http://www.Microcontrol.cn 微控设计网

中国 MSP430 单片机专业网站

第一章 MSP430 单片机（微控制器）基础

第二章　　MSP430 单片机基本实践

第三章 MSP430 单片机外围接口实验

第四 章 MSP430 微处理器综合实践设计

第二章　　MSP430 单片机基本实践

2．1　　MSP430 程序设计基础

MSP430 单片机的程序设计可以使用汇编语言，也可以使用 C 语言，这一部分将讲述这些内容。如果使用汇编语言，需要了解汇编机器指令；如果使用 C 语言，需要了解 C 语言的相关知识。无论汇编还是 C 语言，都需要掌握 MSP430 微处理器的结构、原理、接口等才可以进行软件与硬件的设计。设计的基本过程可以用图 2.1.1 所示的流图来说明。在这一部分不讲解接口方面的知识。

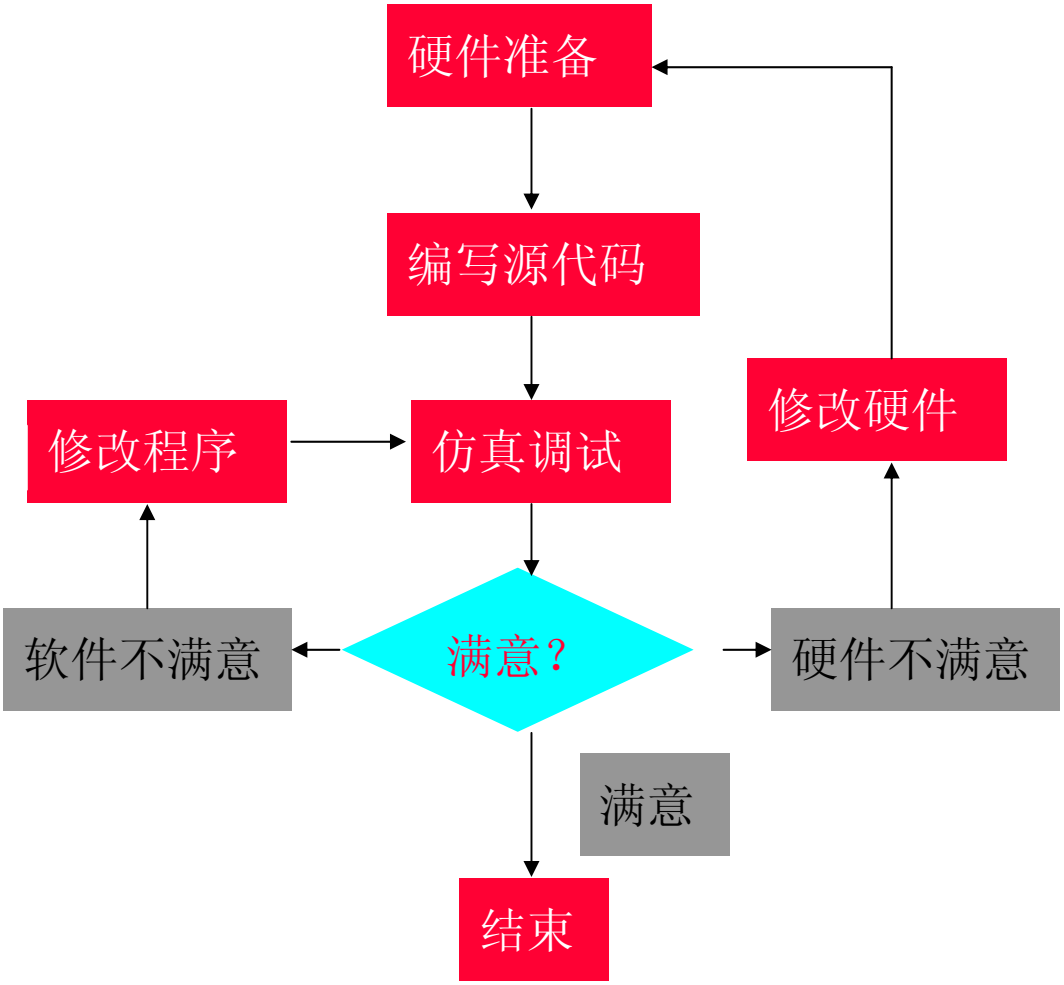


图 2.1.1　　项目设计开发简要流图

MSP430 的内核 CPU 结构是按照精简指令集和高透明的宗旨来设计的，使用的指令有硬件

执行的内核指令（只有 27 条）和基于现有硬件结构的高效率的仿真指令.仿真指令使用内核指令及芯片内额外配置的常数发生器 CG1、CG2。在讲解指令系统之前首先分析 MSP430 与指令系统相关的 CPU 结构与存储器系统。

2．1．1　MSP430 的 16 位 CPU

MSP430 系列采用的是“冯-诺依曼”结构，ROM、RAM 在同一地址空间，使用一组地址数据总线。中央处理单元采用了精简的、高透明的、高效率的正交设计，它包括一个 16 位的 ALU（算术逻辑运算单元），16 个寄存器，一个指令控制单元，16 个寄存器中有 4 个为特殊用途，扮演重要角色，分别是：程序计数器、堆栈指针、状态寄存器、常数发生器。程序流程通过程序计数器控制，而程序执行的现场状态体现在程序状态字中。在表 2.1 中对 16 个寄存器作了简要说明。

表 2.1　MSP430　CPU 中的 16 个寄存器

简写	功能
R0	程序计数器 PC，指示下一条将要执行的指令的地址
R1	堆栈指针 SP，指向堆栈的栈顶
R2	状态寄存器 SR/常数发生器 CG1
R3	常数发生器 CG2
R4	工作寄存器 R4
……	……
R15	工作寄存器 R15

1　程序计数器 PC

MSP430 的指令根据其操作数的多少，其指令长度分别为 1、2 或 3 字长。程序计数器指示出下一条即将执行的指令的地址。因此程序计数器 PC 的内容总是偶数，指向偶字节地址。其内容在调试程序时，可以通过寄存器窗口查看。

2.1.2　系统堆栈指针 SP

系统堆栈是在系统调用子程序或进入中断服务程序时，保护程序计数器 PC。而堆栈指针 SP 总是指向堆栈的顶部。系统在进行将数据压入堆栈操作时，总是先将堆栈指针 SP 的值减 2，然后再将数据送到 SP 所指的 RAM 单元。将数据从堆栈中弹出正好相反：先将数据从 SP 所指示的内存单元取出，再将 SP 的值加 2。堆栈的操作有两种情况：隐式与显示。系统对堆栈的操作为隐式，主要为自动保存 PC 的数值。在用户程序中也可对 SP 操作。下面举例说明。图 2.1（a）表示进行堆栈操作之前的 RAM 情况；图 2.1（b）表示执行 PUSH #8H 操作之后的情况；图 2.1（c）表示执行 POP R15 之后的情况。在后面将会详细讲解。

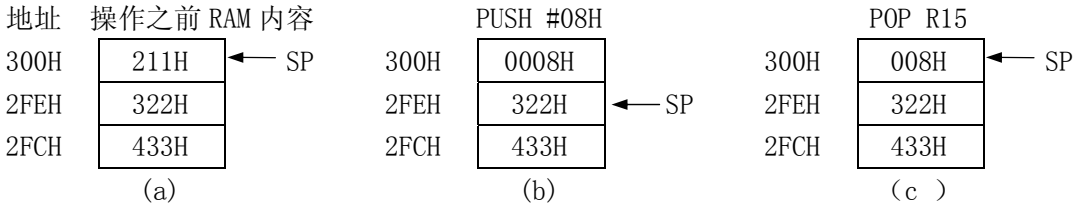


图 2.1　对堆栈的操作

2　状态寄存器 SR

状态寄存器记录程序执行过程中的现场情况，在程序设计中有相当重要的地位。MSP430

的状态寄存器为 16 位，目前只用到前 9 位，其结构如下：

15…9	8	7	6	5	4	3	2	1	0
保留	V	SCG1	SCG0	OSCOff	CPUOff	GIE	N	Z	C

Bit0　C　　进位标志　　当运算结果产生进位时置位，否则复位。

Bit1　Z　　零标志　　当运算结果为 0 时置位，否则复位。

Bit2　N　　负标志　　当运算结果为负时置位，否则复位。

Bit3　GIE　　中断控制位　置位允许中断，复位禁止所有的中断。该位由中断复位，RETI 指令置位，也可以用指令改变。

Bit4　CPUOff　　CPU 控制位　置位使 CPU 进入关闭模式，此时除了 RAM 内容、端口、寄存器保持外，CPU 处于停止状态，可用所有允许的中断将 CPU 从此状态唤醒。

Bit5　OSCOff　　晶振控制位　置位使晶体振荡器处于停止状态，CPU 从此状态唤醒只有在 GIE 置位的情况下由外部中断或 NMI 唤醒，要设置 OSCOff=1 必须同时设置 CPUOff=1。

Bit6　SCG0　　此位与位 7 一起控制系统时钟发生器的 4 种活动状态。

Bit7　SCG1　　此位与位 6 一起控制系统时钟发生器的 4 种活动状态。

SCG1　SCG0	时钟发生器的状态
0　0	SMCLK，ACLK
0　1	SMCLK，ACLK
1　0	ACLK
1　1	ACLK

Bit8　V　　当算术运算结果超出有符号数范围时置位。

3　常数发生器 CG1 与 CG2

在 16 个寄存器中 R2 与 R3 为常数发生器，利用 CPU 的 27 条内核指令配合常数发生器可以生成一些简洁高效的模拟指令。表 2.2 给出了 CG1、CG2 可以产生的常数。

表 2.2　　CG1、CG2 可以产生的常数

寄存器	As	常数	说明
R2	00	-	寄存器模式
R2	01	（0）	绝对寻址模式
R2	10	00004H	+4，位处理
R2	11	00008H	+8，位处理
R3	00	0000H	0，字处理
R3	01	00001H	+1
R3	10	0002H	+2，位处理
R3	11	0FFFFH	-1，字处理

通过下面的例子，看看模拟指令怎样利用常数发生器的。

```
CLR　DST　　；将 DST 单元清零
```

这不是内核指令，是一条模拟指令，汇编器将 As=00,R3=0，用

MOV　R3，DST

来模拟。

4　通用寄存器

R4-R15 为通用工作寄存器。MSP430 的通用寄存器是 430 活动的大部分场所，可以执行算术逻辑运算，也可以作为临时的暂存单元。可以字操作，也可以字节操作。比如：

MOV　　#1234H,R15　　执行后 R15 内容为 1234H
MOV.B　#23H,　R15　　执行后 R15 内容为 0023H
ADD.B　#34H,　R15　　执行后 R15 内容为 0057H

MSP430 指令的寻址方式包括立即寻址、索引寻址、符号寻址和绝对寻址。这四种寻址方式均可用于源操作数，而索引、符号和绝对寻址方式只可用于目的操作数。源操作数和目的操作数的指令集需占用代码存储器中的 1 到 3 个字。

5　MSP430 的存储器组织

MSP430 系列的存储空间采用“冯-诺依曼”结构，ROM、RAM 在同一地址空间，使用一组地址数据总线。而存储空间的组织又分大模式与小模式，在小模式时，总的寻址空间为 64K；大模式时，总的寻址空间为 1MB。小模式时采用线形寻址空间；大模式时代码可访问 16 个 64KB 的代码段，数据可访问的地址空间为 16 个 64KB 的页，即为分段分页方式。图 2.2 为总的存储空间示意图。

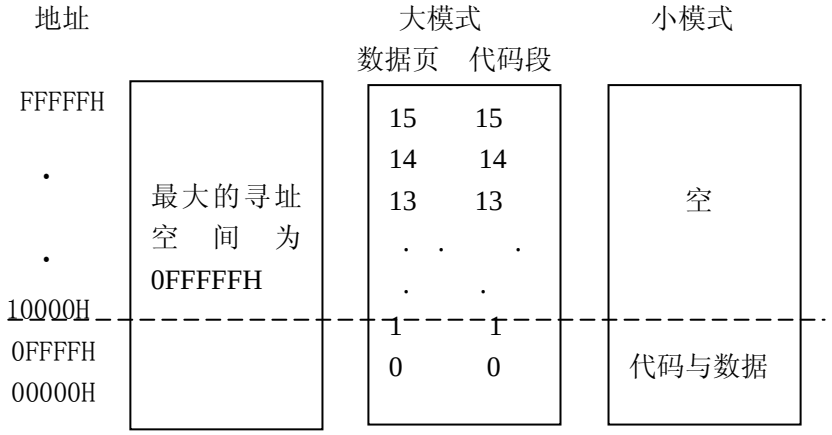


图 2.2　MSP430 存储器总的组织模式

当存储器组织为 64KB 或更少时采用小模式，地址空间为最低的 64KB，而目前的器件都设计成小模式，最大的存储空间组织为 60KB。在小模式中，所有的程序存储器、数据存储器、I/O 口、其它外围模块的控制器等等都安排在 64KB 空间中。现在我们只讨论 64K 存储空间的使用情况。

由于采用“冯-诺依曼”结构，ROM、RAM 在同一地址空间，从 0000H 到 0FFFFH 这一段范围内从低到高分别是：特殊功能寄存器、外围模块、数据存储器、程序存储器、中断向量表。根据具体的不同型号其存储器的具体组织不一样。在表 2.3 中列举了几个常用的 MSP430 器件的存储器组织结构。

表 2.3　常用 MSP430 器件的存储器组织结构

	MSP430F1121	MSP430F449	MSP430F123	MSP430C325
存储器大小	4K	60K	8K	16K

中断向量地址	0FFFFH-0FFE0H	0FFFFH-0FFE0H	0FFFFH-0FFE0H	0FFFFH-0FFE0H
代码存储器地址	0FFFFH-0F000H	01100H-0FFFFH	0E000H-0FFFFH	0C00H-0FFFFH
信息存储器大小	256	256	256	
信息存储器地址	0EF00H-0EFFFH	01000H-010FFH	01000H-10FFH	
引导存储器大小	1K	1K	1K	
引导存储器地址	0800H-0BFFH	0C00H-0FFFH	0C00H-1000H	
数据存储器大小	256	2K	256	512
RAM 地址	0200H-02FFH	0200H-09FFH	0200H-02FFH	0200H-03FFH
16 位外围模块地址	0100H-01FFH	0100H-01FFH	0100H-01FFH	0100H-01FFH
8 位外围模块地址	0010H-00FFH	0010H-00FFH	0010H-00FFH	0010H-00FFH
特殊功能寄存器地址	0000H-000FH	0000H-000FH	0000H-000FH	0000H-000FH

表 2.3 列举了几个常用的 MSP430 器件的存储器结构，我们可以看出它们有相同之处、有不同之处。

相同的地方在于：

所有器件的中断向量放在相同的地方：0FFE0H-0FFFFH；

所有器件的 8 位、16 位外围模块所占用相同范围的存储器地址；

所有器件的特殊功能寄存器所占用相同范围的存储器地址；

数据存储器开始于相同的地址，都是从 0200H 处开始；

代码存储器的最高地址都是 0FFFFH。

不同之处在于：

不同型号器件的代码存储器容量不一样，从它的型号参见第一章的命名规则可看出；

代码存储器的起址不一样，每一种器件的代码存储器的起始地址为：

起始地址 = 10000H - 该器件的代码容量；

信息存储器仅 FLASH 型的有，而且不同的器件地址也不一样，但容量都是 256 字节；

引导存储器仅 FLASH 型的有，而且不同的器件地址也不一样，但容量都是 1K 字节；

数据存储器的结束地址各器件也不一样，其结束地址为：

数据存储器的末地址 = 该器件数据 RAM 容量 + 0200H；

中断向量的具体内容因器件不同而不同；

所有器件的 8 位、16 位外围模块地址范围内的具体内容因器件不同而不同。

从表 2.3 也可看出 MSP430 系列器件在存储空间的全部范围安排了：ROM、RAM、以及外围模块的寻址地址等等。这些部件通过内部总线与 CPU 相连接，而且有的可以字/字节访问、有的只能字访问、有的只能字节操作，图 2.3 是片内的数据总线结。下面将予以分别讨论。

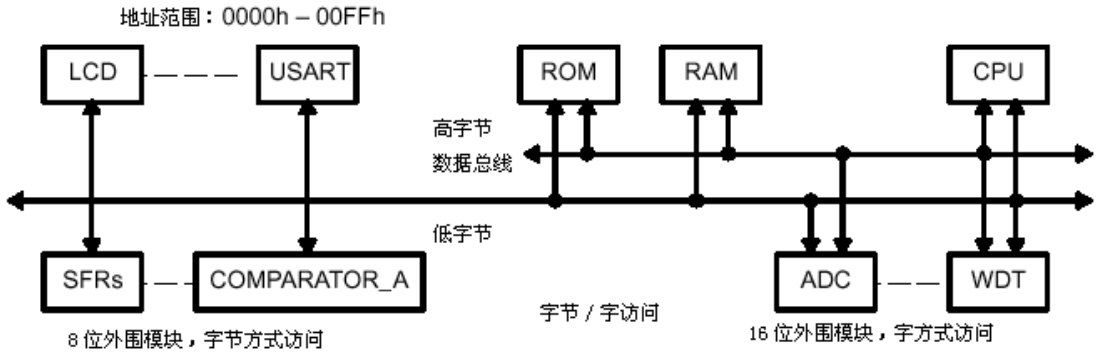


图 2.3　各种模块与总线的连接

6　数据存储器 RAM

MSP430 的数据存储器位于存储器地址空间的 0200H 以上，这些存储器一般用做数据的保存与堆栈的使用，同时也是数据运算的场所，在特殊场合还可以用做程序存储器使用。可以字操作、也可以字节操作，通过指令后缀加以区别。但用做程序存储器时只能字操作。字与字节操作情况参见图 2.4。

字（高字节）							
字（低字节）							
15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	10
字节							
字节							

图 2.4　存储器中的位、字节、字

在字节操作时，每 8 位为一个操作单位；在字操作时，每两个字节为一个操作对象，而且*对准偶地址*操作。比如：

MOV.B　#20H，&221H　　执行后地址 221H 内容为 20H
MOV.B　#324H,&221H　　执行后地址 221H 内容为 24H
MOV.W　#3234H,&222H　　执行后字节地址 222H 内容为 34H，地址 223H 内容为 32H
MOV.W　#324H,&221H　　执行后地址 221H 内容为 03H，地址 220H 内容为 24H

RAM 空间还可以进行运算，比如：

MOV.B　#33H,&220H　　执行后地址 220H 内容为 33H
ADD.B　#22H,&220H　　执行后地址 220H 内容为 55H
MOV.B　#11H,&221H　　执行后地址 221H 内容为 11H
ADD　#1234H,&220H　　执行后地址 220H 内容为 89H,执行后地址 221H 内容为 23H
RLA　&220H　　执行后地址 220H 内容为 12H, 执行后地址 221H 内容为 47H

FLASH 型的器件还有信息存储区，也可以用来做数据 RAM 使用，同时它是 FLASH 型，数据掉电后不丢失，可以保存重要参数。

7　程序存储器 ROM

程序 ROM 区为 0FFFFH 以下一定数量存储空间，可存放指令代码，可存放数据表格，程序代码必须偶地址寻址。而程序代码又可分三种情况：中断向量区、用户程序代码、系统引导程序（个别器件才有，比如 FLASH 型）。

中断向量区用来说明相应中断的中断服务程序首地址。比如某应用程序的中断向量区如图 2.5 所示。

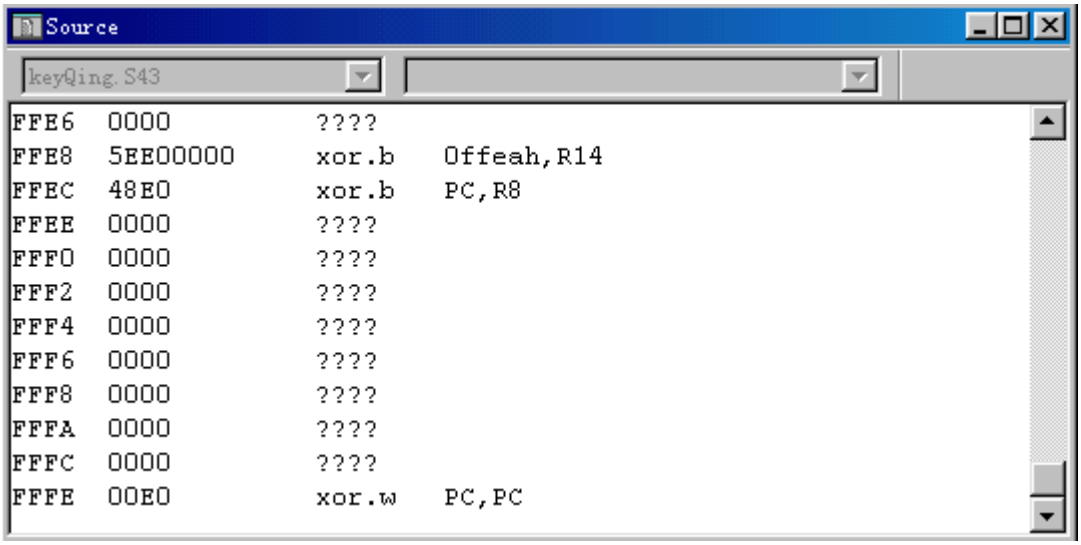


图 2.5　中断向量区

图 2.5 的内容究竟是什么含义呢，这是调试环境中的一个窗口。其中最左边一列 FFFE、FFFC、FFFA 等等表示地址，中间一列 00E0、0000、48E0、5EE00000 等等表示对应左边地址的相应内容，最右边 xor.w　pc,pc 等等表示汇编指令或相应地址中数据的反汇编指令。这里着重要理解的是第二栏数据：00E0、0000、0000、48E0、5EE00000 等等，这些数据就是我们所说的中断向量，它只表示相应中断的中断服务程序首地址。还要注意这些数据的格式，00E0 实质上是 4 位 16 进制数 0E000H，48E0 实质上是 0E048H，因为在地址栏（最左边）是从偶地址开始的，也就是 00 是 0FFFEH 地址的内容，E0H 是 0FFFFH 地址的内容。而 5EE00000 呢？这是因为汇编器在反汇编时将这些数据反汇编成一条指令而写在一起的。实质上它们只是 0E05EH 与 0000H 两个 4 位 16 进制数据。

图 2.5 也同时说明在这个应用程序中，用户用了三个中断，写了三个中断服务程序，它们的中断服务程序的首地址分别是：E000H—系统复位或非屏蔽中断的中断服务程序的开始地址；E048H—定时器 A 中断；E05E—P1 口中断。

不同的器件，中断向量含义不一样。表 2.4、表 2.5、表 2.6、表 2.7 为不同器件的中断向量表情况。

表 2.4　　MSP430X3XX、MSP430X11XX 中断向量表

中断源	中断标志	系统中断	地址	优先级
上电、外部复位、看门狗	WDTIFG	复位	0FFFEH	15，最高
NMI / 振荡器故障	NMIFG/OFIFG	非屏蔽 / 可屏蔽	0FFFCH	14

I/O 专用	P0IFG.0	可屏蔽	0FFFAH	13
I/O 专用	P0IFG.1	可屏蔽	0FFF8H	12
比较器 A		可屏蔽	0FFF6H	11
看门狗定时器	WDTIFG	可屏蔽	0FFF4H	10
定时器 A	CCIFG0	可屏蔽	0FFF2H	9
定时器 A	TAIFG	可屏蔽	0FFF0H	8
串口接收	URXIFG	可屏蔽	0FFEEH	7
串口发送	UTXIFG	可屏蔽	0FFECH	6
ADC	ADCIFG	可屏蔽	0FFEAH	5
定时器/端口		可屏蔽	0FFE8H	4
P2	P2IFG.0-7	可屏蔽	0FFE6H	3
P1	P1IFG.0-7	可屏蔽	0FFE4H	2
基本定时器	BTIFG	可屏蔽	0FFE2H	1
P0	P0IFG.2-7	可屏蔽	0FFE0H	0，最低

表 2.5　　MSP430X13X、MSP430X14X 中断向量表

中断源	中断标志	系统中断	地址	优先级
上电、外部复位、看门狗、FLASH 存储器	WDTIFG	复位	0FFFEH	15，最高
NMI 、振荡器故障、FLASH 访问出错	NMIFG、OFIFG、ACCVIFG	非屏蔽 / 可屏蔽	0FFFCH	14
定时器 B	BCCIFG0	可屏蔽	0FFFAH	13
定时器 B	BCCIFG1-6、TBIFG	可屏蔽	0FFF8H	12
比较器 A	CMPAIFG	可屏蔽	0FFF6H	11
看门狗定时器	WDTIFG	可屏蔽	0FFF4H	10
串口 0 接收	URXIFG0	可屏蔽	0FFF2H	9
串口 0 发送	UTXIFG0	可屏蔽	0FFF0H	8
ADC	ADCIFG	可屏蔽	0FFEEH	7
定时器 A	CCIFG0	可屏蔽	0FFECH	6
定时器 A	CCIFG1-2、TAIFG	可屏蔽	0FFFAH	5
P1	P1IFG.0-7	可屏蔽	0FFE8H	4
串口 1 接收	URXIFG1	可屏蔽	0FFE6H	3
串口 1 发送	URXIFG0	可屏蔽	0FFE4H	2
P2	P2IFG.0-7	可屏蔽	0FFE2H	1
			0FFE0H	0，最低

表 2.6　　MSP430F41X 中断向量表

中断源	中断标志	系统中断	地址	优先级
上电、外部复位、看门狗、FLASH	WDTIFG	复位	0FFFEH	15，最高

NMI 、振荡器故障、FLASH 访问出错	NMIFG、OFIFG、ACCVIFG	非屏蔽 / 可屏蔽	0FFFCH	14
			0FFFAH	13
			0FFF8H	12
比较器 A	CMPAIFG	可屏蔽	0FFF6H	11
看门狗定时器	WDTIFG	可屏蔽	0FFF4H	10
				9
				8
				7
定时器 A	CCIFG0	可屏蔽	0FFF2H	6
定时器 A	CCIFG1-2、TAIFG	可屏蔽	0FFFAH	5
P1	P1IFG.0-7	可屏蔽	0FFE4H	4
			0FFE6H	3
			0FFE4H	2
P2	P2IFG.0-7	可屏蔽	0FFE2H	1
基本定时器	BTIFG	可屏蔽	0FFE2H	0

表 2.7　　MSP430X43X、MSP430X44X 中断向量表

中断源	中断标志	系统中断	地址	优先级
上电、外部复位、看门狗、FLASH	WDTIFG	复位	0FFFEH	15，最高
NMI 、振荡器故障、FLASH 访问出错	NMIFG、OFIFG、ACCVIFG	非屏蔽 / 可屏蔽	0FFFCH	14
定时器 B	BCCIFG0	可屏蔽	0FFFAH	13
定时器 B	BCCIFG1-6、TBIFG	可屏蔽	0FFF8H	12
比较器 A	CMPAIFG	可屏蔽	0FFF6H	11
看门狗定时器	WDTIFG	可屏蔽	0FFF4H	10
串口 0 接收	URXIFG0	可屏蔽	0FFF2H	9
串口 0 发送	UTXIFG0	可屏蔽	0FFF0H	8
ADC	ADCIFG	可屏蔽	0FFEEH	7
定时器 A	CCIFG0	可屏蔽	0FFF2H	6
定时器 A	CCIFG1-2、TAIFG	可屏蔽	0FFFAH	5
P1	P1IFG.0-7	可屏蔽	0FFE4H	4
串口 1 接收	URXIFG1	可屏蔽	0FFE6H	3
串口 1 发送	URXIFG0	可屏蔽	0FFE4H	2
P2	P2IFG.0-7	可屏蔽	0FFE2H	1
基本定时器	BTIFG	可屏蔽	0FFE2H	0

注：非屏蔽中断不受中断控制位的控制，发生事件，就可产生中断。

程序 ROM 除了中断向量表而外的其它空间就可随意用做用户程序区。

对于 FLASH 型的器件还有 1K 字节的引导 ROM（自动加载程序），这是一段出厂时已经固化的程序。为闪速存储器的读、写、擦除等操作提供环境。

8　外围模块寄存器地址

从表 2.3 中可以看出 MSP430 的外围模块的寻址被安排在 0010H-01FFH 这一段区域。同时还分字寻址与字节寻址。

字模块是经全部 16 位总线相连的模块，位于存储空间 100H-1FFH。这部分空间又被分割成 16 个帧，每一帧 8 个字，一般每个字模块占用一到三个帧的地址空间。表 2.8 为 MSP430F449 的字模块的存储空间使用情况。

表 2.8　MSP430F449 字模块的空间分割

地址	说明	地址	说明
1F0H-1FFH	保留	170H-17FH	定时器 A
1E0H-1EFH	保留	160H-16FH	定时器 A
1D0H-1DFH	保留	150H-15FH	ADC12 转换
1C0H-1CFH	保留	140H-14FH	ADC12 转换
1B0H-1BFH	保留	130H-13FH	硬件乘法器
1A0H-1AFH	ADC12 控制与中断	120H-12FH	看门狗、FLASH 控制
190H-19FH	定时器 B	110H-11FH	保留
180H-18FH	定时器 B	100H-10FH	保留

字节模块是经总线的低 8 位相连的模块，只能以字节方式来访问，而用字方式对字节模块进行操作，读时高字节将产生不预期的结果，写时高字节被忽略。字节模块占用存储空间 0000H-00FFH，一般也 8 字分为一组，共 16 组，表 2.9 为 MSP430F449 字节模块地址分配。

表 2.9　MSP430F449 字节模块的地址分配

地址	说明	地址	说明
0F0H-0FFH	保留	070H-7FH	串口 1/串口 0
0E0H-0EFH	保留	60H-6FH	保留
0D0H-0DFH	保留	50H-5FH	比较器 A、系统时钟
0C0H-0CFH	保留	40H-4FH	基本定时器
0B0H-0BFH	保留	30H-3FH	端口 6/端口 5
0A0H-0AFH	液晶模块	20H-2FH	端口 2/端口 1
090H-09FH	液晶模块	10H-1FH	端口 3/端口 4
080H-08FH	ADC12 存储控制	00H-0FH	SFR

特殊功能寄存器 SFR 处于存储空间的最低位置，位于 0000H-000FH，16 个字节，只能字节方式访问。目前只用了最前面的 6 个字节：

0000H　IE1　中断允许 1
0001H　IE2　中断允许 2
0002H　IFG1　中断标志 1
0003H　IFG2　中断标志 2
0004H　ME1　模块允许 1
0005H　ME2　模块允许 2

其中各位的含义请见附录与后面各部分的讲解。

2.1．2　寻址模式

MSP430 有 7 种寻址方式，其中源操作数可用全部的 7 种方式寻址，而目的操作数只有 4 种方式寻址。但都可访问整个地址空间，由 As 与 Ad 模式位的内容确定，祥见表 2.10。对于任何有效的源与目的操作数的组合都可能构成一条合法的 MSP430 汇编语句。下面将详细讨论这 7 种寻址模式。

表 2.10　寻址模式

As / Ad	寻址模式	语法	说明
00/0	寄存器寻址	Rn	寄存器内容即为操作数
01/1	变址寻址	X(Rn)	（Rn+X）指向操作数，X 存于后续字中
01/1	符号寻址	ADDR	(PC+X)指向操作数，X 存于后续字中，使用了变址模式的 X(PC)
01/1	绝对寻址	&ADDR	指令后续字包含绝对地址
10/—	间接寄存器寻址	@ Rn	Rn 为指向操作数的指针
11/—	间接增量寻址	@ Rn+	Rn 为指向操作数的指针，取数之后 Rn 再加一
11/—	立即寻址	# N	指令后续字中包含立即数 N，使用了间接增量模式的 @PC+

1　寄存器寻址模式

汇编源程序

```
MOV     Rn  ,     Rm
```

这种寻址模式的操作数在寄存器中，可以是源操作数、可以是目的操作数、也可以既是源操作数又是目的操作数。下面的语句都属于寄存器寻址：

```
MOV     R10,R11          ；源与目的操作数均寄存器寻址
MOV     #345H,R10        ；目的操作数寄存器寻址
ADD     #2298H,R10       ；目的操作数寄存器寻址
MOV     R10,&220H        ；源操作数寄存器寻址
……
```

举例:　　MOV　R10,R11

解释:　　将寄存器 R10 内的数据取出,送达 R11

执行前		执行后		ROM 内容
R10	0A023H	R10	0A023H	MOV　R10,R11
R11	0FA0H	R11	0A023H	
PC	PCold	PC	Pcold+2	

2　变址寻址模式

汇编源程序

MOV[.B]　　X　（ Rn ），　Y （ Rm ）

这种寻址模式的操作数的地址为寄存器内容加上寄存器前的偏移量，此地址中的数据即为所寻址的操作数。可以是源操作数、可以是目的操作数、也可以既是源操作数又是目的操作数。下面的语句都属于变址寻址：

```
MOV          2  (R10),   R11       ；源操作数变址寻址
MOV          R11,   4  (R10)       ；目的操作数变址寻址
MOV          2  (R10), 3 (R12)       ；源与目的操作数都变址寻址
ADD          R12,   4  (R12)        ；目的操作数变址寻址
ADD          @R10,  4  (R12)       ；目的操作数变址寻址
……
```

举例:　　MOV　　2 (R10),6 (R11)

解释:　　将地址：寄存器 R10 内的数加 2 内的数取出,送达地址：R11 加 6

图 2.6 为语句执行之前的情况：

PC 指针　— 0E008H

R10　　— 0200H

R11　　— 0220H

RAM　　地址 200H 以后的内容见图

也可以看出这条指令占用 ROM 三字长：4A9B　0002　0006

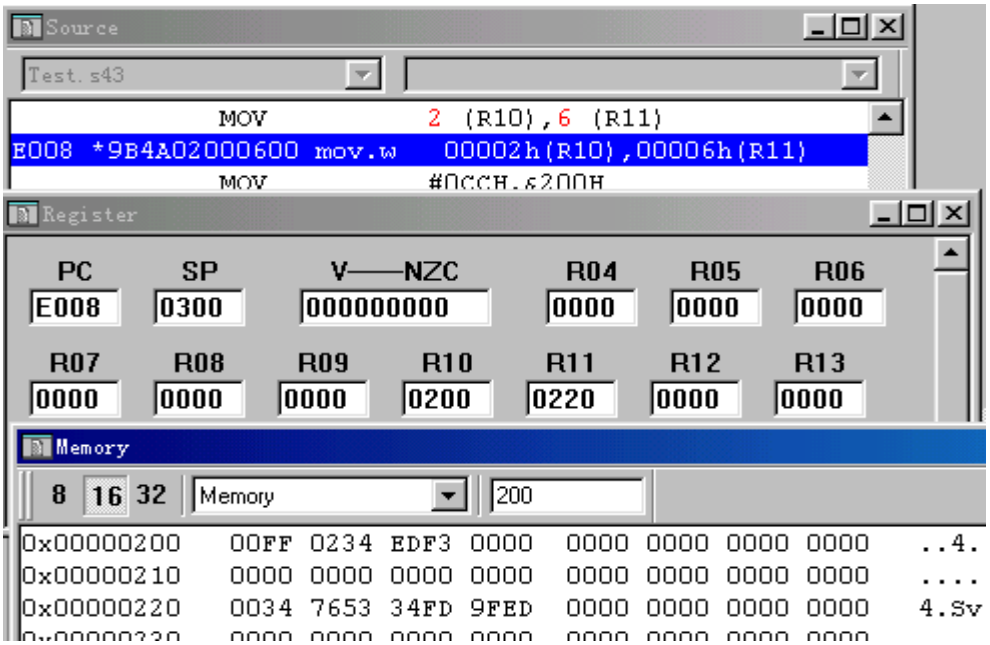


图 2.6　MOV　2 (R10),6 (R11) 执行之前

图 2.7 为语句执行之后的情况：

PC 指针　— 0E00EH

R10　　— 0200H

R11　　— 0220H

RAM 的内容发生了改变：字 226H 内的数据由 0FEDH 改成了 0234H，其余没变。

分析：变址寻址模式的数据地址为寄存器内容加上指令中寄存器前面的偏移量。

这里源与目的都是变址寻址，

源操作数为 ： R10+2 地址中的数据：RAM 中地址为 200+2H 的数据 0234H

目的地址为 ： R11+6=226H

也就是说指令 MOV　2 (R10),6 (R11)的意图在于将地址 202H 中的数 0234H 送达地址 0226H，而源操作数不变，寄存器中内容不变。

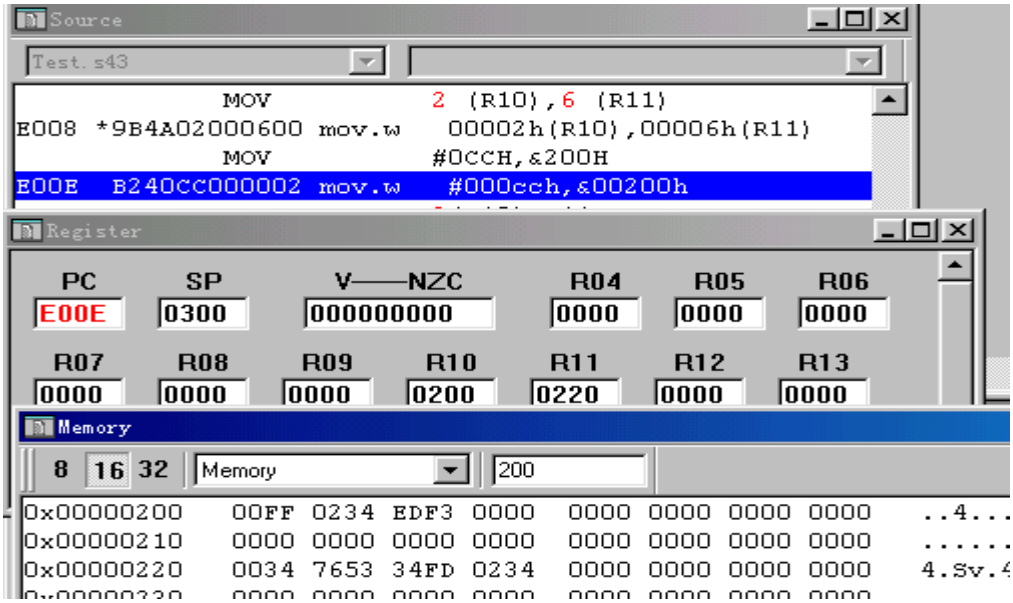


图 2.7　MOV　2 (R10),6 (R11) 执行之后

3　符号模式

汇编源程序

MOV　EDE ，　TONI

这种寻址模式的操作数的地址为 EDE、TONI。在指令代码中，紧跟操作码的数据为所寻地址与当前 PC（程序计数器）的差。这种寻址模式可用于源操作数、也可用于目的操作数、也可以既是源操作数又是目的操作数。下面的语句都属于符号寻址模式：

```
MOV . B     R6,   LOOP0      ；目的操作数符号寻址
MOV         TAB,  R5         ；源操作数符号寻址
ADD         TAB,  &220H      ；源操作数符号寻址
SUB.B       TAB,  &230H      ；源操作数符号寻址
    ……
```

举例：　MOV　TAB ，R5

……

TAB　DW　13F2H,2213H,3ED4H

解释：　此语句的目的在于将符号 TAB 所表示的数据作为地址，再将这个地址中的数据送达 R5。指令执行前后分别见图 2.8 与图 2.9

分析：　　执行之前，R5=43F2H

TAB 为一个地址标号，指示在 JMP　TTAW 语句之后的地址处

而 TAB 标号所指示的地址处的数据为字 13F2H

该语句的执行就送数 13F2H 到 R5

结果如图 2.9

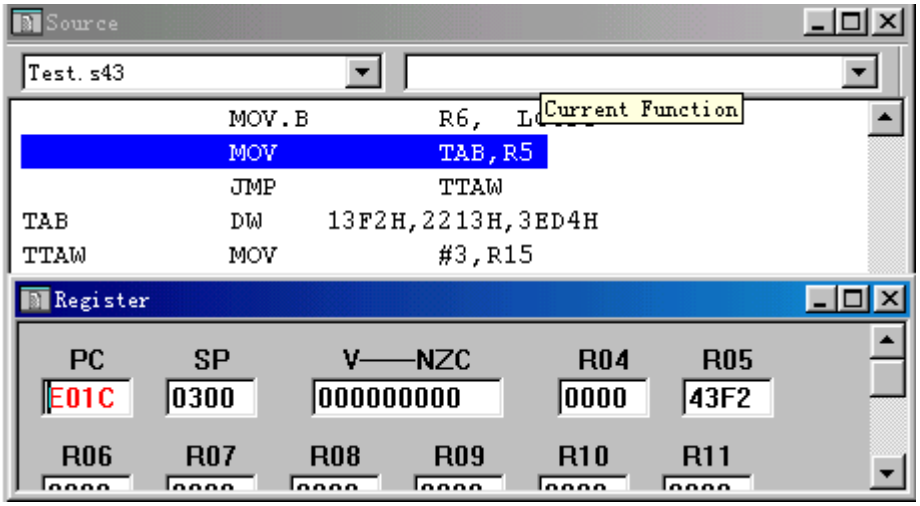


图 2.8　执行 MOV　TAB ， R5 指令之前

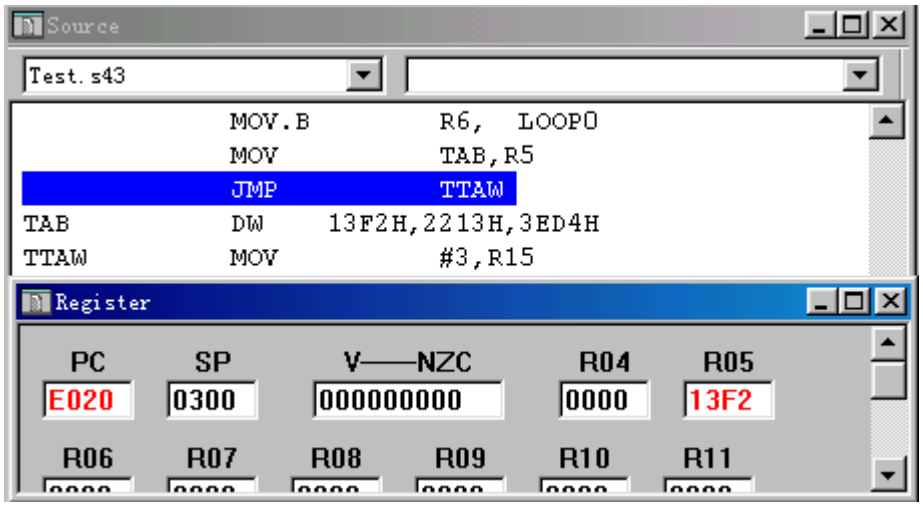


图 2.9　执行 MOV　TAB ， R5 指令之后

4　绝对寻址模式

汇编源程序

MOV　　&EDE ，　&TONI

这种寻址模式中的 EDE、TONI 即为操作数的地址。在指令代码中，紧跟操作码的一个或两个字就是操作数的地址。这种寻址模式可用于源操作数、也可用于目的操作数、也可以既是源操作数又是目的操作数。下面的语句都属于绝对寻址模式：

MOV　　#2345H，　&RESETT　　；目的操作数绝对寻址

MOV　　&RES　，　R15　　；　源操作数绝对寻址

```
MOV      &220H  ,  R15          ;  源操作数绝对寻址
MOV.B    R5,       &200H        ;  目的操作数绝对寻址
  ……
```

举例：

```
Reset          MOV  #2345H, R6
AAA            MOV  R6, R7
               SUB  &AAA , &Reset
```

解释：最后一句将地址 Rese t 中的数据减去地址 AAA 中的数据，再将结果送达地址 Rese t ，指令 SUB &AAA , &Reset 执行前后分别见图 2.10 与图 2.11

分析：指令执行之前，从图 2.10 可以看出

AAA=0E004H

Reset=0E000H

而地址 0E000H 中的数据为 4036H

地址 0E004H 中的数据为 4607H

所以指令执行的实质就是 4036H—4607H = FA2FH （C=1）

再将结果 FA2FH 送达 &Reset ：地址 0E000H

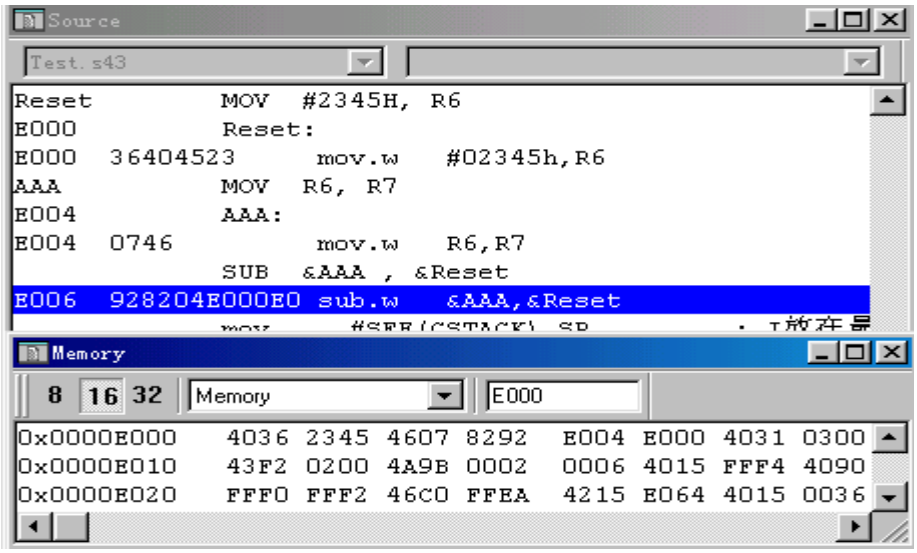


图 2.10　　指令 SUB &AAA , &Reset 执行前

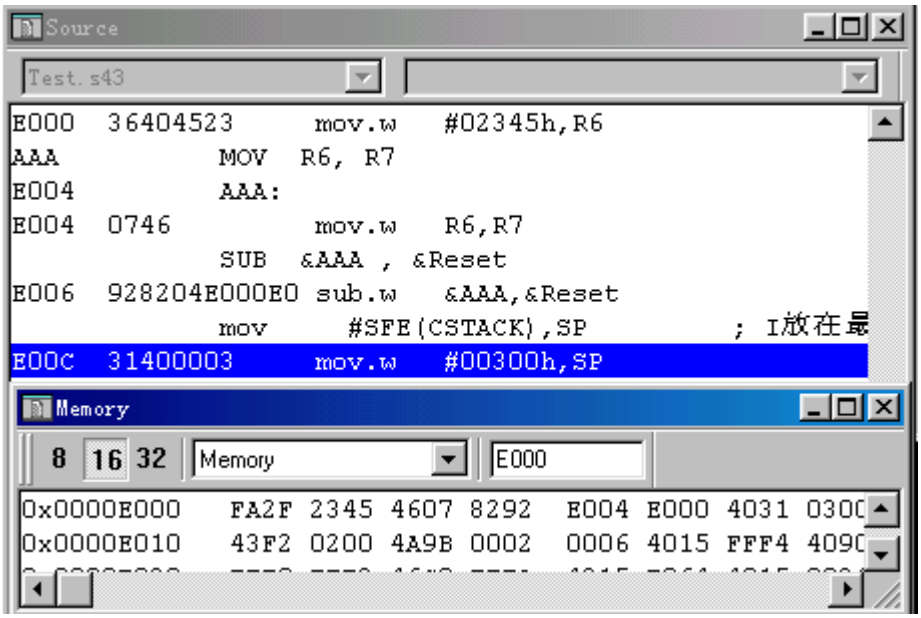


图 2.11　　指令 SUB　&AAA ， &Reset 执行后

5　间接寻址模式

汇编源程序

MOV　　@ R10 ， 2 (R11)

这种寻址模式将地址放在寄存器中，即寄存器中数据为所寻址数据的地址，而寄存器中的数据不改变。这种模式只对源操作数有效，对于目的操作数只能用变址寻址模式 0（Rd）替代。以下指令都是间接寻址模式：

MOV　　@R5 ,R6

MOV　　@R5, 2(R6)

ADD　　@R5 ,&220H

MOV　　@R5, &AAA

SUB.B　　@R6 ,4 (R7)

……

举例：　SUB.B　　@R4 ,4 (R5)

解释：　以 R5 中数据加 4 为地址的数据减以 R4 为地址的数据，再将结果送达以 R5 中数据加 4 的地址，指令执行前后分别见图 2.12 与图 2.13

分析：　执行指令之前，R4 = 220H

R5 = 230H

&220H = 0E34FH

目的操作数地址 R5+4 = 234H

目的操作数内容 &234H = 0

指令的执行：字节减法， 0—4FH = 0B1H （C=1，有借位）

指令执行结果， &234H = 0B1H ， 其余（寄存器与其余 RAM 单元）不变

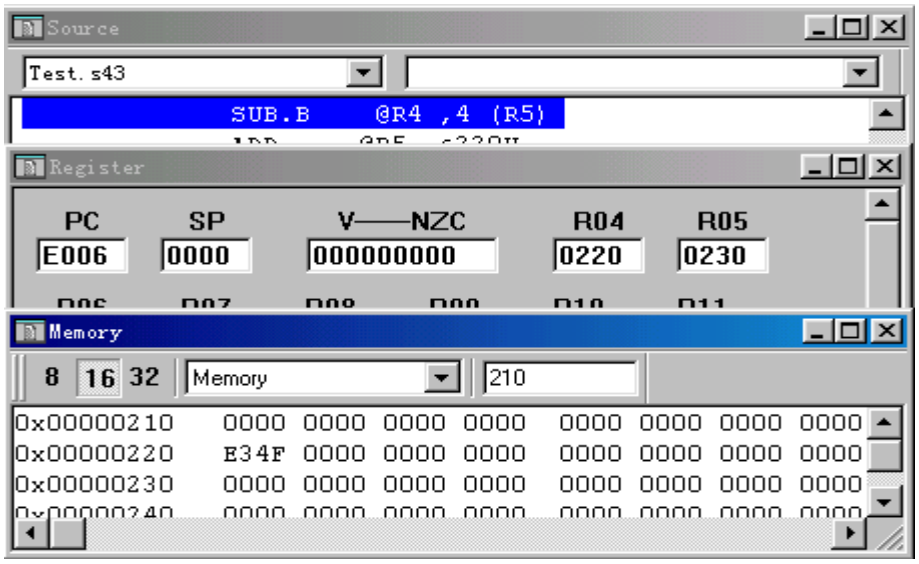


图 2.12　指令 SUB.B　　@R4 ,4 (R5)执行前

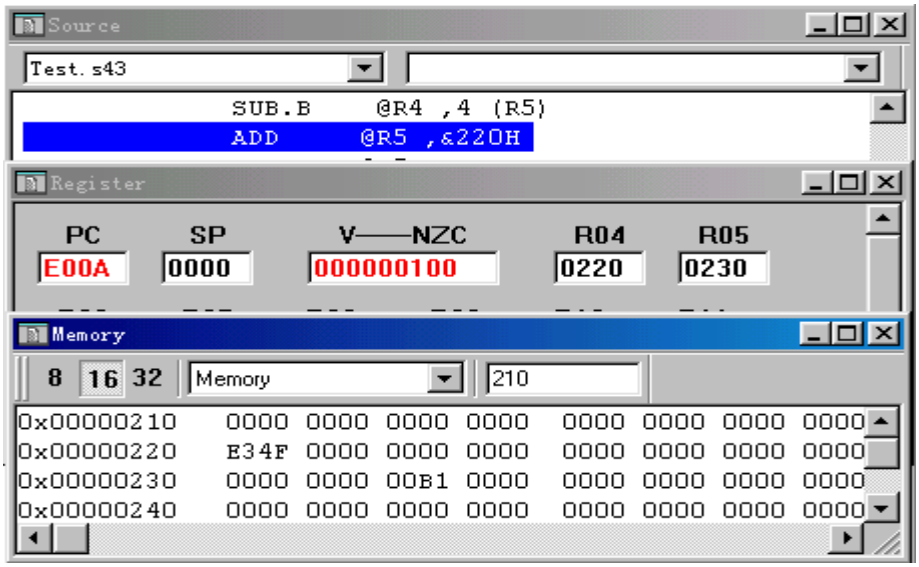


图 2.13　指令 SUB.B　　@R4 ,4 (R5)执行后

6　间接增量寻址模式

汇编源程序

```
MOV　@ R10 + , 0 ( R 11 )
```

这种寻址模式将地址放在寄存器中，即寄存器中数据为所寻址数据的地址，而源寄存器中的数据增加 1（字节操作）或 2（字操作）。这种模式只对源操作数有效，对于目的操作数只能用变址寻址模式 0（Rd），同时 INC / INCD　Rd（手动改变目的操作数指针）替代。以下指令都是间接寻址模式：

```
MOV　　@R5 +,R6
MOV　　@R5+, 2(R6)
ADD　　@R5 +,&220H
```

```
MOV      @R5+, &AAA
SUB.B    @R6 +,4 (R7)
……
```

举例：　SUB　@R4+ ,4 (R5)

解释：　以 R5 中数据加 4 为地址的数据减以 R4 为地址的数据，再将结果送达以 R5 中数据加 4 的地址，同时 R4 自动指向下一数据(R4+2→R4)，指令执行前后分别见图 2.14 与图 2.15

分析：　执行指令之前，R4 = 220H

R5 = 230H

&220H = 0B23DH

目的操作数地址 R5+4 = 234H

目的操作数内容 &234H = DEF2H

指令的执行：字减法，0DEF2H—0B23DH = 2CB5H (C=0，没有借位)，R4 加 2

指令执行结果，&234H = 2CB5H，R4 = 222H

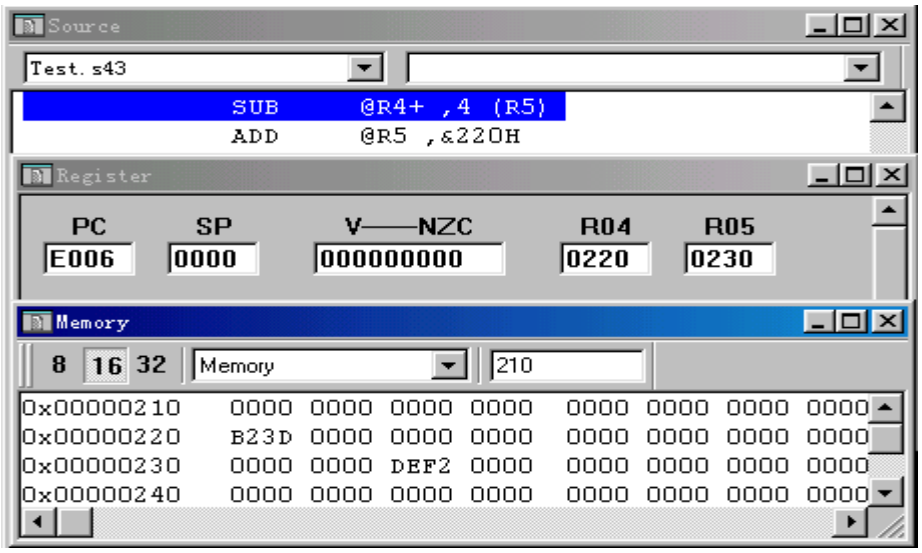


图 2.14　指令 SUB　@R4+ ,4 (R5)执行前

间接增量寻址模式常用在需大量数据搬动或查表等等情况。在传送了第一个数之后，数据指针 Rn 自动增加，指向下一个数，可以为程序设计带来很多方便，而且指针 Rn 的增加加一还是加二根据执行的是字操作还是字节操作而定。

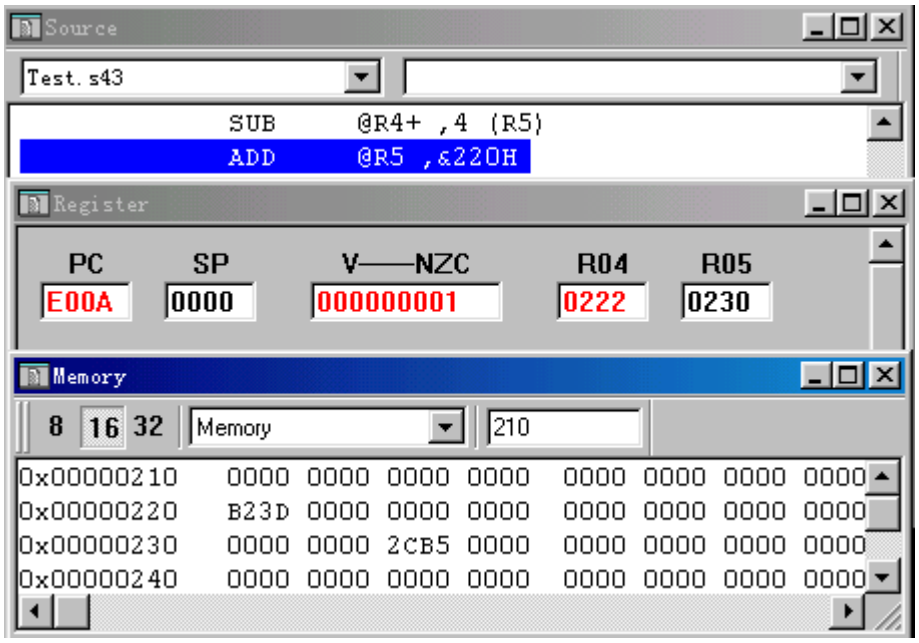


图 2.15　指令 SUB　　@R4+ ,4 (R5)执行后

7　　立即寻址模式

汇编源程序

MOV　#2345H , TONI

这种寻址模式的操作数能够立即得到，在指令代码中紧跟操作码后。下面列举的指令都使用了立即寻址模式：

MOV　　#1234H , R6

MOV　　#1234H , 2(R6)

ADD　　#1234H ,&220H

MOV　　#1234H , AAA

SUB.B　　#12H , &220H

举例：MOV　　#1234H , R6

解释：将立即数 1234H 送达 R6

执行结果：R6 = 1234H

2.1.3　指令格式

1　指令书写格式

MSP430 汇编源文件是由汇编程序指令组成的 ASCⅡ字符文件。是由一条一条的指令语句组成的。每一条语句使用如下格式：

[标号]　（伪）指令助记符　[操作数 1]，[操作数 2]　[；注释]

标号汇编器将翻译成该行语句的物理地址，书写时对齐最左边，不必用冒号。

指令助记符指明该语句将执行的操作。

操作数 1、操作数 2 指明该语句的操作数，如果有两个操作数，则操作数 1 为源操作数，操作数 2 为目的操作数，两操作数之间用逗号间隔。如果只有一个操作数，

则为目的操作数或既为目的操作数又为源操作数。

注释为设计者为该语句写的解释，之前用分号间隔。

比如：

```
START    MOV    #2345H，R15    ；R15 设置初始值
LOOP     DEC    R15            ；R15 减一
         JNZ    LOOP           ；R15 没有减到 0 就继续减一，直到减完
```

指令书写中的常用符号：

Rn	R0—R15　16 个寄存器一般指 R4-R15
#	后面的数为立即数
&	后面的数据为具体的地址
@	后面数据中的内容为最终寻址地址
+	内容增加
—	内容减少
.W / . B	字操作 / 字节操作
dst	目的操作数
src	源操作数
PC / R0	程序计数器
SP / R1	堆栈指针
TOS	堆栈顶
C	进位位
N	负位
V	溢出位
Z	零位
MSB	最高有效位
LSB	最低有效位

以上说的是 MSP430 汇编指令的书写格式，可以看出与其它汇编指令的书写格式有些不一样，下面我们来看看 MSP430 的指令代码格式。而指令代码格式又因操作数多少而不一样。

2　双操作数指令（内核指令）

该指令格式使用双操作数，由 4 个域组成，共有 16 位代码：

- 操作码域——4 位　　[操作码]
- 源域　　——6 位　　[源寄存器+As]
- 字节操作识别符—1 位　　[B/W]
- 目的域——5 位　　[目的寄存器+Ad]

源域由 2 个寻址位和 4 位寄存器（R0-R15）组成；而目的域由 1 个寻址位和 4 位寄存器数（R0-R15）组成。

字节识别符 B/W 表明指令是以一个字节（B/W=1）还是以一个字（B/W=0） 的形式执行。

15　12	11　8	7	6	5　4	3　0

操 作 码	源寄存码	Ad	B/W	As	目的寄存码
操作码域					

下面一些指令为双操作数指令。

			状	态	位	
			V	N	Z	C
ADD[.W];ADD.B	src,dst	src+dst→dst	*	*	*	*
ADDC[.W];ADDC.B	src,dst	src+dst+C→dst	*	*	*	*
AND[.W];AND.B	src,dst	src.and.dst→dst	0	*	*	*
BIC[.W];BIC.B	src,dst	.not.src.and.dst → dst	-	-	-	-
BIS[.W];BIS.B	src,dst	src.or.dst→dst	-	-	-	-
BIT[.W];BIT.B	src,dst	src.and.dst	0	*	*	*
CMP[.W];CMP.B	src,dst	dst-src	*	*	*	*
DADD[.W];DADD.B	src,dst	src+dst+C→dst(十进制)	*	*	*	*
MOV[.W];MOV.B	src,dst	src→dst	-	-	-	-
SUB[.W];SUB.B	src,dst	dst+.not.src+1→dst	*	*	*	*
SUBC[.W];SUBC.B	src,dst	dst+.not.src+C→dst	*	*	*	*
XOR[.W];XOR.B	src,dst	src.xor.dst→dst	*	*	*	*

注意：将状态寄存器 SR 用于目的操作数的操作会用其结果覆盖 SR 的内容；但是在以下的操 作中状态位不受影响。

例如：ADD　　#3，SR　　；　操作：　（SR）+3→SR

3　单操作数指令（内核指令）

该指令格式使用单操作数，由 2 个域组成，共 16 位：

- 操作码域——9 位且 4 个 MSB 为“1h”
- 字节操作识别符——1 位　　[B/W]
- 目的域——6 位　　[目的寄存器+Ad]

目的域由 2 个寻址位和 4 位寄存器数（R0-R15）组成。目的域位的位置与 2 个操作数指令的位置相同。

字节识别符 B/W 表明指令是以一个字节（B/W=1）还是以一个字（B/W=0） 的形式执行。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	×	×	×	×	×	B/W	Ad		目	的	寄	存	器	
操　作　码　域									目　的　域								

下面一些指令为单操作数指令。

			状态位			
			V	N	Z	C
RRA[.W];RRA.B	dst	MSB→MSB→……LSB→C	0	*	*	*
RRC[.W];RRC.B	dst	C→MSB……LSB→C	*	*	*	*
PUSH[.W];PUSH.B	src	SP-2→SP,src→@SP	-	-	-	-
SWAP	dst	交换字节	-	-	-	-
CALL	dst	PC+2→@SP,dst→SP	-	-	-	-
TETI		从中断返回	*	*	*	*
		TOS→SR,SP+2→SP				
		TOS→PC,SP+2→SZP				
SXT	dst	位 7→位 8→……位 15	0	*	*	*

4　条件和无条件转移（内核指令）

（无）条件转移的指令格式包括个域，共 16 位：

- 操作码域——6 位
- 转移偏移域——10 位

操作码由 OP-Code（3 位）和下列条件决定的 3 位组成。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	1	×	×	×	×	×	×	×	×	×	×	×	×
操作码			转移码			符号	偏移							
操作码域						转移/偏移域								

条件转移使指令可转移到相对于当前地址范围在-511~+512 字之间的地址。汇编器计算出有符号的偏移，并将它们插入操作码。

JC/JHS	标号	进位位被置时转移到标号语句
JEQ/JZ	标号	零位被置时转移到标号语句
JGE	标号	（N.XOR.V）=0 时转移到标号语句
JL	标号	（N.XOR.V）=1 时转移到标号语句
JMP	标号	无条件转移到标号语句
JN	标号	负位被置时转移到标号语句
JNC/JLO	标号	进位位复位时转移到标号语句
JNE/JNZ	标号	零位复位时转移到标号语句

注意：条件转移和无条件转移不影响状态位

——已经发生的转移会用偏移改变 PC 值：PCnew=PCold+2 +2*偏移；

——还未发生的转移则在上次指令处继续执行程序。

5　无需 ROM 补偿的仿真指令

下列指令可以无需额外的 ROM 而被精简指令集仿真。汇编器接收仿真指令的助记符并

插入适当的内核指令操作码。

助记符		说　明	状态位 V	N	Z	C	仿　真
ADC[.W]	dst	进位位加至目的操作数	*	*	*	*	ADDC #0,dst
ADC.B	dst	进位位加至目的操作数	*	*	*	*	ADDC.B#0,dst
DADC[.W]	dst	十进制进位位加至目的操作数	*	*	*	*	DADD #0,dst
DADC.B	dst	十进制进位位加至目的操作数	*	*	*	*	DADD.B #0,dst
DEC[.W]	dst	目的操作数减 1	*	*	*	*	SUB #1,dst
DEC.B	dst	目的操作数减 1	*	*	*	*	SUB.B #1,dst
DECD[.W]	dst	目的操作数减 2	*	*	*	*	SUB #2,dst
DECD.B	dst	目的操作数减 2	*	*	*	*	SUB.B #2,dst
INC[.W]	dst	目的操作数增 1	*	*	*	*	ADD #1,dst
INC.B	dst	目的操作数增 1	*	*	*	*	ADD.B #1,dst
INCD[.W]	dst	目的操作数增 2	*	*	*	*	ADD #2,dst
INCD.B	dst	目的操作数增 2	*	*	*	*	ADD.B #2,dst
SBC[.W]	dst	从目的操作数中减进位位	*	*	*	*	SUBC #0,dst
SBC.B	dst	从目的操作数中减进位位	*	*	*	*	SUBC.B #0,dst
逻辑指令							
INV[.W]	dst	目的操作数求反	*	*	*	*	XOR #0FFFFh,dst
INV.B	dst	目的操作数求反	*	*	*	*	XOR.B#0FFFFh,dst
RLA[.W]	dst	算术左移	*	*	*	*	ADD dst,dst
RLA.B	dst	算术左移	*	*	*	*	ADD.B dst,dst
RLC[.W]	dst	通过进位左移	*	*	*	*	ADDC dst,dst
RLC.B	dst	通过进位左移	*	*	*	*	ADDC.B dst,dst
数据指令(共用)							
CLR[.W]		清除目的操作数	-	-	-	-	MOV #0,dst
CLR.B		清除目的操作数	-	-	-	-	MOV.B #0,dst
CLRC		清除进位位	-	-	-	0	BIC #1,SR
CLRN		清除负位	-	0	-	-	BIC #4,SR
CLRZ		清除零位	-	-	0	-	BIC #2,SR
POP	dst	项目从堆栈弹出	-	-	-	-	MOV @SP+,dst
SETC		置进位位	-	-	-	1	BIS #1,SR
SETN		置负位	-	1	-	-	BIS #4,SR
SETZ		置零位	-	-	1	-	BIS #2,SR
TST[.W]	dst	测试目的操作数	0	*	*	1	CMP #0,dst
TST.B	dst	测试目的操作数	0	*	*	1	CMP.B #0,dst
程序流指令							
BR	dst	转移到……	-	-	-	-	MOV dst,PC
DINT		禁止中断	-	-	-	-	BIC #8,SR
EINT		使能中断	-	-	-	-	BIS #8,SR

NOP	空操作	- - - -	MOV #0h,#0h
RET	从子程序返回	- - - -	MOV @SP+,PC

注意：以上指令的仿真可以由 R2 和 R3 的内容仿真

——寄存器 R2（CG1）包含立即数 2 和 4；

——寄存器 R3（CG2）包含-1 或 0FFFh、0、+1 和+2，这取决于寻址位 As。

——汇编器根据所用的立即数设置寻址位。

6　指令集表

表 2.11　　MSP430 指令集速查表

助记符	操作数	解释	V N Z C
*ADC [.W]; ADC.B	dst	dst + C ->dst	* * * *
ADD [.W]; ADD.B	src,dst	src +dst ->dst	* * * *
ADDC[.W]; ADDC.B	src,dst	src + dst +C->dst	* * * *
AND[.W]; AND.B	src,dst	src.and.dst->dst	0 * * *
BIC[.W]; BIC.B	src,dst	.not.src.and.dst->dst	- - - -
BIS[.W]; BIS.B	src,dst	src.or.dst->dst	- - - -
BIT[.W]; BIT.B	src,dst	src.and.dst->dst	0 * * *
*BR	dst	转移到	- - - -
CALL	dst	PC+2-> 堆栈，dst->PC	- - - -
*CLR[.W]; CLR.B	dst	清除目的操作数	- - - -
*CLRC		清除进位位	- - - 0
*CLRN		清除负位	- 0 - -
*CLRZ		清除零位	- - 0 -
CMP[.W]; CMP.B	src,dst	dst-src	* * * *
*DADC[.W]; DADC.B	dst	dst+C->dst(十进制)	* * * *
DADD[.W]; DADD.B	src,dst	src+dst+C->dst (十进制)	* * * *
*DEC[.W]; DEC.B	dst	dst-1 ->dst	* * * *
*DECD[.W]; DECD.B	dst	dst-2 ->dst	* * * *
*DINT		禁止中断	- - - -
*EINT		使能中断	- - - -
*INC[.W]; INC.B	dst	dst+1->dst	* * * *
*INCD[.W]; INCD.B	dst	dst+2->dst	* * * *
*INV[.W]; INV.B	dst	目的操作数求反	* * * *
JC/JHS	标号	进位位被置时转移到标号语句	- - - -
JGE	标号	(N.XOR.V)=0 时转移到标号语句	- - - -
JL	标号	(N.XOR.V)=1 时转移到标号语句	- - - -
JMP	标号	无条件转移到标号语句	- - - -
JN	标号	负位被置时转移到标号语句	- - - -
JNC/JLO	标号	进位位复位时转移到标号语句	- - - -

JNE/JNZ	标号	零位复位时转移到标号语句	- - - -
MOV[.W]; MOV.B	src,dst	src->dst	- - - -
*NOP		空操作	- - - -
*POP[.W]; POP.B	dst	项目从堆栈弹出，SP+2→SP	- - - -
PUSH[.W]; PUSH.B	src	SP-2→SP,src→@SP	- - - -
RETI		从中断返回 TOS→SR,SP+2→SP TOS→PC,SP+2→SZP	* * * *
*RET		从子程序返回 TOS→PC,SP+2>SP	- - - -
*RLA[.W]; RLA.B	dst	算术左移	* * * *
*RLC[.W]; RLC.B	dst	通过进位左移	* * * *
RRA[.W]; RRA.B	dst	MSB→MSB→...LSB→C	0 * * *
RRC[.W]; RRC.B	dst	C→MSB→.....LSB→C	* * * *
*SBC[.W]; SBC.B	dst	从目的操作数中减去进位	* * * *
*SETC		置进位位	- - - 1
*SETN		置负位	- 1 - -
*SETZ		置零位	- - 1 -
SUB[.W]; SUB.B	src,dst	dst+.not.src +1 →dst	* * * *
SUBC[.W]; SUBC.B	src,dst	dst+.not.src +C →dst	* * * *
SWAP	dst	交换字节	- - - -
SXT	dst	位 7-.位 8→位 9→...→位 15	0 * * *
*TST[.W]; TST.B	dst	测试目的操作数	0 * * 1
XOR[.W]; XOR.B	src,dst	src.xor.dst→dst	* * * *

注意：带有*标志的指令是仿真指令

1. 所有带有标志（*）的指令是仿真指令。仿真指令的使用结合了 CPU 的结构和执行方法的内核指令，使得代码效率更高和速度更快。
2. “.and.”、“.or.”、“.not.”和“.xor.”分别表示逻辑“与”、“或”、“非”和“异或”操作。
3. “→”表示“写内容到”。
4. “src”和“dst”分别表示源操作数和目的操作数。
5. 状态位中“*”表示影响，“-”表示不影响，“0”和“1”表示清零和置位。

7　MSP430 指令的时钟周期与指令长度

MSP430 的指令执行速度（指令所用的时钟周期数，这里时钟周期指 MCLK 的周期）和指令长度（所占用存储器空间）与指令的格式和寻址模式密切相关。在不同的寻址模式下，CPU 寻找操作数的路径不一样，当然要花不同的时间与占用不同的存储空间。表 2.12、表 2.13 示出了 MSP430 指令的时钟周期数（单位为“1 个 MCLK 周期”）与指令长度（单位为“字”）。

表 2.12　双操作数指令的时钟周期数与指令长度表

寻址模式		周期数	指令长度（字）	实例
As	Ad			
00, Rn	0, Rm	1	1	MOV　R5,R15
	0, PC	2	1	BR　R5
00, Rn	1,x(Rm)	4	2	ADD　R5,3(R15)
	1, EDE		2	XOR　R5,EDE
	1, &EDE		2	MOV　R5,&EDE
01, x(Rn)	0, Rm	3	2	MOV　2(R5),R15
01, EDE			2	AND　EDE, R15
01, &EDE				MOV　&EDE,R15
01, x(Rn)	1, x(Rm)	6	3	ADD　3(R5),3(R15)
01, EDE	01,TONI		3	CMP　EDE , TONI
01, &EDE	1,&TONI		3	ADD　EDE ,&TONI
10, @Rn	0, Rm	2	1	AND　@R5,R15
10, @Rn	1,x(Rm)	5	2	XOR　@R5,3(R15)
	1, EDE		2	MOV　@R5,EDE
	1,&EDE		2	XOR　@R5,&EDE
11,@Rn+	0, Rm	2	1	ADD　@R5+,R15
	0, PC	3	1	BR　@R5+
11, #N	0, Rm	2	2	MOV　#2380H,R5
	0, PC	3	2	BR　#2AEH
11, @Rn+	1, x(Rm)	5	2	MOV　@R5+,2(R15)
11, #N	1, EDE		3	ADD　#3245H,EDE
11, @Rn+	1, @EDE		2	MOV　@R5+,&EDE
11, #N			3	ADD　#2345H,&EDE

表 2.13　单操作数指令的时钟周期数与指令长度表

寻址模式 A(s/d)	周期数		指令长度	实例
	RRA RRC SWPB SXT	PUSH / CALL		
00,Rn	1	3/4	1	SWPB　R5
01,x(Rn)	4	5	2	CALL　2(R5)
01,EDE	4	5	2	PUSH　EDE
01,&EDE				SXT　&EDE
10,@Rn	3	4	1	RRC　@R5
11,@Rn+	3	4/5	1	SWPB　@R5+
11,#N			2	CALL　#2344H

对于跳转类指令它们的时钟周期数与指令长度是固定的。

Jxx（任意条件跳转指令)无论跳转与否都要花费 2 个时钟周期，指令长度都是 1 个字长。

中断返回指令要花费 5 个时钟周期，指令长度都是 1 个字长。

除了 CPU 的指令，CPU 还有一些操作也将花时间，但不占用存储空间（因为不是程序指令）。它们是以下一些操作：

中断响应　　花费 6 个时钟周期；

WDT 复位　　花费 4 个时钟周期；

系统复位　　花费 4 个时钟周期。

2.1.4　汇编语言程序设计

汇编语言程序设计基本上是汇编指令的堆砌，但这并不是简单的堆砌。大家都说高级语言才是模块化的设计语言，同样在使用汇编语言进行程序设计时，也要讲究使用模块化的结构。在这一部分，将讲述与汇编语言程序设计相关的问题。

1　汇编伪指令

在 2.5 小节讲述了 MSP430 的指令系统，在进行汇编程序设计时，它们是程序的主体。但是还有一些伪指令，它们提供程序数据并控制汇编过程，也是必不可少的。一般地汇编器伪指令能帮助用户完成以下的事情：

将代码与数据汇编到规定的段中；

在存储器中用未初始化的变量保留空间；

控制汇编后列表文件的格式；

初始化存储器；

汇编条件块；

定义全局变量；

规定汇编器可以从中获得宏的库；

产生符号化的调试信息。

常用的汇编伪指令有以下一些：

模块控制伪指令：NAME、MODULE、ENDMOD 等；

段控制伪指令：ASEG 、RSEG 、STACK、COMMON、ORG、ALIGN、EVEN 等；

数值分派伪指令：SET、EQU（=）、DEFINE、sfrb、sfrw 等；

数据定义与分配伪指令：DB、DW、DL、DF、DS 等；

下面分别予以介绍。

2　模块控制伪指令

模块控制伪指令标志一个源程序模块的开始与结束，并给模块命名与指示其类型。使用的助记符为：NAME、MODULE、ENDMOD、END 等。

其中 NAME (PROGRAM)标志一个程序模块的开始；

MODULE (LIBRARY)表示一个库模块的开始；

ENDMOD 表示当前汇编模块的结束；

END 表示一个汇编文件的最后模块的结束。

使用格式如下：

```
NAME symbol [(expr)]
MODULE symbol [(expr)]
ENDMOD [label]
END [label]
```

在下面的例子里定义了 3 个模块。

```
MODULE
    Module #1
    ENDMOD
    MODULE
     Module #2
    ENDMOD
    MODULE
     Last module
END
```

3　段控制伪指令

段控制伪指令说明了代码与数据是怎样生成的。使用如下一些助记符：

ASEG	一个绝对段的开始；
RSEG	一个可重定位段（相对段）的开始；
STACK	定义堆栈段；
COMMON	定义公共段；
ORG	设置特定的定位指针；
ALIGN	通过插入一些填充字节用以校准程序计数器；
EVEN	通过插入一些填充字节使程序计数器对准偶地址；

使用语法如下：

```
ASEG [start [(align)]]
```

```
RSEG segment [:type] [(align)]
STACK segment [:type] [(align)]
COMMON segment [:type] [(align)]
ORG expr
ALIGN align [,code]
EVEN
```

举例说明：

下面的程序段将定位 子程序 subr 开始在 123 以后的地址空间

```
  7     00007B               subr     ASEG     123(8)
  8     000100 34400A00               MOV      #10,R4
  9     000104 0485                   SUB      R5,R4
 10     000106 3041                   RET
 11     000108
 12     000108                        END      main
```

下面的程序段将定义 100 字节的可重定位段作为堆栈使用。数据先使用高地址空间，后使用低地址空间。

```
        STACK   rpnstack
parms   DS      100
opers   DS      100
        END
```

下面定义了两个公共段用于存放变量。

```
        NAME    common1
        COMMON  data
count   DS        4
        ENDMOD

        NAME    common2
        COMMON  data
up      DS        1
        ORG     $+2
down    DS        1

        END
```

下面的程序段使用 ALIGN 伪指令能确保子程序开始在正确的地址。

```
  1     000000                        NAME     align
```

```
 2    000000
 3    000000 34400700      main    MOV     #7,R4
 4    000004 9012FA00              CALL    subr
 5    000008 0445                  MOV     R5,R4
 6    00000A
 7    00000A 000000000000          ALIGN   8
 8    000100 35400A00      subr    MOV     #10,R5
 9    000104 0554                  ADD     R4,R5
10    000106 3041                  RET
11    000108

12    000108                       END     main
```

下面的程序段使用 EVEN 伪指令确保数据表格开始在偶地址。

```
 1    000000                       NAME    even
 2    000000
 3    000000 01                    DB      1
 4    000001 00                    EVEN
 5    000002 01000A006400          DW      1,10,100
 6    000008                       END
```

4　数据分配伪指令

这类指令有以下一些：

SET (VAR, ASSIGN)	赋予一个临时值；
EQU (=)	在当前模块中赋予一个永久的值；
DEFINE	定义一个整个文件中都有效的值；
sfrb	寄存器类型的字节；
sfrw	寄存器类型的字。

使用语法如下：

```
label SET expr
label EQU expr
label = expr
label DEFINE expr
[const] sfrb register = value
[const] sfrw register = value
```

其中，

label	定义一个标志符、
expr	标志符的值、

register 特殊功能寄存器、
value　特殊功能寄存器的值。

在下面的例子中使用了局部变量与全局变量，在模块 add1 中定义了符号 value ，同样在模块 add2 中也定义了符号 value，但它们表示两个不同的量，都只在各自的模块内部有效，这是局部变量。而在模块 add1 中定义的 locn 则为全局变量，在两个模块中表示同一个值。

```
        NAME    add1
locn    DEFINE  100H
value   EQU     77
        MOV     locn,R4
        ADD     #value,R4

        ENDMOD

        NAME    add2
value   EQU     88
        MOV     locn,R5
        ADD     #value,R5

        END
```

5　数据定义伪指令

指令格式如下：
DB　定义一个 8 位常数；
DW　定义一个 16 位常数；
DL　定义一个 32 位常数；
DF　定义一个 32 位浮点常数；
DS　分配 n 个字节单元。

使用语法：

```
DB  expr[,expr]
DW  expr[,expr]
DL  expr[,expr]
DF  expr[,expr]
DS  expr
```

下面一条语句在地址 table 处预留了 10 个字节空间。

```
    table   DS  0xA
```

下面的语句段在地址为 F_TAB 处定义了一些 16 位常数。

```
F_TAB           DW    0EEFH,   0D4EH,    0BDAH
                DW    0B30H,   09F8H,    08E1H,   07E9H
```

我们经常用的数码管显示程序中有一个显示段码表，实质上也是使用 DB 伪指令定义的一些常数表，如下所示。

```
TABLED:   DB  3FH , 06H ,  5BH   ,4FH        ;0   1   2   3
          DB  66H , 6DH ,  7DH ,   07H        ;4   5   6   7
          DB  7FH , 6FH ,  77H,    7CH        ;8   9   A   B
          DB  39H , 5EH,   79H  ,  71H        ;C   D   E   F
```

6　常用汇编程序设计方法

在前面讲述了机器指令与汇编伪指令，它们分别是机器能执行的指令与汇编器能使用的指令。程序设计实质上就是按照一定的原则、按照一定的思路、方法等将这些指令组织起来，让 CPU 按设计者的思想执行指令，用以实现一定的功能，最终解决我们要解决的问题。那么使用什么样的方法组织指令、或怎样编写程序呢，使用模块化的程序结构！

程序最终要完成设计者的任务。将这些任务划分为一些子任务：任务 1，任务 2……，那么相应地使用程序模块 1，模块 2……来完成任务 1，任务 2……。

接下来的事情就是编写相应的模块以完成相应的子任务。也就是具体的程序编写。用汇编语言编写程序大体上可以分为 3 个步骤：

（1）　确定算法，画出流程图；

（2）　确定数据，包括工作单元的数量，分配存放单元等；

（3）　使用机器指令与汇编伪指令按流程图进行程序的编写。

程序编写一般的原则是：节省数据单元，缩短程序长度，加快运行时间。都知道单片机的资源有限，特别是 RAM 与 ROM，MSP430 最少的 RAM 与 ROM 尺寸分别是 128 字节与 1K 字节，最多也就是 2K 字节 RAM，60K 字节 ROM。而运行时间的多少将直接影响到程序的效能。

下面介绍常用结构的汇编设计方法。

A　顺序结构

所谓顺序结构就是 CPU 执行完了一条指令再执行下一条指令，程序计数器（PC）的内容每次增加当前指令的字节数。比如下面的程序。

```
MOV    #2345H,R5
MOV    R5,R8
MOV    @R8,&220H
……
```

B　散转结构

在进行键盘操作时，相应的按键实现相应的功能。那么为什么按“1”号键就能实现“1”号键所指定的功能呢。很简单，因为按了“1”号按键之后，程序跳转到了能实现“1”号按键功能的程序段执行了。散转结构的流程图如图 2.17，是一个死循环的结构，一般情况下可

用这种结构作为主环，不断地查询键盘的按键值，再根据键值执行相应的程序。

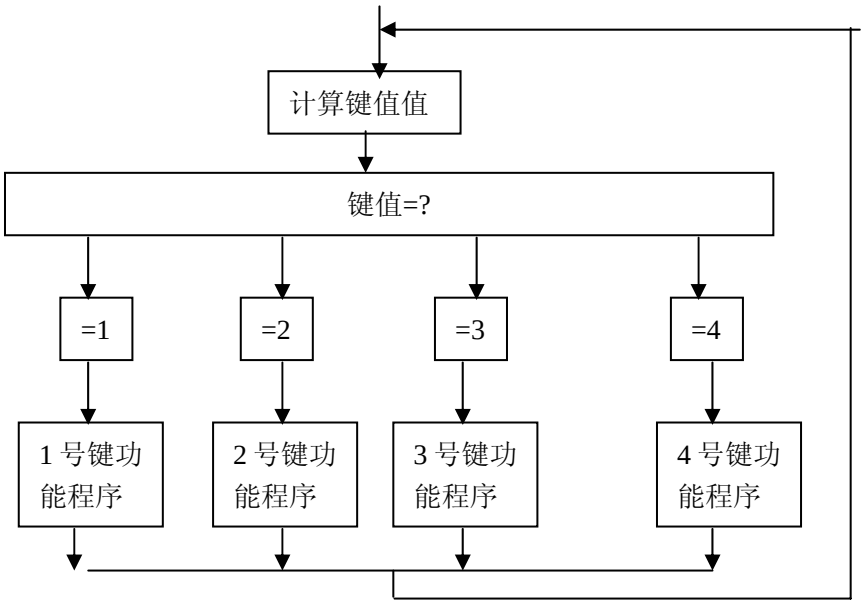


图 2.17　散转结构流程图

下面的程序段使用 BR 指令实现按键值散转的功能。按键值保存在 R9 中，JMP0、JMP1、JMP2……为每一个功能程序段的开始地址，将写在跳转表 JMPTAB 中。

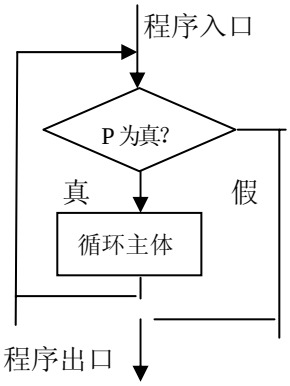
```
        ……
MAIN    CALL  KEY
        BR    JMPTAB(R9)
        ……
JMPTAB  DW    JMP0
        DW    JMP1
        DW    JMP2
        DW    JMP3
        ……
JMP0    实现按键 0 功能的程序段
        ……
        JMP   MAIN
JMP1    实现按键 1 功能的程序段
        ……
        JMP   MAIN
JMP2    实现按键 2 功能的程序段
        ……
        JMP   MAIN
JMP3    实现按键 3 功能的程序段
        ……
        JMP   MAIN
```

……

其中每一个功能程序段都要用 JMP　MAIN 作为结束，用以构成一个主循环。

C　循环结构

循环结构在程序设计中也同样占相当重要的地位。循环结构的使用可以使得程序量缩小，代码空间节省。常用的循环结构有两种形式，其流程分别如下所示。



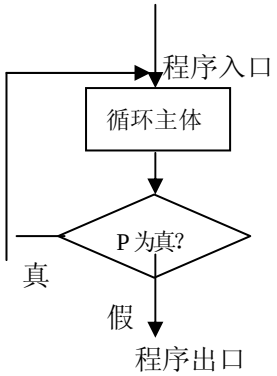


图 2.18“WHILE”循环结构　　图 2.19　“DO WHILE”循环结构

图 2.18 与图 2.19 所表示的两种循环结构的差别在于前者是先判断再执行，后者是先执行再判断。所以在使用的时候一定要注意使用的是哪种结构，因为这涉及到与一个循环的几个要素相关的参数。

循环结构的要素：循环变量初值，循环条件是否满足，循环主体，循环变量的改变。

下面要将 200H 开始的 20 字数据搬移到 260H 为开始的地址空间，使用循环结构。使用 R5 为循环变量，初值为 0，循环主体为数据移动，循环条件为 R5≠20，循环变量的改变为 R5 增 1。程序与流图如下：

```
        MOV   R5，#0                ；变量初值
LOOP    MOV   200H（R5），260H（R5）；循环主体
        INC   R5                    ；循环变量的改变
        CMP   #20，R5               ；循环条件的判断
        JNZ   LOOP                  ；满足条件则循环
        ……                          ；不满足条件则退出
```

常用的软件延时程序也是一个典型的循环结构，下面的程序将延时 60000 次，花费时间为 60000*3 个时钟周期，因为减一与判断跳转为 3 个时钟周期。

```
        MOV     #60000,R15
LOOP    DEC     R15
        JNZ     LOOP
```

D　选择结构

选择结构首先对一个条件语句进行测试，当条件为真时，执行一个方向上的流程；当条件为假时执行另一个方向上的流程。如图 2.20 所示。

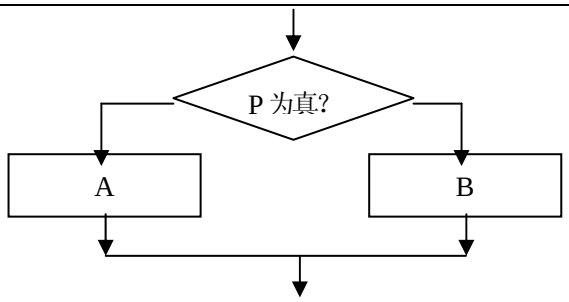


图 2.20　　选择结构流程图

这种结构在程序设计时同样也是很重要的。在汇编语言中，常使用 JXX　XXXX 指令来实现。下面举例说明。

在温控系统中，当表示温度的数据大于或等于 50 时，由 P1.0 引脚输出 0，关闭加热设备；当表示温度的数据小于 50 时，由 P1.0 输出 1，控制加热设备加热。当然这只是一个简单的“开关算法”，不会有多少精度，只是说明选择结构的使用。表示温度的数据存放在 R15 中，那么 R15 是否小于 50 是判断条件。

```
        BIS.B   #1,P1DIR
        CLRC
        SUB.B   #50，R15
        JC      KAI
        BIC.B   #1，&P1OUT
        JMP     COMM
KAI     BIS.B   #1,&P1OUT
COMM    ……
        ……
```

2.1.5　C 语言程序设计基础

MSP430 程序设计除了可使用汇编语言，使用 C 语言更可提高你的设计效率，加快你的开发进度。这里简要介绍如何使用 C 语言进行 MSP430 的程序设计。

1　　MSP430　C 语言的数据类型

MSP430 C 编译器支持的数据类型见下表（表 2.13）。

表 2.13　　MSP430 的 C 语言数据类型

数据类型	所占字节数	数据表示范围	注释
sfrb, sfrw	1		声明字节或字长度的 I/O 类型
char（默认类型）	1	0—255	等价于 unsigned char
char（使用 -c 选项）	1	-128—127	等价于 signed char

signed char	1	-128—127	
unsigned char	1	0—255	
short, int	2	-32768—32767	
unsigned short unsigned int	2	0—65535	
long	4	12147483648— 2147483647	
unsigned long	4	0—4294967295	
pointer	2		指针类型
float	4	18E-38—39E+38	浮点类型
double, long double	4	18E-38—39E+38	

说明与举例：

sfrw 用于定义地址范围为 0x100 to 0x1FF 的片内外围模块的功能寄存器。看门狗控制寄存器的地址为 120H，则：

```
sfrw    WDTCTL = 0x120;
void func(void)
{
  WDTCTL = 0x5A08;
}
```

sfrb 用于定义地址范围为 0x00 to 0xFF 的片内外围模块的功能寄存器以及特殊功能寄存器。P1 口的输出寄存器的地址为 11H，输入寄存器为 10H，则：

```
sfrb P0OUT = 0x11;
sfrb P0IN = 0x10;
void func()
{
  P0OUT = 4;     /* P0.2=1 */
  P0OUT |= 4;
  P0OUT &= ~8
  if (P0IN & 2) printf("ON");
}
```

MSP430 的 C 语言指针类型分为代码指针与数据指针。都占两个字节长度，都能指向 0000H 到 0FFFFH 范围的内存单元。

浮点数为标准的 IEEE 浮点数格式，占 4 字节：

31	30　　23	22　　0

符号位 S	指数部分 EXPONENT	尾数部分 MANTISSA

所表示的数值为：

$$(-1)^{S} * 2^{(EXPONENT-127)} * 1.MANTISSA$$

2　表达式语句（结构）

C 语言是一种结构化的程序设计语言。程序的最基本元素是表达式语句。所有的语句由“;”隔开，或者说每一条语句后面都有一个“;”。比如下面的一些语句：

```
X=a+b;
Z=(x+y)-1;   I++;
……
```

除了一些运算类的表达式语句外，C 语言还提供了十分丰富的程序控制语句。程序控制语句对于实现特定的算法显得相当重要。下面讲述常用的程序控制语句。

A　条件语句

条件语句又叫做分支语句，使用关键词 if 构成，表达条件选择的含义：如果怎样…，就怎样…，或否则就怎样…。语句表达形式有 3 种(a、b、c)：

```
a       if(条件表达式) 语句
b       if(条件表达式) 语句 1
        else   语句 2
c       if(条件表达式) 语句 1
            else  if(条件表达式) 语句 2
                else  if(条件表达式) 语句 3
                ……
```

下面的程序段将在 x 与 y 中选出较大的数。

```
Char  max(char x,char y)
   {
   if(x<y)
   {return(y);}
   else {return(x);}
   }
```

B　开关语句

开关语句是一种实现多方向条件分支的语句。虽然可用条件语句嵌套实现，但如果用开关语句则可使得程序条理分明，可读性强。开关语句由关键词 switch 构成，一般形式如下：

```
switch(表达式)
    {
    case  常量表达式 1:语句 1
                        break;
    case   常量表达式 2:语句 2
```

```
                    break;
    case   常量表达式 3:语句 3
                    break;
           ……
    default:        语句 d
    }
```

开关语句的执行过程是：将 switch 后面的表达式的值与 case 后面的常量表达式逐个比较，当遇到相等时则执行 case 后面的语句，break 语句的功能是终止当前语句的执行，使得程序能跳出 switch 语句。如果没有相等的情况，则执行语句 d。

在键盘程序中常使用开关语句。

```
Switch(key())
    case  0:key0();
            break ;
    case  1:key1();
            break ;
    case  2:key2();
            break ;
    case  3:key3();
            break ;
    ……
```

先调用键盘子程序得到按键值，再对按键值进行比较，再执行相应的按键程序。

C　　循环语句

循环语句给需要进行反复多次的操作提供了方便。在 C 语言中提供了 4 种循环控制语句。它们的构成形式为：

a　while(条件表达式) 语句;

当条件满足时，就反复执行后面的语句，一直执行到条件不满足时。以软件延时程序为例说明该语句是如何执行的。

```
void delay(long v)
        {
        while(v!=0)v--;
        }
```

该程序段使用 while 语句，先判断 v 的值是否为“0”，当不为“0”时执行其后的语句，当 v=0 时，退出循环。在循环体中同样也要条件以使得能够退出循环。

b　do 语句　while（条件表达式）

先执行一次循环体的语句，再判断条件是否满足，以决定是否再执行循环体。下面的程序将数组 BUFF[20]中的全部数据相加。

```
Int  x=0;
```

```
Char I=0;
Do{
    x =BUFF[I] + x;
    I=I+1;
    }
while(I<20);
……
```

c　for([初值设定表达式]；[循环条件表达式]；[条件更新表达式])语句

for 语句常用于需固定循环次数的循环。下面的程序段同样实现将数组 BUFF[20]中的全部数据相加的功能。

```
Int  x=0;
Char I=0;
for(I=0;I<20;I++)
    x =BUFF[I] + x;
……
```

d　goto　语句标号

goto 语句常用于跳转到一个固定的地址标号。其中固定的地址标号是一个带“：”的标志符。比如：

```
……
MM：……
……
goto  MMM
……
```

D　　返回语句

return(表达式);

该语句主要用于函数的返回参数。“表达式”为返回值。

3　　函数的定义与调用

在 C 语言中函数是基本模块，一个 C 语言程序是由若干个函数(至少一个，是主函数)构成的。但只有一个主函数 main()，同时 C 程序都是由主函数 main()开始，它是程序的起点。使用函数可大大提高编程效率。函数有两种：编译系统提供的标准库函数，用户自定义函数。标准库函数可直接调用，而用户自定义函数需自己编写或定义之后才能调用。函数定义的一般形式为：

```
函数类型　函数名（形式参数表）
    形式参数说明
    {
```

```
局部变量定义
函数体语句
}
```

其中，函数类型说明了自定义函数返回值的类型；

函数名是自定义函数的名字；

形式参数表中列出了在主调用函数与被调用函数之间传递数据的形式参数，形式参数的类型必须加以说明，主调函数可以是主函数，也可以是其他函数；

局部变量定义将定义在函数内部使用的局部变量；

函数体语句是为了完成该函数功能而写的各种语句的总合。

上面是一般函数的定义，在 MSP430 系统中还经常使用中断函数，中断函数的定义在形式上有些不一样，下面是中断函数定义的格式：

```
[存储变量类型] interrupt [中断矢量变量] 函数类型 函数名（形式参数表）
形式参数说明
    {
    局部变量定义
    函数体语句
    }
```

式中 interrupt 说明了该函数是中断服务函数；

[中断矢量变量]说明了该中断服务函数的在中断向量表中的中断地址；

其它与一般函数的定义相同。

下面的函数是经常使用的延时函数。

```
void delay(long v)
    {
    while(v!=0)v--;
    }
```

其中 void 定义该函数没有返回参数，v 是由调用函数传递进来的形式参数。

下面的函数计算了一个整数的正整数次幂。

```
int  mizhi(char  x,  char n)
    {
    int i,p;
    p=1;
    for(i =1; i <=n;++i)
    p=p*x;
    return(p);
    }
```

其中第一个 int 定义了整个函数将返回一个整数类型的值，这个值将传递给调用函数。X

与 n 为调用函数传递过来的形式参数。

那么中断服务函数的定义就要注意其它的一些问题：主函数的设置要能使得满足中断条件时响应中断。否则中断函数的编写毫无意义！下面是一个利用定时器中断实现在 P1.0 端口输出方波的完整程序。

```
#include <msp430x11x1.h>
void main(void)
{
  TACTL = TASSEL1 + TACLR;              // 设置定时器 A
  CCTL0 = CCIE;                         // CCR0 中断使能
  CCR0 = 20000;
  P1DIR |= 0x01;                        // P1.0 为输出口
  TACTL |= MC0;                         // 以增计数模式开始 Timer_a
  _EINT();                              // 总的中断使能
   for (;;)
  {
    _BIS_SR(CPUOFF);                    // 关 CPU
    _NOP();                             //
  }
}

interrupt[TIMERA0_VECTOR] void Timer_A (void)// 定义定时器 A 中断函数
{
  P1OUT ^= 0x01;                        // P1.0 求反
}
```

在这个程序中，主函数就是设置了能够使得定时器 A 进入中断的一些参数，然后休眠。主要来看看中断服务函数：

interrupt 表明是一个中断服务函数；

TIMERA0_VECTOR 声明了该函数的入口地址，在 msp430x11x1.h 文件中可以找到对 TIMERA0_VECTOR 的说明：#define TIMERA0_VECTOR　(9 * 2) /* 0xFFF2 Timer A CC0 */；也就是说中断入口地址为 0xFFE0+18。

在 C 语言中函数须先声明或定义再调用。为了保险起见，建议读者最好在程序的开始先对将要用到的函数进行声明。如果调用了一个没有声明或定义的函数，将会导致编译报错，同样如果先调用，再定义函数也会编译报错。

4　MSP430 C 语言标准库函数

MSP430 C 语言编译环境提供了大量的标准库函数。要使用这些标准库函数，非常简单，

只要在程序的开始声明要使用的库函数所在的头文件，之后在程序中就可以直接调用了。头文件的声明使用　#include　"****.h"语法即可。

常用的有以下一些头文件：这里对其中的函数只作一个简要介绍，详细使用情况可以查看软件中的帮助文件。

1　ctype.h　　字符处理类

isalnum	int isalnum(int c)	字母还是数字
isalpha	int isalpha(int c)	是否字母
iscntrl	int iscntrl(int c)	是否控制码
isdigit	int isdigit(int c)	是否数字
isgraph	int isgraph(int c)	是否为可打印的非空字符
islower	int islower(int c)	是否小写字母
isprint	int isprint(int c)	是否为可打印字符
ispunct	int ispunct(int c)	是否为表示标点符号的字符
isspace	int isspace(int c)	是否为空白字符
isupper	int isupper(int c)	是否为大写字符
isxdigit	int isxdigit(int c)	是否为 16 进制数
tolower	int tolower(int c)	转换为小写字符
toupper	int toupper(int c)	转换为大写字符

2　math.h　　数学类

acos	double acos(double arg)	反余弦函数
asin	double asin(double arg)	反正弦函数
atan	double atan(double arg)	反正切函数
atan2	double atan2(double arg1,double arg2)	带象限的反正切函数
ceil	double ceil(double arg)	大于或等于 arg 的最小正整数
cos	double cos(double arg)	余弦函数
cosh	double cosh(double arg)	双余弦函数
exp	double exp(double arg)	指数函数
fabs	double fabs(double arg)	双精度的浮点绝对值
floor	double floor(double arg)	小于或等于 arg 的最大正整数
fmod	double fmod(double arg1,double arg2)	浮点数的余数
frexp	double frexp(double arg1, int *arg2)	将浮点数分为两部分
ldexp(double arg1, int arg2)		乘以 2 的幂
log	double log(double arg)	自然对数函数
log10	double log10(double arg)	以 10 为底的对数函数
modf	double modf(double value, double *iptr)	拆开为整数部分与小数部分
pow	double pow(double arg1, double arg2)	求幂函数
sin	double sin(double arg)	正弦函数

```
sinh    double sinh(double arg)          双曲正弦
sqrt    double sqrt(double arg)          平方根函数
tan double tan(double x)                 正切函数
tanh    double tanh(double arg)          双曲正切函数
```

3 setjmp.h 非局部跳转

```
longjmp void longjmp(jmp_buf env, int val)  长跳转
setjmp  int setjmp(jmp_buf env)             设置返回点跳转
```

4 stdio.h 输入与输出类函数

```
getchar int getchar(void)                获得字符
gets    char *gets(char *s)              读字符串
printf  int printf(const char *format, ...) 写格式化数据
putchar int putchar(int value)           写字符函数
puts    int puts(const char *s)          写字符串函数
scanf   int scanf(const char *format, ...)读格式化数据
sprintf int sprintf(char *s, const char *format,)   将格式化数据写入字符串
sscanf  int sscanf(const char *s,const char *format,...)从字符串中读取格式化数据
```

5 stdlib.h 通用子程序类

```
abort   void abort(void)                 非正常结束程序
abs int abs(int j)                       绝对值函数
atof    double atof(const char *nptr)    转换 ASCII 为双精度
atoi    int atoi(const char *nptr)       转换 ASCII 为整数
atol    long atol(const char *nptr)      转换 ASCII 为长整形
bsearch void *bsearch(const void         在数组中搜索
            *key, const void *base,
            size_t nmemb, size_t size,
            int (*compare) (const void
            *_key, const void *_base));
calloc  void *calloc(size_t nelem, size_t elsize) 为目标数组分配存储器单元
div div_t div(int numer, int denom)      除法运算函数
exit    void exit(int status)            结束程序
free    void free(void *ptr)             释放存储器单元
labs    long int labs(long int j)        整形数取绝对值
ldiv    ldiv_t ldiv(long int numer, long int denom)     长整形除法
malloc  void *malloc(size_t size)        分配存储器
qsort   void qsort(const void *base,     数组排序
```

```
        size_t nmemb, size_t size,
        int (*compare) (const void
        *_key, const void *_base));
rand    int rand(void)                  随机数生成函数
realloc void *realloc(void *ptr, size_t size)重新分配存储器单元函数

srand   void srand(unsigned int seed)   设置随机数的种子
strtod  double strtod(const char        将字符串转换为双精度数
        *nptr, char **endptr)
strtol  long int strtol(const char      将字符串转换为长整形数
        *nptr, char **endptr, int
        ase)
strtoul unsigned long int strtoul       将字符串转换为无符号整形数
        const char *nptr, char
        **endptr, base int)
```

6 string.h 字符串处理类

```
memchr  void *memchr(const void *s,     在存储器中搜索字符诶
        int c, size_t n)
memcmp  int memcmp(const void *s1,      比较存储器内容
        const void *s2, size_t n)
memcpy  void *memcpy(void *s1,          拷贝存储器内容
        const void *s2, size_t n)
memmove void *memmove(void *s1,         移动存储器内容
        const void *s2, size_t n)
memset  void *memset(void *s,           置存储器
        int c, size_t n)
strcat  char *strcat(char *s1,          逻辑字符串
        const char *s2)
strchr  char *strchr(const char *s, int c)在字符串中找某一个字符
strcmp  int strcmp(const char *s1,      比较两个字符串
        const char *s2)
strcoll int strcoll(const char *s1,     比较字符串
        const char *s2)
strcpy  char *strcpy(char *s1, const    拷贝字符串
        char *s2)
strcspn size_t strcspn(const char       在字符串中跨过被排除的字符
        *s1, const char *s2)
strerror    char *strerror(int errnum)  给出一个错误信息字符串
strlen  size_t strlen(const char *s)    计算字符串长度函数
```

strncat	char *strncat(char *s1, const char *s2, size_t n)	将指定数量的字符与字符串连接起来
strncmp	int strncmp(const char *s1, const char *s2, size_t n)	将指定数量的字符与字符串相比较
strncpy	char *strncpy(char *s1, const char *s2, size_t n)	在字符串中复制指定的字符
strpbrk	char *strpbrk(const char *s1, const char *s2)	在字符串中寻找任何指定的字符
strrchr	char *strrchr(const char *s, int c)	从字符串的右端开始寻找字符
strspn	size_t strspn(const char *s1, const char *s2)	在字符串中统计与分析字符
strstr	char *strstr(const char *s1, const char *s2)	在字符串中搜索子字符串
strtok	char *strtok(char *s1, const char *s2)	将标志前的字符剪掉
strxfrm	size_t strxfrm(char *s1, const char *s2, size_t n)	转换字符串并返回其长度

5　C 语言编程实例

下面的程序将在 P1.0 输出方波。程序中使用了两个函数：一个是主函数，另一个是中断函数。主函数主要是对看门狗定时器与端口进行设置，在主函数中有一个循环（for 语句），这是整个程序的主循环。

中断服务程序怎么知道就是看门狗定时器的中断服务程序呢，这个由它的中断向量标志[WDT_VECTOR]决定了进入中断之后 CPU 到地址 0FFE0H+WDT_VECTOR 中去找 PC 的内容。

```
#include  <msp430x11x1.h>
void main(void)
{
  WDTCTL = WDT_ADLY_250;                    // 设置看门狗定时时间 250ms
  IE1 |= WDTIE;                             // 使能 WDT 中断
  P1DIR |= 0x01;                            // 设置 P1.0 为输出方向
  _EINT();                                  // 使能总的中断
    for (;;)                                // 主循环
  {
    _BIS_SR(LPM3_bits);                     // CPU and DCO 都不需要了
    _NOP();
  }
}
interrupt[WDT_VECTOR] void watchdog_timer(void)
```

```
{
  P1OUT ^= 0x01;                          // P1.0 求反以输出方波
}
```

2．2　开发环境（实验 1）

MSP430 的 IAR 最新版本软件按照默认安装之后，由程序组的 IAR Systems－－IAR Embedded Workbench KickStart for MSP430 V3 －－IAR Embedded Workbench 可以进入 IAR 的 MSP430 开发环境（见图 2.2.1）。

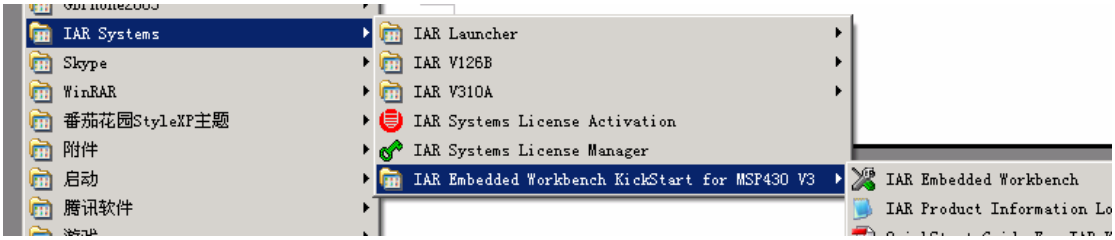


图 2.2.1　　进入开发环境的方法

第一次进入开发环境之后的界面如图 2.2.2。这时，需要添加一个工作区，以及在工作区中添加一个项目，然后在项目中添加自己的程序代码。最后编译，调试。下面将分别讲述如何操作。

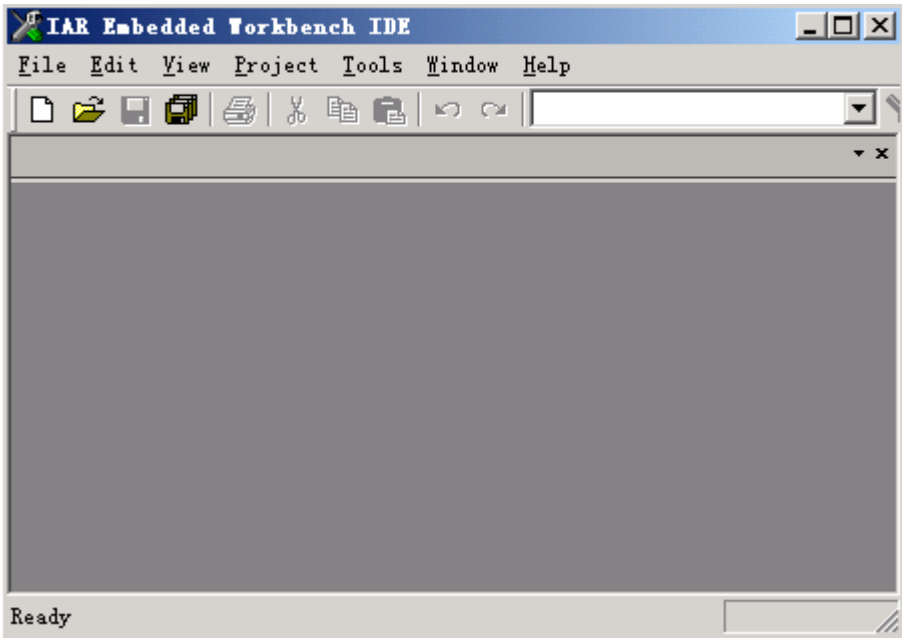


图 2.2.2　第一次进入开发环境的界面

添加一个新的工作区的方法为：

点击菜单：File－－New－－Workspace，见图 2.2.3。

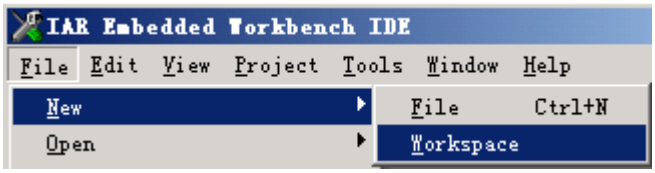


图 2.2.3　添加新的工作区

屏幕将出现新的工作区（如图 2.2.4），这时工作区内没有任何内容，需要添加一个用户项目。具体方法见图 2.2.5。图示为添加新的项目，也可以打开以前的项目（使用菜单中下面条目）。按添加新项目之后，出现图 2.2.6 所示的界面，按 ok 即可。之后需要填写项目名称，以及存放的位置，见图 2.2.7，注意记住自己输入的项目名称以及位置。生成新项目之后，在工作区中显示出来，见图 2.2.8。如果此时关闭 IAR 开发环境，会提示是否保存工作区文件以及项目文件，同时要求输入工作区名以及路径。如果再次打开 IAR 开发环境，会提示是否打开以前编辑过(或已经存在)的工作区文件，如图 2.2.9 所示。这时点击 test 文件名，再点击“open”，即可进入图 2.2.8 所示的刚才正在编辑的项目工程文件“test 项目”。

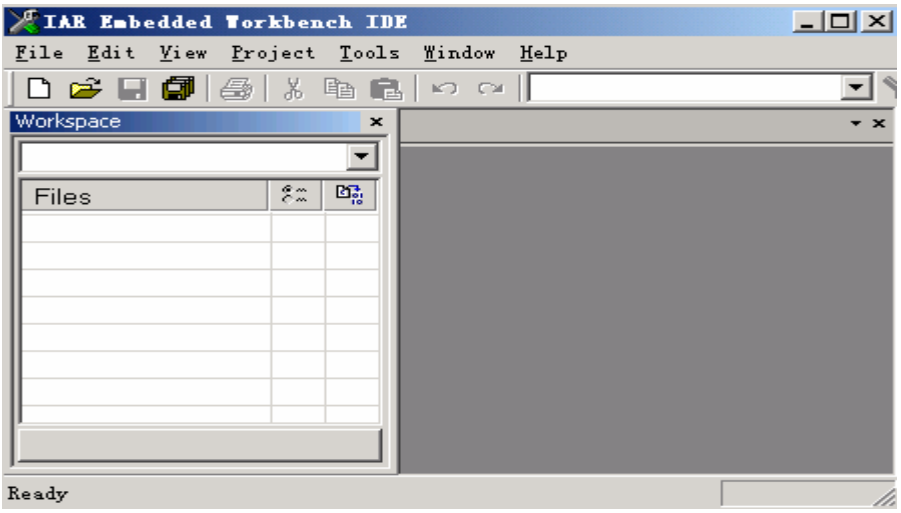


图 2.2.4　刚添加没有项目的工作区

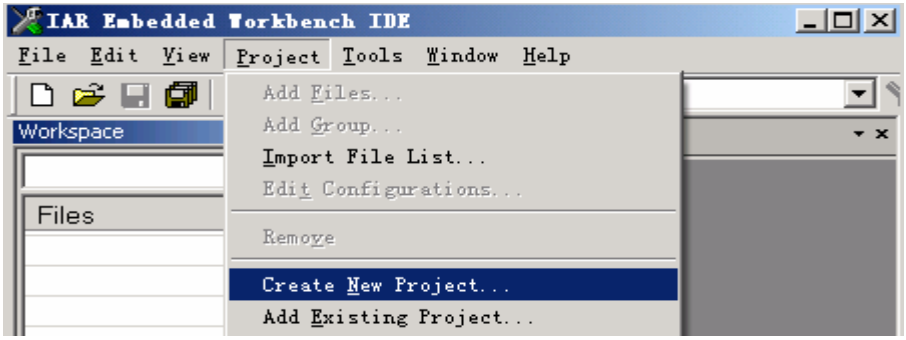


图 2.2.5　在工作区添加项目

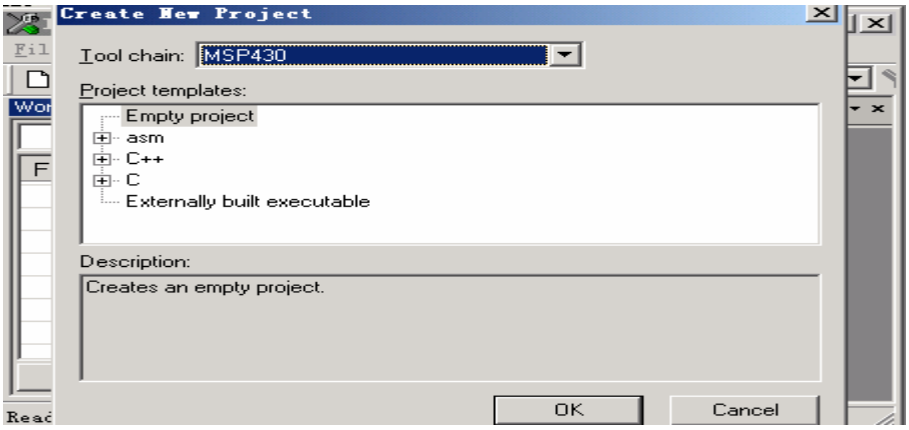


图 2.2.6　在工作区添加项目

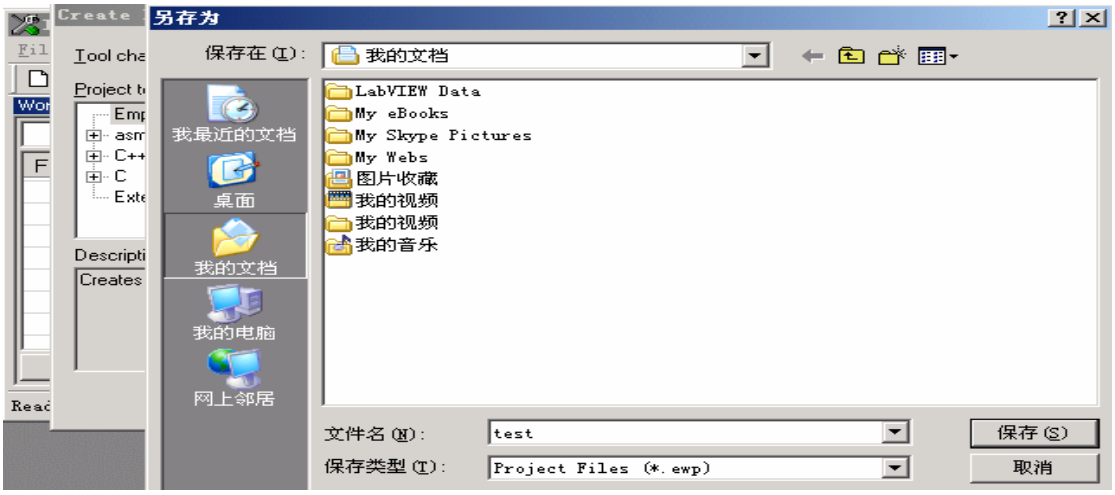


图 2.2.7　输入的项目名称以及存放位置

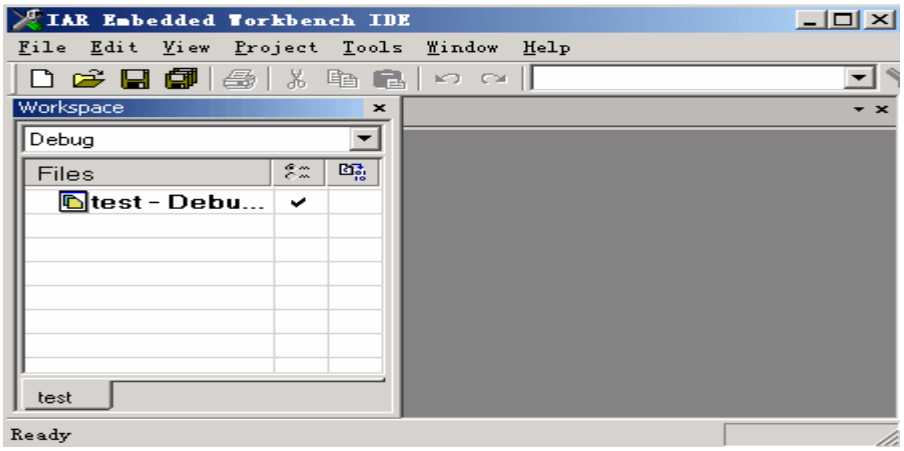


图 2.2.8　生成新项目之后，在工作区中显示出来

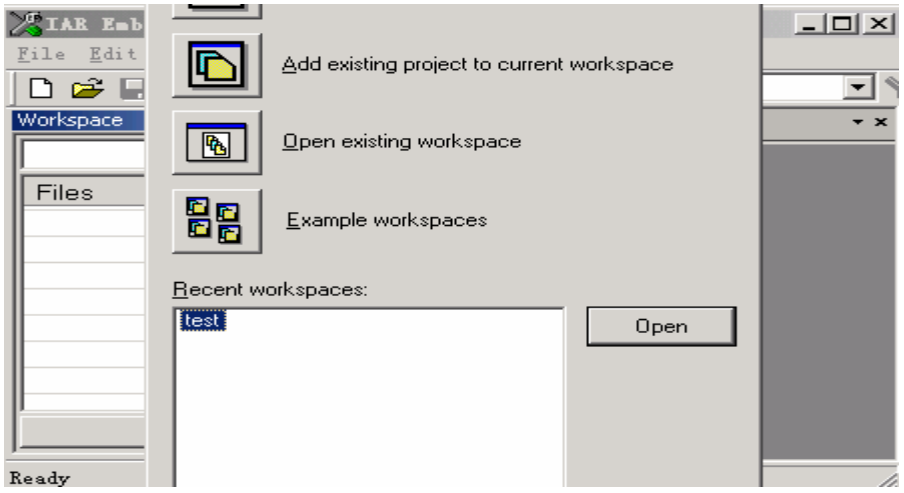


图 2.2.9　进入开发环境时 提示是否打开已经存在的工作区文件

此时的项目文件为空的项目，并没有任何实质内容，需要将所编写的源文件添加到这个空的项目中。在 IAR 开发环境中提供有现成的源代码文件，可以直接先使用。路径在（如果默认安装）C:\Program Files\IAR Systems\Embedded Workbench

4.0\430\FET_examples\fet440\C-source（C 语言例程）或者 C:\Program Files\IAR Systems\Embedded Workbench 4.0\430\FET_examples\fet440\asm-source（汇编例程）。可以直接先感受系统提供的例程。鼠标右键点击“test”项目文件名，使用添加文件（如图 2.2.10）。按照上述路径将 c 源程序“fet440_1.c”添加到项目文件中（如图 2.2.11），添加之后的项目就包含具体内容：可以使用源程序中的语句来具体说明我们这个项目要完成的什么事情。但是还不完整，还需要一些其他具体信息，最关键的是使用什么芯片来实现我们这个项目。同样鼠标右键点击 test 项目名，使用 Options 操作（如图 2.2.12）进行进一步设置。

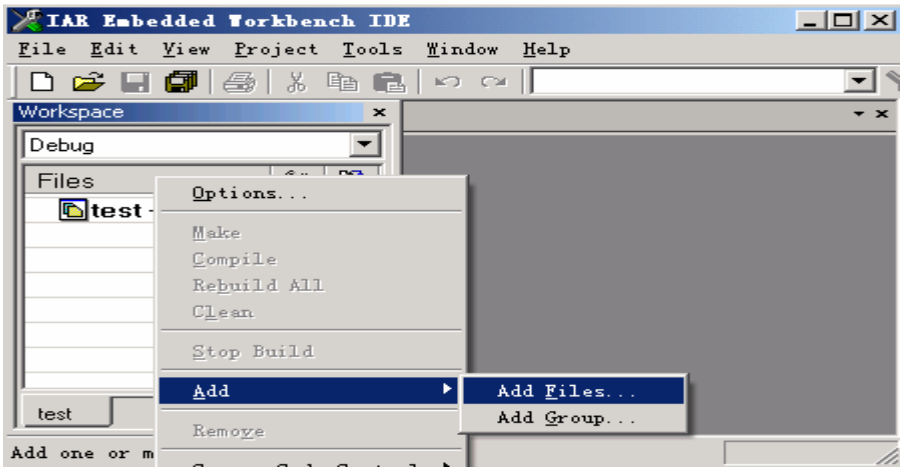


图 2.2.10　　添加源代码的操作

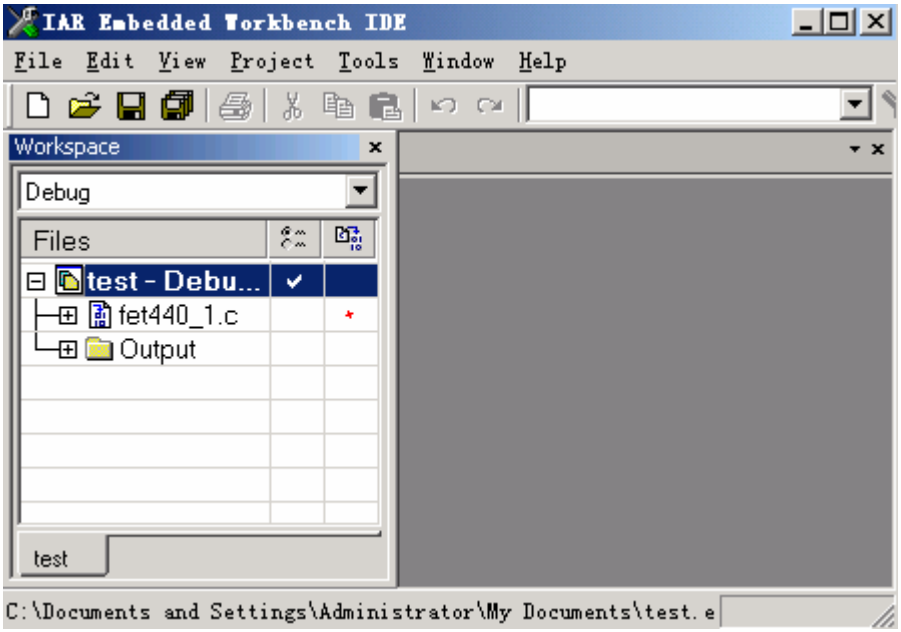


图 2.2.11 有源代码的工程项目

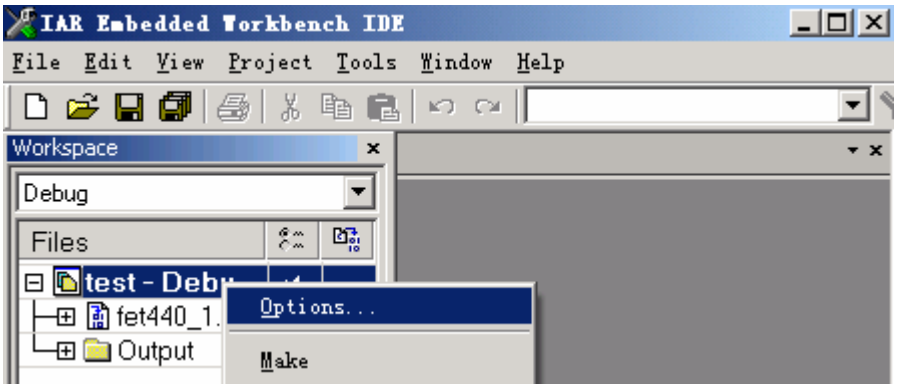


图 2.2.12 进一步设置工程的操作

点击 Options 操作之后，进入详细的操作界面（如图 2.2.13）。在这里有很多需要对“test”项目进行操作的地方。首先要确定我们这个工程项目需要使用的芯片是什么，图 2.2.14 所示的操作可以解决这个问题。图 2.2.14 所示的选择是 MSP430F449 芯片。

是使用硬件乘法器还是不使用硬件乘法器、在项目中只有汇编还是不是、浮点数据使用 32 位还是 64 位等，都可以选择。

以上这些是常用操作界面的最常用选项，其他可以使用默认设置。

下面需要设置调试方式：模拟还是联机。见图 2.2.15，选择 FET　Debugger 为联机调试，否则为模拟调试。联机调试将源程序代码先下载到器件的 FLASH　ROM 中，然后在实时在 MSP430 单片机中运行，同时通过调试软件与 MSP430 单单片机通讯，将 MCU 中的全部信息回传到调试计算机，为最真是的调试方法。

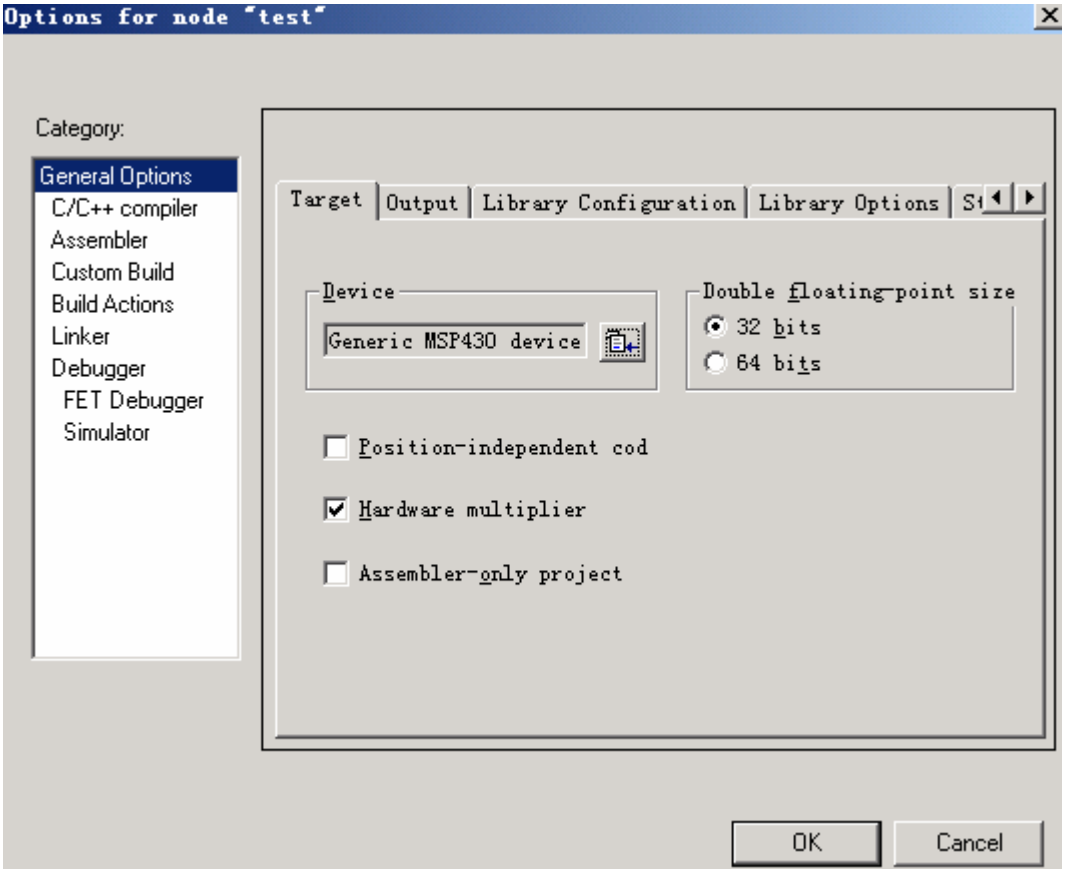


图 2.2.13　　Options 操作界面

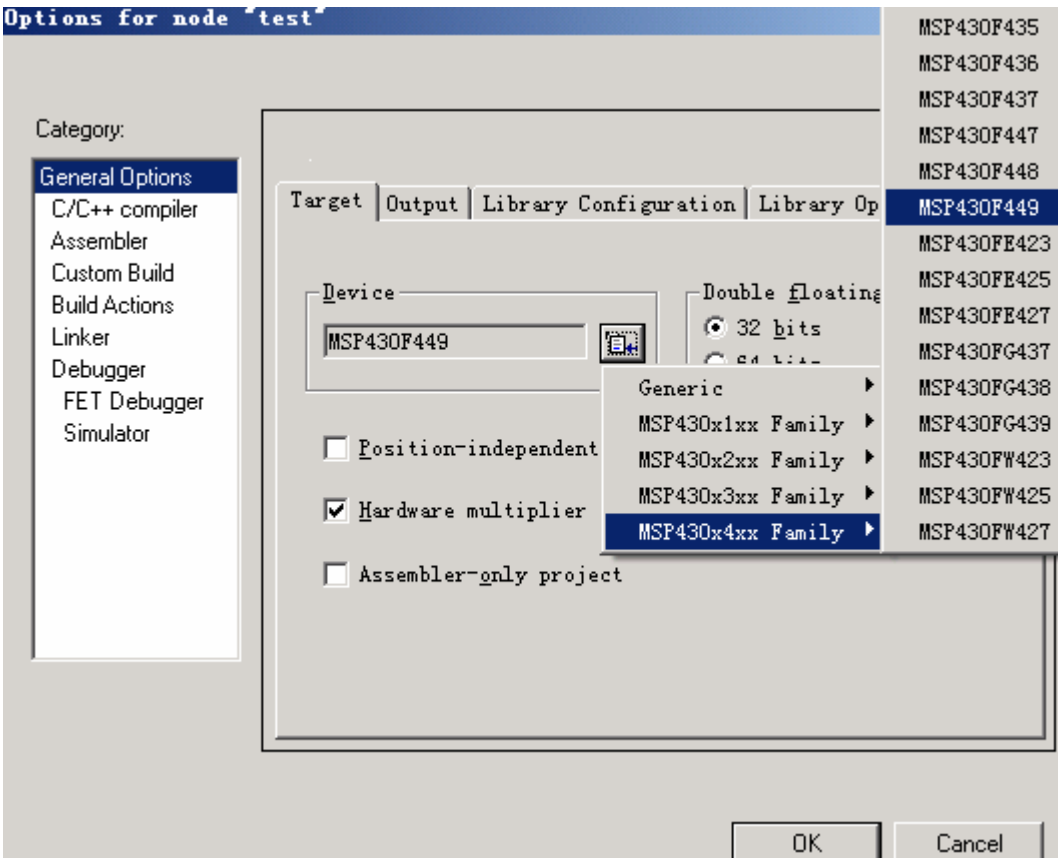


图 2.2.14　　Options 中选择器件

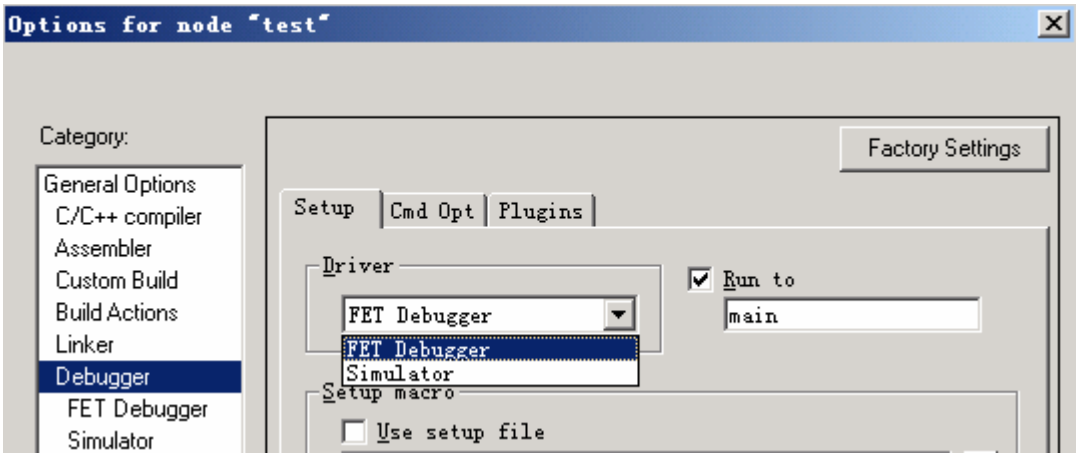


图 2.2.15　　Options 中选择调试方式

现在调试环境已经设置完毕。下面我们将进入进入 MSP430 内部，仔细瞧瞧 MSP430 单片机是什么样子。点击调试按钮或者在菜单中选择调试或者使用快捷键 Ctrl + D（见图 2.2.16）进入调试状态（注意进入调试状态之前确认目标系统通过仿真器与调试计算机连接可靠）。如果联机可靠，则点击调试之后，进入图 2.2.17 所示的调试界面，在这里我们可以清楚

地看到 MSP430F449 内部的所有情况。而调试的目的也就是真实地查看我们所设计的程序是否按照我们设计的要求运行。可以通过全速运行直接查看程序运行所发生的现象是否满足要求，也可以通过一步一步地运行、或者运行一段程序，再查看 MSP430 的内部情况，看是否正确以便调整程序。具体的方法体现在图 2.2.17 所示的项目设计开发简要流图中。

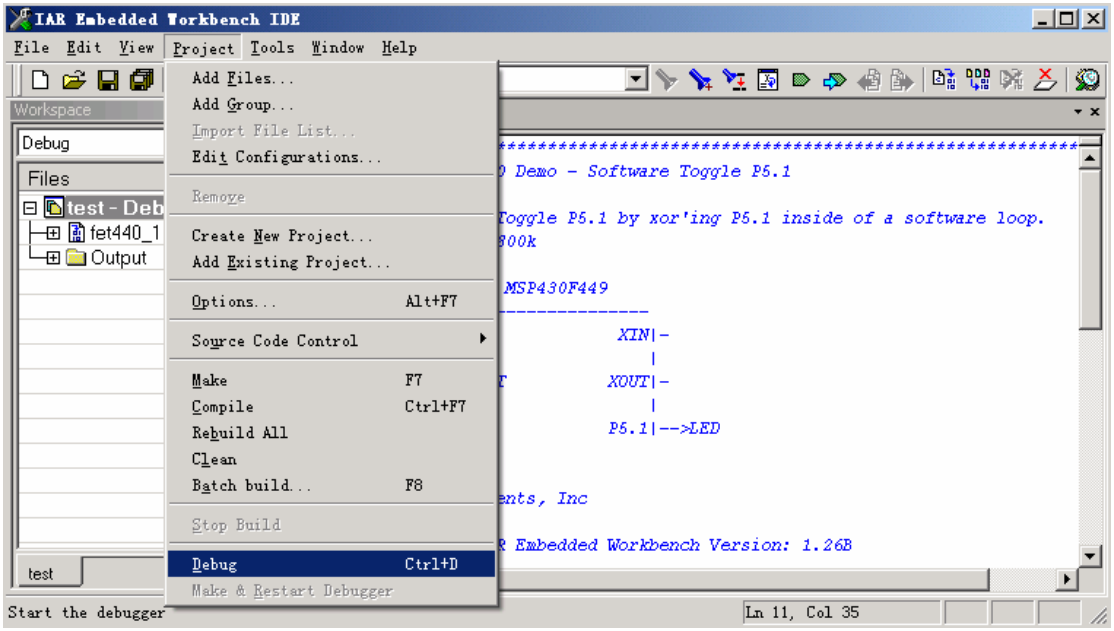


图 2.2.16　　进入调试的方法

MSP430F449 单片机内部究竟有什么一些东西呢？进入调试环境后，出来图 2.2.18 所示的界面，可以看到下面为查看存储器的窗口，由右至左依次为变量观察窗口（变量 i 的详细情况可以通过这里了解）、寄存器窗口（可以查看 CPU 内部寄存器以及片内外围设备所涉及的寄存器内容）、反汇编窗口（能看到 C 语言通过编译之后，在机器中的程序存储器中的什么位置存放了什么样的机器能执行的代码）等等。还有其他一些窗口都可以通过 View 下拉菜单选择打开。

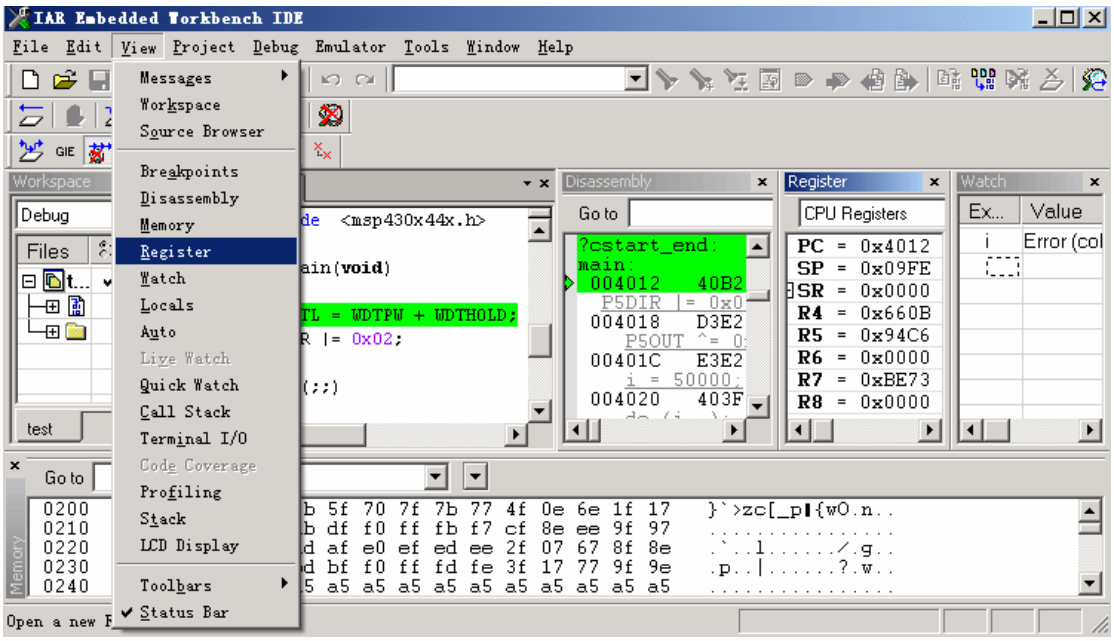


图 2.2.18　　调试环境界面

FLASH 存储器为存放程序以及一些固定数据表格的地方，是必不可少的。在图 2.2.18 所示的调试环境界面中，反映在反汇编窗口。在反汇编窗口的最左边为 FLASH 存储器的物理地址，后面为机器代码，下面为反汇编语句。FLASH 存储器的起始地址由于不同的器件 FLASH 的容量大小不一样，导致具体的起始位置也不一样。

RAM 为存放变量，以及进行大量临时数据存储的地方，更重要的是系统堆栈位于 RAM 中，在图 2.2.18 所示的调试环境界面中，RAM 由地址 200H 开始。当然了，要使用 C 语言，则直接查看 RAM 很不方便，因为我们设计者并不知道某一个变量存放在 RAM 中的某一个具体位置。使用 Watch 窗口则可以很方便实现变量的查看，在图 2.2.18 所示的调试环境界面中位于最右边的 Watch 窗口可以查看在程序中使用到的变量 i 。

在寄存器窗口中可以看到 CPU 寄存器的内容。CPU 寄存器为与 CPU 最关系紧密的寄存器，操作时间最快。CPU 寄存器有 16 个：R0－－R15。除了紧靠 CPU 的 CPU 寄存器外，还有片内外围设备所涉及的寄存器，通过寄存器的下拉菜单可以选择（见图 2.2.19）。这些寄存器有：CPU 寄存器、特殊功能寄存器、看门狗寄存器、硬件乘法器寄存器、所有端口寄存器、基本定时器寄存器、系统时钟寄存器、电源监控寄存器、液晶驱动控制寄存器、两个串口寄存器、定时器 TA　TB 寄存器、FLASH 操作寄存器、比较器寄存器、12 位模拟数字转换器寄存器等。这些寄存器会随着所选择的器件的不同而有所不同，如果所使用的器件为 MSP430F133 则没有第二个串口所涉及的寄存器、也没有硬件乘法器所涉及的寄存器，因为 MSP430F133 没有这些硬件资源。对于这些资源本书在以后所安排的实践环节中都会涉及到，并通过具体的实验教会读者如何使用它们。片内外围设备所涉及的寄存器被安排在地址 200H 以下空间。

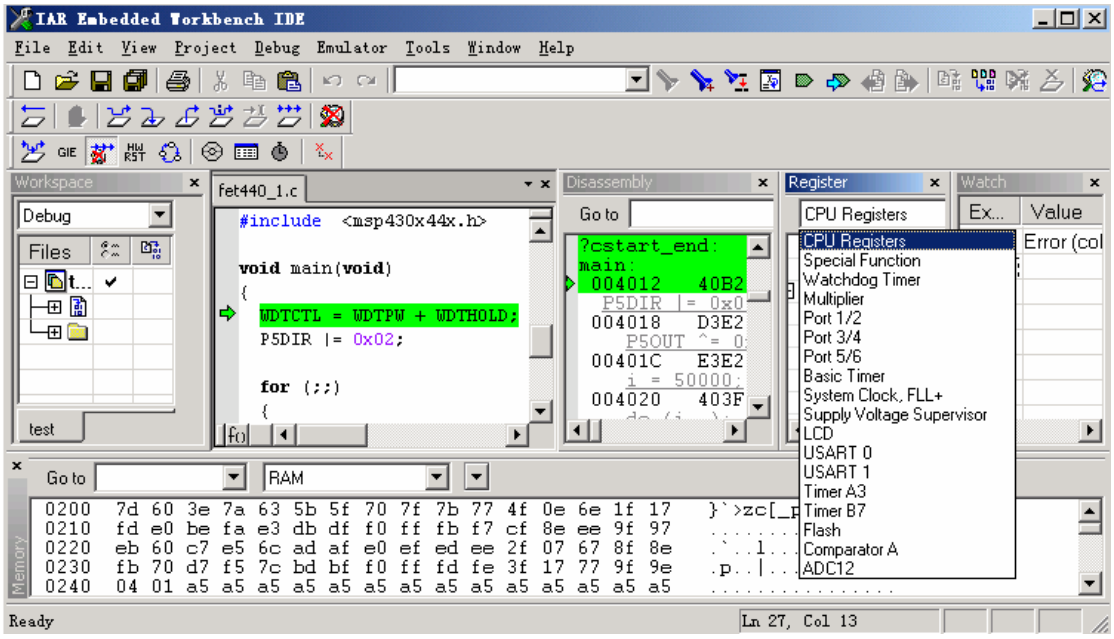
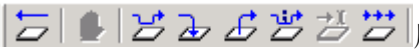


图 2.2.19　　CPU 外围模块所涉及的寄存器

接下来的事情，就是运行这个程序，按钮  用来操作程序运行。按钮“  ”为复位操作，按钮“  ”为源程序级单步运行操作，按钮“  ”为汇编代码级单步操作，按钮“  ”为连续运行操作，

2．3　　端口的认识与体会

MSP430F449 有可以输入与输出的端口 6 个 8 位，一共 48 个：P1、P2、P3、P4、P5、P6。其中 P1、P2 一共 16 个端口可以使用作为外部中断输入。所有的端口都是通过对应的寄存器控制来实现输入、输出、中断等相应的应用。

还是使用上一节的例程（直接打开 test 工程文件即可）。进入调试环境之后，打开端口 P5、P5 窗口（如图 2.3.1），可以看到 P5、P5 端口所涉及到的寄存器有 P5IN、P5OUT、P5DIR、P5SEL（P6 略），下面将详细解释它们。

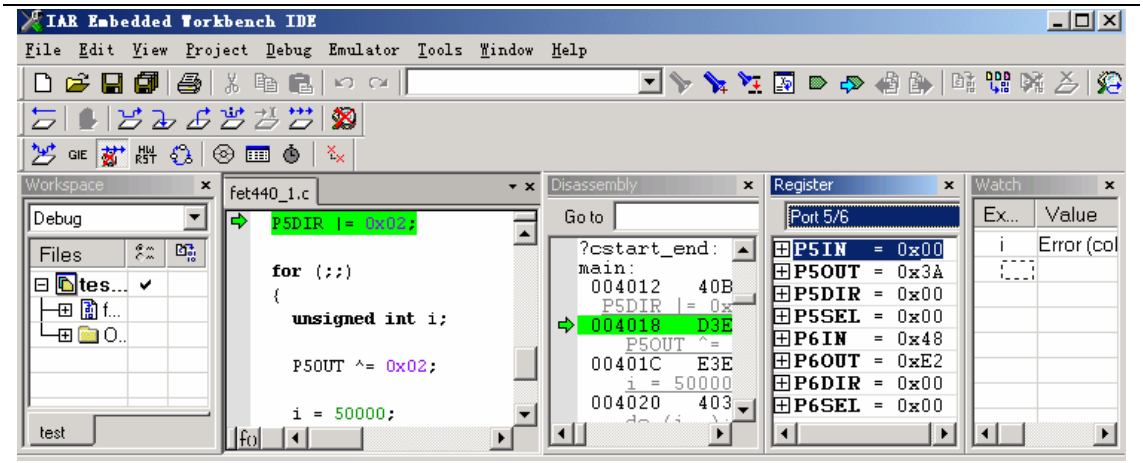


图 2.3.1　　端口 P5、P6 的寄存器

2．3．1 端口初步认识实验（实验 2）

还是以在 test 工程项目中的文件：C:\Program Files\IAR Systems\Embedded Workbench 4.0\430\FET_examples\fet440\C-source\fet440_1.c 为例，该程序的全部源代码如下：

```
#include  <msp430x44x.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;          // Stop watchdog timer
  P5DIR |= BIT1;                     // Set P5.1 to output direction
  for (;;)                           //死循环
  {
    unsigned int i;
    P5OUT ^= BIT1;                   // Toggle P5.1 using exclusive-OR
    i = 50000;                       // Delay
    do (i--);
    while (i != 0);
  }
}
```

具体含义为：首先设置 P51 为输出，然后在每隔一个固定的时间后将 P51 的输出具体内容求反。在硬件上，MSP430F449 系统的 P51 端口连接了一只发光二极管，则实际的运行结果为连接在 P51 端口上的发光二极管闪烁。

当执行语句 P5OUT ^= BIT1;时，可以看到寄存器窗口中 P5OUT 的位 1 被取反，同时可以看到连接到 P51 上发光二极管的显示状态也被取反了（原来亮则执行后熄灭，原来熄灭则执行之后亮）。

改变语句 i = 50000;中的参数可以改变执行后灯闪烁的频率。

在程序中，有一些保留字，比如 P5DIR、P5OUT、BIT1 等。这些在 msp430x44x.h 文件中都已经申明或定义过了。下面将 msp430x44x.h 的内容列举如下，本书的很多地方都会直接引用它们，其他系列器件的头文件内容大同小异，不作介绍。

在程序中使用的 P5DIR、P5OUT、BIT1 等的含义非常明显：P5DIR 就是端口 P5 的输入

输出方向寄存器，P5OUT 就是端口 P5 的输出寄存器，BIT1 就是一个数据的位 1（由位 0 开始），P5OUT ^= BIT1 的含义就是 P5OUT.1 求反。

2．3．2　头文件 msp430x44x.h 内容

2．3．3　端口相关寄存器以及端口相关知识

1　端口 P1、P2

在 MSP320X3XX、MSP430F1XX、MSP430F4XX 中都有 P1、P2。由一系列寄存器对其控制，P1、P2 口的结构如图 2.3.2 所示。可以看出有 7 个控制寄存器对每个端口操作，图中的 n=1、2,两个端口共 14 个寄存器，对它们的访问须用字节指令以绝对模式进行访问。

P1DIR、P2DIR　　P1、P2 端口方向寄存器

相互独立的 8 位分别定义了 8 条引脚的输入/输出方向。8 位在 PUC 后都被复位（即默认为输入方式）。一般地，在使用端口时，都要先定义该寄存器，使引脚的输入/输出满足设计者的要求。

0：　I/O 引脚被切换成输入模式

1：　I/O 引脚被切换成输出模式

P1IN、P2IN　　P1、P2 端口输入寄存器

输入寄存器是 CPU 扫描 I/O 引脚信号的只读寄存器，用户不能对它写入，只能通过读取该寄存器中内容以知道 I/0 端口的输入信号。此时引脚的方向必须选定为输入。

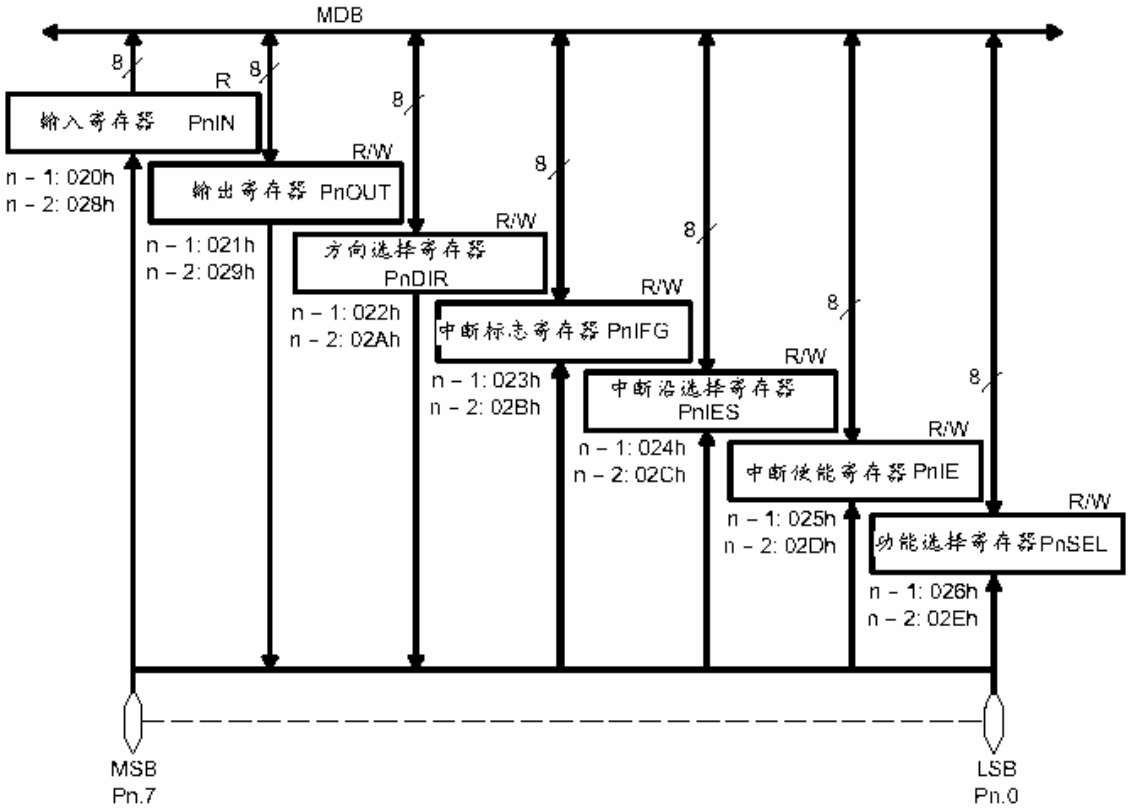


图 2.3.2　　P1、P2 口结构

P1OUT、P2OUT　　P1、P2 端口输出寄存器

该寄存器为 I/O 端口的输出缓冲寄存器。可用所有包含目的操作数的指令修改以达到改变 I/O 口状态的目的。在读取时输出缓存的内容与引脚方向定义无关。改变方向寄存器的内容，输出缓存的内容不受影响。

P1IE、P2IE　　P1、P2 端口引脚中断允许寄存器

P0 口的 8 条引脚都可能引起中断事件的发生，每一条引脚都有一位用以控制该引脚是否使能中断。P1IE、P2IE 的各位定义如下：

7	6	5	4	3	2	1	0
PnIE.7	PnIE.6	PnIE.5	PnIE.4	PnIE.3	PnIE.2	PnIE.1	PnIE.0

0：　禁止该位中断

1：　允许该位中断

P1IES、P2IES　　P1、P2 端口引脚中断触发沿选择寄存器

如果允许 P0 口的某个引脚中断，还须定义该引脚的中断触发沿，该寄存器的 8 位分别了 P0 口的 8 条引脚的中断触发沿。

0：　对应引脚有由低到高的电平跳变（上升沿）使相应标志置位

1：　对应引脚有由高到低的电平跳变（下降沿）使相应标志置位

P1IFG、P2IFG　　P1、P2 端口中断标志寄存器

该寄存器有 6 个标志位，它们含有相应引脚是否有待处理中断的信息，或相应引脚是否与中断请求。如果 P1、P2 口的某个引脚允许中断，同时选择上升沿，当在该引脚发生电平由低向高跳变时，P1IFG 或 P2IFG 的相应位就会置位，表明在该引脚上有中断事件发生。8 个标志位分别对应 8 位，如下所示。

7	6	5	4	3	2	1	0
PnIFG.7	PnIFG.6	PnIFG.5	PnIFG.4	PnIFG.3	PnIFG.2	PnIFG.1	PnIFG.0

0：　没有中断

1：　有中断请求。

P1SEL、P2SEL　　P1、P2 功能选择寄存器

P1、P2 两端口还有其它片内外设功能，考虑减少引脚，将这些功能与芯片外的联系通过复用 P1、P2 引脚的方式以实现。P1SEL、P2SEL 用来选择引脚的 I/O（输入/输出）端口功能与外围模块功能。

0：　选择引脚为 I/O 端口

1：　选择引脚为外围模块功能

2　端口 P3、P4、P5、P6

这些端口没有中断能力，其余功能与 P1、P2 一样：能实现输入/输出功能，实现外围模

块功能。每个端口有 4 个寄存器供用户使用。用户可通过这 4 个寄存器对它们进行访问与控制。图 2.3.3 为 P3、P4、P5、P6 的结构。

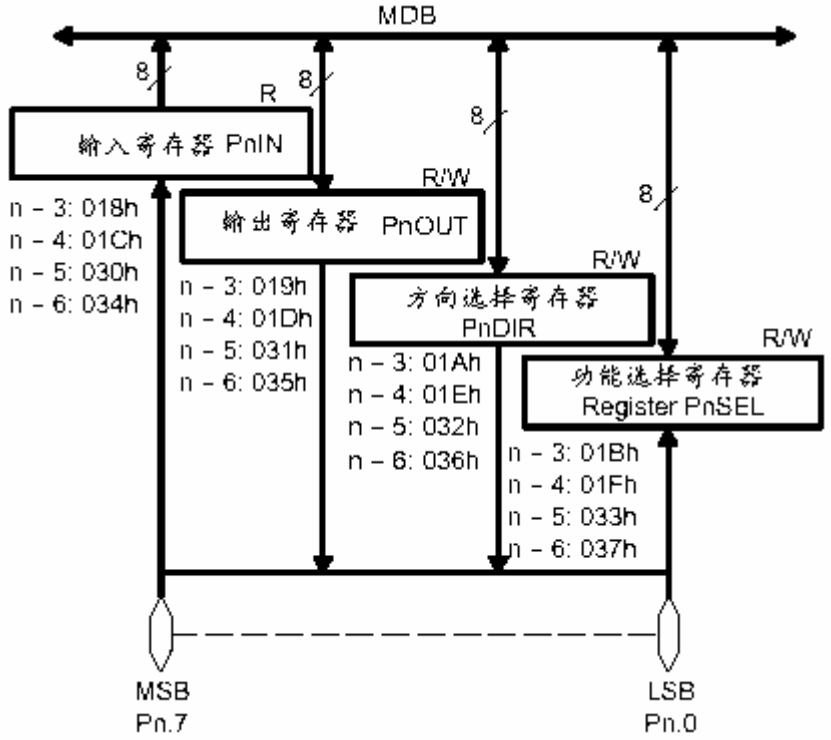


图 2.3.3　　P3、P4、P5、P6 的结构

在 MSP430F1XXX 系列器件中都有 P3、P5、P6 两端口；在 MSP430F11XX、MSP430F12XX、MSP430X3XX 等系列器件中有 P3 端口；在 MSP430F13X、14X、MSP430X3XX 等系列器件中有 P4 端口。

PnDIR　　P3、P4、P5、P6 端口方向寄存器

相互独立的 8 位分别定义了 8 条引脚的输入/输出方向。8 位在 PUC 后都被复位。一般地，在使用端口时，都要先定义该寄存器，使引脚的输入/输出满足设计者的要求。

0：　I/O 引脚被切换成输入模式

1：　I/O 引脚被切换成输出模式

PnIN　　P3、P4、P5、P6 端口输入寄存器

输入寄存器是 CPU 扫描 I/O 引脚信号的只读寄存器，用户不能对它写入，只能通过读取该寄存器中内容以知道 I/0 端口的输入信号。此时引脚的方向必须选定为输入。

PnOUT　　P3、P4、P5、P6 端口输出寄存器

该寄存器为 I/O 端口的输出缓冲寄存器。可用所有包含目的操作数的指令修改以达到改变 I/O 口状态的目的。在读取时输出缓存的内容与引脚方向定义无关。改变方向寄存器的内容，输出缓存的内容不受影响。

PnSEL　　P3、P4、P5、P6 功能选择寄存器

P3、P4、P5、P6 端口还有其它片内外设功能，考虑减少引脚，将这些功能与芯片外的联系通过复用 P1、P2 引脚的方式以实现。P1SEL、P2SEL 用来选择引脚的 I/O（输入/输出）端口功能与外围模块功能。

0：　选择引脚为 I/O 端口

1：　选择引脚为外围模块功能

2．3．4　端口输入输出实验（实验 3）

本实验将演示端口的输入与输出功能，练习如何实现端口的输入与输出为实验目的。实验电路见图 2.3.4。

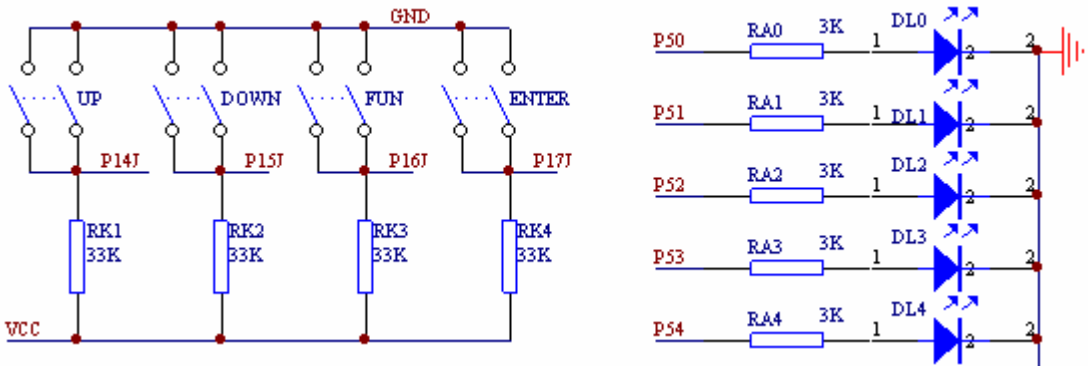


图 2.3.4　输入输出实验电路

电路中，4 只按钮连接在 P14、P15、P16、P17 上，4 只发光二极管连接在 P50、P51、P52、P53 上。

实验要求：当按下连接在 P14 上的按钮时，连接在 P50 上的发光二极管亮，没有按则熄；
当按下连接在 P15 上的按钮时，连接在 P51 上的发光二极管亮，没有按则熄；
当按下连接在 P16 上的按钮时，连接在 P52 上的发光二极管亮，没有按则熄；
当按下连接在 P17 上的按钮时，连接在 P53 上的发光二极管亮，没有按则熄；

对于这样一个要求我们如何实现呢？很明显，4 只按钮应该作为输入使用，而发光二极管应该作为输出使用。所以第一步应该是端口的设置：设置 P14、P15、P16、P17 为输入，设置 P50、P51、P52、P53 为输出，对应的语句为：

```
P1DIR　=　0x0F;
P5DIR　=　0x0f;
```

当按钮按下的时候，根据电路图，可以知道，对应的输入寄存器为“0”，而没有按下的时候，对应的输入寄存器为“1”，因为有上拉电阻。所以程序就很简单了：读取并判断 P14 等的输入信息，如果高电平则表示没有按钮按下，P50 等输出低电平，如果低电平则表示有按钮按下，P50 等输出高电平。

```
for(;;)
    {
    if（(P1IN&BIT4）　==　BIT4）
            P5OUT &= ~BIT0;
    else
            P5OUT |= BIT0;
    if（(P1IN&BIT5）　==　BIT5）
            P5OUT &= ~BIT1;
```

```
        else
                P5OUT |= BIT1；
        if（（P1IN&BIT6） ==  BIT6）
                P5OUT &= ~BIT2；
        else
                P5OUT |= BIT2
        if（（P1IN&BIT7） ==  BIT7）
                P5OUT &= ~BIT3；
        else
                P5OUT |= BIT3；
    }
```

2．3．5　　端口中断与输出实验（实验 4）

端口 P1、P2 都具有中断能力，什么是中断呢？中断在处理器中的地位非常高！处理器具有实时的能力完全源自中断。

简单地说，中断就是中途打断。就是计算机在正常处理事务的时候，有更紧急的事情需要马上处理，由中断机制来完成。

举个简单的例子来说明中断的必要性与如何实现中断：

现在你正在吃饭或者你正在看书；

突然电话响了，你的上司来电话说有重要的事情，要你马上处理；

这时，你需要思考与判断：应该继续将手头的事情做完、还是处理电话中交代的事？

最后，你得出结论，停止吃饭或者看书，马上出发，去处理电话中的事情；

正在处理电话中的事情；

处理完毕，回来，继续吃饭或者看书。

以上的例子非常清楚地讲述了什么是中断以及如何出理中断：

现在你正在吃饭；	——吃饭是 CPU 现在运行的程序；
突然电话响了；	——CPU 得到中断请求；
你的上司来电话；	——具体的中断源，是哪个中断请求；
你需要思考与判断；	——中断仲裁；
你得出结论，停止吃饭，去处理电话中事情；	——中断响应；
正在处理电话中的事情；	——执行中断中的语句，完成中断程序；
处理完毕，回来，继续吃饭或者看书。	——中断返回。

中断不但可以处理紧急事务，还可以将 CPU 完全解放出来。通过这个实验读者可以发现这一点。本实验还是实现上一实验的目的（电路图同，略）。本实验使用中断实现。

当按下连接在 P14 上的按钮时，连接在 P50 上的发光二极管亮，没有按则熄；

当按下连接在 P15 上的按钮时，连接在 P51 上的发光二极管亮，没有按则熄；

当按下连接在 P16 上的按钮时，连接在 P52 上的发光二极管亮，没有按则熄；

当按下连接在 P17 上的按钮时，连接在 P53 上的发光二极管亮，没有按则熄；

由于使用中断，所以本实验的程序分为两部分：主程序（CPU 的日常事务），中断服务程序（CPU 的紧急事务）。由于主程序没有其他事做，所以，只有初始化好两个端口之后就睡眠，CPU 睡眠后，就消耗很少能量了。这也是低功耗的设计方法。睡眠有很多等级，如果处于深度睡眠，则 CPU 基本不消耗能量。

端口应该如何设置？对于 P5 端口，与上一实验相同；但对于 P1 端口则不一样了，需要对中断进行设置。首先使用 P1DIR 将 P14、P15、P16、P17 设置为输入，然后使用 P1IE 将 P14、P15、P16、P17 设置为可以中断，最后使用 P1IES 将 P14、P15、P16、P17 设置为下降沿中断。

是否这些语句运行之后就能够进入中断呢？还有一个重要的地方，就是必须首先允许中断，如果没有允许中断，那怎么也进入不了中断。所以千万不要忘记了，如果你的设计有中断，一定要将中断的所有通路打开！

还有一个地方也是比较关键的，就是 MSP430 所有的处理器都内含看门狗，而且默认处于打开状态，看门狗是什么呢，在后面会单独讲解，这里简单说一下：看门狗是为计算机系统安全考虑的一个外围设备，在计算机出故障的时候，可以通过看门狗自动恢复计算机的继续运行。在看门狗启动之后，就必须不断在看门狗时间内将其内容通过程序清除，所以，先不考虑其安全性，关闭看门狗。

主程序语句如下：

```
void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;          // 关闭看门狗
    P5DIR  |=   0x0F;                  // 设置 P5.0－P5.3 为输出方向
    P1DIR   =   0x0F;                  // 设置 P1.4 －P1.7 为输入方向
    P1IE    =   0xf0;                  // 设置 P1.4 －P1.7 可以中断
    P1IES   =   0xf0;                  // 设置 P1.4 －P1.7 为下降沿中断
    _BIS_SR(LPM3_bits + GIE);          // 进入最低功耗睡眠，打开总总断开关
  for (;;)
  {
  }
}
```

下面编写中断服务程序。本中断服务程序要实现当按钮按下时对应的发光管亮，否则熄。程序上应该与上一实验一样，只不过，不需要 CPU 无时无刻地判断是否按钮按下，而是通过中断实现。这样将 CPU 解放出来，可以完成更多的日常事务。

中断程序有对应的编程规则：语句“#pragma vector=PORT1_VECTOR”指明中断函数是为什么中断服务的，即中断向量。本程序将为端口 P1 服务，所以使用 PORT1_VECTOR 中断向量，该 PORT1_VECTOR 的含义在头文件中有明显表述，请仔细查阅。语句“__interrupt void p1int(void)”为中断函数的函数声明。以下为具体的中断函数内容。

需要注意的是：P1 的中断标志不能自动清除，需要人为清除，所以程序最后需要清除端口 P1 的中断标志，否则会引起中断嵌套，引起死循环。

```
#pragma vector=PORT1_VECTOR
__interrupt void p1int(void)
    {
    if（（P1IN&BIT5）  ==  BIT5）
            P5OUT &= ~BIT1 ；
    else
            P5OUT |= BIT1；
    if（（P1IN&BIT6）  ==  BIT6）
            P5OUT &= ~BIT2；
    else
            P5OUT |= BIT2；
    if（（P1IN&BIT7）  ==  BIT7）
            P5OUT &= ~BIT3；
    else
            P5OUT |= BIT3；
    P1IFG   =   0   ；
    }
```

以上程序不完善，不能实现按钮按着的时候对应灯亮，松开的时候对应灯熄。如何实现，留给读者自己实现。

2．4　　端口趣味实验－－音频（实验 5）

如何让单片机发声？首先要知道什么是声音－－声音由震动产生；扬声器发声的原理：处在磁场中有电流的线圈产生震动，继而发声。不同的频率产生不同的声音；MSP430 的端口可以输出不同的频率；继而可以推动扬声器产生不同的声音。图 2.4.1 为本实验的电路图。

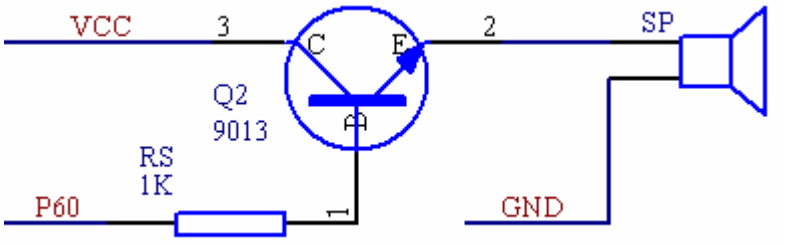


图 2.4.1　　扬声器电路

下面三段程序将发出什么声音呢？

第一段程序（已经设置了P60为输出方向）

```
 void gun(void)
{
```

```
        for(ff=40;ff<200;ff++)
          {
          P6OUT ^= BIT0;                    // 对输出置反
          for(tmp=0;tmp<ff;tmp++);          // 延时
          }
}
```

第二段程序

```
void s1(void)
{
        for(ff=30;ff<500;ff++)
          {
          P6OUT ^= BIT0;                    // 对输出置反
          for(tmp=0;tmp<ff;tmp++);          // 延时
          }
}
```

第三段程序

```
void s2(void)
{
        for(ff=50;ff<100;ff++)
          {
          P6OUT ^= BIT0;                    // 对输出置反
          for(tmp=0;tmp<ff;tmp++);          // 延时
          }
}
```

以上三段程序的结构完全一样，但是效果完全不一样！
主要为：
频率不一样；
各频率段的延时不一样。
第三段频率高，延时短。

这里只是一个提示，读者可以在此基础之上实现更美妙的声音输出。

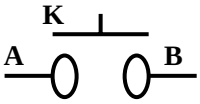
2．5　　键盘原理与应用实践

在上一节的两个实验，已经使用过键盘了，通过键盘来控制发光管的亮与熄。在单片机应用中，键盘是人机对话的输入设备，借助键盘可向系统设置参数，发出控制指令等等。但具体应用中，很少到大街上买一个通用的计算机键盘！而多数键盘由设计者自行设计。在这部

分将详细讲述键盘在 MSP430 单片机中的应用。

1　按键的工作原理

在单片机设计中常用轻触按键作为输入设备—键盘的单元电路。它一般结构是由两个电极与一个弹簧金属片构成的，如右图所示。当金属弹簧片上的按键 K 按下时，两个电极 A 与 B 被连通。但实际使用并非如此简单，因为单片机的运行速度相对于操作者按下按键的速度来是太快了，所以就不得不考虑更多的一些细节问题了。



先看看我们按按键的细节过程。这个细节主要是按下按键的前后都有抖动！如果按键的 A 端接地，B 端接上拉电阻，则平时按键的 B 端为高电平，当按键按下时为低电平，松开后又是高电平，这是理论值。而实际上呢，B 端的情况如下图所示（图 2.4.1）。

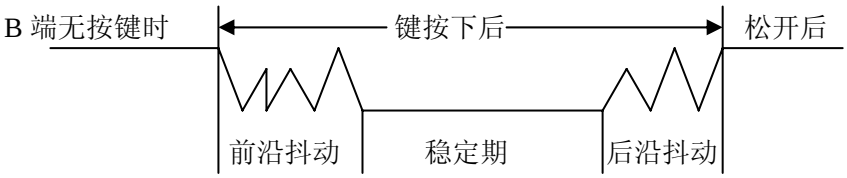


图 2.4.1　按键的抖动

如果将按键的 B 端信号送到 MCU，系统会认为有几个低电平，按了几次按键呢。因为 MCU 认为低电平是按键按下，而抖动过程有很多低电平，同时还有不短的时间，一般在 10 毫秒左右。那么怎么解决呢，常用的有 3 种办法：使用 R-S 触发器构成消抖动开关以消抖动；使用电阻与电容构成积分器将抖动滤除；使用软件延时的办法消除抖动。前两种方法使用硬件没什么好说的，最后一种使用的是软件的方法，怎么回事？其实很简单：当 MCU 得知按键的 B 端出现低电平时，就知道可能有按键按下了，于是等待 10 毫秒，10 毫秒之后再检测按键的 B 端，如果还是低电平，则就一般的机械按键而言，已经是处于稳定期了，按键的抖动被消除了；如果 10 毫秒之后按键的 B 端没有低电平了，则说明是干扰信号，而非按键按下。

2　键盘程序的一般编写方法

键盘是由若干上述的独立按键按一定的规则组合而成的。也就是说键盘的基本元素是按键，那么消除按键的抖动是必须的。键盘是由若干按键构成的，究竟是哪一个按下了呢，这就是通过判键得到键值。MCU 知道了是哪一个按键按下了之后是不是就该退出呢？如果这时退出，而按键还没有松开，则 MCU 又知道了有按键按下，又去消除抖动，再判断是哪个按键按下，如果还没有松开，则又被认为有按键按下，…….这下惨了,本来你按了一次按键,而系统则认为你按了很多次按键。所以得到键值之后，还有一件事情就是等待按下按键的松开（注意一点：如果系统中使用了看门狗，则在这里请不断地清空看门狗，因为如果使用者一直很长时间按着键，则看门狗超时、系统复位）。综合起来，一般的键盘程序有如下三个步骤：

A　消除按键抖动（如果使用硬件，则可略）；

B　判断是哪个按键按下，识别键码；

C　等待按键松开。

3　通用独立按键式键盘设计（实验 6）

在系统中需要少量按键时，可使用按键与单片机的 I/O 口线直接连接的方法构成。我们平常使用的电子表、计步机等，它们体积都很小巧，不可能使用很多按键构成输入部分，一般有 4 个左右的按键。在这种情况下，就使用独立按键式键盘。如图 2.4.2 所示。

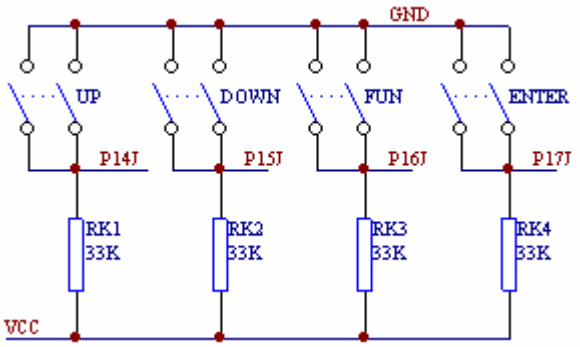


图 2.4.2　　4 只独立按键直接与 4 条口线相连

在软件的编写上，可采用查询方式，也可采用中断方式。因为 MSP430 的 P0、P1、P2 等三个 8 位端口都有中断能力，建议读者使用中断方式。键盘程序主要是按上述的三个步骤进行编写。在主程序中必须设置 P1 口使之能进入中断。

主程序：（提供端口设置，使之能进入中断）

```
    unsigned char   keybuf;                    //    全局变量，键值缓存
……
    WDTCTL = WDTPW + WDTHOLD;          // 关闭看门狗
    P1DIR    =    0x0F;                    // 设置 P1.4 －P1.7 为输入方向
    P1IE     =    0xf0;                    // 设置 P1.4 －P1.7 可以中断
    P1IES    =    0xf0;                    // 设置 P1.4 －P1.7 为下降沿中断
    _BIS_SR(LPM3_bits + GIE);          // 进入最低功耗睡眠，打开总总断开关
    ……
```

是否有按键按下的判断：由于连接在键盘上的端口都有上拉电阻，所以平常这些端口为高电平，按下之后为低电平，所以在按下之后可以判断 P1IN 的内容中连接在键盘上的那些位，应该为“0”。所以下面的程序如果返回 0xf0 则表示没有按键按下，如果返回不是 0xf0 则表示有按键按下。

```
unsigned char p1keyj(void)                 //    判键子程序
    {unsigned char x;
     x=(P1IN&0Xf0);                        //    P14--P17 接有按键
     return(x);                            //    有按键返回  非全 1
    }
```

下面为键值的查找。下面的思路为一个一个地查找。

```
unsigned char   keycode()                    //   找哪个按键被按下，查键值子程序
    {
    unsigned char   x;
    if((P1IN&0xf0)= = 0x80)                  //   是否第一个按键
        then x=0;
        else
        if((P1IN&0xf0)= = 0x40)              //   是否第二个按键
            then x=1;
            else
            if((P1IN&0xf0)= = 0x20)          //   是否第三个按键
                then x=2;
                else
                if((P1IN&0xf0)= = 0x20) //   是否第四个按键
                    x=3;
    return(x);
    }
```

中断服务程序：

```
    #pragma vector=PORT1_VECTOR
    __interrupt void p1int(void)
    {                                   //端口 1 的中断服务程序
    while(p1keyj()!=0xf0)               //没有按键按下，返回全 1－－0xf0
        {
        delay(500);                     //延时消除抖动
        while(p1keyj()!=0xf0)
            {
            keybuf   =   keycode();//确信有按键按下，找按键得键值，送到全局变量 keybuf
            while(p1keyj()= =0)         //等待按键松开
                    ;                   //做对应键盘的事务
            }
        }
    }
```

以上程序使用中断方式，使用查询方式的程序差不多，这里略，请读者自己编写。

汇编主程序：

```
        ……
        BIC.B    # 0F0H,  &P1DIR      ；P1.4 －P1.7 为输入模式
        BIS.B    #0F0H,   &P1IE       ；P1.4 －P1.7 中断使能
        BIS.B    # 0F0H,  &P1IES      ；P1.4 －P1.7 下降沿触发中断
        EINT                          ；总中断使能
```

```
        ……
    中断服务程序：                          ；出口参数：按键键值在 R5 中
P1KEY3  CALL    #KEYJ3          ；判断是否有按键
        JNC     KEYEND          ；没有则退出
        CALL    #DELAY10MS      ；如有，则延时 10 毫秒          } 消抖动
        CALL    #KEYJ3          ；再判键
        JNC     KEYEND          ；如没有按下则退出
        CALL    #KEYCODE3       ；如有，则调认键程序得到键值
        PUSH    R5              ；保护键值
KEYLOOP CALL    #KEYJ3          ；等待按键松开
        JC      KEYLOOP         ；没有松开，则继续等待
        POP     R5              ；按键松开之后，恢复键值
KEYEND  RETI

KEYJ3   BIT.B   #07H，  &P1IN   ；判断有无按键按下，如果有，则 C=1
        RET                     ；如果没有按键按下，则 C=0

KEYCODE  BIT.B  #BIT4,  &P1IN   ；判断 3 个按键中是哪一个被按下
        JNC     K2
        MOV   #0,   R5          ；如果是接到 P1.4 的按键，则输出 R5=0
        RET
K2      BIT.B  #BIT5,   &P1IN
        JNC   K3
        MOV   #1,   R5          ；如果是接到 P1.5 的按键，则输出 R5=1
        RET
K3      BIT.B  #BIT6,   &P1IN
        JNC   K4
        MOV   #2,   R5          ；如果是接到 P1.6 的按键，则输出 R5=2
K4      BIT.B  #BIT7,   &P1IN
        JNC   K5
        MOV   #3,   R5          ；如果是接到 P1.7 的按键，则输出 R5=3
K5      RET
```

2．6　列扫描式式键盘原理与应用（实验 7）

如果应用系统需要较多的按键呢，采用独立式键盘的结构则不能满足需要。比如需 16 个按键，则独立式键盘将花费系统系统资源为 16 条 I/O 口线，显然是不科学的。这里使用行列扫描的方法实现键盘接口（也被叫作矩阵键盘），则可使用少量的 I/O 口线连接较多的按键。图 2.6.1 为通过 MSP430 的 P1 口接的 4*4=16 个按键（编号为 0~15）构成的行列式扫描键盘。下面来分析如何在行列式扫描键盘上实现键盘的三个步骤：判键消抖动，键码识别，等待按

下按键的松开。

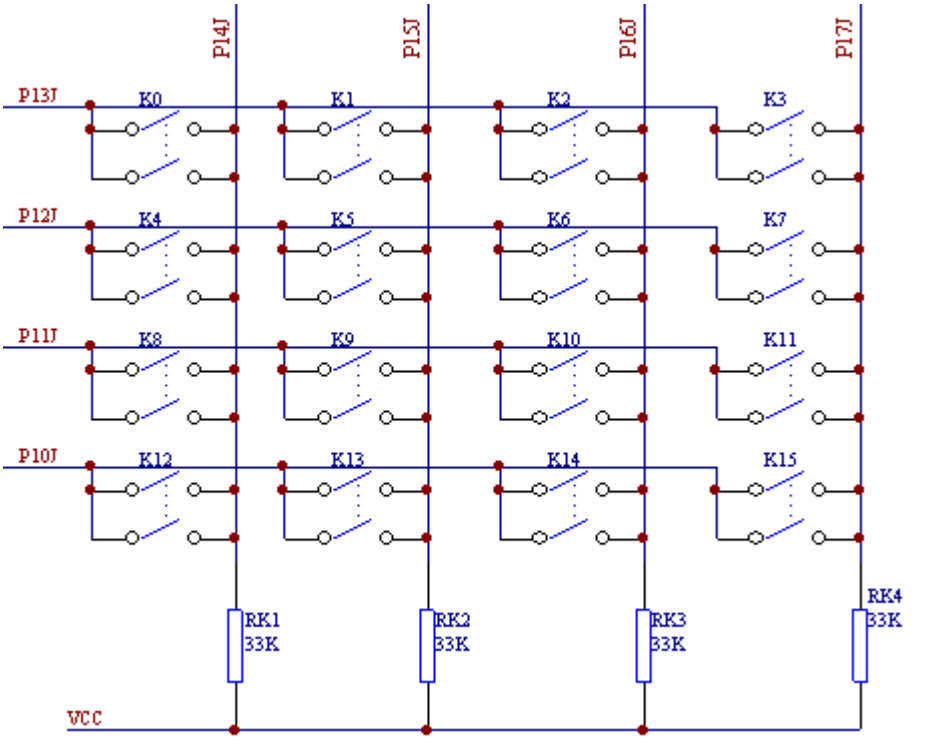


图 2.6.1　　扫描键盘接口

A　判断有无按键按下的判键

在图 2.6.1 中，P1 口的 8 条 I/O 口线被分为 4 条行线 P1.0~P1.3，4 条列线 P1.4~P1.7，其中列线分别由电阻上拉到 VCC。在行线与列线的每一个交界处有个按键，按键的两端 A 与 B 分别接在行线与列线上。如果有按键按下，则与之相接的行线与列线被连通。在检测是否有按键按下时，先让四条行线 P1.0~P1.3 输出低电平，读列线 P1.4~P1.7，如果有按键按下，则列线读进来的数据为非“1”；如果没有按键按下，则因所有的列线被上拉，读入 MCU 的数据为“1”。由此即可判断是否有按键被按下。相应程序如下。

汇编程序：

```
KEYJUDGE    MOV.B    #0FH,&P1DIR
            MOV.B    #00H,&P1OUT
            BIT.B    #0F0H,&P1IN      ; 读四根列线是否有低,是否有按健
            RET                       ; 若有，C=1
```

C 语言程序：

```
unsigned char keyj(void)
    {unsigned char x;
    P1DIR=0x0f;
    P1OUT=0x0;          //键盘硬件：P10--P13 为行线，最上面一根为 P10
    x=(P1IN&0XF0);      //          P14--P17 为列线，
```

```
return(x);                    // 无按键，返回 0xf0;   有按键返回 非 1
}
```

按键抖动的消除同样是使用软件延时的办法，当检测到有按键按下之后，等待 10 毫秒在检测是否有按键被按下，如果这时有按下的键，则已经是键的稳定期。具体的程序请看稍后的键盘子程序。

B　按键识别，得到键码

对于行列式矩阵键盘常使用扫描的方法识别按键。在前面判键部分已经讲到，通过让四条行线 P1.0~P1.3 输出低电平，读列线 P1.4~P1.7 的办法来得知是否有按键被按下。那么可以用同样的办法来确认究竟是哪一个按键被按下。

假定图 2.5.1 中的 15 号键被按下，那么下面的办法将能找到按下的按键。

（1）　输出 P1.0 为低电平，其余 P1.1~P1.7 都是高电平；

（2）　读入列线，如果 P1.4 为低电平，则说明“K12”号键被按下，因为没有被按下，所以为高（被上拉）；再测试 P1.5，看是否为低电平；……直到测试完 P1.7；

（3）　然后再输出 P1.1 为低电平，其余 P1.2~P1.7、P1.0 都是高电平；

（4）　读入列线，如果 P1.4 为低电平，则说明“K8”号键被按下，因为没有被按下，所以为高（被上拉）；再测试 P1.5，看是否为低电平；……直到测试完 P1.7；

（5）　输出 P1.2 为低，……

（6）　读列线，……

（7）　输出 P1.3 为低电平…….

（8）　这时，读列线，就会发现 P1.7 为低电平了，为什么被上拉的 I/O 口线会是低电平呢，因为它与低电平输出的 P1.3 连在了一起！由此就找到了被按下的按键连接在 P1.3 与 P1.7 上。这种方法常被称为扫描法，因为行输出是一个 0，一个 0 地输出，就象电视机的扫描线，一行一行地扫一样。

上面所讲述的办法找到了被按下按键的确切位置，但识别按键最终要送出一个表示按键位置的键值。究竟给一个什么数据来表示该位置上的按键呢。一般的方法是将输出的行扫描码与读入的列值组合起来，就是表示被按下按键位置的数据，比如第 15 号按键被按下，则行扫描码为 08H，而读入的列数据为 08H（P1 口的高 4 位），所以 15 号按键的键值为“88H”。同样 9 号键的键值为“42H”。很明显，这样的键值数据太疏松了，跨度很大，这样会给使用键盘的其他程序带来很大的不便。

4 条行线，4 条列线连接了 16 个按键，如果能让所有 16 个按键的编码为 0~15，与图中所给的按键编号一样，那是最理想的。我们发现每一行线都通过 4 个按键与 4 条列线相连接，能不能给第一条行线上的 4 个按键编码为 0~3；后一条行线上的 4 个按键为前一条行线上对应的按键键值加 4 呢？答案是肯定的。先看如下的程序框图（图 2.6.2）。

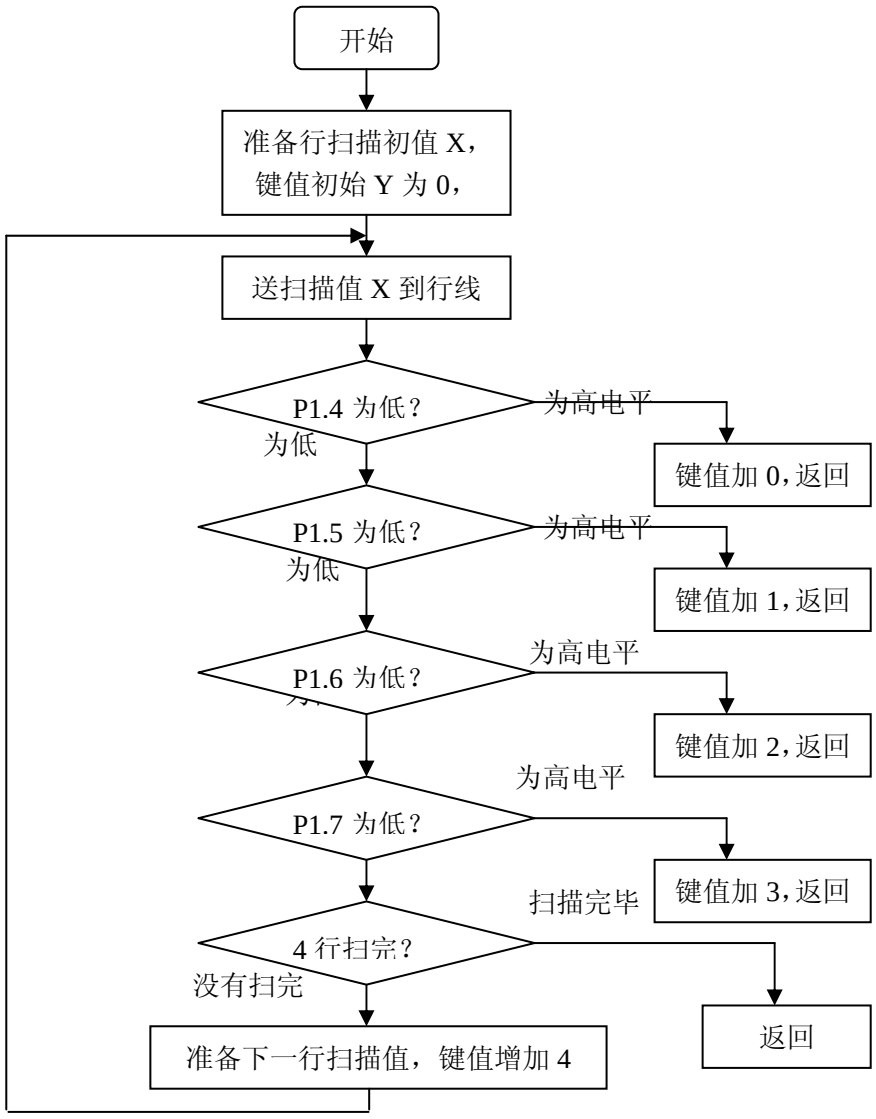


图 2.6.2　扫描式键盘的认键流程图

行内相邻两键键值为加 1 递增，行间每列上两相邻两键为加 4 递增。这样便实现了按键键值的自然顺序编码。汇编程序如下。

```
KEYCODE        MOV.B     #0,&P5DIR              ；关闭显示
               MOV.B    #0FH,&P1DIR ;低 4 位作为扫描线行输出，高 4 位作为列线读入
               MOV       #0,R9
               MOV       #1,R8
KEYCODELOOP    MOV.B    R8,&P1OUT              ;R8 为扫描信号的输出
               BIT.B     #10H,&P1IN
               JC        KEYCODE1               ;测试 P1.4
               BIT.B     #20H,&P1IN
```

```
                JC          KEYCODE2            ;测试 P1.5
                BIT.B       #40H,&P1IN
                JC          KEYCODE3            ;测试 P1.6
                BIT.B       #80H,&P1IN
                JC          KEYCODE4            ;测试 P1.7
                RLA.B       R8                  ；改变扫描码
                ADD.B       #4,R9               ；键值的行间改变
                CMP.B       #12,R9              ；四 根行线扫描完了吗
                JNZ         KEYCODELOOP
                RET
KEYCODE1        ADD         #0,R9               ；键值的行内改变
                RET
KEYCODE2        ADD         #1,R9
                RET
KEYCODE3        ADD         #2,R9
                RET
KEYCODE4        ADD         #3,R9
                RET
```

C 语言程序如下：

```
unsigned char key(void)                 //此程序键盘为 4 行*3 列，12 个按键
  {
    unsigned char x=0xff;
    P1DIR=0X0F;
    P1OUT=0X01;                         //扫描第一行
    if((P1IN&0X70)==0X10)               //如果第一行、第一根列线上有键按下，则键值为 0
       x=0;
       else
          if((P1IN&0X70)==0X20)         //如果第一行、第二根列线上有键按下，则键值为 1
            x=1;
                else
                  if((P1IN&0X70)==0x40)  //  如果第一行、第三根列线上有键按
                    x=2;                 //下，则键值为 2
       else
        {
          P1OUT=0X2;                    //扫描第二行
          if((P1IN&0X70)==0X10)         //如果第二行、第一根列线上有键按下，则键值为 3
             x=3;
             else
               if((P1IN&0X70)==0X20)              //如果第二行、第二根列线上有键按
                  x=4;                            //下，则键值为 4
```

```
                    else
                      if((P1IN&0X70)==0x40)        //如果第二行、第三根列线上有键按
                        x=5;                       //下，则键值为 5
            else
            {
             P1OUT=0X4;                            //扫描第三行，以下与上相同
             if((P1IN&0X70)==0X10)
             x=6;
               else
                 if((P1IN&0X70)==0X20)
                 x=7;
                   else
                   if((P1IN&0X70)==0x40)
                     x=8;
              else
              {P1OUT=8;                 //扫描第四行
              if((P1IN&0X70)==0X10)
        x=9;
        else
            if((P1IN&0X70)==0X20)
              x=10;
                  else
                    if((P1IN&0X70)==0x40)
                      x=11;
                      }
              }
          }
  return(x);
  }
```

C　等待按键松开

这与独立式键盘一样，反复调用判键子程序，直到判断结果为没有按键按下为止。程序略，与前相同。

D　完整的扫描式键盘程序

上面所讲述的只是键盘程序的一些详细细节与实用键盘子程序，完整的键盘程序则有很多中方式。图 2.6.3 所示的键盘流程图是工作在查询方式的键盘程序，可以看出，采用此方式时，MCU 一直在查询有没有按键被按下，MCU 不能做其他的事情，只有等按了按键之后，才会有时间处理相应的事务。这种方式 MCU 的效率肯定低下。在实际应用中，键盘只是输入部分。它的功能是送给 MCU 用户的输入。所以为了提高 MCU 的效率，所以为了提高 MCU 的效率，常使用中断的方式实现键盘输入。

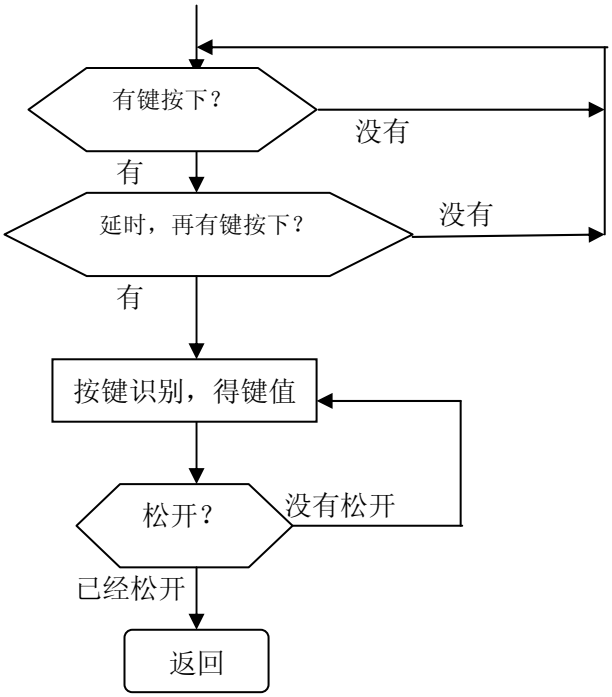


图 2.6.3　查询式键盘程序

中断方式的键盘实现又有两种形式：直接 I/O 口中断与定时器中断。它们的流程图分别见图 2.6.4 与图 2.6.5。

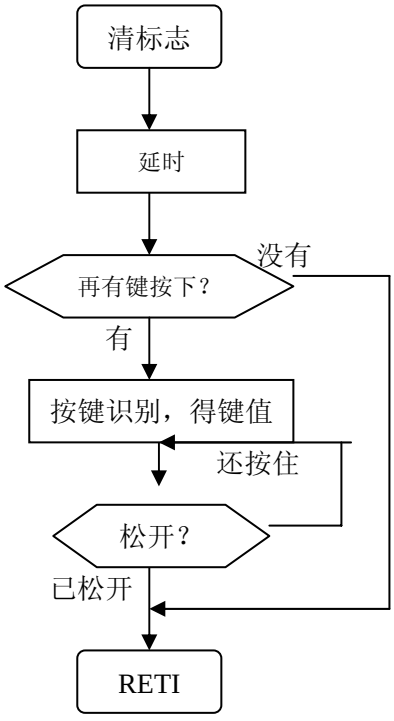


图 2.6.4　I/O 口中断方式键盘流程图

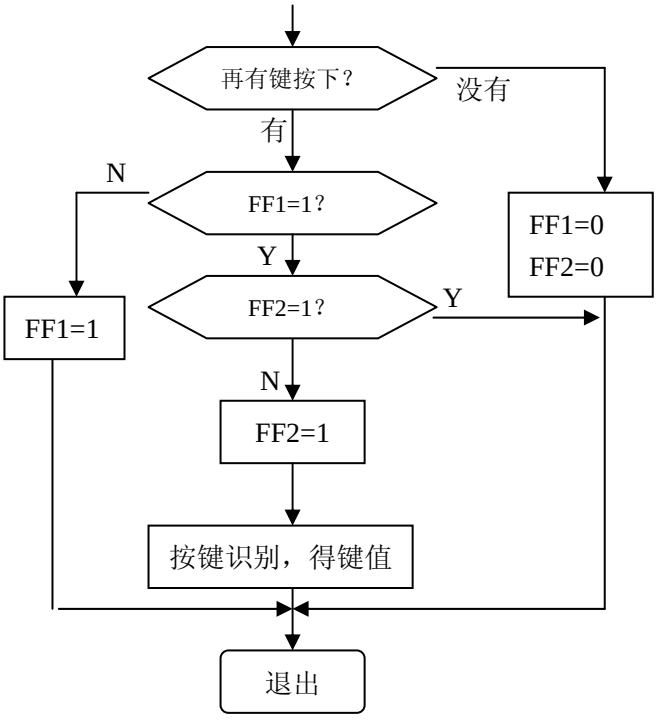


图 2.6.5　定时器中断方式键盘流程图

两流程框图的差别在于，使用 I/O 口线中断方式有延时消除抖动的过程，而使用定时器中断查键盘则没有延时去抖动细节。因为定时器的定时时间就是消抖动的时间，比如 10 毫秒。首先设置两个标志位 FF1、FF2，其中 FF1 用于去抖动标志，FF2 用于按键识别完成标志。初始值都是 0，表示没有去抖动与没有完成按键的识别。当进入中断时，如果 FF1 为 0，则没有消抖动，那么判断是否有键按下，如有，设置 FF1=1，并退出程序，等待下次中断（等 10 毫秒）。如果再次中断，则继续判断是否有键按下，如有，说明是真正的按键被按下，而且已是键盘的稳定期，调键码识别子程序得到键值，设置 FF2=1，表示完成了按键的识别；如果没有按键被按下，则设置 FF1=0，退出。如果这时再次中断，显然都是一次按键，FF2=1 同时表明没有松开按键，也就退出了。等到按键被松开时，设置 FF1=0、FF2=0，为再次按键作准备。

使用 I/O 口中断方式键盘程序如下(使用定时器中断方式键盘程序请读者自己编写)：

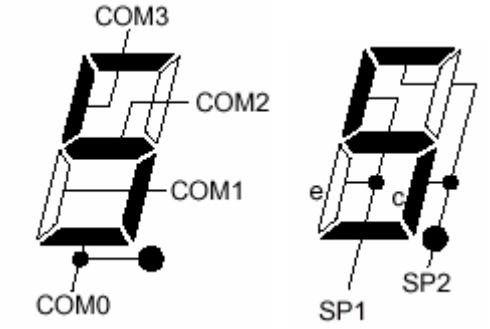
```
P1KEY_INT  CALL   # KEYJUDGE     ；判断是否有按键
           JNC    KEYEND         ；没有则退出
           CALL   #DELAY10MS     ；如有，则延时 10 毫秒          消抖动
           CALL   # KEYJUDGE     ；再判键
           JNC    KEYEND         ；如没有按下则退出
           CALL   #KEYCODE       ；如有，则调认键程序得到键值
           PUSH   R5             ；保护键值
KEYLOOP    CALL   # KEYJUDGE     ；等待按键松开
           JC     KEYLOOP        ；没有松开，则继续等待
           POP    R5             ；按键松开之后，恢复键值
KEYEND     RETI
```

2．7　液晶显示原理与应用

MSP430 单片机的很多系列都有液晶驱动的能力，比如：41 系列、42 系列、43 系列、44 系列、31 系列、32 系列、33 系列等都可以直接驱动液晶。数码管显示可能读者非常熟悉，直接使用单片机的引脚或使用功率型接口芯片即可驱动；而液晶显示需要特殊的电压，一般单片机的口线（I/O）不能直接驱动液晶。而上述所列举的 430 系列单片机能直接驱动液晶，本文将以 430F43X、MSP430F44X 为例（具体型号为 MSP430F435）说明在 MSP430 中如何使用液晶显示。

2.7.1　SP430 液晶显示原理

MSP430 的液晶显示有静态、2MUX、3MUX、4MUX 四种显示模式，而最常用还是 4MUX 模式。通俗讲，就是有四个公共端（相当于数码管扫描显示的位选端），若干个驱动端的模式。这种模式的最大优点就是能使用最少的引脚提供最多的液晶显示段。图 2.7.1 表示了 4MUX 的显示原理。其中（a）说明了一个“8”字的四个公共端、 (b)说明了两个驱动端，当分别给公共端与驱动端液晶信号时，就显示对应的数码。



(a)　　(b)

图 2.7.1　4MUX 显示模式下的公共端与驱动端

本文将以 4MUX 方式为例讲述液晶应用。在这种方式下，一个“8”字的显示只需要给“COM0、COM1、COM2、COM3、SP1、SP2”对应的信号。而 COM0、COM1、COM2、COM3 公共信号，所有的“8”字都需要与它们相连接。故真正表示一个“8”字的显示的就只与 SP1、SP2 有关系了。所以在 4MUX 方式下，只需要两个驱动端即可表示一个“8”字的显示。

在 MSP430 系列能驱动液晶显示的单片机中，专门开辟了一片存储空间（LCDMEM1~LCDMEM20）存放要显示的信息，被称为液晶显示缓存，简称液晶显存。MSP430F435 一共有 20 字节单元液晶显存，如果使用 4MUX 方式显示，可以显示 160 段液晶笔画。这时，每个显存将对应两个驱动端。图 2.7.2 表示了在 4MUX 方式下的液晶显存、液晶显示、液晶驱动端之间的对应关系。

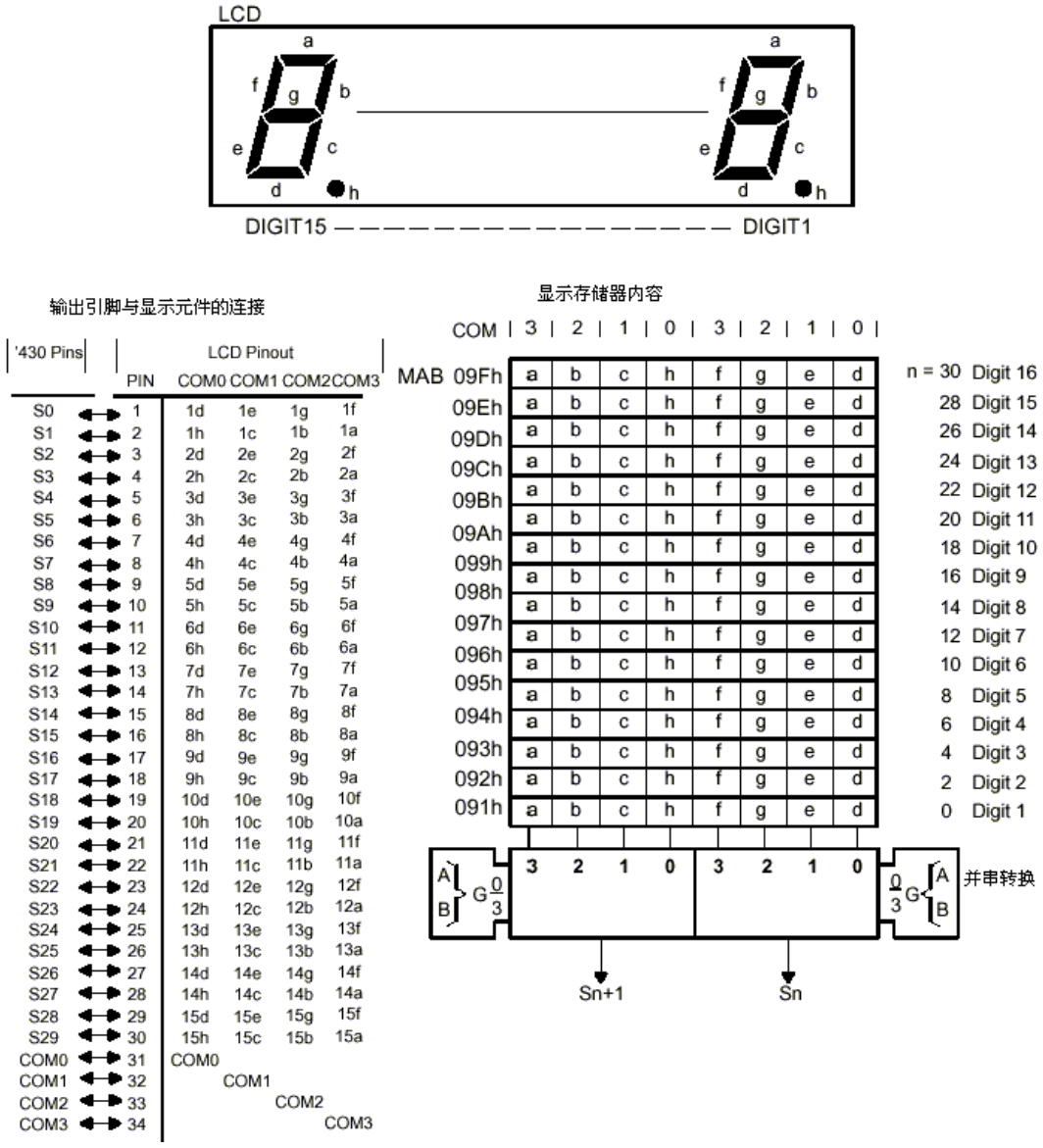


图 2.7.2　　4MUX 方式下的液晶显示原理

这时要显示“1234567890”则送液晶显存数据为“0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f,”即可。如果液晶引脚的 S0、S1、S2、S3 分别连接在单片机的 S0、S1、S2、S3，则给 LCDMEM1，LCDMEM2 送数据“0x3f,、0x06”液晶上就显示“0、1”字符。

2.7.2　液晶简介

此液晶一共 160 段，见图 3。上面为图标与 6 个“米”字，可以显示数字与英文字母，用来表达一些指示性含义的符号。下面一排为 7 个“8”字与符号，显示数字，可以有具体的量纲含义数据。

图 2.7.3 与图 2.7.4 为此液晶“8”字、“米”字各笔段的定义。

表 1 为此液晶引脚真值表。

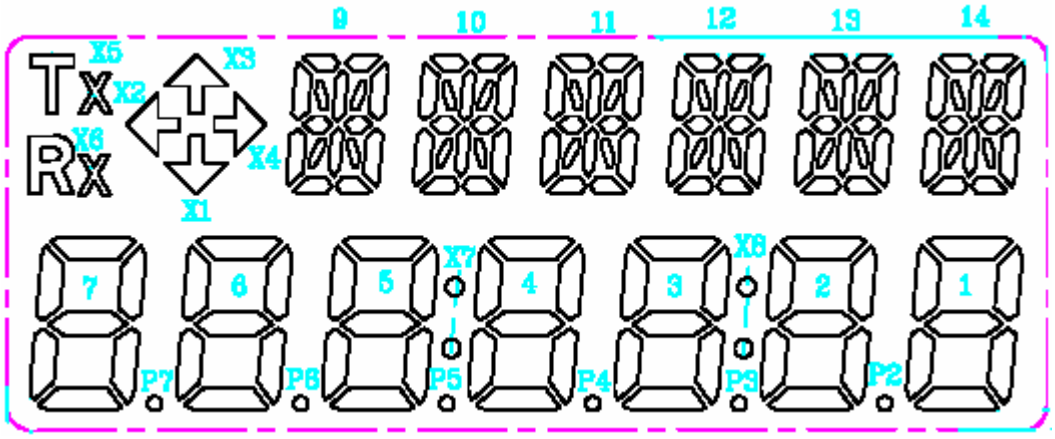


图 2.7.3　　液晶段码布局

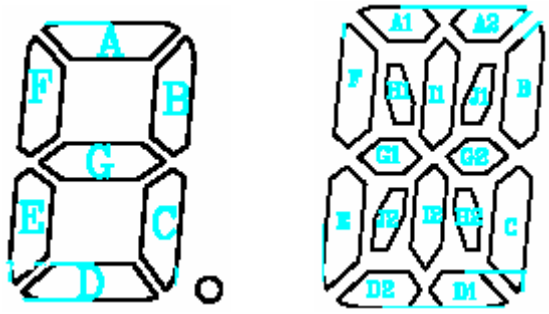


图 2.7.4　　“8”字、“米”字笔段定义

PIN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
COM1	1D	P1	2D	P2	3D	P3	4D	P4	5D	P5	6D	P6	7D	P7	X6
COM2	1C	1E	2C	2E	3C	3E	4C	4E	5C	5E	6C	6E	7C	7E	X7
COM3	1B	1G	2B	2G	3B	3G	4B	4G	5B	5G	6B	6G	7B	7G	X5
COM4	1A	1F	2A	2F	3A	3F	4A	4F	5A	5F	6A	6F	7A	7F	X8

PIN	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
COM1	X2	COM1				9F	9A1	9J1	9A2	10F	10A1	10J1	10A2	11F	11A1
COM2	X3		COM2			9E	9H1	9I1	9B	10E	10H1	10I1	10B	11E	11H1
COM3	X4			COM3		9J2	9G1	9G2	9C	10J2	10G1	10G2	10C	11J2	11G1
COM4	X1				COM4	9D2	9I2	9H2	9D1	10D2	10I2	10H2	10D1	11D2	11I2

PIN	31	32	33	34	35	36	37	38	39	40	41	42	43	44	
COM1	11J1	11A2	12F	12A1	12J1	12A2	13F	13A1	13J1	13A2	14F	14A1	14J1	14A2	
COM2	11I1	11B	12E	12H1	12I1	12B	13E	13H1	13I1	13B	14E	14H1	14I1	14B	
COM3	11G2	11C	12J2	12G1	12G2	12C	13J2	13G1	13G2	13C	14J2	14G1	14G2	14C	
COM4	11H2	11D1	12D2	12I2	12H2	12D1	13D2	13I2	13H2	13D1	14D2	14I2	14H2	14D1	

表 2.7.1　　引脚真值表

2.7.3　硬件连接

硬件连接最简单，只需要 3 只电阻、一只晶体。然后将液晶的公共端与单片机的公共端；液晶的各驱动段与单片机对应连接就可以了。具体电路如图 2.7.5 所示（MSP430F435）。

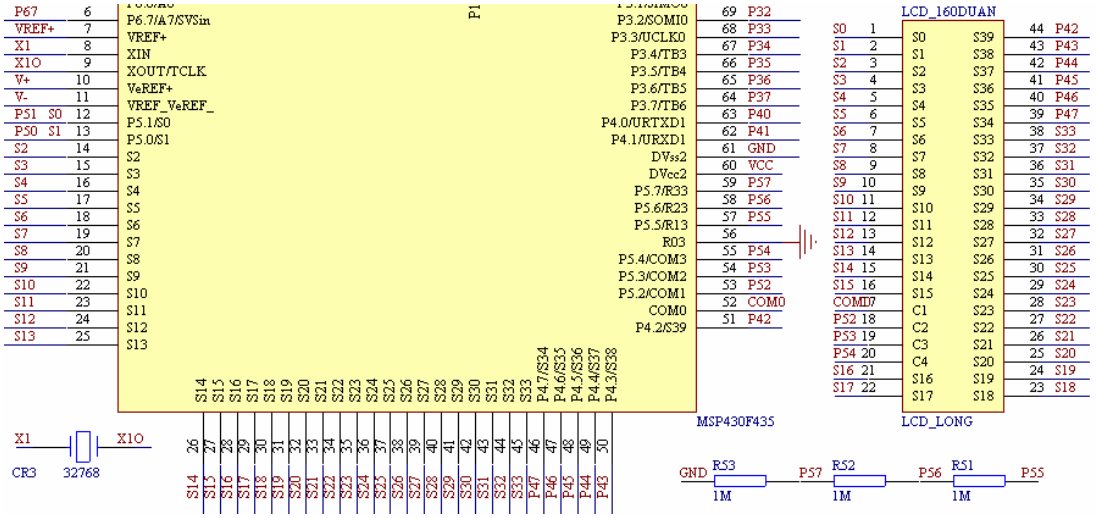


图 2.7.5　液晶显示电路

2.7.4　如何显示你的需要

有此液晶，你可以在 430 的基础上充分发挥你的想象力，流畅地在电子设计中表达你的智慧。这里以设计大家非常熟悉的 TI　430 日展示手表为例，说明液晶的应用。同时你会发现这块液晶的应用将给手表的使用者带来方便与非常直观的显示。

手表的基本（必备的）功能：走时间（见图 2.7.6）、时间校准等；

手表的扩展（可选的）功能：闹钟、系列时间控制器（见图 2.7.7）等。

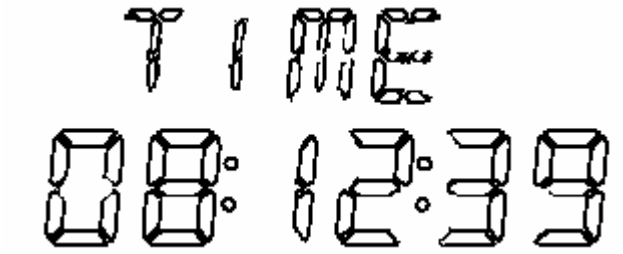


图 2.7.6　显示当前时间为 08：12：39

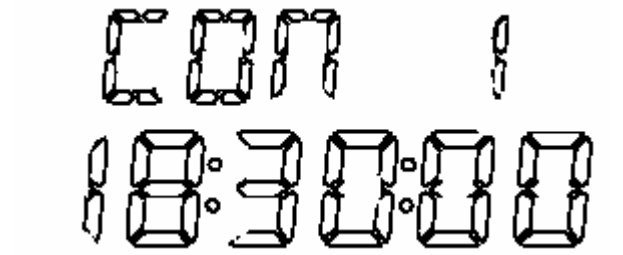


图 2.7.7　显示到达第一个控制时间点 18：30：00

图 2.7.6 上排显示“TIME”，表示下面的数字表示时间；图 2.7.7 上排显示“CON　1”表示下面的数字含义为控制时间点 1 已经到达。下面分析如何将这些信息显示在液晶上。

最简单的方法为：将仿真器与带有段码液晶的目标电路板连接；然后进入调试环境，这时，打开 SFR 窗口中的 LCD 显示缓存；修改 LCDCTL 的值为 0xfd；再修改任意显示缓存的

值，比如 LCDMEM4，如果输入 0，则液晶最下排的中间数字显示马上消失，因为送的显示值为 0，则各段均不显示，这时再修改 LCDMEM4 的值为 1，则液晶最下排的中间数字的笔段 A 显示，如果改为 0x25，则液晶最下排的中间数字显示笔段 A、C、F 显示。

所以数字 0~9 显示时使用这样的段码表：

unsigned char lcd_seg[]={0x3f,0x06,0x5b,0x4f, 0x66 , 0x6d, 0x7d , 0x07，0x7f,0x6f}；

而显示最上面一排符号时，如果要经常使用很多符号也可以使用段码表，如果显示符号不是频繁，也可以不使用段码表，而直接在需要的地方临时翻译段码即可。这时也可以使用上面的方法：比如要在第九个符号的位置（左边第一个“米”字）显示“T”，则修改 LCDMEM9=90H、LCDMEM10= 12H ，则显示“T”。其他依此类推。

2.7.4　程序举例（实验 8）

此示例程序的运行，液晶显示将如图 2.7.6 所示。

首先要设置液晶显示需要的参数：刷新时钟频率、液晶显示打开、液晶显示模式等。

C 语言语句：　　LCDCTL = 0XFD;

汇编语句：　　MOV.B　#0FDH,　&LCDCTL

然后将各显示数据写入显存即可，这里对数字的写入使用循环语句。

```
#include   <msp430x44x.h>
unsigned char lcd_seg[]={0x3f,0x06,0x5b,0x4f,  0x66,0x6d,0x7d,0x07,  0x7f,0x6f };
unsigned char lcd_data[ ]={ 0, 8, 1, 2, 3, 9};
void main(void)
    {
    int I;
    WDTCTL = WDTPW + WDTHOLD;        //    必须的，因为默认为打开
    LCDCTL = 0xfd;
    LCDMEM[7]=0xa;            //显示两个表示时钟的”冒号”
    LCDMEM[8]=0x90;           //以下 8 句显示  TIME
    LCDMEM[9]=0x12;
    LCDMEM[10]=0x80;
    LCDMEM[11]=0x2;
    LCDMEM[12]=0x93;
    LCDMEM[13]=0x72;
    LCDMEM[14]=0x5b;
    LCDMEM[15]=0x94;
    For ( I=0;I<7;I++)                    //以下语句显示数组 lcd_data 中的数字
        LCDMEM[I] = lcd_seg[lcd_data[I]];
   for (;;)
      {
      _BIS_SR(SCG1+SCG0+CPUOFF);        // 低功耗
      _NOP();
      }
    }
```

2.8　　数码管显示设计与应用

数码管显示在日常生活中应用非常多，比如洗衣机、微波炉、电冰箱、空调机等，这些显示多使用数码管，当然也有使用液晶显示的。MSP430F449 的端口非常多，与数码管接口可以使用直接口线连接，又可以使用芯片扩展，它们各有利弊，使用芯片扩展将使用少量的口线，这里重点讲解如何使用芯片扩展。

2.8.1　　数码管的原理

数码管为 8 只发光二极管按照一定规则排列构成。如图 2.8.1 所示，图中的 abcdefg“.”为数码管的 8 只发光二极管的排列规则，由此可见：当这些发光二极管由不同的显示组合可以得到不同的数字显示。

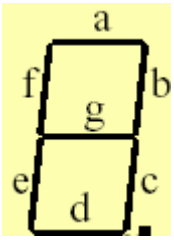


图 2.8.1

如果要显示数字“2”，如果 HGFEDCBA 对应一个字节的高到低位

则显示码为：HGFEDCBA

0 10 11 0 1 1　　0x5b

这样，每个数字就对应一个显示码。一般地，先将常用的显示数据对应的显示码放在数组中，然后在数组中查表求得对应的显示码，再将对应的发光二级管显示。

2.8.2　　使用 74HC373 扩展数码管显示（实验 9）

电路图见图 2.8.2。

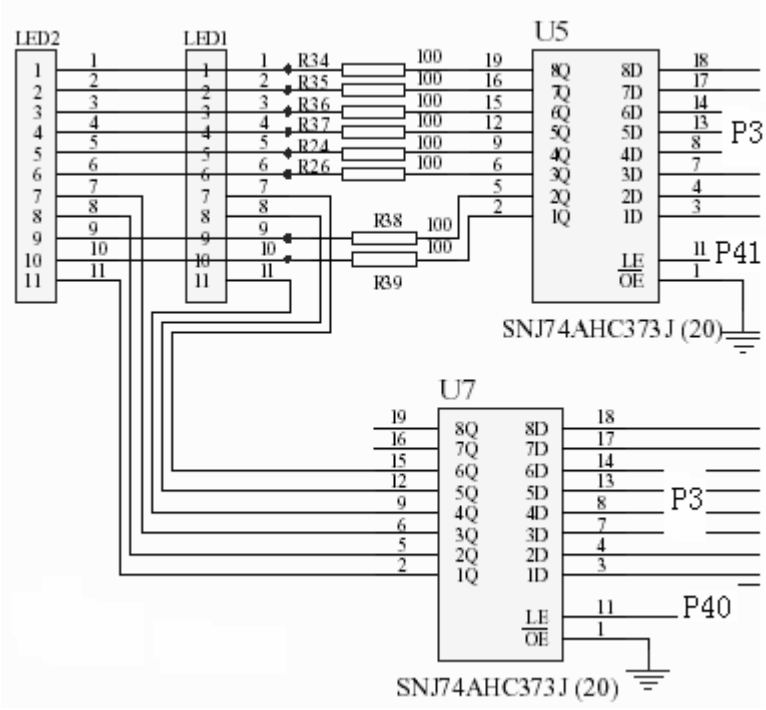


图 28.2　74HC373 扩展的数码管显示电路

硬件连接：

6 只数码管的显示电路，通过两片 373 扩展，其中 U5 为 8 位段码输出，U7 为 6 只数码管的位选信号。也都连接在 P3 端口上。两片 74HC373 的输出由 P41、P40 数据锁存。

扫描显示软件分析：

扫描显示的原理在于利用人眼睛的视觉暂停，让每个数码管只显示一点时间，所有的数码管轮流显示，而人眼睛看起来就象所有的都在显示一样。所以硬件上所有的数码管的段码端都连接在一起，而每一个数码管的公共端（地）不连接在一起，而由 U7 选中每个是数码管。

所以软件上，很明显，分几个步骤。

第一、将要显示的数码转换为段码。可使用查表的方式。比如要显示“1、2、3、4、5、6、7、8”分别在 8 个数码管上，首先安排段码表，设置一个数组 seg[]：

unsigned char seg[]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71}

则 5 的段码为 seg[5]。

第二、将要显示的段码输出，这里使用 U5 输出。

第三、每输出一个要显示的段码，则使用 U7 选中应该显示的数码管。

第四、延时一小段时间。这个时间不能长，也不能短。太长则 8 只数码管看起来很抖动，太短则 8 只数码管一片模糊。

第五、循环第一到第四。

具体程序编写。

1　数据显示码表

```
const unsigned char NUM_LED[20]=
                        {0xd7,0x14,0xcd,0x5d,0x1E,   // 0 ~ 4
                        0x5b,0xdb,0x15,0xdf,0x5f,    // 5 ~ 9
                        0x9f,0xda,0xc3,0xdc,0xcb,    // a ~ e
                        0x8b,0x00,0x40,0x8
                        };                          //f,0x00 使 LED 不显示
```

2　延时程序是必须的。

```
void delay(int v)
    {
        while(v!=0)v--;
    }
```

3　端口初始化

```
/************************************************************
*       端口初始化
************************************************************/
void init_LED(void){
    char tmpv;
    P3DIR   = 0xff;      // 设置 p3 输出
    P3OUT   = 0x00;      // 设置 初始值为 0
    P4DIR |= 0x03;       // 设置 p4.0,p4.1 输出
    P4OUT &= 0xfc;       // 设置初始值
    for(tmpv=0;tmpv<LED_IN_USE;tmpv++)
    {                    // 初始化缓冲区
      led_Buf[tmpv] = 0x13;
    }
```

4　扫描显示

```
/*********************************************************
*       LED 显示   ,该函数可以放到定时器中断中
*********************************************************/
void led_Display(){
        unsigned tmp ,i ;
        tmp      = 0x01;
        for(i=0;i<6;i++)
        {
        P3OUT   = NUM_LED[led_Buf[i]];  // 设置显示值
        P4OUT |= 0x02;                  // 打开数据锁存器
        P4OUT &= 0XFD;                  // 关闭数据锁存
        P3OUT   = ~(tmp<<i);            // 设置那只 LED 显示
```

```
            P4OUT   |= 0x01;                                  // 打开控制锁存
            P4OUT   &= 0XFE;                                   // 关闭控制锁存
            delay(300);
            }
    }
```

要显示的数据放在数组 disbuffer[ ]中。数组的第一个数据对应第一个数码管。

```
    void disp(void)
     {
     unsigned char i=0;
     unsigned char temp_wei=0x04,temp_duan=0 ;
     P1DIR   =   0x1f ;
     for(i=0;i<8;i++)
        {
          P1OUT   &=   ~BIT3 ;
          temp_duan=seg[disbuffer[i]] ;
          for(j=0;j<8;j++)
                {
                 if(temp_duan&0x80)       P1OUT   |=     BIT0 ;
                 else         P1OUT   &=   ~BIT0;
                 temp_duan=temp_duan<<1;
                 P2OUT   &=     ~BIT0;     P2OUT   |=   BIT0;
                }
           P1OUT   =   (P3IN&0xf8) | temp_wei;
           P1OUT   |=     BIT3;
           temp_wei++;
           delay(80);
        }
      for(i=0;i<8;i++)
      { P1OUT   &=   ~BIT0;     P2OUT   &=     ~BIT0;     P2OUT   |=   BIT0;
      }
     P1DIR=0XF;
     P1OUT=0XF;
     P1IFG=0;
     P1IE=0xf0;
     }
```

5　主程序

```
     /*****************************************/
     void main(void)
     {
```

```
    char i=1,j;
    unsigned tmpv ;
     WDTCTL = WDTHOLD + WDTPW;          //关闭看门狗
     init_LED();
     while(1)
     {                                          //循环
         for(tmpv=0;tmpv<LED_IN_USE;tmpv++)
         {                              // 初始化缓冲区
            led_Buf[tmpv] = i;
         }
         i++;
         if(i==20)
             i=0;
        for(j=0;j<100;j++)
             led_Display();
     }
}
```

2.8.3　使用 74HC164 与 74HC138 扩展数码管显示（实验 10）

硬件电路见 2.8.3

8 只数码管的显示电路，通过 164 串行移位输出 8 位段码，138 位 8 只数码管的位选信号。也都连接在 P1 端口上。可以发现同样的显示需要的口线比前面的少。

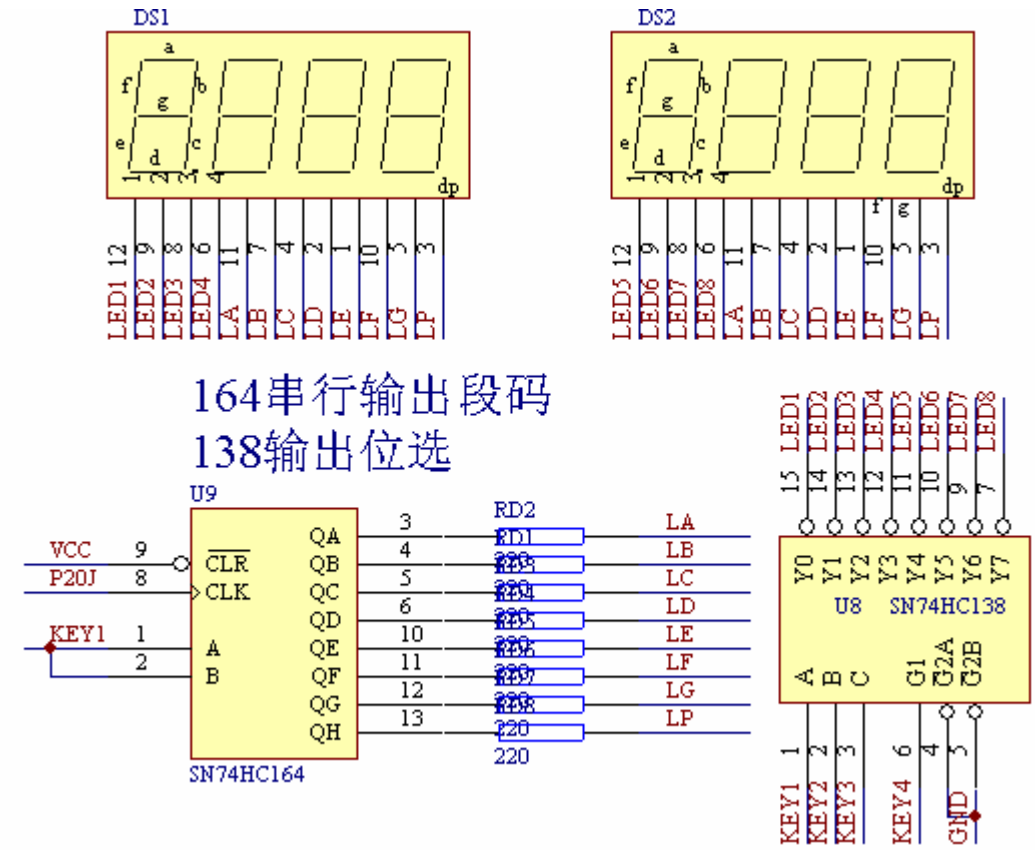


图 2.8.3　74HC164、74HC138 扩展的扫描显示

软件设计与前面基本相同，分几个步骤：

第一、将要显示的数码转换为段码。可使用查表的方式。比如要显示“1、2、3、4、5、6、7、8”分别在 8 个数码管上，首先安排段码表，设置一个数组 seg[]：

unsigned char seg[]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71}

则 5 的段码为 seg[5]。

第二、将要显示的段码输出，这里使用 164 移位输出。

第三、每输出一个要显示的段码，则使用 138 选中应该显示的数码管。

第四、延时一小段时间。这个时间不能长，也不能短。太长则 8 只数码管看起来很抖动，太短则 8 只数码管一片模糊。

第五、循环第一到第四。

显示程序如下：（主程序略）

```
void disp(void)
 {
 unsigned char i=0;
 unsigned char temp_wei=0x04,temp_duan=0 ;
 P1DIR    =    0x1f ;
```

```
for(i=0;i<8;i++)
  {
    P1OUT  &=  ~BIT3 ;
    temp_duan=seg[disbuffer[i]] ;
    for(j=0;j<8;j++)
        {
         if(temp_duan&0x80)      P1OUT  |=    BIT0 ;
         else        P1OUT  &=  ~BIT0;
         temp_duan=temp_duan<<1;
         P2OUT  &=    ~BIT0;    P2OUT  |=  BIT0;
        }
     P1OUT  =  (P3IN&0xf8) | temp_wei;
     P1OUT  |=    BIT3;
     temp_wei++;
     delay(80);
  }
 for(i=0;i<8;i++)
 { P1OUT  &=  ~BIT0;    P2OUT  &=    ~BIT0;    P2OUT  |=  BIT0;
 }
P1DIR=0XF;
P1OUT=0XF;
P1IFG=0;
P1IE=0xf0;
}
```

2．9　在数码管上显示键值（实验 8）

此为基本综合实验，将显示与键盘结合，关键在于主程序的编写。请读者自己编写。

2．10　在液晶上显示键值（实验 11）

此为基本综合实验，将显示与键盘结合，关键在于主程序的编写。请读者自己编写。

2．11　MSP430 定时器的使用

MSP430 系列单片机的定时器相当丰富。有基本定时器 BT、定时器 TA、定时器 TB、看门狗定时器 WDT 等。其中看门狗主要用于程序的完善性控制等方面；基本定时器除了可以用于定时之外，还可以用于液晶显示的时序控制；TA、TB 基本相同，而且功能复杂，这里将详细讲解，并使用之实现时钟。

看门狗的目的在于阻止程序跑飞，其原理在于：看门狗定时器设置一定时时间，比如 250 毫秒，这个时间是所有用户程序一定能在此时间内执行完该程序的一个时间，设置好这个定时时间之后，所有用户程序就必须在这个设定的时间内将看门狗计数器的值清零，使计数器重新计数，如果 CPU 执行程序正确，则看门狗计数器始终能在规定的时间内被用户程序清零而始终不能计数到 250 毫秒，而当 CPU 执行程序跑飞（PC 值指向用户程序以外），看门狗计数器得不到用户程序清零，能计数到 250 毫秒，发生溢出，导致 CPU 复位，这样 CPU 又会重新运行用户程序。所以使用看门狗时，用户软件必须周期性地在 WDTCTL 的 CNTCL 位写“1”，使得看门狗计数器复位以防止其超过设定的定时时间。

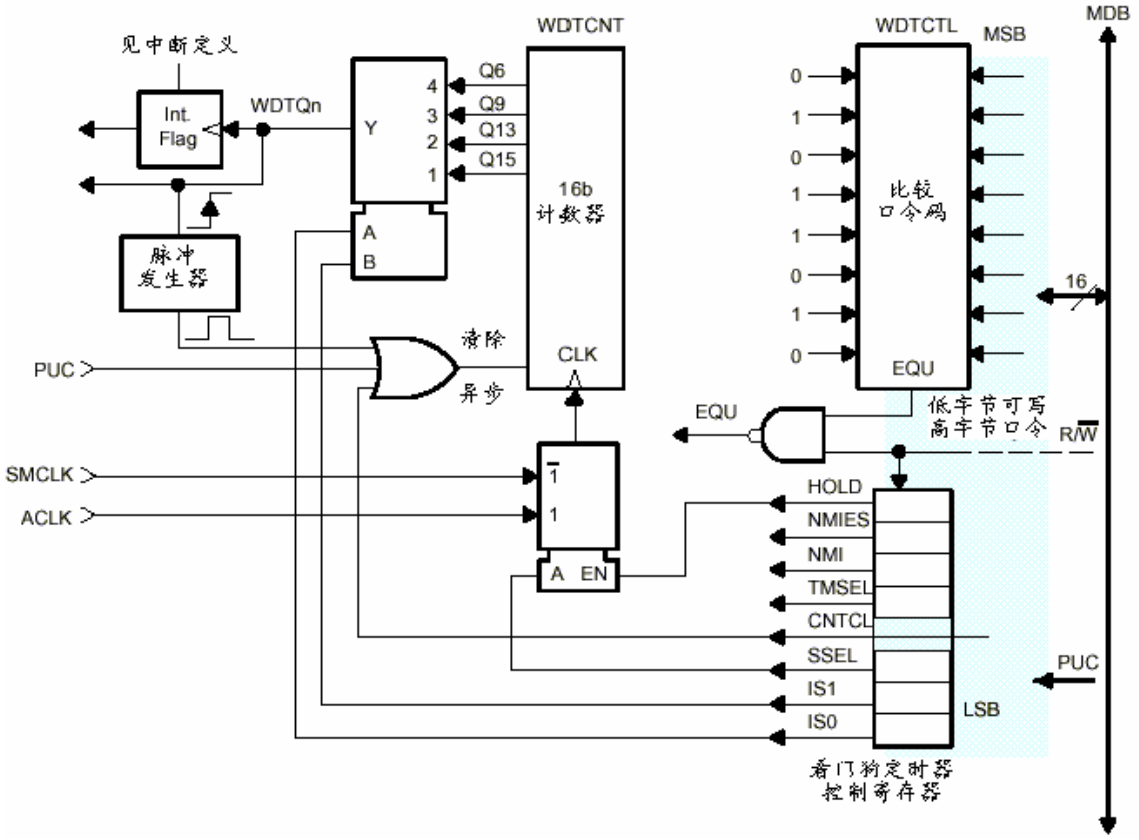


图 2.11.1　看门狗定时器的结构原理

BT 是 MSP430X3XX、MSP430F4XX 系列器件中的模块。它也是一个定时器，它通常向其它外围模块提供低频控制信号。BT 可以是两个 8 位定时器也可以是一个 16 位定时器，它有两个计数单元（BTCNT1、BTCNT2）与一个控制单元（BTCTL）。通过控制寄存器 BTCTL 的设置，用户可以方便地使用 BT。

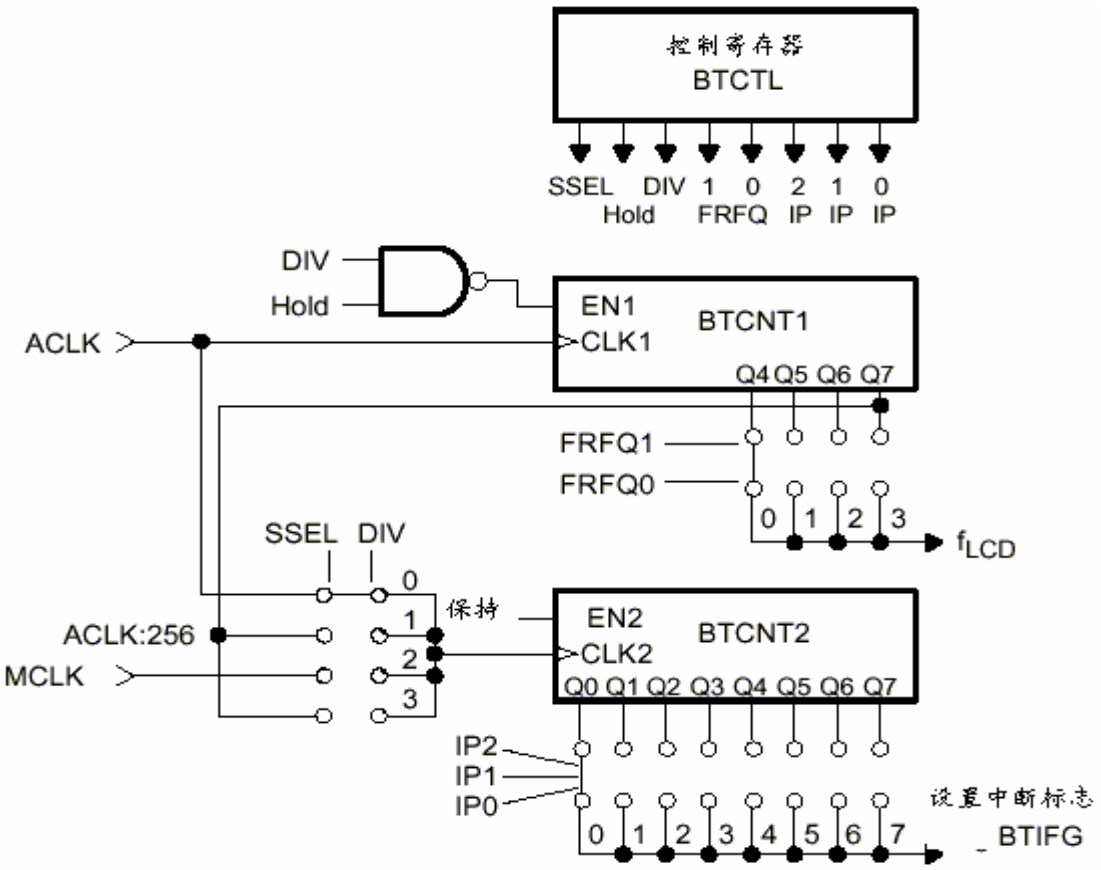


图 2.11.2　　定时器 BT 的结构原理

TA、TB 基本相同，这里将详细讲解定时器 A

定时器 A 是 MSP430 所有系列都有的模块，是一个用途非常广泛的通用 16 位定时/计数器。它有以下一些特点：

16 位计数器，四种工作模式；

多种可选的计数器时钟源；

多个具有可配置输入端的捕获/比较寄存器；

有 8 种输出模式的多个可配置的输出单元；

Timer_A 可支持同时进行的多种时序控制、多个捕获/比较功能、多种输出波形（PWM），也可以是几种功能的组合，每个捕获/比较寄存器可以以硬件方式支持实现串行通讯。

Timer_A 具有中断能力。中断可由计数器溢出引起，也可来自具有捕获或比较功能的捕获/比较寄存器。每个捕获/比较模块可独立编程，由捕获或比较外部信号以产生中断，外部信号可以是上升沿，也可是下降沿，也可都是。

在不同的 MSP430 器件中，Timer_A 模块中的捕获/比较器的数量不一样，比如在 MSP430F435 中 Timer_A 模块含有 3 个捕获/比较器（简称 CCR），因此也经常称 Timer_A3，表示该模块含有 3 个 CCR。还是先看看 Timer_A 的结构原理图。见图 2.11.3。图中，可以将 Timer_A 分解成几个部分：计数器部分、捕获/比较寄存器、输出单元。其中计数器部分完成时钟源的选择、分频，模式控制，计数等功能。捕获/比较器用于捕获事件发生的时间或产生时间间隔。输出模块用于产生用户需要的输出信号。

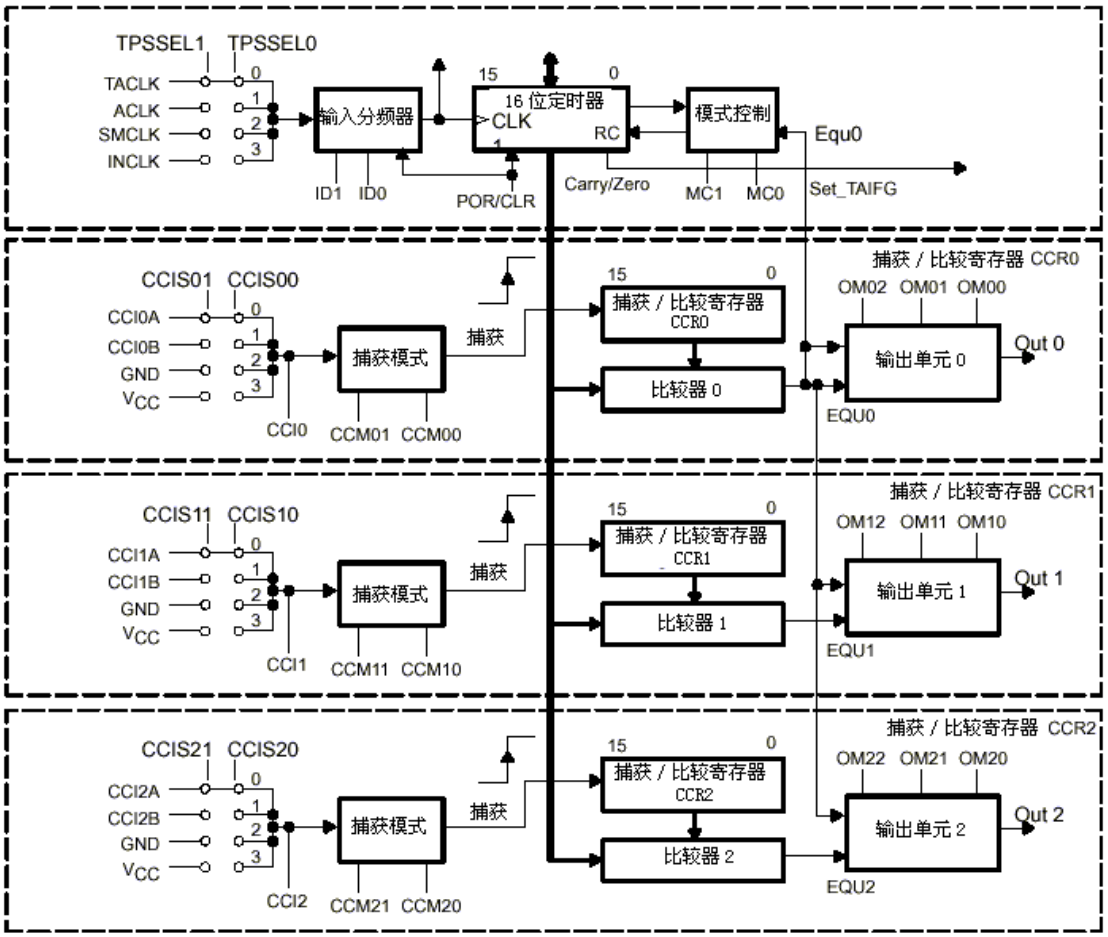


图 2.11.3　定时器 A 的结构原理

定时器 A 的寄存器在 IAR 的调试环境中见图 2.11.4 所示。操作这些寄存器就可以实现 TA 的所有功能。

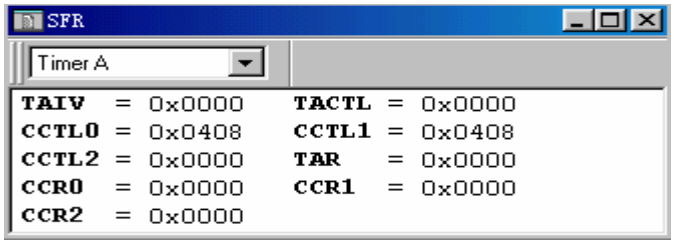


图 2.11.4　TA 的寄存器

其中 TACTL 为最主要的控制寄存器，它决定 TA 的输入时钟信号、TA 的工作模式、TA 的开启与停止、中断的申请等工作。TACTL 寄存器为 16 位寄存器，必须使用字指令对其访问。该寄存器在 POR 信号后全部复位，但在 PUC 信号后不受影响。该寄存器中各位的含义：

15~10	9	8	7	6	5	4	3	2	1	0
未 用	SSEL1	SSEL0	ID1	ID0	MC1	MC0	未用	CLR	TAIE	TAIFG

SSEL1、SSEL0　选择输入分频器的输入时钟源。

SSEL1	SSEL0	输入信号	输入信号说明
0	0	TACLK	使用外部引脚信号作为输入（见手册）

0	1	ACLK	辅助时钟
1	0	MCLK	系统主时钟
1	1	INCLK	外部输入时钟（见芯片手册）

ID1、ID0　　选择输入分频器的分频系数。

00：　直通，不分频

01：　1/2 分频

10：　1/4 分频

10：　1/8 分频

MC1、MC0　　选择定时器模式

00：　停止模式，用于定时器暂停

01：　增计数到 CCR0 模式，该模式下，计数器计数到 CCR0，再清零计数

10：　连续增计数模式，计数器增计数到“FFFFH”，再清零计数

11：　增/减模式，增计数到 CCR0，再减计数到 0

CLR　　定时器清除位，计数器内容清零

TAIE　　中断允许位，该位允许定时器溢出中断

TAIFG　　定时器溢出标志位，在不同的定时器模式下，该位置位条件不一样。

增计数模式：　当定时器由 CCR0 计数到“0”时 TAIFG 置位

连续模式：　当定时器由 0FFFFH 计数到“0”时 TAIFG 置位

增/减模式：　当定时器由“1”减计数到“0”时 TAIFG 置位

由此可见 TACTL 寄存器几乎控制了 Timer_A 的第一部分——计数器部分。下面我们对照原理图理解。

工作模式控制如图 2.11.5。由 MC1、MC0 两位选择。

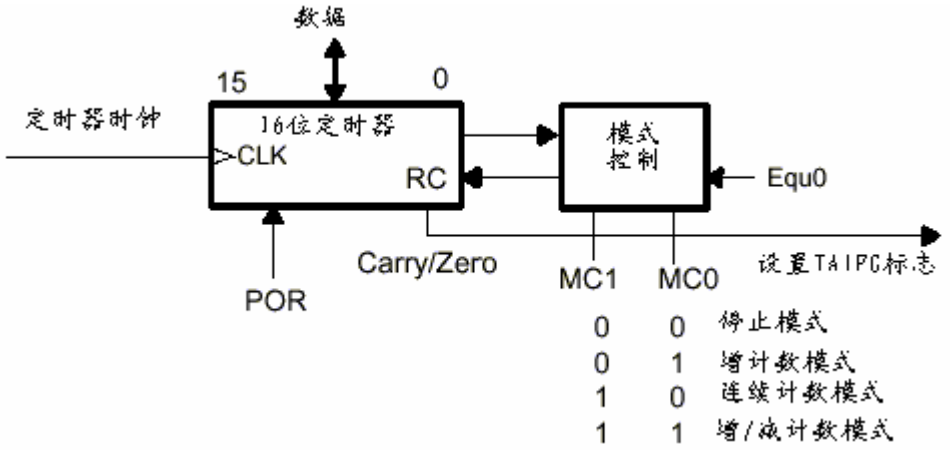


图 2.11.5　　模式控制原理图

计数器输入时钟源的选择与分频控制如图 2.11.6。由 SSEL0、SSEL1 两位选择时钟源，然后再由 ID0、ID1 选择分频系数将输入信号分频，分频后的信号才用于计数器计数。在 MSP430F4XX 系列器件中，INCLK 信号经过反向驱动之后再送入，这一点与其他器件有点差别。

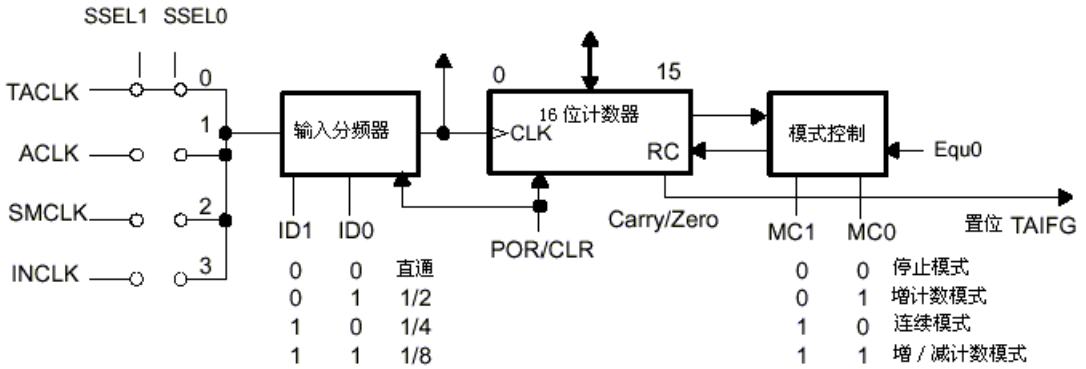


图 2.11.6　　计数器输入时钟源的选择与分频控制

Timer_A 定时器共有 4 种工作模式。由控制寄存器 TACTL 中 MC0、MC1 两位决定（祥见 TACTL 寄存器介绍）。

停止模式

当 MC1=MC0=“0”时，定时器暂停。暂停时定时器的值（TAR 的内容）不受影响。当定时器在暂停后重新计数时，计数器将从暂停时的值开始以暂停前的计数方向计数。如果不能这样，则可通过 TACTL 中 CLR 控制位来清除定时器的方向记忆特性。

增计数模式

当 MC0=“1”，MC1=“0”时定时器工作在增计数模式。该模式用于定时周期小于 65536 的连续计数方式。捕获/比较寄存器 CCR0 的数据定义定时器的计数周期。

增计数模式的计数器活动规则：当计数器 TAR 增计数到 CCR0 的值或当计数值与 CCR0 相等（或定时器值大于 CCR0 的值）时，定时器复位并从“0”开始重新计数。图 2.11.7 说明了增计数模式的计数过程。当定时器的值等于 CCR0 的值时，设置标志位 CCIFG0 为“1”，而当定时器从 CCR0 计数到“0”时，设置标志位 TAIFG 为“1”。

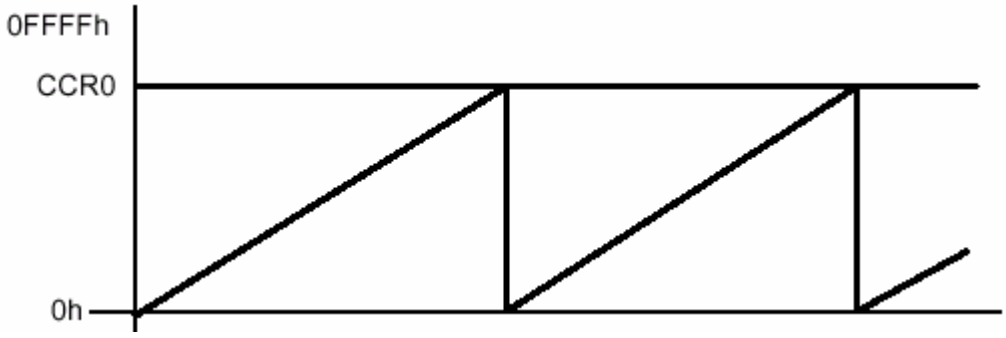


图 2.11.7　　增计数模式的定时器 TAR

连续计数模式

在需要 65536 个时钟周期的定时应用场合常用连续模式。其典型的应用是产生多个独立的时序信号。在这种计数模式中，CCR0 的工作方式与其他比较寄存器的工作方式相同。利用捕获/比较寄存器与各输出单元的输出模式可以捕获各种外部事件发生的定时器数据（事件发生的时间）或者产生不同类型的输出信号。

连续模式的计数器活动规则：定时器从它的当前值开始计数，当计数到 0FFFFH 后又从“0”开始重新计数，如图 2.11.8, 当定时器从“0FFFFH”计数到“0”时，设置标志位 TAIFG。

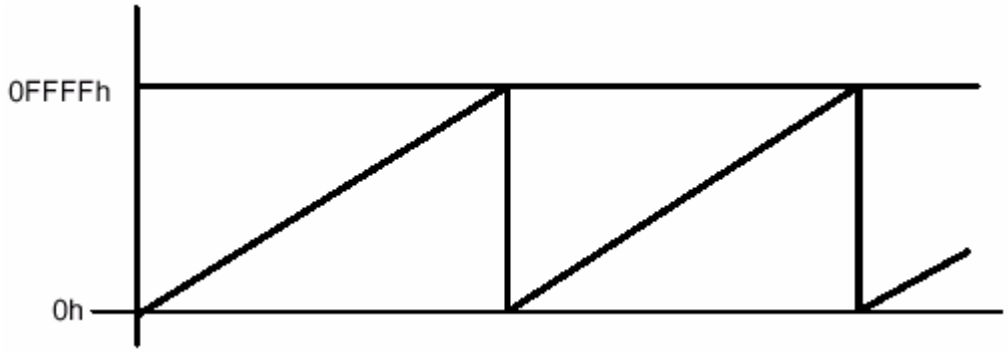


图 2.11.8　　连续计数模式定时器 TAR

如果相应的中断允许，则每当一个定时间隔到都会产生中断请求。那么在连续模式下，须将下一事件发生的时间在当前的中断程序中加到 CCRx 中。在图 2.11.9 可看出这种情况：每隔△t 产生中断，须在定时器等于 CCR0a 时产生的中断服务程序中，将 CCR0b 加到 CCR0 寄存器中。

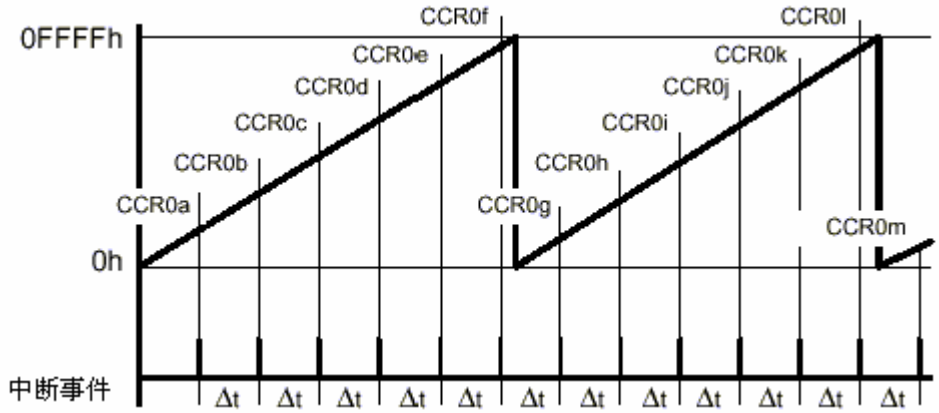


图 2.11.9　　在连续模式时中断与 CCRx 的关系

增／减计数模式

在增减计数模式下，计数器 TAR 的值先增后减：当增计数到 CCR0 的值时，计数器停止增计数，变为减计数，当减到“0”时，设置标志位 TAIFG。由此可见这种模式的计数周期为 CCR0 值的两倍。所以常用于须得到对称波形的场合。这种模式时计数器中数值变化情况如图 2.11.10 所示。

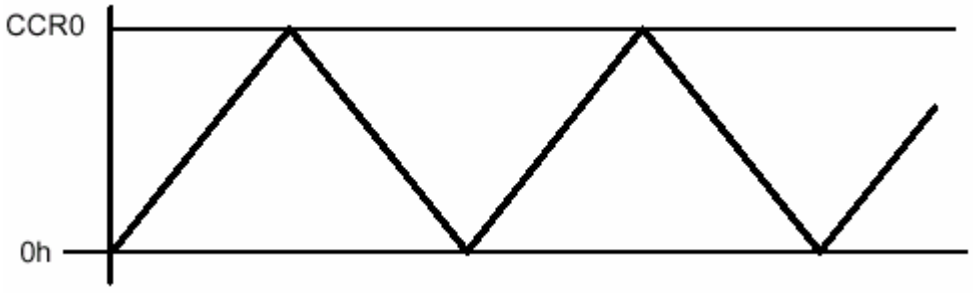


图 2.11.10　增／减计数模式时的计数器 TAR

增/减模式时，中断标志 CCIFG0、TAIFG 会在相等的时间间隔置位。在一个完整的周中，每个标志位只置位一次，分别在半周期时发生。当定时器 TAR 的值从 CCR0-1 增计数到 CCR0 时，中断标志 CCIFG0 置位。当定时器从“1”减计数到“0”时，中断标志 TAIFG 置位。

下面将以设计一段时间的定时为例讲述各种定时器如何初始化。

1、 BT 的初始化：（假设允许中断）

```
IE2 |= BTIE;                          // Enable BT interrupt
BTCTL = BTSSEL+BTIP2+BTIP1+BTIP0;
_EINT();                              // Enable interrupts
```

2、 WDT 的初始化：（假设允许中断）

```
WDTCTL = WDT_MDLY_32;          // Set Watchdog Timer interval to ~30ms
IE1 |= WDTIE;                  // Enable WDT interrupt
_EINT();                       // Enable interrupts
```

3、 TA 的初始化：（假设允许中断）

```
TACTL = TASSEL1 + TAC          // SMCLK, clear TAR
CCTL0 = CCIE;                  // CCR0 interrupt enabled
CCR0 = 50000;
TACTL |= MC1;                  // Start Timer_A in continuous mode
_EINT();                       // Enable interrupts
```

4、 TB 的初始化：（假设允许中断）

```
TBCTL = TBSSEL1 + TBCLR;       // SMCLK, clear TAR
TBCCTL0 = CCIE;                // CCR0 interrupt enabled
TBCCR0 = 50000;
TBCTL |= MC1;                  // Start Timer_A in continuous mode
_EINT();                       // Enable interrupts
```

综合举例（实验 12）：

使用 TA 设计时钟，并在液晶上显示。

首先，设置 LCD：

```
LCDCTL = 0XFD;
BTCTL = BTFRFQ1;                    // STK LCD freq
P5SEL = 0xFC;                       // Common and Rxx all selected
```

然后，设置定时器 TA。

```
TACTL = TASSEL1 + TACLR;            // SMCLK, clear TAR
CCTL0 = CCIE;                       // CCR0 interrupt enabled
CCR0 = 20000;
TACTL |= MC1;                       // Start Timer_A in continuous mode
```

最后，打开中断，写中断服务程序，详细的程序清单如下：

```
#include  <msp430x44x.h>
char digit[20] = {1,0,0,0,0,2,1,8};
unsigned char distab[]=     {      0xaf,0x06,0x6d,0x4f,
                                  0xc6,0xcb,0xeb,0x0e,
```

```
                                    0xef,0xcf
                                };
void main(void)
{
    int i;
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  FLL_CTL0 |= XCAP14PF;                     // Configure load caps
  LCDCTL = 0XFD;
  BTCTL = BTFRFQ1;                          // STK LCD freq
  P5SEL = 0xFC;                             // Common and Rxx all selected
  TACTL = TASSEL1 + TACLR;                  // SMCLK, clear TAR
  CCTL0 = CCIE;                             // CCR0 interrupt enabled
  CCR0 = 20000;
  TACTL |= MC1;                             // Start Timer_A in continuous mode
  _EINT();                                  // Enable interrupts
   for (;;)
  {
    _BIS_SR(CPUOFF);                        // CPU off
    _NOP();                                 // Required only for C-spy
  }
}

interrupt[TIMERA0_VECTOR] void Timer_A (void)
{   char i=0;
  CCR0 += 20000;                            // Add Offset to CCR0
  LCDMEM[7]=0xa;    LCDMEM[8]=0x90;    LCDMEM[9]=0x12;    LCDMEM[10]=0x80;
  LCDMEM[11]=0x2;   LCDMEM[12]=0x93;   LCDMEM[13]=0x72;   LCDMEM[14]=0x5b;
  LCDMEM[15]=0x94;
       for(i=0;i<6;i++)
       LCDMEM[i+1]=distab[digit[i]];
  digit[0]++;
  if(digit[0]==50)
        {
        digit[0]=0;
        digit[1]++;
        if(digit[1]==10)
            {
            digit[1]=0;
            digit[2]++;
            if(digit[2]==6)
                {
```

```
                digit[2]=0;
                digit[3]++;
                if(digit[3]==10)
                        {
                        digit[3]=0;
                        digit[4]++;
                        if(digit[4]==6)
                                {
                                digit[4]=0;
                                 }
                         }
                    }
                }
        }
}
```

2.12　ADC12 原理与应用（实验 13）

MSP430 的 ADC 有 4 种类型：ADC10、ADC12、ADC14、SLOPE（斜边 ADC 转换）等。最常见的在 MSP430F13、14、15、16、43、44 等系列种的 ADC12，本讲将详细讲述 ADC12，同时示例具体应用。

ADC12 的结构如图 2.12.1 所示：

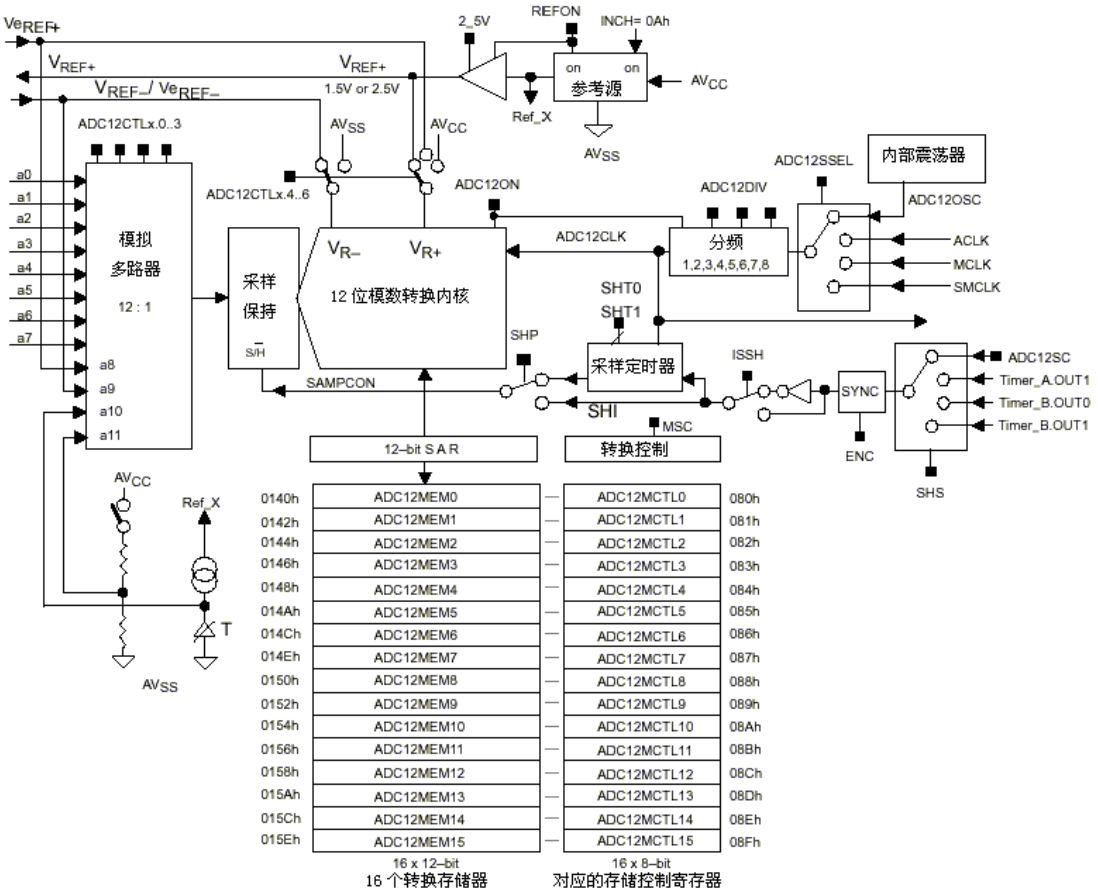


图 2.12.1　ADC12 结构原理

由上图可看出，ADC12 由以下 5 大功能模块构成：

一个带有采样与保持功能的 12 位转换器的 ADC12 内核；

内部参考电压发生器，同时有两种参考电压值可供选择；

可以选择的采样与转换过程中所需要的时钟信号源；

采样以及转换所需的时序控制电路；

转换结果有专门的桶型缓存。

ADC12 的主要特性归纳为如下：

采样速度快，最高可达 200ksps；

12 位转换精度，1 位非线性微分误差，一位非线性积分误差；

内置采样与保持电路，省去外部扩展的麻烦；

模块本身内置转换所需要时钟发生器，同时还有更多种时钟源可提供给 ADC12 模块；

内置温度传感器；

配置有 8 路外部通道与 4 路内部通道；

内置参考电源，而且有参考电压有 6 种可编程的组合；

模数转换有 4 中模式，可灵活地运用以节省软件量以及时间；

ADC12 内核可关断以节省系统能耗。

ADC12 的所有功能都可通过用户软件独立配置。

通过 ADC12 的功能寄存器来使用 ADC12 相当灵活与方便。该模块的寄存器很多：ADC12CTL0 、 ADC12CTL1 、 ADC12IFG 、 ADC12IE 、 ADC12IV 、 ADC12MCTL0~ADC12MCTL15、 ADC12MEM0~ADC12MEM15。部分如下：

ADC12CTL0：

15 ~ 12	11 ~ 8	7	6	5	4	3	2	1	0
SHT1	SHT0	MSC	2.5V	REF ON	ADC12 ON	ADC12 OVIE	ADC12 TVIE	ENC	ADC12 SC

ADC12CTL1：

15 ~ 12	11~10	9	8	7~5	4、3	2、1	0
CSStartAdd	SHS	SHP	ISSH	ADC12 DIV	ADC12 SSEL	CONSEQ	ADC12 BUSY

ADC12MCTLi：

7	6、5、4	3、2、1、0
EOS	Sref　参考电压源	INCH　输入通道

上述表格中阴影部分在使用时要特别注意：只有在 ENC 位为“0”时方可修改。

首先说明参考电压，要将模拟量转换为数字量，必须有参考电压，参考电压有内部与外部两种。如果有使用内部参考电压，则在使用时，确保在 ENC=”0”的前提下，REFON=1，ADC12ON=1。这时只是给 ADC 电路供电，给参考电源部分供电，而转换时究竟是以什么为参考还没有指明，所以下面指明具体的参考电源。内部参考电源有 1.5V、2.5V 两种，当 ADC12CTL0 的第 6 位为“0”时是 1.5V 参考电压，否则为 2.5V。

当然也可以使用外部参考电源，在使用外部参考电源时，必须将正确的标准电压源连接在参考电源引脚端。

在 ADC12MCTLi 寄存器中的第 4、5、6 位将指明参考电源非常细节的情况，下面为第 4、5、6 位所表示的数据对应的参考电源情况：

一共有 6 种情况可供选择，分别为 V_{R+}与 $V_{R_}$的组合。

0：　V R+ = AV CC　　V R– = AV SS

1：　V R+ = V REF+　　V R– = AV SS

2,3：V R+ = Ve REF+　　V R– = AV SS

4：　V R+ = AV CC　　V R– = V REF– / Ve REF–

5：　V R+ = V REF+　　V R– = V REF– / Ve REF–

6,7：V R+ = Ve REF+　　V R– = V REF– / Ve REF–

在确定了转换所需要的参考电压源后，好需要确定采样、保持时间、以及转换的速度与采样速率。

ADC12CTL0 的第 8～15 位将确定采样、保持时间。

ADC12CTL1 的第 3、4 位 ADC12SSEL 选择 ADC12 内核时钟源

0：　ADC12 内部时钟源为 ADC12OSC（来自 AADC12 内部振荡器）；

1：　ACLK；

2：　MCLK；

3：　SMCLK。

ADC12CTL1 的第 5、6、7 位 ADC12DIV 选择 ADC12 时钟源分频因子，共 3 位，分频因子为该 3 位二进制数加 1。

有了以上这些，是否可以进行 ADC 转换呢，还必须有采样保持与转换的启动。ADC12 模块提供多种启动条件。ADC12CTL1 的第 10、11 位 SHS 控制位将决定由什么信号启动采样转换：

0：　ADC12SC；

1：　Timer_A.OUT1；

2：　Timer_B.OUT0；

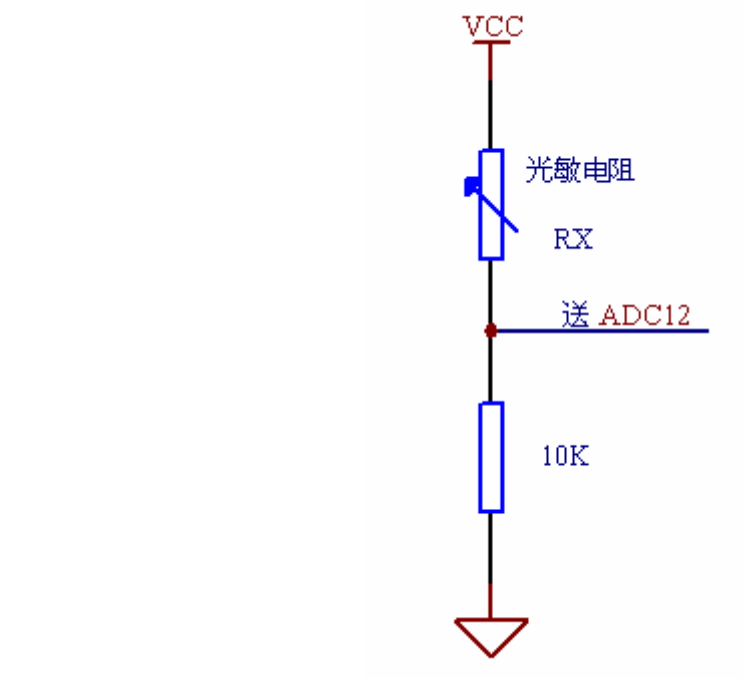
3：　Timer_B.OUT1。

ADC12 模块提供多模拟量输入，确定了以上所讲述的控制位之后，就剩下选取哪个模拟通道的模拟量送 ADC12 模块进行模数转换了。ADC12MCTLi 的最低 4 位所表示的数值将为送达 ADC12 的模拟通道。

在进行模数转换的过程中，ADC12CTL1 的位 0 －－ADC12BUSY 位将为 1，读取转换的结果必须在 ADC12BUSY 位为 0 之后，因为 ADC12BUSY 位为 1 表示正在转换，为 0 表示转换完成。只有在转换完成之后才能读取正确结果。

ADC12 还有 4 种转换模式：单通道单次，单通道多次，多通道单次，多通道多次。这里不详细讲述，下面的例子应用的是单通道单次转换模式。

下面以路灯控制器的设计为例说明 ADC12 的使用方法。路灯在一定的亮度值以上将熄灭，而在亮度低于某个数值时亮起来。下面是简图。当亮度较大、光线较强时，光敏电阻的电阻值比较小，这时它与下面 10K 分压，则送达 ADC12 的电压比较高；而当亮度较小、光线较弱时，光敏电阻的电阻值比较大，这时它与下面 10K 分压，则送达 ADC12 的电压比较低。通过 ADC12 模块转换出具体的表示光强度的数值（光线越强，转换后的数值越大，但不成正比例），再设定一个开启路灯的阀值数据，则可以通过实际测量的光强度数据与阀值数据比较得到是否开启路灯的目的。



下面是具体示例程序：（使用单通道单次定时转换）。

```
#include              "msp430x44x.h"          //使用 MSP430F447
void main(void)
{
   WDTCTL = WDTPW+WDTHOLD;       // 停止看门狗
   P6SEL |= 0x01;                            //定义 P6.0 为模拟输入通道 0
   ADC12CTL0 = ADC12ON+SHT0_2;     // 打开 ADC12 电源，并设置采样时间
   ADC12CTL1 = SHP;
   ADC12CTL0 |= ENC;                       //使能转换
       while (1)
           {
               delay(60000)                              //延时 1 秒（大致，相当于定时器的作用）
               ADC12CTL0 |= ADC12SC;                 // 开始启动转换
               while ((ADC12IFG & ADC12BUSY)==0);  //等待转换的完成
               if(ADC12MEM0<1234)                    //读取转换结果并比较以得到结论
                         P1OUT    |=     BIT0;          //当亮度低于阀值时打开路灯
               else      P1OUT    ^=    BIT0;          //当亮度高于或等于阀值时关闭路灯
           }
}
```

2.13　MSP430 串行异步通讯原理与实现（实验 14）

本讲讲述串口功能与连接的实现。大多数 MSP430 芯片都有硬件异步通讯功能，有一些器件有两个通讯端口，也有少数没有。没有硬件串口的芯片可以实现软件（模拟）串口。下面表格为 430 系列芯片串口的情况。

系列芯片	F11 系列	F12 系列	F13 系列	F14 系列	F15 系列	F16 系列
串口数量	0	1	1	2	1	2

芯片系列	F2 系列	C31 系列	C32 系列	C33 系列	F41 系列	F42 系列
串口数量	1	0	0	1	0	1

芯片系列	FW42 系列	FE42 系列	FG43 系列	F43 系列	F44 系列	
串口数量	0	1	1	1	2	

对于没有硬件串口的芯片也可以实现软件串口，这里先讲硬件串口，后讲软件串口。然后再讲串口的链路实现。

先看串口功能的实现。

图 2.13.1 是 MSP430 系列芯片硬件串口的框图。

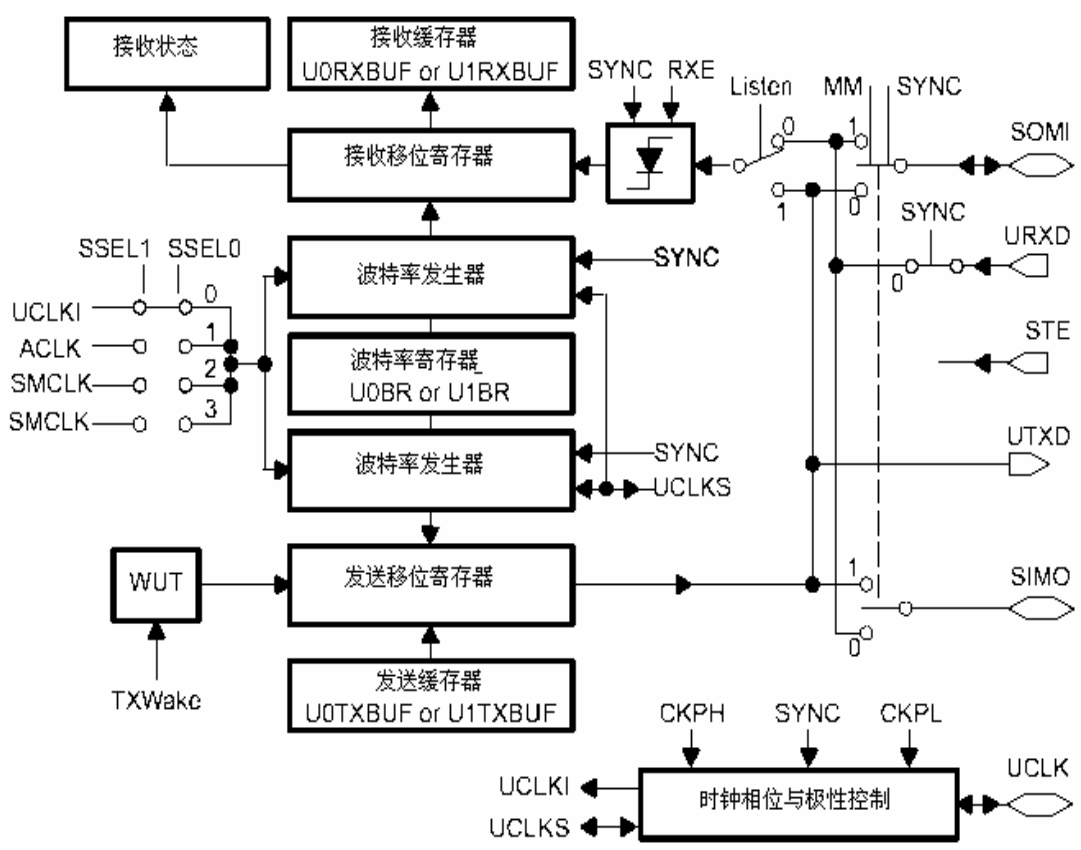


图 2.13.1　　串口通讯硬件原理框图

在该框图中，串口通讯由 3 部分构成：通讯速度的控制（数据位流的产生）、接收控制部分、发送控制部分。

波特率生成部分由时钟输入选择与分频、波特率发生器、调整器、波特率寄存器等组成。串行通信时，接收与发送以什么样的速率将数据位收进或送出呢，这个速率就由波特率生成构件控制。图 2.13.2 为其较为详细的结构。

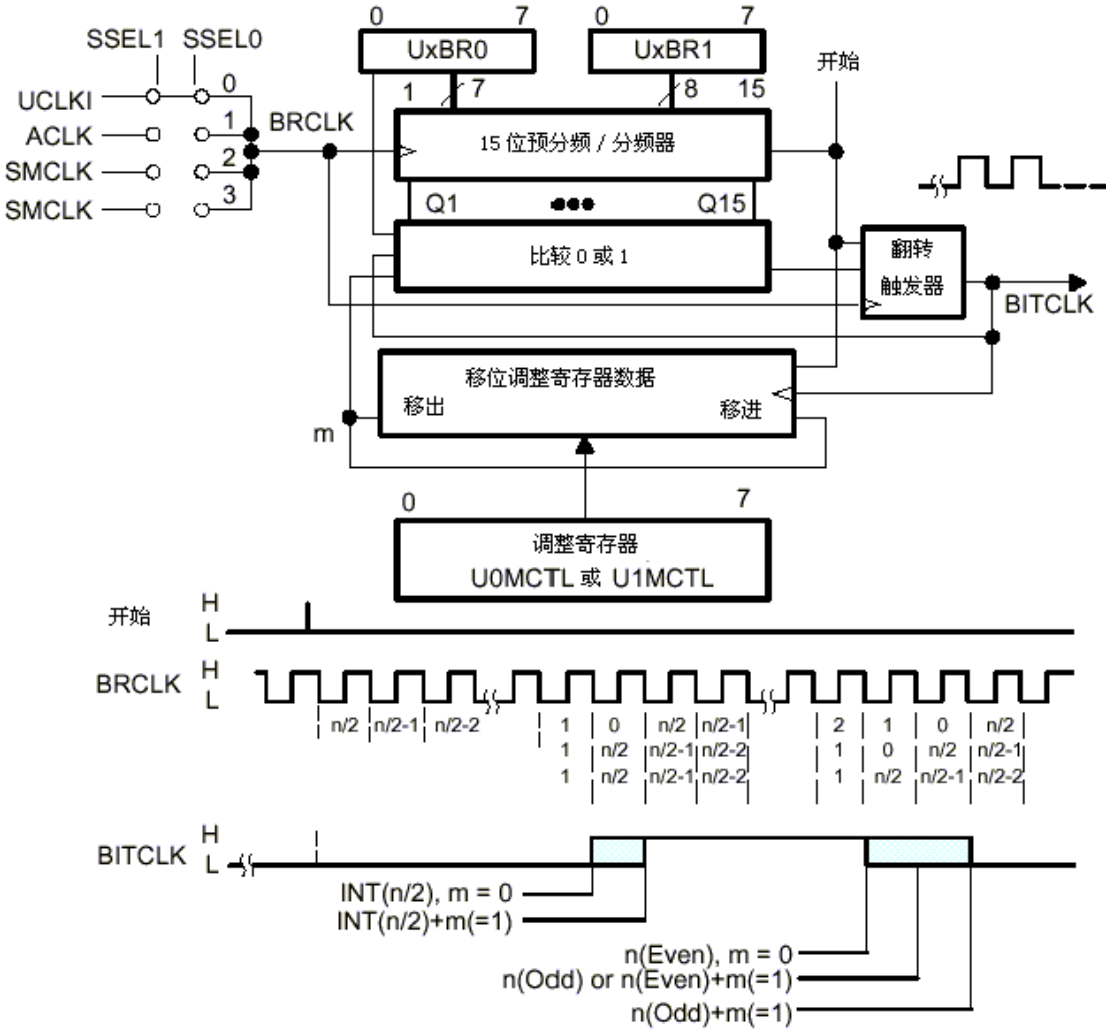


图 2.13.2　　串口通讯硬件原理图

整个模块的时钟源来自内部 3 时钟或外部输入时钟，由 SSEL1、SSEL0 选择，以决定最终进入模块的时钟信号 BRCLK 的频率。时钟信号 BRCLK 送入一个 15 位的分频器，通过一系列的硬件控制，最终输出移出与移进两移位寄存器使用的移位位时钟 BITCLK 信号。那么这个信号（BITCLK）究竟是怎样产生的呢，该图的下半部分的一个波特率产生例子可以看出，是分频器在起作用。当计数器减计数到“0”时，输出触发器翻转，送给 BITCLK 信号。所以 BITCLK 信号周期的一半就是定时器（分频计数器）的定时时间。

接收控制部分与发送控制部分分别由两个移位寄存器构成。接收时，当接收到一个完整数据，产生一个信号（URXIFG0＝1），表示接收到完整数据，可以将此数据取走。而在发送时，当一个数据正在发送过程中，UTXIFG0＝1，此时，不能再发送数据，必须等当前数据发

送完毕（UTXIFG0＝0）时，方可继续发送。

串口接收一般采用中断方式，而发送数据则多采用主动方式。下面是一段简单的完整通讯程序，实现功能：将接受的数据原样送回。

```
#include <msp430x44x.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;   // 停止看门狗
  UTCTL0 = SSEL0;             // UCLK = ACLK，选择时钟来源
  UBR00 = 0x03;               // 32k/9600 - 3.41   波特率寄存器低字节
  UBR10 = 0x00;               // 32k/9600          波特率寄存器高字节
  UMCTL0 = 0x51;              // 由于波特率计算有余数，填写波特率调整寄存器
  UCTL0 = CHAR;               //   数据格式为 8 位数据
  ME1 |= UTXE0 + URXE0;       //   使能串口 TXD 与 RXD
  IE1 |= URXIE0;              // 让串口接收到数据后能产生中断
  P2SEL |= 0x30;              //   定义 P2.4,P2.5 为串口功能引脚
  P2DIR |= 0x10;              // 串口发送数据端口为输出，接收数据端口为输入
  _EINT();                    // 整个系统使能中断（开总中断）
  _BIS_SR(LPM3_bits);         //   初始化完毕，进入睡眠状态，主程序完毕
}

interrupt[UART0RX_VECTOR] void usart0_rx (void)
{
  while ((IFG1 & UTXIFG0) == 0);          // 当发送缓存为空时
  TXBUF0 = RXBUF0;                        //发送数据到串口
}
```

而对于没有硬件串口的型号，如何实现异步串口功能？

先分析异步串口的原理。图 2.13.3 是异步串口的时序图。

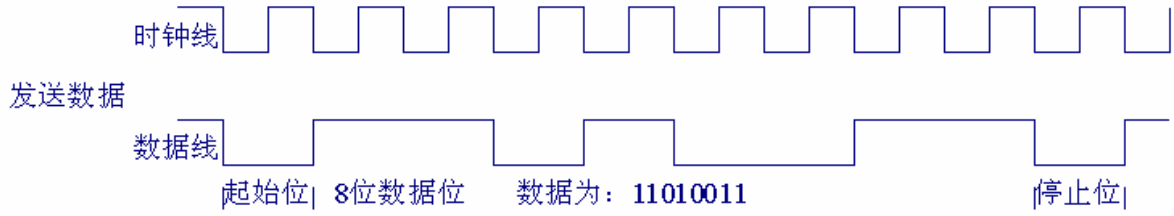


图 2.13.3　　异步通讯时序

可以看出异步串口由一根口线构成：数据线，在数据发送时，数据线严格按照其时序将数据移位送至数据线，就可以了。图中的时钟是隐含的，由波特率确定。比如串口波特率为 9600，则时钟的周期为 1/9600 秒。在数据线上的数据按照：起始位、数据位、停止位等格式顺序排列。而起始位、数据位、停止位等的多少由通讯双方定义的通讯规约决定。

这样在没有硬件串口的情况下，完全可以模拟以上时序发送异步串行数据。下面的工作将完成上图的数据传送。

首先产生波特率。下面的延时程序可以完成此工作，延时时间为 1/9600 秒（系统时钟为 1M 时的延时循环参数）。

```
void Delay_9600(void)
        {
         unsigned int v=104;
         while(v!=0)v--;
        }
```

以上的延时程序用于产生通讯位率。下面定义 P1.0 为通讯数据发送端，P2.0 为通讯数据接收端。则按照通讯规约的时序图，每发送一位数据，调用一次延时程序：

调用以下程序之前已经进行了相应的端口方向设置。

```
void send_byte(char in)  //输入变量为即将发送的数据
    {
    char I=0;                       //定义一个循环变量，循环发送 8 个数据位
    P1OUT &= ~BIT0;                 //发送起始位
    Delay_9600();
    for(I=0;I<8;I++)
        {
        if(in&1)
            P1OUT |= BIT0;
        else    P1OUT &= ~BIT0;     //将数据位送到端口
        in = in>>1;                 //准备下位数据
        Delay_9600();               //位时钟
        }
    P1OUT &= ~BIT0;                 //发送停止位
    Delay_9600();
    }
```

数据的接收可以使用中断的方式，设置 P2.0 为设为输入，并使能中断。当串行数据送达 P2.0 时，该端口将产生中断，在中断服务程序中，顺序接受 10 位数据，并去掉最先位（起始位）与最后位（停止位），再将中间 8 位组合成一个字节，即为接收到的数据，然后退出中断，等待其他数据的接收。这里要注意接收每一位数据的采样点。

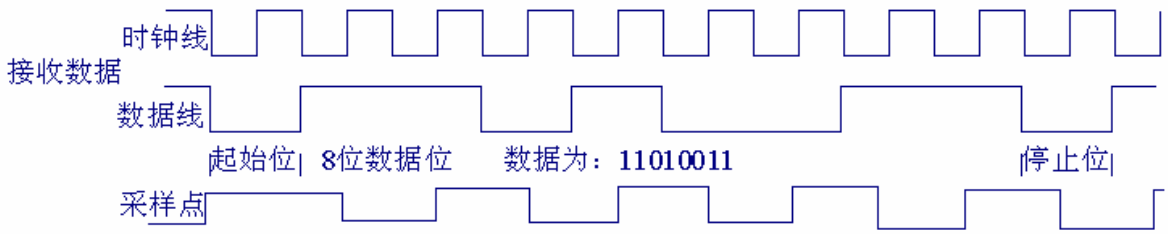


图 2.13.4　　模拟接收时序图

可以将起始位采取不予理睬的办法，如图 2.13.4 所示，则第一位数据的采样位于进入中断后的 1.5 位时钟处。然后延时一个位时钟的时间，再采样下一为数据。注意所有的采样点位

于每一位数据的中间位置，其原因很明显：在时间上可以最大程度地容错。具体的接收程序略。

现在对于有无硬件串口，都可以进行串口通讯了，但这只能近距离 TTL 电平连接，对于远距离呢，必须使用对应的硬件电路实现通讯链路。常用的通讯链路有 RS232，RS485，红外线等。所有通讯链路的实现都只是将通讯双方以一定的电气规约联系起来。

RS232 链路可以将通讯双方在 15 米以内有效连接。RS232 规定逻辑电平 0 为＋3～＋15V 电压，逻辑电平 1 为－15～－3V 电压，通常以 MAX232 芯片实现电平转换，具体电路如下图。

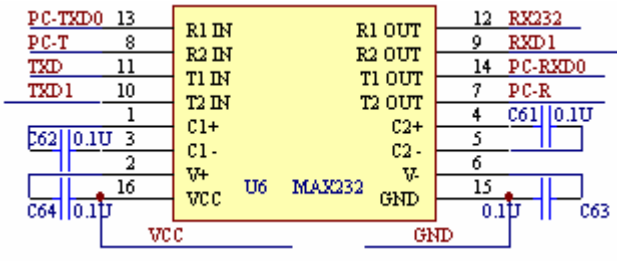


图 2.13.5　　RS232 通讯电路

如果要实现较远的通讯距离，则通常选择 RS485 总线。RS485 通讯距离能达到 1.2Km，使用双绞线，但只能半双工通讯（即不能同时发送与接收数据）。使用 RS485 可以方便进行单片机网络构成，所有节点都挂在 RS485 总线上。RS485 总线使用差分电压，具有很高的抗干扰能力。规定总线 A 高于 B0.2V 时为数据 1，总线 B 高于 A0.2V 时为数据 0。下面是典型的 RS485 电路如下图所示。其中 P60J 为总线方向控制，因为 MAX485 器件半办双工，在同一时间只能是数据发送或数据接收，所以需要控制器件所处的工作状态，要么是发送数据，要么是接收数据。

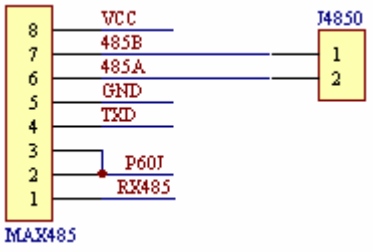


图 2.13.6　　RS485 通讯电路

前面两种通讯链路都是有线连接，下面讲讲最常见的红外通讯链路。红外通讯是将红外线作为通讯的载体。下图是红外线调制原理，当需要送处的信号为 1 时，就将调制信号送出；当要送出信号 0 时，就不发送任何信号。

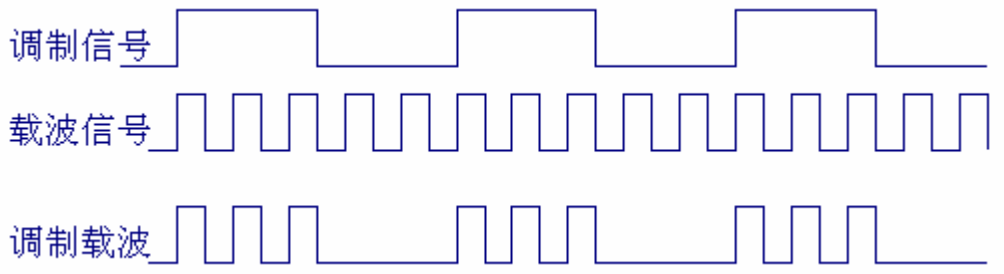


图 2.13.7　　红外通讯调制波形

下图是常用的红外线调制解调电路。常用的红外载波信号为 38K 的方波信号，下图使用 NE555 产生 38K 方波信号；而接收部分采用一体红外接收器件，将红外接收管、放大电路、解调制等集成为一体，给使用带来极大方便，只需要一个输出上来电阻即可。

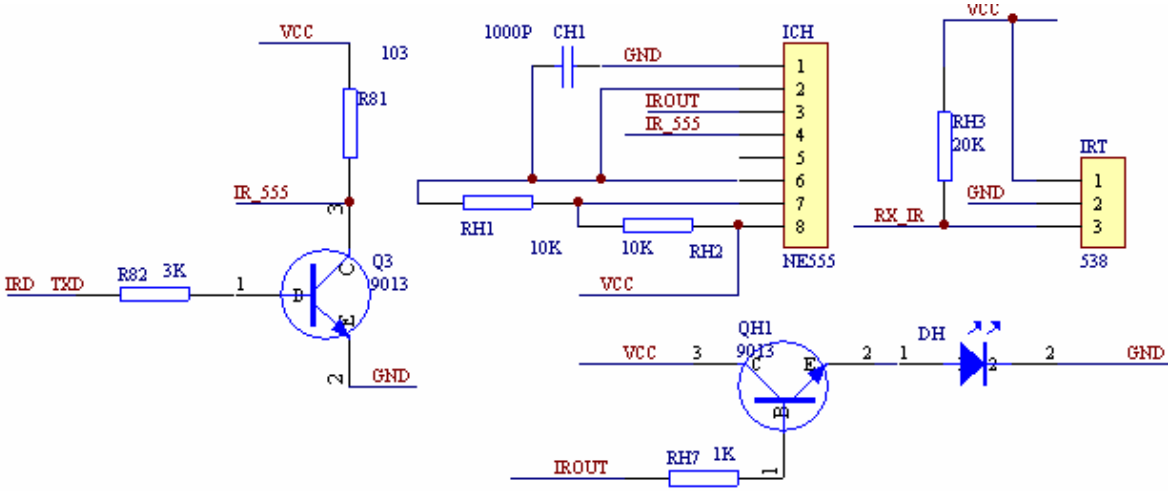


图 2.13.8　　红外通讯电路图

红外通讯可实现近距离无线连接，但不能绕过障碍物，还有很多其他通讯链路，这里不一一介绍。

第三章　　MSP430 微处理器综合实践设计

通过前面的学习，已经掌握了 16 位 MSP430 微处理器的基础知识，同时也通过实验深刻理解了 MSP430 微处理器的原理等，在这里，读者完全可以更加深入地灵活运用了！本章将带领读者进入 MSP430 的综合应用阶段。

但是本章的章节讲解有简有略，对于比较简单的实践环节就只提示，对于比较复杂的实践环节将做重点讲解。

3．1　　电子温度计设计（实验 15）

MSP430F449 片内 ADC12 的 A10 为片内温度传感器，温度与输出电压存在图 3.1.1 所示的关系。

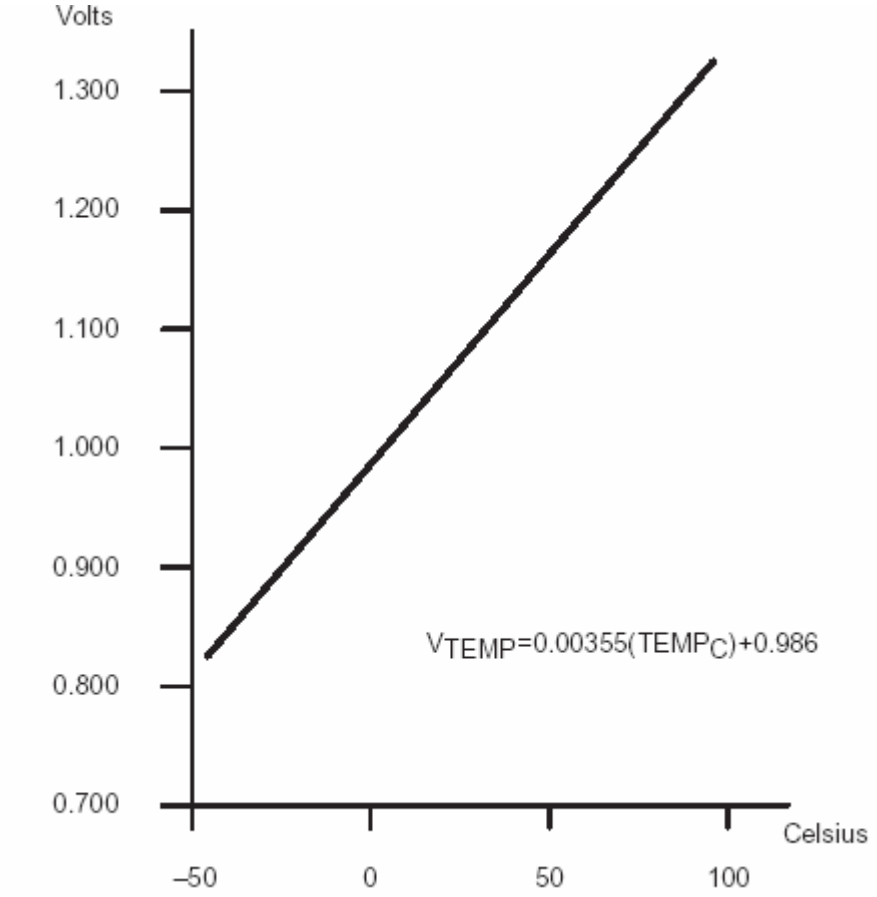


图 3.1.1　　ADC12 温度传感器的温度－－电压关系

实验要求：

通过 ADC12 模块可以将温度传感器感知的温度数字化，再显示出来；

显示可以使用液晶，也可以使用数码管；

温度值需要大致准确；

提高要求：

能实现温度标定（需要标准温度表）、编写相关标定程序以及标定显示界面、以及操作界面（需要键盘）。

实验应用相关知识点（请查阅相关知识）：

ADC12 模块；

液晶显示模块；

数码管显示；

程序编写；

键盘程序应用。

3．2　　简单温度控制设计（实验 16）

在以上实践环节的基础之上，增加难度：实现温度的控制。

分析：

需要控制准确，则需要测量准确，ADC12 能实现 2 位转换精确度，可以量化模拟量在 0－－4095 之间，如果测量温度范围在 0－－100 度、对应电压为 0.8V－－1.3V，则量化的数值为（参考电压为 0.8V－－1.3V）0－－4095 之间，则可以分辨到 100/4095＝0.024 摄氏度，非常精确了！

测量能实现高分辨率，但部等于精确度，还需要校准，需要高准确度的标准表校对。

控制需要好的控制算法，比如 PID 控制、模糊控制、01 控制等。这里采用最简单的模糊控制。

控制对象为小功率加热器加热的一定容积的保温箱。

实现：

1、 温度测量已经实现，ADC12 以及前面的实验。

2、 控制算法。

采样简单的模糊控制算法，关于模糊控制，这里部细讲，请查阅相关知识。采用最简单的模糊控制算法：单输入，单输出。

首先明确控制对象：一定容积的保温箱内的温度。

其次，模糊控制算法的输入：一定容积的保温箱内的温度与目标的差值。

最后，模糊控制算法的输出：加热器的功率输出。

算法的输入输出关系。采用简单的模糊控制算法，所以，关系比较简单：（表 3.2.1）

表 3.2.1　　简单的模糊控制算法输入输出关系

输入：温度与目标的差值	20	10	3	0
输出：加热器的功率	全功率	中功率	小功率	0

3、输出功率控制方法。

那么如何实现输出功率控制呢。先看看图 3.2.1 的 4 个不同信号 a、b、c、d。假设他们的周期都是 1Hz，则这些信号控制加热器后，加热器的发热不一样。很显然，信号 a 驱动发光二极管最不热，信号 d 驱动发光二极管最热。

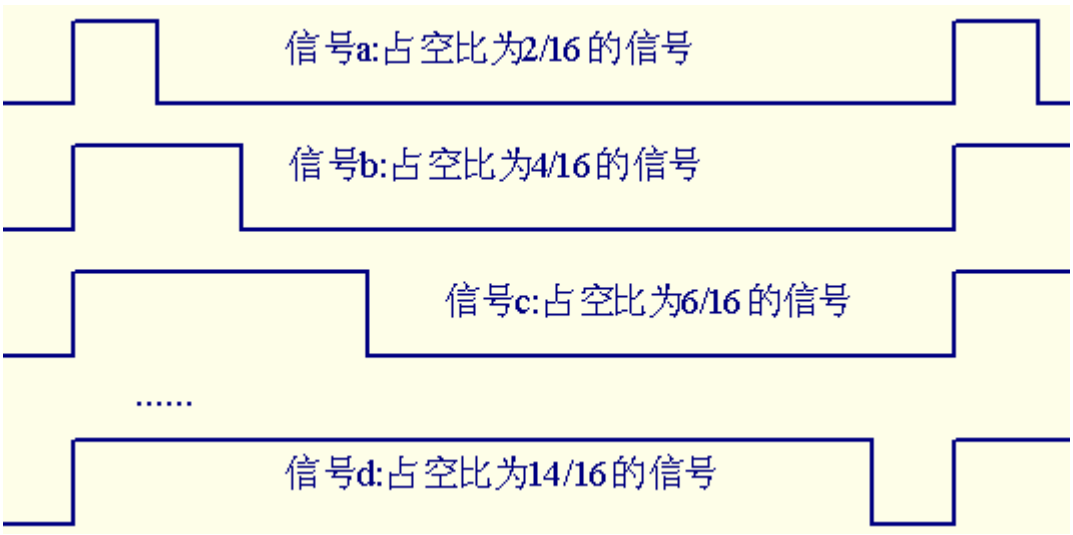


图 3.2.1　　热功率控制信号

实验要求：

通过 ADC12 模块可以将温度传感器感知的温度数字化，再显示出来；

显示可以使用液晶，也可以使用数码管；

温度值需要大致准确；

定值控制；（确定在某一个温度控制点上，不可以修改）

控制误差为正负 3 度。

提高要求：

能实现温度标定（需要标准温度表）、编写相关标定程序以及标定显示界面、以及操作界面（需要键盘）；

引入键盘；

温度控制值的键盘输入；

误差为正负 2 度。

实验应用相关知识点（请查阅相关知识）：

ADC12 模块；

液晶显示模块；

数码管显示；

程序编写；

键盘程序应用；

模糊控制；

3．3　　简单计算器的设计（实验 17）

实验要求：

设计简单计算器，实现“加减乘除”计算。

输入：

键盘有 12 按键：0－－9、*、＃等，分配：

0－－9 为数字键以及符号键。

1 为＋；

2 为一；

3 为×；

4 为“除”；

固定操作的数据位：5 位。

数字键与符号键的区分：如果输入为 5 位则为数字，否则为符号。

＃为取消前面的输入。

* 为确认前面的输入。

举例：

实现“23452 ＋ 25498”的计算，输入如下：

依次输入：“2 3 4 5 2 * 1 * 2 5 4 9 8 * ”，当第二个操作数据输入完毕，在显示器上显示计算结果。

同时，输入数据显示对应的数据。刚才的输入对应的显示为：

输入 2，显示：

2

输入 3，显示：

2　3

输入 4，显示：

2　3　4

输入 5，显示：

2　3　4　5

输入 2，显示：

2　3　4　5　2

输入* ，显示：（清空当前显示，准备显示操作符代号）

输入 1，显示：

1

输入* ，显示：（清空当前显示，准备显示第二操作数据）

以后同前。

最后一个 * 输入之后，运算结果显示出来。

3．4　数字电压表设计（实验 18）

实验要求：

可以调整放大倍数；

显示电压值。

提高要求：

能自动放大倍数调整；

数据语音播报。

3．5　电脑密码锁（实验 19）

实验要求：
如果输入的密码与已经存在的密码一样，则开锁。
提高要求：
密码管理，
能修改密码。

3．6　　可编程波形发生器设计（实验 20）

实验要求：
正弦波；
三角波；
斜波；
方波；
相关知识：
DAC。

3．7　　风扇转速测控（实验 21）

实验要求：
测量风扇转动的速度；
控制转速在某一个固定值。

3．8　　多维机械手臂控制设计（实验 22）

实验要求：
控制机械臂的动作。
相关知识：
舵机的控制。

3．9　　电子称设计（实验 23）

实验要求与相关知识：
压力传感器的知识；
测量压力传感器的输出；
压力的显示。

3．10　　固体数码录音机（实验 24）

在单片机应用中经常要输出语音，比如操作上的语音提示，测量上的语音报告等等。它们的核心都是语音的录入与输出（录音与放音）。本示例演示了如何在 MSP430 系统中如何录

音，与如何放音。

在单片机系统中使用较多的语音设备是 ISD 系列的语音芯片。其中 ISD4004 更有 16 分钟的语音记录时间。本例的固体录音机将选用此芯片为核心设计。

ISD4004 简介

ISD4004 系列工作电压 3V,单片录放时间 8 至 16 分钟,音质好,适用于移动电话及其他便携式电子产品中。芯片采用 CMOS 技术,内含振荡器、防混淆滤波器、平滑滤波器、音频放大器、自动静噪及高密度多电平闪烁存贮阵列。芯片设计是基于所有操作必须由微控制器控制,操作命令可通过串行通信接口(SPI 或 Microwire)送入。芯片采用多电平直接模拟量存储技术, 每个采样值直接存贮在片内闪烁存贮器中,因此能够非常真实、自然地再现语音、音乐、音调和效果声,避免了一般固体录音电路因量化和压缩造成的量化噪声和"金属声"。采样频率可为 4.0,5.3,6.4,8.0kHz,频率越低,录放时间越长,而音质则有所下降,片内信息存于闪烁存贮器中,可在断电情况下保存 100 年(典型值),反复录音 10 万次。

引脚描述

电源:(VCCA,VCCD) 为使噪声最小,芯片的模拟和数字电路使用不同的电源总线,并且分别引到外封装的不同管脚上,模拟和数字电源端最好分别走线,尽可能在靠近供电端处相连,而去耦电容应尽量靠近器件。

地线:(VSSA,VSSD) 芯片内部的模拟和数字电路也使用不同的地线。

同相模拟输入(ANA IN+)　这是录音信号的同相输入端。输入放大器可用单端或差分驱动。单端输入时,信号由耦合电容输入,最大幅度为峰峰值 32mV,耦合电容和本端的 3KΩ电阻输入阻抗决定了芯片频带的低端截止频率。差分驱动时,信号最大幅度为峰峰值 16mV。

反相模拟输入(ANA IN-)　差分驱动时,这是录音信号的反相输入端。信号通过耦合电容输入,最大幅度为峰峰值 16mV

音频输出(AUD OUT)　提供音频输出,可驱动 5KΩ的负载。

片选(SS) 此端为低,即向该 ISD4003 芯片发送指令，两条指令之间为高电平。

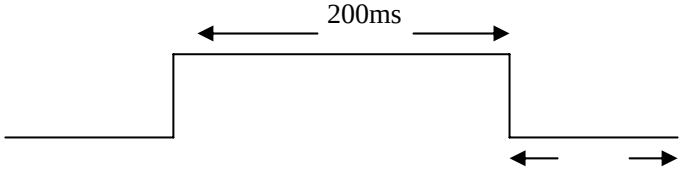
串行输入(MOSI) 此端为串行输入端，主控制器应在串行时钟上升沿之前半个周期将数据放到本端,供 ISD 输入。

串行输出(MISO)　ISD 的串行输出端。ISD 未选中时,本端呈高阻态。

串行时钟(SCLK)　ISD 的时钟输入端,由主控制器产生,用于同步 MOSI 和 MISO 的数据传输。数据在 SCLK 上升沿锁存到 ISD,在下降沿移出 ISD。

中断(/INT)　　本端为漏极开路输出。ISD 在任何操作(包括快进)中检测到 EOM 或 OVF 时,本端变低并保持。中断状态在下一个 SPI 周期开始时清除。中断状态也可用 RINT 指令读取。OVF 标志----指示 ISD 的录、放操作已到达存储器的末尾。EOM 标志----只在放音中检测到内部的 EOM 标志时,此状态位才置 1。

行地址时钟(RAC) 漏极开路输出。每个 RAC 周期表示 ISD 存储器的操作进行了一行(ISD4003 系列中的存储器共产 1200 行，ISD4004 系列中的存贮器共 2400 行)。该信号 175ms 保持高电平,低电平为 25ms。快进模式下,RAC 的 218.75μs 是高电平,31.25μs 为低电平。该端可用于存储管理技术。



25ms

外部时钟(XCLK) 本端内部有下拉元件。芯片内部的采样时钟在出厂前已调校,误差在+1%内。商业级芯片在整个温度和电压范围内, 频率变化在+2.25%内。工业级芯片在整个温度和电压范围内,频率变化在-6/+4%内,此时建议使用稳压电源。若要求更高精度,可从本端输入外部时钟(如附录所列)。由于内部的防混淆及平滑滤波器已设定,故上述推荐的时钟频率不应改变。输入时钟的占空比无关紧要,因内部首先进行了分频。在不外接地时钟时,此端必须接地。

自动静噪(AMCAP) 当录音信号电平下降到内部设定的某一阈值以下时,自动静噪功能使信号衰弱,这样有助于养活无信号(静音)时的噪声。通常本端对地接 1mF 的电容,构成内部信号电平峰值检测电路的一部分。检出的峰值电平与内部设定的阈值作比较,决定自动静噪功能的翻转点。大信号时,自动静噪电路不衰减,静音时衰减 6dB。1mF 的电容也影响自动静噪电路对信号幅度的响应速度。本端接 VCCA 则禁止自动静噪。

SPI(串行外设接口)

ISD4004 工作于 SPI 串行接口。SPI 协议是一个同步串行数据传输协议,协议假定微控制器的 SPI 移位寄存器在 SCLK 的下降沿动作,因此对 ISD4003 而言,在时钟止升沿锁存 MOSI 引脚的数据,在下降沿将数据送至 MISO 引脚。协议的具体内容为:

1、所有串行数据传输开始于 SS 下降沿。

2、SS 在传输期间必须保持为低电平,在两条指令之间则保持为高电平。

3、数据在时钟上升沿移入,在下降沿移出。

4、SS 变低,输入指令和地址后,ISD 才能开始录放操作。

5、指令格式是(5 位控制码)加(11 位地址码)。

6、ISD 的任何操作(含快进)如果遇到 EOM 或 OVF,则产生一个中断,该中断状态在下一个 SPI 周期开始时被清除。

7、使用"读"指令使中断状态位移出 ISD 的 MISO 引脚时,控制及地址数据也应同步从 MOSI 端移入。因此要注意移入的数据是否与器件当前进行的操作兼容。当然,也允许在一个 SPI 周期里,同时执行读状态和开始新的操作(即新移入的数据与器件当前的操作可以不兼容)。

8、所有操作在运行位(RUN)置 1 时开始,置 0 时结束。

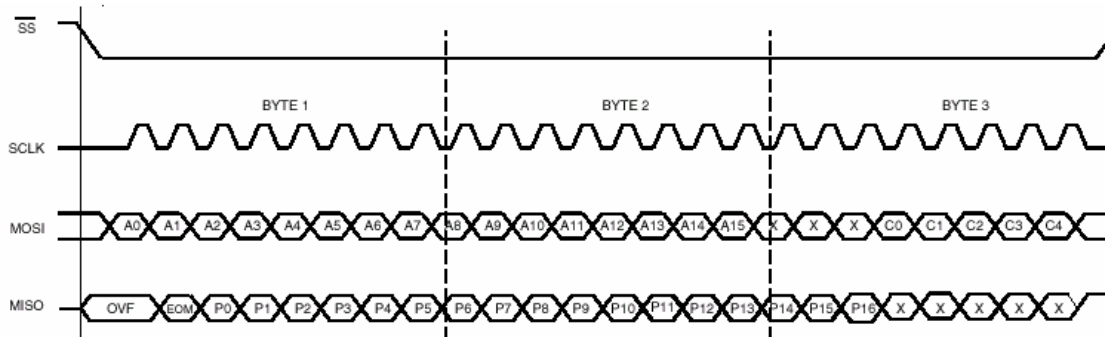
9、所有指令都在 SS 端上升沿开始执行。

ISD4004 指令表

指令	5 位控制码<11 位地址>	操作摘要
POWERUP	00100<XXXXXXXXXXX>	上电:等待 TPUD 后器件可以工作
SET PLAY	11100<A10-A0>	从指定地址开始放音。必须后跟 PLAY 指令使放音继续
PLAY	11110<XXXXXXXXXXX>	从当前地址开始放音(直至 EOM 或 OVF)
SET REC	10100<A10-A0>	从指定地址开始录音。必须后跟 REC 指令录音继续
REC	10110<XXXXXXXXXXX>	从当前地址开始录音(直至 OVF 或停止)
SET MC	11101<A10-A0>	从指定地址开始快进。必须后跟 MC 指令快进继续
MC	11111<XXXXXXXXXXX>	执行快进,直到 EOM.若再无信息,则进入 OVF 状态
STOP	0X110<XXXXXXXXXXX>	停止当前操作
STOP WRDN	0X01X<XXXXXXXXXXX>	停止当前操作并掉电

RINT	0X110<XXXXXXXXXXX>	读状态:OVF 和 EOM

ISD4004 操作时序



固体数码录音机的硬件电路

话筒放大器对语音输入信号放大，以达到 ISD4004 输入信号要求。使用功率放大器驱动扬声器发出语音。由于 ISD4004 与 MCU 的接口为 SPI 串行方式，只需要少量 I/O 口线，这里使用 MSP430F1121 为控制芯片。设置两个按键，录音键与放音键，每个按键第一次按下为启动操作，再次按下为结束操作，停止当前操作。则电路如图 3.10.3 所示。其中的所有器件均为 3V 工作，所以使用电池供电。

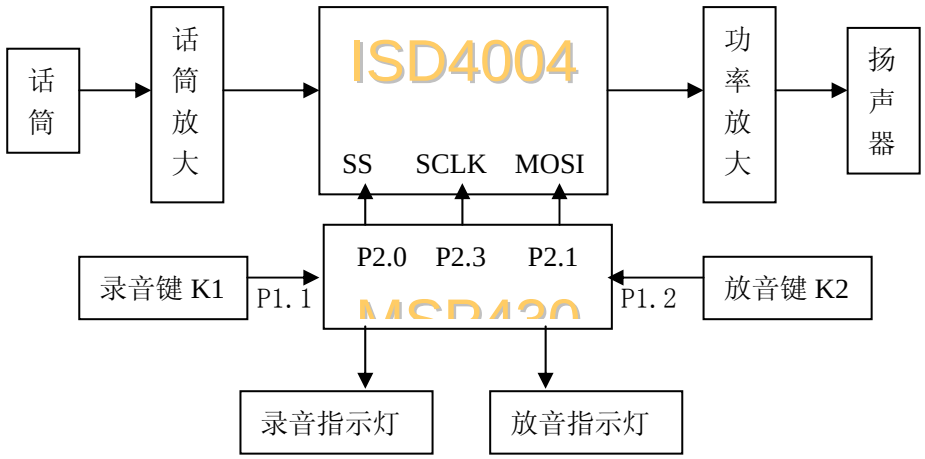


图 3.10.3　固体录音机电路框图

固体数码录音机的软件设计

先看操作界面，对整个录音机的操作只有两个按键，K1、K2 对应录音与放音。系统的运行听从两按键的指挥。所以主程序很简单，查询键盘。如果 K1 按下，则录音，如果 K1 再次按下，则停止录音；如果 K2 按下，则放音，如果 K2 再次按下，则停止放音。程序如下：

```
；约定：200H 字单元，保存录音与放音的起始地址
；        203H 字节单元，为录音键的计数器，用于判断录音键按下的奇数/偶数，用于决定
；        录音操作还是停止录音操作，同时对应指示灯指示。
；        204H 字节单元，为放音键的计数器，用于判断放音键按下的奇数/偶数，用于决定
；        放音操作还是停止放音操作，同时对应指示灯指示。

#include        "msp430x11x1.h"
```

```
                ORG     0F000H
Reset           mov     # 300H ,SP       ；堆栈指针放在最顶端，F133 为 300H
                MOV.B   #0FFH,&P2DIR
                MOV.B   #018H,&P1DIR
                MOV.B   #0FFH,&P2OUT
                MOV.B   #1,&203H        ；初始化为奇数
                MOV.B   #1,&204H        ；初始化为奇数
                BIS.B   #018H,&P1OUT    ；录音指示灯与放音指示灯都熄
                CALL    #DSTOP          ；
                MOV     #0FFFH,R15
MAINLOOP0       DEC     R15             ；初始化 ISD4004 芯片
                JNZ     MAINLOOP0
                CALL    #POWERUP        ；上电
                MOV     #0FFFFH,R15
MAINLOOP1       DEC     R15
                JNZ     MAINLOOP1
MAIN            BIT.B   #02H,&P1IN      ；判断按键，是否录音键
                JNZ     MAIN1
                BIT.B   #4H,&P1IN       ；判断按键，是否放音键
                JNZ     MAIN2
                JMP     MAIN
MAIN1           CALL    #DELAY5
                BIT.B   #02H,&P1IN          ；延时消除抖动
                JZ      MAIN
MAIN1_0         BIT.B   #2,&P1IN        ；判断按键，是否录音键
                JNZ     MAIN1_0
                CMP.B   #1,&203H        ；是录音键偶数次被按下
                JNZ     MAIN11          ；
                MOV.B   #2 , &203H
                MOV     #1200,&200H     ；  从某地址处录音
                BIC.B   #8H,&P1OUT      ；录音指示灯亮
                CALL    #REC_IN         ；是录音键，录音
                JMP MAIN
MAIN11          CALL    #STOPP
                BIS.B   #8H,&P1OUT      ；录音指示灯熄
                MOV.B   #1,&203H        ；&203H 为奇数，下次按键则被认为录音
                JMP     MAIN
MAIN2           CALL    #DELAY5
                BIT.B   #4H,&P1IN       ；延时消除抖动
                JZ      MAIN
MAIN2_0         BIT.B   #4,&P1IN        ；判断按键，是否放音键
```

```
                JNZ         MAIN2_0
                CMP.B       #1,&204H            ；如果奇数次按键，则放音
                JNZ         MAIN22
                MOV.B       #2, &204H
                CALL        #STOPP
                MOV         #1200,&200H         ；从某地址处放音
                CALL        #PLAY_OUT
                JMP         MAIN
MAIN22          CMP.B       #2,&204H            ；如果偶数次按键，则停止放音
                JNZ         MAIN23
                MOV.B       #1,&204H
                CALL        #STOPP
                JMP         MAIN
```

下面是录音子程序。按照 ISD4004 的时序串行输入录音指令。

```
；参数：200H 字单元为录音后存放在 ISD4004 内的具体起始地址，
；          202H 字节单元为 ISD4004 指令
REC_IN          BIC.B     #1,&P2OUT             ;SS=0
                CALL      #SEND_16
                MOV.B     #10100000B,&202H    ;SET REC
                CALL      #SEND_8
                BIS.B     #1,&P2OUT             ;SS=1
                MOV         #03FFH,R15
REC_INLOOP    DEC         R15
                JNZ         REC_INLOOP
                BIC.B     #1,&P2OUT             ;SS=0
                MOV.B     #10110000B,&202H
                CALL      #SEND_8
                BIS.B     #1,&P2OUT             ;SS=1
                RET
```

下面是放音子程序。按照 ISD4004 的时序串行输入放音指令。

```
；参数：200H 字单元为 ISD4004 内的具体起始放音地址，
；          202H 字节单元为 ISD4004 指令
PLAY_OUT      BIC.B     #1,&P2OUT             ;SS=0
                CALL      #SEND_16
                MOV.B     #11100000B,&202H    ;SET PLAY
                CALL      #SEND_8
                BIS.B     #1,&P2OUT           ;SS=1
                NOP
                NOP
                BIC.B     #1,&P2OUT             ;SS=0
```

```
                MOV.B    #11110000B,&202H
                CALL     #SEND_8
                BIS.B    #1,&P2OUT          ;SS=1
                RET
```

MSP430 与 ISD4004 的串行接口程序，串行送 16 位地址程序如下：

```
; 参数：200H 字单元为 ISD4004 内的具体起始录音/放音地址，
SEND_16         MOV          &200H,R13
                MOV          #16,R14
SENDDATALOOP    BIC.B        #8H,&P2OUT     ;SCLK=0    P2.3
                BIT          #001H,R13
                JNZ          SEND_11        ;如果为 1，送 1，否则送 0
                BIC.B        #2,&P2OUT      ;如果为 0,MOSI=0     P2.1
                JMP          SONGE1
SEND_11         BIS.B        #2,&P2OUT      ; 如果为 1,MOSI=1
SONGE1          BIS.B        #8H,&P2OUT         ;SLCK=1
                RRA           R13
                BIC.B        #8H,&P2OUT          ;SCLK=0
                DEC           R14
                JNZ          SENDDATALOOP
                RET
```

MSP430 与 ISD4004 的串行接口程序，串行送 8 位指令程序如下：

```
; 参数：  202H 字节单元为 ISD4004 指令
SEND_8          MOV.B        &202H,R13
                MOV          #8,R14
SENDILOOP       BIC.B        #8H,&P2OUT     ;SCLK=0
                BIT          #001H,R13
                JNZ          SEND_1         ;如果为 1，送 1，否则送 0
                BIC.B        #2,&P2OUT      ;如果为 0,MOSI=0
                JMP          SONGE
SEND_1          BIS.B        #2,&P2OUT      ; 如果为 1,MOSI=1
SONGE           BIS.B        #8H,&P2OUT         ;SLCK=1
                RRA           R13
                NOP
                NOP
                BIC.B        #8H,&P2OUT          ;SCLK=0
                DEC           R14
                JNZ          SENDILOOP
                RET
                COMMON    INTVEC                          ; Interrupt vectors
```

```
ORG     RESET_VECTOR
DW      Reset
END
```

3.11　时间控制器的设计（实验 25）

在实际生活中，时间控制器的使用是非常广泛的。大家很熟悉的打铃系统；酿造厂的发酵罐等等。下面以打铃系统为例，讲述时间控制系统的设计问题（程序以 C 语言为主）。

1　系统设计

整个系统需要一个系统时钟，作为控制器的时间标准。控制输出的时间设定、系统时钟的校准、控制输出的准许与否等，都需要输入设备，这里以前面讲的 12 按键键盘为输入设备。输出使用三极管驱动蜂鸣器。在所有的操作过程中，使用前面讲的 6 位数码管作为显示，显示操作标志，以及输入的数据。（硬件电路略，可参照前面相应部分）

键盘约定

“0”号键为时间设置键与修改功能键。

“1”号键为打铃时间设定功能键。

“11”号键为打铃输出与否修改功能键。

显示界面约定

（1） 走时：系统在没有操作时为走时状态。显示界面样式为：13. 0723.第二位的小数点为秒信号闪烁。最后一位的小数点表示系统的控制是否输出，小数点亮表示每当到设定时间，则输出打铃；小数点熄表示即使到设定时间，也不打铃。为了符合人们习惯，如果小时的十位为“0”，则不予显示。

（2） 整系统时间：功能键为“0”号按键。显示界面样式为：--0723。最左边两位显示中间杠，后面四位紧接着显示输入的设定时间，只有小时与分钟，没有秒。0723 表示输入为现在时间上午 7 点 23 分。

（3） 控制输出时间的设定：功能键为“1”号按键。这个程序指示一个示例，一共设有 6 个打铃输出时间。显示界面为：当按下设定键后，显示为样式： 1 - 0745。最左边的“1 -”表示第一个时间设定点；0745 为原先设定的打铃时间。如果需要修改，则按“0”号键，之后显示界面变为 1 -　　（后面 4 位不显示）。表示第一个打铃设定时间点将被修改，接着由键盘输入数据。如果不需要修改，则按任何非“0”号按键。当设置完毕，按任何键则进入下一个打铃时间设置点，界面样式为：1 - 0745。其余操作与刚才相同。当 6 个打铃点设置完毕，返回走时状态。

全局变量定义

首先走时系统的需使用 5 个变量：（都是字节变量）

s01　0.1 秒计数单元

y3　秒计数单元

y2　分计数单元

y1　小时计数单元

dot　时钟界面中秒闪烁信号指示单元

对于打铃时间，使用了一个结构数组：（下面已经有了初始值）

```
        struct
          {uchar hour,mte;}time[6]={{6,0},{6,5},{6,10},{7,40},{13,40},{17,20}};
```

还需要 6 个显示缓存单元：

```
        x[6]={1,2,0,0,0,0}
```

以及打铃输出控制单元：（初始值为“1”，表示要输出打铃）

```
        bellcontr=1
```

则系统主程序如下：

```
#include <msp430x13x.h>
#define uchar unsigned char
#define unit unsigned int
uchar dot=0, bellcontr=1;
uchar x[6]={1,2,0,0,0,0},y1=20,y2=00,y3=00,s01; /*显缓，计时单元*/
struct                                          //打铃 6 时间
  {uchar hour,mte;}time[6]={{6,0},{6,5},{6,10},{7,40},{13,40},{17,20}};
/*time[5]为 6 个时间控制点*/
const uchar seg[]={0x3f,0x06,0x5b,0x4f,          //显示码表
    0x66,0x6d,0x7d,0x07,
    0x7f,0x6f,0x77,0x7c,
    0x39,0x5e,0x79,0x71,
    0xbf,0x86,0xdb,0xcf,
    0xe6,0xed,0xfd,0x87,
    0xff,0xef,0xf7,0xfc,
    0xb9,0xde,0xf9,0xf1,
    0x80,0x40,0x00,0x73,0xc0};
void main(void)                                  //系统主程序
{
    WDTCTL = WDTPW + WDTHOLD;                    //停止看门狗
          TACTL = TASSEL1 + TACLR;               //设置定时器 A
          CCTL0 = CCIE;                          //使能 CCR0 中断
          CCR0 = 50000;                          //CCR0 初始值
          P4DIR |= 0x08;                         //端口 P4.3 用于驱动电铃
          TACTL |= MC1;                          //开始定时器
           _EINT();                              //开主中断

       while(1)                                  //主循环
          {
          uchar j;
              x[0]=y1/10;          //拆分时间值到对应的显示缓存
              if(x[0]==0)           //如果时间的小时值的十位为“0”
              x[0]=0x22;            //则不予显示（符合习惯）
```

```
  if(dot==1)              //与电子表对应的秒闪烁
  x[1]=y1%10+0x10;
  else
  x[1]=y1%10;             //拆分时间值到对应的显示缓存
  x[2]=y2/10;
  x[3]=y2%10;
  x[4]=y3/10;
  if(bellcontr==1)        //输出控制位在最后单元的小数点显示
  x[5]=y3%10+0x10;    //如果控制信号输出，则最后小数点亮
  else
  x[5]=y3%10;

  dsp();                  //调用显示
if(kbjust()!=0)           //有否按下的按键
     {                    //如果有，则：
       switch(kbscan()) //判断什么键值
           {
            case 0:             //0：调时间
                {
                 key0();
                 break;
                  }
             case 1:            //1：设定时间控制点
                 {key1();
                 break;
                 }
            case 11:            //11：设定到时间控制点后，是否输出
                 {key3();
                  break;
                  }
           }
      }
if(bellcontr ==1)               //如果输出畅通
   {
   for(j=0;j<6;j++)             //判断实时时间与 6 个控制点是否有相等
    if((y1==time[j].hour)&(y2==time[j].mte))
     {
     P4OUT |= BIT3;             //则输出高电平驱动打铃
     break;
     }
     else
```

```
                    P4OUT &= ~BIT3;       //不满足打铃条件输出低电平
                    }
        }
}
```

2　系统时钟

既然是时间控制系统，则首先有系统时钟。系统时钟完成两件事：走时，提供标准时钟；其二为为显示界面提供闪烁的秒信号。这个程序很简单，使用 Timer_A 中断。使用变量单元在主程序中有定义，为全局变量：s01　0.1 秒计数单元

y3　秒计数单元

y2　分计数单元

y1　小时计数单元

dot　时钟界面中秒闪烁信号指示单元

```
interrupt[TIMERA0_VECTOR] void Timer_A (void)
{
CCR0+=60000;                  //增加 CCR0
s01=s01+1;                    //0.1 秒加一
if(s01==10)
          {s01=0;
          dot^=1;                 //求反,小数点是否显示(秒闪烁)
          y3=y3+1;                //10 个 0.1 秒为 1 秒,秒加一
          }
    if(y3==60)
        {
          y3=0;
          y2=y2+1;                //60 秒为 1 分,分加一
          if(y2==60)
            {
            y2=0;
            y1=y1+1;              //60 分为 1 小时,小时加一
            if(y1==24)
                {y1=0 ;           //24 小时再清零
                 }
              }
         }
}
```

3　走时时钟的调整

在有时钟的系统中，时钟的调整是必不可少的。根据最初的约定，按照操作过程与相应的显示格式编写程序。先初始化显示界面，再显示按键号码，最后将输入数据计算为时间值，

具体说明请参考程序中的解释。程序如下：

```
void key0(void)                 //设定实时时间
      {
      x[0]=0x21;
      x[1]=0x21;                 //最前面两显示器显示中间杠——，提示为时间设置
      x[2]=0x22;
      x[3]=0x22;
      x[4]=0x22;
      x[5]=0x22;                 //接着后面 4 位不显示，等待键盘的输入
      x[2]=kbscan();
      x[3]=kbscan();
      x[4]=kbscan();
      x[5]=kbscan();             //键盘输入值送入后面 4 位显示器
      y1=x[2];
      y1=y1*10;
      y1=y1+x[3];
      y1=x[2]*10+x[3];          //前两个输入数据作为时间的小时值被保存
      y2=x[4]*10+x[5];          //最后两个输入作为时间的分钟值被保存
      y3=0;                      //秒被清零
      }
```

4　打铃控制时间的设定

在本系统中可以设定 6 个打铃时间，（当然可以设置很多）每当走时时间与设置的打铃控制时间相同时，同时允许打铃，则输出到电铃。本子程序完成 6 个打铃时间的输入与修改。程序按照前面的约定编写。请参考程序中的注释。程序如下：

```
void key1(void)                          //设定 6 个控制时间点
 {
      uchar tmp;
      uchar i=0;
      for(i=0;i<6;i++)                   //循环 6 次
       {
      x[0]=i+1;                          //第一位显示控制时间点的序号
      x[1]=timjust[i];                    //实现设定序号的自动增加
        x[2]=(time[i].hour)/10;           //在显示器的后面 4 位显示原先的控制时间点与序号
        x[3]=(time[i].hour)%10;           //先调出原来的设定值
        x[4]=(time[i].mte)/10;
        x[5]=(time[i].mte)%10;
        tmp=kbscan();                     //按键为何值
         if(tmp==0)                       //如果按键为“0”则修改，否则不修改，保持不变
```

```
            {
            x[2]=0x22;                  //说明需要改变相应的控制时间点
            x[3]=0x22;                  //先将显示清除，序号不变
            x[4]=0x22;
            x[5]=0x22;
            x[2]=kbscan();              //输入四位数为改变后的控制时间点
            x[3]=kbscan();
            x[4]=kbscan();
            x[5]=kbscan();
            time[i].hour=x[2]*10+x[3]; //记录如相应的缓存
            time[i].mte=x[4]*10+x[5];
            kbscan();                   //任意按键表示设定完成
            }
          else
            if(tmp==0x0b)
            x[1]=0x21;
            else x[1]=0x22;
            timjust[i]=x[1];
            kbscan();
      }
}
```

5　打铃输出控制

通过“11”号按键设定打铃输出还是不输出，这个程序最简单。只是开关量的改变，控制变量 bellcontr 被初始化为“1”，如最低位求反，则为“0”，再求反，又为“1”。程序如下：

```
void key3(void)                         //控制输出与否位求反
    {
    bellcontr^=1;
    }
```

3.12　简易存储示波器的设计（实验 26）

－－MSP430 的 ADC12、运算放大器、TA、串口综合应用

前几讲讲述了 ADC12、TA、串口等片内外围设备的情况，再加上一个新的模块－－运算放大器，这一讲将它们进行综合应用，设计一个简易存储示波器。此设计的目标如下：

输入信号的频率再 DC～20KHz；

输入信号的电压在 20 毫伏～2 伏；

输入信号的波形通过串口传输到 PC 机显示；

输入信号波形 1.5K 深度存储。

一　设计分析与设计思路以及资源分配：

首先，输入信号的频率在 DC～20KHz 范围。则根据采样定理要求采样频率在 40KHz 内可调整。而 ADC12 的最高采样频率在 100KHz，就将采样频率设计在 100KHz 内可调。这里使用定时器 TA 进行定时采样，通过调整 TA 的定时时间到达在指定频率采样的目的。

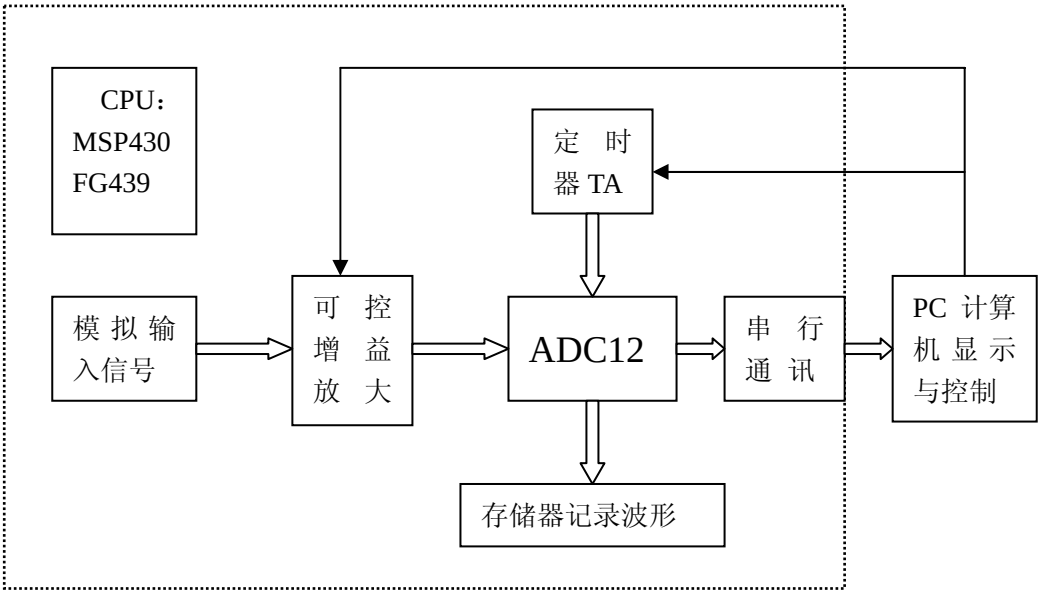
其次，模拟输入信号的电压范围在 20 毫伏～2 伏内，而 ADC12 的参考电压为 1.5V、2.5V，要想达到这么大的动态范围，只有将输入信号调理。这里运用 MSP430 片内自有的运算放大器实现。MSP430 片内有 3 只可编程的运算放大器，每只放大器的放大倍数在 1～16 倍可调，则可以方便地调节输入信号进入 ADC12 的幅度。

第三，使用串口送采集的数据到 PC 机是很方便的。

第四，在 PC 机端写参数选择面板与示波器波形显示屏程序。

最后，示波器的波形能存储以便分析。这里利用 MSP430 片内自有的 2K 字节 RAM 可以方便实现。

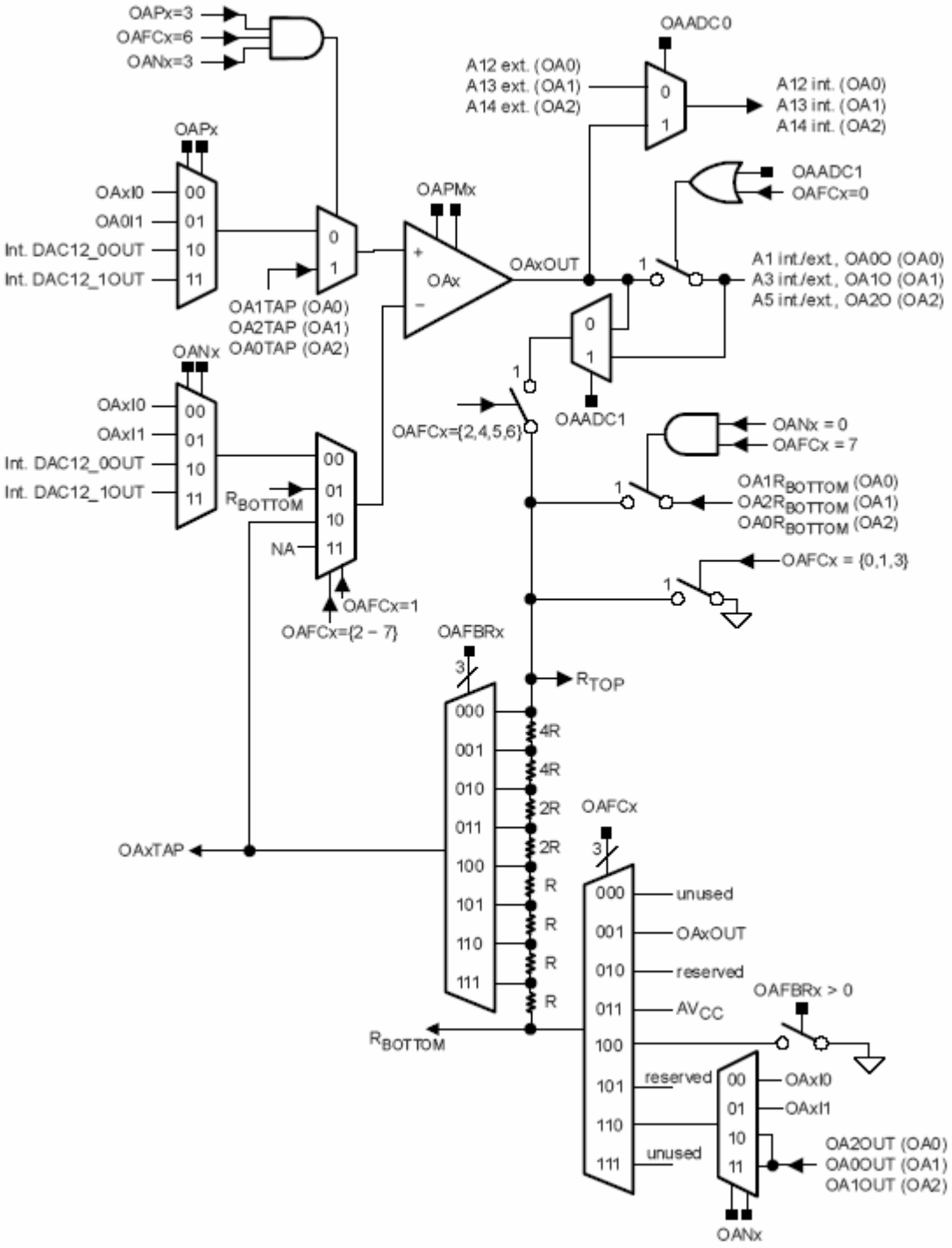
综上所述，此简易存储示波器设计框图如下：



二　具体各部分设计

至此，有了大致的设计思路。首先要介绍 MSP430 片内的运算放大器。因为模拟输入信号最终需要放大（/缩小）到 1.5V、2.5V 内，只有使用运算放大器实现，而运算放大器在

MSP430FG43 系列中，所以本设计将使用 MSP430FG439，MSP430FG439 的片内资源为：定时器 TA、TB、ADC12、运算放大器 OA、通用串口、看门狗等。下图是 MSP430FG439 片内运算放大器的结构框图。



以上运算放大器的结构框图为单只运放的框图，其他两只完全一样。在 MSP430FG439 芯片中，三只运算放大器的引脚都在 P6 端口上。

每只运算放大器的所有功能在两个寄存器中实现：OAiCTL1、OAiCTL0。寄存器各位含义简介如下：

OAiCTL0 中各位：

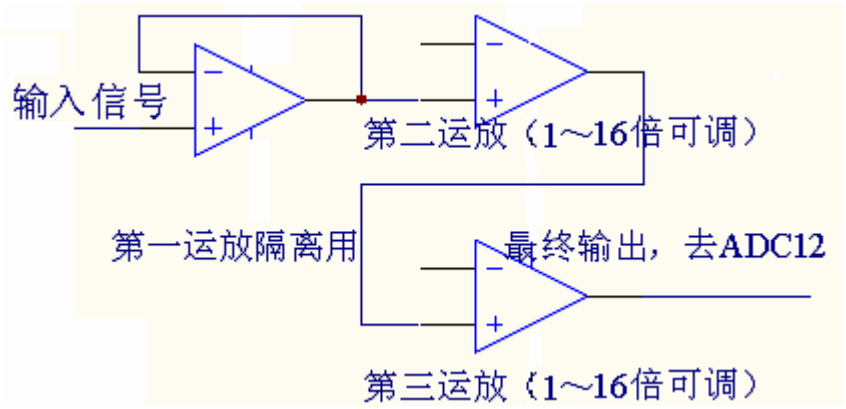
7、6	5、4	3、2	1	0
负输入引脚	正输入引脚	速度控制	输出到端口	输出到端口

OAiCTL1 中各位：

7、6、5	4、3、2	1	0
放大倍数选择	模式选择	保留	轨到轨

通过以上寄存器的各位可以方便地使用放大器。比如要改变放大倍数，直接更改 OAiCTL1 中的第 5、6、7 位的数值即可。OAiCTL1 中的第 5、6、7 位的数值改变将直接影响图中运算放大器反馈电阻的阻值改变：反馈电阻中 4 个 R、两个 2R 两个 4R 的不同接入方式。下面的关键就是如何运用此运放的可方便更改放大倍数这一特点来实现宽的输入电压范围。先看构想。

在了解了运算放大器的使用之后，下面在本设计中配置这三个放大器：第一个放大器用于输入信号与本示波器的隔离，第二第三放大器用于信号放大。本设计思路：使用第一放大器射极跟随（信号隔离），然后根据信号的大小，再调节放大器的放大倍数到需要的值（能在 PC 屏幕上显示比较满意的波形）。只有两级放大，放大器的倍数最大为 256 倍，根据需要，输入信号的幅度范围为 10 毫伏～2 伏，则需要放大 200 倍，完全能满足要求。所以输入信号在 1～200 倍可调放大。电路图如下：



在输入信号为 10 毫伏时，运放放大 200 倍，幅度将达到 2V，在 ADC12 的参考电压取 2.5V 时，转换数据为 2/2.5*4096 = 3276。

在输入信号为 200 毫伏时，运放放大 10 倍，幅度将达到 2V，在 ADC12 的参考电压取 2.5V 时，转换数据为 2/2.5*4096 = 3276。

在输入信号为 2 伏时，运放放大 1 倍，幅度将达到 2V，在 ADC12 的参考电压取 2.5V 时，转换数据为 2/2.5*4096 = 3276。

根据以上分析，如此配置运放是完全可行的。

ADC12 的配置应该为：

片内参考电压；

参考电压 2.5V；

P60 为模拟输入信号 A0，如果设计为多通道示波器，则使用 A1～A7 做其他模拟输入通道，此处为单通道，仅使用 A0；

采样使用主动读取方式，非 ADC12 中断。

采样与保持的时间取最小值，主要为了兼顾最快采样。

下面考虑定时器的设置。

定时器使用 TA，TA 使用 SMCLK，SMCLK 设置为 8MHz，这样，采样频率可以方便调整。本设计要求：输入信号在 DC～20KHz。所以，采样频率最高取 100KHz，最低取 300Hz（300Hz 可以利用视觉暂停避免闪烁），TA 设置如下：

运行在连续计数模式；

SMCLK 为输入时钟；

产生中断，在中断服务程序中实现 ADC12 采样；

定时长度的改变由 PC 机操作实现。

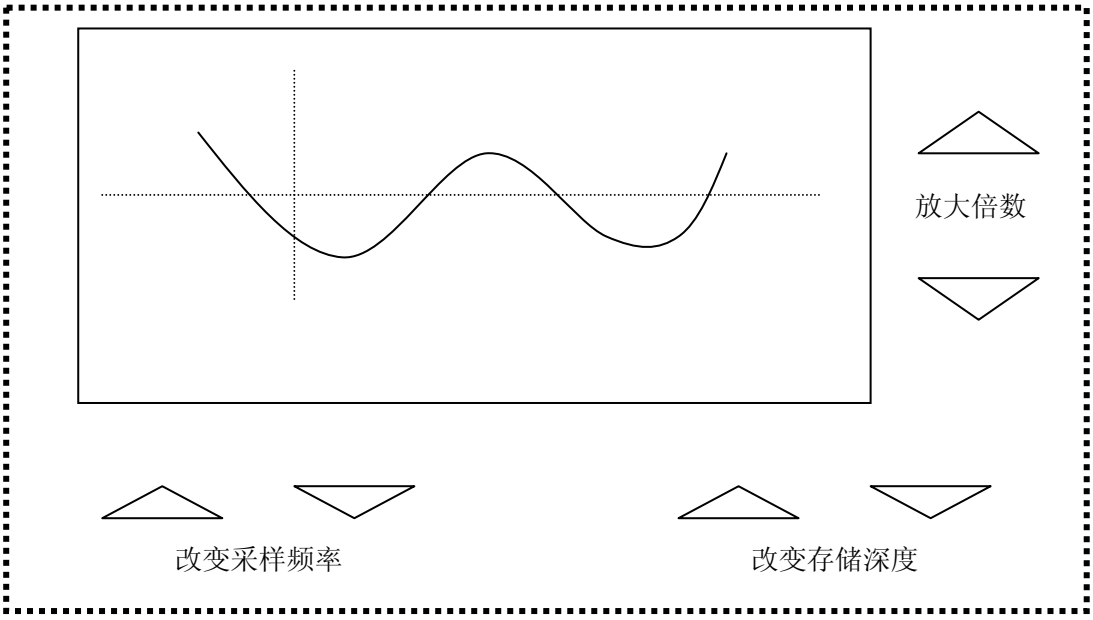
下面设计通讯部分。

使用普通 RS232 串口与计算机通讯，分上传与下达两部分。

上传数据包括：模拟量的 ADC 转换值、采样频率、放大倍数的大小、本次存储的数据长度等信息。

下传的数据包括：要求的采样频率增大还是减小、要求放大倍数增大还是减小、要求本次存储深度等信息。

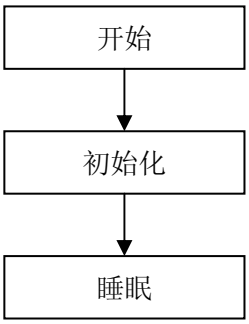
所以，PC 机的屏幕安排如下：三个参数直接使用加减更改，之后下传采集终端。



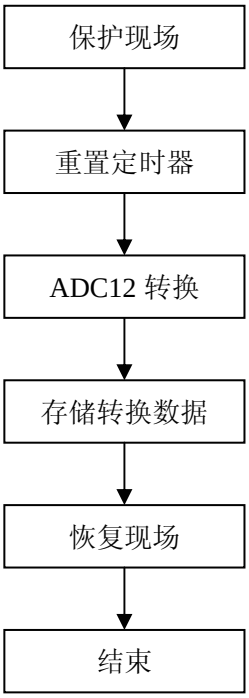
三　程序框图

MSP430 方面需要完成一下几部分事情：主程序编写、定时器中断服务程序的编写、串口通讯程序、定时器中断服务程序、ADC12 转换程序、运算放大器程序等。

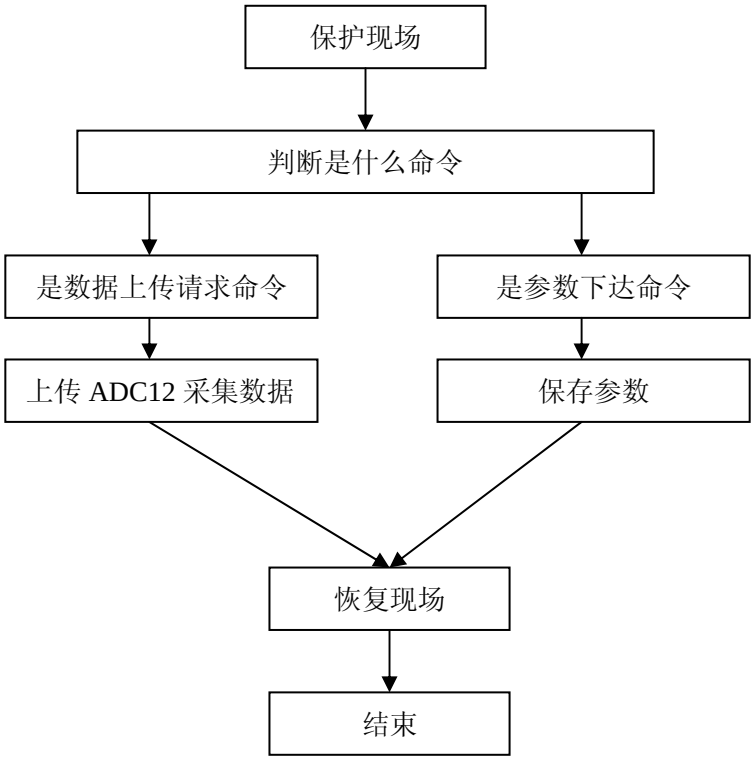
主程序主要完成初始化工作，框图如下：



定时器中断服务程序完成定时采样，整个时序的搭配控制等。所以 ADC12 转换程序涵盖在定时器中断服务程序中。定时参数的改变在此程序中完成。框图如下：



串口通讯程序需要完成通讯数据的上传与下达。数据上传为主动方式，接收 PC 机的数据为被动方式，采用中断。在接收到 PC 机的数据上传请求命令时，上传数据。



四　PC 机程序设计

在 PC 机端主要完成由采集终端送来数据的显示、多个命令参数的更改同时下达到采集终端。

在显示波形时，横坐标为时间，纵坐标为电压（ADC12 结果）。使用 VB、VC 编写（略）。

参数改变：当鼠标点击减小按钮时，对应参数减小一半，当点击增加按钮时，相应参数增加一倍，同时参数下达到采集终端（程序略）。

（以下内容略）

3．13　出租车计价器设计（实验 27）
3．14　电子水表设计（实验 28）
3．15　复杂多相位交通灯设计（实验 29）
3．16　IIC 总线实践（实验 30）
3．17　基于蓝牙的温度数据采集实践（实验 31）
3．18　图形液晶汉字显示设计（实验 32）
3．19　LED 点阵汉字屏显示设计（实验 33）