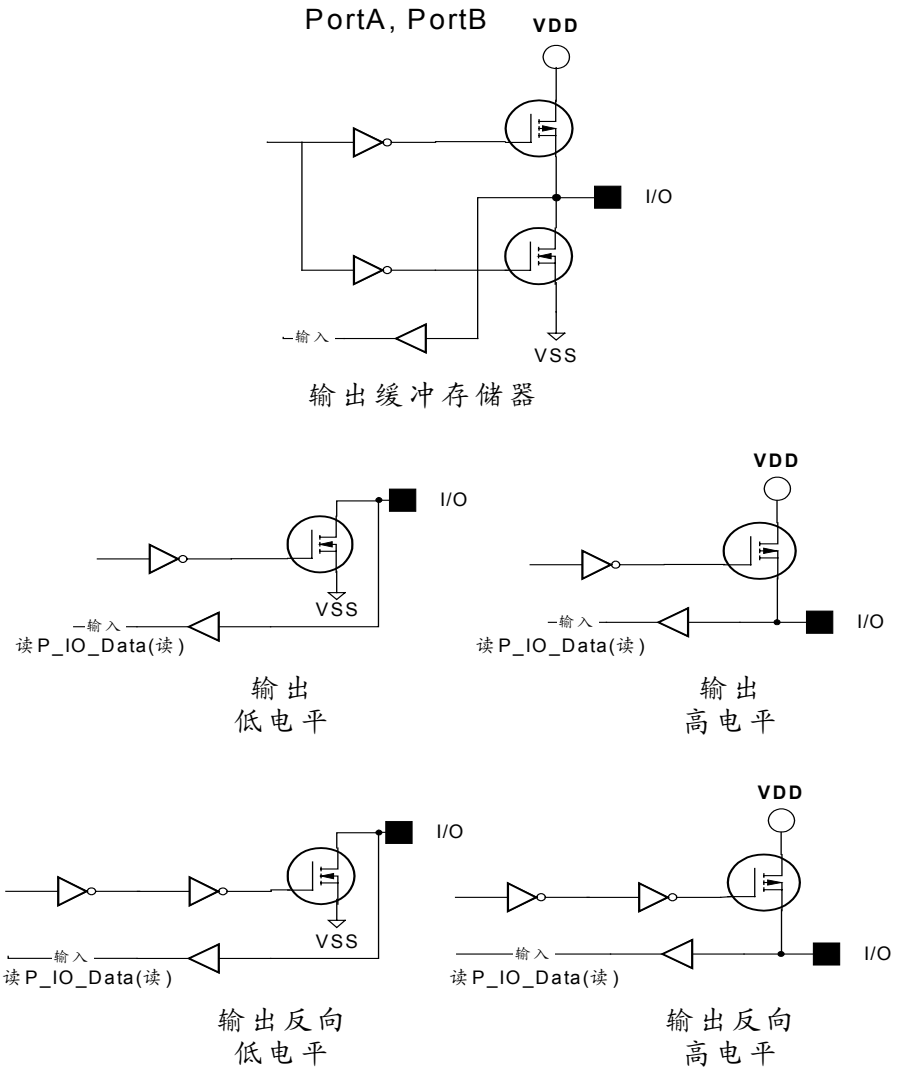


图 2.5 端口结构图

P_IOB_Data(读/写)(7005H)

B 口的数据单元，用于向 B 口写入或从 B 口读出数据。当 B 口处于输入状态时，读出是读 B 口管脚电平状态； 写入是将数据写入 B 口的数据寄存器。当 B 口处于输出状态时，写入输出数据到 B 口的数据寄存器。

P_IOB_Buffer(读/写)(7006H)

B 口的数据向量单元，用于向数据寄存器写入或从该寄存器内读出数据。当 B 口处于输入状态时，写入是将数据写入 B 口的数据寄存器；读出则是从 B 口数据寄存器里读其数值。当 B 口处于输出状态时，写入数据到 B 口的数据寄存器。

P_IOB_Dir(读/写)(7007H)

B 口的方向向量单元，用于设置 IOB 口的状态。‘0’为输入，‘1’为输出。

P_IOB_Attrib(读/写)(7008H)

B 口的属性向量单元，用于设置 IOB 口的属性。

[例 2.8]：

设置 A 口低 8 位为带下拉电阻的输入口，作为按键输入；B 口低 8 位为带数据缓存器的高电平输出口，外接发光二极管显示。当 Key1 按下时对应的 B0 口灯亮，依此类推**……**

程序如下：

```
//**********************A 口,B 口设置子程序*****************************//
      R1 = 0x0000                    // R1 的值为 0x0000
      [P_IOA_Dir] = R1               //设置 A 口的方向向量
      [P_IOA_Attrib] = R1            //设置 A 口的属性向量
      R1 = 0x00FF
      [P_IOA_Data] = R1              //设置 A 口的数据向量

      R2 = 0X00FF                    // R2 的值为 0x00FF
      [P_IOB_Dir] = R2               //设置 B 口的方向向量
      [P_IOB_Attrib] = R2            //设置 B 口的属性向量
      [P_IOB_Data] = R2              //设置 B 口的数据向量
      //************设已经取到键值，判断键值并输出显示的程序***************//
      ……
      R1 = [keycode]                 //将所取的键值送到 R1 中
      CMP R1,0x0000                  //若所取的键值为 0 则表明无键按下，返回主程序
      JE _MAIN
      CMP R1 ,0x0001                 //比较是否为第一键按下
      JE   LOOP1                     //是，跳转到 LOOP1
      CMP R1 ,0x0002                 //比较是否为第二键按下
      JE   LOOP2                     //是，跳转到 LOOP2
      ……
      CMP R1 ,0x0040                 //比较是否为第七键按下
      JE   LOOP7                     //是，跳转到 LOOP7
      R2=[P_IOA_Data]                //否则，是第八个键按下，IOB7 口输出高电平
      R2&=0x0080
      [P_IOB_Data]=R2                //IOB7 口输出高电平，点亮该口的指示灯
 JMP _MAIN                           //返回主程序
      ……
LOOP1：
      R2=[P_IOA_Data]                //是第一键按下，IOB0 口输出高电平
      R2&=0x0001                     //确保只有 IOB0 口输出高电平
```

```
        [P_IOB_Data]=R2
  JMP _MAIN
LOOP2：
        R2=[P_IOA_Data]              //是第二键按下，IOB1 口输出高电平
        R2&=0x0002                   //确保只有 IOB1 口输出高电平
        [P_IOB_Data]=R2
  JMP _MAIN
          ……
LOOP7：

        R2=[P_IOA_Data]              //是第七键按下，IOB6 口输出高电平
        R2&=0x0040                   //确保只有 IOB6 口输出高电平
        [P_IOB_Data]=R2
  JMP _MAIN
```

A 口低 8 位的属性设置及对应的口位状态如表 2.3所示：（不考虑高 8 位）

表2.3 A 口设置

地址	——	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
7002H	Dir	–	–	–	–	–	–	–	–	0	0	0	0	0	0	0	0
7003H	Attrib	–	–	–	–	–	–	–	–	0	0	0	0	0	0	0	0
7000H	Data	–	–	–	–	–	–	–	–	1	1	1	1	1	1	1	1
——	状态	----------								带上拉电阻的输入							

B 口低 8 位的属性设置及对应的口位状态如下表 2.4所示：（不考虑高 8 位）

表2.4 B 口设置

地址	——	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
7007H	Dir	–	–	–	–	–	–	–	–	1	1	1	1	1	1	1	1
7008H	Attrib	–	–	–	–	–	–	–	–	1	1	1	1	1	1	1	1
7005H	Data	–	–	–	–	–	–	–	–	1	1	1	1	1	1	1	1
——	状态	----------								带数据缓存器的高电平输出							

硬件原理图如下所示：

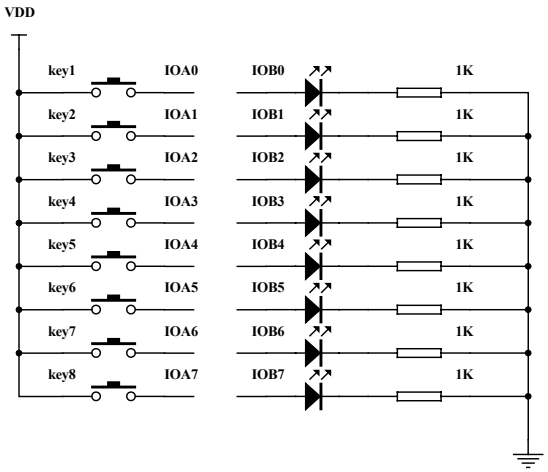


图 2.6　8 键键盘硬件原理

B 口的特殊功能

正如前面提到的，B 口除了具有常规的输入/输出端口功能外，还有一些特殊的功能，如下表 2.5所示：

表2.5 B 口的特殊功能

口位	特殊功能	功能描述	备注
IOB0	SCK	串行接口 SIO 的时钟信号	参见串行设备接口 SIO 的功能设置内容
IOB1	SDA	串行接口 SIO 的数据传送信号	参见串行设备接口 SIO 的功能设置内容
IOB2	EXT1	外部中断源(下降沿触发)	IOB2 设为输入状态
	Feedback_Output1	与 IOB4 组成一个 RC 反馈电路，以获得振荡信号，作为外部中断源 EXT1	设 置 IOB2 为反相输出方式，见图 2.7 所示的框图及P_FeedBack(写)(7009H)单元的描述
IOB3	EXT2	外部中断源(下降沿触发)	IOB3 设为输入状态
	Feedback_Output2	与 IOB5 组成一个 RC 反馈电路，以获得一个振荡信号，作为外部中断源 EXT2	设置 IOB3 为反相输出方式，见图 2.7 所示的框图及P_FeedBack(写)(7009H)单元的描述
IOB4	Feedback_Input1		见图 2.7 所示
IOB5	Feedback_Input2		见图 2.7 所示
IOB6	---		
IOB7	Rx	通用异步串行数据接收端口	参见通用异步串行接口部分
IOB8	APWMO	TimerA 脉宽调制输出	参见定时器/计数器部分，P_Feedback(写) (7009H) 单元部
IOB9	BPWMO	TimerB 脉宽调制输出	参见定时器/计数器部分
IOB10	Tx	通用异步串行数据发送端口	参见通用异步串行端口 UART部分

注：

1．口位默认为带下拉电阻的输入管脚

2．PWM：脉宽调制(Pulse Width Modulation)

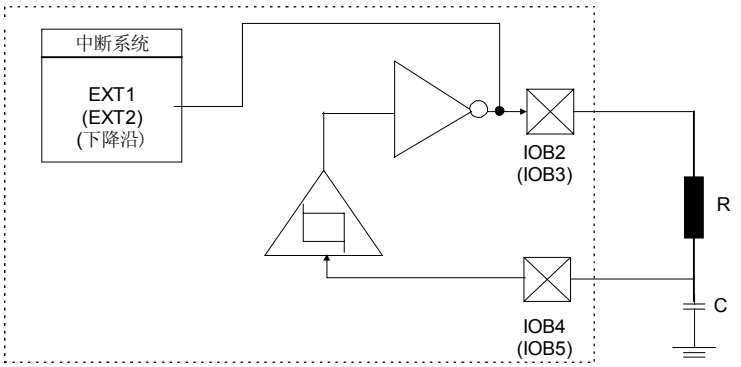
P_FeedBack(写)(7009H)

B 口工作方式的控制单元，用于控制 B 口的 IOB2 (IOB3)和 IOB4 (IOB5)用作普通 I/O 口，或作为特殊功能口。其特殊功能包括以下两个部分：（1）.单个 IOB2 或 IOB3 口可设置为外部中断的输入口。（2）.设置 P_FeedBack 单元， 再将 IOB2(IOB3)和 IOB4(IOB5)之间连接一个电阻和电容（电路连接如图 2.7）形成反馈电路以产生振荡信号，此信号可作为外部中断源输入 EXT1 或 EXT2。当然此时所得到的中断频率与 RC 振荡器的频率是一致的。由于该频率较高，所以通常情况下都是通过(1)获得外部中断信号。此特殊功能仅运用于：当外部电路需要用到一定频率的振荡信号时，可以在 IOB2(IOB3)端获得。P_FeedBack 的设置的如表 2.6所示。

表2.6 P_FeedBack 的设置

b15 – b4	b3	b2	b1	b0
---	FBKEN3	FBKEN2	---	---
	1：设定 IOB3 和 IOB5 之间形成反馈功能 0：IOB3、IOB5 作为普通的 I/O 口(默认)	1：设定 IOB2 和 IOB4 之间形成反馈功能 0：IOB2、IOB4 作为普通的 I/O 口(默认)		

图 2.7 为 IOB2，IOB3，IOB4 及 IOB5 的反馈结构示意图。通过在 IOB2 (IOB3)和 IOB4 (IOB5)之间增加一个 RC 电路形成反馈回路，即可在 IOB2(IOB3)端得到振荡源频率信号。为使反馈回路正常工作，必须将 IOB2 (IOB3) 设置成反相输出口，且将 IOB4 (IOB5)设置成悬浮式输入口。



当P_FeedBack(写)($7009H)单元的FBKEN2(FBKEN3)位被置为"1"时
IOB2、IOB4或IOB3、IOB5之间的反馈结构

图 2.7　反馈结构示意图

[例 2.9]：

以下程序说明如何编程使得通过在 IOB2 和 IOB4 增加一个外部 RC 电路形成反馈回路，以获得振荡源频率。

```
//将 IOB4 设置成悬浮式输入口，IOB2 设置成反相输出口
R1=0x0004;
```

```
[P_IOB_Dir]=R1；
R1=0x0010；
[P_IOB_Attrib]=R1；
[P_IOB_Data]=R1
// 写入 P_FeedBack 口，设定 IOB2，IOB4 为特殊功能口
R1=0x0004；
[P_FeedBack]=R1；
```

IOB8 和 IOB10 的控制向量由 TAON、TXPinEn 设置

IOB8 和 IOB10 的应用由控制向量 TAON 和 TXPinEn 来控制。

TAON	**TXPinEn**	IOB8	IOB10
0	0	普通 I/O 端口	普通 I/O 端口
0	1	普通 I/O 端口	Tx 端口
1	0	APWMO 端口	普通 I/O 端口
1	1	APWMO 端口	Tx 端口

注：

1. TAON：TimerA 脉宽调制输出 APWMO 的允通信号(详细内容请参见定时器/计数器部分)
2. APWMO：TimerA 脉宽调制的输出信号(详细内容请参见定时器/计数器部分)

[例 2.10]：图 2.8 为 APWMO 信号输出波形

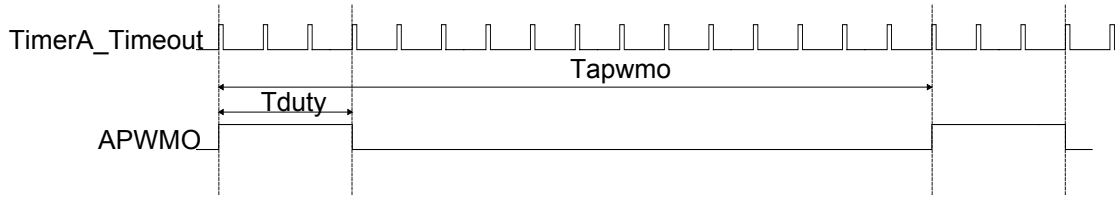


图 2.8　APWMO 信号时序图

程序段如下：

```
R1=0x0100；                    //将 IOB8 设置成同相输出端口
[P_IOB_Dir]=R1；
[P_IOB_Attrib] = R1；
R1=0x0000；
[P_IOB_Data]=R1；
//设置 TimerA 的 APWMO 信号的周期 Tapwmo=(12.288MHz / 512) / 16 = 1.5KHz，设
//置信号的占空比 APWMO
//Tduty= (3/16)*Tapwmo。详细内容请参见定时器/计数器部分
R1=0x00F0；                    //选择计数频率与占空比
[P_TimerA_Ctrl]=R1；

R1=0xFDFF；                    //设置 TimerA 的计数初值
[P_TimerA_Data]=R1；
```

2.4 时钟电路

μ'nSP™时钟电路采用晶体振荡器电路。图 2.9 为 SPCE061A 时钟电路的接线图。外接晶振采用 32768Hz。推荐使用外接 32768Hz 晶振，因阻容振荡的电路时钟不如外接晶振准确。

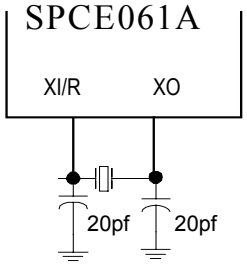


图 2.9　SPCE061A 与振荡器的连接

实时时钟 RTC(Real Time Clock)

32768Hz 实时时钟通常用于钟表、实时时钟延时以及其它与时间相关类产品。SPCE061A 通过对 32768Hz 实时时钟源分频而提供了多种实时时钟中断源。例如，用作唤醒源的中断源 IRQ5_2Hz，表示系统每隔 0.5 秒被唤醒一次，由此可作为精确的计时基准。

除此之外，SPCE061A 还支持 RTC 振荡器强振模式/自动模式的转换(参考**系统时钟**部分)。

2.5 锁相环 PLL (Phase Lock Loop)振荡器

PLL 电路的作用是将系统提供的实时时钟的基频(32768Hz)进行倍频，调整至 49.152MHz、40.96MHz、32.768MHz、24.576MHz 或 20.480MHz。系统默认的 PLL 自激振荡频率为 24.576MHz。PLL 的作用如图 2.10 所示：

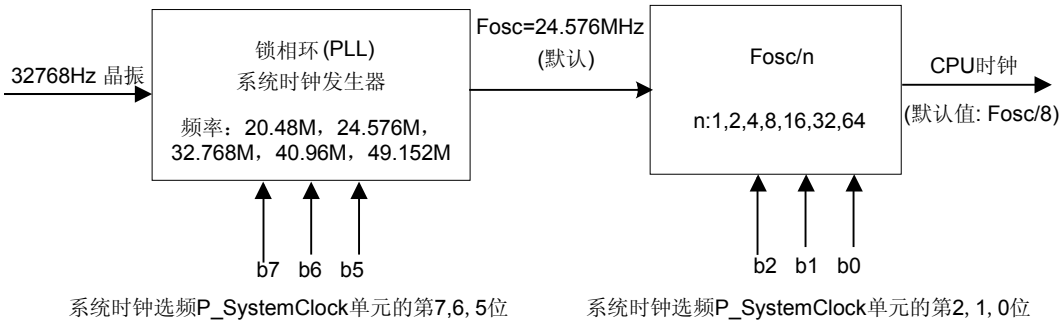


图 2.10　锁相环电路框图

2.6 系统时钟

32768 的实时时钟经过 PLL 倍频电路产生系统时钟频率(Fosc)，Fosc 再经过分频得到 CPU 时钟频率(CPUCLK)可通过对 P_SystemClock(写)(7013H)单元编程来控制。默认的 Fosc、CPUCLK 分别为 24.576MHz 和 Fosc/8。用户可以通过对 P_SystemClock 单元编程完成对系统时钟和 CPU 时钟频率的定义。

此外，32768Hz RTC 振荡器有两种工作方式：强振模式和自动弱振模式。处于强振模式时，RTC 振荡器始终运行在高耗能的状态下。处于自动弱振模式时，系统在上电复位后的前 7.5s 内处于强振模式，然后自动切换到弱振模式以降低功耗。CPU 被唤醒后默认的时钟频率为 Fosc/8，用户可以根据需要调整该值。CPU 被唤醒后经过 32 个时钟周期的缓冲时间后再进行其它的操作，这样可以避免在系统被唤醒后造成 ROM 读取错误。

在 SPCE061A 内，P_SystemClock(写)(7013H)单元（如表 2.7所示）控制着系统时钟和 CPU 时钟。第 0~2 位用来改变 CPUCLK，若将第 0~2 位置为“111”可以使 CPU 时钟停止工作，系统切换至低功耗的睡眠状态；通过设置该单元的第 5~7 位可以改变系统时钟的频率(如表 2.9所示)。此外，在睡眠状态下，通过设置该单元的第 4 位可以接通或关闭 32768Hz 实时时钟。

表2.7 设置 P_SystemClock 单元

b15-b8	b7~b5	b4[1]	b3	b2	b1	b0
---	PLL 频率选择	32768Hz 睡眠状态	32768Hz 方式选择	CPU 时钟选择		
		1：在睡眠状态下，32768Hz 时钟仍处于工作状态(默认) 0：在睡眠状态下，32768H 时钟被关闭	1：32768Hz 时钟处强振模式 0：32768Hz 时钟处自动弱振模式(默认)			

表2.8 CPU 时钟频率 (CPUCLK) 选择

b2	b1	b0	CPUCLK
0	0	0	Fosc

0	0	1	Fosc/2
0	1	0	Fosc/4
0	1	1	Fosc/8[2]
1	0	0	Fosc/16
1	0	1	Fosc/32
1	1	0	Fosc/64
1	1	1	停止(睡眠状态)

表2.9　PLL 频率 (Fosc) 选择

b7	b6	b5	Fosc
0	0	0	24.576MHz
0	0	1	20.48MHz
0	1	0	32.768MHz
0	1	1	40.96MHz
1	-	-	49.152MHz

注：

[1]只有当 b0~b2 同时被置为“1”时(即睡眠状态)b4 设置才有效。

[2]上电复位或系统从备用状态(睡眠状态)被唤醒后，默认的 CPU 时钟频率为 Fosc/8。

[例 2.11]：编写延时 1ms 的子程序

分析：从上面的内容我们可以看出 32768Hz 的实时时钟经过 PLL 倍频电路之后会产生多个 Fosc 供选择，在一个 Fosc 下又可选择不同的 CPUCLK。这就给我们编写延时程序提供了更大的选择空间，同时也要求我们在编写延时子程序之前必须选定相应的 Fosc 与 CPUCLK。

```
        ……
        R1=0x0023                    //Fosc 选择 20MHz，CPUCLK 选择 Fosc/8
       [P_System_Clock]=R1
       R3=0x0000                     //R3 的初值为 0
  DELAY:                             //延时子程序
       R2=0x0000
   LOOP1:
       R2+=1                         //内循环延时 100us
       CMP R2,21
       JB   LOOP1
       R3+=1                         //外循环计数加一
       CMP R3,10                     //当计数到 10 时，延时 1 秒结束
       JB   DELAY
       RETF
```

2.7 时间基准信号

时间基准信号，简称时基信号，来自于 32768Hz 实时时钟，通过频率选择组合而成。时基信号发生器的选频逻辑 TMB1 为 TimerA 的时钟源 B 提供各种频率选择信号并为中断系统提供中断源(IRQ6)信号。此外，时基信号发生器还可以通过分频产生 2Hz、4Hz、1024Hz、2048Hz 以及 4096Hz 的时基信号，为中断系统提供各种实时中断源(IRQ4 和 IRQ5)信号。时基信号发生器的结构如图 2.11 所示。

P_Timebase_Setup(写)(700EH)

时基信号发生器通过对 P_Timebase_Setup(写)(700EH)单元（如表 2.10所示）的编程写入来进行选频操作。

表2.10 P_Timebase_Setup 单元

b15- b4	B3	b2	b1	b0
---	TMB2 选频逻辑		TMB1 选频逻辑	

表2.11 选频逻辑

b3	b2	TMB2	b1	b0	TMB1
0	0	128Hz*	0	0	8Hz**
0	1	256Hz	0	1	16Hz
1	0	512Hz	1	0	32Hz
1	1	1024Hz	1	1	64Hz
*：默认的 TMB2 输出频率为 128Hz			**：默认的 TMB1 输出频率为 8Hz		

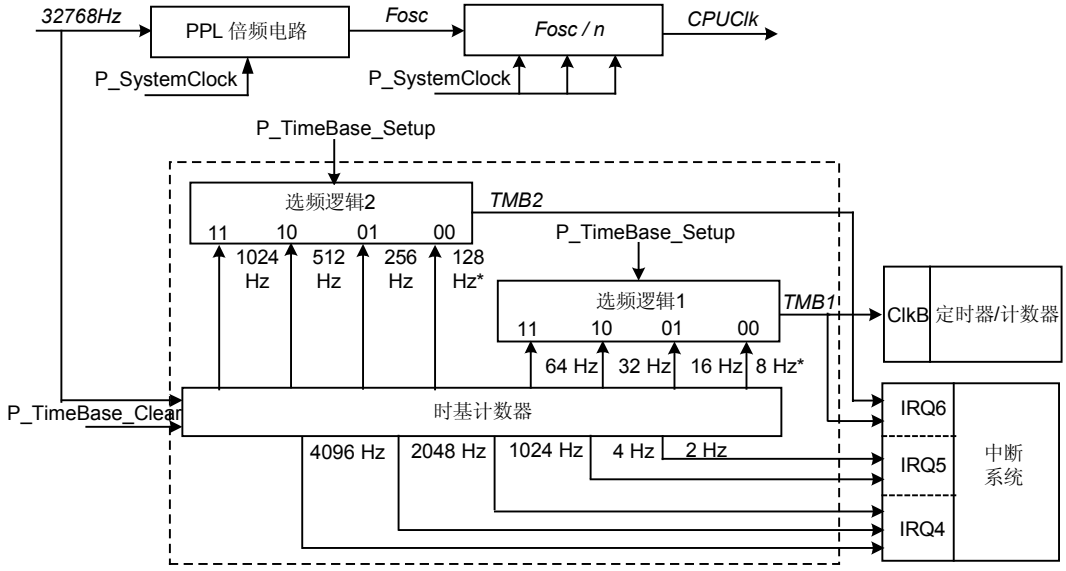


图 2.11　时基信号发生器的结构

[例 2.12]：选择 8Hz 的时基频率，进入中断后 IOB 口输出高电平。通过示波器可观察输出为高电平。

```
.INCLUDE hardware.inc          //包括头文件
.CODE
.PUBLIC _main                  //主函数
_main:
    INT OFF                    //关中断
    R1=0xFFFF                  //设置 B 口为带数据缓存器的高电平输出
    [P_IOB_Dir]=R1             //设置方向向量
    [P_IOB_Attrib]=R1          //设置属性向量
    [P_IOB_Data]=R1            //设置数据向量
    R1=0x0000                  //选择中断频率为 8Hz
    [P_TimeBase_Setup]=R1
    R1=0x0002                  //设置时基中断
    [P_INT_Ctrl]=R1
    INT IRQ                    //开中断
WAIT:                          //主程序循环
    JMP WAIT
.TEXT                          //定义中断子程序段
.PUBLIC _IRQ6
_IRQ6:
    R1=0x0001                  //判断是否为 IRQ_TMB2 中断
    TEST R1, [P_INT_Ctrl]
    JNZ IRQ_TMB2               //是，进入该中断；否，进入 IRQ_TMB1
IRQ_TMB1:
    PUSH R1,R4 TO [SP]         //将寄存器压栈
    R1=0x0002                  //清中断标识
    [P_INT_Clear]=R1
    R2=0xFFFF                  //B 口输出高电平
    [P_IOB_Data]=R2
    POP R1,R4 FROM [SP]        //出栈
    RETI                       //子程序返回
IRQ_TMB2:                      //中断子程序 IRQ_TMB2
    PUSH R1,R4 TO [SP]         //堆栈
    R1=0x0001                  //清中断标识
    [P_INT_Clear]=R1
    POP R1,R4 FROM [SP]        //出栈
    RETI                       //返回
//*******上述的例子和中断有关的部分参照第 5 章中断内容***************
```

P_Timebase_Clear(写)(700FH)

P_Timebase_Clear (写)(700FH)单元是控制端口，设置该单元可以完成时基计数器复位和时间校准。向该单元写入任意数值后，时基计数器将被置为“0”，以此可对时基信号发生器进行精确的时间校准。

2.8 定时器/计数器

SPCE061A 提供了两个 16 位的定时/计数器：TimerA 和 TimerB。TimerA 为通用计数器；TimerB 为多功能计数器。TimerA 的时钟源由时钟源 A 和时钟源 B 进行“与”操作而形成；TimerB 的时钟源仅为时钟源 A。TimerA 的结构如图 2.12 所示，TimerB 的结构如图 2.13 所示。

定时器发生溢出后会产生一个溢出信号(TAOUT/TBOUT)。一方面，它会作为定时器中断信号传输给 CPU 中断系统；另一方面，它又会作为 4 位计数器计数的时钟源信号，输出一个具有 4 位可调的脉宽调制占空比输出信号 APWMO 或 BPWMO(分别从 IOB8 和 IOB9 输出)，可用来控制马达或其它一些设备的速度。此外，定时器溢出信号还可以用于触发 ADC 输入的自动转换过程和 DAC 输出的数据锁存。

向定时器的P_TimerA_Data(读/写)(700AH)单元或P_TimerB_Data(读/写)(700CH)单元写入一个计数值 N 后，选择一个合适的时钟源，定时器/计数器将在所选的时钟频率下开始以递增方式计数 N，N+1，N+2，…0xFFFE，0xFFFF。当计数达到 0xFFFF 后，定时器/计数器溢出，产生中断请求信号，被 CPU 响应后送入中断控制器进行处理。同时，N 值将被重新载入定时器/计数器并重新开始计数。

图 2.8 是一个占空比为 3/16 的脉宽调制输出信号的时序。通过写入P_TimerA_Ctrl(700BH)单元的第 6~9 位，可选择设置 APWMO 输出波形的脉宽占空比；同理，写入 P_TimerB_Ctrl(700DH)单元的第 6~9 位，便可选择设置 BPWMO 输出波形的脉宽占空比。

时钟源 A 是高频时钟源，来自带锁相环的晶体振荡器输出 Fosc；时钟源 B 的频率来自 32768Hz 实时时钟系统，也就是说，时钟源 B 可以作为精确的计时器。例如，2Hz 定时器可以作为实时时钟的时钟源。

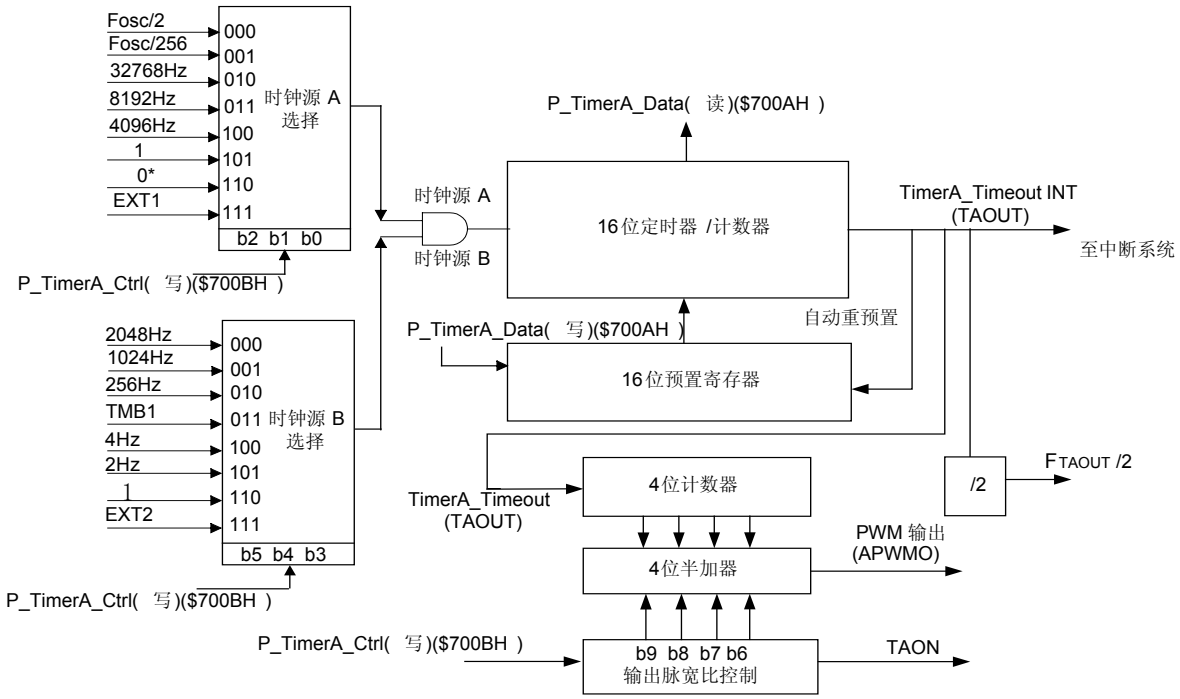


图 2.12　TimerA 结构

从上面的结构我们可以看出时钟源 A 是一个高频时钟源，时钟源 B 是一个低频时钟源。时钟源 A 和时钟源 B 的组合，为 TimerA 提供了多种计数速度。若以 ClkA 作为门控信号，‘1’表示允许时钟源 B 信号通过，而‘0’则表示禁止时钟源 B 信号通过而停止 TimerA 的计数。？例如，如果时钟源 A 为“1”，TimerA 时钟频率将取决于时钟源 B；如果时钟源 A 为“0”，将停止 TimerA 的计数。EXT1 和 EXT2 为外部时钟源。

P_TimerA_Data(读/写)(700AH)

TimerA 的数据单元，用于向 16 位预置寄存器写入数据(预置计数初值)或从其中读取数据。在写入数值以后，计数器便会在所选择的频率下进行加一计数，直至计数到 0xFFFF 产生溢出。.溢出后 P_TimerA_Data 中的值将会被重置，再以置入的值继续加一计数。读到这儿你会发现计数初值对于计数器/定时器的应用非常重要，那么怎样计算计数初值呢？一般说来分为以下几步：1.选择需要的计数频率。2. 计算相应的计数初值。下面我们以 TimerA 选择 2048Hz，fosc/2 作为计数频率进行讲解。

[例 2.13]：//**********TimerA 计数频率选择 2048Hz**********//

R1=0x0005

[P_TimerA_ctrl]=R1　　//选择 2048Hz

分析：要完成 1 秒的定时，计数次数应该为 2048 次，若选择每 0.5 秒产生一次计数溢出，则需要计数 1024 次，1024 转化为 16 进制数为 400，0xFFFF-0x0400=0xFBFF，所以 P_TimerA_Data 设置如下：

[P_TimerA_Data]=0xFBFF

……

[例 2.14]：//********************TimerA 计数频率选择 fosc/2**********************//

默认系统时钟为 24.576MHz，下面计算初值假设计数初值为 X，则计数次数为 FFFF-X，在将计数次数转化为 10 进制数代入下面的公式：

$24.576\times10^6/2/$ (转化后的 10 进制数值)= 计数溢出频率

计数溢出频率为 8KHz 的计数次数 Y 如下：

$24.576\times10^6/2/Y=8000$

Y=1536=(600)H

所以计数初值[P_TimerA_Data]=0xF9FF

TimerB 初值的计算方法和 TimerA 一样，只是频率的选择范围不同。

P_TimerA_Ctrl(写)(700BH)

TimerA 的控制单元如表 2.12所示。用户可以通过设置该单元的第 0~5 位来选择 TimerA 的时钟源(时钟源 A、B)。设置该单元的第 6~9 位（如表 2.13 所示），TimerA 将输出不同频率的脉宽调制信号，即对脉宽占空比输出 APWMO 进行控制。

表2.12 P_TimerA_Ctrl 单元

b15 – b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
---	占空比的设置（表 2.13）				时钟源 B 选择位(表 2.15)			时钟源 A 选择位(表 2.14)		

表2.13 设置 B6-B9 位

b9	b8	b7	b6	脉宽占空比(APWMO)	TAON[1]
0	0	0	0	关断	0
0	0	0	1	1/16	1
0	0	1	0	2/16	1
0	0	1	1	3/16	1
0	1	0	0	4/16	1
0	1	0	1	5/16	1
0	1	1	0	6/16	1
0	1	1	1	7/16	1
1	0	0	0	8/16	1

续

b9	b8	b7	b6	脉宽占空比(APWMO)	TAON[1]
1	0	0	1	9/16	1
1	0	1	0	10/16	1
1	0	1	1	11/16	1
1	1	0	0	12/16	1
1	1	0	1	13/16	1
1	1	1	0	14/16	1
1	1	1	1	TAOUT[2]触发信号	1

注：

[1]：TAON 是 TimerA(APWMO)的脉宽调制信号输出允许位，默认值为“0”，当 TimerA 的第 6~9 位不全为零时 TAON=1；

[2]：TAOUT 是 TimerA 的溢出信号，当 TimerA 的计数从 N 达到 0xFFFF 后(用户通过设置 P_TimerA_Data (写)(700AH)单元指定 N 值)，发生计数溢出。产生的溢出信号可以作为 TimerA 的中断信号被送至中断控制系统；同时 N 值将被重新载入预置寄存器，使 Timer 重新开始计数。TAOUT 触发信号(TAOUT/2)的占空比为 50%，频率为 $F_{TAOUT}/2$，其它输入信号的频率为 $F_{TAOUT}/16$。请参考 TimerA 结构图。

表2.14 设置 b0－b2 位

b2	b1	b0	时钟源 A 的频率
0	0	0	Fosc/2
0	0	1	Fosc/256
0	1	0	32768Hz
0	1	1	8192Hz
1	0	0	4096Hz
1	0	1	1
1	1	0	0*
1	1	1	EXT1

表2.15 设置 b3－b5 位

b5	b4	b3	时钟源 B 的频率
0	0	0	2048Hz
0	0	1	1024Hz
0	1	0	256Hz
0	1	1	TMB1
1	0	0	4Hz
1	0	1	2Hz
1	1	0	1*
1	1	1	EXT2

注：

*代表默认值为 1。若以 ClkA 作为门控信号，‘1’表示允许时钟源 B 信号通过，而‘0’则表示禁止时钟源 B 信号通过而停止 TimerA 的计数。如果时钟源 A 为‘1’，TimerA 时钟频率将取决于时钟源 B；如果时钟源 A 为‘0’，将停止 TimerA 的计数。

P_TimerB_Data(读/写)(700CH)

TimerB 的数据单元，用于向 16 位预置寄存器写入数据(预置计数初值)或从其中读取数据。写入数据后，计数器就会以设定的数值往上累加直至溢出。计数初值的计算方法和 TimerA 相同。

P_TimerB_Ctrl(写)(700DH)

TimerB 的控制单元（如表 2.16所示）。用户可以通过设置该单元的第 0~2 位来选择 TimerB 的时钟源。设置第 6~9 位，TimerB 将输出不同频率的脉宽调制信号，即对脉宽占空比输出 BPWMO 进行控制。

表2.16 设置 P_TimerB_Ctrl 单元

b15 - b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
---	Output_pulse_ctrl				---			时钟源 A 选择位		

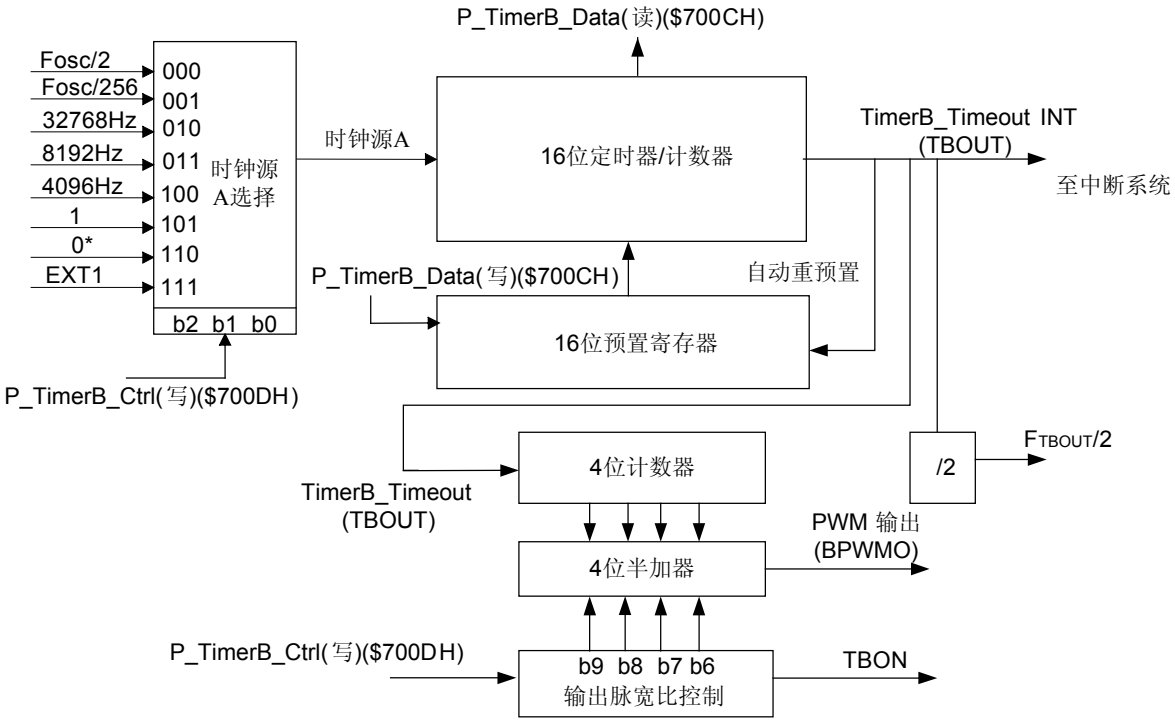
b9	b8	b7	b6	脉宽占空比(BPWMO)	TBON[1]
0	0	0	0	关断	0
0	0	0	1	1/16	1
0	0	1	0	2/16	1
0	0	1	1	3/16	1
0	1	0	0	4/16	1
0	1	0	1	5/16	1
0	1	1	0	6/16	1
0	1	1	1	7/16	1
1	0	0	0	8/16	1
1	0	0	1	9/16	1
1	0	1	0	10/16	1
1	0	1	1	11/16	1
1	1	0	0	12/16	1
1	1	0	1	13/16	1
1	1	1	0	14/16	1
1	1	1	1	TBOUT[2] 触发信号	1

注：

[1]：TBON 是 TimerB(BPWMO)的脉宽调制信号输出允许位，默认值为‘0’；

[2]：TBOUT 是 TimerB 的溢出信号，当 TimerB 的计数从 N 达到 0xFFFF 后(用户通过设置 P_TimerB_Data (写)(700CH)单元指定 N 值)，发生计数溢出。产生的溢出信号可以作为 TimerB 的中

断信号被送至中断控制系统；同时 N 值将被重新载入预置寄存器，使 Timer 重新开始计数。TBOUT 触发信号(TBOUT/2)的占空比为 50%，频率为 $F_{TBOUT}/2$，其它输入信号的频率为 FTBOUT/16。



TimerB 结构

[例 2.15]：

//通过 TimerA 计数，当计数达到 2 秒时从 IOB 口相应的输出高或低电平，最终输
//出连续方波

```
.DEFINE         P_IOB_Data      0x7005
.DEFINE         P_IOB_Dir       0x7007
.DEFINE         P_IOB_Attrib    0x7008
.DEFINE         P_INT_Ctrl      0x7010
.DEFINE         P_INT_Clear     0x7011
.DEFINE         P_TimerA_Ctrl   0x700b
.DEFINE         P_TimerA_Data    0x700A
.CODE
.PUBLIC _main
_main:
R1=0xFFFF                       //设置 B 口为带数据缓存器的低电平输出
[P_IOB_Dir]=R1                  //设置 3 个属性向量
```

```
[P_IOB_Attrib]=R1
R1=0x0000
[P_IOB_Data]=R1

R1=0x1000
[P_INT_Ctrl]=R1
INT OFF

R1=0x000D
[P_TimerA_Ctrl]=R1          //TimerA 时时钟频率选择为 1024Hz

R1=0xF7FF                   //TimerA 计数初值为 0xF7FF
[P_TimerA_Data]=R1          //计数满 2 秒时产生溢出
R2=0x0000                   //R2 的初值位 0
loop:                           //主程序循环
R1=0x1000
TEST R1,[P_INT_Ctrl]        //检查标识位，是否溢出
JZ   loop                   //没溢出，继续计数
  [P_INT_Clear]=R1          //溢出，清标识位
R2 ^= 0xFFFF                //输出方波
[P_IOB_Data]=R2
JMP loop                    //返回
```

2.9 睡眠与唤醒

2.9.1 睡眠

IC 在上电复位开始工作，直到接收到睡眠信号后，才关闭系统时钟(PLL 振荡器)，进入睡眠状态。用户可以通过对 P_SystemClock(读)(7013H)单元写入 *CPUClk STOP* 控制字(CPU 睡眠信号)使系统从运行状态转入备用状态。系统进入睡眠状态后，程序计数器(PC)会停在程序的下一条指令计数上，当有任一唤醒事件发生后开始由此继续执行程序。

2.9.2 唤醒

系统接收到唤醒信号后接通 PLL 振荡器，同时 CPU 会响应唤醒事件的处理并进行初始化。IRQ3_KEY 为触键唤醒源(IOA7~0)，其它中断信号 (FIQ、IRQ1~IRQ6 及 UART IRQ)

都可以作为唤醒源。唤醒操作完成后，程序将会从进入睡眠后指令计数的断点处开始被继续执行。关于触键唤醒源，请参考**I/O 端口结构** 。

[例 2.16]：说明如何编程实现系统从工作状态进入睡眠状态后，由触键引起唤醒。在进入睡眠状态之前，首先要将 IOA[7~0]设置为输入状态且允许 RQ3_KEY 中断来实现触键唤醒。

```
//IOA[7~0]设置
R1=0x0000；                    //设置 A 口为带下拉电阻的输入口
[P_IOA_Attrib]=R1；            //设置 3 个属性向量
[P_IOA_Dir]=R1；
[P_IOA_Data]=R1；

//中断设置(允许 IRQ3_KEY 触键中断，关于其它的中断设置，请参考第五章中断部分)
INT OFF；                      //关中断
R1=0x0080；                    //设置中断标置
[P_Int_Ctrl]=R1；
INT IRQ；                      //开中断

//读 P_IOA_Latch 单元，以锁存 IOA[0～7]的数据，用于触键唤醒
R1=[P_IOA_Latch]；             //锁存 A 口低 8 位的数据

//将 P_SystemClock(写)7013H 单元的第 0~2 位置为“111”，使系统进入睡眠状态，
//参见系统时钟部分
R1=0x0007；                    //系统进入睡眠状态
[P_SystemClock]=R1；
//IRQ3 子程序(端口 A 的触键唤醒源被触发后，调用 IRQ3 中断服务子程序)：
.TEXT
.PUBLIC _IRQ3
 _IRQ3:
    R1 = 0x0100；              //比较是否为 L_IRQ3_Ext1 中断
    TEST R1,[P_INT_Ctrl]；
    JNZ L_IRQ3_Ext1；          //是，则进入；否，进行下面的判断
    R1 = 0x0200；
    TEST R1,[P_INT_Ctrl]；     //是否为 L_IRQ3_Ext2 中断
    JNZ L_IRQ3_Ext2；          //是，进入该中断；否，执行下面的程序
L_IRQ3_KeyChange_WakeUp:       //不是上面的两种中断则一定为键唤醒中断
    R1 = 0x0080；              //清除 IRQ3 触键中断请求
    [P_INT_Clear]= R1；
      :
    (处理系统被唤醒后的任务)
      :
    RETI

L_IRQ3_Ext2:
```

```
    [P_INT_Clear] = R1;              //清除 IRQ3_EXT2 中断请求
    RETI

L_IRQ3_Ext1:
    [P_INT_Clear] = R1;              //清除 IRQ3_EXT1 中断请求
    RETI
```

2.10 模-数转换器 ADC

SPCE061A 有 8 路可复用 10 位 ADC 通道，其中一路通道(MIC_In)用于语音输入，模拟信号经过自动增益控制器和放大器放大后进行 A/D 转换。其余 7 路通道(Line_In)和 IOA[0~6] 管脚复用，可以直接通过引线(IOA[0~6])输入，用于将输入的模拟信号 (如电压信号) 转换为数字信号。SPCE061A 的 A/D 转换范围是整个输入范围，即，最大的模拟信号输入电压范围：0V～AV_{dd}。非法的 A/D 模拟信号(超过 VDD+0.3 V/低于 VSS–0.3 V)将影响转换电路的工作范围，从而降低 ADC 的性能。由于 Line_In 通道和 IOA[0~6]共用管脚，建议用户选择其他的 IO 管脚（非 IOA[0~6]），以避免由于非法 IO 信号造成电压不稳(超过 VDDIO+0.7V/低于 VSSIO-0.7V)而降低 ADC 的性能。

ADC 的最大输入电压由 P_ADC_Ctrl(写)($7015H)的第 7 位和第 8 位的值决定。第 7 位 VEXTREF 控制着 ADC 的参考电压为 AV_{dd}/外部参考电压。第 8 位 V2VREFB 控制着 2V 电压源是否起作用。如果起作用，用户可向 VEXTREF 管脚输入 2V 电压。此反馈回路把 ADC 的最高参考电压设置为 2V。如果用户指定的参考电压源的值不超过 AV_{dd}，它还可以被当作 ADC 的最高参考电压。

在 ADC 内，由 DAC0 和逐次逼近寄存器 SAR(Successive Approximation Register)组成逐次逼近式模-数转换器。向 P_ADC_Ctrl(写)($7015H)单元第 0 位(ADE)写入“1”，可以激活 ADC。系统默认的设置为屏蔽 ADC(ADE=0)。当 ADE=1 时，应对 P_ADC_Ctrl(写)($7015H)和 P_ADC_MUX_Ctrl(写)($702BH)的其他控制位进行合理的设置。

通过设置 P_ADC_MUX_Ctrl(写)($702BH)的第 0~2 位，可以为 A/D 转换选择输入通道。通道包括 MIC_In 和 Line_In 两种。运行时，如果 MIC_In 通道和 Line_In 通道都处于直接工作状态，程序会检查 P_ADC_Ctrl(W)($7015H)的第 15 位。只有当前的模数转换完成后，才能切换通道。当 MIC_In 通道处于定时器锁存状态，它可以优先访问 ADC。然后，用户可以从 P_ADC_MUX_Ctrl (读) ($702BH)的 FailB 位可查看到 Line_In ADC 被 MIC_In 通道的 ADC 打断。

用户可通过读取 P_ADC(读)($7014H)单元的内容，取得从 MIC_In 通道输入的模拟信号的转换结果。P_ADC_LINEIN_Data(读)($702CH)单元向用户提供了从指定的 Line_In 通道输入的模拟信号的转换结果。

选择 MIC_In 通道后，可通过设置 P_DAC_Ctrl(写)($702AH)的第 3 和 4 位，选择 A/D 转换的触发事件。当 P_ADC(读)($7014H)单元的数据被读取/TimerA/TimerB 事件发生后，可执行 A/D 转换。然而，在选择 Line_In 通道后，只有在读取 P_ADC_LINEIN_Data(读)($702CH)单元的内容后，才执行 A/D 转换，且不能使用定时器锁存数据。

进入睡眠状态后，ADC 被屏蔽(包括 AGC 和 V_{MIC})。注意，上电复位后不论 ADC 是否被激活，V_{MIC} 信号都默认为 ON。V_{MIC} 用于向外部的 MIC 提供电源，V_{MIC} =AV_{DD}。即，V_{MIC} 的状态和 ADC 的状态无关。所以，不使用 V_{MIC} 时，用户必须把 P_ADC_Ctrl(写)($7015H)单元的第 1 位 MIC_ENB 置为‘1’，以屏蔽 V_{MIC}。

硬件 ADC 的最高速率限定为(Fosc/32/16)Hz，如果速率超过此值，当从 P_ADC(读)($7014H)/ P_ADC_LINEIN_Data(读)($702CH)单元读出数据时会发生错误。

在 P_ADC_Ctrl(写)($7015H)单元的第 5 位 DAC_OUT，用户可设置两个通道的音频 DAC 的最大输出电平为 2mA/3mA(默认)。DAC_OUT 的设置可改变 DAC 输出的功率。

ADC 的最大响应率：(Fosc/32/16)

系统时钟	20.48MHz	24.576MHz	32.768MHz	40.96MHz	49.152MHz
响应率	40KHz	48KHz	64KHz	80KHz	96KHz

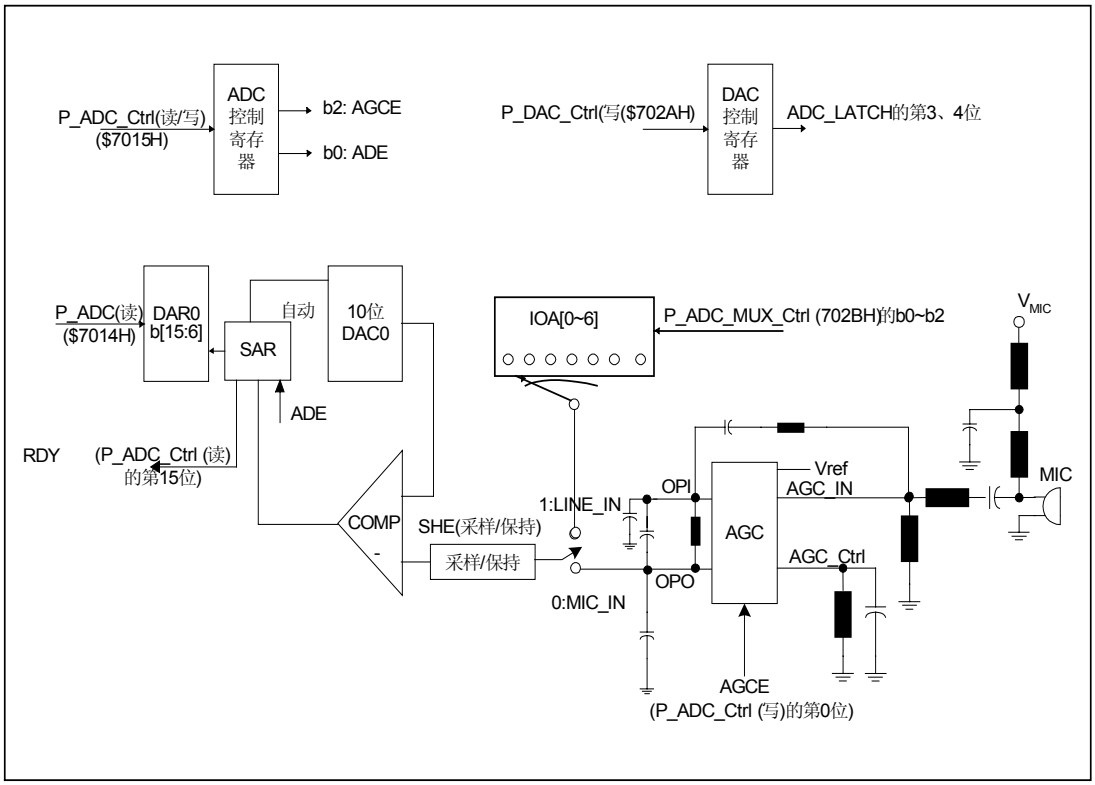


图 2.14　ADC 输入接口的结构

在 ADC 自动方式被启用后，会产生出一个启动信号，即 RDY=0。此时，DAC0 的电压模拟量输出值与外部的电压模拟量输入值进行比较，以尽快找出外部电压模拟量的数字量输出值。逐次逼近式控制首先将 SAR 中数据的最高有效位试设为‘1’，而其它位则全设为‘0’，即 10 0000 0000B。这时，DAC0 输出电压 V_{DAC0}(1/2 满量程)就会与输入电压 V_{in} 进行比较。如果 Vin>V_{DAC0}，则保持原先设置为‘1’的位(最高有效位)仍为‘1’；否则，该位会被清‘0’。接着，逐次逼近式控制又将下一位试设为‘1’，其余低位依旧设为‘0’，即 110000 0000B，V_{DAC0} 与 V_{in} 进行比较的结果若 Vin>V_{DAC0}，则仍保持原先设置位的值，否则便清‘0’该位。这个逐次逼近的过程一直会延续到 10 位中的所有位都被测试之后，A/D 转换的结果保存在 SAR 内。

当 10 位 A/D 转换完成时，RDY 会被置‘1’。此时，用户通过读取 P_ADC (7014H)或 P_ADC_MUX_Data(702CH)单元可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自动清‘0’来重新开始进行 A/D 转换。若未读取 P_ADC (7014H) 或 P_ADC_MUX_Data(702CH)单元中的数据，RDY 仍保持为‘1’，则不会启动下一次的 A/D 转换。外部信号由 LIN_IN[1~7]即 IOA[0~6]或通道 MIC_IN 输入。从 LIN_IN[1~7]输入的模拟信号直接被送入缓冲器 P_ADC_MUX_Data(702CH)；从 MIC_IN 输入的模拟信号则要经过缓冲器和放大器。AGC 功能将通过 MIC_IN 通道输入的模拟信号的放大值控制在一定范围内，然后放大信号经采样-保持模块被送至比较器参与 A/D 转换值的确定，最后送入 P_ADC (7014H)。

P_ADC(读/写)(7014H)

P_ADC 单元（如表 2.17所示）储存 MIC 输入的 A/D 转换的数据。逐次逼近式的 ADC 由一个 10 位 DAC(DAC0) 、一个 10 位缓存器 DAR0、一个逐次逼近寄存器 SAR 和一个比较器 COMP 组成。

表2.17 P_ADC 单元

b15 – b6	b5 – b0
DAR0(读/写)	---

P_ADC(读)：读出本单元实际为 A/D 转换输出的 10 位数字量。而且，如果 P_DAC_Ctrl (702AH)单元第 3、4 位被设为‘00’，那么在转换过程里读出本单元(7014H)亦会触发 A/D 转换重新开始。

P_ADC_Ctrl(读/写)(7015H)

P_ADC_Ctrl 单元（如表 2.18所示）为 ADC 的控制端口。

表2.18 P_ADC_Ctrl 单元

b15[2]	b8	b7	b6	B2	B1	b0	控制功能描述
RDY (读)	V2VREFB (写)	VEXTREF (写)	DAC_I (写)	AGCE (写)	MIC_ ENB (写)	ADE (写)	
0	-	-	-	-	-		10 位模/数转换未完成

1	-	-	-	-	-		10 位模/数转换完成，输出 10 位数字量
-	0	-	-	-	-		打开 2V 电压输出，其可作外部 AD 参考电压输入
-	1	-	-	-	-		关闭 2V 电压输出（默认）
-	-	0	-	-	-		不使用外部参考电压，AD 参考电压为 Vdd（默认）
-	-	1	-	-	-		外部参考电压管脚使能，从 VEXTREF 脚输入外部参考电压
-			0	-	-		DAC 电流= 3mA @V_{DD}=3V[1]
-			1	-	-		DAC 电流= 2mA @V_{DD}=3V
-	-	-	-	0	-		取消自动增益控制功能[3]
-	-	-	-	1	-		设置自动增益控制功能
					0		MIC 模式被使能，Vmic = AVdd
					1		MIC 模式被屏蔽
-	-	-	-	-		0	禁止模/数转换工作
-	-	-	-	-		1	允许模/数转换工作

注：

[1]　此为 DAC_I 的缺省选择。

[2]　b15 只用于 MIC_IN 通道输入。

[3]　当模拟信号通过麦克风的 MIC_IN 通道输入时，可选择 AGCE 为‘1’，即运算放大器的增益可在其线性区域内自动调整。AGCE 缺省选择为‘0’，即取消自动增益控制功能。

[4]　写入时需注意 b5 = 1, b4=1, b3 =1。

P_ADC_MUX_Ctrl (读/写)(702BH)

ADC 多通道控制是通过对 P_ADC_MUX_Ctrl (702BH)单元（如表 2.19所示）编程实现的。

表2.19 P_ADC_MUX_Ctrl 单元

b15	b14	b13-b3	b2	b1	b0	控制功能描述
Ready_MUX (读)[1]	FAIL (读) [2]	-	Channel_sel (读/写)			
0		-	-	-	-	10 位模/数转换未完成
1	0	-	-	-	-	10 位模/数转换完成
-		-	0	0	0	模拟电压信号通过 MIC_IN 输入
-		-	0	0	1	模拟电压信号通过 LINE_IN1 输入
-		-	0	1	0	模拟电压信号通过 LINE_IN2 输入
-		-	0	1	1	模拟电压信号通过 LINE_IN3 输入
-		-	1	0	0	模拟电压信号通过 LINE_IN4 输入
-		-	1	0	1	模拟电压信号通过 LINE_IN5 输入
-		-	1	1	0	模拟电压信号通过 LINE_IN6 输入
-		-	1	1	1	模拟电压信号通过 LINE_IN7 输入

注：

[1]. Ready_MUX 只用于 Line_in[7:1].

[2]. 一般情况下，该位总为‘0’。以下情况除外：由于 MIC_IN 的优先级高于 AD LINE_IN，所以在 LIN_IN AD 转换过程里又有 MIC_IN 时，若 AD 切换到 MIC 输入，原 LINE_IN 的数据会出现问题，此时 FAIL 被置为‘1’。 MIC AD 完成之后，该位被清为‘0’。

ADC 的多路 LINE_IN 输入将与 IOA[0~6]共用，即：

IOA6	IOA5	IOA4	IOA3	IOA2	IOA1	IOA0
LIN_IN 7	LIN_IN 6	LIN_IN 5	LIN_IN 4	LIN_IN 3	LIN_IN 2	LIN_IN 1

P_ADC_MUX_Data(读) (702CH)

P_ADC_MUX_Data 单元用于读出 LINE_IN[7:1]10 位 ADC 转换的数字数据，即：

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6
D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

[例 2.17]：

```
//*************************************************************
//Note:通过模拟量输入口 LINE_IN 输入电压值，通过读取 P_ADC_MUX_Data
//单元可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自
//动清'0'来重新开始进行 A/D 转换。若未读取 P_ADC_MUX_Data 单元中的数据
//RDY 仍保持为'1'，则不会启动下一次的 A/D 转换。
//*************************************************************

.DEFINE        P_IOB_DATA        0x7005
.DEFINE        P_IOB_DIR         0x7007
.DEFINE        P_IOB_ATTRI       0x7008
.DEFINE        P_INT_Ctrl        0x7010
.DEFINE        P_INT_CLEAR       0x7011

.DEFINE        P_ADC_Ctrl        0x7015
.DEFINE        P_ADC_MUX_Ctrl    0x702b
.DEFINE        P_ADC_MUX_DATA    0x702C
.DEFINE        P_DAC_Ctrl        0x702A


.CODE
.PUBLIC _main                    //主程序
_main:
    R1=0xffff                    //r1 的值为 0xffff
    [P_IOB_ATTRI]=r1             //IOB 口设置为带数据缓存器的低电平输出口
    [P_IOB_DIR]=r1
    R1=0x0000
    [P_IOB_DATA]=r1;
```

```
        R1=0x0001                   //选择通道 LINE_IN 为 IOA0
        [P_ADC_MUX_Ctrl]=R1
        R1 = 0x0001                 //设置 P_ADC_Ctrl 单元，允许 A/D 转换
        [P_ADC_Ctrl] = R1

            NOP                     //等待
            NOP
            NOP
            NOP
            NOP
    _AD:
            R2=[P_ADC_MUX_Ctrl]   //读寄存器[P_ADC_MUX_Ctrl]的 B15 位
                                  //判断是否转换完毕
            TEST R2,0x8000
            JZ_AD                  //否，继续转换
            R1=[P_ADC_MUX_DATA] //是，则读出[P_ADC_MUX_DATA]转换结果
                                  //同时触发 A/D 重新转换
            [P_IOB_DATA]=R1;       //IOB 口输出转换结果
            JMP    _AD             //跳转到_AD 处
```

[例 2.18]：

```
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////
    //Note:采用自动方式即定时器 A 溢出执行 ADC 转换，通过 A/D 将 MIC_IN 输入的
语
    //音信号转换为数字信号，再通过 D/A 的两个通道 AUD1 和 AUD2 播放。
    // Date: 2000/12/07
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///////
    .DEFINE  TIMER_DATA_FOR_8KHZ (0xffff - 1500)
    // 时钟频率为 Fosc/2,采样率为 8kHz
.DEFINE         P_TimerA_Ctrl   0x700b
.DEFINE         P_TimerA_Data   0x700a
.DEFINE         P_ADC           0x7014
.DEFINE         P_ADC_Ctrl      0x7015
.DEFINE         P_DAC1          0x7017
.DEFINE         P_DAC2          0x7016
.DEFINE         P_DAC_Ctrl      0x702A
.DEFINE         P_INT_Ctrl      0x7010
.DEFINE         P_INT_Clear     0x7011

//AD 初始化子程序
.CODE
```

```
_InitAD_DA:
        INT OFF;
        R1 = 0x0030;                    // 时钟频率为 CLKA 的 Fosc/2
        [P_TimerA_Ctrl] = R1;
        R1 = TIMER_DATA_FOR_8KHZ; // 数据采样率为 8kHz
        [P_TimerA_Data] = R1;           //置入计数初值

        R1 = 0x0015;                    // 设置具有 AGC 控制功能
        [P_ADC_Ctrl] = R1;              // 采用自动方式、且通过 MIC_IN 通道输入
        R1 = 0x00A8;
        //通过定时器 A 的溢出锁存数据，ADC 为通过 TimerA 溢出触发 ADC 自动方
        //式转换
        [P_DAC_Ctrl] = R1;
        R1 = 0x1000;
        [P_INT_Ctrl] = R1;              // 设置 IRQ1_TM 中断

        INT IRQ;                        //开中断
        RETF                            //子程序返回
//主程序
.PUBLIC _main                           //主程序的开始
_main:
    CALL _InitAD_DA                     //调用 AD 初始化子程序
loop:                                   //主程序循环
    NOP
    JMP loop

.text
.PUBLIC _IRQ1                           //中断子程序
_IRQ1:
        PUSH R1TO [SP]                  //堆栈
        R1 = [P_ADC]                    //读取 A/D 转换的数值
        [P_DAC1] = R1                   //DAC 输出
        [P_DAC2] = R1
        R1 = 0x1000
        [P_INT_Clear] = R1               //清中断标识位
        POP R1 FROM [SP]                 //出栈
        RETI                             //中断返回
```

2.10.2 ADC 直流电气特性

ADC 直流电气项目	项目符号	最小值	典型值	最大值	单位
ADC 分辨率	RESO		10		bit
ADC 有效位数	ENOB	8			bit

ADC 信噪比	SNR	50			dB
ADC 积分非线性	INL		±4		LSB[1]
ADC 差分非线性	DNL		±0.5		LSB
ADC 转换率	F_{CONV}			96K[2]	Hz
电源电流 @Vdd=3V	I_{ADC}		3.4		mA
功耗 @Vdd=3V	P_{ADC}		10.2		mW

注：

[1] LSB 表示为最小有效单位，在 VRT=3V 的情况下，1LSB 为 2.93 mv。

[2] 此由最大采样率(Samplerate_max)得来，即 Samplerate_max =ADC 响应率/16=1536KHz/16=96KHz。

2.10.3 MIC_IN 通道方式 ADC

2.10.3.1 ADC 范围

MIC_In 通道方式的 ADC，其最大参考电压可达 AV_{dd}，即来自 MIC_In 通道的模拟信号的电压范围从 0V 到 AV_{dd}。信号从 MIC_In 管脚输入，经过缓存器后被放大。放大器的增益倍数可以通过外部电路进行调整。然后，AGC 把 MIC_In 信号控制在指定的范围内。

2.10.3.2 设置

用户必须先把 P_ADC_Ctrl(写)($7015H)单元的第 0 位 ADE 置为‘1’，第 1 位 MIC_ENB 置为‘0’，从而激活 A/D 和 MIC_In 通道(上电复位之后，VMIC 默认被打开)。然后，把第 2 位 AGCE 置为‘1’，激活 AGC。第 3、4 位用于设定 MIC_In 通道的 ADC 的触发方式(Timer 锁存和直接方式)。P_ADC_MUX_Ctrl(读/写)($702BH)的第 0～2 位为‘0’时，模拟电压信号通过 MIC_In 通道输入。

2.10.3.3 操作

当触发 MIC_In 通道输入后，产生一个开始信号(b15(RDY) = 0)。然后，DAC0 输出信号通过同外部输入信号进行按位的比较以得到输入信号的数字信号。逐次逼近式模-数转换器首先设置最高位，然后清除 SAR 的其它位(10 0000 0000B)。这时，DAC0 输出电压(1/2 满量程) 与输入电压 V_{in} 进行比较。如果 $V_{in} > V_{DAC}$，保持原先设置为‘1’的位(最高有效位)仍为‘1’；否则，该位会被清‘0’。这个过程重复 10 次，直到这些位都被比较过。转换结果保存到 SAR。A/D 转换完成之后 P_ADC_Ctrl (读) ($7015H)的第 15 位 RDY 被置为”1”。

1.Timer 锁存模式

当 10 位 A/D 转换完成时，用户通过读取 P_ADC ($7014H)/P_ADC_MUX_Data($702BH)单元可以获得 10 位 A/D 转换的数据。

定时器事件可由 Timer A、Timer B。从 P_ADC(R) ($7014H)读取数据后，不论处于直接状态还是 Timer 状态触发，P_ADC_Ctrl (读) ($7015H)的第 15 位 RDY 将被清除为”0”且开始继续执行 A/D 转换。若 A/D 转换结果没被读取，第 15 位 RDY 继续保持为”1”且不再继续执行 A/D 转换。注意，P_ADC_Ctrl (读) ($7015H)的第 15 位 RDY 与 P_ADC_MUX_Ctrl (R) ($702BH) 的第 15 位 RDY 的作用基本相同。

2. 直接模式

设置 P_DAC_Ctrl(W)($702AH)的第 3 和 4 位，可以指定 MIC ADC 的工作模式为直接模式。进行 A/D 转换之前，用户必须先读取 P_ADC (读) ($7014H)单元的内容，以激活 ADC，然后通过读取 P_DAC_Ctrl($702AH)单元的第 15 位，循环查询 ADC 的状态。完成 A/D 转换之后，程序再一次读取 P_ADC (读) ($7014H)单元的内容来得到转换结果。

2.10.3.4 MIC_In 前端放大器

MIC_In 通道有两阶 OP 放大器(参考 ADC 的组成图)。屏蔽 AGC 后，第一阶放大器的增益为 ***15V/V***。二阶放大器(OPAMP2)的增益为 ***60K/(1K+Rext)***，可以通过 Rext 来调整增益的大小，Rext 的增减和 OPAMP2 的增益的变化呈反比。例如，如果 Rext＝5.1K(ohm)，OPAMP2 的增益＝60K/(1K+5.1K) = 9.8 (19.8dB)，全部的 MIC 放大器的增益＝OPAMP1x OPAMP2 = 15 X 9.8 = 147(43.3dB)。

AGC 被激活之后(P_ADC_Ctrl (W) ($7015H)的 b2=1)能自动调整增益的值，以防止信号饱和。当 OPAMP2 的输出＞0.9Avdd 时，AGC 自动降低 OPAMP1 的增益，以防止被放大的信号饱和。

2.10.4 LINE_IN 模式的 ADC 操作

SPCE061A 提供 7 个 Line_In 通道，它们与 IOA[6:0]共用 7 个管脚。

如果把这七个管脚当作 Line_In 通道，用户必须首先把相应的 IOA 管脚设置为“输入”。注意，由于 IO 口带有内部上拉和下拉输入电阻，这会影响外部 Line_In 信号的电平。所以，IOA[6:0]最好被设置成悬浮的输入口，用于 Line_In 通道输入。

2.10.4.1 ADC 范围

通过设置 P_ADC_Ctrl(写)($7015H)单元的第 7 位 VEXTREF，可以设置由 Line_In 通道输入的最大电压值。VEXTREF＝0 时，最大电压可达 AV_{DD}，即来自 Line_In 通道的模拟信号的电压范围从 0V 到 AV_{dd}。VEXTREF＝1 时，VEXTREF 管脚被激活，这时，必须输入外部信号到该管脚作为 Line_In 通道的最大电压。VEXTREF 可取的值的范围从 0V 到 AV_{dd}。所以，Line_In 通道的输入电压范围从 0V 到 VEXTREF，VEXTREF 的值越低，Line_In 通道的电压范围越小。也就是说，输入的信号的信噪比 SNR 越低。SPCE 提供了一个内置的 2V 电压源 (通过设置 P_ADC_Ctrl(写)($7015H)单元的第 8 位 V2VREFB＝0 来激活) ，它可以被连接到 VEXTREF 管脚，作为 Line_In 通道的最大参考电压。

2.10.4.2 设置

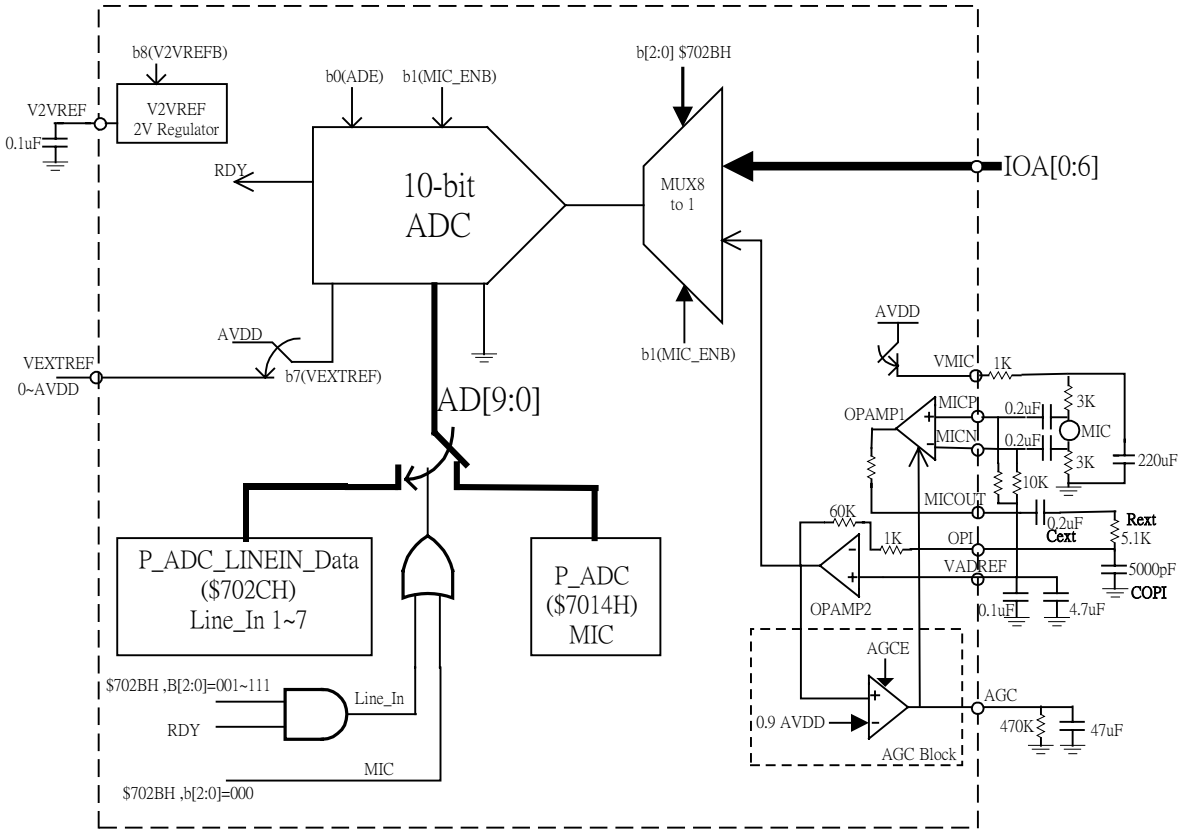
由于 SPCE061A 共拥有 8 个 A/D 转换通道，但只有一个 ADC，所以用户必须在切换通道之前通过查看 P_ADC_MUX_Ctrl (读) ($702BH)单元/ P_ADC_Ctrl (读) ($7015H)单元的第 15 位 RDY 的值，以确认 ADC 为空闲状态。通道切换可通过设置 P_ADC_MUX_Ctrl(读/写)($702BH)单元的第 0~2 位完成。如果 RDY 不为 1，即 ADC 正在工作，这时对 P_ADC_MUX_Ctrl(读/写) ($702BH)单元的第 0~2 位进行的任何操作都无效。

2.10.4.3 操作

MIC_In 通道的 A/D 转换拥有多种触发方式(通过设置 P_DAC_Ctrl(写)($702AH)单元的第 3 和 4 位)，而 Line_In 通道的 A/D 转换只能通过用户读取 P_ADC_LINEIN_Data(读)($702CH)单元的数据来触发。当 MIC_In 通道处于定时器锁存状态时，MIC_In 通道的优先级高于 Line_In 通道。所以，同时有来自 MIC_In 和 Line_In 通道的 A/D 转换时，Line_In 通道的 A/D 转换会被 MIC_In 通道的 A/D 需求打断。为保证从 P_ADC_LINEIN_Data(读)($702CH)单元读取到正确的数据，用户必须通过 P_ADC_MUX_Ctrl(读/写) ($702BH)单元的第 14 位 FailB 的值，确认 A/D 是否成功/被打断。

当 MIC_In 通道处于定时器锁存状态时/第一次 MIC_In 通道的 A/D 转换完成后，查看 P_ADC_MUX_Ctrl(读/写) ($702BH)单元的值是非常必要的。

当采用 Line_In 通道的 A/D 转换时，通过读 P_ADC_LINEIN_Data(读)($702CH)单元的值，可以开始进行 A/D 转换操作，同时，P_ADC_MUX_Ctrl(读/写)($702BH)单元的第 15 位 RDY 被清除为"0"。当 RDY 变为 1 时，表示 ADC 完成工作，如果 P_ADC_MUX_Ctrl(读/写)($702BH)单元的第 14 位 FailB 的值为 1，用户可以从 P_ADC_LINEIN_Data(读)($702CH)得到转换结果。注意，读 P_ADC_LINEIN_Data(读)($702CH)单元的值可再次触发 A/D 转换。如果 FailB 的值为 0，说明 Line_In 通道的 A/D 转换被 MIC_In 通道的 A/D 操作打断，P_ADC_LINEIN_Data(读)($702CH)单元的值是一个错误的值。



ADC 简图

注意，用户可以通过调整麦克风电路来增强抗扰度。为达到更好的前端 A/D 效果，可以参考上面的 ADC 简图和 RC 表，调整麦克风带通滤波器的带宽。

处于定时器锁存状态时，MIC_In 通道的 A/D 转换的优先级高于其他 Line_In 通道的 A/D 转换，以便保证能及时捕捉到从 MIC_In 通道输入的声音信号。多路开关随即可自动地接通到 MIC_In 通道，执行 MIC_In 通道的 A/D 转换。如果 Line_In 通道的 A/D 操作被 MIC_In 通道的 A/D 打断，P_ADC_MUX_Ctrl(读/写)($702BH)单元第 14 位 FailB 的值被置为 1。相反，如果当前正在执行 MIC_In 通道的 A/D，则用户无法通过设定 P_ADC_MUX_Ctrl(读/写)($702BH)单元的值控制通道转换。注意，只有当 MIC_In 通道处于定时器锁存状态且 Line_In 通道正在被占用时 FailB 才起作用。

2.11 DAC 方式音频输出

SPCE061A 为音频输出提供两个 DAC 通道，DAC1 和 DAC2 输出的模拟电流信号通过 DAC1 和 DAC2 管脚输出。DAC 的输出范围从 0x0000 到 0xFFFF。如果 DAC 的输出数据被处理成 PCM 数据，必须让 DAC 输出数据的直流电平保持为 0x8000，且仅高 10 位数据起作用。DAC1 和 DAC2 的输出数据应写入 P_DAC1(写) ($7017)和 P_DAC2(写) ($7016)单元。上电复位后，两个 DAC 均被自动打开，此时会消耗少量的电流(几毫安)。所以如不需要用它们，尽量将 P_DAC_Ctrl(写)($702AH)单元的第 1 位置为‘1’，关闭 DAC 输出。

DAC 的直流电压必须保证平稳地变化。否则，可能由于电压的突变引起扬声器产生杂音。采用 ramp up/down 技术，可以减缓电压的变化幅度，从而输出高质量的音频数据。它的应用场合包括：被唤醒/上电复位后首次使用 DAC 时，上电复位功能被关闭/进入睡眠状态之前。

P_DAC2(读/写)(7016H)

在 DAC 方式下，该单元是一个带 10 位缓冲寄存器 DAR2 的 10 位 D/A 转换单元(DAC2)。

b15 – b6	b5 – b0
DA2_Data(读/写)	---

P_DAC2(写)：通过此单元直接写入 10 位数据到 10 位缓存器 DAR2，来锁存 DAC2 的输入数字量值(无符号数)。

P_DAC2(读)：从 DAR2 内读出 10 位数据。

P_DAC1(读/写)(7017H)

该单元为一个带 10 位缓存器(DAR1)的 10 位 D/A 转换单元(DAC1)。用于向 DAR1 写入或从其中读出 10 位数据。

b15 – b6	b5 – b0

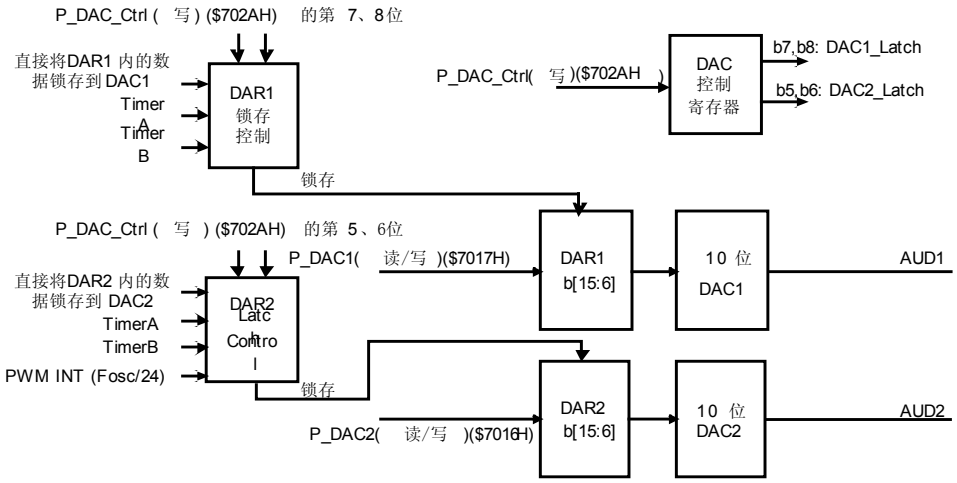
DA1_Data(读/写)	---

P_DAC_Ctrl(写) (702AH)

DAC 音频输出方式的控制单元（如表 2.20所示），其中第 5~8 位用于选择 DAC 输出方式下的数据锁存方式；第 3、4 位用来控制 A/D 转换方式；第 1 位总为‘0’，用于双 DAC 音频输出。b9~b15 为保留位。表 2.20详细地列出了 P_DAC_Ctrl 单元的 b3~b8 的控制功能。

表2.20 P_DAC_Ctrl 单元

b8	b7	b6	b5	b4	b3
DAC1_Latch(写)		DAC2_Latch(写)		AD_Latch(写)	
00：直接将 DAR1 内数据锁存到 DAC1 内(缺省设置) 01：通过 TimerA 溢出 DAR1 内的数据锁存到 DAC1 内 10：通过 TimerB 溢出将 DAR1 内的数据锁存到 DAC1 内 11：通过 TimerA 或 TimerB 的溢出将 DAR1 内的数据锁存到 DAC1 内		00：直接将 DAR2 内的数据锁存到 DAC2 内(缺省设置) 01：通过 TimerA 溢出将 DAR2 内的数据锁存到 DAC2 内 10：通过 TimerB 溢出将 DAR2 内的数据锁存到 DAC2 内 11：通过 TimerA 或 TimerB 的溢出将 DAR2 内的数据锁存到 DAC2 内		00：通过读 ADC(读)(7014H)触发 ADC 自动转换(缺省设置) 01：通过 TimerA 溢出触发 A/D 转换 10：通过 TimerB 溢出触发 A/D 转换 11：通过 TimerA 或 TimerB 的溢出触发 A/D 转换	



音频输出的结构

2.11.1.2 DAC 输出特性

最初，DAC 被设计用作音频输出设备。这一章，向用户说明它在音频输出方面的应用。通常，DAC 的最大输出电流和 Avdd 成正比。参考表“AVdd vs. DAC 输出电流 I_{MAX}”，DAC 的最大输出电流范围是“典型电流值±10%”。例如，AVdd=3.0V、DAC@3mA 时，DAC 的最大输出电流范围是 2.7mA~3.3mA。

由于已知 DAC 的最大输出电流，则模拟输出电压的范围由 DAC 的负载而定。由于 DAC 本身的物理特性，最大的输出电压将比 Avdd 低 0.3V~0.4V。例如，当电流为 3mA，AVdd=3V，电阻为 866Ohm 时，向 P_DAC 写入 0xFFC0，这时，最大的输出电压为 3mAX867Ohm=2.6V。如果电阻值大于 867Ohm，输出电压值的变化不一定根据电阻的值呈正比。如表所注的数据为 AVdd=3V 时，DAC 的最大输出电压和电阻的变化表。

表2.21 AVdd vs. DAC 输出电流 I_{MAX}

AVdd(V)	Min. DAC current(mA)	Typical current(mA)	Max. DAC current(mA)
2.4	2.16	2.4	2.64
2.7	2.43	2.7	2.97
3.0	2.7	3.0	3.3
3.3	2.97	3.3	3.63
3.6	3.24	3.6	3.96

[例 2.19]：

```
//*************************************************************
// Note:本实验采取直接方式，通过编程实现一个锯齿波，将实验板的
// DAC 输出端接示波器可以观察到锯齿波形,同时也可以听到 AUD1 和 AUD2
//两端的扬声器有持续间断的声音。
// Date: 2002/06/19
//*************************************************************
.DEFINE        P_DAC_Ctrl     0x702A
.DEFINE        P_DAC1         0x7017
.DEFINE        P_DAC2         0x7016
.CODE
.PUBLIC _main;
_main:
      INT OFF;                        //关中断
      R1=0x0000;                      //设置 DAC 输出的数据锁存方式，A/D 转换方式
      [P_DAC_Ctrl]=R1;
      R3=0x0040                       //D/A 转换为 10 位，即 B15~B6
      R1=0x0;
MainLoop1:
      [P_DAC1]=R1;                    //输出 0
      [P_DAC2]=R1;

      CALL Delay                      //调用延时子程序
      R1+=R3                          //幅度提高 1 个台阶
     JMP MainLoop1                    //返回 MainLoop1

Delay:                                //延时子程序
```

```
    R2=0
DelayLoop:
    R2+=2048
    JNZ DelayLoop                 //记到溢出时 R2 变为 0，延时结束
    RETF                          //子程序返回
```

2.12 低电压监测/低电压复位 (LVD/LVR)

SPCE061A 可通过编程设置低电压监测和低电压复位功能，目的是为了通过对系统的电源电压进行监控，而使系统运行在一个正常、可靠的工作环境，并在一旦出现电源异常的情况下能立即采取相应的措施，使系统及时恢复正常。

2.12.1 低电压监测

低电压监测功能可以提供系统内电源电压的使用情况。如果系统电压 Vcc 低于用户设定的电压监测低限电压 V_{LVD}，P_LVD_Ctrl 单元的第 15 位(LVD 监测标志位)将被置为“1”；反之，当 Vcc > V_{LVD} 时，该位被置为“0”。

SPCE061A 具有 4 级电压监测低限：2.4V、2.8V、3.2 和 3.6V，可通过对 P_LVD_Ctrl 单元编程进行控制，参见图 2.16。假定 V_{LVD}=3.2V，当系统电压 Vcc 低于 3.2V 时，P_LVD_Ctrl 单元的第 15 位返回值为“1”，这样，CPU 可以通过可编程电压监测低限来完成低电压监测。系统默认的电压监测低限为 2.4V。

P_LVD_Ctrl(读/写)(7019H)

b15	b14 - b2	b1	b0
Result_of_LVD	---	LVD_level_define(写)	
0：$V_{DD} > V_{LVD}$ 1：$V_{DD} < V_{LVD}$			

b1	b0	电压监测低限 (V_{LVD})
0	0	2.4V* (默认)
0	1	2.8V
1	0	3.2V
1	1	3.6V

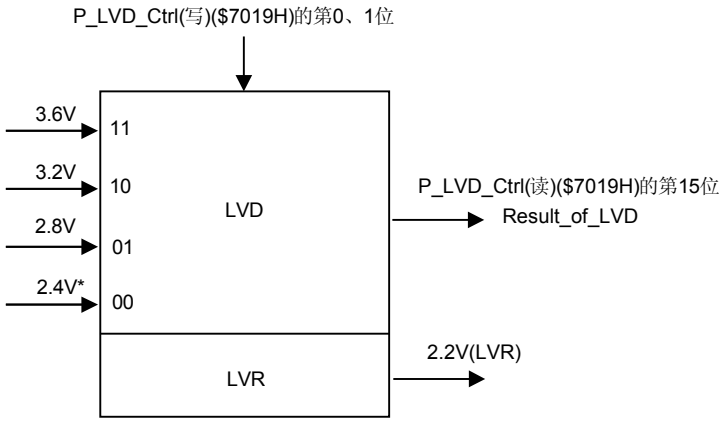


图 2.16　低电压监测(LVD)/低电压复位(LVR)

2.12.2 低电压复位(LVR)

通过某种方式，使单片机内存各寄存器的值变为初始的操作称为复位。SPCE061A 复位电路如图 2.17 所示，在 RESB 端加上一个低电平就可令其复位。该电路具有手动和上电复位两种功能。

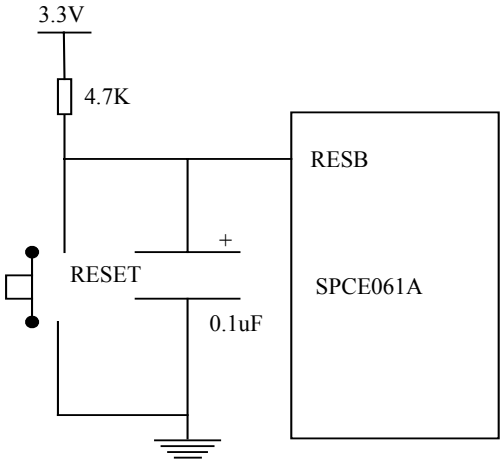


图 2.17　复位电路

当电源电压低于 2.2V 时，系统会变得不稳定且易出故障。导致电源电压过低的原因很多，如电压的反跳、负载过重、电池能量不足……。如果电源电压低于 2.2V 时，会在 4 个时钟周期之后产生一个复位信号，使系统复位。LVR 时序如图 2.18 所示。

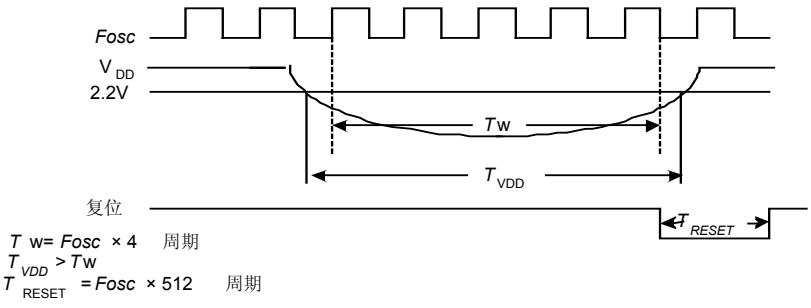


图 2.18　复位示意图

2.13 串行设备输入输出端口(SIO)

串行输入输出端口 SIO 提供了一个 1 位的串行接口，用于与其它设备进行数据通讯。在 SPCE061A 内通过 IOB0 和 IOB1 这 2 个端口实现与设备进行串行数据交换功能。其中，IOB0 用来作为时钟端口(SCK)，IOB1 则用来作为数据端口(SDA)，用于串行数据的接收或发送。参见IOB 口的特殊功能。

P_SIO_Ctrl(读/写)(701EH)

用户必须通过设置 P_SIO_Ctrl (701EH) (读/写)单元的第 7 位，将 IOB0、IOB1 分别设置为 SCK 管脚和 SDA 管脚。如果该单元的第 6 位被设置为“0”，串行输入/输出接口可以从用户指定的地址读出数据。该单元的第 3、4 位的作用是让用户自行指定数据传输速度；而通过设置第 0、1 位，可以指定串行设备的寻址位数。

SIO 的控制选择在 P_SIO_Ctrl 单元内进行，详见表 2.22。

表2.22 P_SIO_Ctrl 单元

b7	b6	b5	b4	b3	b2	b1	b0	设置功能说明
SIO_Config	R/W	R/W_EN	Clock_Sel		—	Addr_Select		
X	X	X	X	X	—	0	0	串行设备地址(缺省)设置为 16 位 (A0~A15)
X	X	X	X	X	—	0	1	无地址设置
X	X	X	X	X	—	1	0	串行设备地址设置为 8 位(A0~A7)
X	X	X	X	X	—	1	1	串行设备地址设置为 24 位(A0~A23)
X	X	X	0	0	—	X	X	数据传输速率设为 *CPUClk*/16(缺省设置)
X	X	X	0	1	—	X	X	无用
X	X	X	1	0	—	X	X	数据传输速率设为 *CPUClk*/8
X	X	X	1	1	—	X	X	数据传输速率设为 *CPUClk*/32
1	X	X	X	X	—	X	X	设置 IOB0=SCK(串行接口时钟端口)，IOB1=SDA (串行接口数据端口) 。用户不必设置 IOB0 和 IOB1 的输入输出状态
0	X	X	X	X	—	X	X	用作普通的 I/O 口(默认)
X	1	X	X	X	—	X	X	设置数据帧的写传输
X	0	X	X	X	—	X	X	设置数据帧的读传输(默认)
X	X	1	X	X	—	X	X	关断读/写帧的传输
X	X	0	X	X	—	X	X	接通读/写帧的传输(默认)

关于 CPU 时钟 CPU_CLK 请参考系统时钟部分。**P_SIO_Data(读/写)(701AH)**

P_SIO_Data(读/写)(701AH)

该单元为接收/发送串行数据的缓冲单元。向该单元写入或读出数据，可按串行方式发送或接收数据字节。用户须通过写入 P_SIO_Start (701FH)单元来启动 P_SIO_Data (701AH)单元与串行设备数据交换的过程。传输是从串行设备的起始地址(由 P_SIO_Addr_Low(如表 2.24), P_SIO_Addr_Mid(如表 2.25) 和 P_SIO_Addr_High(如表 2.26)3 个单元指定)开始，然后是数据。

进行写操作时，第一次向 P_SIO_Data (写) (701AH 如表 2.23)单元写入数值是在写入 P_SIO_Start(写)单元任意一个数值之后，即必须先启动数据传输，随后，SIO 将从串行设备的起始地址开始传送，后面接着传送写入 P_SIO_Data 单元中的 8 位数据。

进行读操作时，第一次读 P_SIO_Data (读) (701AH)单元数据是在向 P_SIO_Start (写) (701FH)单元写入任一数值后，SIO 将首先传送串行设备的起始地址。

表2.23　P_SIO_Data

b7	b6	b5	b4	b3	b2	b1	b0
D7	D6	D5	D4	D3	D2	D1	D0

P_SIO_Addr_Low(读/写)(701BH)

串行设备起始地址的低字节(默认值为 00H)。

表2.24 P_SIO_ Addr_Low

b7	b6	b5	b4	b3	b2	b1	b0
A7	A6	A5	A4	A3	A2	A1	A0

P_SIO_Addr_Mid(读/写)(701CH)

串行设备起始地址的中字节(默认值为 00H)。

表2.25 P_SIO_Addr_ Mid

b7	b6	b5	b4	b3	b2	b1	b0
A15	A14	A13	A12	A11	A10	A9	A8

P_SIO_Addr_High(读/写)(701DH)

串行设备起始地址的高字节(默认值为 00H)。

表2.26 P_SIO_Addr_High

b7	b6	b5	b4	b3	b2	b1	b0
A23	A22	A21	A20	A19	A18	A17	A16

P_SIO_Start(读/写)(701FH)

向 P_SIO_Start(写)(701FH 如表 2.27)单元写入任意一个数值，可以启动数据传输。接着，当对 P_SIO_Data (701AH)单元读写操作时会使 SIO 根据 P_SIO_Addr_Low、

P_SIO_Addr_Mid 和 P_SIO_Addr_High 的内容传输读/写操作的起始地址，之后再读写 P_SIO_Data 单元时 SIO 将不再传输此起始地址。

如果需要重新指定一个起始地址进行数据传输，用户可以向 P_SIO_Stop (7020H)单元写入任意一个数值以停止 SIO 操作，然后向 P_SIO_Addr_Low、 P_SIO_Addr_Mid 和 P_SIO_Addr_High 写入新的地址；最后，向 P_SIO_Start(写)(701FH)单元写入任意一个数值重新启动 SIO 操作。

读出 P_SIO_Start(701FH)单元可获取 SIO 的数据传输状态，该单元的第 7 位 Busy 为占用标志位，Busy=‘1’表示正在传输数据，传输操作完成后，该位将被清为‘0’，可以开始传输新的数据字节。

表2.27　P_SIO_Start

b7	b6	b5	b4	b3	b2	b1	b0
Busy	-	-	-	-	-	-	-

P_SIO_Stop(写)(7020H)

向 P_SIO_Stop(写)(7020H)单元写入任一数值，可以停止数据传输。通常，停止数据传输的终止指令应出现在激活数据传输的启动指令之前。但上电复位后的第一个启动命令之前不需要终止命令。

图 2.19 为 SIO 的读写操作时序。

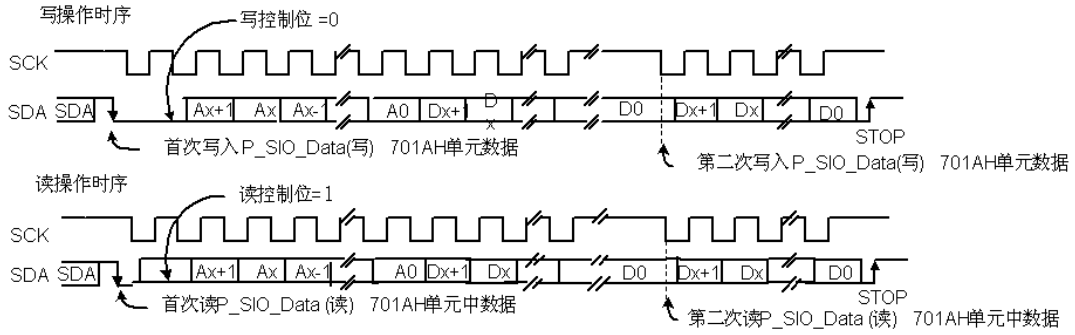


图 2.19　读/写时序

[例 2.20].

通过串行输入/输出接口将 SPCE061A 与 SPRS512C 串行静态 RAM(SSRAM，Serial Static RAM)连接起来（如图 2.20 所示）。在数据传输过程里，SPRS512 会自动在读/写操作的起始地址的基础上递增地址计数器。

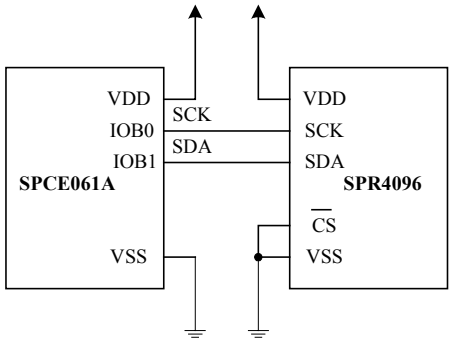


图 2.20　SPCE061A 与 SPR4096 的连接图

SPR4096 是凌阳公司推出的一款含有 4Mbit 的 FLASH 存储器，带有 SIO 接口。

从串行输入/输出接口向 SRAM 写入一个长度为 1 个字节的数据。SPR4096 有 24 位地址线，工作频率为 CPUclk/4，可进行串行数据帧的写操作。注意，首先应将 SIO 设置为输出口。

```
R1=0x00CB;                  //设置串行输入/输出控制端口(输出状态)
[P_SIO_Ctrl]=R1;            //B0,B1 口分别为 SCK,SDA 管脚，串行地址为 24
                            //位
R1=0x0000;                  //写入串行输入输出启动端口
[P_SIO_Start]=R1;
R1=0x0000;                  //写入起始地址到地址单元
[P_SIO_Addr_High]=R1;       // Addr23~Addr16 =0000 0000
[P_SIO_Addr_Mid]=R1;        //Addr15~Addr8 =0000 0000
[P_SIO_Addr_Low]=R1;        //Addr7~Addr0 =0000 0000
R1=0x0011;                  //向缓冲单元写入传送数据,d7~d0 =0001 0001
[P_SIO_Data]=R1;
L_busy:
R1=0x0080;                  //读出占用标志 Busy
TEST R1, [P_SIO_Start];     //忙，则没写完
JNZ L_Busy;
[P_SIO_Stop]=R1;            //向 P_SIO_Stop(写)(701FH)单元写入任意一个数
                            //值，结束数据帧的写操作
```

2.14 通用异步串行接口 UART

UART 模块提供了一个全双工标准接口，用于完成 SPCE061A 与外设之间的串行通讯。借助于 IOB 口的特殊功能和 UART IRQ 中断，可以同时完成 UART 接口的接收发送数据的过程。此外，UART 还可以缓冲地接收数据。也就是说，它可以在读取缓存器内当前数据之前接收新的数据。但是，如果新的数据被接收到缓存器之前一直未从中读取先前

的数据，会发生数据丢失。P_UART_Data (7023H) (读/写)单元可以用于接收和发送数据的缓存,向该单元写入数据，可以将发送的数据送入缓存器；从该单元读数据，可以从缓存器读出数据字节。UART 模块的接收管脚 Rx 和发送管脚 Tx 分别与 IOB7 和 IOB10 共用。

使用 UART 模块进行通讯时，必须事先分别将管脚 Rx(IOB7)、Tx(IOB10)设置为输入状态、输出状态。然后，通过设置 P_UART_BaudScalarLow (7024H)、P_UART_BaudScalarHigh (7025H) 单元指定所需波特率。同时，设置 P_UART_Command1(7021H)和 P_UART_Command2 (7022H) 单元以激活 UART 通讯功能。以上设置完成后，UART 将处于激活状态。设置 P_UART_Command1 单元的第 6、7 位可以激活 UART IRQ 中断，并决定中断是由 TxRDY 或 RxRDY 信号触发以及由二者共同触发。设置 P_UART_Command2 单元的第 6、7 位可以激活 UART Tx、Rx 管脚功能。当μ'nSP™接收或发送一个字节数据时，P_UART_Command2 (7022H)单元的第 6、7 位被置为“1”且同时触发 UART IRQ。无论 UART IRQ 中断是否被激活，UART 接收/发送功能都可以由 P_UART_Command2 (7022H)单元的第 6、7 位控制。在任意时刻读出 P_UART_Command2 (7022H)单元将清除 UART IRQ 中断标志。

注意：UART IRQ 中断向量存储在 0xFFFFH 单元，相对于其它 IRQ 中断来说，该中断的优先级别最低。

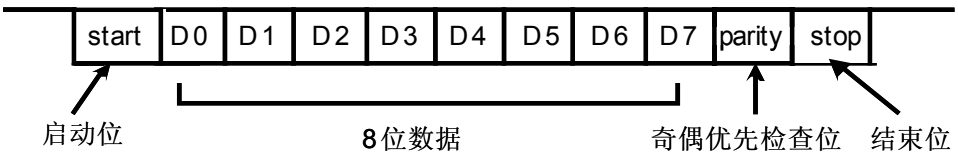


图 2.21　UART 数据帧的格式

P_UART_Command1(写)(7021H)

P_UART_Command1 单元为 UART 控制端口（如表 2.28所示）。设置该单元的第 2、3 位可以控制数据奇偶校验功能。第 6、7 位控制着 UART IRQ 中断，二者的区别在于：如果第 6 位 TxIntEn=1，中断由 TxRDY 信号触发，即数据发送完毕将产生 UART IRQ 中断；如果第 7 位 RxIntEn=1，中断由 RxRDY 信号触发，即数据接收完毕将产生 UART IRQ 中断。如果该单元的第 5 位 I_Reset=1，所有 UART 控制寄存器、状态寄存器将恢复为系统默认值。

表2.28 P_UART_Command1 单元

b7	b6	b5	b4	b3	b2	b1	b0	功能
RxIntEn	TxIntEn	I_Reset	-	Parity	P_Check	-	-	
1	–	–	–	–	–	–	–	允许 UART IRQ 中断(由 RxRDY 信号触发)
0	–	–	–	–	–	–	–	禁止 UART IRQ 中断
–	1	–	–	–	–	–	–	允许UART IRQ中断(由TxRDY信号触发)
–	0	–	–	–	–	–	–	禁止 UART IRQ 中断

–	–	1	–	–	–	–	–	内部复位信号复位
–	–	0	–	–	–	–	–	内部复位信号置位
–	–	–	–	1	–	–	–	激活偶校检功能
–	–	–	–	0	–	–	–	激活奇校检功能
–	–	–	–	–	1	–	–	激活奇偶校验功能
–	–	–	–	–	0	–	–	屏蔽奇偶校检功能

P_UART_Command1(写)(7021H)单元的缺省值为 00H。

P_UART_Command2(写)(7022H)

该单元写入时为 UART 数据发送/接收控制端口，第 6、7 位分别控制着数据发送和接收管脚的允通/禁止。P_UART_Command2(写)(7022H)单元的缺省值为 00H。

b7	b6	b5	b4	b3	b2	b1	b0
RxPinEn	TxPinEn	-	-	-	-	-	-
1：允通接收管脚 0：禁止接收管脚	1：允通发送管脚 0：禁止发送管脚						

此时 IOB7 和 IOB10 必须分别被设置为输入和输出管脚作为 Rx 和 Tx 管脚。当发送管脚被允通时，IOB10 Tx 输出管脚将自动被置为高电平。

P_UART_Command2(读)(7022H)

P_UART_Command2 单元读出为 UART 状态信息(如表 2.29所示)。第 7 位是 RxRDY 标志位，当接收到数据时该标志位被置为“1”，读 P_UART_Data 单元将清除该标志位；第 6 位是 TxRDY 标志位，当通过写入本单元第 6 位为'1'来允通发送管脚后，该标志位被置为“1”，表示发送器的数据缓存器为空，已准备好可以发送写入 P_UART_Data 单元的数据。

向 P_UART_Data 单元写入数据可以清除 TxRDY 标志位。读出 P_UART_Command2 (7022H)单元的第 3~5 位是传输错误标志位，如果在传输过程中发生错误，相应位将被置为“1”；读 P_UART_Data(7023H)单元数据将清除错误标志位。

表2.29 P_UART_Command2 单元

b7	b6	b5	b4	b3	b2	b1	b0
RxRDY	TxRDY	FE	OE	PE	-	-	-
1：数据已接收完毕 0：未接收到数据	1：数据发送已准备好 0：数据发送未准备好	1：存在帧错误 0：无帧错误	1：存在溢出错误 0：无溢出错误	1：存在奇偶校验错误 0：无奇偶校验错误			

以上错误信号代表传输过程可能出现的错误。表 2.30给出了出错的原因及解决方法。

表2.30

错误类型	原因	解决方法
FE (帧错误)	发送管脚 TX 和接收管脚 RX 的数据帧的格式或波特率不一致	1. 使用一致的数据格式 2. 设置一致的波特率
OE (溢出错误)	接收端 RX 接收数据的速度低于发送端 TX 发送数据的速度，从而导致 RX 端数据溢出	1. 提高接收数据的速度 2. 降低数据传输速度
PE (奇偶校验错误)	传输条件差，可能有噪声干扰	改善传输条件

P_UART_Data(读/写)(7023H)

b7	b6	b5	b4	b3	b2	b1	b0
数据							

P_UART_BaudScalarLow(读/写)(7024H)

P_UART_BaudScalarHigh (读/写)(7025H)

P_UART_BaudScalarHigh (7025H)和 P_UART_BaudScalarLow(7024H)单元的组合控制数据的传输速率(波特率)。UART 波特率的计算公式如下：

波特率= (Fosc / 4) / Scale--------当 Fosc=49.152MHz, 40.960MHz 或 32.768MHz

波特率= (Fosc / 2) / Scale--------当 Fosc=24.576MHz 或 20.480MHz

由此可得出 Scale 的值 (Scale 为 7024H 单元和 7025H 单元组成的十进制整数)。

表 2.31列出当 Fosc = 24.576MHz 或 49.152MHz 时常用的波特率值。

表2.31

波特率(bps)	高字节(7025H)	低字节(7024H)	Scale (十进制)	实际波特率(bps)
1500 (最小值)	1FH	FFH	8192	1500
2400	14H	00H	5120	2400
4800	0AH	00H	2560	4800
9600	05H	00H	1280	9600
19200	02H	80H	640	19200
38400	01H	40H	320	38400
48000 (默认值)	01H*	00H*	256	48000
51200	00H	F0H	240	51200
57600	00H	D5H	213	57690
102400	00H	78H	120	102400
115200 (最大值)	00H	6BH	107	114841

[例 2.21]: 用 UART 来接收 PC 机的 RS232 串行接口的数据。由于 UART 每次只接收 8 位数据，所以在接收了 2 个 8 位数据之后才将其存入 16 位的 SRAM 内。

```
R1=0x0480;                          //将 IOB10、IOB7 分别设为输出、输入状态
[P_IOB_Attrib]=R1;
R1=0x0400;
[P_IOB_Dir]=R1;
R1=0x006b;                          //设置波特率= 12.288MHz/107 = 114.84KHz
[P_UART_BaudScalarLow]=R1;          //(与 115.2KHz 最接近，所以可以选择
R1=0x00;                            //115.2KHz)
[P_UART_BaudScalarHigh]=R1;
```

```
//***********设置 UART_command1 和 UART_command2 单元*****************//
        R1=0xc0；
        [P_UART_Command1]=R1；
        [P_UART_Command2]=R1；
//************************主程序*****************************************//
 L_begin_loop：
        R4=0；                          // SRAM 的接收数据队列的起始地址
 L_loop：
        R2=0；
        R2=R2 LSL 4；                   //清除移位缓存器
        CALL F_UART_RECV；
        R1=R1 LSL 4；                   //左移 8 位
        R1=R1 LSL 4；
        R3=R1；
        R3=R3 and 0xFF00；              //R3 =高 8 位
        R2=0；
        R2=R2 LSL 4；                   //清除移位缓存器
        CALL F_UART_RECV；
        R1=R1 and 0x00FF；               //R1 =低 8 位
        R1=R1 or R3；                    //R1=从 UART 接收来的最终数据字
        [R4++]=R1；                      //将接收的数据字存入 SRAM 的数据队列
        CMP R4，0x800；                  //SRAM 中的数据队列接收到 2048 个数据字
         JNE L_loop；
        JMP L_begin_loop；
//************************ UART 子程序*****************************//
F_UART_RECV：
    PUSH R2，R3 to [SP]；
 L_RxRDY：
    R2= 0x0080；                        //检查 RxRDY 是否为 1
    TEST R2， [P_UART_Command2]；
    JZ  L_RxRDY                         //比较结果为 0，数据未接受，循环
    R1 = [P_UART_Data]；                //再次写入数据
    POP R2，R3 from [SP]；
    RETF；                              //返回到主程序
```

2.15 保密设定

如果希望将内部的闪存进行保密设定，可将 P_{FUSE} 接 5V, P_{VIN} 接 GND 并维持 1s 以上即可将内部保险丝熔化，此后就无法再完成 read, download 和 debug 等功能。因此用户使用过程中一定要慎重。

2.16 看门狗计数器（WatchDog）

SPCE061A 的 WatchDog 的清除时间周期为 0.75s。因为 WatchDog 的溢出复位信号 WatchDog_Reset 是由 4Hz 时基信号经 4 分频之后产生的，即每 4 个 4Hz 时基信号(1s)将会产生一个 WatchDog_Reset 信号，如图 2.22 所示。而清除 WatchDog 的 WatchDog_Clear 信号却可以发生在 4Hz 信号(0.25s)之间的任意一个时刻点上。假如 WatchDog_Clear 信号发生在 4Hz 信号尾端的 0.01s 即第 0.25s 时刻，此时虽然 WatchDog 被清掉，但由于它发生在 4Hz 信号之后，再经 3 个 4Hz 信号即 0.75s，如果一直没有 WatchDog_Clear 信号，便会产生出一个 WatchDog_Reset 信号。

注意：SPCE061A 的看门狗功能是上电自动使能，不能够被屏蔽。因此用户使用时，注意要在 0.75s 内，进行清狗操作。

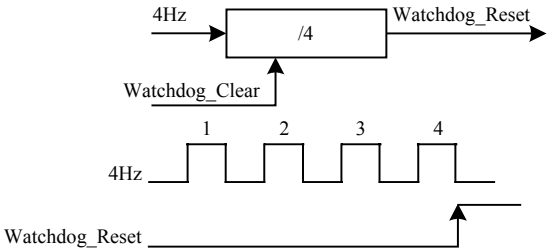


图 2.22　WatchDog 的结构和信号时序

当然，如果 WatchDog_Clear 信号发生在 4Hz 信号始端的 0.01s，则经过 0.99s 若无 WatchDog_Clear 信号便会产生 WatchDog_Reset 信号。因此，清除 WatchDog 的时间周期为 0.75s。

表 2.32 列出了 WatchDog 配置单元 P_WatchDog_Clear 及其内 B0 对 WatchDog 清除的控制。清除 WatchDog 只需写入 P_WatchDog_Clear 单元‘0x0001’即可。此外，若 32768Hz 振荡器被打开，则在空闲方式期间 WatchDog 功能是被激活的。

表2.32　WatchDog 的配置及 WatchDog 的清除

配置单元	读写属性	存储地址	配置单元功能说明
P_WatchDog_Clear	写	7012H	清除 WatchDog 单元
B15~B1	B0	控制位功能解释	
－－－	WatchDog_Clear		
0～0	0	不清除 WatchDog	
0～0	1	清除 WatchDog	