

13 指令系统

13.1 寻址方式

寻址方式是每一种计算机的指令集中不可缺少的部分。寻址方式规定了数据的来源和目的地。对不同的程序指令，来源和目的地的规定也会不同。在 STC 单片机中的寻址方式可概括为：

- 立即寻址
- 直接寻址
- 间接寻址
- 寄存器寻址
- 相对寻址
- 变址寻址
- 位寻址

13.1.1 立即寻址

立即寻址也称立即数，它是在指令操作数中直接给出参加运算的操作数，其指令格式如下：

如：MOV A, #70H

这条指令的功能是将立即数 70H 传送到累加器 A 中。

13.1.2 直接寻址

在直接寻址方式中，指令操作数域给出的是参加运算操作数地址。直接寻址方式只能用来表示特殊功能寄存器、内部数据寄存器和位地址空间。其中特殊功能寄存器和位地址空间只能用直接寻址方式访问。

如：ANL 70H, #48H

表示 70H 单元中的数与立即数 48H 相“与”，结果存放在 70H 单元中。其中 70H 为直接地址，表示内部数据存储器 RAM 中的一个单元。

13.1.3 间接寻址

间接寻址采用 R0 或 R1 前添加“@”符号来表示。例如，假设 R1 中的数据是 40H，内部数据存储器 40H 单元所包含的数据为 55H，那么如下指令：

MOV A, @R1

把数据 55H 传送到累加器。

13.1.4 寄存器寻址

寄存器寻址是对选定的工作寄存器 R7~R0、累加器 A、通用寄存器 B、地址寄存器和进位 C 中的数进行操作。其中寄存器 R7~R0 由指令码的低 3 位表示，ACC、B、DPTR 及进位位 C 隐含在指令码中。因此，寄存器寻址也包含一种隐含寻址方式。

寄存器工作区的选择由程序状态字寄存器 PSW 中的 RS1、RS0 来决定。指令操作数指定的寄存器均指当前工作区中的寄存器。

如：INC R0 ;(R0)+1 → R0

13.1.5 相对寻址

相对寻址是将程序计数器 PC 中的当前值与指令第二字节给出的数相加，其结果作为转移指令的转移地址。转移地址也称为转移目的地址，PC 中的当前值称为基地址，指令第二字节给出的数称为偏移量。由于目的地址是相对于 PC 中的基地址而言，所以这种寻址方式称为相对寻址。偏移量为带符号的数，所能表示的范围为+127 ~ -128。这种寻址方式主要用于转移指令。

如: JC 80H ;C=1 跳转

表示若进位位 C 为 0, 则程序计数器 PC 中的内容不改变, 即不转移。若进位位 C 为 1, 则以 PC 中的当前值为基址, 加上偏移量 80H 后所得到的结果作为该转移指令的目的地址。

13.1.6 变址寻址

在变址寻址方式中, 指令操作数指定一个存放变址基值的变址寄存器。变址寻址时, 偏移量与变址基值相加, 其结果作为操作数的地址。变址寄存器有程序计数器 PC 和地址寄存器 DPTR。

如: MOVC A, @A+DPTR

表示累加器 A 为偏移量寄存器, 其内容与地址寄存器 DPTR 中的内容相加, 其结果作为操作数的地址, 取出该单元中的数送入累加器 A。

13.1.7 位寻址

位寻址是指对一些内部数据存储器 RAM 和特殊功能寄存器进行位操作时的寻址。在进行位操作时, 借助于进位位 C 作为位操作累加器, 指令操作数直接给出该位的地址, 然后根据操作码的性质对该位进行位操作。位地址与字节直接寻址中的字节地址形式完全一样, 主要由操作码加以区分, 使用时应注意。

如: MOV C, 20H ;片内位单元操作型指令

13.2 指令表

助记符	指令说明	字节	时钟
ADD A,Rn	寄存器内容加到累加器	1	1 ^[4]
ADD A,direct	直接地址单元的数据加到累加器	2	1
ADD A,@Ri	间接地址单元的数据加到累加器	1	1
ADD A,#data	立即数加到累加器	2	1
ADDC A,Rn	寄存器带进位加到累加器	1	1
ADDC A,direct	直接地址单元的数据带进位加到累加器	2	1
ADDC A,@Ri	间接地址单元的数据带进位加到累加器	1	1
ADDC A,#data	立即数带进位加到累加器	2	1
SUBB A,Rn	累加器带借位减寄存器内容	1	1
SUBB A,direct	累加器带借位减直接地址单元的内容	2	1
SUBB A,@Ri	累加器带借位减间接地址单元的内容	1	1
SUBB A,#data	累加器带借位减立即数	2	1
INC A	累加器加1	1	1
INC Rn	寄存器加1	1	1
INC direct	直接地址单元加1	2	1
INC @Ri	间接地址单元加1	1	1
DEC A	累加器减1	1	1
DEC Rn	寄存器减1	1	1
DEC direct	直接地址单元减1	2	1
DEC @Ri	间接地址单元减1	1	1
INC DPTR	地址寄存器DPTR加1	1	1

助记符	指令说明	字节	时钟
MUL AB	A乘以B, B存放高字节, A存放低字节	1	2
DIV AB	A除以B, B存放余数, A存放商	1	6
DA A	累加器十进制调整	1	3
ANL A,Rn	累加器与寄存器相与	1	1
ANL A,direct	累加器与直接地址单元相与	2	1
ANL A,@Ri	累加器与间接地址单元相与	1	1
ANL A,#data	累加器与立即数相与	2	1
ANL direct,A	直接地址单元与累加器相与	2	1
ANL direct,#data	直接地址单元与立即数相与	3	1
ORL A,Rn	累加器与寄存器相或	1	1
ORL A,direct	累加器与直接地址单元相或	2	1
ORL A,@Ri	累加器与间接地址单元相或	1	1
ORL A,#data	累加器与立即数相或	2	1
ORL direct,A	直接地址单元与累加器相或	2	1
ORL direct,#data	直接地址单元与立即数相或	3	1
XRL A,Rn	累加器与寄存器相异或	1	1
XRL A,direct	累加器与直接地址单元相异或	2	1
XRL A,@Ri	累加器与间接地址单元相异或	1	1
XRL A,#data	累加器与立即数相异或	2	1
XRL direct,A	直接地址单元与累加器相异或	2	1
XRL direct,#data	直接地址单元与立即数相异或	3	1
CLR A	累加器清0	1	1
CPL A	累加器取反	1	1
RL A	累加器循环左移	1	1
RLC A	累加器带进位循环左移	1	1
RR A	累加器循环右移	1	1
RRC A	累加器带进位循环右移	1	1
SWAP A	累加器高低半字节交换	1	1
CLR C	清零进位位	1	1
CLR bit	清0直接地址位	2	1
SETB C	置1进位位	1	1
SETB bit	置1直接地址位	2	1
CPL C	进位位求反	1	1
CPL bit	直接地址位求反	2	1
ANL C,bit	进位位和直接地址位相与	2	1
ANL C,/bit	进位位和直接地址位的反码相与	2	1

助记符	指令说明	字节	时钟
ORL C,bit	进位位和直接地址位相或	2	1
ORL C,/bit	进位位和直接地址位的反码相或	2	1
MOV C,bit	直接地址位送入进位位	2	1
MOV bit,C	进位位送入直接地址位	2	1
MOV A,Rn	寄存器内容送入累加器	1	1
MOV A,direct	直接地址单元中的数据送入累加器	2	1
MOV A,@Ri	间接地址中的数据送入累加器	1	1
MOV A,#data	立即数送入累加器	2	1
MOV Rn,A	累加器内容送入寄存器	1	1
MOV Rn,direct	直接地址单元中的数据送入寄存器	2	1
MOV Rn,#data	立即数送入寄存器	2	1
MOV direct,A	累加器内容送入直接地址单元	2	1
MOV direct,Rn	寄存器内容送入直接地址单元	2	1
MOV direct,direct	直接地址单元中的数据送入另一个直接地址单元	3	1
MOV direct,@Ri	间接地址中的数据送入直接地址单元	2	1
MOV direct,#data	立即数送入直接地址单元	3	1
MOV @Ri,A	累加器内容送入间接地址单元	1	1
MOV @Ri,direct	直接地址单元数据送入间接地址单元	2	1
MOV @Ri,#data	立即数送入间接地址单元	2	1
MOV DPTR,#data16	16位立即数送入数据指针	3	1
MOVC A,@A+DPTR	以DPTR为基址变址寻址单元中的数据送入累加器	1	4
MOVC A,@A+PC	以PC为基址变址寻址单元中的数据送入累加器	1	3
MOVX A,@Ri	扩展地址(8位地址)的内容送入累加器A中	1	3 ^[1]
MOVX A,@DPTR	扩展RAM(16位地址)的内容送入累加器A中	1	2 ^[1]
MOVX @Ri,A	将累加器A的内容送入扩展RAM(8位地址)中	1	3 ^[1]
MOVX @DPTR,A	将累加器A的内容送入扩展RAM(16位地址)中	1	2 ^[1]
PUSH direct	直接地址单元中的数据压入堆栈	2	1
POP direct	栈底数据弹出送入直接地址单元	2	1
XCH A,Rn	寄存器与累加器交换	1	1
XCH A,direct	直接地址单元与累加器交换	2	1
XCH A,@Ri	间接地址与累加器交换	1	1
XCHD A,@Ri	间接地址的低半字节与累加器交换	1	1
ACALL addr11	短调用子程序	2	3
LCALL addr16	长调用子程序	3	3
RET	子程序返回	1	3
RETI	中断返回	1	3

助记符	指令说明	字节	时钟
AJMP addr11	短跳转	2	3
LJMP addr16	长跳转	3	3
SJMP rel	相对跳转	2	3
JMP @A+DPTR	相对于DPTR的间接跳转	1	4
JZ rel	累加器为零跳转	2	1/3 ^[2]
JNZ rel	累加器非零跳转	2	1/3 ^[2]
JC rel	进位位为1跳转	2	1/3 ^[2]
JNC rel	进位位为0跳转	2	1/3 ^[2]
JB bit,rel	直接地址位为1则跳转	3	1/3 ^[2]
JNB bit,rel	直接地址位为0则跳转	3	1/3 ^[2]
JBC bit,rel	直接地址位为1则跳转, 该位清0	3	1/3 ^[2]
CJNE A,direct,rel	累加器与直接地址单元不相等跳转	3	2/3 ^[3]
CJNE A,#data,rel	累加器与立即数不相等跳转	3	1/3 ^[2]
CJNE Rn,#data,rel	寄存器与立即数不相等跳转	3	2/3 ^[3]
CJNE @Ri,#data,rel	间接地址单元与立即数不相等跳转	3	2/3 ^[3]
DJNZ Rn,rel	寄存器减1后非零跳转	2	2/3 ^[3]
DJNZ direct,rel	直接地址单元减1后非零跳转	3	2/3 ^[3]
NOP	空操作	1	1

^[1]: 访问外部扩展 RAM 时, 指令的执行周期与寄存器 BUS_SPEED 中的 SPEED[2:0]位有关

^[2]: 对于条件跳转语句的执行时间会依据条件是否满足而不同。当条件不满足时, 不会发生跳转而继续执行下一条指令, 此时条件跳转语句的执行时间为 1 个时钟; 当条件满足时, 则会发生跳转, 此时条件跳转语句的执行时间为 3 个时钟。

^[3]: 对于条件跳转语句的执行时间会依据条件是否满足而不同。当条件不满足时, 不会发生跳转而继续执行下一条指令, 此时条件跳转语句的执行时间为 2 个时钟; 当条件满足时, 则会发生跳转, 此时条件跳转语句的执行时间为 3 个时钟。

[4]: 上表中每条指令的执行时钟数为实际测量的数据, 下面指令描述部分为原始规格, 部分指令的执行时钟数可能与实际不符, 请以上表的数据为准。

13.3 指令详解 (中文)

ACALL addr11

功能: 绝对调用

说明: ACALL 指令实现无条件调用位于 addr11 参数所表示地址的子例程。在执行该指令时, 首先将 PC 的值增加 2, 即使得 PC 指向 ACALL 的下一条指令, 然后把 16 位 PC 的低 8 位和高 8 位依次压入栈, 同时把栈指针两次加 1。然后, 把当前 PC 值的高 5 位、ACALL 指令第 1 字节的 7~5 位和第 2 字节组合起来, 得到一个 16 位目的地址, 该地址即为即将调用的子例程的入口地址。要求该子例程的起始地址必须与紧随 ACALL 之后的指令处于同 1 个 2KB 的程序存储页中。ACALL 指令在执行时不会改变各个标志位。

举例: SP 的初始值为 07H, 标号 SUBRTN 位于程序存储器的 0345H 地址处, 如果执行位于地址 0123H

处的指令:

ACALL SUBRTN

那么 SP 变为 09H, 内部 RAM 地址 08H 和 09H 单元的内容分别为 25H 和 01H, PC 值变为 0345H。

指令长度(字节):

2

执行周期:

3

二进制编码:

A10	A9	A8	1	0	0	0	1		A7	A6	A5	A4	A3	A2	A1	A0
-----	----	----	---	---	---	---	---	--	----	----	----	----	----	----	----	----

注意: a10 a9 a8 是 11 位目标地址 addr11 的 A10~A8 位, a7 a6 a5 a4 a3 a2 a1 a0 是 addr11 的 A7~A0 位。

操作: ACALL

(PC)←(PC)+ 2

(SP)←(SP) + 1

((SP)) ←(PC₇₋₀)

(SP)←(SP) + 1

((SP))←(PC₁₅₋₈)

(PC₁₀₋₀)←页码地址

ADD A, <src-byte>

功能: 加法

说明: ADD 指令可用于完成把 src-byte 所表示的源操作数和累加器 A 的当前值相加。并将结果置于累加器 A 中。根据运算结果, 若第 7 位有进位则置进位标志为 1, 否则清零; 若第 3 位有进位则置辅助进位标志为 1, 否则清零。如果是无符号整数相加则进位置位, 显示当前运算结果发生溢出。如果第 6 位有进位生成而第 7 位没有, 或第 7 位有进位生成而第 6 位没有, 则置 OV 为 1, 否则 OV 被清零。在进行有符号整数的相加运算的时候, OV 置位表示两个正整数之和为一负数, 或是两个负整数之和为一正数。

本类指令的源操作数可接受 4 种寻址方式: 寄存器寻址、直接寻址、寄存器间接寻址和立即寻址。

举例: 假设累加器 A 中的数据为 0C3H(11000011B), R0 的值为 0AAH(10101010B)。执行如下指令:

ADD A, R0

累加器 A 中的结果为 6DH(01101101B), 辅助进位标志 AC 被清零, 进位标志 C 和溢出标志 OV 被置 1。

ADD A, Rn

指令长度(字节):

1

执行周期:

1

二进制编码:

0	0	1	0	1	r	r	r
---	---	---	---	---	---	---	---

操作: ADD

(A)← (A) + (Rn)

ADD A, direct

指令长度(字节):

2

执行周期:

1

二进制编码:

0	0	1	0	0	1	0	1		direct address
---	---	---	---	---	---	---	---	--	----------------

操作: ADD

(A)← (A) + (direct)

ADD A, @Ri

指令长度(字节):

1

执行周期:

1

二进制编码:

0	0	1	0	0	1	1	i
---	---	---	---	---	---	---	---

操作: ADD

$(A) \leftarrow (A) + ((Ri))$

ADD A, #data

指令长度(字节): 2

执行周期: 1

二进制编码:

0	0	1	0	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

操作: ADD

$(A) \leftarrow (A) + \#data$

ADDC A, <src-byte>

功能: 带进位的加法

说明: 执行 ADCD 指令时, 把 src-byte 所代表的源操作数连同进位标志一起加到累加器 A 上, 并将结果置于累加器 A 中。根据运算结果, 若在第 7 位有进位生成, 则将进位标志置 1, 否则清零; 若在第 3 位有进位生成, 则置辅助进位标志为 1, 否则清零。如果是无符号数整数相加, 进位的置位显示当前运算结果发生溢出。

如果第 6 位有进位生成而第 7 位没有, 或第 7 位有进位生成而第 6 位没有, 则将 OV 置 1, 否则将 OV 清零。在进行有符号整数相加运算的时候, OV 置位, 表示两个正整数之和为一负数, 或是两个负整数之和为一正数。

本类指令的源操作数允许 4 种寻址方式: 寄存器寻址、直接寻址、寄存器间接寻址和立即寻址。

举例: 假设累加器 A 中的数据为 0C3H(11000011B), R0 的值为 0AAH(10101010B), 进位标志为 1, 执行如下指令:

ADDC A,R0

累加器 A 中的结果为 6EH(01101110B), 辅助进位标志 AC 被清零, 进位标志 C 和溢出标志 OV 被置 1。

ADDC A, Rn

指令长度(字节): 1

执行周期: 1

二进制编码:

0	0	1	1	1	r	r	r
---	---	---	---	---	---	---	---

操作: ADDC

$(A) \leftarrow (A) + (C) + (Rn)$

ADDC A, direct

指令长度(字节): 2

执行周期: 1

二进制编码:

0	0	1	1	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作: ADDC

$(A) \leftarrow (A) + (C) + (\text{direct})$

ADDC A, @Ri

指令长度(字节): 1

执行周期: 1

二进制编码:

0	0	1	1	0	1	1	i
---	---	---	---	---	---	---	---

操作: ADDC

$(A) \leftarrow (A) + (C) + ((Ri))$

ADDC A, #data

指令长度(字节): 2

执行周期: 1

0	0	1	1	0	1	0	0		immediate data
---	---	---	---	---	---	---	---	--	----------------

操作: ADDC

 $(A) \leftarrow (A) + (C) + \#data$

AJMP addr11

功能: 绝对跳转

说明: AJMP 指令用于将程序转到相应的目的地址去执行, 该地址在程序执行过程之中产生, 由 PC 值 (两次递增之后) 的高 5 位、操作码的 7~5 位和指令的第 2 字节连接形成。要求跳转的目的地址和 AJMP 指令的后一条指令的第 1 字节位于同一 2KB 的程序存储页内。

举例: 假设标号 JMPADR 位于程序存储器的 0123H, 指令:

AJMP JMPADR

位于 0345H, 执行完该指令后 PC 值变为 0123H。

指令长度(字节): 2

执行周期: 3

A10	A9	A8	0	0	0	0	1		A7	A6	A5	A4	A3	A2	A1	A0
-----	----	----	---	---	---	---	---	--	----	----	----	----	----	----	----	----

注意: 目的地址的 A10~A8=a10~a8, A7~A0=a7~a0。

操作: AJMP

 $(PC) \leftarrow (PC) + 2$ $(PC_{10-0}) \leftarrow \text{page address}$

ANL <dest-byte>, <src-byte>

功能: 对字节变量进行逻辑与运算

说明: ANL 指令将由<dest-byte>和<src-byte>所指定的两个字节变量逐位进行逻辑与运算, 并将运算结果存放在<dest-byte>所指定的目的操作数中。该指令的执行不会影响标志位。

两个操作数组合起来允许 6 种寻址模式。当目的操作数为累加器时, 源操作数允许寄存器寻址、直接寻址、寄存器间接寻址和立即寻址。当目的操作数是直接地址时, 源操作数可以是累加器或立即数。

注意: 当该指令用于修改输出端口时, 读入的原始数据来自于输出数据的锁存器而非输入引脚。

举例: 如果累加器的内容为 0C3H(11000011B), 寄存器 0 的内容为 55H(01010101B), 那么指令:

ANL A,R0

执行结果是累加器的内容变为 41H(01000001H)。

当目的操作数是可直接寻址的数据时, ANL 指令可用来把任何 RAM 单元或者硬件寄存器中的某些位清零。屏蔽字节将决定哪些位将被清零。屏蔽字节可能是常数, 也可能是累加器在计算过程中产生。如下指令:

ANL P1, #01110011B

将端口 1 的位 7、位 3 和位 2 清零。

ANL A, Rn

指令长度(字节): 1

执行周期: 1

0	1	0	1	1	r	r	r
---	---	---	---	---	---	---	---

操作: ANL

$(A) \leftarrow (A) \wedge (R_n)$

ANL A, direct

指令长度(字节): 2

执行周期: 1

二进制编码:

0	1	0	1	0	1	0	1	
---	---	---	---	---	---	---	---	--

 direct address

操作: ANL

$(A) \leftarrow (A) \wedge (\text{direct})$

ANL A, @Ri

指令长度(字节): 1

执行周期: 1

二进制编码:

0	1	0	1	0	1	1	i
---	---	---	---	---	---	---	---

操作: ANL

$(A) \leftarrow (A) \wedge ((R_i))$

ANL A, #data

指令长度(字节): 2

执行周期: 1

二进制编码:

0	1	0	1	0	1	0	0	
---	---	---	---	---	---	---	---	--

 immediate data

操作: ANL

$(A) \leftarrow (A) \wedge \#data$

ANL direct, A

指令长度(字节): 2

执行周期: 1

二进制编码:

0	1	0	1	0	0	1	0	
---	---	---	---	---	---	---	---	--

 direct address

操作: ANL

$(\text{direct}) \leftarrow (\text{direct}) \wedge (A)$

ANL direct, #data

指令长度(字节): 3

执行周期: 1

二进制编码:

0	1	0	1	0	0	1	1		
---	---	---	---	---	---	---	---	--	--

 direct address

--	--	--	--

 immediate data

操作: ANL

$(\text{direct}) \leftarrow (\text{direct}) \wedge \#data$

ANL C, <src-bit>

功能: 对位变量进行逻辑与运算

说明: 如果 src-bit 表示的布尔变量为逻辑 0, 清零进位标志位; 否则, 保持进位标志的当前状态不变。

在汇编语言程序中, 操作数前面的 "/" 符号表示在计算时需要先对被寻址位取反, 然后才作为源操作数, 但源操作数本身不会改变。该指令在执行时不会影响其他各个标志位。

源操作数只能采取直接寻址方式。

举例: 下面的指令序列当且仅当 P1.0=1、ACC.7=1 和 OV=0 时, 将进位标志 C 置 1:

MOV C, P1.0 ; LOAD CARRY WITH INPUT PIN STATE

ANL C, ACC.7 ; AND CARRY WITH ACCUM. BIT.7

ANL C, /OV ; AND WITH INVERSE OF OVERFLOW FLAG

ANL C, bit

指令长度(字节): 2

执行周期: 1
 二进制编码:

1	0	0	0	0	0	1	0		bit address
---	---	---	---	---	---	---	---	--	-------------

操作: ANL
 $(C) \leftarrow (C) \wedge (\text{bit})$

ANL C, /bit

指令长度(字节): 2

执行周期: 1
 二进制编码:

1	0	1	1	0	0	0	0		bit address
---	---	---	---	---	---	---	---	--	-------------

操作: ANL
 $(C) \leftarrow (C) \wedge (\overline{\text{bit}})$

CJNE <dest-byte>, <src-byte>, rel

功能: 若两个操作数不相等则转移

说明: CJNE 首先比较两个操作数的大小, 如果二者不等则程序转移。目标地址由位于 CJNE 指令最后 1 个字节的有符号偏移量和 PC 的当前值(紧邻 CJNE 的下一条指令的地址)相加而成。如果目标操作数作为一个无符号整数, 其值小于源操作数对应的无符号整数, 那么将进位标志置 1, 否则将进位标志清零。但操作数本身不会受到影响。

<dest-byte> 和 <src-byte> 组合起来, 允许 4 种寻址模式。累加器 A 可以与任何可直接寻址的数据或立即数进行比较, 任何间接寻址的 RAM 单元或当前工作寄存器都可以和立即常数进行比较。

举例: 设累加器 A 中值为 34H, R7 包含的数据为 56H。如下指令序列:

```
CJNE R7,#60H, NOT_EQ
;
;
NOT_EQ: JC      REQ_LOW      ; IF R7 < 60H.
;
;
```

的第 1 条指令将进位标志置 1, 程序跳转到标号 NOT_EQ 处。接下去, 通过测试进位标志, 可以确定 R7 是大于 60H 还是小于 60H。

假设端口 1 的数据也是 34H, 那么如下指令:

WAIT: CJNE A,P1, WAIT

清除进位标志并继续往下执行, 因为此时累加器的值也为 34H, 即和 P1 口的数据相等。

(如果 P1 端口的数据是其他的值, 那么程序在此不停地循环, 直到 P1 端口的数据变成 34H 为止。)

CJNE A, direct, rel

指令长度(字节): 3

执行周期: 2 或 3

二进制编码:

1	0	1	1	0	1	0	1		direct address		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

操作: $(PC) \leftarrow (PC) + 3$
 $\text{IF } (A) <> (\text{direct})$
 THEN
 $(PC) \leftarrow (PC) + \text{relative offset}$
 $\text{IF } (A) < (\text{direct})$
 THEN

(C) $\leftarrow 1$

ELSE

(C) $\leftarrow 0$

CJNE A, #data, rel

指令长度(字节): 3

执行周期: 1 或 3

二进制编码:	1	0	1	1	0	1	0	0		immediate data		rel. address
--------	---	---	---	---	---	---	---	---	--	----------------	--	--------------

操作: (PC) \leftarrow (PC) + 3

IF (A) $<>$ (data)

THEN

(PC) \leftarrow (PC) +relative offset

IF (A) $<$ (data)

THEN

(C) $\leftarrow 1$

ELSE

(C) $\leftarrow 0$

CJNE Rn, #data, rel

指令长度(字节): 3

执行周期: 2 或 3

二进制编码:	1	0	1	1	1	r	r	r		immediate data		rel. address
--------	---	---	---	---	---	---	---	---	--	----------------	--	--------------

操作: (PC) \leftarrow (PC) + 3

IF (Rn) $<>$ (data)

THEN

(PC) \leftarrow (PC) +relative offset

IF (Rn) $<$ (data)

THEN

(C) $\leftarrow 1$

ELSE

(C) $\leftarrow 0$

CJNE @Ri, #data, rel

指令长度(字节): 3

执行周期: 2 或 3

二进制编码:	1	0	1	1	0	1	1	i		immediate data		rel. address
--------	---	---	---	---	---	---	---	---	--	----------------	--	--------------

操作: (PC) \leftarrow (PC) + 3

IF (Ri) $<>$ (data)

THEN

(PC) \leftarrow (PC) +relative offset

IF (Ri) $<$ (data)

THEN

(C) $\leftarrow 1$

ELSE

(C) $\leftarrow 0$

CLRA

功能: 清除累加器
说明: 该指令用于将累加器 A 的所有位清零, 不影响标志位。
举例: 假设累加器 A 的内容为 5CH(01011100B), 那么指令:
 CLR A
 执行后, 累加器的值变为 00H(00000000B)。

指令长度(字节): 1
执行周期: 1
二进制编码:

1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---

操作: CLR
 $(A) \leftarrow 0$

CLR bit

功能: 清零指定的位
说明: 将 bit 所代表的位清零, 没有标志位会受到影响。CLR 可用于进位标志 C 或者所有可直接寻址的位。
举例: 假设端口 1 的数据为 5DH(01011101B), 那么指令:
 CLR P1.2
 执行后, P1 端口被设置为 59H(01011001B)。

CLR C

指令长度(字节): 1
执行周期: 1
二进制编码:

1	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

操作: CLR
 $(C) \leftarrow 0$

CLR bit

指令长度(字节): 2
执行周期: 1
二进制编码:

1	1	0	0	0	0	1	0	bit address
---	---	---	---	---	---	---	---	-------------

操作: CLR
 $(bit) \leftarrow 0$

CPL A

功能: 累加器 A 求反
说明: 将累加器 A 的每一位都取反, 即原来为 1 的位变为 0, 原来为 0 的位变为 1。该指令不影响标志位。
举例: 设累加器 A 的内容为 5CH(01011100B), 那么指令:
 CPL A
 执行后, 累加器的内容变成 0A3H (10100011B)。

指令长度(字节): 1
执行周期: 1
二进制编码:

1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---

操作: CPL

(A) $\leftarrow (\overline{A})$

CPL bit

功能: 将 bit 所表示的位求反

说明: 将 bit 变量所代表的位取反, 即原来位为 1 的变为 0, 原来为 0 的变为 1。没有标志位会受到影响。CPL 可用于进位标志 C 或者所有可直接寻址的位。

注意: 如果该指令被用来修改输出端口的状态, 那么 bit 所代表的数据是端口锁存器中的数据, 而不是从引脚上输入的当前状态。

举例: 设 P1 端口的数据为 5BH(01011011B), 那么指令:

CPL P1.1

CPL P1.2

执行完后, P1 端口被设置为 5DH(01011101B)。

CPL C

指令长度(字节): 1

执行周期: 1

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

操作: CPL

(C) $\leftarrow (\overline{C})$

CPL bit

指令长度(字节): 2

执行周期: 1

1	0	1	1	0	0	1	0	bit address
---	---	---	---	---	---	---	---	-------------

操作: CPL

(bit) $\leftarrow (\overline{bit})$

DAA

功能: 在加法运算之后, 对累加器 A 进行十进制调整

说明: DA 指令对累加器 A 中存放的由此前的加法运算产生的 8 位数据进行调整(ADD 或 ADDC 指令可以用来实现两个压缩 BCD 码的加法), 生成两个 4 位的数字。

如果累加器的低 4 位(位 3~位 0)大于 9(xxxx1010~xxxx 1111), 或者加法运算后, 辅助进位标志 AC 为 1, 那么 DA 指令将把 6 加到累加器上, 以在低 4 位生成正确的 BCD 数字。若加 6 后, 低 4 位向上有进位, 且高 4 位都为 1, 进位则会一直向前传递, 以致最后进位标志被置 1; 但在其他情况下进位标志并不会被清零, 进位标志会保持原来的值。

如果进位标志为 1, 或者高 4 位的值超过 9(1010xxxx~1111xxxx), 那么 DA 指令将把 6 加到高 4 位, 在高 4 位生成正确的 BCD 数字, 但不清除标志位。若高 4 位有进位输出, 则置进位标志为 1, 否则, 不改变进位标志。进位标志的状态指明了原来的两个 BCD 数据之和是否大于 99, 因而 DA 指令使得 CPU 可以精确地进行十进制的加法运算。注意, OV 标志不会受影响。

DA 指令的以上操作在一个指令周期内完成。实际上, 根据累加器 A 和机器状态字 PSW 中的不同内容, DA 把 00H、06H、60H、66H 加到累加器 A 上, 从而实现十进制转换。

注意: 如果前面没有进行加法运算, 不能直接用 DA 指令把累加器 A 中的十六进制数据转换为 BCD 数, 此外, 如果先前执行的是减法运算, DA 指令也不会有所预期的效果。

举例: 如果累加器中的内容为 56H (01010110B), 表示十进制数 56 的 BCD 码, 寄存器 3 的内容为 67H (01100111B), 表示十进制数 67 的 BCD 码。进位标志为 1, 则指令:

ADDC A,R3

DA A

先执行标准的补码二进制加法, 累加器 A 的值变为 0BEH, 进位标志和辅助进位标志被清零。

接着, DA 执行十进制调整, 将累加器 A 的内容变为 24H (00100100B), 表示十进制数 24 的 BCD 码, 也就是 56、67 及进位标志之和的后两位数字。DA 指令会把进位标志置位 1, 这表示在进行十进制加法时, 发生了溢出。56、67 以及 1 的和为 124。

把 BCD 格式的变量加上 0IH 或 99H, 可以实现加 1 或者减 1。假设累加器的初始值为 30H (表示十进制数 30), 指令序列:

ADD A, #99H

DA A

将把进位 C 置为 1, 累加器 A 的数据变为 29H, 因为 $30+99=129$ 。加法和的低位数据可以看作减法运算的结果, 即 $30-1=29$ 。

指令长度(字节): 1

执行周期: 3

二进制编码:

1	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

操作: DA

-contents of Accumulator are BCD

IF $[(A_{3:0}) > 9] \vee [(AC) = 1]$

THEN($A_{3:0}$) $\leftarrow (A_{3:0}) + 6$

AND

IF $[(A_{7:4}) > 9] \vee [(C) = 1]$

THEN ($A_{7:4}$) $\leftarrow (A_{7:4}) + 6$

DEC byte

功能: 把 BYTE 所代表的操作数减 1

说明: BYTE 所代表的变量被减去 1。如果原来的值为 00H, 那么减去 1 后, 变成 OFFH。没有标志位会受到影响。该指令支持 4 种操作数寻址方式: 累加器寻址、寄存器寻址、直接寻址和寄存器间接寻址。

注意: 当 DEC 指令用于修改输出端口的状态时, BYTE 所代表的数据是从端口输出数据锁存器中获取的, 而不是从引脚上读取的输入状态。

举例: 假设寄存器 0 的内容为 7FH (01111111B), 内部 RAM 的 7EH 和 7FH 单元的内容分别为 00H 和 40H。则指令:

DEC @R0

DEC R0

DEC @R0

执行后, 寄存器 0 的内容变成 7EH, 内部 RAM 的 7EH 和 7FH 单元的内容分别变为 OFFH 和 3FH。

DEC A

指令长度(字节): 1

执行周期: 1
 二进制编码:

0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---

操作: DEC
 $(A) \leftarrow (A) - 1$

DEC Rn

指令长度(字节): 1
 执行周期: 1
 二进制编码:

0	0	0	1	1	r	r	r
---	---	---	---	---	---	---	---

 操作: DEC
 $(R_n) \leftarrow (R_n) - 1$

DEC direct

指令长度(字节): 2
 执行周期: 1
 二进制编码:

0	0	0	1	0	1	0	1		Direct address
---	---	---	---	---	---	---	---	--	----------------

 操作: DEC
 $(direct) \leftarrow (direct) - 1$

DEC @Ri

指令长度(字节): 1
 执行周期: 1
 二进制编码:

0	0	0	1	0	1	1	i
---	---	---	---	---	---	---	---

 操作: DEC
 $((R_i)) \leftarrow ((R_i)) - 1$

DIV AB

功能: 除法

说明: DIV 指令把累加器 A 中的 8 位无符号整数除以寄存器 B 中的 8 位无符号整数，并将商置于累加器 A 中，余数置于寄存器 B 中。进位标志 C 和溢出标志 OV 被清零。

例外: 如果寄存器 B 的初始值为 00H (即除数为 0)，那么执行 DIV 指令后，累加器 A 和寄存器 B 中的值是不确定的，且溢出标志 OV 将被置位。但在任何情况下，进位标志 C 都会被清零。

举例: 假设累加器的值为 251 (0FBH 或 1111011B)，寄存器 B 的值为 18 (12H 或 00010010B)。

则指令:

DIV AB

执行后，累加器的值变成 13 (0DH 或 00001101B)，寄存器 B 的值变成 17 (11H 或 00010001B)，正好符合 $251 = 13 \times 18 + 17$ 。进位和溢出标志都被清零。

指令长度(字节): 1
 执行周期: 6
 二进制编码:

1	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

 操作: DIV
 $(A)_{15-8} (B)_{7-0} \leftarrow (A)/(B)$

DJNZ <byte>, <rel-addr>

功能: 减 1，若非 0 则跳转

说明: DJNZ 指令首先将第 1 个操作数所代表的变量减 1，如果结果不为 0，则转移到第 2 个操作数所指定的地址处去执行。如果第 1 个操作数的值为 00H，则减 1 后变为 OFFH。该指令不影响标志位。跳转目标地址的计算：首先将 PC 值加 2（即指向下一条指令的首字节），然后将第 2 操作数表示的有符号的相对偏移量加到 PC 上去即可。

byte 所代表的操作数可采用寄存器寻址或直接寻址。

注意: 如果该指令被用来修改输出引脚上的状态，那么 byte 所代表的数据是从端口输出数据锁存器中获取的，而不是直接读取引脚。

举例: 假设内部 RAM 的 40H、50H 和 60H 单元分别存放着 01H、70H 和 15H，则指令：

DJNZ 40H, LABEL_1

DJNZ 50H, LABEL_2

DJNZ 60H, LABEL_3

执行之后，程序将跳转到标号 LABEL2 处执行，且相应的 3 个 RAM 单元的内容变成 00H、6FH 和 15H。之所以第 1 个跳转没被执行，是因为减 1 后其结果为 0，不满足跳转条件。使用 DJNZ 指令可以方便地在程序中实现指定次数的循环，此外用一条指令就可以在程序中实现中等长度的时间延迟（2~512 个机器周期）。指令序列：

MOV R2,#8

TOOOLE: CPL P1.7

DJNZ R2, TOOGLE

将使得 P1.7 的电平翻转 8 次，从而在 P1.7 产生 4 个脉冲，每个脉冲将持续 3 个机器周期，其中 2 个为 DJNZ 指令的执行时间，1 个为 CPL 指令的执行时间。

DJNZ Rn, rel

指令长度(字节): 2

执行周期: 2 或 3

1	1	0	1	1	r	r	r		rel. address
---	---	---	---	---	---	---	---	--	--------------

操作: DJNZ

$(PC) \leftarrow (PC) + 2$

$(Rn) \leftarrow (Rn) - 1$

IF $(Rn) > 0$ or $(Rn) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

DJNZ direct, rel

指令长度(字节): 3

执行周期: 2 或 3

1	1	0	1	0	1	0	1		direct address		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

操作: DJNZ

$(PC) \leftarrow (PC) + 2$

$(direct) \leftarrow (direct) - 1$

IF $(direct) > 0$ or $(direct) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

INC <byte>

功能: 加 1

说明: INC 指令将<byte>所代表的数据加 1。如果原来的值为 FFH，则加 1 后变为 00H，该指令

不影响标志位。支持 3 种寻址模式：寄存器寻址、直接寻址、寄存器间接寻址。

注意：如果该指令被用来修改输出引脚上的状态，那么 byte 所代表的数据是从端口输出数据锁存器中获取的，而不是直接读的引脚。

举例：假设寄存器 0 的内容为 7EH(0111110B)，内部 RAM 的 7E 单元和 7F 单元分别存放着 OFFH 和 40H，则指令序列：

INC @R0

INC R0

INC @R0

执行完毕后，寄存器 0 的内容变为 7FH，而内部 RAM 的 7EH 和 7FH 单元的内容分别变成 00H 和 41H。

INC A

指令长度(字节)： 1

执行周期： 1

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

操作： INC

$(A) \leftarrow (A)+1$

INC Rn

指令长度(字节)： 1

执行周期： 1

0	0	0	0	1	r	r	r
---	---	---	---	---	---	---	---

操作： INC

$(R_n) \leftarrow (R_n)+1$

INC direct

指令长度(字节)： 2

执行周期： 1

0	0	0	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作： INC

$(\text{direct}) \leftarrow (\text{direct})+1$

INC @Ri

指令长度(字节)： 1

执行周期： 1

0	0	0	0	0	1	1	i
---	---	---	---	---	---	---	---

操作： INC

$((R_i)) \leftarrow ((R_i)) + 1$

INC DPTR

功能： 数据指针加 1

说明： 该指令实现将 DPTR 加 1 功能。需要注意的是，这是 16 位的递增指令，低位字节 DPL 从 FFH 增加 1 之后变为 00H，同时进位到高位字节 DPH。该操作不影响标志位。

该指令是唯一 1 条 16 位寄存器递增指令。

举例： 假设寄存器 DPH 和 DPL 的内容分别为 12H 和 0FEH，则指令序列：

IINC DPTR

INC DPTR

INC DPTR

执行完毕后, DPH 和 DPL 变成 13H 和 01H。

指令长度(字节): 1

执行周期: 1

二进制编码:

1	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

操作: INC

$(\text{DPTR}) \leftarrow (\text{DPTR}) + 1$

JB bit, rel

功能: 若位数据为 1 则跳转

说明: 如果 bit 代表的位数据为 1, 则跳转到 rel 所指定的地址处去执行; 否则, 继续执行下一条指令。跳转的目标地址按照如下方式计算: 先增加 PC 的值, 使其指向下一条指令的首字节地址, 然后把 rel 所代表的有符号的相对偏移量 (指令的第 3 个字节) 加到 PC 上去, 新的 PC 值即为目标地址。该指令只是测试相应的位数据, 但不会改变其数值, 而且该操作不会影响标志位。

举例: 假设端口 1 的输入数据为 11001010B, 累加器的值为 56H (01010110B)。则指令:

JB P1.2, LABEL1

JB ACC.2, LABEL2

将导致程序转到标号 LABEL2 处去执行。

指令长度(字节): 3

执行周期: 1 或 3

二进制编码:

0	0	1	0	0	0	0	0	bit address	rel. address
---	---	---	---	---	---	---	---	-------------	--------------

操作: JB

$(\text{PC}) \leftarrow (\text{PC}) + 3$

IF (bit) = 1

THEN

$(\text{PC}) \leftarrow (\text{PC}) + \text{rel}$

JBC bit, rel

功能: 若位数据为 1 则跳转并将其清零

说明: 如果 bit 代表的位数据为 1, 则将其清零并跳转到 rel 所指定的地址处去执行。如果 bit 代表的位数据为 0, 则继续执行下一条指令。跳转的目标地址按照如下方式计算: 先增加 PC 的值, 使其指向下一条指令的首字节地址, 然后把 rel 所代表的有符号的相对偏移量 (指令的第 3 个字节) 加到 PC 上去, 新的 PC 值即为目标地址, 而且该操作不会影响标志位。注意: 如果该指令被用来修改输出引脚上的状态, 那么 byte 所代表的数据是从端口输出数据锁存器中获取的, 而不是直接读取引脚。

举例: 假设累加器的内容为 56H(01010110B), 则指令序列:

JBC ACC.3, LABEL1

JBC ACC.2, LABEL2

将导致程序转到标号 LABEL2 处去执行, 且累加器的内容变为 52H (01010010B)。

指令长度(字节): 3

执行周期: 1 或 3

二进制编码:

0	0	0	1	0	0	0	0	bit address	rel. address
---	---	---	---	---	---	---	---	-------------	--------------

操作: JB

```

 $(PC) \leftarrow (PC) + 3$ 
IF (bit) = 1
THEN
 $(bit) \leftarrow 0$ 
 $(PC) \leftarrow (PC) + rel$ 

```

JC rel

功能: 若进位标志为 1, 则跳转

说明: 如果进位标志为 1, 则程序跳转到 rel 所代表的地址处去执行; 否则, 继续执行下面的指令。跳转的目标地址按照如下方式计算: 先增加 PC 的值, 使其指向紧接 JC 指令的下一条指令的首地址, 然后把 rel 所代表的有符号的相对偏移量 (指令的第 2 个字节) 加到 PC 上去, 新的 PC 值即为目标地址。该操作不会影响标志位。

举例: 假设进位标志此时为 0, 则指令序列:

JC LABEL1

CPL C

JC LABEL2

执行完毕后, 进位标志变成 1, 并导致程序跳转到标号 LABEL2 处去执行。

指令长度(字节): 2

执行周期: 1 或 3

0	1	0	0	0	0	0	0	rel. address
---	---	---	---	---	---	---	---	--------------

操作: JC

$(PC) \leftarrow (PC) + 2$

IF (C) = 1

THEN

$(PC) \leftarrow (PC) + rel$

JMP @A+DPTR

功能: 间接跳转

说明: 把累加器 A 中的 8 位无符号数据和 16 位的数据指针的值相加, 其和作为下一条将要执行的指令的地址, 传送给程序计数器 PC。执行 16 位的加法时, 低字节 DPL 的进位会传到高字节 DPH。累加器 A 和数据指针 DPTR 的内容都不会发生变化。不影响任何标志位。

举例: 假设累加器 A 中的值是偶数 (从 0 到 6)。下面的指令序列将使得程序跳转到位于跳转表 JMP_TBL 的 4 条 AJMP 指令中的某一条去执行:

MOV DPTR, #JMP_TBL

JMP @A+DPTR

JMP-TBL: AJMP LABEL0

AJMP LABEL1

AJMP LABEL2

AJMP LABEL3

如果开始执行上述指令序列时, 累加器 A 中的值为 04H, 那么程序最终会跳转到标号 LABEL2 处去执行。

注意: AJMP 是一个 2 字节指令, 因而在跳转表中, 各个跳转指令的入口地址依次相差 2 个字节。

指令长度(字节): 1
 执行周期: 4
 二进制编码:

0	1	1	1	0	0	1	1
---	---	---	---	---	---	---	---

 操作: JMP
 $(PC) \leftarrow (A) + (DPTR)$

JNB bit, rel

功能: 如果 bit 所代表的位不为 1 则跳转
 说明: 如果 bit 所表示的位为 0, 则转移到 rel 所代表的地址去执行; 否则, 继续执行下一条指令。
 跳转的目标地址如此计算: 先增加 PC 的值, 使其指向第一条指令的首字节地址, 然后把 rel 所代表的有符号的相对偏移量(指令的第 3 个字节)加到 PC 上去, 新的 PC 值即为目标地址。该指令只是测试相应的位数据, 但不会改变其数值, 而且该操作不会影响标志位。
 举例: 假设端口 1 的输入数据为 11001010B, 累加器的值为 56H (01010110B)。则指令序列:
 JNB P1.3, LABEL1
 JNB ACC.3, LABEL2
 执行后将导致程序转到标号 LABEL2 处去执行。

指令长度(字节): 3
 执行周期: 1 或 3
 二进制编码:

0	0	1	1	0	0	0	0	bit address	rel. address
---	---	---	---	---	---	---	---	-------------	--------------

 操作: JNB
 $(PC) \leftarrow (PC) + 3$
 $IF (bit) = 0$
 $THEN (PC) \leftarrow (PC) + rel$

JNC rel

功能: 若进位标志非 1 则跳转
 说明: 如果进位标志为 0, 则程序跳转到 rel 所代表的地址处去执行; 否则, 继续执行下面的指令。
 跳转的目标地址按照如下方式计算: 先增加 PC 的值加 2, 使其指向紧接 JNC 指令的下一条指令的地址, 然后把 rel 所代表的有符号的相对偏移量(指令的第 2 个字节)加到 PC 上去, 新的 PC 值即为目标地址。该操作不会影响标志位。
 举例: 假设进位标志此时为 1, 则指令序列:
 JNC LABEL1
 CPL C
 JNC LABEL2
 执行完毕后, 进位标志变成 0, 并导致程序跳转到标号 LABEL2 处去执行。

指令长度(字节): 2
 执行周期: 1 或 3
 二进制编码:

0	1	0	1	0	0	0	0	rel. address
---	---	---	---	---	---	---	---	--------------

 操作: JNC
 $(PC) \leftarrow (PC) + 2$
 $IF (C) = 0$
 $THEN (PC) \leftarrow (PC) + rel$

JNZ rel

功能: 如果累加器的内容非 0 则跳转

说明: 如果累加器 A 的任何一位为 1, 那么程序跳转到 rel 所代表的地址处去执行, 如果各个位都为 0, 继续执行下一条指令。跳转的目标地址按照如下方式计算: 先把 PC 的值增加 2, 然后把 rel 所代表的有符号的相对偏移量(指令的第 2 个字节)加到 PC 上去, 新的 PC 值即为目标地址。操作过程中累加器的值不会发生变化, 不会影响标志位。

举例: 设累加器的初始值为 00H, 则指令序列:

JNZ LABEL1

INC A

JNZ LAEEL2

执行完毕后, 累加器的内容变成 01H, 且程序将跳转到标号 LABEL2 处去执行。

指令长度(字节): 2

执行周期: 1 或 3

0	1	1	1	0	0	0	0	rel. address
---	---	---	---	---	---	---	---	--------------

操作: JNZ

$(PC) \leftarrow (PC) + 2$

IF ($A \neq 0$)

THEN $(PC) \leftarrow (PC) + rel$

JZ rel

功能: 若累加器的内容为 0 则跳转

说明: 如果累加器 A 的任何一位为 0, 那么程序跳转到 rel 所代表的地址处去执行, 如果各个位都为 0, 继续执行下一条指令。跳转的目标地址按照如下方式计算: 先把 PC 的值增加 2, 然后把 rel 所代表的有符号的相对偏移量(指令的第 2 个字节)加到 PC 上去, 新的 PC 值即为目标地址。操作过程中累加器的值不会发生变化, 不会影响标志位。

举例: 设累加器的初始值为 01H, 则指令序列:

JZ LABEL1

DEC A

JZ LAEEL2

执行完毕后, 累加器的内容变成 00H, 且程序将跳转到标号 LABEL2 处去执行。

指令长度(字节): 2

执行周期: 1 或 3

0	1	1	0	0	0	0	0	rel. address
---	---	---	---	---	---	---	---	--------------

操作: JZ

$(PC) \leftarrow (PC) + 2$

IF ($A = 0$)

THEN $(PC) \leftarrow (PC) + rel$

LCALL addr16

功能: 长调用

说明: LCALL 用于调用 addr16 所指地址处的子例程。首先将 PC 的值增加 3, 使得 PC 指向紧随

LCALL 的下一条指令的地址，然后把 16 位 PC 的低 8 位和高 8 位依次压入栈（低位字节在先），同时把栈指针加 2。然后再把 LCALL 指令的第 2 字节和第 3 字节的数据分别装入 PC 的高位字节 DPH 和低位字节 DPL，程序从新的 PC 所对应的地址处开始执行。因而子例程可以位于 64KB 程序存储空间的任何地址处。该操作不影响标志位。

举例: 栈指针的初始值为 07H，标号 SUBRTN 被分配的程序存储器地址为 1234H。则执行如下位于地址 0123H 的指令后：

LCALL SUBRTN

栈指针变成 09H，内部 RAM 的 08H 和 09H 单元的内容分别为 26H 和 01H，且 PC 的当前值为 1234H。

指令长度(字节) : 3

执行周期: 3

0	0	0	1	0	0	1	0		addr15-addr8		addr7-addr0
---	---	---	---	---	---	---	---	--	--------------	--	-------------

操作: LCALL

$(PC) \leftarrow (PC) + 3$

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (PC_{7:0})$

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (PC_{15:8})$

$(PC) \leftarrow addr_{15:0}$

LJMP addr16

功能: 长跳转

说明: LJMP 使得 CPU 无条件跳转到 addr16 所指的地址处执行程序。把该指令的第 2 字节和第 3 字节分别装入程序计数器 PC 的高位字节 DPH 和低位字节 DPL。程序从新 PC 值对应的地址处开始执行。该 16 位目标地址可位于 64KB 程序存储空间的任何地址处。该操作不影响标志位。

举例: 假设标号 JMPADR 被分配的程序存储器地址为 1234H。则位于地址 1234H 的指令：

LJMP JMPADR

执行完毕后，PC 的当前值变为 1234H。

指令长度(字节) : 3

执行周期: 3

0	0	0	0	0	0	1	0		addr15-addr8		addr7-addr0
---	---	---	---	---	---	---	---	--	--------------	--	-------------

操作: LJMP

$(PC) \leftarrow addr_{15:0}$

MOV <dest-byte>, <src-byte>

功能: 传送字节变量

说明: 将第 2 操作数代表字节变量的内容复制到第 1 操作数所代表的存储单元中去。该指令不会改变源操作数，也不会影响其他寄存器和标志位。

MOV 指令是迄今为止使用最灵活的指令，源操作数和目的操作数组合起来，寻址方式可达 15 种。

举例: 假设内部 RAM 的 30H 单元的内容为 40H，而 40H 单元的内容为 10H。端口 1 的数据为 11001010B (0CAH)。则指令序列：

```

MOV R0, #30H      ; R0<=30H
MOV A, @R0        ; A <= 40H
MOV R1, A         ; R1 <= 40H
MOV B, @R1        ; B <= 10H
MOV @R1, P1       ; RAM (40H) <= 0CAH
MOV P2, P1        ; P2 #0CAH

```

执行完毕后, 寄存器 0 的内容为 30H, 累加器和寄存器 1 的内容都为 40H, 寄存器 B 的内容为 10H, RAM 中 40H 单元和 P2 口的内容均为 0CAH。

MOV A,Rn

指令长度(字节): 1

执行周期: 1

二进制编码:

1	1	1	0	1	r	r	r
---	---	---	---	---	---	---	---

操作: MOV

(A) ←(Rn)

***MOV A,direct**

指令长度(字节): 2

执行周期: 1

二进制编码:

1	1	1	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作: MOV

(A) ←(direct)

注意: MOV A,ACC 是无效指令。

MOV A,@Ri

指令长度(字节): 1

执行周期: 1

二进制编码:

1	1	1	0	0	1	1	i
---	---	---	---	---	---	---	---

操作: MOV

(A) ←((Ri))

MOV A,#data

指令长度(字节): 2

执行周期: 1

二进制编码:

0	1	1	1	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

操作: MOV

(A) ←#data

MOV Rn,A

指令长度(字节): 1

执行周期: 1

二进制编码:

1	1	1	1	1	r	r	r
---	---	---	---	---	---	---	---

操作: MOV

(Rn) ←(A)

MOV Rn,direct

指令长度(字节): 2

执行周期: 1

二进制编码:

1	0	1	0	1	r	r	r	direct address
---	---	---	---	---	---	---	---	----------------

操作: MOV

(Rn) ←(direct)

MOV Rn,#data

指令长度(字节): 2

执行周期: 1

二进制编码:	0	1	1	1	1	r	r	r		immediate data
--------	---	---	---	---	---	---	---	---	--	----------------

操作: MOV

(Rn)←#data

MOV direct,A

指令长度(字节): 2

执行周期: 1

二进制编码:	1	1	1	1	0	1	0	1		direct address
--------	---	---	---	---	---	---	---	---	--	----------------

操作: MOV

(direct)←(A)

MOV direct,Rn

指令长度(字节): 2

执行周期: 2

二进制编码:	1	0	0	0	1	r	r	r		direct address
--------	---	---	---	---	---	---	---	---	--	----------------

操作: MOV

(direct)←(Rn)

MOV direct,direct

指令长度(字节): 3

执行周期: 1

二进制编码:	1	0	0	0	0	1	0	1		dir.addr.(src)		dir.addr.(dest)
--------	---	---	---	---	---	---	---	---	--	----------------	--	-----------------

操作: MOV

(direct)←(direct)

MOV direct,@Ri

指令长度(字节): 2

执行周期: 1

二进制编码:	1	0	0	0	0	1	1	i		direct address
--------	---	---	---	---	---	---	---	---	--	----------------

操作: MOV

(direct)←((Ri))

MOV direct,#data

指令长度(字节): 3

执行周期: 1

二进制编码:	0	1	1	1	0	1	0	1		direct address		immediate data
--------	---	---	---	---	---	---	---	---	--	----------------	--	----------------

操作: MOV

(direct)←#data

MOV @Ri,A

指令长度(字节): 1

执行周期: 1

二进制编码:	1	1	1	1	0	1	1	i
--------	---	---	---	---	---	---	---	---

操作: MOV

((Ri))←(A)

MOV @Ri,direct

指令长度(字节): 2

执行周期: 1

二进制编码:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>i</td><td></td><td>direct address</td></tr></table>	1	0	1	0	0	1	1	i		direct address
1	0	1	0	0	1	1	i		direct address		

操作: MOV

 $((Ri)) \leftarrow (\text{direct})$ **MOV @Ri, #data**

指令长度(字节): 2

执行周期: 1

二进制编码:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>i</td><td></td><td>immediate data</td></tr></table>	0	1	1	1	0	1	1	i		immediate data
0	1	1	1	0	1	1	i		immediate data		

操作: MOV

 $((Ri)) \leftarrow \#data$ **MOV <dest-bit>, <src-bit>**

功能: 传送位变量

说明: 将<src-bit>代表的布尔变量复制到<dest-bit>所指定的数据单元中去, 两个操作数必须有一个是进位标志, 而另外一个是可直接寻址的位。本指令不影响其他寄存器和标志位。

举例: 假设进位标志 C 的初值为 1, 端口 P3 中的数据是 11000101B, 端口 1 的数据被设置为 35H(00110101B)。则指令序列:

MOV P1.3, C

MOV C, P3.3

MOV P1.2, C

执行后, 进位标志被清零, 端口 1 的数据变为 39H (00111001B)。

MOV C, bit

指令长度(字节): 2

执行周期: 1

二进制编码:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td></td><td>bit address</td></tr></table>	1	0	1	0	0	0	1	0		bit address
1	0	1	0	0	0	1	0		bit address		

操作: MOV

 $(C) \leftarrow (\text{bit})$ **MOV bit, C**

指令长度(字节): 2

执行周期: 1

二进制编码:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td></td><td>bit address</td></tr></table>	1	0	0	1	0	0	1	0		bit address
1	0	0	1	0	0	1	0		bit address		

操作: MOV

 $(\text{bit}) \leftarrow (C)$ **MOV DPTR, #data 16**

功能: 将 16 位的常数存放到数据指针

说明: 该指令将 16 位常数传递给数据指针 DPTR。16 位的常数包含在指令的第 2 字节和第 3 字节中。其中 DPH 中存放的是#data16 的高字节, 而 DPL 中存放的是#data16 的低字节。不影响标志位。

该指令是唯一一条能一次性移动 16 位数据的指令。

举例: 指令:

MOV DPTR, #1234H

将立即数 1234H 装入数据指针寄存器中。DPH 的值为 12H, DPL 的值为 34H。

指令长度(字节): 3

执行周期:	1
二进制编码:	[1 0 0 1 0 0 0 0 immediate data15-8 immediate data7-0]
操作:	MOV
	(DPTR) \leftarrow #data ₁₅₋₀
	DPH DPL \leftarrow #data ₁₅₋₈ #data ₇₋₀

MOVCA , @A+ <base-reg>

- 功能:** 把程序存储器中的代码字节数据（常数数据）转送至累加器 A
- 说明:** MOVC 指令将程序存储器中的代码字节或常数字节传送到累加器 A。被传送的数据字节的地址是由累加器中的无符号 8 位数据和 16 位基址寄存器（DPTR 或 PC）的数值相加产生的。如果以 PC 为基址寄存器，则在累加器内容加到 PC 之前，PC 需要先增加到指向紧邻 MOVC 之后的语句的地址；如果是以 DPTR 为基址寄存器，则没有此问题。在执行 16 位的加法时，低 8 位产生的进位会传递给高 8 位。本指令不影响标志位。
- 举例:** 假设累加器 A 的值处于 0~4 之间，如下子例程将累加器 A 中的值转换为用 DB 伪指令（定义字节）定义的 4 个值之一：

REL-PC: INC A

```

MOVC A, @A+PC
RET
DB 66H
DB 77H
DB 88H
DB 99H

```

如果在调用该子例程之前累加器的值为 01H，执行完该子例程后，累加器的值变为 77H。MOVC 指令之前的 INC A 指令是为了在查表时越过 RET 而设置的。如果 MOVC 和表格之间被多个代码字节所隔开，那么为了正确地读取表格，必须将相应的字节数预先加到累加器 A 上。

MOVC A,@A+DPTR

指令长度(字节):	1
执行周期:	4
二进制编码:	[1 0 0 1 0 0 1 1]
操作:	MOVC
	(A) \leftarrow ((A)+(DPTR))

MOVC A,@A+PC

指令长度(字节):	1
执行周期:	3
二进制编码:	[1 0 0 0 0 0 1 1]
操作:	MOVC
	(PC) \leftarrow (PC)+1
	(A) \leftarrow ((A)+(PC))

MOVX <dest-byte> , <src-byte>

- 功能:** 外部传送
- 说明:** MOVX 指令用于在累加器和外部数据存储器之间传递数据。因此在传送指令 MOV 后附加

了 X。MOVX 又分为两种类型，它们之间的区别在于访问外部数据 RAM 的间接地址是 8 位的还是 16 位的。

对于第 1 种类型，当前工作寄存器组的 R0 和 R1 提供 8 位地址到复用端口 P0。对于外部 I/O 扩展译码或者较小的 RAM 阵列，8 位的地址已经够用。若要访问较大的 RAM 阵列，可在端口引脚上输出高位的地址信号。此时可在 MOVX 指令之前添加输出指令，对这些端口引脚施加控制。

对于第 2 种类型，通过数据指针 DPTR 产生 16 位的地址。当 P2 端口的输出缓冲器发送 DPH 的内容时，P2 的特殊功能寄存器保持原来的数据。在访问规模较大的数据阵列时，这种方式更为有效和快捷，因为不需要额外指令来配置输出端口。

在某些情况下，可以混合使用两种类型的 MOVX 指令。在访问大容量的 RAM 空间时，既可以用数据指针 DP 在 P2 端口上输出地址的高位字节，也可以先用某条指令，把地址的高位字节从 P2 端口上输出，再使用通过 R0 或 R1 间接寻址的 MOVX 指令。

举例：假设有一个分时复用地址/数据线的外部 RAM 存储器，容量为 256B(如:Intel 的 8155 RAM / I/O / TIMER)，该存储器被连接到 8051 的端口 P0 上，端口 P3 被用于提供外部 RAM 所需的控制信号。端口 P1 和 P2 用作通用输入/输出端口。R0 和 R1 中的数据分别为 12H 和 34H，外部 RAM 的 34H 单元存储的数据为 56H，则下面的指令序列：

MOVX A, @R1

MOVX @R0, A

将数据 56H 复制到累加器 A 以及外部 RAM 的 12H 单元中。

MOVX A,@Ri

指令长度(字节)： 1

执行周期： 3 或 1

二进制编码:	1	1	1	0	0	0	1	i
--------	---	---	---	---	---	---	---	---

操作： MOVX

(A) ← ((Ri))

MOVX A,@DPTR

指令长度(字节)： 1

执行周期： 1 或 2

二进制编码:	1	1	1	0	0	0	0	0
--------	---	---	---	---	---	---	---	---

操作： MOVX

(A) ← ((DPTR))

MOVX @Ri,A

指令长度(字节)： 1

执行周期： 3 或 1

二进制编码:	1	1	1	1	0	0	1	i
--------	---	---	---	---	---	---	---	---

操作： MOVX

((Ri))←(A)

MOVX @DPTR,A

指令长度(字节)： 1

执行周期： 2 或 1

二进制编码:	1	1	1	1	0	0	0	0
--------	---	---	---	---	---	---	---	---

操作： MOVX

(DPTR)←(A)

MUL AB

功能: 乘法

说明: 该指令可用于实现累加器和寄存器 B 中的无符号 8 位整数的乘法。所产生的 16 位乘积的低 8 位存放在累加器中，而高 8 位存放在寄存器 B 中。若乘积大于 255(0FFH)，则置位溢出标志；否则清零标志位。在执行该指令时，进位标志总是被清零。

举例: 假设累加器 A 的初始值为 80(50H)，寄存器 B 的初始值为 160 (0A0H)，则指令：

MUL AB

求得乘积 12800 (3200H)，所以寄存器 B 的值变成 32H (00110010B)，累加器被清零，溢出标志被置位，进位标志被清零。

指令长度(字节): 1

执行周期: 2

二进制编码:

1	0	1	0	0	1	0	0
---	---	---	---	---	---	---	---

操作: $(A)_{7:0} \leftarrow (A) \times (B)$

$(B)_{15:8}$

NOP

功能: 空操作

说明: 执行本指令后，将继续执行随后的指令。除了 PC 外，其他寄存器和标志位都不会有变化。

举例: 假设期望在端口 P2 的第 7 号引脚上输出一个长时间的低电平脉冲，该脉冲持续 5 个机器周期（精确）。若是仅使用 SETB 和 CLR 指令序列，生成的脉冲只能持续 1 个机器周期。因而需要设法增加 4 个额外的机器周期。可以按照如下方式来实现所要求的功能（假设中断没有被启用）：

CLR P2.7

NOP

NOP

NOP

NOP

SETB P2.7

指令长度(字节): 1

执行周期: 1

二进制编码:

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

操作: NOP

$(PC) \leftarrow (PC) + 1$

ORL <dest-byte>, <src-byte>

功能: 两个字节变量的逻辑或运算

说明: ORL 指令将由<dest-byte>和<src_byte>所指定的两个字节变量进行逐位逻辑或运算，结果存放在<dest-byte>所代表的数据单元中。该操作不影响标志位。

两个操作数组合起来，支持 6 种寻址方式。当目的操作数是累加器 A 时，源操作数可以采用寄存器寻址、直接寻址、寄存器间接寻址或者立即寻址。当目的操作数采用直接寻址方式时，源操作数可以是累加器或立即数。

注意: 如果该指令被用来修改输出引脚上的状态，那么<dest-byte>所代表的数据是从端口输出数据锁存器中获取的数据，而不是从引脚上读取的数据。

举例: 假设累加器 A 中数据为 0C3H(11000011B), 寄存器 R0 中的数据为 55H(01010101), 则指令:

ORL A, R0

执行后, 累加器的内容变成 0D7H(11010111B)。当目的操作数是直接寻址数据字节时, ORL 指令可用来把任何 RAM 单元或者硬件寄存器中的各个位设置为 1。究竟哪些位会被置 1 由屏蔽字节决定, 屏蔽字节既可以是包含在指令中的常数, 也可以是累加器 A 在运行过程中实时计算出的数值。执行指令:

ORL P1, #00110010B

之后, 把 1 口的第 5、4、1 位置 1。

ORL A, Rn

指令长度(字节): 1

执行周期: 1

0	1	0	0	1	r	r	r
---	---	---	---	---	---	---	---

操作: ORL

(A) \leftarrow (A) \vee (Rn)

ORL A, direct

指令长度(字节): 2

执行周期: 1

0	1	0	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作: ORL

(A) \leftarrow (A) \vee (direct)

ORL A, @Ri

指令长度(字节): 1

执行周期: 1

0	1	0	0	0	1	1	i
---	---	---	---	---	---	---	---

操作: ORL

(A) \leftarrow (A) \vee ((Ri))

ORL A, #data

指令长度(字节): 2

执行周期: 1

0	1	0	0	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

操作: ORL

(A) \leftarrow (A) \vee #data

ORL direct, A

指令长度(字节): 2

执行周期: 1

0	1	0	0	0	0	1	0	direct address
---	---	---	---	---	---	---	---	----------------

操作: ORL

(direct) \leftarrow (direct) \vee (A)

ORL direct, #data

指令长度(字节): 3

执行周期: 1

0	1	0	0	0	0	1	1	direct address	immediate data
---	---	---	---	---	---	---	---	----------------	----------------

操作: ORL

(direct) \leftarrow (direct) \vee #data

ORL C, <src-bit>

功能: 位变量的逻辑或运算

说明: 如果<src-bit>所表示的位变量为 1，则置位进位标志；否则，保持进位标志的当前状态不变。

在汇编语言中，位于源操作数之前的“/”表示将源操作数取反后使用，但源操作数本身不发生变化。在执行本指令时，不影响其他标志位。

举例: 当执行如下指令序列时，当且仅当 P1.0=1 或 ACC.7=1 或 OV=0 时，置位进位标志 C：

```
MOV C, P1.0      ;LOAD CARRY WITH INPUT PIN P10
ORL C, ACC.7     ;OR CARRY WITH THE ACC.BIT 7
ORL C, /OV       ;OR CARRY WITH THE INVERSE OF OV
```

ORL C, bit

指令长度(字节): 2

执行周期: 1 或 4

0	1	1	1	0	0	1	0		bit address
---	---	---	---	---	---	---	---	--	-------------

操作: ORL

$(C) \leftarrow (C) \vee (\text{bit})$

ORL C, /bit

指令长度(字节): 2

执行周期: 1

1	0	1	0	0	0	0	0		bit address
---	---	---	---	---	---	---	---	--	-------------

操作: ORL

$(C) \leftarrow (C) \vee (\overline{\text{bit}})$

POP direct

功能: 出栈

说明: 读取栈指针所指定的内部 RAM 单元的内容，栈指针减 1。然后，将读到的内容传送到由 direct 所指示的存储单元（直接寻址方式）中去。该操作不影响标志位。

举例: 设栈指针的初值为 32H，内部 RAM 的 30H~32H 单元的数据分别为 20H、23H 和 01H。则执行指令：

POP DPH

POP DPL

之后，栈指针的值变成 30H，数据指针变为 0123H。此时指令：

POP SP

将把栈指针变为 20H。

注意: 在这种特殊情况下，在写入出栈数据（20H）之前，栈指针先减小到 2FH，然后再随着 20H 的写入，变成 20H。

指令长度(字节): 2

执行周期: 1

1	1	0	1	0	0	0	0		direct address
---	---	---	---	---	---	---	---	--	----------------

操作: POP

$(\text{direct}) \leftarrow ((\text{SP}))$

$(\text{SP}) \leftarrow (\text{SP}) - 1$

PUSH direct

功能: 压栈

说明: 栈指针首先加 1, 然后将 direct 所表示的变量内容复制到由栈指针指定的内部 RAM 存储单元中去。该操作不影响标志位。

举例: 设在进入中断服务程序时栈指针的值为 09H, 数据指针 DPTR 的值为 0123H。则执行如下指令序列:

PUSH DPL

PUSH DPH

之后, 栈指针变为 0BH, 并把数据 23H 和 01H 分别存入内部 RAM 的 0AH 和 0BH 存储单元之中。

指令长度(字节): 2

执行周期: 1

二进制编码:

1	1	0	0	0	0	0	0	direct address
---	---	---	---	---	---	---	---	----------------

操作: PUSH

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (\text{direct})$

RET

功能: 从子例程返回

说明: 执行 RET 指令时, 首先将 PC 值的高位字节和低位字节从栈中弹出, 栈指针减 2。然后, 程序从形成的 PC 值所对应的地址处开始执行, 一般情况下, 该指令和 ACALL 或 LCALL 配合使用。改指令的执行不影响标志位。

举例: 设栈指针的初值为 0BH, 内部 RAM 的 0AH 和 0BH 存储单元中的数据分别为 23H 和 01H。则指令:

RET

执行后, 栈指针变为 09H。程序将从 0123H 地址处继续执行。

指令长度(字节): 1

执行周期: 3

二进制编码:

0	0	1	0	0	0	1	0
---	---	---	---	---	---	---	---

操作: RET

$(PC_{15-8}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

$(PC_{7-0}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

RETI

功能: 中断返回

说明: 执行该指令时, 首先从栈中弹出 PC 值的高位和低位字节, 然后恢复中断启用, 准备接受同优先级的其他中断, 栈指针减 2。其他寄存器不受影响。但程序状态字 PSW 不会自动恢复到中断前的状态。程序将继续从新产生的 PC 值所对应的地址处开始执行, 一般情况下是此次中断入口的下一条指令。在执行 RETI 指令时, 如果有一个优先级较低的或同优先级的其他中断在等待处理, 那么在处理这些等待中的中断之前需要执行 1 条指令。

举例: 设栈指针的初值为 0BH, 结束在地址 0123H 处的指令执行结束期间产生中断, 内部 RAM 的 0AH 和 0BH 单元的内容分别为 23H 和 01H。则指令:

RETI

执行完毕后, 栈指针变成 09H, 中断返回后程序继续从 0123H 地址开始执行。

指令长度(字节): 1

执行周期: 3

0	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

操作: RETI

$(PC_{15:8}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

$(PC_{7:0}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

RL A

功能: 将累加器 A 中的数据位循环左移

说明: 将累加器中的 8 位数据均左移 1 位, 其中位 7 移动到位 0。该指令的执行不影响标志位。

举例: 假设累加器 A 的内容为 0C5H (11000101B), 则指令:

RL A

执行后, 累加器的内容变成 8BH (10001011B), 且标志位不受影响。

指令长度(字节): 1

执行周期: 1

0	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

操作: RL

$(An+1) \leftarrow (An) \quad n = 0-6$

$(A0) \leftarrow (A7)$

RLC A

功能: 带进位循环左移

说明: 累加器的 8 位数据和进位标志一起循环左移 1 位。其中位 7 移入进位标志, 进位标志的初始状态值移到位 0。该指令不影响其他标志位。

举例: 假设累加器 A 的值为 0C5H(11000101B), 则指令:

RLC A

执行后, 将把累加器 A 的数据变为 8BH(10001011B), 进位标志被置位。

指令长度(字节): 1

执行周期: 1

0	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

操作: RLC

$(An+1) \leftarrow (An) \quad n = 0-6$

$(A0) \leftarrow (C)$

$(C) \leftarrow (A7)$

RR A

功能: 将累加器的数据位循环右移

说明: 将累加器的 8 个数据位均右移 1 位, 位 0 将被移到位 7, 即循环右移, 该指令不影响标志位。

举例: 设累加器的内容为 0C5H (11000101B), 则指令:

RR A

执行后累加器的内容变成 0E2H (11100010B), 标志位不受影响。

指令长度(字节): 1

执行周期: 1

二进制编码:

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

操作: RR

$(An) \leftarrow (An+1)$ $n = 0 - 6$

$(A7) \leftarrow (A0)$

RRC A

功能: 带进位循环右移

说明: 累加器的 8 位数据和进位标志一起循环右移 1 位。其中位 0 移入进位标志, 进位标志的初始状态值移到位 7。该指令不影响其他标志位。

举例: 假设累加器的值为 0C5H(11000101B), 进位标志为 0, 则指令:

RRC A

执行后, 将把累加器的数据变为 62H(01100010B), 进位标志被置位。

指令长度(字节): 1

执行周期: 1

二进制编码:

0	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---

操作: RRC

$(An+1) \leftarrow (An)$ $n = 0 - 6$

$(A7) \leftarrow (C)$

$(C) \leftarrow (A0)$

SETB <bit>

功能: 置位

说明: SETB 指令可将相应的位置 1, 其操作对象可以是进位标志或其他可直接寻址的位。该指令不影响其他标志位。

举例: 设进位标志被清零, 端口 1 的输出状态为 34H(00110100B), 则指令:

SETB C

SETB P1.0

执行后, 进位标志变为 1, 端口 1 的输出状态变成 35H(00110101B)。

SETB C

指令长度(字节): 1

执行周期: 1

二进制编码:

1	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

操作: SETB

$(C) \leftarrow 1$

SETB bit

指令长度(字节): 2

执行周期: 1

二进制编码:	1 1 0 1 0 0 1 0	bit address
--------	-----------------	-------------

操作: SETB

 $(\text{bit}) \leftarrow 1$

SJMP rel

功能: 短跳转

说明: 程序无条件跳转到 rel 所示的地址去执行。目标地址按如下方法计算: 首先 PC 值加 2, 然后将指令第 2 字节(即 rel)所表示的有符号偏移量加到 PC 上, 得到的新 PC 值即短跳转的目标地址。所以, 跳转的范围是当前指令(即 SJMP)地址的前 128 字节和后 127 字节。

举例: 设标号 RELADR 对应的指令地址位于程序存储器的 0123H 地址, 则指令:

SJMP RELADR

汇编后位于 0100H。当执行完该指令后, PC 值变成 0123H。

注意: 在上例中, 紧接 SJMP 的下一条指令的地址是 0102H, 因此, 跳转的偏移量为 0123H-0102H=21H。另外, 如果 SJMP 的偏移量是 0FEH, 那么构成只有 1 条指令的无限循环。

指令长度(字节): 2

执行周期: 3

二进制编码:	1 0 0 0 0 0 0 0	rel. address
--------	-----------------	--------------

操作: SJMP

 $(\text{PC}) \leftarrow (\text{PC}) + 2$ $(\text{PC}) \leftarrow (\text{PC}) + \text{rel}$

SUBB A, <src-byte>

功能: 带借位的减法

说明: SUBB 指令从累加器中减去<src-byte>所代表的字节变量的数值及进位标志, 减法运算的结果置于累加器中。如果执行减法时第 7 位需要借位, SUBB 将会置位进位标志(表示借位); 否则, 清零进位标志。(如果在执行 SUBB 指令前, 进位标志 C 已经被置位, 这意味着在前面进行多精度的减法运算时, 产生了借位。因而在执行本条指令时, 必须把进位连同源操作数一起从累加器中减去。) 如果在进行减法运算的时候, 第 3 位处向上有借位, 那么辅助进位标志 AC 会被置位; 如果第 6 位有借位; 而第 7 位没有, 或是第 7 位有借位, 而第 6 位没有, 则溢出标志 OV 被置位。

当进行有符号整数减法运算时, 若 OV 置位, 则表示在正数减负数的过程中产生了负数; 或者, 在负数减正数的过程中产生了正数。

源操作数支持的寻址方式: 寄存器寻址、直接寻址、寄存器间接寻址和立即数寻址。

举例: 设累加器中的数据为 0C9H(11001001B)。寄存器 R2 的值为 54H(01010100B), 进位标志 C 被置位。则如下指令:

SUBB A, R2

执行后, 累加器的数据变为 74H(01110100B), 进位标志 C 和辅助进位标志 AC 被清零, 溢出标志 C 被置位。

注意: 0C9H 减去 54H 应该是 75H, 但在上面的计算中, 由于在 SUBB 指令执行前, 进位标志 C 已经被置位, 因而最终结果还需要减去进位标志, 得到 74H。因此, 如果在进行单

精度或者多精度减法运算前，进位标志 C 的状态未知，那么应改采用 CLR C 指令把进位标志 C 清零。

SUBB A, Rn

指令长度(字节) : 1

执行周期: 1

二进制编码:	1	0	0	1	1	r	r	r
--------	---	---	---	---	---	---	---	---

操作: SUBB

(A) \leftarrow (A) - (C) - (Rn)**SUBB A, direct**

指令长度(字节) : 2

执行周期: 1

二进制编码:	1	0	0	1	0	1	0	1	direct address
--------	---	---	---	---	---	---	---	---	----------------

操作: SUBB

(A) \leftarrow (A) - (C) - (direct)**SUBB A, @Ri**

指令长度(字节) : 1

执行周期: 1

二进制编码:	1	0	0	1	0	1	1	i
--------	---	---	---	---	---	---	---	---

操作: SUBB

(A) \leftarrow (A) - (C) - ((Ri))**SUBB A, #data**

指令长度(字节) : 2

执行周期: 1

二进制编码:	1	0	0	1	0	1	0	0	immediate data
--------	---	---	---	---	---	---	---	---	----------------

操作: SUBB

(A) \leftarrow (A) - (C) - #data**SWAP A**

功能: 交换累加器的高低半字节

说明: SWAP 指令把累加器的低 4 位（位 3~位 0）和高 4 位（位 7~位 4）数据进行交换。实际上 SWAP 指令也可视为 4 位的循环指令。该指令不影响标志位。

举例: 设累加器的内容为 0C5H (11000101B)，则指令:

SWAP A

执行后，累加器的内容变成 5CH (01011100B)。

指令长度(字节) : 1

执行周期: 1

二进制编码:	1	1	0	0	0	1	0	0
--------	---	---	---	---	---	---	---	---

操作: SWAP

(A₃₋₀) \leftrightarrow (A₇₋₄)**XCH A, <byte>**

功能: 交换累加器和字节变量的内容

说明: XCH 指令将<byte>所指定的字节变量的内容装载到累加器，同时将累加器的旧内容写入<byte>所指定的字节变量。指令中的源操作数和目的操作数允许的寻址方式：寄存器寻址、直接寻址和寄存器间接寻址。

举例: 设 R0 的内容为地址 20H，累加器的值为 3FH (00111111B)。内部 RAM 的 20H 单元的内容为 75H (01110101B)。则指令：

XCH A, @R0

执行后，内部 RAM 的 20H 单元的数据变为 3FH (00111111B)，累加器的内容变为 75H(01110101B)。

XCH A, Rn

指令长度(字节)： 1

执行周期： 1

1	1	0	0	1	r	r	r
---	---	---	---	---	---	---	---

操作： XCH

(A) \leftrightarrow (Rn)

XCH A, direct

指令长度(字节)： 2

执行周期： 1

1	1	0	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作： XCH

(A) \leftrightarrow (direct)

XCH A, @Ri

指令长度(字节)： 1

执行周期： 1

1	1	0	0	0	1	1	i
---	---	---	---	---	---	---	---

操作： XCH

(A) \leftrightarrow ((Ri))

XCHD A, @Ri

功能: 交换累加器和@Ri 对应单元中的数据的低 4 位

说明: XCHD 指令将累加器内容的低半字节（位 0~3，一般是十六进制数或 BCD 码）和间接寻址的内部 RAM 单元的数据进行交换，各自的高半字（位 7~4）节不受影响。另外，该指令不影响标志位。

举例: 设 R0 保存了地址 20H，累加器的内容为 36H (00110110B)。内部 RAM 的 20H 单元存储的数据为 75H (01110101B)。则指令：

XCHD A, @R0

执行后，内部 RAM 20H 单元的内容变成 76H (01110110B)，累加器的内容变为 35H(00110101B)。

指令长度(字节)： 1

执行周期： 1

1	1	0	1	0	1	1	i
---	---	---	---	---	---	---	---

操作： XCHD

(A₃₋₀) ↔ (R_{i3-0})

XRL <dest-byte>, <src-byte>

功能: 字节变量的逻辑异或

说明: XRL 指令将<dest-byte>和<src-byte>所代表的字节变量逐位进行逻辑异或运算, 结果保存在<dest-byte>所代表的字节变量里。该指令不影响标志位。

两个操作数组合起来共支持 6 种寻址方式: 当目的操作数为累加器时, 源操作数可以采用寄存器寻址、直接寻址、寄存器间接寻址和立即数寻址; 当目的操作数是可直接寻址的数据时, 源操作数可以是累加器或者立即数。

注意: 如果该指令被用来修改输出引脚上的状态, 那么 dest-byte 所代表的数据就是从端口输出数据锁存器中获取的数据, 而不是从引脚上读取的数据。

举例: 如果累加器和寄存器 0 的内容分别为 0C3H (11000011B) 和 0AAH(10101010B), 则指令:

XRL A, R0

执行后, 累加器的内容变成 69H (01101001B)。

当目的操作数是可直接寻址字节数据时, 该指令可把任何 RAM 单元或者寄存器中的各个位取反。具体哪些位会被取反, 在运行过程当中确定。指令:

XRL P1, #00110001B

执行后, P1 口的位 5、4、0 被取反。

XRL A, Rn

指令长度 (字节): 1

执行周期: 1

0	1	1	0	1	r	r	r
---	---	---	---	---	---	---	---

操作: XRL

(A) ←(A) ▼ (R_n)

XRL A, direct

指令长度 (字节): 2

执行周期: 1

0	1	1	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

操作: XRL

(A) ←(A) ▼ (direct)

XRL A, @Ri

指令长度 (字节): 1

执行周期: 1

0	1	1	0	0	1	1	i
---	---	---	---	---	---	---	---

操作: XRL

(A) ←(A) ▼ ((R_i))

XRL A, #data

指令长度 (字节): 2

执行周期: 1

0	1	1	0	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

操作: XRL

(A) \leftarrow (A)  #data

XRL direct, A

指令长度(字节): 2

执行周期: 1

0	1	1	0	0	0	1	0		direct address
---	---	---	---	---	---	---	---	--	----------------

操作: XRL

(direct) \leftarrow (direct)  (A)

XRL direct, #data

指令长度(字节): 3

执行周期: 1

0	1	1	0	0	0	1	1		direct address		immediate data
---	---	---	---	---	---	---	---	--	----------------	--	----------------

操作: XRL

(direct) \leftarrow (direct)  #data

13.4 指令详解 (英文)

ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label "SUBRTN" is at program memory location 0345H. After executing the instruction,

ACALL SUBRTN

at location 0123H, SP will contain 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC will contain 0345H.

Bytes: 2

Cycles: 3

A10	A9	A8	1	0	0	0	1		A7	A6	A5	A4	A3	A2	A1	A0
-----	----	----	---	---	---	---	---	--	----	----	----	----	----	----	----	----

Operation: ACALL

(PC) \leftarrow (PC) + 2

(SP) \leftarrow (SP) + 1

((SP)) \leftarrow (PC₇₋₀)

(SP) \leftarrow (SP) + 1

((SP)) \leftarrow (PC₁₅₋₈)
 (PC₁₀₋₀) \leftarrow page address

ADD A, <src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct register-indirect, or immediate.

Example: The Accumulator holds 0C3H(11000011B) and register 0 holds 0AAH (10101010B). The instruction, ADD A, R0 will leave 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A, Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	1	r	r	r
---	---	---	---	---	---	---	---

Operation: ADD

(A) \leftarrow (A) + (Rn)

ADD A, direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

Operation: ADD

(A) \leftarrow (A) + (direct)

ADD A, @Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	1	1	i
---	---	---	---	---	---	---	---

Operation: ADD

(A) \leftarrow (A) + ((Ri))

ADD A, #data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

Operation: ADD

(A) \leftarrow (A) + #data

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the Carry flag and the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H(11000011B) and register 0 holds 0AAH (10101010B) with the Carry. The instruction,

ADDC A,R0

will leave 6EH (01101110B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADDC A, Rn

Bytes: 1

Cycles: 1

0	0	1	1	1	r	r	r
---	---	---	---	---	---	---	---

Operation: ADDC

(A) \leftarrow (A) + (C) + (Rn)

ADDC A, direct

Bytes: 2

Cycles: 1

0	0	1	1	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

Operation: ADDC

(A) \leftarrow (A) + (C) + (direct)

ADDC A, @Ri

Bytes: 1

Cycles: 1

0	0	1	1	0	1	1	i
---	---	---	---	---	---	---	---

Operation: ADDC

(A) \leftarrow (A) + (C) + ((Ri))

ADDC A, #data

Bytes: 2

Cycles: 1

0	0	1	1	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

Operation: ADDC

(A) \leftarrow (A) + (C) + #data

AJMP addr11

Function: Absolute Jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2K block of program memory as the first byte of the instruction following AJMP.

Example: The label “JMPADR” is at program memory location 0123H. The instruction,
AJMP JMPADR
is at location 0345H and will load the PC with 0123H.

Bytes:	2
Cycles:	3
Encoding:	A10 A9 A8 0 0 0 0 1 A7 A6 A5 A4 A3 A2 A1 A0
Operation:	AJMP $(PC) \leftarrow (PC) + 2$ $(PC_{10:0}) \leftarrow \text{page address}$

ANL <dest-byte>, <src-byte>

Function:	Logical-AND for byte variables
Description:	ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data. <i>Note:</i> When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch not the input pins.
Example:	If the Accumulator holds 0C3H(11000011B) and register 0 holds 55H (01010101B) then the instruction, ANL A,R0 will leave 41H (01000001B) in the Accumulator. When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The instruction, ANL P1, #01110011B will clear bits 7, 3, and 2 of output port 1.

ANL A, Rn

Bytes:	1
Cycles:	1
Encoding:	0 1 0 1 1 r r r
Operation:	ANL $(A) \leftarrow (A) \wedge (R_n)$

ANL A, direct

Bytes:	2
Cycles:	1
Encoding:	0 1 0 1 0 1 0 1 direct address
Operation:	ANL $(A) \leftarrow (A) \wedge (\text{direct})$

ANL A, @Ri

Bytes:	1
Cycles:	1

Encoding:

0	1	0	1	0	1	1	i
---	---	---	---	---	---	---	---

Operation: ANL

$(A) \leftarrow (A) \wedge ((Ri))$

ANL A, #data

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	1	0	0		immediate data
---	---	---	---	---	---	---	---	--	----------------

Operation: ANL

$(A) \leftarrow (A) \wedge \#data$

ANL direct, A

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	0	1	0		direct address
---	---	---	---	---	---	---	---	--	----------------

Operation: ANL

$(direct) \leftarrow (direct) \wedge (A)$

ANL direct, #data

Bytes: 3

Cycles: 1

Encoding:

0	1	0	1	0	0	1	1		direct address		immediate data
---	---	---	---	---	---	---	---	--	----------------	--	----------------

Operation: ANL

$(direct) \leftarrow (direct) \wedge \#data$

ANL C, <src-bit>

Function: Logical-AND for bit variables

Description: If the Boolean value of the source bit is a logical 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash (“ / ”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected*. No other flags are affected.

Only direct addressing is allowed for the source operand.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC. 7 = 1, and OV = 0:

MOV C, P1.0 ; LOAD CARRY WITH INPUT PIN STATE

ANL C, ACC.7 ; AND CARRY WITH ACCUM. BIT.7

ANL C, /OV ; AND WITH INVERSE OF OVERFLOW FLAG

ANL C, bit

Bytes: 2

Cycles: 1

Encoding:

1	0	0	0	0	0	1	0		bit address
---	---	---	---	---	---	---	---	--	-------------

Operation: ANL

$(C) \leftarrow (C) \wedge (\text{bit})$

ANL C, /bit

Bytes: 2

Cycles: 1

Encoding:

1	0	1	1	0	0	0	0		bit address
---	---	---	---	---	---	---	---	--	-------------

Operation: ANL

$$(C) \leftarrow (C) \wedge (\overline{bit})$$

CJNE <dest-byte>, <src-byte>, rel

Function: Compare and Jump if Not Equal

Description: CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The Accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence,

CJNE R7,#60H, NOT_EQ

;	; R7 = 60H.
NOT_EQ:	JC REQ_LOW	; IF R7 < 60H.
;	; R7 > 60H.

sets the carry flag and branches to the instruction at label NOT-EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 34H, then the instruction,

WAIT: CJNE A,P1, WAIT

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on P1, the program will loop at this point until the P1 data changes to 34H.)

CJNE A, direct, rel

Bytes: 3

Cycles: 2 or 3

1	0	1	1	0	1	0	1		direct address		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

Operation: $(PC) \leftarrow (PC) + 3$

IF (A) <> (direct)

THEN

$(PC) \leftarrow (PC) + \text{relative offset}$

IF (A) < (direct)

THEN

$(C) \leftarrow 1$

ELSE

$(C) \leftarrow 0$

CJNE A, #data, rel

Bytes: 3

Cycles: 1 or 3

1	0	1	1	0	1	0	0		immediate data		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

Operation: $(PC) \leftarrow (PC) + 3$

```

IF (A) <> (data)
THEN
    (PC) ←(PC) +relative offset
IF (A) < (data)
THEN
    (C) ← 1
ELSE
    (C) ← 0

```

CJNE Rn, #data, rel

Bytes: 3
 Cycles: 2 or 3
 Encoding:

1	0	1	1	1	r	r	r		immediate data		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

 Operation: (PC) ←(PC) +3
 IF (Rn) <> (data)
 THEN
 (PC) ←(PC) +relative offset
 IF (Rn) < (data)
 THEN
 (C) ← 1
 ELSE
 (C) ← 0

CJNE @Ri, #data, rel

Bytes: 3
 Cycles: 2 or 3
 Encoding:

1	0	1	1	0	1	1	i		immediate data		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

 Operation: (PC) ←(PC) +3
 IF (Ri) <> (data)
 THEN
 (PC) ←(PC) +relative offset
 IF (Ri) < (data)
 THEN
 (C) ← 1
 ELSE
 (C) ← 0

CLRA

Function: Clear Accumulator
 Description: The Accumulator is cleared (all bits set on zero). No flags are affected.
 Example: The Accumulator contains 5CH (01011100B). The instruction,
 CLR A
 will leave the Accumulator set to 00H (00000000B).

Bytes: 1
 Cycles: 1
 Encoding:

1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---

Operation: CLR
 (A) \leftarrow 0

CLR bit

Function: Clear bit
 Description: The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.
 Example: Port 1 has previously been written with 5DH (01011101B). The instruction,
 CLR P1.2
 will leave the port set to 59H (01011001B).

CLR C

Bytes: 1
 Cycles: 1
 Encoding:

1	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

 Operation: CLR
 (C) \leftarrow 0

CLR bit

Bytes: 2
 Cycles: 1
 Encoding:

1	1	0	0	0	0	1	0	bit address
---	---	---	---	---	---	---	---	-------------

 Operation: CLR
 (bit) \leftarrow 0

CPL A

Function: Complement Accumulator
 Description: Each bit of the Accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to a zero and vice-versa. No flags are affected.
 Example: The Accumulator contains 5CH(01011100B). The instruction,
 CPL A
 will leave the Accumulator set to 0A3H (10100011B).

Bytes: 1
 Cycles: 1
 Encoding:

1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---

 Operation: CPL
 (A) \leftarrow (\overline{A})

CPL bit

Function: Complement bit
 Description: The bit variable specified is complemented. A bit which had been a one is changed to zero and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example: Port 1 has previously been written with 5BH(01011011B). The instruction,
CPL P1.1
CPL P1.2
will leave the port set to 5DH(01011101B).

CPL C

Bytes: 1
Cycles: 1
Encoding:

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

Operation: CPL

(C) \leftarrow (\bar{C})

CPL bit

Bytes: 2
Cycles: 1
Encoding:

1	0	1	1	0	0	1	0		bit address
---	---	---	---	---	---	---	---	--	-------------

Operation: CPL

(bit) \leftarrow (\bar{bit})

DAA

Function: Decimal-adjust Accumulator for Addition
Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If Accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set or if the four high-order bits now exceed nine(1010xxxx- 1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn't clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the Accumulator, depending on initial Accumulator and PSW conditions.

Note: DA A cannot simply convert a hexadecimal number in the Accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example: The Accumulator holds the value 56H(01010110B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B) representing the packed BCD digits of the decimal number 67. The carry flag is set. The instruction sequence.

ADDC A,R3

DA A

will first perform a standard two's-complement binary addition, resulting in the value 0BEH (10111110) in the Accumulator. The carry and auxiliary carry flags will be cleared.

The Decimal Adjust instruction will then alter the Accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56,67, and the carry-in. The carry flag will be set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the Accumulator initially holds 30H (representing the digits of 30 decimal), then the instruction sequence,

ADD A, #99H

DA A

will leave the carry set and 29H in the Accumulator, since $30+99=129$. The low-order byte of the sum can be interpreted to mean $30 - 1 = 29$.

Bytes: 1

Cycles: 3

Encoding:

1	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

Operation: DA

-contents of Accumulator are BCD

IF $[(A_{3:0}) > 9] \vee [(AC) = 1]$

THEN $(A_{3:0}) \leftarrow (A_{3:0}) + 6$

AND

IF $[(A_{7:4}) > 9] \vee [(C) = 1]$

THEN $(A_{7:4}) \leftarrow (A_{7:4}) + 6$

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7FH (0111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence,

DEC @R0

DEC R0

DEC @R0

will leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1
 Cycles: 1
 Encoding:

0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---

 Operation: DEC
 $(A) \leftarrow (A) - 1$

DEC Rn

Bytes: 1
 Cycles: 1
 Encoding:

0	0	0	1	1	r	r	r
---	---	---	---	---	---	---	---

 Operation: DEC
 $(Rn) \leftarrow (Rn) - 1$

DEC direct

Bytes: 2
 Cycles: 1
 Encoding:

0	0	0	1	0	1	0	1		Direct address
---	---	---	---	---	---	---	---	--	----------------

 Operation: DEC
 $(direct) \leftarrow (direct) - 1$

DEC @Ri

Bytes: 1
 Cycles: 1
 Encoding:

0	0	0	1	0	1	1	i
---	---	---	---	---	---	---	---

 Operation: DEC
 $((Ri)) \leftarrow ((Ri)) - 1$

DIV AB

Function: Divide
 Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The Accumulator contains 251(OFBH or 11111011B) and B contains 18(12H or 00010010B). The instruction,

DIV AB

will leave 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since $251 = (13 \times 18) + 17$. Carry and OV will both be cleared.

Bytes: 1
 Cycles: 6
 Encoding:

1	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

 Operation: DIV
 $(A)_{15-8}(B)_{7-0} \leftarrow (A)/(B)$

DJNZ <byte>, <rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to OFFH. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively.

The instruction sequence,

DJNZ 40H, LABEL_1

DJNZ 50H, LABEL_2

DJNZ 60H, LABEL_3

will cause a jump to the instruction at label LABEL_2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was not taken because the result was zero.

This instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence,

MOV R2,#8

TOOOLE: CPL P1.7

DJNZ R2, TOOGLE

will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1.

Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

DJNZ Rn, rel

Bytes: 2

Cycles: 2 or 3

1	1	0	1	1	r	r	r		rel. address
---	---	---	---	---	---	---	---	--	--------------

Operation: DJNZ

$(PC) \leftarrow (PC) + 2$

$(Rn) \leftarrow (Rn) - 1$

IF $(Rn) > 0$ or $(Rn) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

DJNZ direct, rel

Bytes: 3

Cycles: 2 or 3

1	1	0	1	0	1	0	1		direct address		rel. address
---	---	---	---	---	---	---	---	--	----------------	--	--------------

Operation: DJNZ

$(PC) \leftarrow (PC) + 2$

$(direct) \leftarrow (direct) - 1$

IF $(direct) > 0$ or $(direct) < 0$

THEN

$(PC) \leftarrow (PC) + rel$

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH will overflow to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7EH (0111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence,

INC @R0

INC R0

INC @R0

will leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H.

INC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

Operation: INC

$(A) \leftarrow (A) + 1$

INC Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	1	r	r	r
---	---	---	---	---	---	---	---

Operation: INC

$(Rn) \leftarrow (Rn) + 1$

INC direct

Bytes: 2

Cycles: 1

Encoding:

0	0	0	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

Operation: INC

$(direct) \leftarrow (direct) + 1$

INC @Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	1	1	i
---	---	---	---	---	---	---	---

Operation: INC

$((Ri)) \leftarrow ((Ri)) + 1$

INC DPTR

Function: Increment Data Pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H will increment the high-order-byte (DPH). No flags are affected.

This is the only 16-bit register which can be incremented.

Example: Register DPH and DPL contains 12H and 0FEH, respectively. The instruction sequence,

IINC DPTR

INC DPTR

INC DPTR

will change DPH and DPL to 13H and 01H.

Bytes: 1

Cycles: 1

Encoding:

1	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

Operation: INC

$(\text{DPTR}) \leftarrow (\text{DPTR}) + 1$

JB bit, rel

Function: Jump if Bit set

Description: If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction.

The bit tested is not modified. No flags are affected

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The instruction sequence,

JB P1.2, LABEL1

JB ACC.2, LABEL2

will cause program execution to branch to the instruction at label LABEL2.

Bytes: 3

Cycles: 1 or 3

Encoding:

0	0	1	0	0	0	0	0	bit address		rel. address
---	---	---	---	---	---	---	---	-------------	--	--------------

Operation: JB

$(\text{PC}) \leftarrow (\text{PC}) + 3$

IF (bit) = 1

THEN

$(\text{PC}) \leftarrow (\text{PC}) + \text{rel}$

JBC bit, rel

Function: Jump if Bit is set and Clear bit

Description: If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. *The bit will not be cleared if it is already a zero.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example: The Accumulator holds 56H (01010110B). The instruction sequence,
JBC ACC.3, LABEL1
JBC ACC.2, LABEL2
will cause program execution to continue at the instruction identified by the label LABEL2, with
the Accumulator modified to 52H (01010010B).

Bytes:	3
Cycles:	1 or 3
Encoding:	[0 0 0 1 0 0 0 0 bit address rel. address]
Operation:	JB
	$(PC) \leftarrow (PC) + 3$
	IF (bit) = 1
	THEN
	$(bit) \leftarrow 0$
	$(PC) \leftarrow (PC) + rel$

JC rel

Function: Jump if Carry is set
Description: If the carry flag is set, branch to the address indicated; otherwise proceed with the next
instruction. The branch destination is computed by adding the signed relative-displacement in the
second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The instruction sequence,
JC LABEL1
CPL C
JC LABEL2
will set the carry and cause program execution to continue at the instruction identified by the
label LABEL2.

Bytes:	2
Cycles:	1 or 3
Encoding:	[0 1 0 0 0 0 0 0 rel. address]
Operation:	JC
	$(PC) \leftarrow (PC) + 2$
	IF (C) = 1
	THEN
	$(PC) \leftarrow (PC) + rel$

JMP @A+DPTR

Function: Jump indirect
Description: Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer, and
load the resulting sum to the program counter. This will be the address for subsequent instruction

fetches. Sixteen-bit addition is performed (modulo 65536): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

Example: An even number from 0 to 6 is in the Accumulator. The following sequence of instructions will branch to one of four AJMP instructions in a jump table starting at JMP_TBL:

```
MOV DPTR, #JMP_TBL
JMP @A+DPTR
JMP-TBL: AJMP LABEL0
AJMP LABEL1
AJMP LABEL2
AJMP LABEL3
```

If the Accumulator equals 04H when starting this sequence, execution will jump to label LABEL2. Remember that AJMP is a two-byte instruction, so the jump instructions start at every other address.

Bytes:	1								
Cycles:	4								
Encoding:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	1	0	0	1	1
0	1	1	1	0	0	1	1		
Operation:	JMP $(PC) \leftarrow (A) + (DPTR)$								

JNB bit, rel

Function: Jump if Bit is not set

Description: If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The instruction sequence,

```
JNB P1.3, LABEL1
JNB ACC.3, LABEL2
```

will cause program execution to continue at the instruction at label LABEL2.

Bytes:	3											
Cycles:	1 or 3											
Encoding:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>bit address</td><td> </td><td>rel. address</td></tr></table>	0	0	1	1	0	0	0	0	bit address		rel. address
0	0	1	1	0	0	0	0	bit address		rel. address		
Operation:	JNB $(PC) \leftarrow (PC) + 3$ IF (bit) = 0 THEN $(PC) \leftarrow (PC) + rel$											

JNC rel

Function: Jump if Carry not set

Description: If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The instruction sequence,

JNC LABEL1

CPL C

JNC LABEL2

will clear the carry and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 1 or 3

Encoding:

0	1	0	1	0	0	0	0		rel. address
---	---	---	---	---	---	---	---	--	--------------

Operation: JNC

$(PC) \leftarrow (PC) + 2$

IF (C) = 0

THEN $(PC) \leftarrow (PC) + rel$

JNZ rel

Function: Jump if Accumulator Not Zero

Description: If any bit of the Accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally holds 00H. The instruction sequence,

JNZ LABEL1

INC A

JNZ LABEL2

will set the Accumulator to 01H and continue at label LABEL2.

Bytes: 2

Cycles: 1 or 3

Encoding:

0	1	1	1	0	0	0	0		rel. address
---	---	---	---	---	---	---	---	--	--------------

Operation: JNZ

$(PC) \leftarrow (PC) + 2$

IF (A) ≠ 0

THEN $(PC) \leftarrow (PC) + rel$

JZ rel

Function: Jump if Accumulator Zero

Description: If all bits of the Accumulator are zero, branch to the address indicated; otherwise proceed with

the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally contains 01H. The instruction sequence,

JZ LABEL1

DEC A

JZ LAEEL2

will change the Accumulator to 00H and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 1 or 3

Encoding:

0	1	1	0	0	0	0	0	rel. address
---	---	---	---	---	---	---	---	--------------

Operation: JZ

$(PC) \leftarrow (PC) + 2$

IF (A) = 0

THEN $(PC) \leftarrow (PC) + rel$

LCALL addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K-byte program memory address space. No flags are affected.

Example: Initially the Stack Pointer equals 07H. The label “SUBRTN” is assigned to program memory location 1234H. After executing the instruction,

LCALL SUBRTN

at location 0123H, the Stack Pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1234H.

Bytes: 3

Cycles: 3

Encoding:

0	0	0	1	0	0	1	0	addr15-addr8		addr7-addr0
---	---	---	---	---	---	---	---	--------------	--	-------------

Operation: LCALL

$(PC) \leftarrow (PC) + 3$

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (PC_{7:0})$

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (PC_{15:8})$

$(PC) \leftarrow \text{addr}_{15:0}$

LJMP addr16

Function: Long Jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label “JMPADR” is assigned to the instruction at program memory location 1234H. The instruction,

LJMP JMPADR

at location 0123H will load the program counter with 1234H.

Bytes: 3

Cycles: 3

0	0	0	0	0	0	1	0		addr15-addr8		addr7-addr0
---	---	---	---	---	---	---	---	--	--------------	--	-------------

Operation: LJMP

(PC) \leftarrow addr₁₅₋₀

MOV <dest-byte>, <src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (0CAH).

```
MOV R0, #30H      ; R0<=30H
MOV A, @R0        ; A<=40H
MOV R1, A         ; R1<=40H
MOV B, @R1        ; B<=10H
MOV @R1, P1       ; RAM (40H)<=0CAH
MOV P2, P1        ; P2 #0CAH
```

leaves the value 30H in register 0, 40H in both the Accumulator and register 1, 10H in register B, and 0CAH(11001010B) both in RAM location 40H and output on port 2.

MOVA,Rn

Bytes: 1

Cycles: 1

1	1	1	0	1	r	r	r
---	---	---	---	---	---	---	---

Operation: MOV

(A) \leftarrow (Rn)

*MOVA,direct

Bytes: 2

Cycles: 1

1	1	1	0	0	1	0	1		direct address
---	---	---	---	---	---	---	---	--	----------------

Operation: MOV

(A) \leftarrow (direct)

***MOVA,ACC is not a valid instruction.**

MOV A,@Ri

Bytes: 1
 Cycles: 1
 Encoding:

1	1	1	0	0	1	1	i
---	---	---	---	---	---	---	---

 Operation: MOV
 $(A) \leftarrow ((Ri))$

MOV A,#data

Bytes: 2
 Cycles: 1
 Encoding:

0	1	1	1	0	1	0	0	immediate data
---	---	---	---	---	---	---	---	----------------

 Operation: MOV
 $(A) \leftarrow \#data$

MOV Rn,A

Bytes: 1
 Cycles: 1
 Encoding:

1	1	1	1	1	r	r	r
---	---	---	---	---	---	---	---

 Operation: MOV
 $(Rn) \leftarrow (A)$

MOV Rn,direct

Bytes: 2
 Cycles: 1
 Encoding:

1	0	1	0	1	r	r	r	direct address
---	---	---	---	---	---	---	---	----------------

 Operation: MOV
 $(Rn) \leftarrow (\text{direct})$

MOV Rn,#data

Bytes: 2
 Cycles: 1
 Encoding:

0	1	1	1	1	r	r	r	immediate data
---	---	---	---	---	---	---	---	----------------

 Operation: MOV
 $(Rn) \leftarrow \#data$

MOV direct,A

Bytes: 2
 Cycles: 1
 Encoding:

1	1	1	1	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

 Operation: MOV
 $(\text{direct}) \leftarrow (A)$

MOV direct,Rn

Bytes: 2
 Cycles: 1
 Encoding:

1	0	0	0	1	r	r	r	direct address
---	---	---	---	---	---	---	---	----------------

 Operation: MOV
 $(\text{direct}) \leftarrow (Rn)$

MOV direct,direct

Bytes: 3
 Cycles: 1

Encoding:	1	0	0	0	0	1	0	1		dir.addr. (src)		dir.addr. (dest)
-----------	---	---	---	---	---	---	---	---	--	-----------------	--	------------------

Operation: MOV
(direct)←(direct)

MOV direct, @Ri

Bytes: 2
Cycles: 1
Encoding: 1 0 0 0 0 1 1 i
Operation: MOV
(direct)←((Ri))

MOV direct, #data

Bytes: 3
Cycles: 1
Encoding: 0 1 1 1 0 1 0 1
Operation: MOV
(direct)←#data

MOV @Ri, A

Bytes: 1
Cycles: 1
Encoding: 1 1 1 1 0 1 1 i
Operation: MOV
((Ri))←(A)

MOV @Ri, direct

Bytes: 2
Cycles: 1
Encoding: 1 0 1 0 0 1 1 i
Operation: MOV
((Ri))←(direct)

MOV @Ri, #data

Bytes: 2
Cycles: 1
Encoding: 0 1 1 1 0 1 1 i
Operation: MOV
((Ri))←#data

MOV <dest-bit>, <src-bit>

- Function: Move bit data
Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.
Example: The carry flag is originally set. The data present at input Port 3 is 11000101B. The data previously written to output Port 1 is 35H (00110101B).
MOV P1.3, C
MOV C, P3.3
MOV P1.2, C

will leave the carry cleared and change Port 1 to 39H (00111001B).

MOV C, bit

Bytes: 2

Cycles: 1

Encoding:	1	0	1	0	0	0	1	0		bit address
-----------	---	---	---	---	---	---	---	---	--	-------------

Operation: MOV

(C)← (bit)

MOV bit, C

Bytes: 2

Cycles: 1

Encoding:	1	0	0	1	0	0	1	0		bit address
-----------	---	---	---	---	---	---	---	---	--	-------------

Operation: MOV

(bit)← (C)

MOV DPTR , #data 16

Function: Load Data Pointer with a 16-bit constant

Description: The Data Pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,

MOV DPTR, #1234H

will load the value 1234H into the Data Pointer: DPH will hold 12H and DPL will hold 34H.

Bytes: 3

Cycles: 1

Encoding:	1	0	0	1	0	0	0	0	immediate data15-8		immediate data7-0
-----------	---	---	---	---	---	---	---	---	--------------------	--	-------------------

Operation: MOV

(DPTR) ← #data₁₅₋₀DPH DPL ← #data₁₅₋₈ #data₇₋₀**MOVCA , @A+ <base-reg>**

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit Accumulator contents and the contents of a sixteen-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the Accumulator. The following instructions will translate the value in the Accumulator to one of four values defined by the DB (define byte) directive.

REL-PC: INC A

MOVC A, @A+PC

RET

DB 66H

DB 77H

DB 88H

DB 99H

If the subroutine is called with the Accumulator equal to 01H, it will return with 77H in the Accumulator. The INC A before the MOVC instruction is needed to “get around” the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the Accumulator instead.

MOVC A,@A+DPTR

Bytes: 1

Cycles: 4

Encoding:

1	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Operation: MOVC

(A) \leftarrow ((A)+(DPTR))**MOVC A,@A+PC**

Bytes: 1

Cycles: 3

Encoding:

1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

Operation: MOVC

(PC) \leftarrow (PC)+1(A) \leftarrow ((A)+(PC))**MOVX <dest-byte> , <src-byte>**

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, hence the “X” appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM. In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the Data Pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be addressed via the Data Pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example: An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM.

Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H.

Location 34H of the external RAM holds the value 56H. The instruction sequence,

MOVX A, @R1

MOVX @R0, A

copies the value 56H into both the Accumulator and external RAM location 12H.

MOVX A,@Ri

Bytes: 1

Cycles: 3 or 1

Encoding:	1	1	1	0	0	0	1	i
-----------	---	---	---	---	---	---	---	---

Operation: MOVX

(A) ← ((Ri))

MOVX A,@DPTR

Bytes: 1

Cycles: 2 or 1

Encoding:	1	1	1	0	0	0	0	0
-----------	---	---	---	---	---	---	---	---

Operation: MOVX

(A) ← ((DPTR))

MOVX @Ri,A

Bytes: 1

Cycles: 3 or 1

Encoding:	1	1	1	1	0	0	1	i
-----------	---	---	---	---	---	---	---	---

Operation: MOVX

((Ri))←(A)

MOVX @DPTR,A

Bytes: 1

Cycles: 2 or 1

Encoding:	1	1	1	1	0	0	0	0
-----------	---	---	---	---	---	---	---	---

Operation: MOVX

(DPTR)←(A)

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned eight-bit integers in the Accumulator and register B. The low-order byte of the sixteen-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

The carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H).

The instruction,

MUL AB

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

Bytes: 1

Cycles: 2

Encoding:	1	0	1	0	0	1	0	0
-----------	---	---	---	---	---	---	---	---

Operation: $(A)_{7:0} \leftarrow (A) \times (B)$
 $(B)_{15:8}$

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Example: It is desired to produce a low-going output pulse on bit 7 of Port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence.

CLR P2.7

NOP

NOP

NOP

NOP

SETB P2.7

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Operation: NOP

$(PC) \leftarrow (PC) + 1$

ORL <dest-byte>, <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the instruction,

ORL A, R0

will leave the Accumulator holding the value 0D7H (11010111B).

When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time. The instruction,

ORL P1, #00110010B

will set bits 5,4, and 1 of output Port 1.

ORL A, Rn

Bytes: 1

Cycles: 1
 Encoding:

0	1	0	0	1	r	r	r
---	---	---	---	---	---	---	---

 Operation: ORL
 $(A) \leftarrow (A) \vee (R_n)$

ORL A, direct

Bytes: 2
 Cycles: 1
 Encoding:

0	1	0	0	0	1	0	1		direct address
---	---	---	---	---	---	---	---	--	----------------

 Operation: ORL
 $(A) \leftarrow (A) \vee (\text{direct})$

ORL A, @Ri

Bytes: 1
 Cycles: 1
 Encoding:

0	1	0	0	0	1	1	i
---	---	---	---	---	---	---	---

 Operation: ORL
 $(A) \leftarrow (A) \vee ((R_i))$

ORL A, #data

Bytes: 2
 Cycles: 1
 Encoding:

0	1	0	0	0	1	0	0		immediate data
---	---	---	---	---	---	---	---	--	----------------

 Operation: ORL
 $(A) \leftarrow (A) \vee \#data$

ORL direct, A

Bytes: 2
 Cycles: 1
 Encoding:

0	1	0	0	0	0	1	0		direct address
---	---	---	---	---	---	---	---	--	----------------

 Operation: ORL
 $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$

ORL direct, #data

Bytes: 3
 Cycles: 1
 Encoding:

0	1	0	0	0	0	1	1		direct address		immediate data
---	---	---	---	---	---	---	---	--	----------------	--	----------------

 Operation: ORL
 $(\text{direct}) \leftarrow (\text{direct}) \vee \#data$

ORL C, <src-bit>

Function: Logical-OR for bit variables
 Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise.
 A slash (“ / ”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if and only if P1.0 = 1, ACC. 7 = 1, or OV = 0:

```
MOV C, P1.0      ;LOAD CARRY WITH INPUT PIN P10
ORL C, ACC.7     ;OR CARRY WITH THE ACC.BIT 7
```

ORL C, /OV ;OR CARRY WITH THE INVERSE OF OV

ORL C, bit

Bytes: 2

Cycles: 1 or 4

0	1	1	1	0	0	1	0		bit address
---	---	---	---	---	---	---	---	--	-------------

Operation: ORL

$(C) \leftarrow (C) \vee (\text{bit})$

ORL C, /bit

Bytes: 2

Cycles: 1

1	0	1	0	0	0	0	0		bit address
---	---	---	---	---	---	---	---	--	-------------

Operation: ORL

$(C) \leftarrow (C) \vee (\overline{\text{bit}})$

POP direct

Function: Pop from stack

Description: The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The instruction sequence,

POP DPH

POP DPL

will leave the Stack Pointer equal to the value 30H and the Data Pointer set to 0123H. At this point the instruction,

POP SP

will leave the Stack Pointer set to 20H. Note that in this special case the Stack Pointer was decremented to 2FH before being loaded with the value popped (20H).

Bytes: 2

Cycles: 1

1	1	0	1	0	0	0	0		direct address
---	---	---	---	---	---	---	---	--	----------------

Operation: POP

$(\text{direct}) \leftarrow ((\text{SP}))$

$(\text{SP}) \leftarrow (\text{SP}) - 1$

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.

Example: On entering an interrupt routine the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The instruction sequence,

PUSH DPL

PUSH DPH

will leave the Stack Pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	0	0	0	
---	---	---	---	---	---	---	---	--

 direct address

Operation: PUSH

$(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (\text{direct})$

RET

Function: Return from subroutine

Description: RET pops the high-and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The Stack Pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

RET

will leave the Stack Pointer equal to the value 09H. Program execution will continue at location 0123H.

Bytes: 1

Cycles: 3

Encoding:

0	0	1	0	0	0	1	0
---	---	---	---	---	---	---	---

Operation: RET

$(PC_{15:8}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

$(PC_{7:0}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

RETI

Function: Return from interrupt

Description: RETI pops the high- and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt had been pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.

Example: The Stack Pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction,

RETI

will leave the Stack Pointer equal to 09H and return program execution to location 0123H.

Bytes: 1
Cycles: 3
Encoding:

0	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

Operation: RETI
 $(PC_{15:8}) \leftarrow ((SP))$
 $(SP) \leftarrow (SP) - 1$
 $(PC_{7:0}) \leftarrow ((SP))$
 $(SP) \leftarrow (SP) - 1$

RLA

Function: Rotate Accumulator Left
Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B). The instruction,
 RLA
 leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.
Bytes: 1
Cycles: 1
Encoding:

0	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

Operation: RL
 $(An+1) \leftarrow (An) \quad n = 0-6$
 $(A0) \leftarrow (A7)$

RLCA

Function: Rotate Accumulator Left through the Carry flag
Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,
 RLC A
 leaves the Accumulator holding the value 8BH (10001011B) with the carry set.
Bytes: 1
Cycles: 1
Encoding:

0	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

Operation: RLC
 $(An+1) \leftarrow (An) \quad n = 0-6$
 $(A0) \leftarrow (C)$
 $(C) \leftarrow (A7)$

RR A

Function: Rotate Accumulator Right

Description: The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction,
RR A
leaves the Accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

Operation: RR
 $(An) \leftarrow (An+1) \quad n = 0 - 6$
 $(A7) \leftarrow (A0)$

RRC A

Function: Rotate Accumulator Right through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,
RRC A
leaves the Accumulator holding the value 62H (01100010B) with the carry set.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Operation: RRC
 $(An+1) \leftarrow (An) \quad n = 0 - 6$
 $(A7) \leftarrow (C)$
 $(C) \leftarrow (A0)$

SETB <bit>

Function: Set bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B).
The instructions,
SETB C
SETB P1.0
will leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Operation: SETB
 $(C) \leftarrow 1$

SETB bit

Bytes:	2										
Cycles:	1										
Encoding:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td></td><td>bit address</td></tr></table>	1	1	0	1	0	0	1	0		bit address
1	1	0	1	0	0	1	0		bit address		
Operation:	SETB										
	$(\text{bit}) \leftarrow 1$										

SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128bytes preceding this instruction to 127 bytes following it.

Example: The label “RELADR” is assigned to an instruction at program memory location 0123H. The instruction,

SJMP RELADR

will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.

(Note: Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H - 0102H) = 21H. Put another way, an SJMP with a displacement of OFEH would be an one-instruction infinite loop).

Bytes: 2

Cycles: 3

Encoding:	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td>rel. address</td></tr></table>	1	0	0	0	0	0	0	0		rel. address
1	0	0	0	0	0	0	0		rel. address		

Operation: SJMP

$(\text{PC}) \leftarrow (\text{PC}) + 2$

$(\text{PC}) \leftarrow (\text{PC}) + \text{rel}$

SUBB A, <src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow)flag if a borrow is needed for bit 7, and clears C otherwise.(If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the Accumulator along with the source operand).AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry

flag is set. The instruction,

SUBB A, R2

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A, Rn

Bytes: 1

Cycles: 1

Encoding:

1	0	0	1	1	r	r	r
---	---	---	---	---	---	---	---

Operation: SUBB

(A) \leftarrow (A) - (C) - (Rn)

SUBB A, direct

Bytes: 2

Cycles: 1

Encoding:

1	0	0	1	0	1	0	1		direct address
---	---	---	---	---	---	---	---	--	----------------

Operation: SUBB

(A) \leftarrow (A) - (C) - (direct)

SUBB A, @Ri

Bytes: 1

Cycles: 1

Encoding:

1	0	0	1	0	1	1	i
---	---	---	---	---	---	---	---

Operation: SUBB

(A) \leftarrow (A) - (C) - ((Ri))

SUBB A, #data

Bytes: 2

Cycles: 1

Encoding:

1	0	0	1	0	1	0	0		immediate data
---	---	---	---	---	---	---	---	--	----------------

Operation: SUBB

(A) \leftarrow (A) - (C) - #data

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction,

SWAP A

leaves the Accumulator holding the value 5CH (01011100B).

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	0	0
---	---	---	---	---	---	---	---

Operation: SWAP

(A₃₋₀) ↔ (A₇₋₄)

XCH A, <byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (0011111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,
XCH A, @R0
will leave RAM location 20H holding the values 3FH (0011111B) and 75H (01110101B) in the accumulator.

XCH A, Rn

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	1	r	r	r
---	---	---	---	---	---	---	---

Operation: XCH

(A) ↔ (Rn)

XCH A, direct

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

Operation: XCH

(A) ↔ (direct)

XCH A, @Ri

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	1	i
---	---	---	---	---	---	---	---

Operation: XCH

(A) ↔ ((Ri))

XCHD A, @Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3-0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM

location 20H holds the value 75H (01110101B). The instruction,
XCHD A, @R0
 will leave RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the
 accumulator.

Bytes: 1
 Cycles: 1
 Encoding:

1	1	0	1	0	1	1	i
---	---	---	---	---	---	---	---

 Operation: XCHD

(A₃₋₀) ↔ (R_{i3-0})

XRL <dest-byte>, <src-byte>

Function: Logical Exclusive-OR for byte variables
 Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.
 The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.
(Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.)
 Example: If the Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,
XRL A, R0
 will leave the Accumulator holding the value 69H (01101001B).
 When the destination is a directly addressed byte, this instruction can complement combination of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The instruction,
XRL P1, #00110001B
 will complement bits 5,4 and 0 of output Port 1.

XRL A, Rn

Bytes: 1
 Cycles: 1
 Encoding:

0	1	1	0	1	r	r	r
---	---	---	---	---	---	---	---

 Operation: XRL

(A) ←(A)  (Rn)

XRL A, direct

Bytes: 2
 Cycles: 1
 Encoding:

0	1	1	0	0	1	0	1	direct address
---	---	---	---	---	---	---	---	----------------

 Operation: XRL

(A) ←(A)  (direct)

XRL A, @Ri

Bytes: 1
 Cycles: 1
 Encoding:

0	1	1	0	0	1	1	i
---	---	---	---	---	---	---	---

 Operation: XRL

(A) \leftarrow (A)  ((Ri))

XRL A, #data

Bytes: 2
 Cycles: 1
 Encoding:

0	1	1	0	0	1	0	0		immediate data
---	---	---	---	---	---	---	---	--	----------------

 Operation: XRL

(A) \leftarrow (A)  #data

XRL direct, A

Bytes: 2
 Cycles: 1
 Encoding:

0	1	1	0	0	0	1	0		direct address
---	---	---	---	---	---	---	---	--	----------------

 Operation: XRL

(direct) \leftarrow (direct)  (A)

XRL direct, #data

Bytes: 3
 Cycles: 1
 Encoding:

0	1	1	0	0	0	1	1		direct address		immediate data
---	---	---	---	---	---	---	---	--	----------------	--	----------------

 Operation: XRL

(direct) \leftarrow (direct)  #data

14 中断系统

(C 语言程序中使用中断号大于 31 的中断时, 在 Keil 中编译会报错, 解决办法请参考章节“[扩展 Keil 对中断号数量的支持, 中断号大于 31 编译出错的处理](#)”)

中断系统是为使 CPU 具有对外界紧急事件的实时处理能力而设置的。

当中央处理机 CPU 正在处理某件事的时候外界发生了紧急事件请求, 要求 CPU 暂停当前的工作, 转而去处理这个紧急事件, 处理完以后, 再回到原来被中断的地方, 继续原来的工作, 这样的过程称为中断。实现这种功能的部件称为中断系统, 请示 CPU 中断的请求源称为中断源。微型机的中断系统一般允许多个中断源, 当几个中断源同时向 CPU 请求中断, 要求为它服务的时候, 这就存在 CPU 优先响应哪一个中断源请求的问题。通常根据中断源的轻重缓急排队, 优先处理最紧急事件的中断请求源, 即规定每一个中断源有一个优先级别。CPU 总是先响应优先级别最高的中断请求。

当 CPU 正在处理一个中断源请求的时候(执行相应的中断服务程序), 发生了另外一个优先级比它还高的中断源请求。如果 CPU 能够暂停对原来中断源的服务程序, 转而去处理优先级更高的中断请求源, 处理完以后, 再回到原低级中断服务程序, 这样的过程称为中断嵌套。这样的中断系统称为多级中断系统, 没有中断嵌套功能的中断系统称为单级中断系统。

用户可以用关总中断允许位(EA/IE.7)或相应中断的允许位屏蔽相应的中断请求, 也可以用打开相应的中断允许位来使 CPU 响应相应的中断申请, 每一个中断源可以用软件独立地控制为开中断或关中断状态, 部分中断的优先级别均可用软件设置。高优先级的中断请求可以打断低优先级的中断, 反之, 低优先级的中断请求不可以打断高优先级的中断。当两个相同优先级的中断同时产生时, 将由查询次序来决定系统先响应哪个中断。

14.1 STC8H 系列中断源

下表中 √ 表示对应的系列有相应的中断源

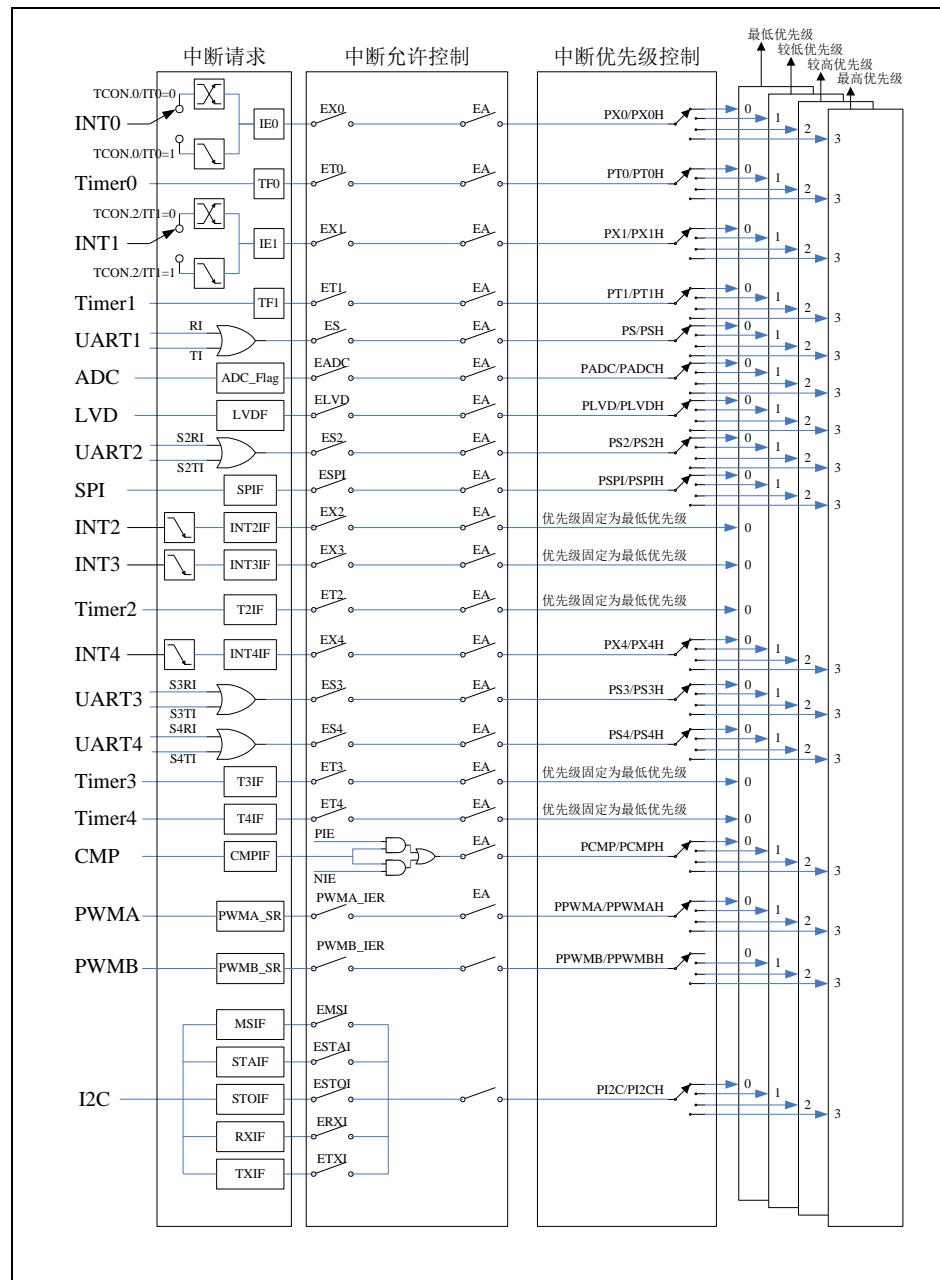
中断源	STC8H1K16系列	STC8H1K08系列	STC8H3K64S4系列	STC8H3K64S2系列	STC8H8K64U-AW系列	STC8H8K64U-B/C/D系列	STC8H1K08T系列	STC8H4K64TLCD系列	STC8H4K64TL系列	STC8H2K12U系列	STC8H2K32U系列
外部中断 0 中断 (INT0) 支持下降沿和边沿中断	√	√	√	√	√	√	√	√	√	√	√
定时器 0 中断 (Timer0)	√	√	√	√	√	√	√	√	√	√	√
外部中断 1 中断 (INT1) 支持下降沿和边沿中断	√	√	√	√	√	√	√	√	√	√	√
定时器 1 中断 (Timer1)	√	√	√	√	√	√	√	√	√	√	√
串口 1 中断 (UART1)	√	√	√	√	√	√	√	√	√	√	√
模数转换中断 (ADC)	√	√	√	√	√	√	√	√	√	√	√
低压检测中断 (LVD)	√	√	√	√	√	√	√	√	√	√	√
串口 2 中断 (UART2)	√	√	√	√	√	√	√	√	√	√	√
串行外设接口中断 (SPI)	√	√	√	√	√	√	√	√	√	√	√

中断源	STC8H1K16系列	STC8H1K08系列	STC8H3K64S4系列	STC8H3K64S2系列	STC8H8K64U-A系列	STC8H8K64U-B/C/D系列	STC8H1K08T系列	STC8H4K64TL系列	STC8H4K64LCD系列	STC8H2K12U系列	STC8H2K32U系列
外部中断 2 中断 (INT2) 支持下降沿中断	√	√	√	√	√	√	√	√	√	√	√
外部中断 3 中断 (INT3) 支持下降沿中断	√	√	√	√	√	√	√	√	√	√	√
定时器 2 中断 (Timer2)	√	√	√	√	√	√	√	√	√	√	√
外部中断 4 中断 (INT4)	√	√	√	√	√	√	√	√	√	√	√
串口 3 中断 (UART3)			√		√	√		√	√		
串口 4 中断 (UART4)			√		√	√		√	√		
定时器 3 中断 (Timer3)	√		√	√	√	√		√	√		√
定时器 4 中断 (Timer4)	√		√	√	√	√		√	√		√
比较器中断 (CMP)	√	√	√	√	√	√	√	√	√	√	√
I2C 总线中断	√	√	√	√	√	√	√	√	√	√	√
PWMA	√	√	√	√	√	√	√	√	√	√	√
PWMB	√	√	√	√	√	√	√	√	√	√	√
USB 中断					√	√				√	√
触摸按键中断							√	√	√		
RTC 中断						√	√	√	√	√	√
P0 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√		√	√		√
P1 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√	√	√	√	√	√
P2 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√		√	√		√
P3 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√	√	√	√		√
P4 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√		√	√		
P5 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√	√	√	√	√	√
P6 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√					
P7 口中断 支持下降沿、上升沿、高电平和低电平中断			√	√		√					
DMA_M2M 中断						√	√	√	√		
DMA_ADC 中断						√	√	√	√		

中断源	STC8H1K16系列	STC8H1K08系列	STC8H3K64S4系列	STC8H3K64S2系列	STC8H8K64U-A系列	STC8H8K64U-B/C/D系列	STC8H1K08T系列	STC8H4K64TL系列	STC8H4K64LCD系列	STC8H2K32UN系列
DMA_SPI 中断						√	√	√	√	
DMA_UR1T 中断						√	√	√	√	
DMA_UR1R 中断						√	√	√	√	
DMA_UR2T 中断						√	√	√	√	
DMA_UR2R 中断						√	√	√	√	
DMA_UR3T 中断						√	√	√	√	
DMA_UR3R 中断						√	√	√	√	
DMA_UR4T 中断						√	√	√	√	
DMA_UR4R 中断						√	√	√	√	
DMA_LCM 中断						√	√	√	√	
LCM 中断						√		√	√	
定时器 11 中断									√	√

14.2 STC8H 中断及中断优先级结构图

注: 高优先级中断可以打断正在执行中的低优先级中断, 跟传统 STC89C52 一样



注: 由于 STC8H 系列的中断源太多, 无法在此结构图中全部画出, 上面仅给出早期 STC8H 系列的部分中断结构图

14.3 STC8H 系列中断向量地址及同级中断优先级中断查询次序表

中断源	中断向量	次序	优先级设置	优先级	中断请求位	中断允许位
INT0	0003H	0	PX0PX0H	0/1/2/3	IE0	EX0
Timer0	000BH	1	PT0,PT0H	0/1/2/3	TF0	ET0
INT1	0013H	2	PX1,PX1H	0/1/2/3	IE1	EX1
Timer1	001BH	3	PT1,PT1H	0/1/2/3	TF1	ET1
UART1	0023H	4	PS,PSH	0/1/2/3	RI TI	ES
ADC	002BH	5	PADC,PADCH	0/1/2/3	ADC_FLAG	EADC
LVD	0033H	6	PLVD,PLVDH	0/1/2/3	LVDF	ELVD
PCA	003BH	7	PPCA,PPCAH	0/1/2/3	CF	ECF
					CCF0	ECCF0
					CCF1	ECCF1
					CCF2	ECCF2
					CCF3	ECCF3
UART2	0043H	8	PS2,PS2H	0/1/2/3	S2RI S2TI	ES2
SPI	004BH	9	PSPI,PSPIH	0/1/2/3	SPIF	ESPI
INT2	0053H	10		0	INT2IF	EX2
INT3	005BH	11		0	INT3IF	EX3
Timer2	0063H	12		0	T2IF	ET2
INT4	0083H	16	PX4,PX4H	0/1/2/3	INT4IF	EX4
UART3	008BH	17	PS3,PS3H	0/1/2/3	S3RI S3TI	ES3
UART4	0093H	18	PS4,PS4H	0/1/2/3	S4RI S4TI	ES4
Timer3	009BH	19		0	T3IF	ET3
Timer4	00A3H	20		0	T4IF	ET4
CMP	00ABH	21	PCMP,PCMPH	0/1/2/3	CMPIF	PIE NIE
I2C	00C3H	24	PI2C,PI2CH	0/1/2/3	MSIF	EMSI
					STAIF	ESTAI
					RXIF	ERXI
					TXIF	ETXI
					STOIF	ESTOI

中断源	中断向量	次序	优先级设置	优先级	中断请求位	中断允许位
USB	00CBH	25	PUSB,PUSBH	0/1/2/3	USB Events	EUSB
PWMA	00D3H	26	PPWMA,PPWMAH	0/1/2/3	PWMA_SR	PWMA_IER
PWMB	00DBH	27	PPWMB,PPWMBH	0/1/2/3	PWMB_SR	PWMB_IER
TKSU	011BH	35	PTKSU,PTKSUH	0/1/2/3	TKIF	ETKSUI
RTC	0123H	36	PRTC,PRTCH	0/1/2/3	ALAIF	EALAI
					DAYIF	EDAYI
					HOURIF	EHOURI
					MINIF	EMINI
					SECIF	ESECI
					SEC2IF	ESEC2I
					SEC8IF	ESEC8I
					SEC32IF	ESEC32I
P0 中断	012BH	37	PINIPL[0], PINIPH[0]	0/1/2/3	P0INTF	P0INTE
P1 中断	0133H	38	PINIPL[1], PINIPH[1]	0/1/2/3	P1INTF	P1INTE
P2 中断	013BH	39	PINIPL[2], PINIPH[2]	0/1/2/3	P2INTF	P2INTE
P3 中断	0143H	40	PINIPL[3], PINIPH[3]	0/1/2/3	P3INTF	P3INTE
P4 中断	014BH	41	PINIPL[4], PINIPH[4]	0/1/2/3	P4INTF	P4INTE
P5 中断	0153H	42	PINIPL[5], PINIPH[5]	0/1/2/3	P5INTF	P5INTE
P6 中断	015BH	43	PINIPL[6], PINIPH[6]	0/1/2/3	P6INTF	P6INTE
P7 中断	0163H	44	PINIPL[7], PINIPH[7]	0/1/2/3	P7INTF	P7INTE
DMA_M2M 中断	017BH	47	M2MIP[1:0]	0/1/2/3	M2MIF	M2MIE
DMA_ADC 中断	0183H	48	ADCIP[1:0]	0/1/2/3	ADCIF	ADCIE
DMA_SPI 中断	018BH	49	SPIIP[1:0]	0/1/2/3	SPIIF	SPIIE
DMA_UR1T 中断	0193H	50	UR1TIP[1:0]	0/1/2/3	UR1TIF	UR1TIE
DMA_UR1R 中断	019BH	51	UR1RIP[1:0]	0/1/2/3	UR1RIF	UR1RIE
DMA_UR2T 中断	01A3H	52	UR2TIP[1:0]	0/1/2/3	UR2TIF	UR2TIE
DMA_UR2R 中断	01ABH	53	UR2RIP[1:0]	0/1/2/3	UR2RIF	UR2RIE
DMA_UR3T 中断	01B3H	54	UR3TIP[1:0]	0/1/2/3	UR3TIF	UR3TIE
DMA_UR3R 中断	01BBH	55	UR3RIP[1:0]	0/1/2/3	UR3RIF	UR3RIE
DMA_UR4T 中断	01C3H	56	UR4TIP[1:0]	0/1/2/3	UR4TIF	UR4TIE
DMA_UR4R 中断	01CBH	57	UR4RIP[1:0]	0/1/2/3	UR4RIF	UR3RIE
DMA_LCM 中断	01D3H	58	LCMIP[1:0]	0/1/2/3	LCMIF	LCMIE
LCM 中断	01DBH	59	LCMIFIP[1:0]	0/1/2/3	LCMIFIF	LCMIFIE
Timer11	021BH	67		0	T11IF	ET11I

在 C 语言中声明中断服务程序

```
void    INT0_Routine(void)      interrupt 0;
void    TM0_Routine(void)       interrupt 1;
void    INT1_Routine(void)      interrupt 2;
void    TM1_Routine(void)       interrupt 3;
void    UART1_Routine(void)     interrupt 4;
void    ADC_Routine(void)       interrupt 5;
void    LVD_Routine(void)       interrupt 6;
void    PCA_Routine(void)       interrupt 7;
void    UART2_Routine(void)     interrupt 8;
void    SPI_Routine(void)       interrupt 9;
void    INT2_Routine(void)      interrupt 10;
void    INT3_Routine(void)      interrupt 11;
void    TM2_Routine(void)       interrupt 12;
void    INT4_Routine(void)      interrupt 16;
void    UART3_Routine(void)     interrupt 17;
void    UART4_Routine(void)     interrupt 18;
void    TM3_Routine(void)       interrupt 19;
void    TM4_Routine(void)       interrupt 20;
void    CMP_Routine(void)       interrupt 21;
void    I2C_Routine(void)       interrupt 24;
void    USB_Routine(void)       interrupt 25;
void    PWMA_Routine(void)      interrupt 26;
void    PWMB_Routine(void)      interrupt 27;
```

中断号超过31的C语言中断服务程序不能直接用interrupt声明，请参考章节“[扩展Keil对中断号数量的支持，中断号大于31编译出错的处理](#)”的处理方法，汇编语言不受影响

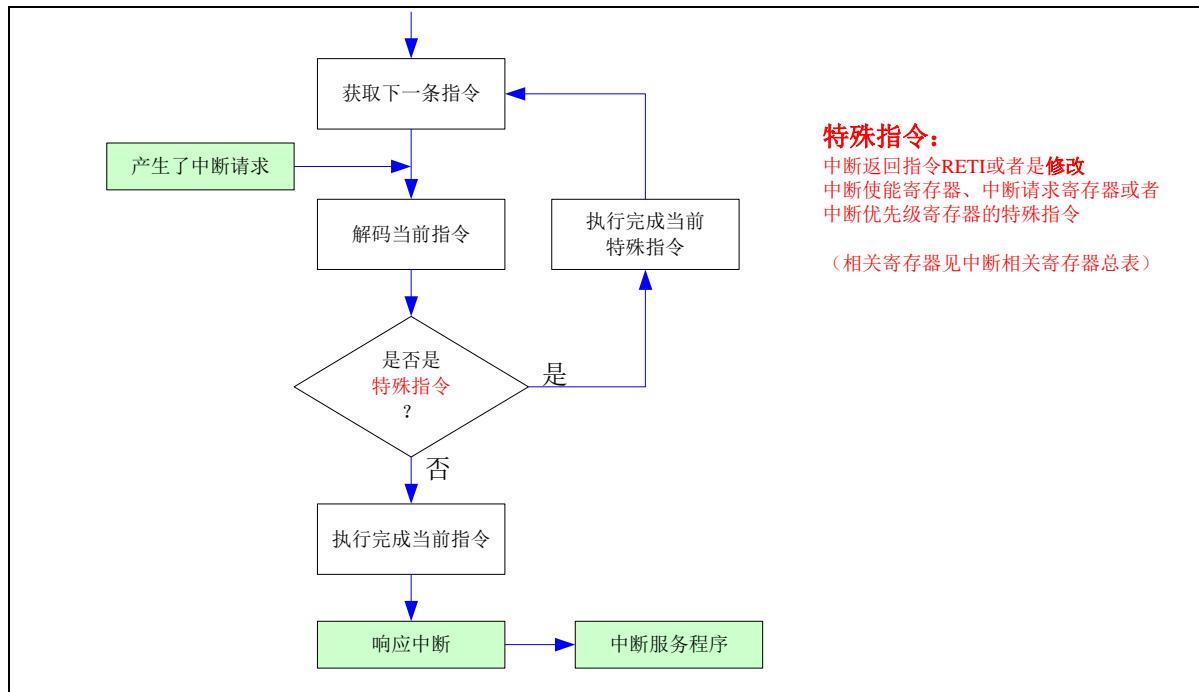
14.4 多级流水线内核的中断响应

STC 的增强型 8051 (例如: STC8G/STC8H 系列) 和 32 位 8051 (例如: STC32G 系列) 的 MCU 内核为多级流水线设计, 在中断响应方面的设计和传统的 8051 (例如: STC89C52 系列) 略有差异。

对于传统的 8051 (例如: STC89C52 系列):

如果当前正在执行的指令是中断返回指令 RETI 或者是访问中断使能寄存器、中断请求寄存器或者中断优先级寄存器的特殊指令时, CPU 等当前的这条特殊的指令执行完, 再执行一条指令才能响应中断请求;

如果当前正在执行的指令不是上面所指的特殊指令, 则等当前指令执行完成后就立即响应中断请求;



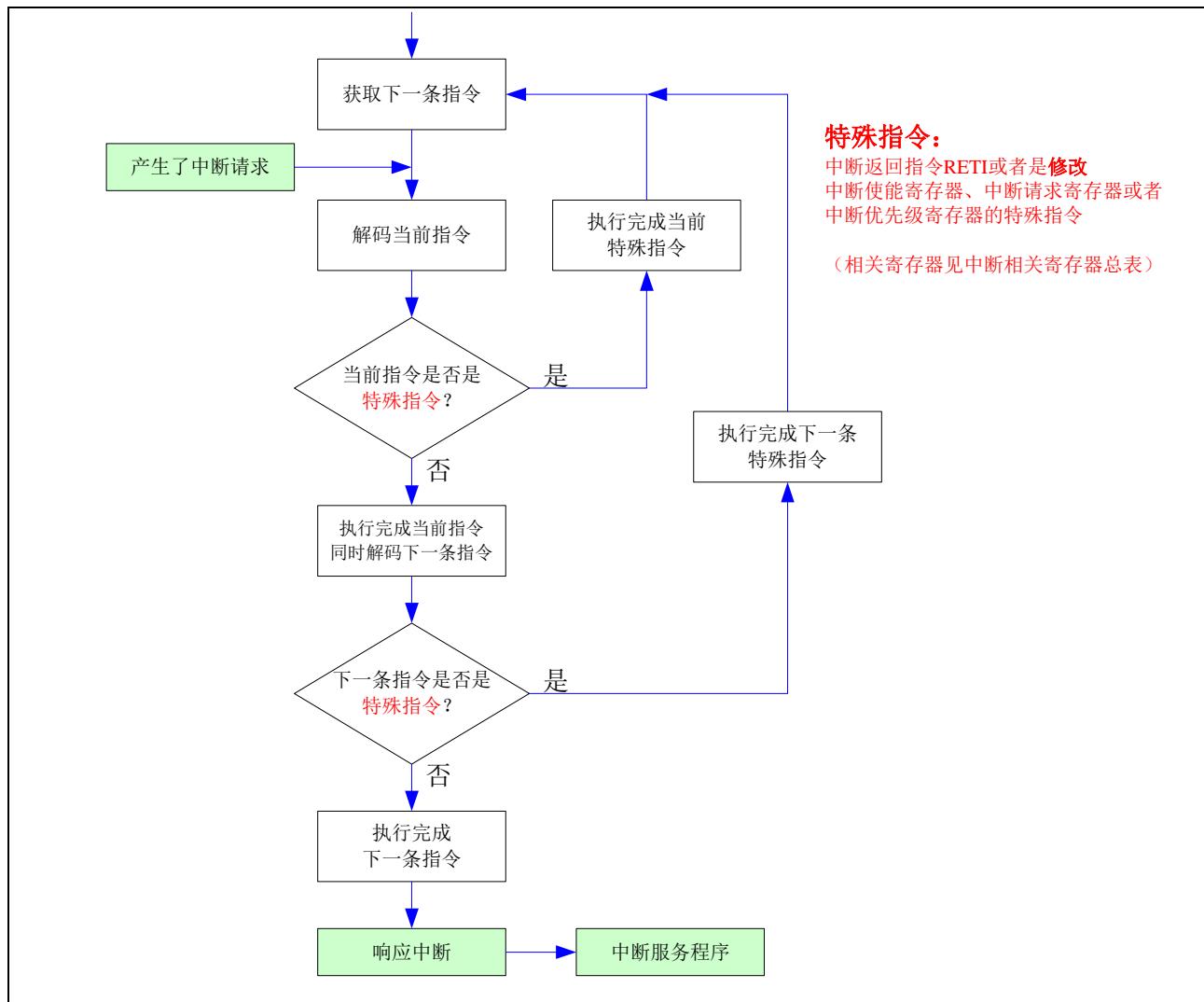
中断相关寄存器总表:

IE	IE2	IP	IPH	IP2
IP2H	IP3	IP3H	TCON	INTCLKO
AUXINTIF	SCON	S2CON	S3CON	S4CON
PCON	ADC_CONTR	SPSTAT	CMPCR1	LCMIFCFG
LCMIFSTA	T11CR	I2CMSCR	I2CMSST	I2CSLCR
I2CSLST	PWMA_IER	PWMA_SR1	PWMA_SR2	PWMB_IER
PWMB_SR1	PWMB_SR2	P0INTE	P1INTE	P2INTE
P3INTE	P4INTE	P5INTE	P6INTE	P7INTE
P0INTF	P1INTF	P2INTF	P3INTF	P4INTF
P5INTF	P6INTF	P7INTF	PINIPL	PINIPH
DMA_M2M_CFG	DMA_ADC_CFG	DMA_SPI_CFG	DMA_UR1T_CFG	DMA_UR1R_CFG
DMA_UR2T_CFG	DMA_UR2R_CFG	DMA_UR3T_CFG	DMA_UR3R_CFG	DMA_UR4T_CFG
DMA_UR4R_CFG	DMA_LCM_CFG	DMA_M2M_STA	DMA_ADC_STA	DMA_SPI_STA
DMA_UR1T_STA	DMA_UR1R_STA	DMA_UR2T_STA	DMA_UR2R_STA	DMA_UR3T_STA
DMA_UR3R_STA	DMA_UR4T_STA	DMA_UR4R_STA	DMA_LCM_STA	

对于 STC 的增强型 8051 单片机（例如：STC8G/STC8H 系列），由于是多级流水线设计，响应中断上会比传统的 8051（例如：STC89C52 系列）再多执行一条语句：

如果目前正在执行的指令是中断返回指令 RETI 或者是访问中断使能寄存器、中断请求寄存器或者中断优先级寄存器的特殊指令时，CPU 等当前的这条特殊的指令执行完，同时解码下一条指令，直到下一条指令不是特殊指令，则等下一条指令执行完成才能响应中断请求；

如果目前正在执行的指令不是上面所指的特殊指令，则等当前指令执行完成后，同时会解码下一条指令，如果下一条也不是特殊指令，则会等下一条指令执行完成后立即响应中断请求；



14.5 中断相关寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
IE	中断允许寄存器	A8H	EA	ELVD	EADC	ES	ET1	EX1	ETO	EX0	0000,0000
IE2	中断允许寄存器 2	AFH	EUSB ETKSU	ET4	ET3	ES4	ES3	ET2	ESPI	ES2	0000,0000
INTCLKO	中断与时钟输出控制寄存器	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO	x000,x000
IP	中断优先级控制寄存器	B8H	-	PLVD	PADC	PS	PT1	PX1	PT0	PX0	x000,0000
IPH	高中断优先级控制寄存器	B7H	-	PLVDH	PADCH	PSH	PT1H	PX1H	PT0H	PX0H	x000,0000
IP2	中断优先级控制寄存器 2	B5H	PUSB PTKSU	PI2C	PCMP	PX4	PPWMB	PPWMA	PSPI	PS2	0000,0000
IP2H	高中断优先级控制寄存器 2	B6H	PUSBH PTKSUH	PI2CH	PCMMPH	PX4H	PPWMBH	PPWMAH	PSPIH	PS2H	0000,0000
IP3	中断优先级控制寄存器 3	DFH	-	-	-	-	-	PRTC	PS4	PS3	xxxx,x000
IP3H	高中断优先级控制寄存器 3	EEH	-	-	-	-	-	PRTCH	PS4H	PS3H	xxxx,x000
TCON	定时器控制寄存器	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000,0000
AUXINTIF	扩展外部中断标志寄存器	EFH	-	INT4IF	INT3IF	INT2IF	-	T4IF	T3IF	T2IF	x000,x000
SCON	串口 1 控制寄存器	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000,0000
S2CON	串口 2 控制寄存器	9AH	S2SM0	-	S2SM2	S2REN	S2TB8	S2RB8	S2TI	S2RI	0100,0000
S3CON	串口 3 控制寄存器	ACH	S3SM0	S3ST3	S3SM2	S3REN	S3TB8	S3RB8	S3TI	S3RI	0000,0000
S4CON	串口 4 控制寄存器	84H	S4SM0	S4ST4	S4SM2	S4REN	S4TB8	S4RB8	S4TI	S4RI	0000,0000
PCON	电源控制寄存器	87H	SMOD	SMODO	LVDF	POF	GF1	GF0	PD	IDL	0011,0000
ADC_CONTR	ADC 控制寄存器	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]				000x,0000
SPSTAT	SPI 状态寄存器	CDH	SPIF	WCOL	-	-	-	-	-	-	00xx,xxxx
CMPCR1	比较器控制寄存器 1	E6H	CMPEN	CMPIF	PIE	NIE	PIS	NIS	CMPOE	CMPRES	0000,0000

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
LCMIFCFG	LCM 接口配置寄存器	FE50H	LCMIFIE	-	LCMIFIP[1:0]		LCMIFDPS[1:0]		D16_D8	M68_I80	0x00,0000	
LCMIFSTA	LCM 接口状态寄存器	FE53H	-	-	-	-	-	-	-	LCMIFIF	xxxx,xxx0	
T11CR	T11 控制寄存器	FE78H	T11R	T11_C/T	T11CLKO	T11x12	T11CS[1:0]		ET11I	T11IF	0000,0000	
I2CMSCR	I ² C 主机控制寄存器	FE81H	EMSI	-	-	-	MSCMD[3:0]					0xxx,0000
I2CMSST	I ² C 主机状态寄存器	FE82H	MSBUSY	MSIF	-	-	-	-	MSACKI	MSACKO	00xx,xx10	
I2CSLCR	I ² C 从机控制寄存器	FE83H	-	ESTAI	ERXI	ETXI	ESTOI	-	-	SLRST	x000,0xx0	
I2CSLST	I ² C 从机状态寄存器	FE84H	SLBUSY	STAIF	RXIF	TXIF	STOIF	TXING	SLACKI	SLACKO	0000,0000	
PWMA_IER	PWMA 中断使能寄存器	FEC4H	BIE	TIE	COMIE	CC4IE	CC3IE	CC2IE	CC1IE	UIE	0000,0000	
PWMA_SR1	PWMA 状态寄存器 1	FEC5H	BIF	TIF	COMIF	CC4IF	CC3IF	CC2IF	CC1IF	UIF	0000,0000	
PWMA_SR2	PWMA 状态寄存器 2	FEC6H	-	-	-	CC4OF	CC3OF	CC2OF	CC1OF	-	xxx0,000x	
PWMB_IER	PWMB 中断使能寄存器	FEE4H	BIE	TIE	COMIE	CC8IE	CC7IE	CC6IE	CC5IE	UIE	0000,0000	
PWMB_SR1	PWMB 状态寄存器 1	FEE5H	BIF	TIF	COMIF	CC8IF	CC7IF	CC6IF	CC5IF	UIF	0000,0000	
PWMB_SR2	PWMB 状态寄存器 2	FEE6H	-	-	-	CC8OF	CC7OF	CC6OF	CC5OF	-	xxx0,000x	
POINTE	PO 口中断使能寄存器	FD00H	P07INTE	P06INTE	P05INTE	P04INTE	P03INTE	P02INTE	P01INTE	P00INTE	0000,0000	
P1INTE	P1 口中断使能寄存器	FD01H	P17INTE	P16INTE	P15INTE	P14INTE	P13INTE	P12INTE	P11INTE	P10INTE	0000,0000	

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
P2INTE	P2 口中断使能寄存器	FD02H	P27INTE	P26INTE	P25INTE	P24INTE	P23INTE	P22INTE	P21INTE	P20INTE	0000,0000	
P3INTE	P3 口中断使能寄存器	FD03H	P37INTE	P36INTE	P35INTE	P34INTE	P33INTE	P32INTE	P31INTE	P30INTE	0000,0000	
P4INTE	P4 口中断使能寄存器	FD04H	P47INTE	P46INTE	P45INTE	P44INTE	P43INTE	P42INTE	P41INTE	P40INTE	0000,0000	
P5INTE	P5 口中断使能寄存器	FD05H	-	-	P55INTE	P54INTE	P53INTE	P52INTE	P51INTE	P50INTE	xx00,0000	
P6INTE	P6 口中断使能寄存器	FD06H	P67INTE	P66INTE	P65INTE	P64INTE	P63INTE	P62INTE	P61INTE	P60INTE	0000,0000	
P7INTE	P7 口中断使能寄存器	FD07H	P77INTE	P76INTE	P75INTE	P74INTE	P73INTE	P72INTE	P71INTE	P70INTE	0000,0000	
POINTF	PO 口中断标志寄存器	FD10H	P07INTF	P06INTF	P05INTF	P04INTF	P03INTF	P02INTF	P01INTF	P00INTF	0000,0000	
P1INTF	P1 口中断标志寄存器	FD11H	P17INTF	P16INTF	P15INTF	P14INTF	P13INTF	P12INTF	P11INTF	P10INTF	0000,0000	
P2INTF	P2 口中断标志寄存器	FD12H	P27INTF	P26INTF	P25INTF	P24INTF	P23INTF	P22INTF	P21INTF	P20INTF	0000,0000	
P3INTF	P3 口中断标志寄存器	FD13H	P37INTF	P36INTF	P35INTF	P34INTF	P33INTF	P32INTF	P31INTF	P30INTF	0000,0000	
P4INTF	P4 口中断标志寄存器	FD14H	P47INTF	P46INTF	P45INTF	P44INTF	P43INTF	P42INTF	P41INTF	P40INTF	0000,0000	
P5INTF	P5 口中断标志寄存器	FD15H	-	-	P55INTF	P54INTF	P53INTF	P52INTF	P51INTF	P50INTF	xx00,0000	
P6INTF	P6 口中断标志寄存器	FD16H	P67INTF	P66INTF	P65INTF	P64INTF	P63INTF	P62INTF	P61INTF	P60INTF	0000,0000	
P7INTF	P7 口中断标志寄存器	FD17H	P77INTF	P76INTF	P75INTF	P74INTF	P73INTF	P72INTF	P71INTF	P70INTF	0000,0000	
PINIPL	I/O 口中断优先级低寄存器	FD60H	P7IP	P6IP	P5IP	P4IP	P3IP	P2IP	P1IP	POIP	0000,0000	
PINIPH	I/O 口中断优先级高寄存器	FD61H	P7IPH	P6IPH	P5IPH	P4IPH	P3IPH	P2IPH	P1IPH	POIPH	0000,0000	
UR1TOCR	串口 1 接收超时控制寄存器	FD70H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
URITOSR	串口 1 接收超时状态寄存器	FD71H	CTOIF	-	-	-	-	-	-	TOIF	0xxx,xxx0	
UR2TOCR	串口 2 接收超时控制寄存器	FD74H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
UR2TOSR	串口 2 接收超时状态寄存器	FD75H	CTOIF	-	-	-	-	-	-	TOIF	0xxx,xxx0	
SPITOCSR	SPI 从机超时控制寄存器	FD80H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
SPITOSR	SPI 从机超时状态寄存器	FD81H	CTOIF	-	-	-	-	-	-	TOIF	0xxx,xxx0	
I2CTOCR	I2C 从机超时控制寄存器	FD84H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
I2CTOSR	I2C 从机超时状态寄存器	FD85H	CTOIF	-	-	-	-	-	-	TOIF	0xxx,xxx0	
DMA_M2M_CFG	M2M_DMA 配置寄存器	FA00H	M2MIE	-	TXACO	RXACO	M2MIP[1:0]		M2MBAP[1:0]		0x00,0000	
DMA_ADC_CFG	ADC_DMA 配置寄存器	FA10H	ADCIE	-	-	-	ADCMIP[1:0]		ADCBAP[1:0]		0xxx,0000	
DMA_SPI_CFG	SPI_DMA 配置寄存器	FA20H	SPIIE	ACT_TX	ACT_RX	-	SPIIP[1:0]		SPIBAP[1:0]		000x,0000	
DMA_UR1T_CFG	UR1T_DMA 配置寄存器	FA30H	UR1TIE	-	-	-	UR1TIP[1:0]		UR1TBAP[1:0]		0xxx,0000	
DMA_UR1R_CFG	UR1R_DMA 配置寄存器	FA38H	UR1RIE	-	-	-	UR1RIP[1:0]		UR1RBAP[1:0]		0xxx,0000	
DMA_UR2T_CFG	UR2T_DMA 配置寄存器	FA40H	UR2TIE	-	-	-	UR2TIP[1:0]		UR2TBAP[1:0]		0xxx,0000	
DMA_UR2R_CFG	UR2R_DMA 配置寄存器	FA48H	UR2RIE	-	-	-	UR2RIP[1:0]		UR2RBAP[1:0]		0xxx,0000	
DMA_UR3T_CFG	UR3T_DMA 配置寄存器	FA50H	UR3TIE	-	-	-	UR3TIP[1:0]		UR3TBAP[1:0]		0xxx,0000	
DMA_UR3R_CFG	UR3R_DMA 配置寄存器	FA58H	UR3RIE	-	-	-	UR3RIP[1:0]		UR3RBAP[1:0]		0xxx,0000	
DMA_UR4T_CFG	UR4T_DMA 配置寄存器	FA60H	UR4TIE	-	-	-	UR4TIP[1:0]		UR4TBAP[1:0]		0xxx,0000	
DMA_UR4R_CFG	UR4R_DMA 配置寄存器	FA68H	UR4RIE	-	-	-	UR4RIP[1:0]		UR4RBAP[1:0]		0xxx,0000	
DMA_LCM_CFG	LCM_DMA 配置寄存器	FA70H	LCMIE	-	-	-	LCMIP[1:0]		LCMBAP[1:0]		0xxx,0000	
DMA_M2M_STA	M2M_DMA 状态寄存器	FA02H	-	-	-	-	-	-	-	M2MIF	xxxx,xxx0	
DMA_ADC_STA	ADC_DMA 状态寄存器	FA12H	-	-	-	-	-	-	-	ADCIF	xxxx,xxx0	
DMA_SPL_STA	SPI_DMA 状态寄存器	FA22H	-	-	-	-	-	-	TXOVW	RXLOSS	SPIIF	xxxx,x000
DMA_UR1T_STA	UR1T_DMA 状态寄存器	FA32H	-	-	-	-	-	-	TXOVW	-	UR1TIF	xxxx,x0x0
DMA_UR1R_STA	UR1R_DMA 状态寄存器	FA3AH	-	-	-	-	-	-	-	RXLOSS	UR1RIF	xxxx,xx00
DMA_UR2T_STA	UR2T_DMA 状态寄存器	FA42H	-	-	-	-	-	-	TXOVW	-	UR2TIF	xxxx,x0x0

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
DMA_UR2R_STA	UR2R_DMA 状态寄存器	FA4AH	-	-	-	-	-	-	RXLOSS	UR2RIF	xxxx,xx00
DMA_UR3T_STA	UR3T_DMA 状态寄存器	FA52H	-	-	-	-	-	TXOVW	-	UR3TIF	xxxx,x0x0
DMA_UR3R_STA	UR3R_DMA 状态寄存器	FA5AH	-	-	-	-	-	-	RXLOSS	UR3RIF	xxxx,xx00
DMA_UR4T_STA	UR4T_DMA 状态寄存器	FA62H	-	-	-	-	-	TXOVW	-	UR4TIF	xxxx,x0x0
DMA_UR4R_STA	UR4R_DMA 状态寄存器	FA6AH	-	-	-	-	-	-	RXLOSS	UR4RIF	xxxx,xx00
DMA_LCM_STA	LCM_DMA 状态寄存器	FA72H	-	-	-	-	-	-	TXOVW	LCMIF	xxxx,xx00

14.5.1 中断使能寄存器（中断允许位）

IE (中断使能寄存器)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	EA	ELVD	EADC	ES	ET1	EX1	ET0	EX0

EA: 总中断允许控制位。EA 的作用是使中断允许形成多级控制。即各中断源首先受 EA 控制;其次还受各中断源自己的中断允许控制位控制。

0: CPU 屏蔽所有的中断申请

1: CPU 开放中断

ELVD: 低压检测中断允许位。

0: 禁止低压检测中断

1: 允许低压检测中断

EADC: A/D 转换中断允许位。

0: 禁止 A/D 转换中断

1: 允许 A/D 转换中断

ES: 串行口 1 中断允许位。

0: 禁止串行口 1 中断

1: 允许串行口 1 中断

ET1: 定时/计数器 T1 的溢出中断允许位。

0: 禁止 T1 中断

1: 允许 T1 中断

EX1: 外部中断 1 中断允许位。

0: 禁止 INT1 中断

1: 允许 INT1 中断

ET0: 定时/计数器 T0 的溢出中断允许位。

0: 禁止 T0 中断

1: 允许 T0 中断

EX0: 外部中断 0 中断允许位。

0: 禁止 INT0 中断

1: 允许 INT0 中断

IE2 (中断使能寄存器 2)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IE2	AFH	EUSB ETKSU	ET4	ET3	ES4	ES3	ET2	ESPI	ES2

EUSB: USB 中断允许位。

0: 禁止 USB 中断

1: 允许 USB 中断

ETKSU: 触摸按键中断允许位。

0: 禁止触摸按键中断

1: 允许触摸按键中断

ET4: 定时/计数器 T4 的溢出中断允许位。

0: 禁止 T4 中断

1: 允许 T4 中断

ET3: 定时/计数器 T3 的溢出中断允许位。

0: 禁止 T3 中断

1: 允许 T3 中断

ES4: 串行口 4 中断允许位。

0: 禁止串行口 4 中断

1: 允许串行口 4 中断

ES3: 串行口 3 中断允许位。

0: 禁止串行口 3 中断

1: 允许串行口 3 中断

ET2: 定时/计数器 T2 的溢出中断允许位。

0: 禁止 T2 中断

1: 允许 T2 中断

ESPI: SPI 中断允许位。

0: 禁止 SPI 中断

1: 允许 SPI 中断

ES2: 串行口 2 中断允许位。

0: 禁止串行口 2 中断

1: 允许串行口 2 中断

INTCLKO (外部中断与时钟输出控制寄存器)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
INTCLKO	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO

EX4: 外部中断 4 中断允许位。

0: 禁止 INT4 中断

1: 允许 INT4 中断

EX3: 外部中断 3 中断允许位。

0: 禁止 INT3 中断

1: 允许 INT3 中断

EX2: 外部中断 2 中断允许位。

0: 禁止 INT2 中断

1: 允许 INT2 中断

CMPCR1 (比较器控制寄存器 1)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPCR1	E6H	CMPEN	CMPIF	PIE	NIE	PIS	NIS	CMPOE	CMPRES

PIE: 比较器上升沿中断允许位。

0: 禁止比较器上升沿中断

1: 允许比较器上升沿中断

NIE: 比较器下降沿中断允许位。

0: 禁止比较器下降沿中断

1: 允许比较器下降沿中断

T11 控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T11CR	FE78H	T11R	T11_C/T	T11CLKO	T11x12	T11CS[1:0]	ET11I	T11IF	

ET11I: T11中断允许位。

0: 禁止 T11 中断

1: 允许 T11 中断

I2C 控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CMSCR	FE81H	EMSI	-	-	-	MSCMD[3:0]			
I2CSLCR	FE83H	-	ESTAI	ERXI	ETXI	ESTOI	-	-	SLRST

EMSI: I²C 主机模式中断允许位。

- 0: 禁止 I²C 主机模式中断
- 1: 允许 I²C 主机模式中断

ESTAI: I²C 从机接收 START 事件中断允许位。

- 0: 禁止 I²C 从机接收 START 事件中断
- 1: 允许 I²C 从机接收 START 事件中断

ERXI: I²C 从机接收数据完成事件中断允许位。

- 0: 禁止 I²C 从机接收数据完成事件中断
- 1: 允许 I²C 从机接收数据完成事件中断

ETXI: I²C 从机发送数据完成事件中断允许位。

- 0: 禁止 I²C 从机发送数据完成事件中断
- 1: 允许 I²C 从机发送数据完成事件中断

ESTOI: I²C 从机接收 STOP 事件中断允许位。

- 0: 禁止 I²C 从机接收 STOP 事件中断
- 1: 允许 I²C 从机接收 STOP 事件中断

PWMA 中断使能寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PWMA_IER	FEC4H	BIE	TIE	COMIE	CC4IE	CC3IE	CC2IE	CC1IE	UIE

BIE: PWMA 刹车中断允许位。

- 0: 禁止 PWMA 刹车中断
- 1: 允许 PWMA 刹车中断

TIE: PWMA 触发中断允许位。

- 0: 禁止 PWMA 触发中断
- 1: 允许 PWMA 触发中断

COMIE: PWMA 比较中断允许位。

- 0: 禁止 PWMA 比较中断
- 1: 允许 PWMA 比较中断

CC4IE: PWMA 捕获比较通道4中断允许位。

- 0: 禁止 PWMA 捕获比较通道 4 中断
- 1: 允许 PWMA 捕获比较通道 4 中断

CC3IE: PWMA 捕获比较通道3中断允许位。

- 0: 禁止 PWMA 捕获比较通道 3 中断
- 1: 允许 PWMA 捕获比较通道 3 中断

CC2IE: PWMA 捕获比较通道2中断允许位。

- 0: 禁止 PWMA 捕获比较通道 2 中断
- 1: 允许 PWMA 捕获比较通道 2 中断

CC1IE: PWMA 捕获比较通道1中断允许位。

- 0: 禁止 PWMA 捕获比较通道 1 中断
- 1: 允许 PWMA 捕获比较通道 1 中断

UIE: PWMA更新中断允许位。

- 0: 禁止 PWMA 更新中断
- 1: 允许 PWMA 更新中断

PWMB 中断使能寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PWMB_IER	FEE4H	BIE	TIE	COMIE	CC8IE	CC7IE	CC6IE	CC5IE	UIE

BIE: PWMB刹车中断允许位。

- 0: 禁止 PWMB 刹车中断
- 1: 允许 PWMB 刹车中断

TIE: PWMB触发中断允许位。

- 0: 禁止 PWMB 触发中断
- 1: 允许 PWMB 触发中断

COMIE: PWMB比较中断允许位。

- 0: 禁止 PWMB 比较中断
- 1: 允许 PWMB 比较中断

CC8IE: PWMB捕获比较通道8中断允许位。

- 0: 禁止 PWMB 捕获比较通道 8 中断
- 1: 允许 PWMB 捕获比较通道 8 中断

CC7IE: PWMB捕获比较通道7中断允许位。

- 0: 禁止 PWMB 捕获比较通道 7 中断
- 1: 允许 PWMB 捕获比较通道 7 中断

CC6IE: PWMB捕获比较通道6中断允许位。

- 0: 禁止 PWMB 捕获比较通道 6 中断
- 1: 允许 PWMB 捕获比较通道 6 中断

CC5IE: PWMB捕获比较通道5中断允许位。

- 0: 禁止 PWMB 捕获比较通道 5 中断
- 1: 允许 PWMB 捕获比较通道 5 中断

UIE: PWMB更新中断允许位。

- 0: 禁止 PWMB 更新中断
- 1: 允许 PWMB 更新中断

端口中断使能寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0INTE	FD00H	P07INTE	P06INTE	P05INTE	P04INTE	P03INTE	P02INTE	P01INTE	P00INTE
P1INTE	FD01H	P17INTE	P16INTE	P15INTE	P14INTE	P13INTE	P12INTE	P11INTE	P10INTE
P2INTE	FD02H	P27INTE	P26INTE	P25INTE	P24INTE	P23INTE	P22INTE	P21INTE	P20INTE
P3INTE	FD03H	P37INTE	P36INTE	P35INTE	P34INTE	P33INTE	P32INTE	P31INTE	P30INTE
P4INTE	FD04H	P47INTE	P46INTE	P45INTE	P44INTE	P43INTE	P42INTE	P41INTE	P40INTE
P5INTE	FD05H	-	-	P55INTE	P54INTE	P53INTE	P52INTE	P51INTE	P50INTE
P6INTE	FD06H	P67INTE	P66INTE	P65INTE	P64INTE	P63INTE	P62INTE	P61INTE	P60INTE
P7INTE	FD07H	P77INTE	P76INTE	P75INTE	P74INTE	P73INTE	P72INTE	P71INTE	P70INTE

PnINTE.x: 端口中断使能控制位 (n=0~7, x=0~7)

0: 关闭 Pn.x 口中断功能

1: 使能 Pn.x 口中断功能

串口 1 超时控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR1TOCR	FD70H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTOI: 串口1接收超时中断允许位。

0: 禁止串口 1 接收超时中断

1: 允许串口 1 接收超时中断

串口 2 超时控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR2TOCR	FD74H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTOI: 串口2接收超时中断允许位。

0: 禁止串口 2 接收超时中断

SPI 超时控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPITOOCR	FD80H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTOI: SPI从机超时中断允许位。

0: 禁止 SPI 从机超时中断

1: 允许 SPI 从机超时中断

I2C 超时控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTOCR	FD84H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTOI: I2C从机超时中断允许位。

0: 禁止 I2C 从机超时中断

1: 允许 I2C 从机超时中断

LCM 接口配置寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
LCMIFCFG	FE50H	LCMIFIE	-	LCMIFIP[1:0]	LCMIFDPS[1:0]	D16_D8	M68_I80		

LCMIFIE: LCM接口中断允许位。

0: 禁止 LCM 接口中断

1: 允许 LCM 接口中断

DMA 中断使能寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
DMA_M2M_CFG	FA00H	M2MIE	-	TXACO	RXACO	M2MIP[1:0]	M2MBAP[1:0]		
DMA_ADC_CFG	FA10H	ADCIE	-	-	-	ADCMIP[1:0]	ADCBAP[1:0]		
DMA_SPI_CFG	FA20H	SPIIE	ACT_TX	ACT_RX	-	SPIIP[1:0]	SPIBAP[1:0]		
DMA_UR1T_CFG	FA30H	UR1TIE	-	-	-	UR1TIP[1:0]	UR1TBAP[1:0]		
DMA_UR1R_CFG	FA38H	UR1RIE	-	-	-	UR1RIP[1:0]	UR1RBAP[1:0]		
DMA_UR2T_CFG	FA40H	UR2TIE	-	-	-	UR2TIP[1:0]	UR2TBAP[1:0]		
DMA_UR2R_CFG	FA48H	UR2RIE	-	-	-	UR2RIP[1:0]	UR2RBAP[1:0]		
DMA_UR3T_CFG	FA50H	UR3TIE	-	-	-	UR3TIP[1:0]	UR3TBAP[1:0]		
DMA_UR3R_CFG	FA58H	UR3RIE	-	-	-	UR3RIP[1:0]	UR3RBAP[1:0]		
DMA_UR4R_CFG	FA60H	UR4TIE	-	-	-	UR4TIP[1:0]	UR4TBAP[1:0]		
DMA_LCM_CFG	FA70H	LCMIE	-	-	-	LCMIP[1:0]	LCMBAP[1:0]		

M2MIE: DMA_M2M (存储器到存储器DMA) 中断允许位。

0: 禁止 DMA_M2M 中断

1: 允许 DMA_M2M 中断

ADCIE: DMA_ADC (ADC DMA) 中断允许位。

0: 禁止 DMA_ADC 中断

1: 允许 DMA_ADC 中断

SPIIE: DMA_SPI (SPI DMA) 中断允许位。

0: 禁止 DMA_SPI 中断

1: 允许 DMA_SPI 中断

UR1TIE: DMA_UR1T (串口1发送DMA) 中断允许位。

0: 禁止 DMA_UR1T 中断

1: 允许 DMA_UR1T 中断

UR1RIE: DMA_UR1R (串口1接收DMA) 中断允许位。

0: 禁止 DMA_UR1R 中断

1: 允许 DMA_UR1R 中断

UR2TIE: DMA_UR2T (串口2发送DMA) 中断允许位。

0: 禁止 DMA_UR2T 中断

1: 允许 DMA_UR2T 中断

UR2RIE: DMA_UR2R (串口2接收DMA) 中断允许位。

0: 禁止 DMA_UR2R 中断

1: 允许 DMA_UR2R 中断

UR3TIE: DMA_UR3T (串口3发送DMA) 中断允许位。

0: 禁止 DMA_UR3T 中断

1: 允许 DMA_UR3T 中断

UR3RIE: DMA_UR3R (串口3接收DMA) 中断允许位。

0: 禁止 DMA_UR3R 中断

1: 允许 DMA_UR3R 中断

UR4TIE: DMA_UR4T (串口4发送DMA) 中断允许位。

0: 禁止 DMA_UR4T 中断

1: 允许 DMA_UR4T 中断

UR4RIE: DMA_UR4R (串口4接收DMA) 中断允许位。

0: 禁止 DMA_UR4R 中断

1: 允许 DMA_UR4R 中断

LCMIE: DMA_LCM (LCM接口DMA) 中断允许位。

0: 禁止 DMA_LCM 中断

1: 允许 DMA_LCM 中断

14.5.2 中断请求寄存器（中断标志位）

定时器控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TCON	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1: 定时器1溢出中断标志。中断服务程序中，硬件自动清零。

TF0: 定时器0溢出中断标志。中断服务程序中，硬件自动清零。

IE1: 外部中断1中断请求标志。中断服务程序中，硬件自动清零。

IE0: 外部中断0中断请求标志。中断服务程序中，硬件自动清零。

中断标志辅助寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
AUXINTIF	EFH	-	INT4IF	INT3IF	INT2IF	-	T4IF	T3IF	T2IF

INT4IF: 外部中断4中断请求标志。中断服务程序中硬件自动清零。

INT3IF: 外部中断3中断请求标志。中断服务程序中硬件自动清零。

INT2IF: 外部中断2中断请求标志。中断服务程序中硬件自动清零。

T4IF: 定时器4溢出中断标志。中断服务程序中硬件自动清零。

T3IF: 定时器3溢出中断标志。中断服务程序中硬件自动清零。

T2IF: 定时器2溢出中断标志。中断服务程序中硬件自动清零。

(注意: 对于STC8H1K08系列、STC8H1K28系列、STC8H3K64S2系列和STC8H3K64S4系列这4个系列,

T2IF/T3IF/T4IF这三个标志位为只写寄存器, 不可读取)

串口控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SCON	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI
S2CON	9AH	S2SM0	-	S2SM2	S2REN	S2TB8	S2RB8	S2TI	S2RI
S3CON	ACH	S3SM0	S3ST3	S3SM2	S3REN	S3TB8	S3RB8	S3TI	S3RI
S4CON	84H	S4SM0	S4ST4	S4SM2	S4REN	S4TB8	S4RB8	S4TI	S4RI

TI: 串口1发送完成中断请求标志。需要软件清零。

RI: 串口1接收完成中断请求标志。需要软件清零。

S2TI: 串口2发送完成中断请求标志。需要软件清零。

S2RI: 串口2接收完成中断请求标志。需要软件清零。

S3TI: 串口3发送完成中断请求标志。需要软件清零。

S3RI: 串口3接收完成中断请求标志。需要软件清零。

S4TI: 串口4发送完成中断请求标志。需要软件清零。

S4RI: 串口4接收完成中断请求标志。需要软件清零。

电源管理寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	SMOD	SMODO	LVDF	POF	GF1	GF0	PD	IDL

LVDF: 低压检测中断请求标志。需要软件清零。

ADC 控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADC_CONTR	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]			

ADC_FLAG: ADC转换完成中断请求标志。需要软件清零。

SPI 状态寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPSTAT	CDH	SPIF	WCOL	-	-	-	-	-	-

SPIF: SPI数据传输完成中断请求标志。需要软件清零。

比较器控制寄存器 1

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPPCR1	E6H	CMPEN	CMPIF	PIE	NIE	PIS	NIS	CMPOE	CMPRES

CMPIF: 比较器中断请求标志。需要软件清零。

T11 控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T11CR	FE78H	T11R	T11_C/T	T11CLKO	T11x12	T11CS[1:0]	ET11I	T11IF	

T11IF: 定时器T11溢出中断标志。中断服务程序中，硬件自动清零。

I2C 状态寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CMSST	FE82H	MSBUSY	MSIF	-	-	-	-	MSACKI	MSACKO
I2CSLST	FE84H	SLBUSY	STAIF	RXIF	TXIF	STOIF	TXING	SLACKI	SLACKO

MSIF: I²C主机模式中断请求标志。需要软件清零。

ESTAI: I²C从机接收START事件中断请求标志。需要软件清零。

ERXI: I²C从机接收数据完成事件中断请求标志。需要软件清零。

ETXI: I²C从机发送数据完成事件中断请求标志。需要软件清零。

ESTOI: I²C从机接收STOP事件中断请求标志。需要软件清零。

PWMA 状态寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PWMA_SR1	FEC5H	BIF	TIF	COMIF	CC4IF	CC3IF	CC2IF	CC1IF	UIF
PWMA_SR2	FEC6H	-	-	-	CC4OF	CC3OF	CC2OF	CC1OF	-

BIF: PWMA刹车中断请求标志。需要软件清零。

TIF: PWMA触发中断请求标志。需要软件清零。

COMIF: PWMA比较中断请求标志。需要软件清零。

CC4IF: PWMA通道4发生捕获比较中断请求标志。需要软件清零。

CC3IF: PWMA通道3发生捕获比较中断请求标志。需要软件清零。

CC2IF: PWMA通道2发生捕获比较中断请求标志。需要软件清零。

CC1IF: PWMA通道1发生捕获比较中断请求标志。需要软件清零。

UIF: PWMA更新中断请求标志。需要软件清零。

CC4OF: PWMA通道4发生重复捕获中断请求标志。需要软件清零。

CC3OF: PWMA通道3发生重复捕获中断请求标志。需要软件清零。

CC2OF: PWMA通道2发生重复捕获中断请求标志。需要软件清零。

CC1OF: PWMA通道1发生重复捕获中断请求标志。需要软件清零。

PWMB 状态寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PWMB_SR1	FEE5H	BIF	TIF	COMIF	CC8IF	CC7IF	CC6IF	CC5IF	UIF
PWMB_SR2	FEE6H	-	-	-	CC8OF	CC7OF	CC6OF	CC5OF	-

BIF: PWMB刹车中断请求标志。需要软件清零。

TIF: PWMB触发中断请求标志。需要软件清零。

COMIF: PWMB比较中断请求标志。需要软件清零。

CC8IF: PWMB通道8发生捕获比较中断请求标志。需要软件清零。

CC7IF: PWMB通道7发生捕获比较中断请求标志。需要软件清零。

CC6IF: PWMB通道6发生捕获比较中断请求标志。需要软件清零。

CC5IF: PWMB通道5发生捕获比较中断请求标志。需要软件清零。

UIF: PWMB更新中断请求标志。需要软件清零。

CC8OF: PWMB通道8发生重复捕获中断请求标志。需要软件清零。

CC7OF: PWMB通道7发生重复捕获中断请求标志。需要软件清零。

CC6OF: PWMB通道6发生重复捕获中断请求标志。需要软件清零。

CC5OF: PWMB通道5发生重复捕获中断请求标志。需要软件清零。

端口中断标志寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0INTF	FD10H	P07INTF	P06INTF	P05INTF	P04INTF	P03INTF	P02INTF	P01INTF	P00INTF
P1INTF	FD11H	P17INTF	P16INTF	P15INTF	P14INTF	P13INTF	P12INTF	P11INTF	P10INTF
P2INTF	FD12H	P27INTF	P26INTF	P25INTF	P24INTF	P23INTF	P22INTF	P21INTF	P20INTF
P3INTF	FD13H	P37INTF	P36INTF	P35INTF	P34INTF	P33INTF	P32INTF	P31INTF	P30INTF
P4INTF	FD14H	P47INTF	P46INTF	P45INTF	P44INTF	P43INTF	P42INTF	P41INTF	P40INTF
P5INTF	FD15H	-	-	P55INTF	P54INTF	P53INTF	P52INTF	P51INTF	P50INTF
P6INTF	FD16H	P67INTF	P66INTF	P65INTF	P64INTF	P63INTF	P62INTF	P61INTF	P60INTF
P7INTF	FD17H	P77INTF	P76INTF	P75INTF	P74INTF	P73INTF	P72INTF	P71INTF	P70INTF

PnINTF.x: 端口中断请求标志位 (n=0~7, x=0~7)

0: Pn.x 口没有中断请求

1: Pn.x 口有中断请求, 若使能中断, 则会进入中断服务程序。**标志位需软件清 0。****串口 1 超时状态寄存器**

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR1TOSR	FD71H	CTOIF	-	-	-	-	-	-	TOIF

TOIF: 串口1超时中断请求标志。需要软件清零。**(只读寄存器, 需向CTOIF写1清TOIF)****串口 2 超时状态寄存器**

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR2TOSR	FD75H	CTOIF	-	-	-	-	-	-	TOIF

TOIF: 串口2超时中断请求标志。需要软件清零。**(只读寄存器, 需向CTOIF写1清TOIF)****SPI 超时状态寄存器**

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPITOSR	FD81H	CTOIF	-	-	-	-	-	-	TOIF

TOIF: SPI超时中断请求标志。需要软件清零。**(只读寄存器, 需向CTOIF写1清TOIF)****I2C 超时状态寄存器**

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTOSR	FD75H	CTOIF	-	-	-	-	-	-	TOIF

TOIF: I2C超时中断请求标志。需要软件清零。**(只读寄存器, 需向CTOIF写1清TOIF)**

LCM 接口状态寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
LCMIFSTA	FE53H	-	-	-	-	-	-	-	LCMIFIF

LCMIFIF: LCM接口中断请求标志。需要软件清零。

DMA 中断标志寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
DMA_M2M_STA	FA02H	-	-	-	-	-	-	-	M2MIF
DMA_ADC_STA	FA12H	-	-	-	-	-	-	-	ADCIF
DMA_SPI_STA	FA22H	-	-	-	-	-	TXOVW	RXLOSS	SPIIF
DMA_UR1T_STA	FA32H	-	-	-	-	-	TXOVW	-	UR1TIF
DMA_UR1R_STA	FA3AH	-	-	-	-	-	-	RXLOSS	UR1RIF
DMA_UR2T_STA	FA42H	-	-	-	-	-	TXOVW	-	UR2TIF
DMA_UR2R_STA	FA4AH	-	-	-	-	-	-	RXLOSS	UR2RIF
DMA_UR3T_STA	FA52H	-	-	-	-	-	TXOVW	-	UR3TIF
DMA_UR3R_STA	FA5AH	-	-	-	-	-	-	RXLOSS	UR3RIF
DMA_UR4T_STA	FA62H	-	-	-	-	-	TXOVW	-	UR4TIF
DMA_UR4R_STA	FA6AH	-	-	-	-	-	-	RXLOSS	UR4RIF
DMA_LCM_STA	FA72H	-	-	-	-	-	-	TXOVW	LCMIF

M2MIF: DMA_M2M (存储器到存储器DMA) 中断请求标志。需要软件清零。

ADCIF: DMA_ADC (ADC DMA) 中断请求标志。需要软件清零。

SPIIF: DMA_SPI (SPI DMA) 中断请求标志。需要软件清零。。

UR1TIF: DMA_UR1T (串口1发送DMA) 中断请求标志。需要软件清零。

UR1RIF: DMA_UR1R (串口1接收DMA) 中断请求标志。需要软件清零。

UR2TIF: DMA_UR2T (串口2发送DMA) 中断请求标志。需要软件清零。

UR2RIF: DMA_UR2R (串口2接收DMA) 中断请求标志。需要软件清零。

UR3TIF: DMA_UR3T (串口3发送DMA) 中断请求标志。需要软件清零。

UR3RIF: DMA_UR3R (串口3接收DMA) 中断请求标志。需要软件清零。

UR4TIF: DMA_UR4T (串口4发送DMA) 中断请求标志。需要软件清零。

UR4RIF: DMA_UR4R (串口4接收DMA) 中断请求标志。需要软件清零。

LCMIF: DMA_LCM (LCM接口DMA) 中断请求标志。需要软件清零。

14.5.3 中断优先级寄存器

中断优先级控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IP	B8H	-	PLVD	PADC	PS	PT1	PX1	PT0	PX0
IPH	B7H	-	PLVDH	PADCH	PSH	PT1H	PX1H	PT0H	PX0H
IP2	B5H	PUSB PTKSU	PI2C	PCMP	PX4	PPWMB	PPWMA	PSPI	PS2
IP2H	B6H	PUSBH PTKSUH	PI2CH	PCMPh	PX4H	PPWMBH	PPWMAH	PSPIH	PS2H
IP3	DFH	-	-	-	-	-	PRTC	PS4	PS3
IP3H	EEH	-	-	-	-	-	PRTCH	PS4H	PS3H

PX0H,PX0: 外部中断0中断优先级控制位

- 00: INT0 中断优先级为 0 级 (最低级)
- 01: INT0 中断优先级为 1 级 (较低级)
- 10: INT0 中断优先级为 2 级 (较高级)
- 11: INT0 中断优先级为 3 级 (最高级)

PT0H,PT0: 定时器0中断优先级控制位

- 00: 定时器 0 中断优先级为 0 级 (最低级)
- 01: 定时器 0 中断优先级为 1 级 (较低级)
- 10: 定时器 0 中断优先级为 2 级 (较高级)
- 11: 定时器 0 中断优先级为 3 级 (最高级)

PX1H,PX1: 外部中断1中断优先级控制位

- 00: INT1 中断优先级为 0 级 (最低级)
- 01: INT1 中断优先级为 1 级 (较低级)
- 10: INT1 中断优先级为 2 级 (较高级)
- 11: INT1 中断优先级为 3 级 (最高级)

PT1H,PT1: 定时器1中断优先级控制位

- 00: 定时器 1 中断优先级为 0 级 (最低级)
- 01: 定时器 1 中断优先级为 1 级 (较低级)
- 10: 定时器 1 中断优先级为 2 级 (较高级)
- 11: 定时器 1 中断优先级为 3 级 (最高级)

PSH,PS: 串口1中断优先级控制位

- 00: 串口 1 中断优先级为 0 级 (最低级)
- 01: 串口 1 中断优先级为 1 级 (较低级)
- 10: 串口 1 中断优先级为 2 级 (较高级)
- 11: 串口 1 中断优先级为 3 级 (最高级)

PADCH,PADC: ADC中断优先级控制位

- 00: ADC 中断优先级为 0 级 (最低级)
- 01: ADC 中断优先级为 1 级 (较低级)
- 10: ADC 中断优先级为 2 级 (较高级)
- 11: ADC 中断优先级为 3 级 (最高级)

PLVDH,PLVD: 低压检测中断优先级控制位

- 00: LVD 中断优先级为 0 级 (最低级)
- 01: LVD 中断优先级为 1 级 (较低级)
- 10: LVD 中断优先级为 2 级 (较高级)
- 11: LVD 中断优先级为 3 级 (最高级)

PS2H,PS2: 串口2中断优先级控制位

- 00: 串口 2 中断优先级为 0 级 (最低级)
- 01: 串口 2 中断优先级为 1 级 (较低级)
- 10: 串口 2 中断优先级为 2 级 (较高级)
- 11: 串口 2 中断优先级为 3 级 (最高级)

PS3H,PS3: 串口3中断优先级控制位

- 00: 串口 3 中断优先级为 0 级 (最低级)
- 01: 串口 3 中断优先级为 1 级 (较低级)
- 10: 串口 3 中断优先级为 2 级 (较高级)
- 11: 串口 3 中断优先级为 3 级 (最高级)

PS4H,PS4: 串口4中断优先级控制位

- 00: 串口 4 中断优先级为 0 级 (最低级)
- 01: 串口 4 中断优先级为 1 级 (较低级)
- 10: 串口 4 中断优先级为 2 级 (较高级)
- 11: 串口 4 中断优先级为 3 级 (最高级)

PSPIH,PSPI: SPI中断优先级控制位

- 00: SPI 中断优先级为 0 级 (最低级)
- 01: SPI 中断优先级为 1 级 (较低级)
- 10: SPI 中断优先级为 2 级 (较高级)
- 11: SPI 中断优先级为 3 级 (最高级)

PPWMAH,PPWMA: 高级PWMA中断优先级控制位

- 00: 高级 PWMA 中断优先级为 0 级 (最低级)
- 01: 高级 PWMA 中断优先级为 1 级 (较低级)
- 10: 高级 PWMA 中断优先级为 2 级 (较高级)
- 11: 高级 PWMA 中断优先级为 3 级 (最高级)

PPWMBH,PPWMB: 高级PWMB中断优先级控制位

- 00: 高级 PWMB 中断优先级为 0 级 (最低级)
- 01: 高级 PWMB 中断优先级为 1 级 (较低级)
- 10: 高级 PWMB 中断优先级为 2 级 (较高级)
- 11: 高级 PWMB 中断优先级为 3 级 (最高级)

PX4H,PX4: 外部中断4中断优先级控制位

- 00: INT4 中断优先级为 0 级 (最低级)
- 01: INT4 中断优先级为 1 级 (较低级)
- 10: INT4 中断优先级为 2 级 (较高级)
- 11: INT4 中断优先级为 3 级 (最高级)

PCMPH,PCMP: 比较器中断优先级控制位

- 00: CMP 中断优先级为 0 级 (最低级)
- 01: CMP 中断优先级为 1 级 (较低级)
- 10: CMP 中断优先级为 2 级 (较高级)
- 11: CMP 中断优先级为 3 级 (最高级)

PI2CH,PI2C: I2C中断优先级控制位

- 00: I2C 中断优先级为 0 级 (最低级)
- 01: I2C 中断优先级为 1 级 (较低级)
- 10: I2C 中断优先级为 2 级 (较高级)
- 11: I2C 中断优先级为 3 级 (最高级)

PUSBH,PUSB: USB中断优先级控制位

- 00: USB 中断优先级为 0 级 (最低级)
- 01: USB 中断优先级为 1 级 (较低级)
- 10: USB 中断优先级为 2 级 (较高级)
- 11: USB 中断优先级为 3 级 (最高级)

PTKSUH,PTKSU: 触摸按键中断优先级控制位

- 00: 触摸按键中断优先级为 0 级 (最低级)
- 01: 触摸按键中断优先级为 1 级 (较低级)
- 10: 触摸按键中断优先级为 2 级 (较高级)
- 11: 触摸按键中断优先级为 3 级 (最高级)

PRTCH,PRTC: RTC中断优先级控制位

- 00: RTC 中断优先级为 0 级 (最低级)
- 01: RTC 中断优先级为 1 级 (较低级)
- 10: RTC 中断优先级为 2 级 (较高级)
- 11: RTC 中断优先级为 3 级 (最高级)

LCM 接口配置寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
LCMIFCFG	FE50H	LCMIFIE	-	LCMIFIP[1:0]	LCMIFDPS[1:0]	D16_D8	M68_I80		

LCMIFIP[1:0]: LCM接口中断优先级控制位

- 00: LCM 接口中断优先级为 0 级 (最低级)
- 01: LCM 接口中断优先级为 1 级 (较低级)
- 10: LCM 接口中断优先级为 2 级 (较高级)
- 11: LCM 接口中断优先级为 3 级 (最高级)

端口中断优先级控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PINIPL	FD60H	P7IP	P6IP	P5IP	P4IP	P3IP	P2IP	P1IP	P0IP
PINIPH	FD61H	P7IPH	P6IPH	P5IPH	P4IPH	P3IPH	P2IPH	P1IPH	P0IPH

P0IPH,P0IP: P0口中断优先级控制位

- 00: P0 口中断优先级为 0 级 (最低级)
- 01: P0 口中断优先级为 1 级 (较低级)
- 10: P0 口中断优先级为 2 级 (较高级)
- 11: P0 口中断优先级为 3 级 (最高级)

P1IPH,P1IP: P1口中断优先级控制位

- 00: P1 口中断优先级为 0 级 (最低级)
- 01: P1 口中断优先级为 1 级 (较低级)
- 10: P1 口中断优先级为 2 级 (较高级)
- 11: P1 口中断优先级为 3 级 (最高级)

P2IPH,P2IP: P2口中断优先级控制位

- 00: P2 口中断优先级为 0 级 (最低级)
- 01: P2 口中断优先级为 1 级 (较低级)
- 10: P2 口中断优先级为 2 级 (较高级)
- 11: P2 口中断优先级为 3 级 (最高级)

P3IPH,P3IP: P3口中断优先级控制位

- 00: P3 口中断优先级为 0 级 (最低级)
- 01: P3 口中断优先级为 1 级 (较低级)
- 10: P3 口中断优先级为 2 级 (较高级)
- 11: P3 口中断优先级为 3 级 (最高级)

P4IPH,P4IP: P4口中断优先级控制位

- 00: P4 口中断优先级为 0 级 (最低级)
- 01: P4 口中断优先级为 1 级 (较低级)
- 10: P4 口中断优先级为 2 级 (较高级)
- 11: P4 口中断优先级为 3 级 (最高级)

P5IPH,P5IP: P5口中断优先级控制位

- 00: P5 口中断优先级为 0 级 (最低级)
- 01: P5 口中断优先级为 1 级 (较低级)
- 10: P5 口中断优先级为 2 级 (较高级)
- 11: P5 口中断优先级为 3 级 (最高级)

P6IPH,P6IP: P6口中断优先级控制位

- 00: P6 口中断优先级为 0 级 (最低级)
- 01: P6 口中断优先级为 1 级 (较低级)
- 10: P6 口中断优先级为 2 级 (较高级)
- 11: P6 口中断优先级为 3 级 (最高级)

P7IPH,P7IP: P7口中断优先级控制位

- 00: P7 口中断优先级为 0 级 (最低级)
- 01: P7 口中断优先级为 1 级 (较低级)
- 10: P7 口中断优先级为 2 级 (较高级)

11: P7 口中断优先级为 3 级（最高级）

DMA 中断优先级控制寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
DMA_M2M_CFG	FA00H	M2MIE	-	TXACO	RXACO	M2MIP[1:0]		M2MBAP[1:0]	
DMA_ADC_CFG	FA10H	ADCIE	-	-	-	ADCMIP[1:0]		ADCBAP[1:0]	
DMA_SPI_CFG	FA20H	SPIIE	ACT_TX	ACT_RX	-	SPIIP[1:0]		SPIBAP[1:0]	
DMA_UR1T_CFG	FA30H	UR1TIE	-	-	-	UR1TIP[1:0]		UR1TBAP[1:0]	
DMA_UR1R_CFG	FA38H	UR1RIE	-	-	-	UR1RIP[1:0]		UR1RBAP[1:0]	
DMA_UR2T_CFG	FA40H	UR2TIE	-	-	-	UR2TIP[1:0]		UR2TBAP[1:0]	
DMA_UR2R_CFG	FA48H	UR2RIE	-	-	-	UR2RIP[1:0]		UR2RBAP[1:0]	
DMA_UR3T_CFG	FA50H	UR3TIE	-	-	-	UR3TIP[1:0]		UR3TBAP[1:0]	
DMA_UR3R_CFG	FA58H	UR3RIE	-	-	-	UR3RIP[1:0]		UR3RBAP[1:0]	
DMA_UR4T_CFG	FA60H	UR4TIE	-	-	-	UR4TIP[1:0]		UR4TBAP[1:0]	
DMA_UR4R_CFG	FA68H	UR4RIE	-	-	-	UR4RIP[1:0]		UR4RBAP[1:0]	
DMA_LCM_CFG	FA70H	LCMIE	-	-	-	LCMIP[1:0]		LCMBAP[1:0]	

M2MIP: DMA_M2M (存储器到存储器DMA) 中断优先级控制位

- 00: DMA_M2M 中断优先级为 0 级（最低级）
- 01: DMA_M2M 中断优先级为 1 级（较低级）
- 10: DMA_M2M 中断优先级为 2 级（较高级）
- 11: DMA_M2M 中断优先级为 3 级（最高级）

ADCIP: DMA_ADC (ADC DMA) 中断优先级控制位

- 00: DMA_ADC 中断优先级为 0 级（最低级）
- 01: DMA_ADC 中断优先级为 1 级（较低级）
- 10: DMA_ADC 中断优先级为 2 级（较高级）
- 11: DMA_ADC 中断优先级为 3 级（最高级）

SPIIP: DMA_SPI (SPI DMA) 中断优先级控制位

- 00: DMA_SPI 中断优先级为 0 级（最低级）
- 01: DMA_SPI 中断优先级为 1 级（较低级）
- 10: DMA_SPI 中断优先级为 2 级（较高级）
- 11: DMA_SPI 中断优先级为 3 级（最高级）

UR1TIP: DMA_UR1T (串口1发送DMA) 中断优先级控制位

- 00: DMA_UR1T 中断优先级为 0 级（最低级）
- 01: DMA_UR1T 中断优先级为 1 级（较低级）
- 10: DMA_UR1T 中断优先级为 2 级（较高级）
- 11: DMA_UR1T 中断优先级为 3 级（最高级）

UR1RIP: DMA_UR1R (串口1接收DMA) 中断优先级控制位

- 00: DMA_UR1R 中断优先级为 0 级（最低级）
- 01: DMA_UR1R 中断优先级为 1 级（较低级）
- 10: DMA_UR1R 中断优先级为 2 级（较高级）
- 11: DMA_UR1R 中断优先级为 3 级（最高级）

UR2TIP: DMA_UR2T (串口2发送DMA) 中断优先级控制位

- 00: DMA_UR2T 中断优先级为 0 级（最低级）

01: DMA.UR2T 中断优先级为 1 级 (较低级)

10: DMA.UR2T 中断优先级为 2 级 (较高级)

11: DMA.UR2T 中断优先级为 3 级 (最高级)

UR2RIP: DMA.UR2R (串口2接收DMA) 中断优先级控制位

00: DMA.UR2R 中断优先级为 0 级 (最低级)

01: DMA.UR2R 中断优先级为 1 级 (较低级)

10: DMA.UR2R 中断优先级为 2 级 (较高级)

11: DMA.UR2R 中断优先级为 3 级 (最高级)

UR3TIP: DMA.UR3T (串口3发送DMA) 中断优先级控制位

00: DMA.UR3T 中断优先级为 0 级 (最低级)

01: DMA.UR3T 中断优先级为 1 级 (较低级)

10: DMA.UR3T 中断优先级为 2 级 (较高级)

11: DMA.UR3T 中断优先级为 3 级 (最高级)

UR3RIP: DMA.UR3R (串口3接收DMA) 中断优先级控制位

00: DMA.UR3R 中断优先级为 0 级 (最低级)

01: DMA.UR3R 中断优先级为 1 级 (较低级)

10: DMA.UR3R 中断优先级为 2 级 (较高级)

11: DMA.UR3R 中断优先级为 3 级 (最高级)

UR4TIP: DMA.UR4T (串口4发送DMA) 中断优先级控制位

00: DMA.UR4T 中断优先级为 0 级 (最低级)

01: DMA.UR4T 中断优先级为 1 级 (较低级)

10: DMA.UR4T 中断优先级为 2 级 (较高级)

11: DMA.UR4T 中断优先级为 3 级 (最高级)

UR4RIP: DMA.UR4R (串口4接收DMA) 中断优先级控制位

00: DMA.UR4R 中断优先级为 0 级 (最低级)

01: DMA.UR4R 中断优先级为 1 级 (较低级)

10: DMA.UR4R 中断优先级为 2 级 (较高级)

11: DMA.UR4R 中断优先级为 3 级 (最高级)

LCMIP: DMA.LCM (LCM接口DMA) 中断优先级控制位

00: DMA.LCM 中断优先级为 0 级 (最低级)

01: DMA.LCM 中断优先级为 1 级 (较低级)

10: DMA.LCM 中断优先级为 2 级 (较高级)

11: DMA.LCM 中断优先级为 3 级 (最高级)

14.6 范例程序

14.6.1 INT0 中断（上升沿和下降沿），可同时支持上升沿和下降沿

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT0_Isr() interrupt 0
{
    if (P32)                                //判断上升沿和下降沿
    {
        P10 = !P10;                          //测试端口
    }
    else
    {
        P11 = !P11;                          //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80;                         //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IT0 = 0;                                //使能 INT0 上升沿和下降沿中断
    EX0 = 1;                                //使能 INT0 中断
    EA = I;

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz
```

P_SW2	DATA	0BAH
P1M1	DATA	091H

<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0003H</i>
	<i>LJMP</i>	<i>INT0ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>INT0ISR:</i>	<i>JB</i>	<i>P3.2,RISING</i>
	<i>CPL</i>	<i>P1.0</i> ;判断上升沿和下降沿 ;测试端口
	<i>RETI</i>	
<i>RISING:</i>	<i>CPL</i>	<i>P1.1</i> ;测试端口
	<i>RETI</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i> ;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>CLR</i>	<i>IT0</i> ;使能 INT0 上升沿和下降沿中断
	<i>SETB</i>	<i>EX0</i> ;使能 INT0 中断
	<i>SETB</i>	<i>EA</i>
	<i>JMP</i>	<i>\$</i>
	<i>END</i>	

14.6.2 INT0 中断（下降沿）

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void INT0_Isr() interrupt 0
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IT0 = 1;                                  //使能 INT0 下降沿中断
    EX0 = 1;                                  //使能 INT0 中断
    EA = 1;                                  

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0003H</i>	
<i>LJMP</i>	<i>INT0ISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>INT0ISR:</i>	<i>CPL</i>	<i>P1.0</i> ; 测试端口

RETI***START:***

```

MOV      SP, #5FH
ORL      P_SW2,#80H           ;使能访问 XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

SETB    IT0                  ;使能 INT0 下降沿中断
SETB    EX0                  ;使能 INT0 中断
SETB    EA
JMP     $

```

END

14.6.3 INT1 中断（上升沿和下降沿），可同时支持上升沿和下降沿

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT1_Isr() interrupt 2
{
    if (P33)                      //判断上升沿和下降沿
    {
        P10 = !P10;                //测试端口
    }
    else
    {
        P11 = !P11;                //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80;                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
}

```

```

P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

IT1 = 0;           //使能INT1 上升沿和下降沿中断
EX1 = 1;           //使能INT1 中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0013H</i>	
<i>LJMP</i>	<i>INTIISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>INTIISR:</i>		
<i>JB</i>	<i>P3.3,RISING</i>	; 判断上升沿和下降沿
<i>CPL</i>	<i>P1.0</i>	; 测试端口
<i>RETI</i>		
<i>RISING:</i>		
<i>CPL</i>	<i>P1.1</i>	; 测试端口
<i>RETI</i>		
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	

```

MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

CLR      IT1          ;使能INT1 上升沿和下降沿中断
SETB    EX1          ;使能INT1 中断
SETB    EA
JMP     $

END

```

14.6.4 INT1 中断（下降沿）

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT1_Isr() interrupt 2
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IT1 = 1;                                    //使能INT1 下降沿中断
    EX1 = 1;                                    //使能INT1 中断
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0013H</i>	
<i>LJMP</i>	<i>INTIISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>INTIISR:</i>	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>	 	
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	
	; 使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>SETB</i>	<i>IT1</i>	
<i>SETB</i>	<i>EX1</i>	
<i>SETB</i>	<i>EA</i>	
<i>JMP</i>	\$	
<i>END</i>		

14.6.5 INT2 中断（下降沿），只支持下降沿中断

C 语言代码

// 测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void INT2_Isr() interrupt 10
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    INTCLKO = EX2;                           //使能 INT2 中断
    EA = I;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

INTCLKO	DATA	8FH
EX2	EQU	10H
EX3	EQU	20H
EX4	EQU	40H
 P_SW2	DATA	0BAH
 P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
 ORG	DATA	0000H
LJMP	DATA	START
ORG	DATA	0053H

<i>LJMP</i>	<i>INT2ISR</i>
<i>ORG</i>	<i>0100H</i>
<i>INT2ISR:</i>	
<i>CPL</i>	<i>P1.0</i>
<i>RETI</i>	; 测试端口
<i>START:</i>	
<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	; 使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>MOV</i>	<i>INTCLKO,#EX2</i>
<i>SETB</i>	<i>EA</i>
<i>JMP</i>	<i>\$</i>
<i>END</i>	

14.6.6 INT3 中断（下降沿），只支持下降沿中断

C 语言代码

```
// 测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT3_Isr() interrupt 11
{
    P10 = !P10;                                // 测试端口
}

void main()
{
    P_SW2 |= 0x80;                            // 使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
```

```

P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

INTCLKO = EX3;           //使能 INT3 中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>INTCLKO</i>	<i>DATA</i>	<i>8FH</i>
<i>EX2</i>	<i>EQU</i>	<i>10H</i>
<i>EX3</i>	<i>EQU</i>	<i>20H</i>
<i>EX4</i>	<i>EQU</i>	<i>40H</i>
<i>PIM1</i>	<i>DATA</i>	<i>091H</i>
<i>PIM0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>005BH</i>
	<i>LJMP</i>	<i>INT3ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>INT3ISR:</i>	<i>CPL</i>	<i>P1.0</i> ; 测试端口
	<i>RETI</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i> ; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>

```

MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      INTCLKO,#EX3          ;使能 INT3 中断
SETB    EA
JMP     $

END

```

14.6.7 INT4 中断（下降沿），只支持下降沿中断

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT4_Isr() interrupt 16
{
    P10 = !P10;                      //测试端口
}

void main()
{
    P_SW2 |= 0x80;                  //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    INTCLKO = EX4;                  //使能 INT4 中断
    EA = 1;

    while (1);
}

```

汇编代码

```

;测试工作频率为 11.0592MHz

```

P_SW2	DATA	0BAH
INTCLKO	DATA	8FH
EX2	EQU	10H
EX3	EQU	20H
EX4	EQU	40H

```

P1M1      DATA      091H
P1M0      DATA      092H
P0M1      DATA      093H
P0M0      DATA      094H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP     START
        ORG      0083H
        LJMP     INT4ISR

        ORG      0100H
INT4ISR:
        CPL      P1.0          ; 测试端口
        RETI

START:
        MOV      SP, #5FH
        ORL      P_SW2,#80H      ; 使能访问 XFR，没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        MOV      INTCLKO,#EX4    ; 使能 INT4 中断
        SETB     EA
        JMP      $

END

```

14.6.8 定时器 0 中断

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"
```

```

void TM0_Isr() interrupt 1
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x00;
    TL0 = 0x66;                                //65536-11.0592M/12/1000
    TH0 = 0xfc;
    TR0 = 1;                                   //启动定时器
    ET0 = 1;                                   //使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG	0000H	
LJMP	START	
ORG	000BH	
LJMP	TM0ISR	
ORG	0100H	
TM0ISR:	CPL	P1.0
	; 测试端口	

RETI***START:***

```

MOV      SP, #5FH
ORL      P_SW2,#80H          ;使能访问 XFR, 没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      TMOD,#00H
MOV      TL0,#66H           ;65536-11.0592M/12/1000
MOV      TH0,#0FCH
SETB    TR0                 ;启动定时器
SETB    ET0                 ;使能定时器中断
SETB    EA

JMP      $

```

END

14.6.9 定时器 1 中断

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TMI_Isr() interrupt 3
{
    P10 = !P10;                  //测试端口
}

void main()
{
    P_SW2 |= 0x80;              //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
}

```

```

P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

TMOD = 0x00;
TL1 = 0x66;                                //65536-11.0592M/12/1000
TH1 = 0xfc;
TR1 = 1;                                     //启动定时器
ET1 = 1;                                     //使能定时器中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH

P1M1      DATA      091H
P1M0      DATA      092H
P0M1      DATA      093H
P0M0      DATA      094H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP    START
        ORG      001BH
        LJMP    TM1ISR

        ORG      0100H
TM1ISR:
        CPL      P1.0          ; 测试端口
        RETI

START:
        MOV      SP, #5FH
        ORL      P_SW2, #80H      ; 使能访问 XFR，没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H

```

```

MOV      P5M1, #00H
MOV      TMOD, #00H
MOV      TL1, #66H           ;65536-11.0592M/12/1000
MOV      TH1, #0FCH
SETB    TR1                 ;启动定时器
SETB    ET1                 ;使能定时器中断
SETB    EA

JMP      $

END

```

14.6.10 定时器 2 中断

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM2_Isr() interrupt 12
{
    P10 = !P10;               //测试端口
}

void main()
{
    P_SW2 |= 0x80;           //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T2L = 0x66;              //65536-11.0592M/12/1000
    T2H = 0xfc;
    AUXR = 0x10;             //启动定时器
    IE2 = ET2;                //使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

```
; 测试工作频率为 11.0592MHz
```

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>ET2</i>	<i>EQU</i>	<i>04H</i>
<i>AUXINTIF</i>	<i>DATA</i>	<i>0EFH</i>
<i>T2IF</i>	<i>EQU</i>	<i>01H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0063H</i>
	<i>LJMP</i>	<i>TM2ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>TM2ISR:</i>	<i>CPL</i>	<i>P1.0</i> ; 测试端口
	<i>RETI</i>	
<i>START:</i>		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i> ; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>T2L,#66H</i> ; 65536-11.0592M/12/1000
	<i>MOV</i>	<i>T2H,#0FCH</i>
	<i>MOV</i>	<i>AUXR,#10H</i> ; 启动定时器
	<i>MOV</i>	<i>IE2,#ET2</i> ; 使能定时器中断
	<i>SETB</i>	<i>EA</i>
	<i>JMP</i>	\$

END

14.6.11 定时器 3 中断

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM3_Isr() interrupt 19
{
    P10 = !P10; //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T3L = 0x66; //65536-11.0592M/12/1000
    T3H = 0xfc;
    T4T3M = 0x08; //启动定时器
    IE2 = ET3; //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T3L	DATA	0D5H
T3H	DATA	0D4H
T4T3M	DATA	0DIH
IE2	DATA	0AFH
ET3	EQU	20H
AUXINTIF	DATA	0EFH
T3IF	EQU	02H

<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>
<i>ORG</i>		<i>009BH</i>
<i>LJMP</i>		<i>TM3ISR</i>
<i>ORG</i>		<i>0100H</i>
<i>TM3ISR:</i>	<i>CPL</i>	<i>P1.0</i> ;测试端口
	<i>RETI</i>	
<i>START:</i>		
<i>MOV</i>		<i>SP, #5FH</i>
<i>ORL</i>		<i>P_SW2,#80H</i> ;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>		<i>P0M0, #00H</i>
<i>MOV</i>		<i>P0M1, #00H</i>
<i>MOV</i>		<i>P1M0, #00H</i>
<i>MOV</i>		<i>P1M1, #00H</i>
<i>MOV</i>		<i>P2M0, #00H</i>
<i>MOV</i>		<i>P2M1, #00H</i>
<i>MOV</i>		<i>P3M0, #00H</i>
<i>MOV</i>		<i>P3M1, #00H</i>
<i>MOV</i>		<i>P4M0, #00H</i>
<i>MOV</i>		<i>P4M1, #00H</i>
<i>MOV</i>		<i>P5M0, #00H</i>
<i>MOV</i>		<i>P5M1, #00H</i>
<i>MOV</i>		<i>T3L,#66H</i> ;65536-11.0592M/12/1000
<i>MOV</i>		<i>T3H,#0FCH</i>
<i>MOV</i>		<i>T4T3M,#08H</i> ;启动定时器
<i>MOV</i>		<i>IE2,#ET3</i> ;使能定时器中断
<i>SETB</i>		<i>EA</i>
<i>JMP</i>		\$
 <i>END</i>		

14.6.12 定时器 4 中断

C 语言代码

//测试工作频率为11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void TM4_Isr() interrupt 20
{
    P10 = !P10; //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T4L = 0x66; //65536-11.0592M/12/1000
    T4H = 0xfc;
    T4T3M = 0x80; //启动定时器
    IE2 = ET4; //使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T3L	DATA	0D5H
T3H	DATA	0D4H
T4L	DATA	0D3H
T4H	DATA	0D2H
T4T3M	DATA	0DIH
IE2	DATA	0AFH
ET3	EQU	20H
ET4	EQU	40H
AUXINTIF	DATA	0EFH
T3IF	EQU	02H
T4IF	EQU	04H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H

<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>00A3H</i>
	<i>LJMP</i>	<i>TM4ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>TM4ISR:</i>	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>T4L,#66H</i>
	<i>MOV</i>	<i>T4H,#0FCH</i>
	<i>MOV</i>	<i>T4T3M,#80H</i>
	<i>MOV</i>	; 启动定时器
	<i>IE2,#ET4</i>	; 使能定时器中断
	<i>SETB</i>	<i>EA</i>
	<i>JMP</i>	\$
 <i>END</i>		

14.6.13 UART1 中断

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void UART1_Isr() interrupt 4
{
    if (TI)
    {

```

```

    TI = 0;                                //清中断标志
    P10 = !P10;                            //测试端口
}
if (RI)
{
    RI = 0;                                //清中断标志
    P11 = !P11;                            //测试端口
}
}

void main()
{
    P_SW2 |= 0x80;                         //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    SCON = 0x50;
    T2L = 0xe8;                            //65536-11059200/115200/4=0FFE8H
    T2H = 0xff;
    AUXR = 0x15;                            //启动定时器
    ES = 1;                                 //使能串口中断
    EA = 1;                                 //发送测试数据
    SBUF = 0x5a;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H

P5M0	DATA	0CAH
	ORG	0000H
	LJMP	START
	ORG	0023H
	LJMP	UARTIISR
	ORG	0100H
UARTIISR:	JNB	TI,CHECKRI
	CLR	TI ;清中断标志
	CPL	P1.0 ;测试端口
CHECKRI:	JNB	RI,ISREXIT
	CLR	RI ;清中断标志
	CPL	P1.1 ;测试端口
ISREXIT:	RETI	
START:	MOV	SP, #5FH
	ORL	P_SW2,#80H ;使能访问 XFR，没有冲突不用关闭
	MOV	P0M0, #00H
	MOV	P0M1, #00H
	MOV	P1M0, #00H
	MOV	P1M1, #00H
	MOV	P2M0, #00H
	MOV	P2M1, #00H
	MOV	P3M0, #00H
	MOV	P3M1, #00H
	MOV	P4M0, #00H
	MOV	P4M1, #00H
	MOV	P5M0, #00H
	MOV	P5M1, #00H
	MOV	SCON, #50H
	MOV	T2L, #0E8H ;65536-11059200/115200/4=0FFE8H
	MOV	T2H, #0FFH
	MOV	AUXR, #15H ;启动定时器
	SETB	ES ;使能串口中断
	SETB	EA
	MOV	SBUF, #5AH ;发送测试数据
	JMP	\$
	END	

14.6.14 UART2 中断

C 语言代码

```
//测试工作频率为11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"
```

```

void UART2_Isr() interrupt 8
{
    if (S2CON & 0x02)
    {
        S2CON &= ~0x02;                                //清中断标志
        P12 = !P12;                                    //测试端口
    }
    if (S2CON & 0x01)
    {
        S2CON &= ~0x01;                                //清中断标志
        P13 = !P13;                                    //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80;                                  //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    S2CON = 0x10;
    T2L = 0xe8;                                     //65536-11059200/115200/4=0FFE8H
    T2H = 0xff;
    AUXR = 0x14;                                    //启动定时器
    IE2 = ES2;                                      //使能串口中断
    EA = 1;
    S2BUF = 0x5a;                                    //发送测试数据

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
S2CON	DATA	9AH
S2BUF	DATA	9BH
IE2	DATA	0AFH
ES2	EQU	01H
PIM1	DATA	091H
PIM0	DATA	092H

<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>0043H</i>	
	<i>LJMP</i>	<i>UART2ISR</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>UART2ISR:</i>	<i>PUSH</i>	<i>ACC</i>	
	<i>PUSH</i>	<i>PSW</i>	
	<i>MOV</i>	<i>A,S2CON</i>	
	<i>JNB</i>	<i>ACC.I,CHECKRI</i>	
	<i>ANL</i>	<i>S2CON,#NOT 02H</i>	;清中断标志
	<i>CPL</i>	<i>PI.2</i>	;测试端口
<i>CHECKRI:</i>	<i>MOV</i>	<i>A,S2CON</i>	
	<i>JNB</i>	<i>ACC.0,ISREXIT</i>	
	<i>ANL</i>	<i>S2CON,#NOT 01H</i>	;清中断标志
	<i>CPL</i>	<i>PI.3</i>	;测试端口
<i>ISREXIT:</i>	<i>POP</i>	<i>PSW</i>	
	<i>POP</i>	<i>ACC</i>	
	<i>RETI</i>		
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>MOV</i>	<i>S2CON,#10H</i>	
	<i>MOV</i>	<i>T2L,#0E8H</i>	;65536-11059200/115200/4=0FFE8H
	<i>MOV</i>	<i>T2H,#0FFH</i>	
	<i>MOV</i>	<i>AUXR,#14H</i>	;启动定时器
	<i>MOV</i>	<i>IE2,#ES2</i>	;使能串口中断
	<i>SETB</i>	<i>EA</i>	
	<i>MOV</i>	<i>S2BUF,#5AH</i>	;发送测试数据

JMP**\$****END**

14.6.15 UART3 中断

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void UART3_Isr() interrupt 17
{
    if(S3CON & 0x02)
    {
        S3CON &= ~0x02;                                //清中断标志
        P12 = !P12;                                    //测试端口
    }
    if(S3CON & 0x01)
    {
        S3CON &= ~0x01;                                //清中断标志
        P13 = !P13;                                    //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80;                                  //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    S3CON = 0x10;                                  //65536-11059200/115200/4=0FFE8H
    T2L = 0xe8;
    T2H = 0xff;
    AUXR = 0x14;                                    //启动定时器
    IE2 = ES3;                                     //使能串口中断
    EA = 1;
    S3BUF = 0x5a;                                   //发送测试数据

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>S3CON</i>	<i>DATA</i>	<i>0ACh</i>
<i>S3BUF</i>	<i>DATA</i>	<i>0ADH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>ES3</i>	<i>EQU</i>	<i>08H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>008BH</i>
	<i>LJMP</i>	<i>UART3ISR</i>
<i>UART3ISR:</i>	<i>ORG</i>	<i>0100H</i>
	<i>PUSH</i>	<i>ACC</i>
	<i>PUSH</i>	<i>PSW</i>
	<i>MOV</i>	<i>A,S3CON</i>
	<i>JNB</i>	<i>ACC.I,CHECKRI</i>
	<i>ANL</i>	<i>S3CON,#NOT 02H</i>
	<i>CPL</i>	; 清中断标志 ; 测试端口
<i>CHECKRI:</i>	<i>MOV</i>	<i>A,S3CON</i>
	<i>JNB</i>	<i>ACC.0,ISREXIT</i>
	<i>ANL</i>	<i>S3CON,#NOT 01H</i>
	<i>CPL</i>	; 清中断标志 ; 测试端口
<i>ISREXIT:</i>	<i>POP</i>	<i>PSW</i>
	<i>POP</i>	<i>ACC</i>
	<i>RETI</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>

```

MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      S3CON,#10H
MOV      T2L,#0E8H           ;65536-11059200/115200/4=0FFE8H
MOV      T2H,#0FFH
MOV      AUXR,#14H          ;启动定时器
MOV      IE2,#ES3            ;使能串口中断
SETB    EA
MOV      S3BUF,#5AH          ;发送测试数据

JMP      $

```

END

14.6.16 UART4 中断

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void UART4_Isr() interrupt 18
{
    if(S4CON & 0x02)
    {
        S4CON &= ~0x02;           //清中断标志
        P12 = !P12;               //测试端口
    }
    if(S4CON & 0x01)
    {
        S4CON &= ~0x01;           //清中断标志
        P13 = !P13;               //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80;             //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;
}

```

```

S4CON = 0x10;
T2L = 0xe8;                                //65536-11059200/115200/4=0FFE8H
T2H = 0xff;
AUXR = 0x14;                                //启动定时器
IE2 = ES4;                                   //使能串口中断
EA = 1;
S4BUF = 0x5a;                                //发送测试数据

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
S4CON	DATA	84H
S4BUF	DATA	85H
IE2	DATA	0AFH
ES4	EQU	10H
P0M1	DATA	093H
P0M0	DATA	094H
P1M1	DATA	091H
P1M0	DATA	092H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG	ORG	0000H
LJMP	START	
ORG	ORG	0093H
LJMP	LJMP	UART4ISR
ORG	ORG	0100H
UART4ISR:		
PUSH	PUSH	ACC
PUSH	PUSH	PSW
MOV	MOV	A,S4CON
JNB	JNB	ACC.1,CHECKRI
ANL	ANL	S4CON,#NOT 02H ;清中断标志
CPL	CPL	P1.2 ;测试端口
CHECKRI:		
MOV	MOV	A,S4CON
JNB	JNB	ACC.0,ISREXIT
ANL	ANL	S4CON,#NOT 01H ;清中断标志
CPL	CPL	P1.3 ;测试端口
ISREXIT:		
POP	POP	PSW

*POP
RETI*

ACC

START:

<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>S4CON,#10H</i>	
<i>MOV</i>	<i>T2L,#0E8H</i>	;65536-11059200/115200/4=0FFE8H
<i>MOV</i>	<i>T2H,#0FFH</i>	
<i>MOV</i>	<i>AUXR,#I4H</i>	;启动定时器
<i>MOV</i>	<i>IE2,#ES4</i>	;使能串口中断
<i>SETB</i>	<i>EA</i>	
<i>MOV</i>	<i>S4BUF,#5AH</i>	;发送测试数据
<i>JMP</i>	\$	

END

14.6.17 ADC 中断

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void ADC_Isr() interrupt 5
{
    ADC_CONTR &= ~0x20; //清中断标志
    P0 = ADC_RES; //测试端口
    P2 = ADC_RESL; //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
```

```

P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

ADCCFG = 0x00;                                //使能并启动 ADC 模块
ADC_CONTR = 0xc0;                             //使能ADC 中断
EADC = 1;
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>ADC_CONTR</i>	<i>DATA</i>	<i>0BCH</i>
<i>ADC_RES</i>	<i>DATA</i>	<i>0BDH</i>
<i>ADC_RESL</i>	<i>DATA</i>	<i>0BEH</i>
<i>ADCCFG</i>	<i>DATA</i>	<i>0DEH</i>
<i>EADC</i>	<i>BIT</i>	<i>IE.5</i>
<i>PIM1</i>	<i>DATA</i>	<i>091H</i>
<i>PIM0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
ORG <i>0000H</i>		
LJMP <i>START</i>		
ORG <i>002BH</i>		
LJMP <i>ADCISR</i>		
ORG <i>0100H</i>		
<i>ADCISR:</i>	<i>ANL</i>	<i>ADC_CONTR,#NOT 20H</i> ; 清中断标志
	<i>MOV</i>	<i>P0,ADC_RES</i> ; 测试端口
	<i>MOV</i>	<i>P2,ADC_RESL</i> ; 测试端口
	<i>RETI</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i> ; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>

```

MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      ADCCFG#00H
MOV      ADC_CONTR,#0C0H      ;使能并启动ADC 模块
SETB    EADC                 ;使能ADC 中断
SETB    EA

JMP     $

END

```

14.6.18 LVD 中断

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void LVD_Isr() interrupt 6
{
    PCON &= ~LVDF;           //清中断标志
    P10 = !P10;              //测试端口
}

void main()
{
    P_SW2 |= 0x80;          //使能访问XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    PCON &= ~LVDF;          //上电需要清中断标志
    RSTCFG = LVD3V0;         //设置LVD 电压为3.0V
    ELVD = 1; //使能LVD 中断
}

```

```
EA = I;
```

```
while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>	
<i>RSTCFG</i>	<i>DATA</i>	<i>0FFH</i>	
<i>ENLVR</i>	<i>EQU</i>	<i>40H</i>	;RSTCFG6
<i>LVD2V2</i>	<i>EQU</i>	<i>00H</i>	;LVD@2.2V
<i>LVD2V4</i>	<i>EQU</i>	<i>01H</i>	;LVD@2.4V
<i>LVD2V7</i>	<i>EQU</i>	<i>02H</i>	;LVD@2.7V
<i>LVD3V0</i>	<i>EQU</i>	<i>03H</i>	;LVD@3.0V
<i>ELVD</i>	<i>BIT</i>	<i>IE.6</i>	
<i>LVDF</i>	<i>EQU</i>	<i>20H</i>	;PCON.5
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>0033H</i>	
	<i>LJMP</i>	<i>LVDISR</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>LVDISR:</i>	<i>ANL</i>	<i>PCON,#NOT LVDF</i>	;清中断标志
	<i>CPL</i>	<i>P1.0</i>	;测试端口
	<i>RETI</i>		
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	

MOV	P5M1, #00H	
ANL	PCON,#NOT LVDF	;上电需要清中断标志
MOV	RSTCFG,# LVD3V0	;设置 LVD 电压为 3.0V
SETB	ELVD	;使能 LVD 中断
SETB	EA	
JMP	\$	
END		

14.6.19 比较器中断

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void CMP_Isr() interrupt 21
{
    CMPCRI &= ~0x40;                                //清中断标志
    P10 = !P10;                                     //测试端口
}

void main()
{
    P_SW2 |= 0x80;                                  //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CMPCR2 = 0x00;                                 //使能比较器模块
    CMPCRI = 0x80;                                //使能比较器边沿中断
    CMPCRI |= 0x30;                               //P3.6 为 CMP+ 输入脚
    CMPCRI &= ~0x08;                            //P3.7 为 CMP- 输入脚
    CMPCRI |= 0x04;                               //使能比较器输出
    CMPCRI |= 0x02;
    EA = 1;

    while (1);
}
```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>CMPCR1</i>	<i>DATA</i>	<i>0E6H</i>
<i>CMPCR2</i>	<i>DATA</i>	<i>0E7H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>00ABH</i>	
<i>LJMP</i>	<i>CMPISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>CMPISR:</i>	<i>ANL</i>	<i>CMPCR1,#NOT 40H</i>
	<i>CPL</i>	<i>P1.0</i> ;清中断标志
	<i>RETI</i>	;测试端口
<i>START:</i>	 	
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i> ;使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>CMPCR2,#00H</i>	
<i>MOV</i>	<i>CMPCR1,#80H</i> ;使能比较器模块	
<i>ORL</i>	<i>CMPCR1,#30H</i> ;使能比较器边沿中断	
<i>ANL</i>	<i>CMPCR1,#NOT 08H</i> ;P3.6 为 CMP+ 输入脚	
<i>ORL</i>	<i>CMPCR1,#04H</i> ;P3.7 为 CMP- 输入脚	
<i>ORL</i>	<i>CMPCR1,#02H</i> ;使能比较器输出	
<i>SETB</i>	<i>EA</i>	
<i>JMP</i>	\$	
 <i>END</i>		

14.6.20 SPI 中断

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void SPI_Isr() interrupt 9
{
    SPSTAT = 0xc0;                                //清中断标志
    P10 = !P10;                                    //测试端口
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    SPCTL = 0x50;                                //使能 SPI 主机模式
    SPSTAT = 0xc0;                                //清中断标志
    IE2 = ESPI;                                   //使能 SPI 中断
    EA = 1;
    SPDAT = 0x5a;                                //发送测试数据

    while (1);
}
```

汇编代码

;测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
SPSTAT	DATA	0CDH
SPCTL	DATA	0CEH
SPDAT	DATA	0CFH
IE2	DATA	0AFH
ESPI	EQU	02H
PIM1	DATA	091H
PIM0	DATA	092H
P0M1	DATA	093H

<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>004BH</i>
	<i>LJMP</i>	<i>SPIISR</i>
	<i>ORG</i>	<i>0100H</i>
SPIISR:		
	<i>MOV</i>	<i>SPSTAT,#0C0H</i>
	<i>CPL</i>	; 清中断标志
	<i>RET</i>	; 测试端口
START:		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>SPCTL,#50H</i>
	<i>MOV</i>	; 使能 SPI 主机模式
	<i>MOV</i>	<i>SPSTAT,#0C0H</i>
	<i>MOV</i>	; 清中断标志
	<i>MOV</i>	<i>IE2,#ESPI</i>
	<i>SETB</i>	; 使能 SPI 中断
	<i>EA</i>	
	<i>MOV</i>	<i>SPDAT,#5AH</i>
		; 发送测试数据
	<i>JMP</i>	\$
 END		

14.6.21 I2C 中断

C 语言代码

```
// 测试工作频率为 11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"
```

```

void I2C_Isr() interrupt 24
{
    if (I2CMSST & 0x40)
    {
        I2CMSST &= ~0x40; //清中断标志
        P10 = !P10; //测试端口
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    I2CCFG = 0xc0; //使能 I2C 主机模式
    I2CMSCR = 0x80; //使能 I2C 中断;
    EA = 1;

    I2CMSCR = 0x81; //发送起始命令

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
I2CCFG	XDATA	0FE80H
I2CMSCR	XDATA	0FE81H
I2CMSST	XDATA	0FE82H
I2CSLCR	XDATA	0FE83H
I2CSLST	XDATA	0FE84H
I2CSLADR	XDATA	0FE85H
I2CTXD	XDATA	0FE86H
I2CRXD	XDATA	0FE87H
PIMI	DATA	091H
PIM0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H

P4M1 DATA 0B3H
P4M0 DATA 0B4H
P5M1 DATA 0C9H
P5M0 DATA 0CAH

ORG 0000H
LJMP START
ORG 00C3H
LJMP I2CISR

ORG 0100H

I2CISR:

PUSH ACC
PUSH DPL
PUSH DPH
MOV DPTR,#I2CMSST
MOVX A,@DPTR
ANL A,#NOT 40H ;清中断标志
MOVX @DPTR,A
CPL P1.0 ;测试端口
POP DPH
POP DPL
POP ACC
RETI

START:

MOV SP, #5FH
ORL P_SW2,#80H ;使能访问 XFR，没有冲突不用关闭

MOV P0M0, #00H
MOV P0M1, #00H
MOV P1M0, #00H
MOV P1M1, #00H
MOV P2M0, #00H
MOV P2M1, #00H
MOV P3M0, #00H
MOV P3M1, #00H
MOV P4M0, #00H
MOV P4M1, #00H
MOV P5M0, #00H
MOV P5M1, #00H

MOV A,#0C0H ;使能 I2C 主机模式
MOV DPTR,#I2CCFG
MOVX @DPTR,A
MOV A,#80H ;使能 I2C 中断
MOV DPTR,#I2CMSCR
MOVX @DPTR,A
SETB EA

MOV A,#081H ;发送起始命令
MOV DPTR,#I2CMSCR
MOVX @DPTR,A

JMP \$

END

15 所有的 I/O 口均可中断（4 种模式），不是传统外部中断

产品线	I/O 中断	I/O 中断优先级	I/O 中断唤醒功能
STC8H1K08 系列			
STC8H1K28 系列			
STC8H3K64S2 系列 A 版本	●	1 级	
STC8H3K64S4 系列 A 版本	●	1 级	
STC8H3K64S2 系列 B 版本	●	1 级	●
STC8H3K64S4 系列 B 版本	●	1 级	●
STC8H8K64U 系列 A 版本			
STC8H8K64U 系列 B/C 版本	●	4 级	●
STC8H8K64U 系列 D 版本	●	4 级	●
STC8H4K64TL 系列	●	4 级	●
STC8H4K64TLCD 系列	●	4 级	●
STC8H1K08T 系列	●	4 级	●
STC8H2K12U-A/B 系列	●	4 级	●
STC8H2K32U 系列	●	4 级	●
STC8G1K08-SOP8 系列			
STC8G1K08A-SOP8 系列			

●: 有此功能，且功能正常

●: 有此功能，但有瑕疵，不建议使用

早期设计的【所有 普通 I/O 都支持的外部中断】部分芯片，灰色部分

只能使用 【高电平中断 / 低电平中断】，这 2 个功能需求特殊

不能使用 【下降沿中断 / 上升沿中断】，这 2 个功能需求多

STC8H8K64U, D 版 【所有 普通 I/O 都支持的外部中断】都已完善

后期设计的，或改版完善的【所有 普通 I/O 都支持的外部中断】部分芯片，黑色部分

能使用 【高电平中断 / 低电平中断】，这 2 个功能需求特殊

能使用 【下降沿中断 / 上升沿中断】，这 2 个功能需求多

STC8H 部分系列支持所有的 I/O 中断，且支持 4 种中断模式：下降沿中断、上升沿中断、低电平中断、高电平中断。每组 I/O 口都有独立的中断入口地址，且每个 I/O 可独立设置中断模式。

温馨提示：暂时如果有 I/O 口需同时支持上升沿中断和下降沿中断的需求时，可将每个信号源都各经过 100 欧姆电阻后，一分为二连接到两个不同的 I/O 口，其中一个 I/O 用于检测信号源的上升沿、另外一个 I/O 用于检测信号源的下降沿，从而实现这种需求。

15.1 I/O 口中断相关寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
P0INTE	P0 口中断使能寄存器	FD00H	P07INTE	P06INTE	P05INTE	P04INTE	P03INTE	P02INTE	P01INTE	P00INTE	0000,0000
P1INTE	P1 口中断使能寄存器	FD01H	P17INTE	P16INTE	P15INTE	P14INTE	P13INTE	P12INTE	P11INTE	P10INTE	0000,0000
P2INTE	P2 口中断使能寄存器	FD02H	P27INTE	P26INTE	P25INTE	P24INTE	P23INTE	P22INTE	P21INTE	P20INTE	0000,0000
P3INTE	P3 口中断使能寄存器	FD03H	P37INTE	P36INTE	P35INTE	P34INTE	P33INTE	P32INTE	P31INTE	P30INTE	0000,0000
P4INTE	P4 口中断使能寄存器	FD04H	P47INTE	P46INTE	P45INTE	P44INTE	P43INTE	P42INTE	P41INTE	P40INTE	0000,0000
P5INTE	P5 口中断使能寄存器	FD05H	-	-	P55INTE	P54INTE	P53INTE	P52INTE	P51INTE	P50INTE	xx00,0000
P6INTE	P6 口中断使能寄存器	FD06H	P67INTE	P66INTE	P65INTE	P64INTE	P63INTE	P62INTE	P61INTE	P60INTE	0000,0000
P7INTE	P7 口中断使能寄存器	FD07H	P77INTE	P76INTE	P75INTE	P74INTE	P73INTE	P72INTE	P71INTE	P70INTE	0000,0000
POINTF	P0 口中断标志寄存器	FD10H	P07INTF	P06INTF	P05INTF	P04INTF	P03INTF	P02INTF	P01INTF	P00INTF	0000,0000
P1INTF	P1 口中断标志寄存器	FD11H	P17INTF	P16INTF	P15INTF	P14INTF	P13INTF	P12INTF	P11INTF	P10INTF	0000,0000
P2INTF	P2 口中断标志寄存器	FD12H	P27INTF	P26INTF	P25INTF	P24INTF	P23INTF	P22INTF	P21INTF	P20INTF	0000,0000
P3INTF	P3 口中断标志寄存器	FD13H	P37INTF	P36INTF	P35INTF	P34INTF	P33INTF	P32INTF	P31INTF	P30INTF	0000,0000
P4INTF	P4 口中断标志寄存器	FD14H	P47INTF	P46INTF	P45INTF	P44INTF	P43INTF	P42INTF	P41INTF	P40INTF	0000,0000
P5INTF	P5 口中断标志寄存器	FD15H	-	-	P55INTF	P54INTF	P53INTF	P52INTF	P51INTF	P50INTF	xx00,0000
P6INTF	P6 口中断标志寄存器	FD16H	P67INTF	P66INTF	P65INTF	P64INTF	P63INTF	P62INTF	P61INTF	P60INTF	0000,0000
P7INTF	P7 口中断标志寄存器	FD17H	P77INTF	P76INTF	P75INTF	P74INTF	P73INTF	P72INTF	P71INTF	P70INTF	0000,0000
POIM0	P0 口中断模式寄存器 0	FD20H	P07IM0	P06IM0	P05IM0	P04IM0	P03IM0	P02IM0	P01IM0	P00IM0	0000,0000
PIIM0	P1 口中断模式寄存器 0	FD21H	P17IM0	P16IM0	P15IM0	P14IM0	P13IM0	P12IM0	P11IM0	P10IM0	0000,0000
P2IM0	P2 口中断模式寄存器 0	FD22H	P27IM0	P26IM0	P25IM0	P24IM0	P23IM0	P22IM0	P21IM0	P20IM0	0000,0000
P3IM0	P3 口中断模式寄存器 0	FD23H	P37IM0	P36IM0	P35IM0	P34IM0	P33IM0	P32IM0	P31IM0	P30IM0	0000,0000
P4IM0	P4 口中断模式寄存器 0	FD24H	P47IM0	P46IM0	P45IM0	P44IM0	P43IM0	P42IM0	P41IM0	P40IM0	0000,0000
PSIM0	P5 口中断模式寄存器 0	FD25H	-	-	P55IM0	P54IM0	P53IM0	P52IM0	P51IM0	P50IM0	xx00,0000
P6IM0	P6 口中断模式寄存器 0	FD26H	P67IM0	P66IM0	P65IM0	P64IM0	P63IM0	P62IM0	P61IM0	P60IM0	0000,0000
P7IM0	P7 口中断模式寄存器 0	FD27H	P77IM0	P76IM0	P75IM0	P74IM0	P73IM0	P72IM0	P71IM0	P70IM0	0000,0000
POIM1	P0 口中断模式寄存器 1	FD30H	P07IM1	P06IM1	P05IM1	P04IM1	P03IM1	P02IM1	P01IM1	P00IM1	0000,0000
PIIM1	P1 口中断模式寄存器 1	FD31H	P17IM1	P16IM1	P15IM1	P14IM1	P13IM1	P12IM1	P11IM1	P10IM1	0000,0000
P2IM1	P2 口中断模式寄存器 1	FD32H	P27IM1	P26IM1	P25IM1	P24IM1	P23IM1	P22IM1	P21IM1	P20IM1	0000,0000
P3IM1	P3 口中断模式寄存器 1	FD33H	P37IM1	P36IM1	P35IM1	P34IM1	P33IM1	P32IM1	P31IM1	P30IM1	0000,0000
P4IM1	P4 口中断模式寄存器 1	FD34H	P47IM1	P46IM1	P45IM1	P44IM1	P43IM1	P42IM1	P41IM1	P40IM1	0000,0000
PSIM1	P5 口中断模式寄存器 1	FD35H	-	-	P55IM1	P54IM1	P53IM1	P52IM1	P51IM1	P50IM1	xx00,0000
P6IM1	P6 口中断模式寄存器 1	FD36H	P67IM1	P66IM1	P65IM1	P64IM1	P63IM1	P62IM1	P61IM1	P60IM1	0000,0000
P7IM1	P7 口中断模式寄存器 1	FD37H	P77IM1	P76IM1	P75IM1	P74IM1	P73IM1	P72IM1	P71IM1	P70IM1	0000,0000
PINIPL	I/O 口中断优先级低寄存器	FD60H	P7IP	P6IP	P5IP	P4IP	P3IP	P2IP	P1IP	POIP	0000,0000
PINIPH	I/O 口中断优先级高寄存器	FD61H	P7IPH	P6IPH	P5IPH	P4IPH	P3IPH	P2IPH	P1IPH	POIPH	0000,0000
P0WKUE	P0 口中断唤醒使能寄存器	FD40H	P07WKUE	P06WKUE	P05WKUE	P04WKUE	P03WKUE	P02WKUE	P01WKUE	P00WKUE	0000,0000
P1WKUE	P1 口中断唤醒使能寄存器	FD41H	P17WKUE	P16WKUE	P15WKUE	P14WKUE	P13WKUE	P12WKUE	P11WKUE	P10WKUE	0000,0000
P2WKUE	P2 口中断唤醒使能寄存器	FD42H	P27WKUE	P26WKUE	P25WKUE	P24WKUE	P23WKUE	P22WKUE	P21WKUE	P20WKUE	0000,0000

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
P3WKUE	P3 口中断唤醒使能寄存器	FD43H	P37WKUE	P36WKUE	P35WKUE	P34WKUE	P33WKUE	P32WKUE	P31WKUE	P30WKUE	0000,0000
P4WKUE	P4 口中断唤醒使能寄存器	FD44H	P47WKUE	P46WKUE	P45WKUE	P44WKUE	P43WKUE	P42WKUE	P41WKUE	P40WKUE	0000,0000
P5WKUE	P5 口中断唤醒使能寄存器	FD45H	-	-	P55WKUE	P54WKUE	P53WKUE	P52WKUE	P51WKUE	P50WKUE	xx00,0000
P6WKUE	P6 口中断唤醒使能寄存器	FD46H	P67WKUE	P66WKUE	P65WKUE	P64WKUE	P63WKUE	P62WKUE	P61WKUE	P60WKUE	0000,0000
P7WKUE	P7 口中断唤醒使能寄存器	FD47H	P77WKUE	P76WKUE	P75WKUE	P74WKUE	P73WKUE	P72WKUE	P71WKUE	P70WKUE	0000,0000

15.1.1 端口中断使能寄存器 (PxINTE)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0INTE	FD00H	P07INTE	P06INTE	P05INTE	P04INTE	P03INTE	P02INTE	P01INTE	P00INTE
P1INTE	FD01H	P17INTE	P16INTE	P15INTE	P14INTE	P13INTE	P12INTE	P11INTE	P10INTE
P2INTE	FD02H	P27INTE	P26INTE	P25INTE	P24INTE	P23INTE	P22INTE	P21INTE	P20INTE
P3INTE	FD03H	P37INTE	P36INTE	P35INTE	P34INTE	P33INTE	P32INTE	P31INTE	P30INTE
P4INTE	FD04H	P47INTE	P46INTE	P45INTE	P44INTE	P43INTE	P42INTE	P41INTE	P40INTE
P5INTE	FD05H	-	-	P55INTE	P54INTE	P53INTE	P52INTE	P51INTE	P50INTE
P6INTE	FD06H	P67INTE	P66INTE	P65INTE	P64INTE	P63INTE	P62INTE	P61INTE	P60INTE
P7INTE	FD07H	P77INTE	P76INTE	P75INTE	P74INTE	P73INTE	P72INTE	P71INTE	P70INTE

PnINTE.x: 端口中断使能控制位 (n=0~7, x=0~7)

0: 关闭 Pn.x 口中断功能

1: 使能 Pn.x 口中断功能

15.1.2 端口中断标志寄存器 (PxINTF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0INTF	FD10H	P07INTF	P06INTF	P05INTF	P04INTF	P03INTF	P02INTF	P01INTF	P00INTF
P1INTF	FD11H	P17INTF	P16INTF	P15INTF	P14INTF	P13INTF	P12INTF	P11INTF	P10INTF
P2INTF	FD12H	P27INTF	P26INTF	P25INTF	P24INTF	P23INTF	P22INTF	P21INTF	P20INTF
P3INTF	FD13H	P37INTF	P36INTF	P35INTF	P34INTF	P33INTF	P32INTF	P31INTF	P30INTF
P4INTF	FD14H	P47INTF	P46INTF	P45INTF	P44INTF	P43INTF	P42INTF	P41INTF	P40INTF
P5INTF	FD15H	-	-	P55INTF	P54INTF	P53INTF	P52INTF	P51INTF	P50INTF
P6INTF	FD16H	P67INTF	P66INTF	P65INTF	P64INTF	P63INTF	P62INTF	P61INTF	P60INTF
P7INTF	FD17H	P77INTF	P76INTF	P75INTF	P74INTF	P73INTF	P72INTF	P71INTF	P70INTF

PnINTF.x: 端口中断请求标志位 (n=0~7, x=0~7)

0: Pn.x 口没有中断请求

1: Pn.x 口有中断请求, 若使能中断, 则会进入中断服务程序。标志位需软件清 0。

15.1.3 端口中断模式配置寄存器 (PxIM0, PxIM1)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0IM0	FD20H	P07IM0	P06IM0	P05IM0	P04IM0	P03IM0	P02IM0	P01IM0	P00IM0
P0IM1	FD30H	P07IM1	P06IM1	P05IM1	P04IM1	P03IM1	P02IM1	P01IM1	P00IM1
P1IM0	FD21H	P17IM0	P16IM0	P15IM0	P14IM0	P13IM0	P12IM0	P11IM0	P10IM0
P1IM1	FD31H	P17IM1	P16IM1	P15IM1	P14IM1	P13IM1	P12IM1	P11IM1	P10IM1
P2IM0	FD22H	P27IM0	P26IM0	P25IM0	P24IM0	P23IM0	P22IM0	P21IM0	P20IM0
P2IM1	FD32H	P27IM1	P26IM1	P25IM1	P24IM1	P23IM1	P22IM1	P21IM1	P20IM1
P3IM0	FD23H	P37IM0	P36IM0	P35IM0	P34IM0	P33IM0	P32IM0	P31IM0	P30IM0
P3IM1	FD33H	P37IM1	P36IM1	P35IM1	P34IM1	P33IM1	P32IM1	P31IM1	P30IM1
P4IM0	FD24H	P47IM0	P46IM0	P45IM0	P44IM0	P43IM0	P42IM0	P41IM0	P40IM0
P4IM1	FD34H	P47IM1	P46IM1	P45IM1	P44IM1	P43IM1	P42IM1	P41IM1	P40IM1
P5IM0	FD25H	-	-	P55IM0	P54IM0	P53IM0	P52IM0	P51IM0	P50IM0
P5IM1	FD35H	-	-	P55IM1	P54IM1	P53IM1	P52IM1	P51IM1	P50IM1
P6IM0	FD26H	P67IM0	P66IM0	P65IM0	P64IM0	P63IM0	P62IM0	P61IM0	P60IM0
P6IM1	FD36H	P67IM1	P66IM1	P65IM1	P64IM1	P63IM1	P62IM1	P61IM1	P60IM1
P7IM0	FD27H	P77IM0	P76IM0	P75IM0	P74IM0	P73IM0	P72IM0	P71IM0	P70IM0
P7IM1	FD37H	P77IM1	P76IM1	P75IM1	P74IM1	P73IM1	P72IM1	P71IM1	P70IM1

配置端口的模式

PnIM1.x	PnIM0.x	Pn.x 口中断模式	掉电唤醒支持
0	0	下降沿中断	支持
0	1	上升沿中断	支持
1	0	低电平中断	不支持
1	1	高电平中断	不支持

15.1.4 端口中断优先级控制寄存器 (PINIPL, PINIPH)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PINIPL	FD60H	P7IP	P6IP	P5IP	P4IP	P3IP	P2IP	P1IP	P0IP
PINIPH	FD61H	P7IPH	P6IPH	P5IPH	P4IPH	P3IPH	P2IPH	P1IPH	P0IPH

PxIPH, PxIP: Px口中断优先级控制位

00: Px 口中断优先级为 0 级 (最低级)

01: Px 口中断优先级为 1 级 (较低级)

10: Px 口中断优先级为 2 级 (较高级)

11: Px 口中断优先级为 3 级 (最高级)

15.1.5 端口中断掉电唤醒使能寄存器 (PxWKUE)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0WKUE	FD40H	P07WKUE	P06WKUE	P05WKUE	P04WKUE	P03WKUE	P02WKUE	P01WKUE	P00WKUE
P1WKUE	FD41H	P17WKUE	P16WKUE	P15WKUE	P14WKUE	P13WKUE	P12WKUE	P11WKUE	P10WKUE
P2WKUE	FD42H	P27WKUE	P26WKUE	P25WKUE	P24WKUE	P23WKUE	P22WKUE	P21WKUE	P20WKUE
P3WKUE	FD43H	P37WKUE	P36WKUE	P35WKUE	P34WKUE	P33WKUE	P32WKUE	P31WKUE	P30WKUE
P4WKUE	FD44H	P47WKUE	P46WKUE	P45WKUE	P44WKUE	P43WKUE	P42WKUE	P41WKUE	P40WKUE
P5WKUE	FD45H	-	-	P55WKUE	P54WKUE	P53WKUE	P52WKUE	P51WKUE	P50WKUE
P6WKUE	FD46H	P67WKUE	P66WKUE	P65WKUE	P64WKUE	P63WKUE	P62WKUE	P61WKUE	P60WKUE
P7WKUE	FD47H	P77WKUE	P76WKUE	P75WKUE	P74WKUE	P73WKUE	P72WKUE	P71WKUE	P70WKUE

PnxWKUE: 端口中断掉电唤醒使能控制位 (n=0~7, x=0~7)

0: 关闭 Pn.x 口中断掉电唤醒功能

1: 使能 Pn.x 口中断掉电唤醒功能 (注意: I/O 口中断模式必须选择下降沿中断模式或者上升沿中断模式才能实现掉电唤醒/主时钟停振被唤醒, 高电平中断模式或者低电平中断模式无法掉电唤醒/主时钟停振被唤醒)

15.2 范例程序

15.2.1 P0 口下降沿中断

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P0IM0 = 0x00;                                // 下降沿中断
    P0IM1 = 0x00;
    P0INTE = 0xff;                                // 使能 P0 口中断

    EA = I;

    while (1);
}

//由于中断向量大于 31，在 KEIL 中无法直接编译
//必须借用第 13 号中断入口地址
void common_isr() interrupt 13
{
    unsigned char psw2_st;
    unsigned char intf;

    intf = P0INTF;
    if (intf)
    {
        P0INTF = 0x00;
        if (intf & 0x01)
        {
            //P0.0 口中断
        }
        if (intf & 0x02)
        {
            //P0.1 口中断
        }
        if (intf & 0x04)
```

```

{
    //P0.2 口中断
}
if (intf & 0x08)
{
    //P0.3 口中断
}
if (intf & 0x10)
{
    //P0.4 口中断
}
if (intf & 0x20)
{
    //P0.5 口中断
}
if (intf & 0x40)
{
    //P0.6 口中断
}
if (intf & 0x80)
{
    //P0.7 口中断
}
}
}

// ISR.ASM
//将下面的代码保存为ISR.ASM，然后将文件加入到项目中即可

```

CSEG JMP P0INT_ISR: JMP END	AT 012BH POINT_ISR 006BH ;	; P0 口中断入口地址 ; 借用 13 号中断的入口地址
--	--	--

汇编代码

; 测试工作频率为 11.0592MHz

P0M0	DATA	094H
P0M1	DATA	093H
P1M0	DATA	092H
P1M1	DATA	091H
P2M0	DATA	096H
P2M1	DATA	095H
P3M0	DATA	0B2H
P3M1	DATA	0B1H
P4M0	DATA	0B4H
P4M1	DATA	0B3H
P5M0	DATA	0CAH
P5M1	DATA	0C9H
P6M0	DATA	0CCH
P6M1	DATA	0CBH
P7M0	DATA	0E2H
P7M1	DATA	0EIH
P_SW2	DATA	0BAH

<i>P0INTE</i>	<i>XDATA</i>	<i>0FD00H</i>	
<i>P0INTF</i>	<i>XDATA</i>	<i>0FD10H</i>	
<i>P0IM0</i>	<i>XDATA</i>	<i>0FD20H</i>	
<i>P0IM1</i>	<i>XDATA</i>	<i>0FD30H</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
<i>P0INT_ISR:</i>	<i>ORG</i>	<i>012BH</i>	<i>;P0 口中断入口地址</i>
	<i>PUSH</i>	<i>ACC</i>	
	<i>PUSH</i>	<i>B</i>	
	<i>PUSH</i>	<i>DPL</i>	
	<i>PUSH</i>	<i>DPH</i>	
	<i>MOV</i>	<i>DPTR,#P0INTF</i>	
	<i>MOVX</i>	<i>A,@DPTR</i>	
	<i>MOV</i>	<i>B,A</i>	
	<i>CLR</i>	<i>A</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>A,B</i>	
<i>CHECKP00:</i>	<i>JNB</i>	<i>ACC.0,CHECKP01</i>	
	<i>NOP</i>		<i>;P0.0 口中断</i>
<i>CHECKP01:</i>	<i>JNB</i>	<i>ACC.1,CHECKP02</i>	
	<i>NOP</i>		<i>;P0.1 口中断</i>
<i>CHECKP02:</i>	<i>JNB</i>	<i>ACC.2,CHECKP03</i>	
	<i>NOP</i>		<i>;P0.2 口中断</i>
<i>CHECKP03</i>	<i>JNB</i>	<i>ACC.3,CHECKP04</i>	
	<i>NOP</i>		<i>;P0.3 口中断</i>
<i>CHECKP04:</i>	<i>JNB</i>	<i>ACC.4,CHECKP05</i>	
	<i>NOP</i>		<i>;P0.4 口中断</i>
<i>CHECKP05:</i>	<i>JNB</i>	<i>ACC.5,CHECKP06</i>	
	<i>NOP</i>		<i>;P0.5 口中断</i>
<i>CHECKP06:</i>	<i>JNB</i>	<i>ACC.6,CHECKP07</i>	
	<i>NOP</i>		<i>;P0.6 口中断</i>
<i>CHECKP07:</i>	<i>JNB</i>	<i>ACC.7,P0ISREXIT</i>	
	<i>NOP</i>		<i>;P0.7 口中断</i>
<i>P0ISREXIT:</i>	<i>POP</i>	<i>DPH</i>	
	<i>POP</i>	<i>DPL</i>	
	<i>POP</i>	<i>B</i>	
	<i>POP</i>	<i>ACC</i>	
	<i>RETI</i>		
	<i>ORG</i>	<i>0200H</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	<i>;使能访问 XFR，没有冲突不用关闭</i>

<i>MOV</i>	<i>P0M0,#00H</i>
<i>MOV</i>	<i>P0M1,#00H</i>
<i>MOV</i>	<i>P1M0,#00H</i>
<i>MOV</i>	<i>P1M1,#00H</i>
<i>MOV</i>	<i>P2M0,#00H</i>
<i>MOV</i>	<i>P2M1,#00H</i>
<i>MOV</i>	<i>P3M0,#00H</i>
<i>MOV</i>	<i>P3M1,#00H</i>
<i>CLR</i>	<i>A</i>
<i>MOV</i>	<i>DPTR,# P0IM0</i>
	; 下降沿中断
<i>MOVX</i>	<i>@DPTR,A</i>
<i>MOV</i>	<i>DPTR,# P0IM1</i>
<i>MOVX</i>	<i>@DPTR,A</i>
<i>MOV</i>	<i>DPTR,# P0INTE</i>
<i>MOV</i>	<i>A,#0FFH</i>
<i>MOVX</i>	<i>@DPTR,A</i>
	; 使能 P0 口中断
<i>SETB</i>	<i>EA</i>
<i>JMP</i>	<i>\$</i>
<i>END</i>	

15.2.2 P1 口上升沿中断

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    PIIM0 = 0xff;                                  //上升沿中断
    PIIM1 = 0x00;
    PIINTE = 0xff;                                 //使能 P1 口中断

    EA = 1;
}
```

```

        while (1);
}

//由于中断向量大于31，在KEIL 中无法直接编译
//必须借用第13 号中断入口地址
void common_isr() interrupt 13
{
    unsigned char intf;

    intf = PIINTF;
    if (intf)
    {
        PIINTF = 0x00;
        if (intf & 0x01)
        {
            //P1.0 口中断
        }
        if (intf & 0x02)
        {
            //P1.1 口中断
        }
        if (intf & 0x04)
        {
            //P1.2 口中断
        }
        if (intf & 0x08)
        {
            //P1.3 口中断
        }
        if (intf & 0x10)
        {
            //P1.4 口中断
        }
        if (intf & 0x20)
        {
            //P1.5 口中断
        }
        if (intf & 0x40)
        {
            //P1.6 口中断
        }
        if (intf & 0x80)
        {
            //P1.7 口中断
        }
    }
}

// ISR.ASM
//将下面的代码保存为ISR.ASM，然后将文件加入到项目中即可

```

PIINT_ISR:	CSEG	AT 0133H	<i>;P1 口中断入口地址</i>
	JMP	PIINT_ISR	
	JMP	006BH	<i>;借用13 号中断的入口地址</i>
	END		

汇编代码

;测试工作频率为 11.0592MHz

<i>P0M0</i>	DATA	094H
<i>P0M1</i>	DATA	093H
<i>P1M0</i>	DATA	092H
<i>P1M1</i>	DATA	091H
<i>P2M0</i>	DATA	096H
<i>P2M1</i>	DATA	095H
<i>P3M0</i>	DATA	0B2H
<i>P3M1</i>	DATA	0B1H
<i>P4M0</i>	DATA	0B4H
<i>P4M1</i>	DATA	0B3H
<i>P5M0</i>	DATA	0CAH
<i>P5M1</i>	DATA	0C9H
<i>P6M0</i>	DATA	0CCH
<i>P6M1</i>	DATA	0CBH
<i>P7M0</i>	DATA	0E2H
<i>P7M1</i>	DATA	0E1H
 <i>P_SW2</i>	DATA	0BAH
 <i>PIINTE</i>	XDATA	0FD01H
<i>PIINTF</i>	XDATA	0FD11H
<i>PIIM0</i>	XDATA	0FD21H
<i>PIIM1</i>	XDATA	0FD31H
 <i>PIINT_ISR:</i>	ORG	0000H
	LJMP	START
 <i>CHECKP10:</i>	ORG	0133H
		;P1 口中断入口地址
	PUSH	ACC
	PUSH	B
	PUSH	DPL
	PUSH	DPH
	 MOV	DPTR,#PIINTF
	MOVX	A,@DPTR
	MOV	B,A
	CLR	A
	MOVX	@DPTR,A
	MOV	A,B
	 JNB	ACC.0,CHECKP11
	NOP	
		;P1.0 口中断
	 JNB	ACC.1,CHECKP12
	NOP	
		;P1.1 口中断
	 JNB	ACC.2,CHECKP13
	NOP	
		;P1.2 口中断
	 JNB	ACC.3,CHECKP14
	NOP	
		;P1.3 口中断
	 JNB	ACC.4,CHECKP15

NOP ;P1.4 口中断

CHECKP15:

JNB	ACC.5,CHECKP16
NOP	;P1.5 口中断

CHECKP16:

JNB	ACC.6,CHECKP17
NOP	;P1.6 口中断

CHECKP17:

JNB	ACC.7,PIISREXIT
NOP	;P1.7 口中断

PIISREXIT:

POP	DPH
POP	DPL
POP	B
POP	ACC
RETI	

ORG 0200H

START:

MOV	SP, #5FH
ORL	P_SW2,#80H ;使能访问 XFR, 没有冲突不用关闭

MOV	P0M0,#00H
MOV	P0M1,#00H
MOV	P1M0,#00H
MOV	P1M1,#00H
MOV	P2M0,#00H
MOV	P2M1,#00H
MOV	P3M0,#00H
MOV	P3M1,#00H

CLR	A
MOV	DPTR,# PIIM0 ;下降沿中断
MOVX	@DPTR,A
MOV	DPTR,# PIIM1
MOVX	@DPTR,A
MOV	DPTR,# PIINTE
MOV	A,#0FFH
MOVX	@DPTR,A ;使能 P1 口中断

SETB	EA
-------------	-----------

JMP	\$
------------	-----------

END

15.2.3 P2 口低电平中断

C 语言代码

```
//测试工作频率为 11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"
```

```
void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P2IM0 = 0x00;                                //低电平中断
    P2IM1 = 0xff;
    P2INTE = 0xff;                                //使能 P2 口中断

    EA = I;

    while (1);
}

//由于中断向量大于 31, 在 KEIL 中无法直接编译
//必须借用第 13 号中断入口地址
void common_isr() interrupt 13
{
    unsigned char intf;

    intf = P2INTF;
    if (intf)
    {
        P2INTF = 0x00;
        if (intf & 0x01)
        {
            //P2.0 口中断
        }
        if (intf & 0x02)
        {
            //P2.1 口中断
        }
        if (intf & 0x04)
        {
            //P2.2 口中断
        }
        if (intf & 0x08)
        {
            //P0.3 口中断
        }
        if (intf & 0x10)
        {
            //P2.4 口中断
        }
        if (intf & 0x20)
        {

```

```

        }
        if (intf & 0x40)
        {
            //P2.5 口中断
        }
        if (intf & 0x80)
        {
            //P2.6 口中断
        }
    }
}

```

// ISR.ASM

//将下面的代码保存为ISR.ASM，然后将文件加入到项目中即可

CSEG JMP P2INT_ISR: JMP END	AT 013BH P2INT_ISR 006BH ; 借用 13 号中断的入口地址	; P2 口中断入口地址
--	--	---------------------

汇编代码

; 测试工作频率为 11.0592MHz

P0M0	DATA	094H
P0M1	DATA	093H
P1M0	DATA	092H
P1M1	DATA	091H
P2M0	DATA	096H
P2M1	DATA	095H
P3M0	DATA	0B2H
P3M1	DATA	0B1H
P4M0	DATA	0B4H
P4M1	DATA	0B3H
P5M0	DATA	0CAH
P5M1	DATA	0C9H
P6M0	DATA	0CCH
P6M1	DATA	0CBH
P7M0	DATA	0E2H
P7M1	DATA	0E1H
 P_SW2	DATA	0BAH
 P2INTE	XDATA	0FD02H
P2INTF	XDATA	0FD12H
P2IM0	XDATA	0FD22H
P2IM1	XDATA	0FD32H
 ORG	0000H	
LJMP	START	
 P2INT_ISR:	ORG	013BH
	PUSH	ACC
	PUSH	B

<i>PUSH</i>	<i>DPL</i>
<i>PUSH</i>	<i>DPH</i>
<i>MOV</i>	<i>DPTR,#P2INTF</i>
<i>MOVX</i>	<i>A,@DPTR</i>
<i>MOV</i>	<i>B,A</i>
<i>CLR</i>	<i>A</i>
<i>MOVX</i>	<i>@DPTR,A</i>
<i>MOV</i>	<i>A,B</i>
<i>CHECKP20:</i>	
<i>JNB</i>	<i>ACC.0,CHECKP21</i>
<i>NOP</i>	<i>;P2.0 口中断</i>
<i>CHECKP21:</i>	
<i>JNB</i>	<i>ACC.1,CHECKP22</i>
<i>NOP</i>	<i>;P2.1 口中断</i>
<i>CHECKP22:</i>	
<i>JNB</i>	<i>ACC.2,CHECKP23</i>
<i>NOP</i>	<i>;P2.2 口中断</i>
<i>CHECKP23</i>	
<i>JNB</i>	<i>ACC.3,CHECKP24</i>
<i>NOP</i>	<i>;P2.3 口中断</i>
<i>CHECKP24:</i>	
<i>JNB</i>	<i>ACC.4,CHECKP25</i>
<i>NOP</i>	<i>;P2.4 口中断</i>
<i>CHECKP25:</i>	
<i>JNB</i>	<i>ACC.5,CHECKP26</i>
<i>NOP</i>	<i>;P2.5 口中断</i>
<i>CHECKP26:</i>	
<i>JNB</i>	<i>ACC.6,CHECKP27</i>
<i>NOP</i>	<i>;P2.6 口中断</i>
<i>CHECKP27:</i>	
<i>JNB</i>	<i>ACC.7,P2ISREXIT</i>
<i>NOP</i>	<i>;P2.7 口中断</i>
<i>P2ISREXIT:</i>	
<i>POP</i>	<i>DPH</i>
<i>POP</i>	<i>DPL</i>
<i>POP</i>	<i>B</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	
<i>ORG</i>	<i>0200H</i>
<i>START:</i>	
<i>MOV</i>	<i>SP,#5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	<i>;使能访问 XFR, 没有冲突不用关闭</i>
<i>MOV</i>	<i>P0M0,#00H</i>
<i>MOV</i>	<i>P0M1,#00H</i>
<i>MOV</i>	<i>P1M0,#00H</i>
<i>MOV</i>	<i>P1M1,#00H</i>
<i>MOV</i>	<i>P2M0,#00H</i>
<i>MOV</i>	<i>P2M1,#00H</i>
<i>MOV</i>	<i>P3M0,#00H</i>
<i>MOV</i>	<i>P3M1,#00H</i>
<i>CLR</i>	<i>A</i>
<i>MOV</i>	<i>DPTR,# P2IM0</i>
<i>MOVX</i>	<i>@DPTR,A</i>

```

MOV      DPTR,# P2IM1
MOVX    @DPTR,A
MOV      DPTR,# P2INTE
MOV      A,#0FFH
MOVX    @DPTR,A          ;使能 P2 口中断

SETB    EA

JMP     $

END

```

15.2.4 P3 口高电平中断

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;           //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P3IM0 = 0xff;           //高电平中断
    P3IM1 = 0xff;
    P3INTE = 0xff;          //使能 P3 口中断

    EA = 1;

    while (1);
}

```

//由于中断向量大于 31，在 KEIL 中无法直接编译
//必须借用第 13 号中断入口地址

```

void common_isr() interrupt 13
{
    unsigned char intf;

    intf = P3INTF;
    if (intf)
    {

```

```

P3INTF = 0x00;
if (intf & 0x01)
{
    //P3.0 口中断
}
if (intf & 0x02)
{
    //P3.1 口中断
}
if (intf & 0x04)
{
    //P3.2 口中断
}
if (intf & 0x08)
{
    //P3.3 口中断
}
if (intf & 0x10)
{
    //P3.4 口中断
}
if (intf & 0x20)
{
    //P3.5 口中断
}
if (intf & 0x40)
{
    //P3.6 口中断
}
if (intf & 0x80)
{
    //P3.7 口中断
}
}

}

// ISR.ASM
//将下面的代码保存为ISR.ASM，然后将文件加入到项目中即可

```

CSEG P3INT_ISR:	AT 0143H JMP P3INT_ISR	; P3 口中断入口地址
	JMP END	; 借用 13 号中断的入口地址

汇编代码

; 测试工作频率为 11.0592MHz

P0M0	DATA	094H
P0M1	DATA	093H
P1M0	DATA	092H
P1M1	DATA	091H
P2M0	DATA	096H
P2M1	DATA	095H
P3M0	DATA	0B2H
P3M1	DATA	0B1H

<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P6M0</i>	<i>DATA</i>	<i>0CCH</i>
<i>P6M1</i>	<i>DATA</i>	<i>0CBH</i>
<i>P7M0</i>	<i>DATA</i>	<i>0E2H</i>
<i>P7M1</i>	<i>DATA</i>	<i>0E1H</i>
<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P3INTE</i>	<i>XDATA</i>	<i>0FD03H</i>
<i>P3INTF</i>	<i>XDATA</i>	<i>0FD13H</i>
<i>P3IM0</i>	<i>XDATA</i>	<i>0FD23H</i>
<i>P3IM1</i>	<i>XDATA</i>	<i>0FD33H</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0143H</i>
		;P3 口中断入口地址
<i>P3INT_ISR:</i>		
	<i>PUSH</i>	<i>ACC</i>
	<i>PUSH</i>	<i>B</i>
	<i>PUSH</i>	<i>DPL</i>
	<i>PUSH</i>	<i>DPH</i>
	<i>MOV</i>	<i>DPTR,#P3INTF</i>
	<i>MOVX</i>	<i>A,@DPTR</i>
	<i>MOV</i>	<i>B,A</i>
	<i>CLR</i>	<i>A</i>
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>MOV</i>	<i>A,B</i>
<i>CHECKP30:</i>		
	<i>JNB</i>	<i>ACC.0,CHECKP31</i>
	<i>NOP</i>	
		;P3.0 口中断
<i>CHECKP31:</i>		
	<i>JNB</i>	<i>ACC.1,CHECKP32</i>
	<i>NOP</i>	
		;P3.1 口中断
<i>CHECKP32:</i>		
	<i>JNB</i>	<i>ACC.2,CHECKP33</i>
	<i>NOP</i>	
		;P3.2 口中断
<i>CHECKP33</i>		
	<i>JNB</i>	<i>ACC.3,CHECKP34</i>
	<i>NOP</i>	
		;P3.3 口中断
<i>CHECKP34:</i>		
	<i>JNB</i>	<i>ACC.4,CHECKP35</i>
	<i>NOP</i>	
		;P3.4 口中断
<i>CHECKP35:</i>		
	<i>JNB</i>	<i>ACC.5,CHECKP36</i>
	<i>NOP</i>	
		;P3.5 口中断
<i>CHECKP36:</i>		
	<i>JNB</i>	<i>ACC.6,CHECKP37</i>
	<i>NOP</i>	
		;P3.6 口中断
<i>CHECKP37:</i>		
	<i>JNB</i>	<i>ACC.7,P3ISREXIT</i>
	<i>NOP</i>	
		;P3.7 口中断
<i>P3ISREXIT:</i>		
	<i>POP</i>	<i>DPH</i>

<i>POP</i>	<i>DPL</i>
<i>POP</i>	<i>B</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	
<i>ORG</i>	<i>0200H</i>
START:	
<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0,#00H</i>
<i>MOV</i>	<i>P0M1,#00H</i>
<i>MOV</i>	<i>P1M0,#00H</i>
<i>MOV</i>	<i>P1M1,#00H</i>
<i>MOV</i>	<i>P2M0,#00H</i>
<i>MOV</i>	<i>P2M1,#00H</i>
<i>MOV</i>	<i>P3M0,#00H</i>
<i>MOV</i>	<i>P3M1,#00H</i>
<i>CLR</i>	<i>A</i>
<i>MOV</i>	<i>DPTR,# P3IM0</i>
	;高电平中断
<i>MOVX</i>	<i>@DPTR,A</i>
<i>MOV</i>	<i>DPTR,# P3IM1</i>
<i>MOVX</i>	<i>@DPTR,A</i>
<i>MOV</i>	<i>DPTR,# P3INTE</i>
<i>MOV</i>	<i>A,#0FFH</i>
<i>MOVX</i>	<i>@DPTR,A</i>
	;使能 P3 口中断
<i>SETB</i>	<i>EA</i>
<i>JMP</i>	<i>\$</i>
 END	

15.2.5 使用拓展 Keil 中断号方案的 I/O 口中断范例

C 语言代码

```
//测试工作频率为11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"
```

```
***** 功能说明 *****
```

本例为 STC8H4K64TLCD 系列的 I/O 口中断测试程序。

由于 Keil 软件的中断号只支持 0~31，而 STC8H 系列的 I/O 口中断的中断号均大于 31

为了能正确响应 I/O 中断，必须对 Keil 的中断进行一些特殊处理

本方案使用的是运行拓展 Keil 的中断号插件，将 Keil 支持的中断号拓展为 0~254

编译本项目前请先运行“Keil 中断向量拓展插件\拓展 Keil 的 C 代码中断号.exe”

测试方法:从 P0.0 口输入信号,根据 P0.0 口的中断模式产生中断,并在 P2.0 口输出反转信号

```
*****
```

```
void P0_isr() interrupt P0INT_VECTOR
{
    char intf;

    intf = P0INTF;
    P0INTF = 0x00;

    if (intf & 0x01)                                //判断中断标志
    {
        P20 = ~P20;                                //反转测试端口
    }
}

void P1_isr() interrupt PIINT_VECTOR
{
    PIINTF = 0x00;
}

void P2_isr() interrupt P2INT_VECTOR
{
    P2INTF = 0x00;
}

void P3_isr() interrupt P3INT_VECTOR
{
    P3INTF = 0x00;
}

void P4_isr() interrupt P4INT_VECTOR
{
    P4INTF = 0x00;
}

void P5_isr() interrupt P5INT_VECTOR
{
    P5INTF = 0x00;
}

void P6_isr() interrupt P6INT_VECTOR
{
    P6INTF = 0x00;
}

void P7_isr() interrupt P7INT_VECTOR
{
    P7INTF = 0x00;
}

void delay()
{
    int i;

    for (i=0; i<100; i++)
    {
        _nop_();
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}
```

```
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;
    P6M0 = 0x00;
    P6M1 = 0x00;
    P7M0 = 0x00;
    P7M1 = 0x00;

    P_SW2 = 0x80;

    P0IM0 &= ~0x01;                                // 设置 P0.0 口中断为下降沿模式
    P0IM1 &= ~0x01;                                // 设置 P0.0 口中断为上升沿模式
//    P0IM0 |= 0x01;                                // 设置 P0.0 口中断为低电平模式
//    P0IM1 |= 0x01;                                // 设置 P0.0 口中断为高电平模式
//    P0IM0 |= 0x01;                                // 使能 P0.0 口中断
//    P0IM1 |= 0x01;

    EA = I;

    while (1)
    {
        P27 = ~P27;
        delay();
    }
}
```

16 定时器/计数器

产品线	定时器					
	T0	T1	T2	T3	T4	T11
STC8H1K08 系列	● ₁₆	● ₁₆	● ₂₄			
STC8H1K28 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H3K64S4 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H3K64S2 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H8K64U 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H4K64TL 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H4K64TLCD 系列	● ₁₆	● ₁₆	● ₂₄	● ₂₄	● ₂₄	
STC8H1K08T 系列	● ₁₆	● ₁₆	● ₂₄			
STC8H2K12U-A/B 系列	● ₂₄	● ₂₄	● ₂₄			● ₂₄
STC8H2K32U 系列	● ₂₄					
STC8G1K08-SOP8 系列	● ₁₆	● ₁₆				
STC8G1K08A-SOP8 系列	● ₁₆	● ₁₆				

●₁₆: 定时器支持 16 位模式

●₂₄: 定时器支持 24 位模式 (8 位预分频+16 位定时)

STC8H 系列单片机内部设置了 5 个 16 位定时器/计数器: T0、T1、T2、T3 和 T4, 它们都具有计数方式和定时方式两种工作方式。

- 对定时器/计数器 T0 和 T1, 用它们在特殊功能寄存器 TMOD 中相对应的控制位 C/T 来选择 T0 或 T1 为定时器还是计数器。
- 对定时器/计数器 T2, 用特殊功能寄存器 AUXR 中的控制位 T2_C/T 来选择 T2 为定时器还是计数器。
- 对定时器/计数器 T3, 用特殊功能寄存器 T4T3M 中的控制位 T3_C/T 来选择 T3 为定时器还是计数器。
- 对定时器/计数器 T4, 用特殊功能寄存器 T4T3M 中的控制位 T4_C/T 来选择 T4 为定时器还是计数器。

定时器/计数器的核心部件是一个加法计数器, 其本质是对脉冲进行计数。只是计数脉冲来源不同:

- 如果计数脉冲来自系统时钟, 则为定时方式, 此时定时器/计数器每 12 个时钟或者每 1 个时钟得到一个计数脉冲, 计数值加 1;
- 如果计数脉冲来自单片机外部引脚, 则为计数方式, 每来一个脉冲加 1。

当定时器/计数器 T0、T1 及 T2 工作在定时模式时, 特殊功能寄存器 AUXR 中的 T0x12、T1x12 和 T2x12 分别决定是系统时钟/12 还是系统时钟/1 (不分频) 后让 T0、T1 和 T2 进行计数。

当定时器/计数器 T3 和 T4 工作在定时模式时, 特殊功能寄存器 T4T3M 中的 T3x12 和 T4x12 分别决定是系统时钟/12 还是系统时钟/1 (不分频) 后让 T3 和 T4 进行计数。

当定时器/计数器工作在计数模式时, 对外部脉冲计数不分频。

定时器/计数器 0 有 4 种工作模式:

- 模式 0 (16 位自动重装载模式);
- 模式 1 (16 位不可重装载模式);

- 模式 2 (8 位自动重装模式)；
- 模式 3 (不可屏蔽中断的 16 位自动重装载模式)。

定时器/计数器 1 除模式 3 外，其他工作模式与定时器/计数器 0 相同。T1 在模式 3 时无效，停止计数。

定时器 T2 的工作模式固定为 16 位自动重装载模式。 T2 可以当定时器使用，也可以当串口的波特率发生器和可编程时钟输出。

定时器 3、定时器 4 与定时器 T2 一样，它们的工作模式固定为 16 位自动重装载模式。 T3/T4 可以当定时器使用，也可以当串口的波特率发生器和可编程时钟输出。

16.1 定时器的相关寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
TCON	定时器控制寄存器	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000,0000
TMOD	定时器模式寄存器	89H	T1_GATE	T1_C/T	T1_M1	T1_M0	T0_GATE	T0_C/T	T0_M1	T0_M0	0000,0000
TL0	定时器 0 低 8 位寄存器	8AH									0000,0000
TL1	定时器 1 低 8 位寄存器	8BH									0000,0000
TH0	定时器 0 高 8 位寄存器	8CH									0000,0000
TH1	定时器 1 高 8 位寄存器	8DH									0000,0000
AUXR	辅助寄存器 1	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2	0000,0001
INTCLKO	中断与时钟输出控制寄存器	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO	x000,x000
WKTCL	掉电唤醒定时器低字节	AAH									1111,1111
WKTCH	掉电唤醒定时器高字节	ABH	WKTEN								0111,1111
T4T3M	定时器 4/3 控制寄存器	D1H	T4R	T4_C/T	T4x12	T4CLKO	T3R	T3_C/T	T3x12	T3CLKO	0000,0000
T4H	定时器 4 高字节	D2H									0000,0000
T4L	定时器 4 低字节	D3H									0000,0000
T3H	定时器 3 高字节	D4H									0000,0000
T3L	定时器 3 低字节	D5H									0000,0000
T2H	定时器 2 高字节	D6H									0000,0000
T2L	定时器 2 低字节	D7H									0000,0000

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
TM0PS	定时器 0 时钟预分频寄存器	FEA0H									0000,0000
TM1PS	定时器 1 时钟预分频寄存器	FEA1H									0000,0000
TM2PS	定时器 2 时钟预分频寄存器	FEA2H									0000,0000
TM3PS	定时器 3 时钟预分频寄存器	FEA3H									0000,0000
TM4PS	定时器 4 时钟预分频寄存器	FEA4H									0000,0000
T11CR	定时器 T11 控制寄存器	FE78H	T11R	T11_C/T	T11CLKO	T11x12	T11CS[1:0]	ET11I	T11IF		0000,0000
T11PS	定时器 T11 时钟预分频寄存器	FE79H									0000,0000
T11H	定时器 T11 高字节	FE7AH									0000,0000
T11L	定时器 T11 低字节	FE7BH									0000,0000

16.2 定时器 0/1

16.2.1 定时器 0/1 控制寄存器 (TCON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TCON	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1: T1溢出中断标志。T1被允许计数以后，从初值开始加1计数。当产生溢出时由硬件将TF1位置“1”，并向CPU请求中断，一直保持到CPU响应中断时，才由硬件清“0”（也可由查询软件清“0”）。

TR1: 定时器T1的运行控制位。该位由软件置位和清零。当GATE (TMOD.7) =0, TR1=1时就允许T1开始计数，TR1=0时禁止T1计数。当GATE (TMOD.7) =1, TR1=1且INT1输入高电平时，才允许T1计数。

TF0: T0溢出中断标志。T0被允许计数以后，从初值开始加1计数，当产生溢出时，由硬件置“1”TF0，向CPU请求中断，一直保持CPU响应该中断时，才由硬件清0（也可由查询软件清0）。

TR0: 定时器T0的运行控制位。该位由软件置位和清零。当GATE (TMOD.3) =0, TR0=1时就允许T0开始计数，TR0=0时禁止T0计数。当GATE (TMOD.3) =1, TR0=1且INT0输入高电平时，才允许T0计数，TR0=0时禁止T0计数。

IE1: 外部中断1请求源 (INT1/P3.3) 标志。IE1=1，外部中断向CPU请求中断，当CPU响应该中断时由硬件清“0”IE1。

IT1: 外部中断源1触发控制位。IT1=0，上升沿或下降沿均可触发外部中断1。IT1=1，外部中断1程控为下降沿触发方式。

IE0: 外部中断0请求源 (INT0/P3.2) 标志。IE0=1外部中断0向CPU请求中断，当CPU响应外部中断时，由硬件清“0”IE0（边沿触发方式）。

IT0: 外部中断源0触发控制位。IT0=0，上升沿或下降沿均可触发外部中断0。IT0=1，外部中断0程控为下降沿触发方式。

16.2.2 定时器 0/1 模式寄存器 (TMOD)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TMOD	89H	T1_GATE	T1_C/T	T1_M1	T1_M0	T0_GATE	T0_C/T	T0_M1	T0_M0

T1_GATE: 控制定时器1，置1时只有在INT1脚为高及TR1控制位置1时才可打开定时器/计数器1。

T0_GATE: 控制定时器0，置1时只有在INT0脚为高及TR0控制位置1时才可打开定时器/计数器0。

T1_C/T: 控制定时器1用作定时器或计数器，清0则用作定时器（对内部系统时钟进行计数），置1用作计数器（对引脚T1/P3.5外部脉冲进行计数）。

T0_C/T: 控制定时器0用作定时器或计数器，清0则用作定时器（对内部系统时钟进行计数），置1用作计数器（对引脚T0/P3.4外部脉冲进行计数）。

T1_M1/T1_M0: 定时器定时器/计数器1模式选择

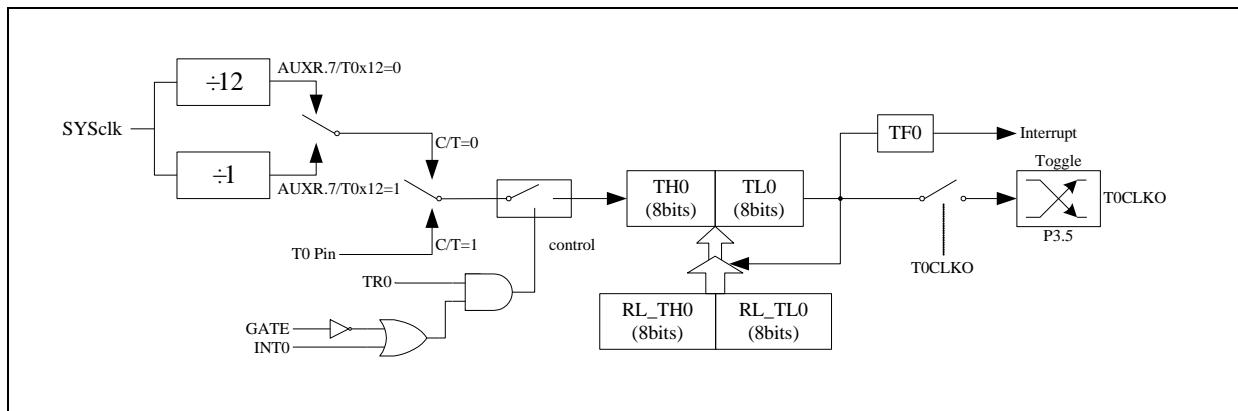
T1_M1	T1_M0	定时器/计数器1工作模式
0	0	16位自动重载模式 当[TH1,TL1]中的16位计数值溢出时，系统会自动将内部16位重载寄存器中的重载值装入[TH1,TL1]中。
0	1	16位不自动重载模式 当[TH1,TL1]中的16位计数值溢出时，定时器1将从0开始计数
1	0	8位自动重载模式 当TL1中的8位计数值溢出时，系统会自动将TH1中的重载值装入TL1中。
1	1	T1停止工作

T0_M1/T0_M0: 定时器定时器/计数器0模式选择

T0_M1	T0_M0	定时器/计数器0工作模式
0	0	16位自动重载模式 当[TH0,TL0]中的16位计数值溢出时，系统会自动将内部16位重载寄存器中的重载值装入[TH0,TL0]中。
0	1	16位不自动重载模式 当[TH0,TL0]中的16位计数值溢出时，定时器0将从0开始计数
1	0	8位自动重载模式 当TL0中的8位计数值溢出时，系统会自动将TH0中的重载值装入TL0中。
1	1	不可屏蔽中断的16位自动重载模式 与模式0相同，不可屏蔽中断，中断优先级最高，高于其他所有中断的优先级，并且不可关闭，可用作操作系统的系统节拍定时器，或者系统监控定时器。

16.2.3 定时器 0 模式 0 (16 位自动重装载模式)

此模式下定时器/计数器 0 作为可自动重装载的 16 位计数器, 如下图所示:



定时器/计数器 0 的模式 0: 16 位自动重装载模式

当 GATE=0 (TMOD.3) 时, 如 TR0=1, 则定时器计数。GATE=1 时, 允许由外部输入 INT0 控制定时器 0, 这样可实现脉宽测量。TR0 为 TCON 寄存器内的控制位, TCON 寄存器各位的具体功能描述见上节 TCON 寄存器的介绍。

当 C/T=0 时, 多路开关连接到系统时钟的分频输出, T0 对内部系统时钟计数, T0 工作在定时方式。当 C/T=1 时, 多路开关连接到外部脉冲输入 P3.4/T0, 即 T0 工作在计数方式。

STC 单片机的定时器 0 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T0 的速率由特殊功能寄存器 AUXR 中的 T0x12 决定, 如果 T0x12=0, T0 则工作在 12T 模式; 如果 T0x12=1, T0 则工作在 1T 模式

定时器 0 有两个隐藏的寄存器 RL_TH0 和 RL_TL0。RL_TH0 与 TH0 共有同一个地址, RL_TL0 与 TL0 共有同一个地址。当 TR0=0 即定时器/计数器 0 被禁止工作时, 对 TL0 写入的内容会同时写入 RL_TL0, 对 TH0 写入的内容也会同时写入 RL_TH0。当 TR0=1 即定时器/计数器 0 被允许工作时, 对 TL0 写入内容, 实际上不是写入当前寄存器 TL0 中, 而是写入隐藏的寄存器 RL_TL0 中, 对 TH0 写入内容, 实际上也不是写入当前寄存器 TH0 中, 而是写入隐藏的寄存器 RL_TH0, 这样可以巧妙地实现 16 位重装载定时器。当读 TH0 和 TL0 的内容时, 所读的内容就是 TH0 和 TL0 的内容, 而不是 RL_TH0 和 RL_TL0 的内容。

当定时器 0 工作在模式 0 (TMOD[1:0]/[M1,M0]=00B) 时, [TH0,TL0] 的溢出不仅置位 TF0, 而且会自动将[RL_TH0,RL_TL0]的内容重新装入[TH0,TL0]。

当 T0CLKO/INTCLKO.0=1 时, P3.5/T1 管脚配置为定时器 0 的时钟输出 T0CLKO。输出时钟频率为 T_0 溢出率/2。

如果 C/T=0, 定时器/计数器 T0 对内部系统时钟计数, 则:

$$T_0 \text{ 工作在 } 1T \text{ 模式 (AUXR.7/T0x12=1) 时的输出时钟频率} = (\text{SYSclk})/(65536-[RL_TH0, RL_TL0])/2$$

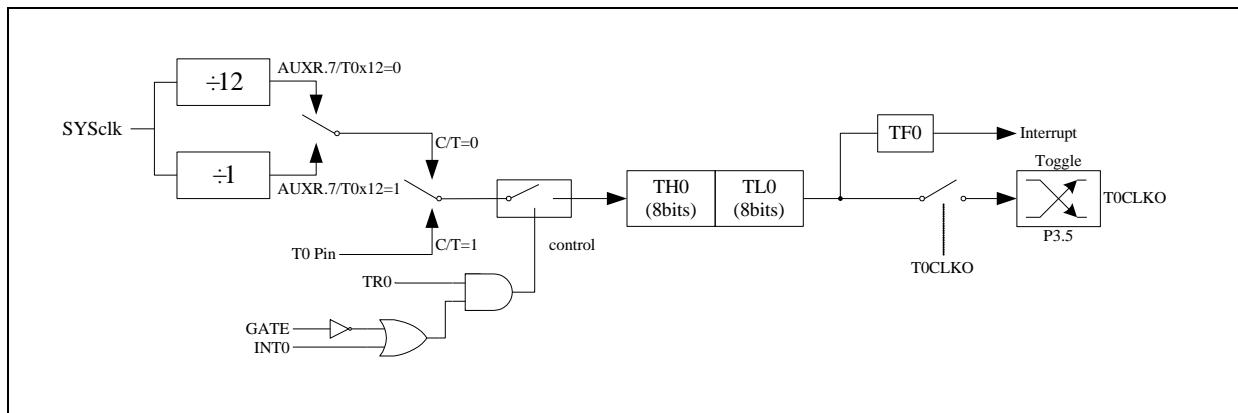
$$T_0 \text{ 工作在 } 12T \text{ 模式 (AUXR.7/T0x12=0) 时的输出时钟频率} = (\text{SYSclk})/12/(65536-[RL_TH0, RL_TL0])/2$$

如果 C/T=1, 定时器/计数器 T0 是对外部脉冲输入(P3.4/T0)计数, 则:

$$\text{输出时钟频率} = (T_0 \text{ Pin CLK}) / (65536-[RL_TH0, RL_TL0])/2$$

16.2.4 定时器 0 模式 1 (16 位不可重装载模式)

此模式下定时器/计数器 0 工作在 16 位不可重装载模式, 如下图所示



定时器/计数器 0 的模式 1: 16 位不可重装载模式

此模式下, 定时器/计数器 0 配置为 16 位不可重装载模式, 由 TL0 的 8 位和 TH0 的 8 位所构成。TL0 的 8 位溢出向 TH0 进位, TH0 计数溢出置位 TCON 中的溢出标志位 TF0。

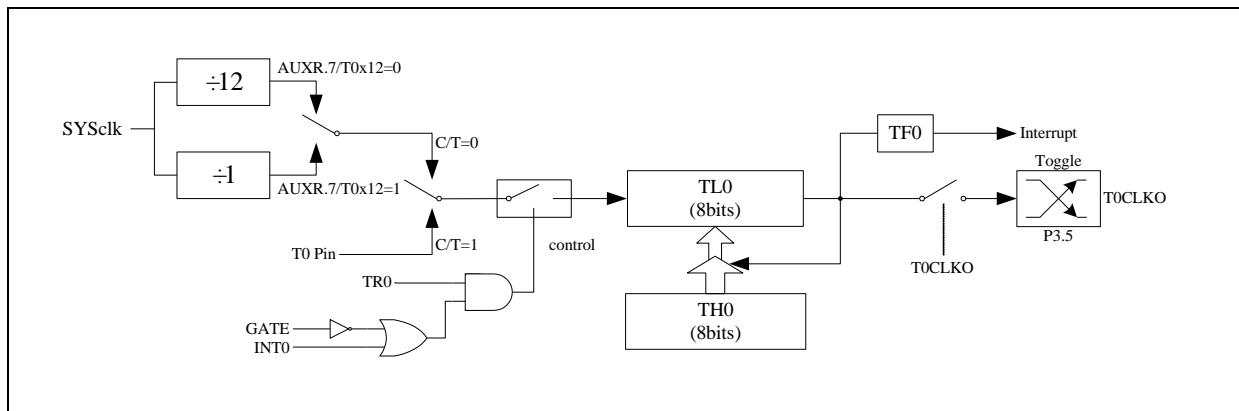
当 GATE=0(TM0D.3)时, 如 TR0=1, 则定时器计数。GATE=1 时, 允许由外部输入 INT0 控制定时器 0, 这样可实现脉宽测量。TR0 为 TCON 寄存器内的控制位, TCON 寄存器各位的具体功能描述见上节 TCON 寄存器的介绍。

当 C/T=0 时, 多路开关连接到系统时钟的分频输出, T0 对内部系统时钟计数, T0 工作在定时方式。当 C/T=1 时, 多路开关连接到外部脉冲输入 P3.4/T0, 即 T0 工作在计数方式。

STC 单片机的定时器 0 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T0 的速率由特殊功能寄存器 AUXR 中的 T0x12 决定, 如果 T0x12=0, T0 则工作在 12T 模式; 如果 T0x12=1, T0 则工作在 1T 模式

16.2.5 定时器 0 模式 2 (8 位自动重装载模式)

此模式下定时器/计数器 0 作为可自动重装载的 8 位计数器, 如下图所示:



定时器/计数器 0 的模式 2: 8 位自动重装载模式

TL0 的溢出不仅置位 TF0，而且将 TH0 的内容重新装入 TL0，TH0 内容由软件预置，重装时 TH0 内容不变。

当 T0CLKO/INTCLKO.0=1 时，P3.5/T1 管脚配置为定时器 0 的时钟输出 T0CLKO。输出时钟频率为 T0 溢出率/2。

如果 C/T=0，定时器/计数器 T0 对内部系统时钟计数，则：

$$\text{T0 工作在 1T 模式 (AUXR.7/T0x12=1) 时的输出时钟频率} = (\text{SYSclk})/(256-\text{TH0})/2$$

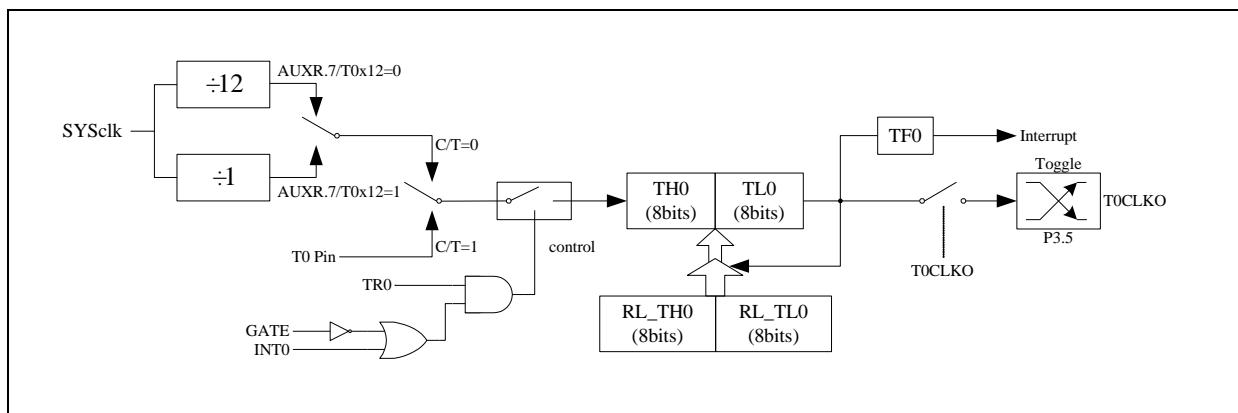
$$\text{T0 工作在 12T 模式 (AUXR.7/T0x12=0) 时的输出时钟频率} = (\text{SYSclk})/12/(256-\text{TH0})/2$$

如果 C/T=1，定时器/计数器 T0 是对外部脉冲输入(P3.4/T0)计数，则：

$$\text{输出时钟频率} = (\text{T0_Pin_CLK}) / (256-\text{TH0})/2$$

16.2.6 定时器 0 模式 3(不可屏蔽中断 16 位自动重装载, 实时操作系统节拍器)

对定时器/计数器 0, 其工作模式模式 3 与工作模式 0 是一样的 (下图定时器模式 3 的原理图, 与工作模式 0 是一样的)。唯一不同的是: 当定时器/计数器 0 工作在模式 3 时, 只需允许 ET0/IE.1(定时器/计数器 0 中断允许位), 不需要允许 EA/IE.7(总中断使能位)就能打开定时器/计数器 0 的中断, 此模式下的定时器/计数器 0 中断与总中断使能位 EA 无关, 一旦工作在模式 3 下的定时器/计数器 0 中断被打开(ET0=1), 那么该中断是不可屏蔽的, 该中断的优先级是最高的, 即该中断不能被任何中断所打断, 而且该中断打开后既不受 EA/IE.7 控制也不再受 ET0 控制, 当 EA=0 或 ET0=0 时都不能屏蔽此中断。故将此模式称为不可屏蔽中断的 16 位自动重装载模式。

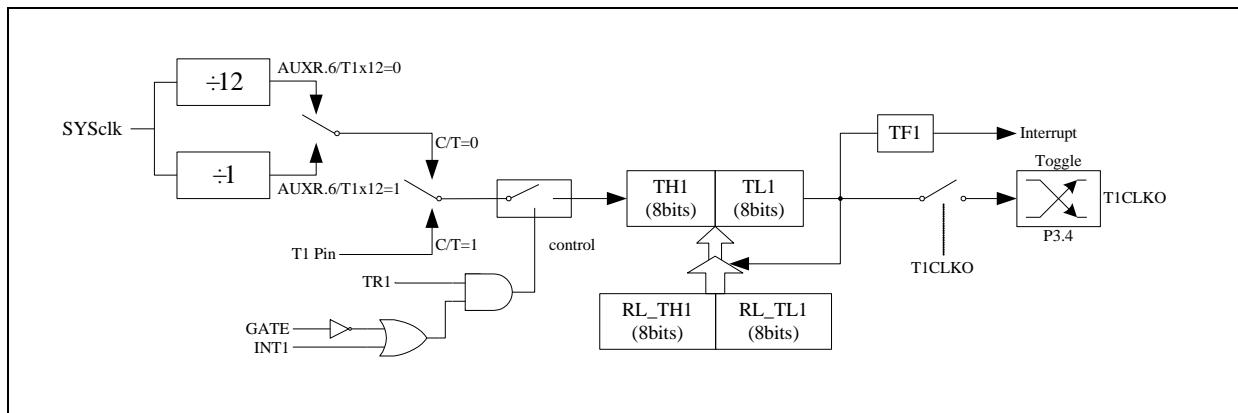


定时器/计数器 0 的模式 3: 不可屏蔽中断的 16 位自动重装载模式

注意: 当定时器/计数器 0 工作在模式 3(不可屏蔽中断的 16 位自动重装载模式)时, 不需要允许 EA/IE.7(总中断使能位), 只需允许 ET0/IE.1(定时器/计数器 0 中断允许位)就能打开定时器/计数器 0 的中断, 此模式下的定时器/计数器 0 中断与总中断使能位 EA 无关。一旦此模式下的定时器/计数器 0 中断被打开后, 该定时器/计数器 0 中断优先级就是最高的, 它不能被其它任何中断所打断(不管是比定时器/计数器 0 中断优先级低的中断还是比其优先级高的中断, 都不能打断此时的定时器/计数器 0 中断), 而且该中断打开后既不受 EA/IE.7 控制也不再受 ET0 控制了, 清零 EA 或 ET0 都不能关闭此中断。

16.2.7 定时器 1 模式 0 (16 位自动重装载模式)

此模式下定时器/计数器 1 作为可自动重装载的 16 位计数器, 如下图所示:



定时器/计数器 1 的模式 0: 16 位自动重装载模式

当 GATE=0 (TMOD.7) 时, 如 TR1=1, 则定时器计数。GATE=1 时, 允许由外部输入 INT1 控制定时器 1, 这样可实现脉宽测量。TR1 为 TCON 寄存器内的控制位, TCON 寄存器各位的具体功能描述见上节 TCON 寄存器的介绍。

当 C/T=0 时, 多路开关连接到系统时钟的分频输出, T1 对内部系统时钟计数, T1 工作在定时方式。当 C/T=1 时, 多路开关连接到外部脉冲输入 P3.5/T1, 即 T1 工作在计数方式。

STC 单片机的定时器 1 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T1 的速率由特殊功能寄存器 AUXR 中的 T1x12 决定, 如果 T1x12=0, T1 则工作在 12T 模式; 如果 T1x12=1, T1 则工作在 1T 模式

定时器 1 有两个隐藏的寄存器 RL_TH1 和 RL_TL1。RL_TH1 与 TH1 共有同一个地址, RL_TL1 与 TL1 共有同一个地址。当 TR1=0 即定时器/计数器 1 被禁止工作时, 对 TL1 写入的内容会同时写入 RL_TL1, 对 TH1 写入的内容也会同时写入 RL_TH1。当 TR1=1 即定时器/计数器 1 被允许工作时, 对 TL1 写入内容, 实际上不是写入当前寄存器 TL1 中, 而是写入隐藏的寄存器 RL_TL1 中, 对 TH1 写入内容, 实际上也不是写入当前寄存器 TH1 中, 而是写入隐藏的寄存器 RL_TH1, 这样可以巧妙地实现 16 位重装载定时器。当读 TH1 和 TL1 的内容时, 所读的内容就是 TH1 和 TL1 的内容, 而不是 RL_TH1 和 RL_TL1 的内容。

当定时器 1 工作在模式 1 (TMOD[5:4]/[M1,M0]=00B) 时, [TH1,TL1] 的溢出不仅置位 TF1, 而且会自动将[RL_TH1,RL_TL1]的内容重新装入[TH1,TL1]。

当 T1CLKO/INTCLKO.1=1 时, P3.4/T0 管脚配置为定时器 1 的时钟输出 T1CLKO。输出时钟频率为 **T1 溢出率/2**。

如果 C/T=0, 定时器/计数器 T1 对内部系统时钟计数, 则:

$$T1 \text{ 工作在 } 1T \text{ 模式 (AUXR.6/T1x12=1) 时的输出时钟频率} = (\text{SYSclk})/(65536-[RL_TH1, RL_TL1])/2$$

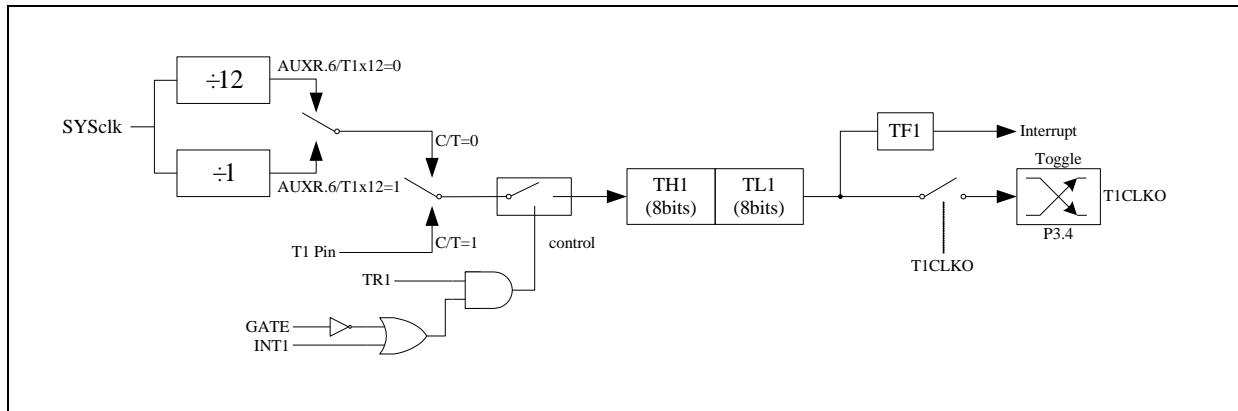
$$T1 \text{ 工作在 } 12T \text{ 模式 (AUXR.6/T1x12=0) 时的输出时钟频率} = (\text{SYSclk})/12/(65536-[RL_TH1, RL_TL1])/2$$

如果 C/T=1, 定时器/计数器 T1 是对外部脉冲输入(P3.5/T1)计数, 则:

$$\text{输出时钟频率} = (T1_Pin_CLK) / (65536-[RL_TH1, RL_TL1])/2$$

16.2.8 定时器 1 模式 1 (16 位不可重装载模式)

此模式下定时器/计数器 1 工作在 16 位不可重装载模式, 如下图所示



定时器/计数器 1 的模式 1: 16 位不可重装载模式

此模式下, 定时器/计数器 1 配置为 16 位不可重装载模式, 由 TL1 的 8 位和 TH1 的 8 位所构成。TL1 的 8 位溢出向 TH1 进位, TH1 计数溢出置位 TCON 中的溢出标志位 TF1。

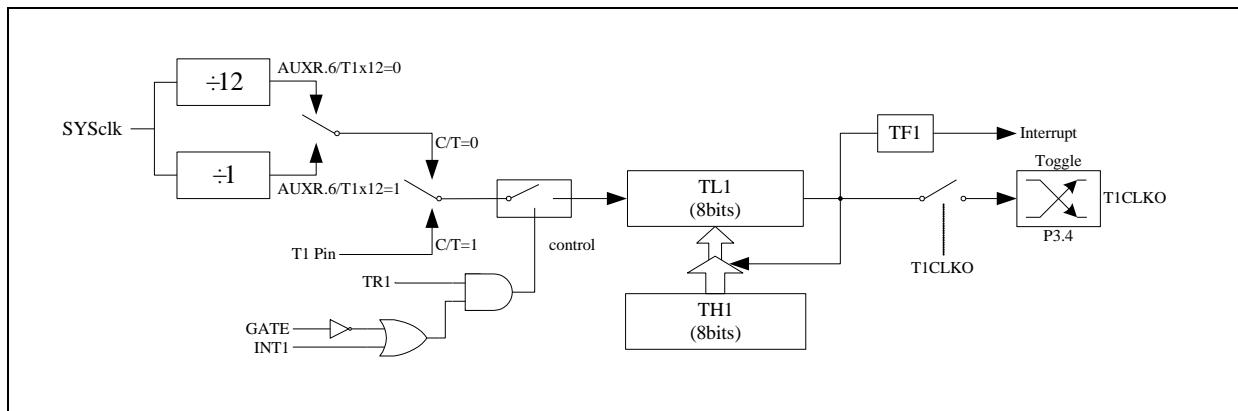
当 GATE=0(TM0D.7)时, 如 TR1=1, 则定时器计数。GATE=1 时, 允许由外部输入 INT1 控制定时器 1, 这样可实现脉宽测量。TR1 为 TCON 寄存器内的控制位, TCON 寄存器各位的具体功能描述见上节 TCON 寄存器的介绍。

当 C/T=0 时, 多路开关连接到系统时钟的分频输出, T1 对内部系统时钟计数, T1 工作在定时方式。当 C/T=1 时, 多路开关连接到外部脉冲输入 P3.5/T1, 即 T1 工作在计数方式。

STC 单片机的定时器 1 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T1 的速率由特殊功能寄存器 AUXR 中的 T1x12 决定, 如果 T1x12=0, T1 则工作在 12T 模式; 如果 T1x12=1, T1 则工作在 1T 模式

16.2.9 定时器 1 模式 2 (8 位自动重装载模式)

此模式下定时器/计数器 1 作为可自动重装载的 8 位计数器, 如下图所示:



定时器/计数器 1 的模式 2: 8 位自动重装载模式

TL1 的溢出不仅置位 TF1，而且将 TH1 的内容重新装入 TL1，TH1 内容由软件预置，重装时 TH1 内容不变。

当 T1CLKO/INTCLKO.1=1 时，P3.4/T0 管脚配置为定时器 1 的时钟输出 T1CLKO。输出时钟频率为 T1 溢出率/2。

如果 C/T=0，定时器/计数器 T1 对内部系统时钟计数，则：

$$\text{T1 工作在 1T 模式 (AUXR.6/T1x12=1) 时的输出时钟频率} = (\text{SYSclk})/(256-\text{TH1})/2$$

$$\text{T1 工作在 12T 模式 (AUXR.6/T1x12=0) 时的输出时钟频率} = (\text{SYSclk})/12/(256-\text{TH1})/2$$

如果 C/T=1，定时器/计数器 T1 是对外部脉冲输入(P3.5/T1)计数，则：

$$\text{输出时钟频率} = (\text{T1_Pin_CLK}) / (256-\text{TH1})/2$$

16.2.10 定时器 0 计数寄存器 (TL0, TH0)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TL0	8AH								
TH0	8CH								

当定时器/计数器0工作在16位模式（模式0、模式1、模式3）时，TL0和TH0组合成为一个16位寄存器，TL0为低字节，TH0为高字节。若为8位模式（模式2）时，TL0和TH0为两个独立的8位寄存器。

16.2.11 定时器 1 计数寄存器 (TL1, TH1)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TL1	8BH								
TH1	8DH								

当定时器/计数器1工作在16位模式（模式0、模式1）时，TL1和TH1组合成为一个16位寄存器，TL1为低字节，TH1为高字节。若为8位模式（模式2）时，TL1和TH1为两个独立的8位寄存器。

16.2.12 定时器 0 的 8 位预分频寄存器 (TM0PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TM0PS	FEA0H								

定时器0的时钟 = 系统时钟SYSclk \div (TM0PS + 1)

16.2.13 定时器 1 的 8 位预分频寄存器 (TM1PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TM1PS	FEA1H								

定时器1的时钟 = 系统时钟SYSclk \div (TM1PS + 1)

16.2.14 辅助寄存器 1 (AUXR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2

T0x12: 定时器0速度控制位

0: 12T 模式, 即 CPU 时钟 12 分频 (FOSC/12)

1: 1T 模式, 即 CPU 时钟不分频分频 (FOSC/1)

T1x12: 定时器1速度控制位

0: 12T 模式, 即 CPU 时钟 12 分频 (FOSC/12)

1: 1T 模式, 即 CPU 时钟不分频分频 (FOSC/1)

16.2.15 中断与时钟输出控制寄存器 (INTCLKO)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
INTCLKO	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO

T0CLKO: 定时器0时钟输出控制

0: 关闭时钟输出

1: 使能 P3.5 口的是定时器 0 时钟输出功能

当定时器 0 计数发生溢出时, P3.5 口的电平自动发生翻转。

T1CLKO: 定时器1时钟输出控制

0: 关闭时钟输出

1: 使能 P3.4 口的是定时器 1 时钟输出功能

当定时器 1 计数发生溢出时, P3.4 口的电平自动发生翻转。

16.2.16 定时器 0 计算公式

定时器模式	定时器速度	周期计算公式
模式0/3 (16位自动重载)	1T	定时器周期 = $\frac{65536 - [TH0, TL0]}{SYSclk}$ (自动重载)
	12T	定时器周期 = $\frac{65536 - [TH0, TL0]}{SYSclk} \times 12$ (自动重载)
模式1 (16位不自动重载)	1T	定时器周期 = $\frac{65536 - [TH0, TL0]}{SYSclk}$ (需软件装载)
	12T	定时器周期 = $\frac{65536 - [TH0, TL0]}{SYSclk} \times 12$ (需软件装载)
模式2 (8位自动重载)	1T	定时器周期 = $\frac{256 - TH0}{SYSclk}$ (自动重载)
	12T	定时器周期 = $\frac{256 - TH0}{SYSclk} \times 12$ (自动重载)

16.2.17 定时器 1 计算公式

定时器模式	定时器速度	周期计算公式
模式0 (16位自动重载)	1T	定时器周期 = $\frac{65536 - [TH1, TL1]}{SYSclk}$ (自动重载)
	12T	定时器周期 = $\frac{65536 - [TH1, TL1]}{SYSclk} \times 12$ (自动重载)
模式1 (16位不自动重载)	1T	定时器周期 = $\frac{65536 - [TH1, TL1]}{SYSclk}$ (需软件装载)
	12T	定时器周期 = $\frac{65536 - [TH1, TL1]}{SYSclk} \times 12$ (需软件装载)
模式2 (8位自动重载)	1T	定时器周期 = $\frac{256 - TH1}{SYSclk}$ (自动重载)
	12T	定时器周期 = $\frac{256 - TH1}{SYSclk} \times 12$ (自动重载)

16.3 定时器 2 (24 位定时器, 8 位预分频+16 位定时)

16.3.1 辅助寄存器 1 (AUXR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2

T2R: 定时器2的运行控制位

0: 定时器 2 停止计数

1: 定时器 2 开始计数

T2_C/T: 控制定时器2用作定时器或计数器, 清0则用作定时器(对内部系统时钟进行计数), 置1用作计数器(对引脚T2/P1.2外部脉冲进行计数)。

T2x12: 定时器2速度控制位

0: 12T 模式, 即 CPU 时钟 12 分频 (FOSC/12)

1: 1T 模式, 即 CPU 时钟不分频分频 (FOSC/1)

16.3.2 中断与时钟输出控制寄存器 (INTCLKO)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
INTCLKO	8FH	-	EX4	EX3	EX2	-	T2CLKO	T1CLKO	T0CLKO

T2CLKO: 定时器2时钟输出控制

0: 关闭时钟输出

1: 使能 P1.3 口的是定时器 2 时钟输出功能

当定时器 2 计数发生溢出时, P1.3 口的电平自动发生翻转。

16.3.3 定时器 2 计数寄存器 (T2L, T2H)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T2L	D7H								
T2H	D6H								

定时器/计数器2的工作模式固定为16位重载模式, T2L和T2H组合成为一个16位寄存器, T2L为低字节,

T2H为高字节。当[T2H,T2L]中的16位计数值溢出时, 系统会自动将内部16位重载寄存器中的重载值装入[T2H,T2L]中。

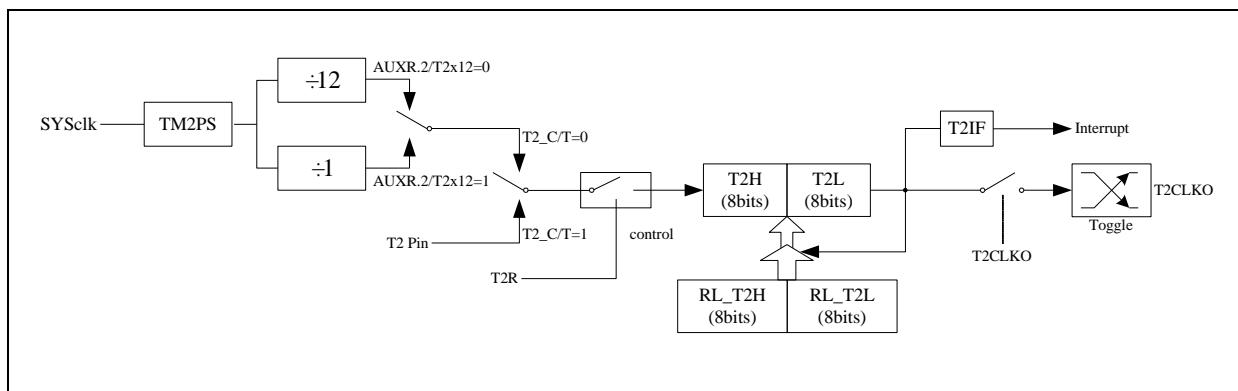
16.3.4 定时器 2 的 8 位预分频寄存器 (TM2PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TM2PS	FEA2H								

定时器2的时钟 = 系统时钟SYSclk ÷ (TM2PS + 1)

16.3.5 定时器 2 工作模式

定时器/计数器 2 的原理框图如下:



定时器/计数器 2 的工作模式: 16 位自动重装载模式

T2R/AUXR.4 为 AUXR 寄存器内的控制位, AUXR 寄存器各位的具体功能描述见上节 AUXR 寄存器的介绍。当 T2_C/T=0 时, 多路开关连接到系统时钟输出, T2 对内部系统时钟计数, T2 工作在定时方式。当 T2_C/T=1 时, 多路开关连接到外部脉冲输 T2, 即 T2 工作在计数方式。

STC 单片机的定时器 2 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T2 的速率由特殊功能寄存器 AUXR 中的 T2x12 决定, 如果 T2x12=0, T2 则工作在 12T 模式; 如果 T2x12=1, T2 则工作在 1T 模式

定时器 2 有两个隐藏的寄存器 RL_T2H 和 RL_T2L。RL_T2H 与 T2H 共有同一个地址, RL_T2L 与 T2L 共有同一个地址。当 T2R=0 即定时器/计数器 2 被禁止工作时, 对 T2L 写入的内容会同时写入 RL_T2L, 对 T2H 写入的内容也会同时写入 RL_T2H。当 T2R=1 即定时器/计数器 2 被允许工作时, 对 T2L 写入内容, 实际上不是写入当前寄存器 T2L 中, 而是写入隐藏的寄存器 RL_T2L 中, 对 T2H 写入内容, 实际上也不是写入当前寄存器 T2H 中, 而是写入隐藏的寄存器 RL_T2H, 这样可以巧妙地实现 16 位重装载定时器。当读 T2H 和 T2L 的内容时, 所读的内容就是 T2H 和 T2L 的内容, 而不是 RL_T2H 和 RL_T2L 的内容。

[T2H,T2L]的溢出不仅置位中断请求标志位 (T2IF), 使 CPU 转去执行定时器 2 的中断程序, 而且会自动将[RL_T2H,RL_T2L]的内容重新装入[T2H,T2L]。

(注意: 对于 STC8H1K08 系列、STC8H1K28 系列、STC8H3K64S2 系列和 STC8H3K64S4 系列这 4 个系列, T2IF 标志位为只写寄存器, 不可读取)

16.3.6 定时器 2 计算公式

定时器速度	周期计算公式
1T	定时器周期 = $\frac{65536 - [T2H, T2L]}{SYSclk/(TM2PS+1)}$ (自动重载)
12T	定时器周期 = $\frac{65536 - [T2H, T2L]}{SYSclk/(TM2PS+1)} \times 12$ (自动重载)

16.4 定时器 3/4 (24 位定时器, 8 位预分频+16 位定时)

16.4.1 定时器及外部计数器 3/4 的外部计数管脚和对外输出时钟管脚的选择

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T3T4PIN	FEACH	-	-	-	-	-	-	-	T3T4SEL
T3T4PS		-	-	-	-	-	-	-	

T3T4SEL: T3/T3CLKO/T4/T4CLKO 脚选择位

T3T4SEL	T3	T3CLKO	T4	T4CLKO
0	P0.4	P0.5	P0.6	P0.7
1	P0.0	P0.1	P0.2	P0.3

注: 在头文件中, T3T4PS 和 T3T4PIN 的定义相同, 两个寄存器名称可以通用

16.4.2 定时器 4/3 控制寄存器 (T4T3M)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T4T3M	D1H	T4R	T4_C/T	T4x12	T4CLKO	T3R	T3_C/T	T3x12	T3CLKO

T4R: 定时器4的运行控制位

- 0: 定时器 4 停止计数
- 1: 定时器 4 开始计数

T4_C/T: 控制定时器4用作定时器或计数器, 清0则用作定时器(对内部系统时钟进行计数), 置1用作计数器(对引脚T4/P0.6外部脉冲进行计数)。

T4x12: 定时器4速度控制位

- 0: 12T 模式, 即 CPU 时钟 12 分频 (FOSC/12)
- 1: 1T 模式, 即 CPU 时钟不分频分频 (FOSC/1)

T4CLKO: 定时器4时钟输出控制

- 0: 关闭时钟输出
- 1: 使能 P0.7 口的是定时器 4 时钟输出功能
当定时器 4 计数发生溢出时, P0.7 口的电平自动发生翻转。

T3R: 定时器3的运行控制位

- 0: 定时器 3 停止计数
- 1: 定时器 3 开始计数

T3_C/T: 控制定时器3用作定时器或计数器, 清0则用作定时器(对内部系统时钟进行计数), 置1用作计数器(对引脚T3/P0.4外部脉冲进行计数)。

T3x12: 定时器3速度控制位

- 0: 12T 模式, 即 CPU 时钟 12 分频 (FOSC/12)
- 1: 1T 模式, 即 CPU 时钟不分频分频 (FOSC/1)

T3CLKO: 定时器3时钟输出控制

- 0: 关闭时钟输出
- 1: 使能 P0.5 口的是定时器 3 时钟输出功能
当定时器 3 计数发生溢出时, P0.5 口的电平自动发生翻转。

16.4.3 定时器 3 计数寄存器 (T3L, T3H)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T3L	D5H								
T3H	D4H								

定时器/计数器3的工作模式固定为16位重载模式，T3L和T3H组合成为一个16位寄存器，T3L为低字节，T3H为高字节。当[T3H,T3L]中的16位计数值溢出时，系统会自动将内部16位重载寄存器中的重载值装入[T3H,T3L]中。

16.4.4 定时器 4 计数寄存器 (T4L, T4H)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T4L	D3H								
T4H	D2H								

定时器/计数器4的工作模式固定为16位重载模式，T4L和T4H组合成为一个16位寄存器，T4L为低字节，T4H为高字节。当[T4H,T4L]中的16位计数值溢出时，系统会自动将内部16位重载寄存器中的重载值装入[T4H,T4L]中。

16.4.5 定时器 3 的 8 位预分频寄存器 (TM3PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TM3PS	FEA3H								

定时器3的时钟 = 系统时钟SYSclk \div (TM3PS + 1)

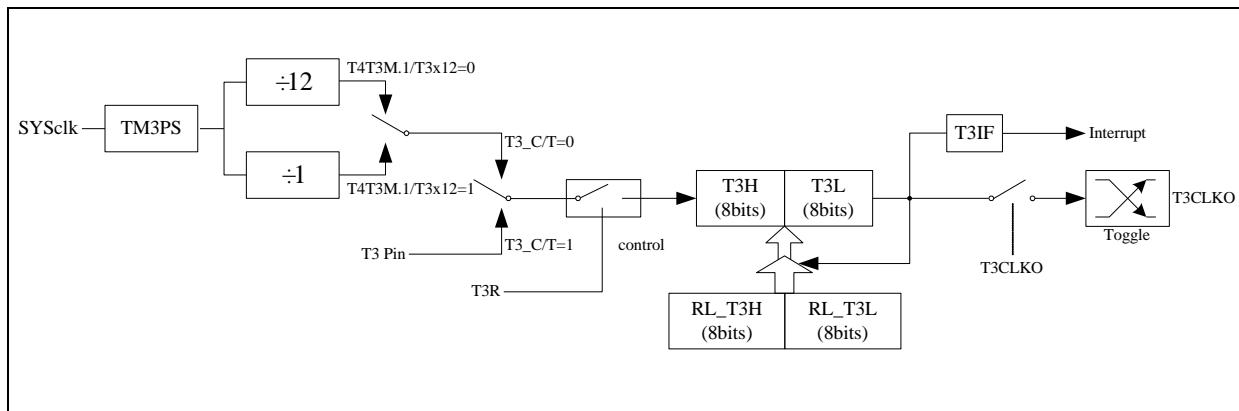
16.4.6 定时器 4 的 8 位预分频寄存器 (TM4PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
TM4PS	FEA4H								

定时器4的时钟 = 系统时钟SYSclk \div (TM4PS + 1)

16.4.7 定时器 3 工作模式

定时器/计数器 3 的原理框图如下:



定时器/计数器 3 的工作模式: 16 位自动重装载模式

T3R/T4T3M.3 为 T4T3M 寄存器内的控制位, T4T3M 寄存器各位的具体功能描述见上节 T4T3M 寄存器的介绍。

当 T3_C/T=0 时, 多路开关连接到系统时钟输出, T3 对内部系统时钟计数, T3 工作在定时方式。当 T3_C/T=1 时, 多路开关连接到外部脉冲输 T3, 即 T3 工作在计数方式。

STC 单片机的定时器 3 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T3 的速率由特殊功能寄存器 T4T3M 中的 T3x12 决定, 如果 T3x12=0, T3 则工作在 12T 模式; 如果 T3x12=1, T3 则工作在 1T 模式

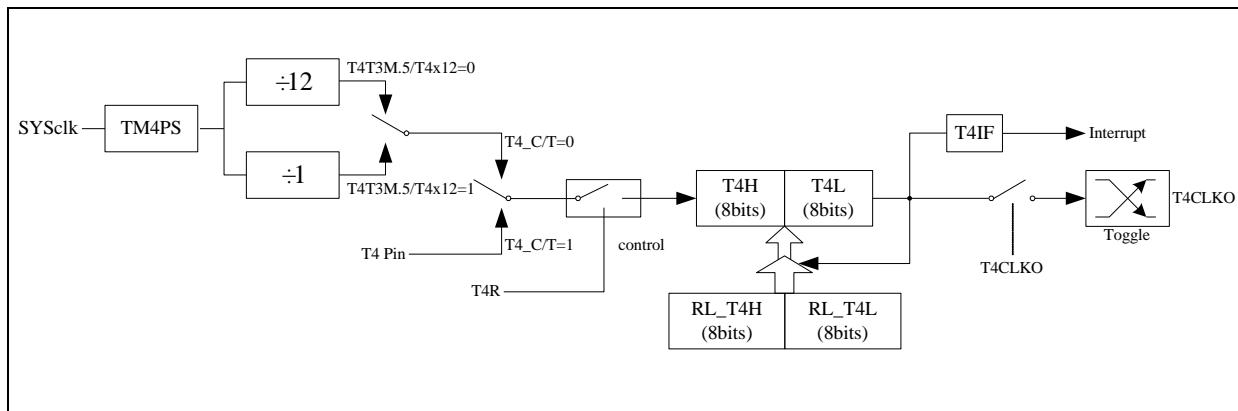
定时器 3 有两个隐藏的寄存器 RL_T3H 和 RL_T3L。RL_T3H 与 T3H 共有同一个地址, RL_T3L 与 T3L 共有同一个地址。当 T3R=0 即定时器/计数器 3 被禁止工作时, 对 T3L 写入的内容会同时写入 RL_T3L, 对 T3H 写入的内容也会同时写入 RL_T3H。当 T3R=1 即定时器/计数器 3 被允许工作时, 对 T3L 写入内容, 实际上不是写入当前寄存器 T3L 中, 而是写入隐藏的寄存器 RL_T3L 中, 对 T3H 写入内容, 实际上也不是写入当前寄存器 T3H 中, 而是写入隐藏的寄存器 RL_T3H, 这样可以巧妙地实现 16 位重装载定时器。当读 T3H 和 T3L 的内容时, 所读的内容就是 T3H 和 T3L 的内容, 而不是 RL_T3H 和 RL_T3L 的内容。

[T3H,T3L]的溢出不仅置位中断请求标志位 (T3IF), 使 CPU 转去执行定时器 3 的中断程序, 而且会自动将[RL_T3H,RL_T3L]的内容重新装入[T3H,T3L]。

(注意: 对于 STC8H1K08 系列、STC8H1K28 系列、STC8H3K64S2 系列和 STC8H3K64S4 系列这 4 个系列, T3IF 标志位为只写寄存器, 不可读取)

16.4.8 定时器 4 工作模式

定时器/计数器 4 的原理框图如下:



定时器/计数器 4 的工作模式: 16 位自动重装载模式

T4R/T4T3M.7 为 T4T3M 寄存器内的控制位, T4T3M 寄存器各位的具体功能描述见上节 T4T3M 寄存器的介绍。

当 T4_C/T=0 时, 多路开关连接到系统时钟输出, T4 对内部系统时钟计数, T4 工作在定时方式。当 T4_C/T=1 时, 多路开关连接到外部脉冲输 T4, 即 T4 工作在计数方式。

STC 单片机的定时器 4 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T4 的速率由特殊功能寄存器 T4T3M 中的 T4x12 决定, 如果 T4x12=0, T4 则工作在 12T 模式; 如果 T4x12=1, T4 则工作在 1T 模式

定时器 4 有两个隐藏的寄存器 RL_T4H 和 RL_T4L。RL_T4H 与 T4H 共有同一个地址, RL_T4L 与 T4L 共有同一个地址。当 T4R=0 即定时器/计数器 4 被禁止工作时, 对 T4L 写入的内容会同时写入 RL_T4L, 对 T4H 写入的内容也会同时写入 RL_T4H。当 T4R=1 即定时器/计数器 4 被允许工作时, 对 T4L 写入内容, 实际上不是写入当前寄存器 T4L 中, 而是写入隐藏的寄存器 RL_T4L 中, 对 T4H 写入内容, 实际上也不是写入当前寄存器 T4H 中, 而是写入隐藏的寄存器 RL_T4H, 这样可以巧妙地实现 16 位重装载定时器。当读 T4H 和 T4L 的内容时, 所读的内容就是 T4H 和 T4L 的内容, 而不是 RL_T4H 和 RL_T4L 的内容。

[T4H,T4L]的溢出不仅置位中断请求标志位 (T4IF), 使 CPU 转去执行定时器 4 的中断程序, 而且会自动将[RL_T4H,RL_T4L]的内容重新装入[T4H,T4L]。

(注意: 对于 STC8H1K08 系列、STC8H1K28 系列、STC8H3K64S2 系列和 STC8H3K64S4 系列这 4 个系列, T4IF 标志位为只写寄存器, 不可读取)

16.4.9 定时器 3 计算公式

定时器速度	周期计算公式
1T	定时器周期 = $\frac{65536 - [T3H, T3L]}{SYSclk/(TM3PS+1)}$ (自动重载)
12T	定时器周期 = $\frac{65536 - [T3H, T3L]}{SYSclk/(TM3PS+1)} \times 12$ (自动重载)

16.4.10 定时器 4 计算公式

定时器速度	周期计算公式
1T	定时器周期 = $\frac{65536 - [T4H, T4L]}{SYSclk/(TM4PS+1)}$ (自动重载)
12T	定时器周期 = $\frac{65536 - [T4H, T4L]}{SYSclk/(TM4PS+1)} \times 12$ (自动重载)

16.5 定时器 T11 (24 位定时器, 8 位预分频+16 位定时)

16.5.1 定时器 T11 控制寄存器 (T11CR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T11CR	FE78H	T11R	T11_C/T	T11CLKO	T11x12	T11CS[1:0]	ET11I	T11IF	

T11R: 定时器T11的运行控制位

0: 定时器 T11 停止计数

1: 定时器 T11 开始计数

T11_C/T: 控制定时器T11用作定时器或计数器, 清0则用作定时器(对内部系统时钟进行计数), 置1用作计数器(对引脚T11/P1.4外部脉冲进行计数)。

T11CLKO: 定时器T11时钟输出控制

0: 关闭时钟输出

1: 使能 P1.5 口的是定时器 T11 时钟输出功能

当定时器 T11 计数发生溢出时, P1.5 口的电平自动发生翻转。

T11x12: 定时器T11速度控制位

0: 12T 模式, 即 T11 时钟源 12 分频 (T11CLK/12)

1: 1T 模式, 即 T11 时钟源不分频分频 (T11CLK /1)

T11CS[1:0]: 定时器T11时钟源 (T11CLK) 选择

T11CS[1:0]	定时器T11时钟源 (T11CLK)
00	系统时钟SYSclk
01	内部高速IRC时钟
10	外部32K晶振
11	内部低速IRC

特别说明: 若定时器 T11 选择外部 32K 或者内部低速 IRC 作为时钟源, 当 MCU 进入主时钟停振/省电模式时, 定时器 T11 会继续工作, 发送 T11 定时中断时可唤醒主时钟停振/省电模式

ET11I: 定时/计数器 T11 的溢出中断允许位。

0: 禁止 T11 中断

1: 允许 T11 中断

T11IF: T11被允许计数以后, 从初值开始加1计数, 当产生溢出时, 由硬件置“1”

T11IF, 向CPU请求中断, 一直保持CPU响应该中断时, 才由硬件清0(也可由查询软件清0)。

16.5.2 定时器 T11 的 8 位预分频寄存器 (T11PS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T11PS	FE79H								

定时器T11的时钟 = 系统时钟SYSclk \div (T11PS + 1)

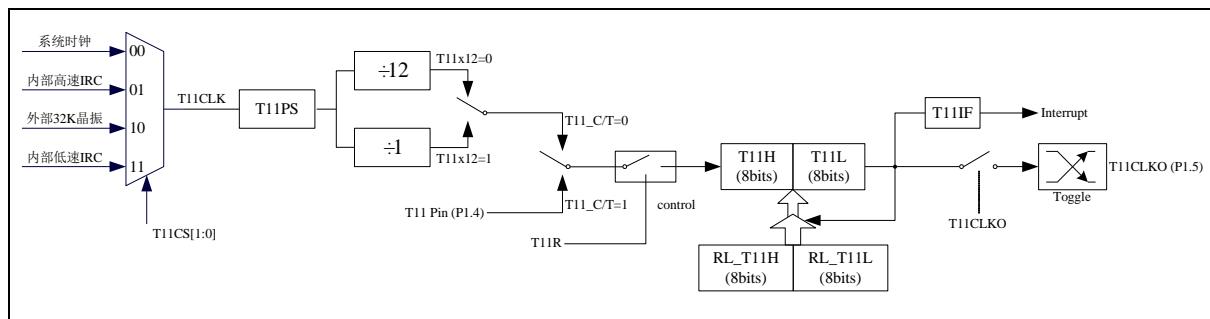
16.5.3 定时器 T11 计数寄存器 (T11L, T11H)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
T11L	FE7BH								
T11H	FE7AH								

定时器/计数器T11的工作模式固定为16位重载模式，T11L和T11H组合成为一个16位寄存器，T11L为低字节，T11H为高字节。当[T11H,T11L]中的16位计数值溢出时，系统会自动将内部16位重载寄存器中的重载值装入[T11H,T11L]中。

16.5.4 定时器 T11 工作模式

定时器/计数器 T11 的原理框图如下:



定时器/计数器 T11 的工作模式: 16 位自动重装载模式

当 T11_C/T=0 时, 多路开关连接到内部时钟 T11CLK, T11 对内部 T11CLK 时钟计数, T11 工作在定时方式。当 T11_C/T=1 时, 多路开关连接到外部脉冲输 T11 (P1.4), 即 T11 工作在计数方式。

STC 单片机的定时器 T11 有两种计数速率: 一种是 12T 模式, 每 12 个时钟加 1, 与传统 8051 单片机相同; 另外一种是 1T 模式, 每个时钟加 1, 速度是传统 8051 单片机的 12 倍。T11 的速率由特殊功能寄存器 T11x12 决定, 如果 T11x12=0, T11 则工作在 12T 模式; 如果 T11x12=1, T11 则工作在 1T 模式

定时器 T11 有两个隐藏的寄存器 RL_T11H 和 RL_T11L。RL_T11H 与 T11H 共有同一个地址, RL_T11L 与 T11L 共有同一个地址。当 T11R=0 即定时器/计数器 T11 被禁止工作时, 对 T11L 写入的内容会同时写入 RL_T11L, 对 T11H 写入的内容也会同时写入 RL_T11H。当 T11R=1 即定时器/计数器 T11 被允许工作时, 对 T11L 写入内容, 实际上不是写入当前寄存器 T11L 中, 而是写入隐藏的寄存器 RL_T11L 中, 对 T11H 写入内容, 实际上也不是写入当前寄存器 T11H 中, 而是写入隐藏的寄存器 RL_T11H, 这样可以巧妙地实现 16 位重装载定时器。当读 T11H 和 T11L 的内容时, 所读的内容就是 T11H 和 T11L 的内容, 而不是 RL_T11H 和 RL_T11L 的内容。

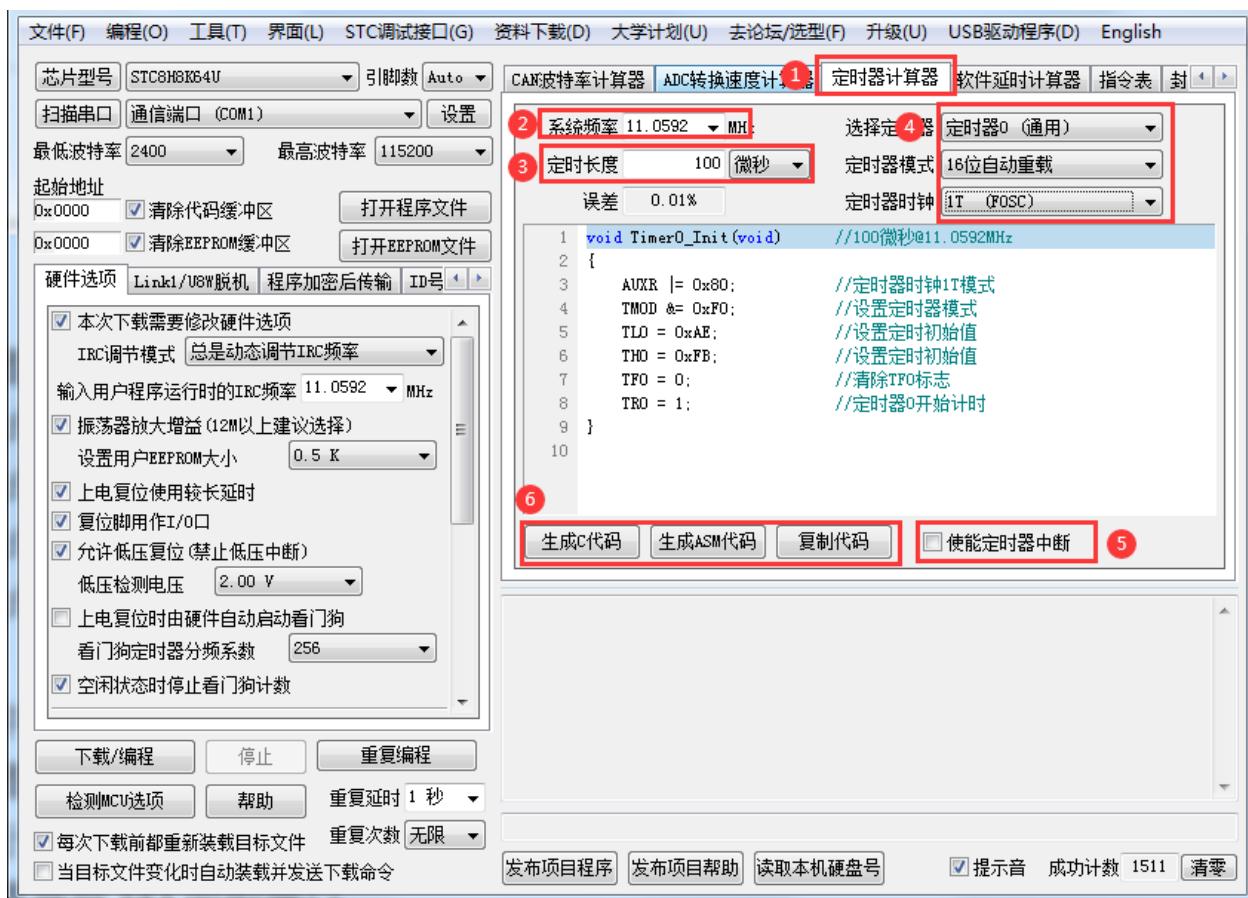
[T11H,T11L]的溢出不仅置位中断请求标志位 (T11IF), 使 CPU 转去执行定时器 T11 的中断程序, 而且会自动将[RL_T11H,RL_T11L]的内容重新装入[T11H,T11L]。

特别的, 定时器 T11 的时钟源可通过 T11CS 寄存器进行选择, 可选择: 系统时钟、内部高速 IRC 时钟、外部 32K 晶振以及内部低速 IRC 时钟

16.5.5 定时器 T11 计算公式

定时器速度	周期计算公式
1T	定时器周期 = $\frac{65536 - [T11H, T11L]}{T11CLK/(T11PS+1)}$ (自动重载)
12T	定时器周期 = $\frac{65536 - [T11H, T11L]}{T11CLK/(T11PS+1)} \times 12$ (自动重载)

16.6 AiCube-ISP | 定时器计算器工具



- ①: 在下载软件中选择“定时器计算器”功能页，进定时器代码生成界面
- ②: 设置系统工作频率（单位：MHz）
- ③: 设置定时时间长度（单位：毫秒/微秒）
- ④: 选择目标定时器，并设置定时器工作模式
- ⑤: 选择是否需要使能定时器中断
- ⑥: 手动生成 C 代码或者 ASM 代码，复制范例

16.7 范例程序

16.7.1 定时器 0（模式 0—16 位自动重载），用作定时

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM0_Isr() interrupt 1
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x00;                             //模式 0
    TL0 = 0x66;                             //65536-11.0592M/12/1000
    TH0 = 0xfc;
    TR0 = 1;                                //启动定时器
    ET0 = 1;                                //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz
```

P_SW2	DATA	0BAH
PIM1	DATA	091H
PIM0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H

<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>000BH</i>
	<i>LJMP</i>	<i>TM0ISR</i>
	<i>ORG</i>	<i>0100H</i>
TM0ISR:	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
START:		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>TMOD, #00H</i>
	<i>MOV</i>	; 模式 0 <i>TL0, #66H</i>
	<i>MOV</i>	; 65536-11.0592MHz/12/1000 <i>TH0, #0FCH</i>
	<i>SETB</i>	<i>TR0</i>
	<i>SETB</i>	; 启动定时器 <i>ET0</i>
	<i>SETB</i>	; 使能定时器中断 <i>EA</i>
	<i>JMP</i>	\$
 END		

16.7.2 定时器 0（模式 1—16 位不自动重载），用作定时

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void TM0_Isr() interrupt 1
{
    TL0 = 0x66;                                // 重设定时参数
    TH0 = 0xfc;
}
```

```

P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                          //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x01;                            //模式 I
    TL0 = 0x66;                            //65536-11.0592M/12/1000
    TH0 = 0xfc;
    TR0 = 1;                               //启动定时器
    ET0 = 1;                               //使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>000BH</i>	
<i>LJMP</i>	<i>TM0ISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>TM0ISR:</i>	<i>MOV</i>	<i>TL0,#66H</i>
	<i>MOV</i>	<i>TH0,#0FCH</i> ;重设定时参数
	<i>CPL</i>	<i>P1.0</i> ;测试端口

RETI***START:***

```

MOV      SP, #5FH
ORL      P_SW2,#80H          ;使能访问 XFR, 没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      TMOD,#01H          ;模式 1
MOV      TL0,#66H            ;65536-11.0592M/12/1000
MOV      TH0,#0FCH
SETB    TR0                 ;启动定时器
SETB    ET0                 ;使能定时器中断
SETB    EA

JMP      $

```

END

16.7.3 定时器 0 (模式 2—8 位自动重载), 用作定时

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM0_Isr() interrupt 1
{
    P10 = !P10;                  //测试端口
}

void main()
{
    P_SW2 |= 0x80;              //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
}

```

```

P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

TMOD = 0x02;           //模式2
TL0 = 0xf4;            //256-II.0592M/12/76K
TH0 = 0xf4;
TR0 = 1;                //启动定时器
ET0 = 1;                //使能定时器中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH

P1M1      DATA      091H
P1M0      DATA      092H
P0M1      DATA      093H
P0M0      DATA      094H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP    START
        ORG      000BH
        LJMP    TM0ISR

        ORG      0100H
TM0ISR:
        CPL      P1.0          ; 测试端口
        RETI

START:
        MOV      SP, #5FH
        ORL      P_SW2, #80H      ; 使能访问 XFR，没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H

```

MOV	P5M1, #00H	
MOV	TMOD, #02H	; 模式 2
MOV	TL0, #0F4H	; 256-11.0592M/12/76K
MOV	TH0, #0F4H	
SETB	TR0	; 启动定时器
SETB	ET0	; 使能定时器中断
SETB	EA	
JMP	\$	
END		

16.7.4 定时器 0（模式 3—16 位自动重载不可屏蔽中断），用作定时

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM0_Isr() interrupt 1
{
    P10 = !P10;                                // 测试端口
}

void main()
{
    P_SW2 |= 0x80;                            // 使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x03;                                // 模式 3
    TL0 = 0x66;                                 // 65536-11.0592M/12/1000
    TH0 = 0xfc;
    TR0 = 1;                                    // 启动定时器
    ET0 = 1;                                    // 使能定时器中断
    EA = 1;                                     // 不受 EA 控制

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>000BH</i>	
<i>LJMP</i>	<i>TM0ISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>TM0ISR:</i>	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	
	; 使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>TMOD, #03H</i>	; 模式 3
<i>MOV</i>	<i>TL0, #66H</i>	; 65536-11.0592M/12/1000
<i>MOV</i>	<i>TH0, #0FCH</i>	
<i>SETB</i>	<i>TR0</i>	; 启动定时器
<i>SETB</i>	<i>ET0</i>	; 使能定时器中断
;	<i>SETB</i>	<i>EA</i>
	; 不受 EA 控制	
<i>JMP</i>	\$	
 <i>END</i>		

16.7.5 定时器 0 (外部计数—扩展 T0 为外部下降沿中断)

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM0_Isr() interrupt 1
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x04;                            //外部计数模式
    TL0 = 0xff;
    TH0 = 0xff;
    TR0 = 1;                                //启动定时器
    ET0 = 1;                                //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz
```

P_SW2	DATA	0BAH
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H

<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>000BH</i>	
	<i>LJMP</i>	<i>TM0ISR</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>TM0ISR:</i>	<i>CPL</i>	<i>P1.0</i>	; 测试端口
	<i>RETI</i>		
<i>START:</i>			
	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>MOV</i>	<i>TMOD,#04H</i>	; 外部计数模式
	<i>MOV</i>	<i>TL0,#0FFH</i>	
	<i>MOV</i>	<i>TH0,#0FFH</i>	
	<i>SETB</i>	<i>TR0</i>	; 启动定时器
	<i>SETB</i>	<i>ET0</i>	; 使能定时器中断
	<i>SETB</i>	<i>EA</i>	
	<i>JMP</i>	\$	
		<i>END</i>	

16.7.6 定时器 0 (测量脉宽—INT0 高电平宽度)

C 语言代码

```
// 测试工作频率为 11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"

void INT0_Isr() interrupt 0
{
    P0 = TL0;                                // TL0 为测量值低字节
    P1 = TH0;                                // TH0 为测量值高字节
    TL0 = 0x00;
    TH0 = 0x00;
}
```

```

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    AUXR = 0x80;                                  //IT 模式
    TMOD = 0x08;                                //使能 GATE,INT0 为 1 时使能计时
    TL0 = 0x00;
    TH0 = 0x00;
    while (P32);                                 //等待 INT0 为低
    TR0 = 1;                                    //启动定时器
    IT0 = 1;                                    //使能 INT0 下降沿中断
    EX0 = 1;
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0003H</i>	
<i>LJMP</i>	<i>INT0ISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>INT0ISR:</i>	<i>MOV</i>	<i>P0,TL0</i>
	; TL0 为测量值低字节	

MOV	P1,TH0	<i>;TH0 为测量值高字节</i>
MOV	TL0,#00H	
MOV	TH0,#00H	
RETI		

START:

MOV	SP, #5FH	
ORL	P_SW2,#80H	<i>;使能访问 XFR，没有冲突不用关闭</i>
MOV	P0M0, #00H	
MOV	P0M1, #00H	
MOV	P1M0, #00H	
MOV	P1M1, #00H	
MOV	P2M0, #00H	
MOV	P2M1, #00H	
MOV	P3M0, #00H	
MOV	P3M1, #00H	
MOV	P4M0, #00H	
MOV	P4M1, #00H	
MOV	P5M0, #00H	
MOV	P5M1, #00H	
MOV	AUXR,#80H	<i>;IT 模式</i>
MOV	TMOD,#08H	<i>;使能 GATE,INT0 为 1 时使能计时</i>
MOV	TL0,#00H	
MOV	TH0,#00H	
JB	P3.2,\$	<i>;等待 INT0 为低</i>
SETB	TR0	<i>;启动定时器</i>
SETB	IT0	<i>;使能 INT0 下降沿中断</i>
SETB	EX0	
SETB	EA	
JMP	\$	

END

16.7.7 定时器 0（模式 0），时钟分频输出

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
```

```

P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

TMOD = 0x00;                                //模式0
TL0 = 0x66;                                  //65536-11.0592M/12/1000
TH0 = 0xfc;                                   //启动定时器
TR0 = 1;                                     //使能时钟输出
INTCLKO = 0x01;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>INTCLKO</i>	<i>DATA</i>	<i>8FH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>
<i>ORG</i>		<i>0100H</i>
<i>START:</i>		
<i>MOV</i>		<i>SP, #5FH</i>
<i>ORL</i>		<i>P_SW2,#80H</i> ; 使能访问 XFR，没有冲突不用关闭
<i>MOV</i>		<i>P0M0, #00H</i>
<i>MOV</i>		<i>P0M1, #00H</i>
<i>MOV</i>		<i>P1M0, #00H</i>
<i>MOV</i>		<i>P1M1, #00H</i>
<i>MOV</i>		<i>P2M0, #00H</i>
<i>MOV</i>		<i>P2M1, #00H</i>
<i>MOV</i>		<i>P3M0, #00H</i>
<i>MOV</i>		<i>P3M1, #00H</i>
<i>MOV</i>		<i>P4M0, #00H</i>
<i>MOV</i>		<i>P4M1, #00H</i>
<i>MOV</i>		<i>P5M0, #00H</i>
<i>MOV</i>		<i>P5M1, #00H</i>
<i>MOV</i>		<i>TMOD, #00H</i> ; 模式0
<i>MOV</i>		<i>TL0, #66H</i> ; 65536-11.0592M/12/1000
<i>MOV</i>		<i>TH0, #0FCH</i>

<i>SETB</i>	<i>TR0</i>	;启动定时器
<i>MOV</i>	<i>INTCLKO,#01H</i>	;使能时钟输出
<i>JMP</i>	\$	
<i>END</i>		

16.7.8 定时器 1（模式 0—16 位自动重载），用作定时

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void TMI_Isr() interrupt 3
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x00;                             //模式 0
    TL1 = 0x66;                             //65536-11.0592M/12/1000
    TH1 = 0xfc;
    TR1 = 1;                                //启动定时器
    ET1 = 1;                                //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>

<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>001BH</i>
	<i>LJMP</i>	<i>TMIISR</i>
	<i>ORG</i>	<i>0100H</i>
TMIISR:	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
START:		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>TMOD,#00H</i>
	<i>MOV</i>	; 模式 0 TL1,#66H ; 65536-11.0592M/12/1000
	<i>MOV</i>	<i>TH1,#0FCH</i>
	<i>SETB</i>	<i>TR1</i>
	<i>SETB</i>	; 启动定时器 <i>ET1</i>
	<i>SETB</i>	; 使能定时器中断 <i>EA</i>
	<i>JMP</i>	\$
		END

16.7.9 定时器 1（模式 1—16 位不自动重载），用作定时

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
```

```
#include "intrins.h"

void TM1_Isr() interrupt 3
{
    TL1 = 0x66;                                //重设定时参数
    TH1 = 0xfc;
    P10 = !P10;                                 //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x10;                                //模式 I
    TL1 = 0x66;                                //65536-11.0592M/12/1000
    TH1 = 0xfc;
    TR1 = 1;                                    //启动定时器
    ET1 = 1;                                    //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG		0000H
LJMP		START
ORG		001BH
LJMP		TMIISR

ORG	0100H
------------	--------------

TMIISR:

MOV	TL1,#66H	;重设定时参数
MOV	TH1,#0FCH	
CPL	P1.0	;测试端口
RETI		

START:

MOV	SP, #5FH	
ORL	P_SW2,#80H	;使能访问 XFR，没有冲突不用关闭
MOV	P0M0, #00H	
MOV	P0M1, #00H	
MOV	P1M0, #00H	
MOV	P1M1, #00H	
MOV	P2M0, #00H	
MOV	P2M1, #00H	
MOV	P3M0, #00H	
MOV	P3M1, #00H	
MOV	P4M0, #00H	
MOV	P4M1, #00H	
MOV	P5M0, #00H	
MOV	P5M1, #00H	
MOV	TMOD,#10H	;模式 1
MOV	TL1,#66H	;65536-11.0592M/12/1000
MOV	TH1,#0FCH	
SETB	TR1	;启动定时器
SETB	ET1	;使能定时器中断
SETB	EA	
JMP	\$	

END

16.7.10 定时器 1（模式 2—8 位自动重载），用作定时

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM1_Isr() interrupt 3
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR，没有冲突不用关闭
    P0M0 = 0x00;
    P0M1 = 0x00;
```

```

P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

TMOD = 0x20;           //模式2
TLI = 0xf4;            //256-11.0592M/12/76K
THI = 0xf4;
TR1 = 1;                //启动定时器
ET1 = 1;                //使能定时器中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>001BH</i>	
<i>LJMP</i>	<i>TMIISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>TMIISR:</i>	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	
	; 使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	

```

MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      TMOD,#20H          ;模式2
MOV      TL1,#0F4H           ;256-11.0592M/12/76K
MOV      TH1,#0F4H
SETB    TR1                 ;启动定时器
SETB    ET1                 ;使能定时器中断
SETB    EA

JMP     $

END

```

16.7.11 定时器 1 (外部计数—扩展 T1 为外部下降沿中断)

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM1_Isr() interrupt 3
{
    P10 = !P10;                      //测试端口
}

void main()
{
    P_SW2 |= 0x80;                  //使能访问XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x40;                    //外部计数模式
    TL1 = 0xff;
    TH1 = 0xff;
    TR1 = 1;                        //启动定时器
    ET1 = 1;                        //使能定时器中断
    EA = 1;
}

```

```

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>001BH</i>	
<i>LJMP</i>	<i>TMIISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>TMIISR:</i>	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	
	; 使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>TMOD,#40H</i>	
<i>MOV</i>	<i>TL1,#0FFH</i>	
<i>MOV</i>	<i>TH1,#0FFH</i>	
<i>SETB</i>	<i>TR1</i>	
<i>SETB</i>	<i>ET1</i>	
<i>SETB</i>	<i>EA</i>	
	; 启动定时器	
	; 使能定时器中断	
<i>JMP</i>	\$	

END

16.7.12 定时器 1 (测量脉宽—INT1 高电平宽度)

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void INT1_Isr() interrupt 2
{
    P0 = TL1;                                //TL1 为测量值低字节
    P1 = TH1;                                //TH1 为测量值高字节
    TL1 = 0x00;
    TH1 = 0x00;
}

void main()
{
    P_SW2 |= 0x80;                           //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    AUXR = 0x40;                            //IT 模式
    TMOD = 0x80;                            //使能 GATE,INT1 为 1 时使能计时
    TL1 = 0x00;
    TH1 = 0x00;
    while (P33);                           //等待 INT1 为低
    TR1 = 1;                                //启动定时器
    IT1 = 1;                                //使能 INT1 下降沿中断
    EX1 = 1;
    EA = 1;

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
AUXR	DATA	8EH

<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0013H</i>	
<i>LJMP</i>	<i>INTIISR</i>	
<i>ORG</i>	<i>0100H</i>	
INTIISR:		
<i>MOV</i>	<i>P0,TL1</i>	<i>;TL1</i> 为测量值低字节
<i>MOV</i>	<i>P1,TH1</i>	<i>;TH1</i> 为测量值高字节
<i>MOV</i>	<i>TL1,#00H</i>	
<i>MOV</i>	<i>TH1,#00H</i>	
<i>RETI</i>		
START:		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	<i>;使能访问 XFR，没有冲突不用关闭</i>
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>AUXR,#40H</i>	<i>;IT 模式</i>
<i>MOV</i>	<i>TMOD,#80H</i>	<i>;使能 GATE, INT1 为 1 时使能计时</i>
<i>MOV</i>	<i>TL1,#00H</i>	
<i>MOV</i>	<i>TH1,#00H</i>	
<i>JB</i>	<i>P3.3,\$</i>	<i>;等待 INT1 为低</i>
<i>SETB</i>	<i>TR1</i>	<i>;启动定时器</i>
<i>SETB</i>	<i>IT1</i>	<i>;使能 INT1 下降沿中断</i>
<i>SETB</i>	<i>EX1</i>	
<i>SETB</i>	<i>EA</i>	
<i>JMP</i>	<i>\$</i>	
 END		

16.7.13 定时器 1（模式 0），时钟分频输出

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    TMOD = 0x00;                                  //模式 0
    TL1 = 0x66;                                   //65536-11.0592M/12/1000
    TH1 = 0xfc;
    TR1 = 1;                                     //启动定时器
    INTCLKO = 0x02;                               //使能时钟输出

    while (1);
}
```

汇编代码

```
; 测试工作频率为 11.0592MHz
```

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>INTCLKO</i>	<i>DATA</i>	<i>8FH</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>

```

        ORG      0100H
START:
        MOV      SP, #5FH
        ORL      P_SW2,#80H           ;使能访问 XFR, 没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        MOV      TMOD,#00H          ;模式0
        MOV      TLI,#66H            ;65536-11.0592M/12/1000
        MOV      TH1,#0FCH
        SETB    TR1                 ;启动定时器
        MOV      INTCLKO,#02H        ;使能时钟输出

        JMP      $

```

END

16.7.14 定时器 1（模式 0）做串口 1 波特率发生器

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)           //加2 操作是为了让 Keil 编译器
                                                               //自动实现四舍五入运算

bit     busy;
char   wptr;
char   rptr;
char   buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {

```

```
RI = 0;
buffer[wptr++] = SBUF;
wptr &= 0x0f;
}
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x00;
    TLI = BRT;
    TH1 = BRT >> 8;
    TR1 = 1;
    AUXR = 0x40;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
    {
        UartSend(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    ES = 1;
    EA = 1;
    UartSendStr("Uart Test !r\n");

    while (1)
    {
        if (rptr != wptr)
```

```

    {
        UartSend(buffer[rptr++]);
        rptr &= 0x0f;
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	DATA	0BAH
<i>AUXR</i>	DATA	8EH
<i>BUSY</i>	BIT	20H.0
<i>WPTR</i>	DATA	21H
<i>RPTR</i>	DATA	22H
<i>BUFFER</i>	DATA	23H
		<i>;16 bytes</i>
<i>P1M1</i>	DATA	091H
<i>P1M0</i>	DATA	092H
<i>P0M1</i>	DATA	093H
<i>P0M0</i>	DATA	094H
<i>P2M1</i>	DATA	095H
<i>P2M0</i>	DATA	096H
<i>P3M1</i>	DATA	0B1H
<i>P3M0</i>	DATA	0B2H
<i>P4M1</i>	DATA	0B3H
<i>P4M0</i>	DATA	0B4H
<i>P5M1</i>	DATA	0C9H
<i>P5M0</i>	DATA	0CAH
	ORG	0000H
	LJMP	START
	ORG	0023H
	LJMP	UART_ISR
	ORG	0100H
UART_ISR:		
	PUSH	ACC
	PUSH	PSW
	MOV	PSW,#08H
	JNB	TI,CHKRI
	CLR	TI
	CLR	BUSY
CHKRI:		
	JNB	RI,UARTISR_EXIT
	CLR	RI
	MOV	A,WPTR
	ANL	A,#0FH
	ADD	A,#BUFFER
	MOV	R0,A
	MOV	@R0,SBUF
	INC	WPTR
UARTISR_EXIT:		
	POP	PSW

POP *ACC*
RETI

UART_INIT:

<i>MOV</i>	<i>SCON,#50H</i>
<i>MOV</i>	<i>TMOD,#00H</i>
<i>MOV</i>	<i>TL1,#0E8H</i>
<i>MOV</i>	<i>TH1,#0FFH</i>
<i>SETB</i>	<i>TR1</i>
<i>MOV</i>	<i>AUXR,#40H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

;65536-11059200/115200/4=0FFE8H

UART_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>SBUF,A</i>
<i>RET</i>	

UART_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SENDEND</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART_SENDSTR</i>

SENDEND:

<i>RET</i>	
------------	--

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	;使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART_INIT</i>
<i>SETB</i>	<i>ES</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
<i>XRL</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>

```

JZ           LOOP
MOV          A,RPTR
ANL          A,#0FH
ADD          A,#BUFFER
MOV          R0,A
MOV          A,@R0
LCALL       UART_SEND
INC          RPTR
JMP          LOOP

STRING:     DB           'Uart Test !',0DH,0AH,00H

END

```

16.7.15 定时器 1（模式 2）做串口 1 波特率发生器

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (256 - (FOSC / 115200+16) / 32)
                                         //加16 操作是为了让Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
        buffer[wptr++] = SBUF;
        wptr &= 0x0f;
    }
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x20;
    TL1 = BRT;
    TH1 = BRT;
    TR1 = 1;
    AUXR = 0x40;
}

```

```

wptr = 0x00;
rptr = 0x00;
busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
    {
        UartSend(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    ES = 1;
    EA = 1;
    UartSendStr("Uart Test !\r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            UartSend(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2 DATA 0BAH

AUXR DATA 8EH

<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>

;16 bytes

<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>

<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>
<i>ORG</i>	<i>0023H</i>
<i>LJMP</i>	<i>UART_ISR</i>
 <i>ORG</i>	 <i>0100H</i>

UART_ISR:

<i>PUSH</i>	<i>ACC</i>
<i>PUSH</i>	<i>PSW</i>
<i>MOV</i>	<i>PSW,#08H</i>
 <i>JNB</i>	<i>TI,CHKRI</i>
<i>CLR</i>	<i>TI</i>
<i>CLR</i>	<i>BUSY</i>

CHKRI:

<i>JNB</i>	<i>RI,UARTISR_EXIT</i>
<i>CLR</i>	<i>RI</i>
<i>MOV</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>@R0,SBUF</i>
<i>INC</i>	<i>WPTR</i>

UARTISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART_INIT:

<i>MOV</i>	<i>SCON,#50H</i>
<i>MOV</i>	<i>TMOD,#20H</i>
<i>MOV</i>	<i>TL1,#0FDH</i>
<i>MOV</i>	<i>TH1,#0FDH</i>
<i>SETB</i>	<i>TR1</i>
<i>MOV</i>	<i>AUXR,#40H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

;256-11059200/115200/32=0FDH

UART_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>SBUF,A</i>
<i>RET</i>	

UART_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SENDEND</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART_SENDSTR</i>

SENDEND:

<i>RET</i>

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART_INIT</i>
<i>SETB</i>	<i>ES</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
<i>XRL</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>JZ</i>	<i>LOOP</i>
<i>MOV</i>	<i>A,RPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>A,@R0</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>RPTR</i>
<i>JMP</i>	<i>LOOP</i>

<i>STRING:</i>	<i>DB</i>	<i>'Uart Test !',0DH,0AH,00H</i>
-----------------------	------------------	---

<i>END</i>

16.7.16 定时器 2 (16 位自动重载) , 用作定时

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM2_Isr() interrupt 12
{
    P10 = !P10;                                //测试端口
}

void main()
{
    P_SW2 |= 0x80;                            //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T2L = 0x66;                                //65536-11.0592M/12/1000
    T2H = 0xfc;
    AUXR = 0x10;                                //启动定时器
    IE2 = ET2;                                  //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz
```

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
IE2	DATA	0AFH
ET2	EQU	04H
AUXINTIF	DATA	0EFH
T2IF	EQU	01H
P1M1	DATA	091H
P1M0	DATA	092H

<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0063H</i>
	<i>LJMP</i>	<i>TM2ISR</i>
	<i>ORG</i>	<i>0100H</i>
TM2ISR:	<i>CPL</i>	<i>P1.0</i>
	<i>RETI</i>	; 测试端口
START:		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>MOV</i>	<i>T2L,#66H</i>
	<i>MOV</i>	; 65536-11.0592M/12/1000
	<i>MOV</i>	<i>T2H,#0FCH</i>
	<i>MOV</i>	<i>AUXR,#10H</i>
	<i>MOV</i>	; 启动定时器
	<i>MOV</i>	<i>IE2,#ET2</i>
	<i>SETB</i>	; 使能定时器中断
		<i>EA</i>
	<i>JMP</i>	\$
	END	

16.7.17 定时器 2 (外部计数—扩展 T2 为外部下降沿中断)

C 语言代码

```
// 测试工作频率为 11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"
```

```

void TM2_Isr() interrupt 12
{
    P10 = !P10;                                // 测试端口
}

void main()
{
    P_SW2 |= 0x80;                            // 使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T2L = 0xff;
    T2H = 0xff;
    AUXR = 0x18;                                // 设置外部计数模式并启动定时器
    IE2 = ET2;                                  // 使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
IE2	DATA	0AFH
ET2	EQU	04H
AUXINTIF	DATA	0EFH
T2IF	EQU	01H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH

<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0063H</i>	
<i>LJMP</i>	<i>TM2ISR</i>	
<i>ORG</i>	<i>0100H</i>	
<i>TM2ISR:</i>	<i>CPL</i>	<i>P1.0</i> ; 测试端口
	<i>RETI</i>	
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>T2L,#0FFH</i>	
<i>MOV</i>	<i>T2H,#0FFH</i>	
<i>MOV</i>	<i>AUXR,#18H</i>	; 设置外部计数模式并启动定时器
<i>MOV</i>	<i>IE2,#ET2</i>	; 使能定时器中断
<i>SETB</i>	<i>EA</i>	
<i>JMP</i>	\$	
 <i>END</i>		

16.7.18 定时器 2, 时钟分频输出

C 语言代码

```
// 测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80; // 使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
```

```

P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

T2L = 0x66;                                //65536-11.0592M/12/1000
T2H = 0xfc;                                 //启动定时器
AUXR = 0x10;                               //使能时钟输出
INTCLKO = 0x04;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
INTCLKO	DATA	8FH
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG		0000H
LJMP		START
ORG		0100H
START:		
MOV		SP, #5FH
ORL		P_SW2,#80H ; 使能访问 XFR，没有冲突不用关闭
MOV		P0M0, #00H
MOV		P0M1, #00H
MOV		P1M0, #00H
MOV		P1M1, #00H
MOV		P2M0, #00H
MOV		P2M1, #00H
MOV		P3M0, #00H
MOV		P3M1, #00H
MOV		P4M0, #00H
MOV		P4M1, #00H
MOV		P5M0, #00H
MOV		P5M1, #00H

```

MOV      T2L,#66H           ;65536-11.0592M/12/1000
MOV      T2H,#0FCH
MOV      AUXR,#10H          ;启动定时器
MOV      INTCLKO,#04H        ;使能时钟输出

JMP      $

```

END

16.7.19 定时器 2 做串口 1 波特率发生器

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
        buffer[wptr++] = SBUF;
        wptr &= 0x0f;
    }
}

void UartInit()
{
    SCON = 0x50;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x15;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void UartSend(char dat)

```

```

{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
    {
        UartSend(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    ES = 1;
    EA = 1;
    UartSendStr("Uart Test !\r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            UartSend(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
AUXR	DATA	8EH
T2H	DATA	0D6H
T2L	DATA	0D7H
BUSY	BIT	20H.0
WPTR	DATA	21H
RPTR	DATA	22H

BUFFER	DATA	23H	;16 bytes
---------------	-------------	------------	------------------

P1M1	DATA	091H
-------------	-------------	-------------

P1M0	DATA	092H
-------------	-------------	-------------

P0M1	DATA	093H
-------------	-------------	-------------

P0M0	DATA	094H
-------------	-------------	-------------

P2M1	DATA	095H
-------------	-------------	-------------

P2M0	DATA	096H
-------------	-------------	-------------

P3M1	DATA	0B1H
-------------	-------------	-------------

P3M0	DATA	0B2H
-------------	-------------	-------------

P4M1	DATA	0B3H
-------------	-------------	-------------

P4M0	DATA	0B4H
-------------	-------------	-------------

P5M1	DATA	0C9H
-------------	-------------	-------------

P5M0	DATA	0CAH
-------------	-------------	-------------

ORG	0000H
------------	--------------

LJMP	START
-------------	--------------

ORG	0023H
------------	--------------

LJMP	UART_ISR
-------------	-----------------

ORG	0100H
------------	--------------

UART_ISR:

PUSH	ACC
-------------	------------

PUSH	PSW
-------------	------------

MOV	PSW,#08H
------------	-----------------

JNB	TI,CHKRI
------------	-----------------

CLR	TI
------------	-----------

CLR	BUSY
------------	-------------

CHKRI:

JNB	RI,UARTISR_EXIT
------------	------------------------

CLR	RI
------------	-----------

MOV	A,WPTR
------------	---------------

ANL	A,#0FH
------------	---------------

ADD	A,#BUFFER
------------	------------------

MOV	R0,A
------------	-------------

MOV	@R0,SBUF
------------	-----------------

INC	WPTR
------------	-------------

UARTISR_EXIT:

POP	PSW
------------	------------

POP	ACC
------------	------------

RETI	
-------------	--

UART_INIT:

MOV	SCON,#50H
------------	------------------

MOV	T2L,#0E8H	;65536-11059200/115200/4=0FFE8H
------------	------------------	--

MOV	T2H,#0FFH
------------	------------------

MOV	AUXR,#15H
------------	------------------

CLR	BUSY
------------	-------------

MOV	WPTR,#00H
------------	------------------

MOV	RPTR,#00H
------------	------------------

RET	
------------	--

UART_SEND:

JB	BUSY,\$
-----------	----------------

SETB	BUSY
-------------	-------------

MOV	SBUF,A
------------	---------------

RET	
------------	--

UART_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SENDEND</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART_SENDSTR</i>

SENDEND:

<i>RET</i>

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
;使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART_INIT</i>
<i>SETB</i>	<i>ES</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
<i>XRL</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>JZ</i>	<i>LOOP</i>
<i>MOV</i>	<i>A,RPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>A,@R0</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>RPTN</i>
<i>JMP</i>	<i>LOOP</i>

<i>STRING:</i>	<i>DB</i>	<i>'Uart Test !',0DH,0AH,00H</i>
----------------	-----------	----------------------------------

<i>END</i>

16.7.20 定时器 2 做串口 2 波特率发生器

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2操作是为了让Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void Uart2Isr() interrupt 8
{
    if(S2CON & 0x02)
    {
        S2CON &= ~0x02;
        busy = 0;
    }
    if(S2CON & 0x01)
    {
        S2CON &= ~0x01;
        buffer[wptr++] = S2BUF;
        wptr &= 0x0f;
    }
}

void Uart2Init()
{
    S2CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart2Send(char dat)
{
    while (busy);
    busy = 1;
    S2BUF = dat;
}

void Uart2SendStr(char *p)
{
    while (*p)
    {
        Uart2Send(*p++);
    }
}

void main()
{
```

```
P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭
```

```
P0M0 = 0x00;
P0M1 = 0x00;
P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

Uart2Init();
IE2 = 0x01;
EA = 1;
Uart2SendStr("Uart Test !\r\n");

while (1)
{
    if (rptr != wptr)
    {
        Uart2Send(buffer[rptr++]);
        rptr &= 0x0f;
    }
}
}
```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>S2CON</i>	<i>DATA</i>	<i>9AH</i>
<i>S2BUF</i>	<i>DATA</i>	<i>9BH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		;16 bytes
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>

P5M0 **DATA** **0CAH**

ORG **0000H**
LJMP **START**
ORG **0043H**
LJMP **UART2_ISR**

ORG **0100H**

UART2_ISR:

PUSH **ACC**
PUSH **PSW**
MOV **PSW,#08H**

MOV **A,S2CON**
JNB **ACC.I,CHKRI**
ANL **S2CON,#NOT 02H**
CLR **BUSY**

CHKRI:

JNB **ACC.0,UART2ISR_EXIT**
ANL **S2CON,#NOT 01H**
MOV **A,WPTR**
ANL **A,#0FH**
ADD **A,#BUFFER**
MOV **R0,A**
MOV **@R0,S2BUF**
INC **WPTR**

UART2ISR_EXIT:

POP **PSW**
POP **ACC**
RETI

UART2_INIT:

MOV **S2CON,#10H**
MOV **T2L,#0E8H** ;65536-11059200/115200/4=0FFE8H
MOV **T2H,#0FFH**
MOV **AUXR,#14H**
CLR **BUSY**
MOV **WPTR,#00H**
MOV **RPTR,#00H**
RET

UART2_SEND:

JB **BUSY,\$**
SETB **BUSY**
MOV **S2BUFA**
RET

UART2_SENDSTR:

CLR **A**
MOVC **A,@A+DPTR**
JZ **SEND2END**
LCALL **UART2_SEND**
INC **DPTR**
JMP **UART2_SENDSTR**

SEND2END:

RET

START:

```

MOV      SP, #5FH
ORL      P_SW2,#80H          ;使能访问 XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL   UART2_INIT
MOV      IE2,#01H
SETB    EA

MOV      DPTR,#STRING
LCALL   UART2_SENDSTR

LOOP:
MOV      A,RPTR
XRL      A,WPTR
ANL      A,#0FH
JZ       LOOP
MOV      A,RPTR
ANL      A,#0FH
ADD      A,#BUFFER
MOV      R0,A
MOV      A,@R0
LCALL   UART2_SEND
INC      RPT
JMP      LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

END

```

16.7.21 定时器 2 做串口 3 波特率发生器

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

#define  FOSC      11059200UL
#define  BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;

```

```
char      rptr;
char      buffer[16];

void Uart3Isr() interrupt 17
{
    if (S3CON & 0x02)
    {
        S3CON &= ~0x02;
        busy = 0;
    }
    if (S3CON & 0x01)
    {
        S3CON &= ~0x01;
        buffer[wptr++] = S3BUF;
        wptr &= 0x0f;
    }
}

void Uart3Init()
{
    S3CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart3Send(char dat)
{
    while (busy);
    busy = 1;
    S3BUF = dat;
}

void Uart3SendStr(char *p)
{
    while (*p)
    {
        Uart3Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
```

P5M1 = 0x00;

```

Uart3Init();
IE2 = 0x08;
EA = 1;
Uart3SendStr("Uart Test !\r\n");

while (1)
{
    if (rptr != wptr)
    {
        Uart3Send(buffer[rptr++]);
        rptr &= 0x0f;
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>S3CON</i>	<i>DATA</i>	<i>0ACh</i>
<i>S3BUF</i>	<i>DATA</i>	<i>0ADH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		; 16 bytes
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>DATA</i>	<i>0000H</i>
<i>LJMP</i>	<i>DATA</i>	<i>START</i>
<i>ORG</i>	<i>DATA</i>	<i>008BH</i>
<i>LJMP</i>	<i>DATA</i>	<i>UART3_ISR</i>
<i>ORG</i>	<i>DATA</i>	<i>0100H</i>
<i>UART3_ISR:</i>		
<i>PUSH</i>	<i>DATA</i>	<i>ACC</i>
<i>PUSH</i>	<i>DATA</i>	<i>PSW</i>
<i>MOV</i>	<i>DATA</i>	<i>PSW,#08H</i>

MOV	<i>A,S3CON</i>	
JNB	<i>ACC.I,CHKRI</i>	
ANL	<i>S3CON,#NOT 02H</i>	
CLR	<i>BUSY</i>	
CHKRI:		
JNB	<i>ACC.0,UART3ISR_EXIT</i>	
ANL	<i>S3CON,#NOT 01H</i>	
MOV	<i>A,WPTR</i>	
ANL	<i>A,#0FH</i>	
ADD	<i>A,#BUFFER</i>	
MOV	<i>R0,A</i>	
MOV	<i>@R0,S3BUF</i>	
INC	<i>WPTR</i>	
UART3ISR_EXIT:		
POP	<i>PSW</i>	
POP	<i>ACC</i>	
RETI		
UART3_INIT:		
MOV	<i>S3CON,#10H</i>	
MOV	<i>T2L,#0E8H</i>	<i>;65536-11059200/115200/4=0FFE8H</i>
MOV	<i>T2H,#0FFH</i>	
MOV	<i>AUXR,#I4H</i>	
CLR	<i>BUSY</i>	
MOV	<i>WPTR,#00H</i>	
MOV	<i>RPTR,#00H</i>	
RET		
UART3_SEND:		
JB	<i>BUSY,\$</i>	
SETB	<i>BUSY</i>	
MOV	<i>S3BUFA,A</i>	
RET		
UART3_SENDSTR:		
CLR	<i>A</i>	
MOVC	<i>A,@A+DPTR</i>	
JZ	<i>SEND3END</i>	
LCALL	<i>UART3_SEND</i>	
INC	<i>DPTR</i>	
JMP	<i>UART3_SENDSTR</i>	
SEND3END:		
RET		
START:		
MOV	<i>SP, #5FH</i>	
ORL	<i>P_SW2,#80H</i>	<i>;使能访问 XFR，没有冲突不用关闭</i>
MOV	<i>P0M0, #00H</i>	
MOV	<i>P0M1, #00H</i>	
MOV	<i>P1M0, #00H</i>	
MOV	<i>P1M1, #00H</i>	
MOV	<i>P2M0, #00H</i>	
MOV	<i>P2M1, #00H</i>	
MOV	<i>P3M0, #00H</i>	
MOV	<i>P3M1, #00H</i>	
MOV	<i>P4M0, #00H</i>	
MOV	<i>P4M1, #00H</i>	

```

MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL    UART3_INIT
MOV      IE2,#08H
SETB    EA

MOV      DPTR,#STRING
LCALL    UART3_SENDSTR

LOOP:
MOV      A,RPTR
XRL      A,WPTR
ANL      A,#0FH
JZ       LOOP
MOV      A,RPTR
ANL      A,#0FH
ADD      A,#BUFFER
MOV      R0,A
MOV      A,@R0
LCALL    UART3_SEND
INC      RPT
JMP      LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

END

```

16.7.22 定时器 2 做串口 4 波特率发生器

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2操作是为了让Keil编译器
                                         //自动实现四舍五入运算

bit      busy;
char    wptr;
char    rptr;
char    buffer[16];

void Uart4Isr() interrupt 18
{
    if (S4CON & 0x02)
    {
        S4CON &= ~0x02;
        busy = 0;
    }
    if (S4CON & 0x01)
    {

```

```
S4CON &= ~0x01;
buffer[wptr++] = S4BUF;
wptr &= 0x0f;
}
}

void Uart4Init()
{
    S4CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart4Send(char dat)
{
    while (busy);
    busy = 1;
    S4BUF = dat;
}

void Uart4SendStr(char *p)
{
    while (*p)
    {
        Uart4Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    Uart4Init();
    IE2 = 0x10;
    EA = 1;
    Uart4SendStr("Uart Test !\r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            Uart4Send(buffer[rptr++]);
        }
    }
}
```

```

        rptr &= 0x0f;
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>S4CON</i>	<i>DATA</i>	<i>84H</i>
<i>S4BUF</i>	<i>DATA</i>	<i>85H</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		<i>;16 bytes</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0093H</i>
	<i>LJMP</i>	<i>UART4_ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>UART4_ISR:</i>		
	<i>PUSH</i>	<i>ACC</i>
	<i>PUSH</i>	<i>PSW</i>
	<i>MOV</i>	<i>PSW,#08H</i>
	<i>MOV</i>	<i>A,S4CON</i>
	<i>JNB</i>	<i>ACC.I,CHKRI</i>
	<i>ANL</i>	<i>S4CON,#NOT 02H</i>
	<i>CLR</i>	<i>BUSY</i>
<i>CHKRI:</i>		
	<i>JNB</i>	<i>ACC.0,UART4ISR_EXIT</i>
	<i>ANL</i>	<i>S4CON,#NOT 01H</i>
	<i>MOV</i>	<i>A,WPTR</i>
	<i>ANL</i>	<i>A,#0FH</i>
	<i>ADD</i>	<i>A,#BUFFER</i>
	<i>MOV</i>	<i>R0,A</i>

```

        MOV      @R0,S4BUF
        INC      WPTR
UART4ISR_EXIT:
        POP      PSW
        POP      ACC
        RETI

UART4_INIT:
        MOV      S4CON,#10H
        MOV      T2L,#0E8H           ;65536-11059200/115200/4=0FFE8H
        MOV      T2H,#0FFH
        MOV      AUXR,#14H
        CLR      BUSY
        MOV      WPTR,#00H
        MOV      RPTR,#00H
        RET

UART4_SEND:
        JB      BUSY,$
        SETB    BUSY
        MOV      S4BUFA,A
        RET

UART4_SENDSTR:
        CLR      A
        MOVC   A,@A+DPTR
        JZ      SEND4END
        LCALL  UART4_SEND
        INC     DPTR
        JMP     UART4_SENDSTR
SEND4END:
        RET

START:
        MOV      SP,#5FH
        ORL      P_SW2,#80H          ;使能访问 XFR，没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        LCALL  UART4_INIT
        MOV      IE2,#10H
        SETB    EA

        MOV      DPTR,#STRING
        LCALL  UART4_SENDSTR

LOOP:
        MOV      A,RPTR

```

<i>XRL</i>	A,WPTR
<i>ANL</i>	A,#0FH
<i>JZ</i>	LOOP
<i>MOV</i>	A,RPTR
<i>ANL</i>	A,#0FH
<i>ADD</i>	A,#BUFFER
<i>MOV</i>	R0,A
<i>MOV</i>	A,@R0
<i>LCALL</i>	UART4_SEND
<i>INC</i>	RPTR
<i>JMP</i>	LOOP

STRING: *DB* 'Uart Test !',0DH,0AH,00H

END

16.7.23 定时器 3 (16 位自动重载) , 用作定时

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM3_Isr() interrupt 19
{
    P10 = !P10; //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T3L = 0x66; //65536-11.0592M/12/1000
    T3H = 0xfc;
    T4T3M = 0x08; //启动定时器
    IE2 = ET3; //使能定时器中断
    EA = 1;

    while (1);
}
```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH

T4T3M      DATA      0DIH
T4L        DATA      0D3H
T4H        DATA      0D2H
T3L        DATA      0D5H
T3H        DATA      0D4H
T2L        DATA      0D7H
T2H        DATA      0D6H
AUXR       DATA      8EH
IE2         DATA      0AFH
ET2         EQU       04H
ET3         EQU       20H
ET4         EQU       40H
AUXINTIF   DATA      0EFH
T2IF        EQU       01H
T3IF        EQU       02H
T4IF        EQU       04H

P1M1       DATA      091H
P1M0       DATA      092H
P0M1       DATA      093H
P0M0       DATA      094H
P2M1       DATA      095H
P2M0       DATA      096H
P3M1       DATA      0B1H
P3M0       DATA      0B2H
P4M1       DATA      0B3H
P4M0       DATA      0B4H
P5M1       DATA      0C9H
P5M0       DATA      0CAH

ORG        ORG       0000H
LJMP       LJMP      START
ORG        ORG       009BH
LJMP       LJMP      TM3ISR

ORG        ORG       0100H
TM3ISR:
CPL        CPL       P1.0           ; 测试端口
RETI       RETI

START:
MOV        MOV       SP, #5FH
ORL        ORL       P_SW2, #80H      ; 使能访问 XFR，没有冲突不用关闭

MOV        MOV       P0M0, #00H
MOV        MOV       P0M1, #00H
MOV        MOV       P1M0, #00H
MOV        MOV       P1M1, #00H
MOV        MOV       P2M0, #00H
MOV        MOV       P2M1, #00H
MOV        MOV       P3M0, #00H
MOV        MOV       P3M1, #00H
MOV        MOV       P4M0, #00H
MOV        MOV       P4M1, #00H

```

```

MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      T3L,#66H           ;65536-11.0592M/12/1000
MOV      T3H,#0FCH
MOV      T4T3M,#08H         ;启动定时器
MOV      IE2,#ET3            ;使能定时器中断
SETB    EA

JMP      $

END

```

16.7.24 定时器 3（外部计数—扩展 T3 为外部下降沿中断）

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM3_Isr() interrupt 19
{
    P10 = !P10;                //测试端口
}

void main()
{
    P_SW2 |= 0x80;             //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T3L = 0xff;
    T3H = 0xff;
    T4T3M = 0x0c;              //设置外部计数模式并启动定时器
    IE2 = ET3;                  //使能定时器中断
    EA = 1;

    while (1);
}

```

汇编代码

```
; 测试工作频率为 11.0592MHz
```

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T4T3M</i>	<i>DATA</i>	<i>0DIH</i>
<i>T4L</i>	<i>DATA</i>	<i>0D3H</i>
<i>T4H</i>	<i>DATA</i>	<i>0D2H</i>
<i>T3L</i>	<i>DATA</i>	<i>0D5H</i>
<i>T3H</i>	<i>DATA</i>	<i>0D4H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>ET2</i>	<i>EQU</i>	<i>04H</i>
<i>ET3</i>	<i>EQU</i>	<i>20H</i>
<i>ET4</i>	<i>EQU</i>	<i>40H</i>
<i>AUXINTIF</i>	<i>DATA</i>	<i>0EFH</i>
<i>T2IF</i>	<i>EQU</i>	<i>01H</i>
<i>T3IF</i>	<i>EQU</i>	<i>02H</i>
<i>T4IF</i>	<i>EQU</i>	<i>04H</i>
<i>PIM1</i>	<i>DATA</i>	<i>091H</i>
<i>PIM0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>009BH</i>
	<i>LJMP</i>	<i>TM3ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>TM3ISR:</i>	<i>CPL</i>	<i>PI.0</i>
	<i>RETI</i>	; 测试端口
<i>START:</i>		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>

```

MOV      T3L,#0FFH
MOV      T3H,#0FFH
MOV      T4T3M,#0CH          ;设置外部计数模式并启动定时器
MOV      IE2,#ET3            ;使能定时器中断
SETB    EA

JMP      $

END

```

16.7.25 定时器 3，时钟分频输出

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;           //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    T3L = 0x66;             //65536-11.0592M/12/1000
    T3H = 0xfc;
    T4T3M = 0x09;           //使能时钟输出并启动定时器

    while (1);
}

```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T4T3M</i>	<i>DATA</i>	<i>0DIH</i>
<i>T4L</i>	<i>DATA</i>	<i>0D3H</i>
<i>T4H</i>	<i>DATA</i>	<i>0D2H</i>
<i>T3L</i>	<i>DATA</i>	<i>0D5H</i>
<i>T3H</i>	<i>DATA</i>	<i>0D4H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>

<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0100H</i>	
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i> ;使能访问 XFR，没有冲突不用关闭	
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>T3L,#66H</i> ;65536-11.0592M/12/1000	
<i>MOV</i>	<i>T3H,#0FCH</i>	
<i>MOV</i>	<i>T4T3M,#09H</i> ;使能时钟输出并启动定时器	
<i>JMP</i>	\$	
<i>END</i>		

16.7.26 定时器 3 做串口 3 波特率发生器

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
//加2操作是为了让Keil编译器
```

//自动实现四舍五入运算

```
bit      busy;
char     wptr;
char     rptr;
char     buffer[16];

void Uart3Isr() interrupt 17
{
    if(S3CON & 0x02)
    {
        S3CON &= ~0x02;
        busy = 0;
    }
    if(S3CON & 0x01)
    {
        S3CON &= ~0x01;
        buffer[wptr++] = S3BUF;
        wptr &= 0x0f;
    }
}

void Uart3Init()
{
    S3CON = 0x50;
    T3L = BRT;
    T3H = BRT >> 8;
    T4T3M = 0x0a;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart3Send(char dat)
{
    while (busy);
    busy = 1;
    S3BUF = dat;
}

void Uart3SendStr(char *p)
{
    while (*p)
    {
        Uart3Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
```

```

P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

Uart3Init();
IE2 = 0x08;
EA = 1;
Uart3SendStr("Uart Test !r\n");

while (1)
{
    if (rptr != wptr)
    {
        Uart3Send(buffer[rptr++]);
        rptr &= 0x0f;
    }
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T4T3M</i>	<i>DATA</i>	<i>0DIH</i>
<i>T4L</i>	<i>DATA</i>	<i>0D3H</i>
<i>T4H</i>	<i>DATA</i>	<i>0D2H</i>
<i>T3L</i>	<i>DATA</i>	<i>0D5H</i>
<i>T3H</i>	<i>DATA</i>	<i>0D4H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>S3CON</i>	<i>DATA</i>	<i>0ACh</i>
<i>S3BUF</i>	<i>DATA</i>	<i>0ADH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		;16 bytes
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>ORG</i>	<i>008BH</i>

LJMP **UART3_ISR****ORG** **0100H****UART3_ISR:**

PUSH	ACC
PUSH	PSW
MOV	PSW,#08H

MOV	A,S3CON
JNB	ACC.I,CHKRI
ANL	S3CON,#NOT 02H
CLR	BUSY

CHKRI:

JNB	ACC.0,UART3ISR_EXIT
ANL	S3CON,#NOT 01H
MOV	A,WPTR
ANL	A,#0FH
ADD	A,#BUFFER
MOV	R0,A
MOV	@R0,S3BUF
INC	WPTR

UART3ISR_EXIT:

POP	PSW
POP	ACC
RETI	

UART3_INIT:

MOV	S3CON,#50H
MOV	T3L,#0E8H
MOV	T3H,#0FFH
MOV	T4T3M,#0AH
CLR	BUSY
MOV	WPTR,#00H
MOV	RPTR,#00H
RET	

;65536-11059200/115200/4=0FFE8H

UART3_SEND:

JB	BUSY,\$
SETB	BUSY
MOV	S3BUFA,A
RET	

UART3_SENDSTR:

CLR	A
MOVC	A,@A+DPTR
JZ	SEND3END
LCALL	UART3_SEND
INC	DPTR
JMP	UART3_SENDSTR

SEND3END:

RET	
------------	--

START:

MOV	SP, #5FH
ORL	P_SW2,#80H
	;使能访问 XFR，没有冲突不用关闭
MOV	P0M0, #00H
MOV	P0M1, #00H

```

MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL   UART3_INIT
MOV     IE2,#08H
SETB    EA

MOV     DPTR,#STRING
LCALL   UART3_SENDSTR

LOOP:
MOV     A,RPTR
XRL     A,WPTR
ANL     A,#0FH
JZ      LOOP
MOV     A,RPTR
ANL     A,#0FH
ADD     A,#BUFFER
MOV     R0,A
MOV     A,@R0
LCALL   UART3_SEND
INC     RPTR
JMP     LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

END

```

16.7.27 定时器 4 (16 位自动重载) , 用作定时

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM4_Isr() interrupt 20
{
    P10 = !P10;           //测试端口
}

void main()
{
    P_SW2 |= 0x80;        //使能访问XFR, 没有冲突不用关闭
    P0M0 = 0x00;
    P0M1 = 0x00;
}

```

```

P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

T4L = 0x66;           //65536-11.0592M/12/1000
T4H = 0xfc;
T4T3M = 0x80;         //启动定时器
IE2 = ET4;             //使能定时器中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T4T3M	DATA	0D1H
T4L	DATA	0D3H
T4H	DATA	0D2H
T3L	DATA	0D5H
T3H	DATA	0D4H
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
IE2	DATA	0AFH
ET2	EQU	04H
ET3	EQU	20H
ET4	EQU	40H
AUXINTIF	DATA	0EFH
T2IF	EQU	01H
T3IF	EQU	02H
T4IF	EQU	04H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG		0000H
LJMP		START
ORG		00A3H

<i>LJMP</i>	<i>TM4ISR</i>
<i>ORG</i>	<i>0100H</i>
<i>TM4ISR:</i>	
<i>CPL</i>	<i>P1.0</i>
<i>RETI</i>	; 测试端口
<i>START:</i>	
<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	; 使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>MOV</i>	<i>T4L,#66H</i>
<i>MOV</i>	<i>T4H,#0FCH</i>
<i>MOV</i>	<i>T4T3M,#80H</i>
<i>MOV</i>	<i>IE2,#ET4</i>
<i>SETB</i>	<i>EA</i>
<i>JMP</i>	\$
<i>END</i>	

16.7.28 定时器 4 (外部计数—扩展 T4 为外部下降沿中断)

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void TM4_Isr() interrupt 20
{
    P10 = !P10; // 测试端口
}

void main()
{
    P_SW2 |= 0x80; // 使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
```

```

P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

T4L = 0xff;
T4H = 0xff;
T4T3M = 0xc0;           // 设置外部计数模式并启动定时器
IE2 = ET4;               // 使能定时器中断
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T4T3M	DATA	0DIH
T4L	DATA	0D3H
T4H	DATA	0D2H
T3L	DATA	0D5H
T3H	DATA	0D4H
T2L	DATA	0D7H
T2H	DATA	0D6H
AUXR	DATA	8EH
IE2	DATA	0AFH
ET2	EQU	04H
ET3	EQU	20H
ET4	EQU	40H
AUXINTIF	DATA	0EFH
T2IF	EQU	01H
T3IF	EQU	02H
T4IF	EQU	04H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG	ORG	0000H
LJMP	START	
ORG	ORG	00A3H
LJMP	TM4ISR	

<i>ORG</i>	<i>0100H</i>	
<i>TM4ISR:</i>		
<i>CPL</i>	<i>P1.0</i>	; 测试端口
<i>RETI</i>		
<i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>T4L,#0FFH</i>	
<i>MOV</i>	<i>T4H,#0FFH</i>	
<i>MOV</i>	<i>T4T3M,#0C0H</i>	; 设置外部计数模式并启动定时器
<i>MOV</i>	<i>IE2,#ET4</i>	; 使能定时器中断
<i>SETB</i>	<i>EA</i>	
<i>JMP</i>	<i>\$</i>	
<i>END</i>		

16.7.29 定时器 4, 时钟分频输出

C 语言代码

```
// 测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80; // 使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
```

```

P3M1 = 0x00;

T4L = 0x66;                                //65536-11.0592M/12/1000
T4H = 0xfc;
T4T3M = 0x90;                            //使能时钟输出并启动定时器

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T4T3M	DATA	0D1H
T4L	DATA	0D3H
T4H	DATA	0D2H
T3L	DATA	0D5H
T3H	DATA	0D4H
T2L	DATA	0D7H
T2H	DATA	0D6H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
	ORG	0000H
	LJMP	START
	ORG	0100H
START:	MOV	SP, #5FH
	ORL	P_SW2,#80H ; 使能访问 XFR，没有冲突不用关闭
	MOV	P0M0, #00H
	MOV	P0M1, #00H
	MOV	P1M0, #00H
	MOV	P1M1, #00H
	MOV	P2M0, #00H
	MOV	P2M1, #00H
	MOV	P3M0, #00H
	MOV	P3M1, #00H
	MOV	P4M0, #00H
	MOV	P4M1, #00H
	MOV	P5M0, #00H
	MOV	P5M1, #00H
	MOV	T4L,#66H ; 65536-11.0592M/12/1000
	MOV	T4H,#0FCH

```

MOV      T4T3M,#90H          ;使能时钟输出并启动定时器
JMP      $

END

```

16.7.30 定时器 4 做串口 4 波特率发生器

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                           //加2 操作是为了让 Keil 编译器
                           //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void Uart4Isr() interrupt 18
{
    if (S4CON & 0x02)
    {
        S4CON &= ~0x02;
        busy = 0;
    }
    if (S4CON & 0x01)
    {
        S4CON &= ~0x01;
        buffer[wptr++] = S4BUF;
        wptr &= 0x0f;
    }
}

void Uart4Init()
{
    S4CON = 0x50;
    T4L = BRT;
    T4H = BRT >> 8;
    T4T3M = 0xa0;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart4Send(char dat)
{
    while (busy);
    busy = 1;
    S4BUF = dat;
}

```

```

}

void Uart4SendStr(char *p)
{
    while (*p)
    {
        Uart4Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    Uart4Init();
    IE2 = 0x10;
    EA = 1;
    Uart4SendStr("Uart Test !\r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            Uart4Send(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T4T3M	DATA	0DIH
T4L	DATA	0D3H
T4H	DATA	0D2H
T3L	DATA	0D5H
T3H	DATA	0D4H
T2L	DATA	0D7H
T2H	DATA	0D6H
S4CON	DATA	84H
S4BUF	DATA	85H
IE2	DATA	0AFH

<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>

;16 bytes

<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>

<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>
<i>ORG</i>	<i>0093H</i>
<i>LJMP</i>	<i>UART4_ISR</i>
 <i>ORG</i>	 <i>0100H</i>

UART4_ISR:

<i>PUSH</i>	<i>ACC</i>
<i>PUSH</i>	<i>PSW</i>
<i>MOV</i>	<i>PSW,#08H</i>
 <i>MOV</i>	<i>A,S4CON</i>
<i>JNB</i>	<i>ACC.I,CHKRI</i>
<i>ANL</i>	<i>S4CON,#NOT 02H</i>
<i>CLR</i>	<i>BUSY</i>

CHKRI:

<i>JNB</i>	<i>ACC.0,UART4ISR_EXIT</i>
<i>ANL</i>	<i>S4CON,#NOT 01H</i>
<i>MOV</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>@R0,S4BUF</i>
<i>INC</i>	<i>WPTR</i>

UART4ISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART4_INIT:

<i>MOV</i>	<i>S4CON,#50H</i>
<i>MOV</i>	<i>T4L,#0E8H</i>
<i>MOV</i>	<i>T4H,#0FFH</i>
<i>MOV</i>	<i>T4T3M,#0A0H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

;65536-11059200/115200/4=0FFE8H

UART4_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>S4BUFA,A</i>
<i>RET</i>	

UART4_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SEND4END</i>
<i>LCALL</i>	<i>UART4_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART4_SENDSTR</i>

SEND4END:

<i>RET</i>

START:

<i>MOV</i>	<i>SP,#5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0,#00H</i>
<i>MOV</i>	<i>P0M1,#00H</i>
<i>MOV</i>	<i>P1M0,#00H</i>
<i>MOV</i>	<i>P1M1,#00H</i>
<i>MOV</i>	<i>P2M0,#00H</i>
<i>MOV</i>	<i>P2M1,#00H</i>
<i>MOV</i>	<i>P3M0,#00H</i>
<i>MOV</i>	<i>P3M1,#00H</i>
<i>MOV</i>	<i>P4M0,#00H</i>
<i>MOV</i>	<i>P4M1,#00H</i>
<i>MOV</i>	<i>P5M0,#00H</i>
<i>MOV</i>	<i>P5M1,#00H</i>
<i>LCALL</i>	<i>UART4_INIT</i>
<i>MOV</i>	<i>IE2,#10H</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART4_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
<i>XRL</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>JZ</i>	<i>LOOP</i>
<i>MOV</i>	<i>A,RPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>A,@R0</i>
<i>LCALL</i>	<i>UART4_SEND</i>
<i>INC</i>	<i>RPTR</i>
<i>JMP</i>	<i>LOOP</i>

<i>STRING:</i>	<i>DB</i>	<i>'Uart Test !',0DH,0AH,00H</i>
-----------------------	-----------	----------------------------------

<i>END</i>

16.7.31 定时器 T11 应用范例

C 语言代码

```
//测试工作频率为 11.0592MHz

/**************** 功能说明 *****/
本例程基于 STC8H2K12U 芯片进行编写测试。
程序演示定时器 T11 的功能。
下载时，选择时钟 24MHZ (用户可自行修改频率)。
*****/



#include "stc8h.h"
#include "intrins.h"

#define MAIN_Fosc          11059200UL
#define TI1MS12T           (65536 - MAIN_Fosc / 12 / 1000)
#define TI1MSIT             (65536 - MAIN_Fosc / 1000)

void tm11isr() interrupt 13
{
//    TI1CR &= ~0x01;                                //借用 13 号中断向量地址，在 isr.asm 进行中断映射
//    P10 = ~P10;                                    //清中断标志 TI1IF，进中断时硬件自动清除
}

void main()
{
    P_SW2 |= 0x80;                                  //扩展寄存器(XFR)访问使能

    P0M1 = 0x00;   P0M0 = 0x00;                      //设置为准双向口
    P1M1 = 0x00;   P1M0 = 0x00;                      //设置为准双向口
    P2M1 = 0x00;   P2M0 = 0x00;                      //设置为准双向口
    P3M1 = 0x00;   P3M0 = 0x00;                      //设置为准双向口
    P4M1 = 0x00;   P4M0 = 0x00;                      //设置为准双向口
    P5M1 = 0x00;   P5M0 = 0x00;                      //设置为准双向口
    P6M1 = 0x00;   P6M0 = 0x00;                      //设置为准双向口
    P7M1 = 0x00;   P7M0 = 0x00;                      //设置为准双向口

    IRCDB = 0x10;        //内部高速 IRC 时钟，停振后被唤醒后起振，等待多少个时钟数后给 MCU 供应时钟
    IAP_TPS = 40;         //根据系统工作频率设置此寄存器
    // (如果系统工作频率为 40MHz，则 IAP_TPS 设置为 40;
    // 如果系统工作频率为 22.1184MHz，则 IAP_TPS 设置为 22)

    //定时器(12T)
//    TI1CR = 0x00;          //做定时器关闭时钟输出,12T 模式,系统时钟做时钟源
//    TI1PS = 0;              //分频系数: (TI1PS+1)分频
//    TI1L = TI1MS12T;
//    TI1H = TI1MS12T >> 8;
//    TI1CR |= 0x82;          //定时器 T11 开始计数，允许中断
//    EA = 1;

    //定时器(IT)
//    TI1CR = 0x10;          //做定时器，关闭时钟输出，IT 模式，系统时钟做时钟源
//    TI1PS = 0;              //分频系数: (TI1PS+1)分频
//    TI1L = TI1MSIT;
//    TI1H = TI1MSIT >> 8;
```

```
// T11CR |= 0x82;                                //定时器11 开始计数, 允许中断
// EA = I;

// 计数模式 - T11 脚(P14)输入脉冲计数
// T11CR = 0x50;
// T11PS = 0;
// T11L = 0xff;
// T11H = 0xff;
// T11CR |= 0x82;                                //定时器11 开始计数, 允许中断
// EA = I;

// X32KCR = 0x80 + 0x40;                          //启动外部32K 晶振, 低增益+0x00, 高增益+0x40.
// while (!(X32KCR & 1));                         //等待时钟稳定

// 选择时钟源
// T11CR = 0x10;
// T11CR = 0x1c;                                  //做定时器关闭时钟输出, IT 模式, 系统时钟做时钟源
// T11CR = 0x18;                                  //做定时器关闭时钟输出, IT 模式,
// T11PS = 0;                                     //内部低速IRC 做时钟源(自动启动内部低速IRC)
// T11L = T11MSIT;                               //做定时器关闭时钟输出, IT 模式,
// T11H = T11MSIT >> 8;                          //外部32K 晶振做时钟源
// T11CR |= 0x82;                                //分频系数: (T11PS+1)分频
// EA = I;                                       //定时器11 开始计数, 允许中断

// 时钟输出
T11CR |= 0x20;                                  //使能P1.5 口定时器11 时钟输出

while (1)
{
    P11 = ~P11;

    //唤醒PD 休眠模式
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    PCON = 0x02;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}

}
```

17 RTC 实时时钟，年/月/日/时/分/秒

产品线	RTC	支持 星期	芯片复位时 寄存器不复位 ^[1]
STC8H1K08 系列			
STC8H1K28 系列			
STC8H3K64S4 系列			
STC8H3K64S2 系列			
STC8H8K64U 系列 A 版本			
STC8H8K64U 系列 B/C/D 版本	●		
STC8H4K64TL 系列	●		
STC8H4K64TLCD 系列	●		
STC8H1K08T 系列	●		
STC8H2K12U-A 系列	●		●
STC8H2K12U-B 系列	●		●
STC8H2K32U 系列	●		●
STC8G1K08-SOP8 系列			
STC8G1K08A-SOP8 系列			

^[1]: 包括硬件复位和软件复位

STC8H 系列部分单片机内部集成一个实时时钟控制电路，主要有如下特性：

- 低功耗：RTC 模块工作电流低至 **2uA@VCC=3.3V、3uA@VCC=5.0V（典型值）**
- 长时间跨度：支持 2000 年~2099 年，并自动判断闰年
- 闹钟：支持一组闹钟设置
- 支持多个中断
 - 一组闹钟中断（每天中断一次，中断的时间点为闹钟寄存器所设置的任意时/分/秒）
 - 日中断（每天中断一次，中断的时间点为每天的 0 时 0 分 0 秒）
 - 小时中断（每小时中断一次，中断的时间点为分/秒均为 0，即整点时）
 - 分钟中断（每分钟中断一次，中断的时间点为秒为 0，即分钟寄存器发生变化时）
 - 秒中断（每秒中断一次，中断的时间点为秒寄存器发生变化时）
 - 1/2 秒中断（每 1/2 秒中断一次）
 - 1/8 秒中断（每 1/8 秒中断一次）
 - 1/32 秒中断（每 1/32 秒中断一次）
- 支持掉电唤醒
- **特别注意：目前带 RTC 功能的 STC8H 系列单片机均无星期功能**

17.1 RTC 相关的寄存器

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
RTCCR	RTC 控制寄存器	FE60H	-	-	-	-	-	-	-	RUNRTC	xxxx,xxx0	
RTCCFG	RTC 配置寄存器	FE61H	-	-	-	-	-	-	-	RTCCKS	SETRTC	xxxx,xx00
RTCien	RTC 中断使能寄存器	FE62H	EALAI	EDAYI	EHOURI	EMINI	ESECI	ESEC2I	ESEC8I	ESEC32I		0000,0000
RTCif	RTC 中断请求寄存器	FE63H	ALAIF	DAYIF	HOURIF	MINIF	SECIF	SEC2IF	SEC8IF	SEC32IF		0000,0000
ALAHOUR	RTC 闹钟的小时值	FE64H	-	-	-							xxx0,0000
ALAMIN	RTC 闹钟的分钟值	FE65H	-	-								xx00,0000
ALASEC	RTC 闹钟的秒值	FE66H	-	-								xx00,0000
ALASSEC	RTC 闹钟的 1/128 秒值	FE67H	-									x000,0000
INIYEAR	RTC 年初始化	FE68H	-									x000,0000
INIMONTH	RTC 月初始化	FE69H	-	-	-	-	-					xxxx,0000
INIDAY	RTC 日初始化	FE6AH	-	-	-							xxx0,0000
INIHOUR	RTC 小时初始化	FE6BH	-	-	-							xxx0,0000
INIMIN	RTC 分钟初始化	FE6CH	-	-								xx00,0000
INISEC	RTC 秒初始化	FE6DH	-	-								xx00,0000
INISSEC	RTC1/128 秒初始化	FE6EH	-									x000,0000
INIWEEK	RTC 星期初始化	FE6FH	-	-	-	-	-	-				xxxx,x000
RTCWEEK	RTC 的星期计数值											
RTCYEAR	RTC 的年计数值	FE70H	-									x000,0000
RTCMONTH	RTC 的月计数值	FE71H	-	-	-	-	-					xxxx,0000
RTCDAY	RTC 的日计数值	FE72H	-	-	-							xx00,0000
RTCHOUR	RTC 的小时计数值	FE73H	-	-	-							xx00,0000
RTCMIN	RTC 的分钟计数值	FE74H	-	-								xx00,0000
RTCSEC	RTC 的秒计数值	FE75H	-	-								xx00,0000
RTCSSEC	RTC 的 1/128 秒计数值	FE76H	-									x000,0000

17.1.1 RTC 控制寄存器 (RTCCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
RTCCR	FE60H	-	-	-	-	-	-	-	RUNRTC

RUNRTC: RTC 模块控制位

0: 关闭 RTC, RTC 停止计数

1: 使能 RTC, 并开始 RTC 计数

17.1.2 RTC 配置寄存器 (RTCCFG)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
RTCCFG	FE61H	-	-	-	-	-	-	RTCCKS	SETRTC

RTCCKS: RTC 时钟源选择

0: 选择外部 32.768KHz 时钟源 (需先软件启动外部 32K 晶振)

1: 选择内部 32K 时钟源 (需先软件启动内部 32K 振荡器)

SETRTC: 设置 RTC 初始值

0: 无意义

1: 触发 RTC 寄存器初始化。当 SETRTC 设置为 1 时, 硬件会自动将寄存器 INIYEAR、INIMONTH、INIDAY、INI_HOUR、INIMIN、INISEC、INISSEC 中的值复制到寄存器 RTCYEAR、RTCMONTH、RTCDAY、RTCHOUR、RTCMIN、RTCSEC、RTCSSEC 中。初始完成后, 硬件会自动将 SETRTC 位清 0。

17.1.3 RTC 中断使能寄存器 (RTCIEN)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
RTCIEN	FE62H	EALAI	EDAYI	EHOURI	EMINI	ESECI	ESEC2I	ESEC8I	ESEC32I

EALAI: 闹钟中断使能位

0: 关闭闹钟中断

1: 使能闹钟中断

EDAYI: 一日 (24 小时) 中断使能位

0: 关闭一日中断

1: 使能一日中断

EHOURI: 一小时 (60 分钟) 中断使能位

0: 关闭小时中断

1: 使能小时中断

EMINI: 一分钟 (60 秒) 中断使能位

0: 关闭分钟中断

1: 使能分钟中断

ESECI: 一秒中断使能位

0: 关闭秒中断

1: 使能秒中断

ESEC2I: 1/2 秒中断使能位

0: 关闭 1/2 秒中断

1: 使能 1/2 秒中断

ESEC8I: 1/8 秒中断使能位

0: 关闭 1/8 秒中断

1: 使能 1/8 秒中断

ESEC32I: 1/32 秒中断使能位

0: 关闭 1/32 秒中断

1: 使能 1/32 秒中断

17.1.4 RTC 中断请求寄存器 (RTCIF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
RTCIF	FE63H	ALAIF	DAYIF	HOURIF	MINIF	SECIF	SEC2IF	SEC8IF	SEC32IF

ALAIF: 闹钟中断请求位。需软件清 0, 软件写 1 无效。

DAYIF: 一日 (24 小时) 中断请求位。需软件清 0, 软件写 1 无效。

HOURIF: 一小时 (60 分钟) 中断请求位。需软件清 0, 软件写 1 无效。

MINIF: 一分钟 (60 秒) 中断请求位。需软件清 0, 软件写 1 无效。

SECIF: 一秒中断请求位。需软件清 0, 软件写 1 无效。

SEC2IF: 1/2 秒中断请求位。需软件清 0, 软件写 1 无效。

SEC8IF: 1/8 秒中断请求位。需软件清 0, 软件写 1 无效。

SEC32IF: 1/32 秒中断请求位。需软件清 0, 软件写 1 无效。

17.1.5 RTC 闹钟设置寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ALAHOUR	FE64H	-	-	-					
ALAMIN	FE65H	-	-						
ALASEC	FE66H	-	-						
ALASSEC	FE67H	-							

ALAHOUR: 设置每天闹钟的小时值。

注意: 设置的值不是 BCD 码, 而是 HEX 码, 比如需要设置小时值 20 到 ALAHOUR, 则需使用如下代码进行设置

```
MOV      DPTR,#ALAHOUR
MOV      A,#14H
MOVX    @DPTR,A
```

ALAMIN: 设置每天闹钟的分钟值。数字编码与 ALAHOUR 相同。

ALASEC: 设置每天闹钟的秒值。数字编码与 ALAHOUR 相同。

ALASSEC: 设置每天闹钟的 1/128 秒值。数字编码与 ALAHOUR 相同。

17.1.6 RTC 实时时钟初始值设置寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
INIYEAR	FE68H	-							
INIMONTH	FE69H								
INIDAY	FE6AH								
INIHOUR	FE6BH	-	-	-					
INIMIN	FE6CH	-	-						
INISEC	FE6DH	-	-						
INISSEC	FE6EH	-							
INIWEEK RTCWEEK	FE6FH	-	-	-	-	-	-		

INIYEAR: 设置当前实时时间的年值。有效值范围 00~99。对应 2000 年~2099 年

注意: 设置的值不是 BCD 码, 而是 HEX 码, 比如需要设置 20 到 INIYEAR, 则需使用如下代码进行设置

```
MOV      DPTR,#INIYEAR
MOV      A,#14H
MOVX    @DPTR,A
```

INIMONTH: 设置当前实时时间的月值。有效值范围 1~12。数字编码与 INIYEAR 相同。

INIDAY: 设置当前实时时间的日值。有效值范围 1~31。数字编码与 INIYEAR 相同。

INIHOUR: 设置当前实时时间的小时值。有效值范围 00~23。数字编码与 INIYEAR 相同。

INIMIN: 设置当前实时时间的分钟值。有效值范围 00~59。数字编码与 INIYEAR 相同。

INISEC: 设置当前实时时间的秒值。有效值范围 00~59。数字编码与 INIYEAR 相同。

INISSEC: 设置当前实时时间的 1/128 秒值。有效值范围 00~127。数字编码与 INIYEAR 相同。

INIWEEK: 设置当前实时时间的星期。有效值范围 0~6。(星期计数值也从此寄存器读取)

当用户设置完成上面的初始值寄存器后, 用户还需要向 SETRTC 位 (RTCCFG.0) 写 1 来触发硬件将初始值装载到 RTC 实时计数器中

另需注意: 硬件不会对初始化数据的有效性进行检查, 需要用户在设置初始值时, 必须保证数据的有效性, 不能超出其有效范围。

17.1.7 RTC 实时时钟计数寄存器

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
RTCYEAR	FE70H	-							
RTCMONTH	FE71H								
RTCDAY	FE72H								
RTCHOUR	FE73H	-	-	-					
RTCMIN	FE74H	-	-						
RTCSEC	FE75H	-	-						
RTCSSEC	FE76H	-							

RTCYEAR: 当前实时时间的年值。注意: 寄存器的值不是 BCD 码, 而是 HEX 码

RTCMONTH: 当前实时时间的月值。数字编码与 YEAR 相同。

RTCDAY: 当前实时时间的日值。数字编码与 YEAR 相同。

RTCHOUR: 当前实时时间的小时值。数字编码与 YEAR 相同。

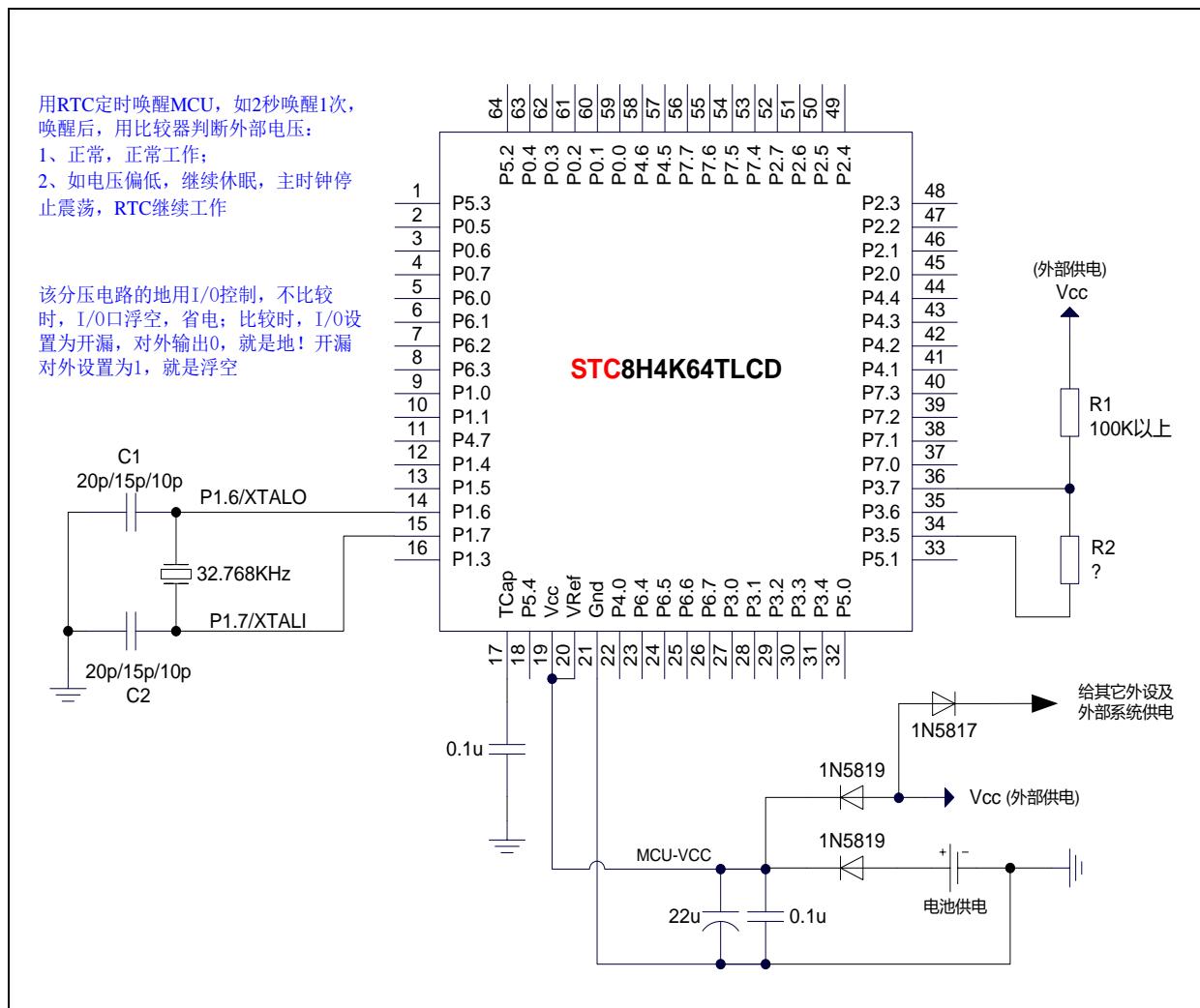
RTCMIN: 当前实时时间的分钟值。数字编码与 YEAR 相同。

RTCSEC: 当前实时时间的秒值。数字编码与 YEAR 相同。

RTCSSEC: 当前实时时间的 1/128 秒值。数字编码与 YEAR 相同。

注意: RTCYEAR、RTCMONTH、RTCDAY、RTCHOUR、RTCMIN、RTCSEC 和 RTCSSEC 均为只读寄存器, 若需要对这些寄存器执行写操作, 必须通过寄存器 INIYEAR、INIMONTH、INIDAT、INIHOU、INIMIN、INISEC、INISSEC 和 SETRTC 来实现。

17.2 RTC 实战线路图



17.3 范例程序

17.3.1 串口打印 RTC 时钟范例

C 语言代码

//测试工作频率为 22.1184MHz，需要将 C 语言代码文件与下面的汇编代码文件加载到同一个项目里使用

```
#include "stc8h.h"
#include "intrins.h"
#include "stdio.h"

#define MAIN_Fosc      22118400L
#define Baudrate      115200L
#define TM            (65536 -(MAIN_Fosc/Baudrate+2)/4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit      BIS_Flag;

void RTC_config(void);

void UartInit(void)
{
    SCON = (SCON & 0x3f) / 0x40;
    T2L = TM;
    T2H = TM>>8;
    AUXR |= 0x15;
}

void UartPutc(unsigned char dat)
{
    SBUF = dat;
    while(TI==0);
    TI = 0;
}

char putchar(char c)
{
    UartPutc(c);
    return c;
}

void RTC_Isr() interrupt 13
{
    if(RTCIF & 0x08)          //判断是否秒中断
    {
        RTCIF &= ~0x08;       //清中断标志
        BIS_Flag = 1;
    }
}

void main(void)
{
    P_SW2 |= 0x80;           //使能 XFR 访问

    P0M1 = 0;    P0M0 = 0;    //设置为准双向口
    P1M1 = 0;    P1M0 = 0;    //设置为准双向口
```

```

P2M1 = 0; P2M0 = 0; //设置为准双向口
P3M1 = 0; P3M0 = 0; //设置为准双向口
P4M1 = 0; P4M0 = 0; //设置为准双向口
P5M1 = 0; P5M0 = 0; //设置为准双向口

UartInit();
RTC_config();
EA = 1;
printf("RTC Test Programme!\r\n"); //UART 发送一个字符串

while (1)
{
    if(B1S_Flag)
    {
        B1S_Flag = 0;

        printf("Year=%bd ", RTCYEAR);
        printf("Month=%bd ", RTCMONTH);
        printf("Day=%bd ", RTCDAY);
        printf("Hour=%bd ", RTCHOUR);
        printf("Minute=%bd ", RTCMIN);
        printf("Second=%bd ", RTCSEC);
        printf("\r\n");
    }
}

void RTC_config(void)
{
// //选择内部低速IRC
// IRC32KCR = 0x80; //启动内部低速振荡器
// while (!(IRC32KCR & 0x01)); //等待时钟稳定
// RTCCFG |= 0x02; //选择内部低速IRC 作为RTC 时钟源

//选择外部32K
X32KCR = 0xc0; //启动外部32K 晶振
while (!(X32KCR & 0x01)); //等待时钟稳定
RTCCFG &= ~0x02; //选择外部32K 作为RTC 时钟源

INIYEAR = 21; //Y:2021
INIMONTH = 12; //M:12
INIDAY = 31; //D:31
INI_HOUR = 23; //H:23
INIMIN = 59; //M:59
INISEC = 50; //S:50
INISSEC = 0; //S/128:0
RTCCFG |= 0x01; //触发RTC 寄存器初始化

RTCIF = 0; //清中断标志
RTCIEN = 0x08; //使能RTC 秒中断
RTCCR = 0x01; // RTC 使能
}

```

汇编代码

; 将以下代码保存为 ASM 格式文件, 一起加载到项目里, 例如: isr.asm

CSEG	AT 0123H
JMP	006BH

END

17.3.2 利用 ISP 软件的用户接口实现不停电下载保持 RTC 参数

C 语言代码

//测试工作频率为 11.0592MHz

***** 功能说明 *****

现有单片机系列的 RTC 模块，在单片机复位后 RTC 相关的特殊功能寄存器也会复位

本例程主要用于解决 ISP 下载后用户的 RTC 参数丢失的问题

解决思路: ISP 下载前，先将 RTC 相关参数通过 ISP 下载软件的用户接口上传到 PC 保存，等待 ISP 下载完成后，下载软件再将保存的相关参数写入到 FLASH 的指定地址（范例中指定的地址为 FE00H）。ISP 下载完成后会立即运行用户代码，用户程序在初始化 RTC 寄存器时，可从 FLASH 的指定地址中读取之前上传的 RTC 相关参数对 RTC 寄存器进行初始化，即可实现不停电下载保持 RTC 参数的目的。

下载时，选择时钟 11.0592MHz

***** */

```
#include "stc8h.h"
#include "intrins.h"
#include "stdio.h"

#include "stc8h.h"
#include "intrins.h"

#define FOSC           11059200UL
#define BAUD          (65536 - FOSC/4/115200)

typedef bit        BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

struct RTC_INIT
{
    BYTE bValidTag;                      //数据有效标志(0x5a)
    BYTE bIniYear;                      //年(RTC 初始化值)
    BYTE bIniMonth;                     //月
    BYTE bIniDay;                       //日
    BYTE bIniHour;                      //时
    BYTE bIniMinute;                   //分
    BYTE bIniSecond;                   //秒
    BYTE bIniSSecond;                  //次秒
    BYTE bAlaHour;                     //时(RTC 闹钟设置值)
    BYTE bAlaMinute;                   //分
    BYTE bAlaSecond;                   //秒
    BYTE bAlaSSecond;                  //次秒
};

struct RTC_INIT code InitBlock _at_ 0xfe00;

void SysInit();
void UartInit();
void RTCInit();
void SendUart(BYTE dat);
void UnpackCmd(BYTE dat);
void IapProgram(WORD addr, BYTE dat);

BOOL fUartBusy;
```

```
BOOL fFetchRtc;
BOOL fReset2Isp;
BYTE bUartStage;

BYTE bDump[7];

void main()
{
    SysInit();                                //系统初始化
    UartInit();
    RTCInit();
    EA = 1;

    fUartBusy = 0;
    fFetchRtc = 0;
    fReset2Isp = 0;
    bUartStage = 0;

    while (1)
    {
        if (fFetchRtc)                        //获取 RTC 数据请求
        {
            fFetchRtc = 0;

            RTCCR = 0;                      //上传当前的 RTC 值时,必须临时停止 RTC
                                                //以免发生进位错误
            bDump[0] = RTCYEAR;              //快速将当前的 RTC 值缓存,
                                                //以缩短 RTC 暂停的时间,减小误差

            bDump[1] = RTCMONTH;
            bDump[2] = RTCDAY;
            bDump[3] = RTCHOUR;
            bDump[4] = RTCMIN;
            bDump[5] = RTCSEC;
            bDump[6] = RTCSSC;

            RTCCR = 1;

            SendUart(0x5a);                //上传 12 字节 RTC 参数
            SendUart(bDump[0]);
            SendUart(bDump[1]);
            SendUart(bDump[2]);
            SendUart(bDump[3]);
            SendUart(bDump[4]);
            SendUart(bDump[5]);
            SendUart(bDump[6]);
            SendUart(ALAHOUR);
            SendUart(ALAMIN);
            SendUart(ALASEC);
            SendUart(ALASSEC);
        }

        if (fReset2Isp)                    //重启请求
        {
            fReset2Isp = 0;

            IAP_CONTR = 0x60;             //软件触发复位到系统 ISP 区
        }
    }
}
```

```
void uart_isr() interrupt UART1_VECTOR
{
    BYTE dat;

    if (TI)
    {
        TI = 0;
        fUartBusy = 0;
    }

    if (RI)
    {
        RI = 0;

        dat = SBUF;
        switch (bUartStage++)
        {
            default:
            case 0:
                L_Check1st:
                if (dat == '@') bUartStage = 1;
                else bUartStage = 0;
                break;
            case 1:
                if (dat == 'F') bUartStage = 2;
                else if (dat == 'R') bUartStage = 7;
                else goto L_Check1st;
                break;
            case 2:
                if (dat != 'E') goto L_Check1st;
                break;
            case 3:
                if (dat != 'T') goto L_Check1st;
                break;
            case 4:
                if (dat != 'C') goto L_Check1st;
                break;
            case 5:
                if (dat != 'H') goto L_Check1st;
                break;
            case 6:
                if (dat != '#') goto L_Check1st;
                bUartStage = 0;
                fFetchRtc = 1; //当前命令序列为获取 RTC 数据命令:"@FETCH#"
                break;
            case 7:
                if (dat != 'E') goto L_Check1st;
                break;
            case 8:
                if (dat != 'B') goto L_Check1st;
                break;
            case 9:
            case 10:
                if (dat != 'O') goto L_Check1st;
                break;
            case 11:
                if (dat != 'T') goto L_Check1st;
                break;
```

```

    case 12:
        if (dat != '#') goto L_Check1st;
        bUartStage = 0;
        fReset2Isp = 1;                                //当前命令序列为重启命令:"@REBOOT#"
        break;
    }
}
}

void rtc_isr() interrupt RTC_VECTOR                //RTC 中断复位程序
{
    RTCIF = 0x00;                                  //清 RTC 中断标志
}

void SysInit()
{
    P_SW2 |= 0x80;

    P0M0 = 0x00; P0MI = 0x00;
    P1M0 = 0x00; P1MI = 0x00;
    P2M0 = 0x00; P2MI = 0x00;
    P3M0 = 0x00; P3MI = 0x00;
    P4M0 = 0x00; P4MI = 0x00;
    P5M0 = 0x00; P5MI = 0x00;
    P6M0 = 0x00; P6MI = 0x00;
    P7M0 = 0x00; P7MI = 0x00;
}

void UartInit()                                     //串口初始化函数
{
    SCON = 0x50;
    AUXR = 0x40;
    TMOD = 0x00;
    TL1 = BAUD;
    TH1 = BAUD >> 8;
    TR1 = 1;
    ES = 1;
}

void RTCInit()                                     //RTC 初始化函数
{
//    IRC32KCR = 0x80;
//    while (!(IRC32KCR & 0x01));
//    RTCCFG |= 0x02;                                //选择内部低速IRC为RTC时钟源

    X32KCR = 0xc0;
    while (!(X32KCR & 0x01));
    RTCCFG &= ~0x02;                               //选择外部部32K为RTC时钟源

    if (InitBlock.bValidTag == 0x5a)
    {
        INIYEAR = InitBlock.bIniYear;
        INIMONTH = InitBlock.bIniMonth;
        INIDAY = InitBlock.bIniDay;
        INIHOUR = InitBlock.bIniHour;
        INIMIN = InitBlock.bIniMinute;
        INISEC = InitBlock.bIniSecond;
        //如果初始化数据块有效,则使用数据块初始化RTC
    }
}

```

```

INISSEC = InitBlock.bIniSSecond;
ALAHOURL = InitBlock.bAlaHour;
ALAMIN = InitBlock.bAlaMinute;
ALASEC = InitBlock.bAlaSecond;
ALASSEC = InitBlock.bAlaSSecond;

IapProgram(0x0000, 0x00); //销毁初始化数据块, 以免重复初始化
}

else
{
    INIYEAR = 23; //否则初始化 RTC 为默认值
    INIMONTH = 1;
    INIDAY = 29;
    INIHOUR = 12;
    INIMIN = 0;
    INISEC = 0;
    INISSEC = 0;
    ALAHOURL = 0;
    ALAMIN = 0;
    ALASEC = 0;
    ALASSEC = 0;
}

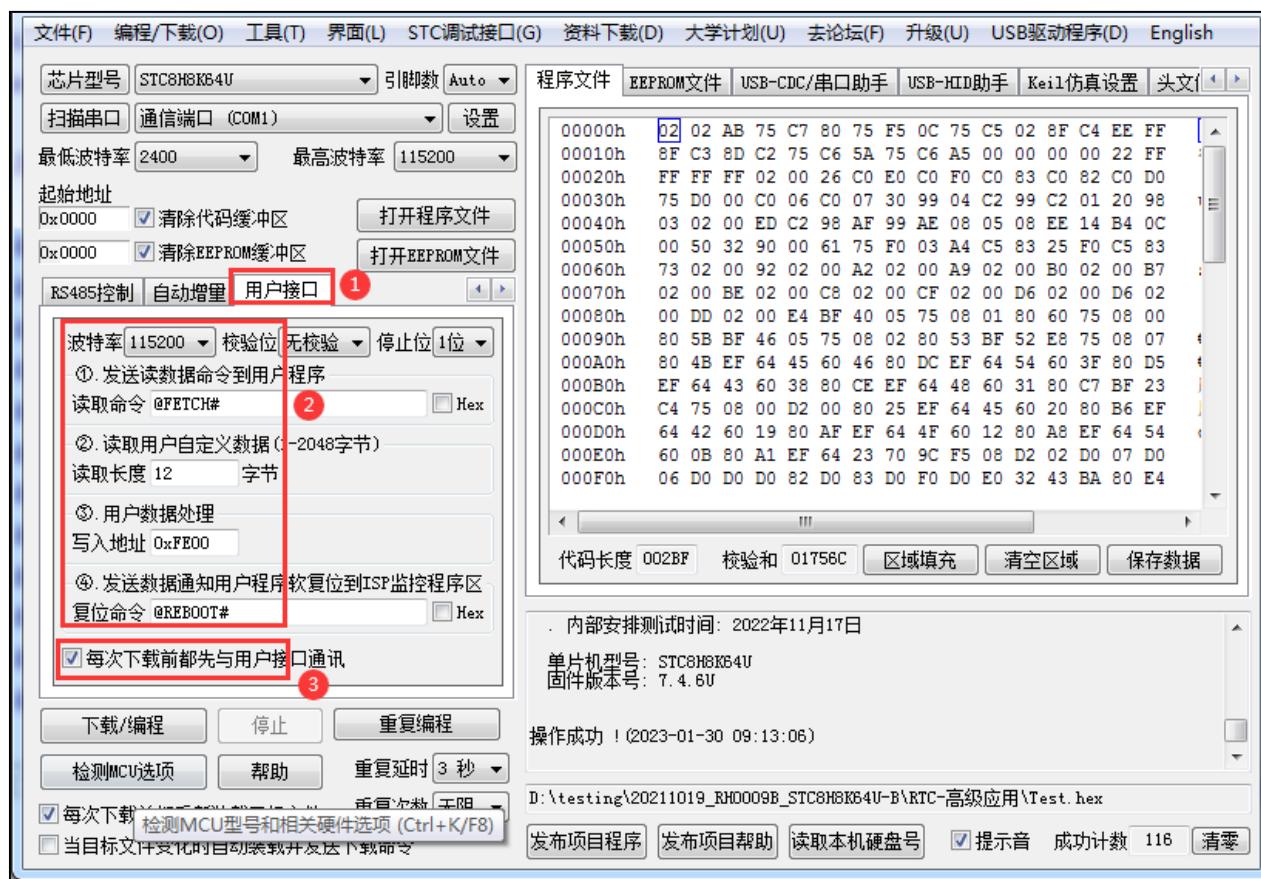
RTCCFG |= 0x01; //写入 RTC 初始值
RTCCR = 0x01; //RTC 开始运行
while (RTCCFG & 0x01); //等待 RTC 初始化完成
RTCIF = 0x00;
RTCIEN = 0x08; //使能 RTC 秒中断
}

void SendUart(BYTE dat) //串口发送函数
{
    while (fUartBusy);
    SBUF = dat;
    fUartBusy = 1;
}

void IapProgram(WORD addr, BYTE dat) //EEPROM 编程函数
{
    IAP_CONTR = 0x80;
    IAP_TPS = 11;
    IAP_CMD = 2;
    IAP_ADDRL = addr;
    IAP_ADDRH = addr >> 8;
    IAP_DATA = dat;
    IAP_TRIG = 0x5a;
    IAP_TRIG = 0xa5;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}

```

ISP 下载软件中“用户接口”的设置如下: (注意, 首次下载不能使能用户接口)



17.3.3 内部 RTC 时钟低功耗休眠唤醒-比较器检测电压程序

***** 本程序功能说明 *****

本例程基于 STC8H8K64U 为主控芯片的实验箱9 进行编写测试, STC8H 系列带 RTC 模块的芯片可通用参考。
读写芯片内部集成的 RTC 模块。

电路连接参考规格书 RTC 章节-RTC 实战线路图。

用 RTC 定时唤醒 MCU, 如 1 秒唤醒 1 次, 唤醒后用比较器判断外部电压: 1, 正常, 正常工作; 2, 如电压偏低, 继续休眠, 主时钟停止震荡, RTC 继续工作。

比较器正极通过电阻分压后输入到 P3.7 口, 比较器负极使用内部 1.19V 参考电压。

该分压电路的地用 I/O(P3.5)控制, I/O 设置为开漏, 不比较时, 对外设置为 1, I/O 口浮空, 省电; 比较时, 对外输出 0, 就是地!

下载时, 选择时钟 24MHZ (用户可自行修改频率).

```
#include "STC8H.h"
#include "stdio.h"
#include "intrins.h"

typedef     unsigned char   u8;
typedef     unsigned int    u16;
typedef     unsigned long   u32;

***** 用户定义宏 *****
#define  MAIN_Fosc    24000000L           //定义主时钟
#define  PrintUart    2                  //1:printf 使用 UART1; 2:printf 使用 UART2
#define  Baudrate     115200L
#define  TM           (65536 -(MAIN_Fosc/Baudrate/4))

***** 本地常量声明 *****
u8 code ledNum[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};

***** 本地变量声明 *****
bit  B_Is;
bit  B_Alarm;                         //闹钟标志
u8   ledIndex;

***** 本地函数声明 *****
void RTC_config(void);
void CMP_config(void);
void Ext_Vcc_Det(void);

***** 串口打印函数 *****
void UartInit(void)
{
#if(PrintUart == 1)
    SCON = (SCON & 0x3f) / 0x40;          //定时器时钟 IT 模式
    AUXR |= 0x40;                        //串口1 选择定时器1为波特率发生器
    AUXR &= 0xFE;
    TLI = TM;
    TH1 = TM>>8;

```

```

TR1 = 1; //定时器1 开始计时

// SCON = (SCON & 0x3f) / 0x40;
// T2L = TM;
// T2H = TM>>8;
// AUXR |= 0x15; //串口1 选择定时器2 为波特率发生器
#else
P_SW2 |= 1; //UART2 switch to: 0: P1.0 P1.1, 1: P4.6 P4.7
S2CON &= ~(1<<7); //8位数据, 1位起始位, 1位停止位, 无校验
T2L = TM;
T2H = TM>>8;
AUXR |= 0x14; //定时器2 时钟 IT 模式开始计时
#endif
}

void UartPutc(unsigned char dat)
{
#if(PrintUart == 1)
    SBUF = dat;
    while(TI == 0);
    TI = 0;
#else
    S2BUF = dat;
    while((S2CON & 2) == 0); //Clear Tx flag
#endif
}

char putchar(char c)
{
    UartPutc(c);
    return c;
}

/*****************/
void main(void)
{
    P_SW2 |= 0x80; //扩展寄存器(XFR)访问使能

    P0M1 = 0x00; P0M0 = 0x00; //设置为准双向口
    P1M1 = 0x00; P1M0 = 0x00; //设置为准双向口
    P2M1 = 0x00; P2M0 = 0x00; //设置为准双向口
    P3M1 = 0xa0; P3M0 = 0x20; //设置为准双向口 //P3.5 设置开漏模式, P3.7 设置高阻输入
    P4M1 = 0x00; P4M0 = 0x00; //设置为准双向口
    P5M1 = 0x00; P5M0 = 0x00; //设置为准双向口
    P6M1 = 0x00; P6M0 = 0x00; //设置为准双向口
    P7M1 = 0x00; P7M0 = 0x00; //设置为准双向口

    IRCDB = 0x10; //内部高速IRC时钟, 停振后被唤醒后起振, 等待多少个时钟数后给MCU供应时钟
    IAP_TPS = 40; //根据系统工作频率设置此寄存器
        // (如果系统工作频率为40MHz, 则IAP_TPS设置为40;
        // 如果系统工作频率为22.1184MHz, 则IAP_TPS设置为22)

    UartInit();
    CMP_config();
    RTC_config();
    EA = 1; //打开总中断
}

```

```

while(1)
{
    if(B_Is)
    {
        B_Is = 0;
        printf("Year=%d,Month=%d,Day=%d,Hour=%d,Minute=%d,Second=%d\r\n",
               RTCYEAR, RTCMONTH, RTCDAY, RTCHOUR, RTCMIN, RTCSEC);

        Ext_Vcc_Det(); //每秒钟检测一次外部电源,
                      //如果外部电源连接则工作,
                      //外部电源断开则进入休眠模式
    }

    if(B_Alarm)
    {
        B_Alarm = 0;
        printf("RTC      Alarm!\r\n");
    }
}

//=====
// 函数: void Ext_Vcc_Det(void)
// 描述: 外部电源检测函数。
// 参数: 无
// 返回: 无
// 版本: V1.0
//=====

void Ext_Vcc_Det(void)
{
    P35 = 0; //比较时, 对外输出 0, 做比较电路的地线
    CMPCRI |= 0x80; //使能比较器模块
    _nop_();
    _nop_();
    _nop_();
    if(CMPCRI & 0x01) //判断是否 CMP+ 电平高于 CMP-, 外部电源连接
    {
        P40 = 0; //LED Power On
        P6 = ~ledNum[ledIndex]; //输出低驱动
        ledIndex++;
        if(ledIndex > 7)
        {
            ledIndex = 0;
        }
    }
    else
    {
        CMPCRI &= ~0x80; //关闭比较器模块
        P35 = 1; //不比较时, 对外设置为 1, I/O 口浮空, 省电
        P40 = 1; //LED Power Off
        _nop_();
        _nop_();
        PCON = 0x02; //STC8H8K64U B 版本芯片使用内部 32K 时钟,
                      //休眠无法唤醒
        _nop_();
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}

```

```

        _nop_();
    }

//=====
// 函数: void CMP_config(void)
// 描述: 比较器初始化函数。
// 参数: 无
// 返回: 无
// 版本: V1.0
//=====

void CMP_config(void)
{
    CMPEXCFG = 0x00;
    // CMPEXCFG |= 0x40;                                //比较器DC 迟滞输入选择, 0:0mV;
                                                       //0x40:10mV; 0x80:20mV; 0xc0:30mV

    // CMPEXCFG &= ~0x04;                            //P3.6 为CMP- 输入脚
    CMPEXCFG |= 0x04;                                //内部 1.19V 参考电压为 CMP- 输入脚

    CMPEXCFG &= ~0x03;                            //P3.7 为CMP+ 输入脚
    // CMPEXCFG |= 0x01;                                //P5.0 为CMP+ 输入脚
    // CMPEXCFG |= 0x02;                                //P5.1 为CMP+ 输入脚
    // CMPEXCFG |= 0x03;                                //ADC 输入脚为 CMP+ 输入脚

    CMPCR2 = 0x00;
    CMPCR2 &= ~0x80;                                //比较器正向输出
    // CMPCR2 |= 0x80;                                //比较器反向输出
    CMPCR2 &= ~0x40;                                //使能 0.1us 滤波
    // CMPCR2 |= 0x40;                                //禁止 0.1us 滤波
    CMPCR2 &= ~0x3f;                                //比较器结果直接输出
    // CMPCR2 |= 0x10;                                //比较器结果经过 16 个去抖时钟后输出

    CMPCRI = 0x00;
    // CMPCRI |= 0x30;                                //使能比较器边沿中断
    // CMPCRI &= ~0x20;                                //禁止比较器上升沿中断
    // CMPCRI |= 0x20;                                //使能比较器上升沿中断
    // CMPCRI &= ~0x10;                                //禁止比较器下降沿中断
    // CMPCRI |= 0x10;                                //使能比较器下降沿中断

    CMPCRI &= ~0x02;                                //禁止比较器输出
    // CMPCRI |= 0x02;                                //使能比较器输出

    P_SW2 &= ~0x08;                                //选择 P3.4 作为比较器输出脚
    // P_SW2 |= 0x08;                                //选择 P4.1 作为比较器输出脚
    CMPCRI |= 0x80;                                //使能比较器模块
}

//=====

// 函数: void RTC_config(void)
// 描述: RTC 初始化函数。
// 参数: 无
// 返回: 无
// 版本: V1.0
//=====

void RTC_config(void)
{
    INIYEAR = 21;                                    //Y:2021
}

```

```

INIMONTH = 12;                                //M:12
INIDAY = 31;                                   //D:31
INI HOUR = 23;                                 //H:23
INIMIN = 59;                                   //M:59
INISEC = 50;                                   //S:50
INISSEC = 0;                                    //S/128:0

ALA HOUR = 0;                                  //闹钟小时
ALAMIN = 0;                                   //闹钟分钟
ALASEC = 0;                                   //闹钟秒
ALASSEC = 0;                                  //闹钟 1/128 秒

//STC8H8K64U B 版本芯片使用内部 32K 时钟, 休眠无法唤醒
//    IRC32KCR = 0x80;                          //启动内部低速IRC.
//    while (!(IRC32KCR & I));                  //等待时钟稳定
//    RTCCFG = 0x03;                            //选择内部低速IRC时钟源, 触发RTC寄存器初始化

X32KCR = 0x80 + 0x40;                         //启动外部32K晶振, 低增益+0x00, 高增益+0x40.
while (!(X32KCR & I));                        //等待时钟稳定
RTCCFG = 0x01;                                //选择外部32K时钟源, 触发RTC寄存器初始化

RTCIF = 0x00;                                 //清中断标志
RTCIEN = 0x88;                               //中断使能, 0x80:闹钟中断, 0x40:日中断, 0x20:小时中断,
//0x10:分钟中断, 0x08:秒中断, 0x04:1/2秒中断,
//0x02:1/8秒中断, 0x01:1/32秒中断
RTCCR = 0x01;                                //RTC使能

while(RTCCFG & 0x01);                         //等待初始化完成, 需要在"RTC使能"之后判断
//设置RTC时间需要32768Hz的1个周期时间,
//大约30.5us. 由于同步, 所以实际等待时间是0~30.5us.
//如果不等待设置完成就睡眠, 则RTC会由于设置没完成,
//停止计数, 唤醒后才继续完成设置并继续计数.

}

/**************** RTC 中断函数 *****/
void RTC_Isr() interrupt 13
{
    if(RTCIF & 0x80)                           //闹钟中断
    {
        P01 = !P01;
        RTCIF &= ~0x80;
        B_Alarm = I;
    }

    if(RTCIF & 0x08)                           //秒中断
    {
        P00 = !P00;
        RTCIF &= ~0x08;
        B_Is = I;
    }
}

/********************* */
//如果开启了比较器中断就需要编写对应的中断函数
void CMP_Isr() interrupt 21
{
    CMPCRI &= ~0x40;                         //清中断标志
//    P10 = CMPCRI & 0x01;                      //中断方式读取比较器比较结果
}

```


18 串口通信

产品线	串口数量				硬件 奇偶 校验	接收 超时 中断
	串口 1	串口 2	串口 3	串口 4		
STC8H1K08 系列	●	●				
STC8H1K28 系列	●	●				
STC8H3K64S4 系列	●	●	●	●		
STC8H3K64S2 系列	●	●				
STC8H8K64U 系列	●	●	●	●		
STC8H4K64TL 系列	●	●	●	●		
STC8H4K64LCD 系列	●	●	●	●		
STC8H1K08T 系列	●	●				
STC8H2K12U-A 系列	●	●				●
STC8H2K12U-B 系列	●	●			●	●
STC8H2K32U 系列	●	●			●	●
STC8G1K08-SOP8 系列	●					
STC8G1K08A-SOP8 系列	●					

STC8H 系列单片机具有 4 个全双工异步串行通信接口。每个串行口由 2 个数据缓冲器、一个移位寄存器、一个串行控制寄存器和一个波特率发生器等组成。每个串行口的数据缓冲器由 2 个互相独立的接收、发送缓冲器构成，可以同时发送和接收数据。

STC8 系列单片机的串口 1 有 4 种工作方式，其中两种方式的波特率是可变的，另两种是固定的，以供不同应用场合选用。串口 2/串口 3/串口 4 都只有两种工作方式，这两种方式的波特率都是可变的。用户可用软件设置不同的波特率和选择不同的工作方式。主机可通过查询或中断方式对接收/发送进行程序处理，使用十分灵活。

串口 1、串口 2、串口 3、串口 4 的通讯口均可以通过功能管脚的切换功能切换到多组端口，从而可以将一个通讯口分时复用为多个通讯口。

18.1 串口功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW1	A2H	S1_S[1:0]	-	-	SPI_S[1:0]	0	-	-	-
P_SW2	BAH	EAXFR	-	I2C_S[1:0]	CMPO_S	S4_S	S3_S	S2_S	-

S1_S[1:0]: 串口 1 功能脚选择位

S1_S[1:0]	RxD	TxD
00	P3.0	P3.1
01	P3.6	P3.7
10	P1.6	P1.7
11	P4.3	P4.4

S4_S: 串口 4 功能脚选择位

S4_S	RxD4	TxD4
0	P0.2	P0.3
1	P5.2	P5.3

S3_S: 串口 3 功能脚选择位

S3_S	RxD3	TxD3
0	P0.0	P0.1
1	P5.0	P5.1

S2_S: 串口 2 功能脚选择位

S2_S	RxD2	TxD2
0	P1.0	P1.1
1	P4.6	P4.7

18.2 串口相关寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
SCON	串口 1 控制寄存器	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000,0000
SBUF	串口 1 数据寄存器	99H									0000,0000
S2CON	串口 2 控制寄存器	9AH	S2SM0	-	S2SM2	S2REN	S2TB8	S2RB8	S2TI	S2RI	0100,0000
S2BUF	串口 2 数据寄存器	9BH									0000,0000
S3CON	串口 3 控制寄存器	ACH	S3SM0	S3ST3	S3SM2	S3REN	S3TB8	S3RB8	S3TI	S3RI	0000,0000
S3BUF	串口 3 数据寄存器	ADH									0000,0000
S4CON	串口 4 控制寄存器	84H	S4SM0	S4ST4	S4SM2	S4REN	S4TB8	S4RB8	S4TI	S4RI	0000,0000
S4BUF	串口 4 数据寄存器	85H									0000,0000
PCON	电源控制寄存器	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011,0000
AUXR	辅助寄存器 1	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2	0000,0001
SADDR	串口 1 从机地址寄存器	A9H									0000,0000
SADEN	串口 1 从机地址屏蔽寄存器	B9H									0000,0000

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
UR1TOCR	串口 1 接收超时控制寄存器	FD70H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
UR1TOSR	串口 1 接收超时状态寄存器	FD71H	CTOIF	-	-	-	-	-	-	-	0xxx,xxx0	
UR1TOTL	串口 1 接收超时长度寄存器	FD72H				TM[15:8]						0000,0000
UR1TOTL	串口 1 接收超时长度寄存器	FD73H				TM[7:0]						0000,0000
UR2TOCR	串口 2 接收超时控制寄存器	FD74H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
UR2TOSR	串口 2 接收超时状态寄存器	FD75H	CTOIF	-	-	-	-	-	-	-	0xxx,xxx0	
UR2TOTL	串口 2 接收超时长度寄存器	FD76H				TM[15:8]						0000,0000
UR2TOTL	串口 2 接收超时长度寄存器	FD77H				TM[7:0]						0000,0000
UR1TOTE	串口 1 接收超时长度寄存器	FD88H				TM[23:16]						0000,0000
UR2TOTE	串口 2 接收超时长度寄存器	FD89H				TM[23:16]						0000,0000
USARTCR2	串口 1 控制寄存器 2	FDC1H	-	-	-	-	-	PCEN	PS	PE	xxxx,x000	
USART2CR2	串口 2 控制寄存器 2	FDC9H	-	-	-	-	-	PCEN	PS	PE	xxxx,x000	

18.3 串口 1

18.3.1 串口 1 控制寄存器 (SCON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SCON	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI

SM0/FE: 当PCON寄存器中的SMOD0位为1时, 该位为帧错误检测标志位。当UART在接收过程中检测到一个无效停止位时, 通过UART接收器将该位置1, 必须由软件清零。当PCON寄存器中的SMOD0位为0时, 该位和SM1一起指定串口1的通信工作模式, 如下表所示:

SM0	SM1	串口1工作模式	功能说明
0	0	模式0	同步移位串行方式
0	1	模式1	可变波特率8位数据方式
1	0	模式2	固定波特率9位数据方式
1	1	模式3	可变波特率9位数据方式

当使能 FE 功能后, 串口的工作模式只能使用 SM1 进行设置, 且只能设置为模式 0 或者模式 1。

FE 当作帧错误标志位时, 此标志位必须软件写 0 清零。

SM2: 允许模式 2 或模式 3 多机通信控制位。当串口 1 使用模式 2 或模式 3 时, 如果 SM2 位为 1 且 REN 位为 1, 则接收机处于地址帧筛选状态。此时可以利用接收到的第 9 位(即 RB8)来筛选地址帧, 若 RB8=1, 说明该帧是地址帧, 地址信息可以进入 SBUF, 并使 RI 为 1, 进而在中断服务程序中再进行地址号比较; 若 RB8=0, 说明该帧不是地址帧, 应丢掉且保持 RI=0。在模式 2 或模式 3 中, 如果 SM2 位为 0 且 REN 位为 1, 接收机处于地址帧筛选被禁止状态, 不论收到的 RB8 为 0 或 1, 均可使接收到的信息进入 SBUF, 并使 RI=1, 此时 RB8 通常为校验位。模式 1 和模式 0 为非多机通信方式, 在这两种方式时, SM2 应设置为 0。

REN: 允许/禁止串口接收控制位

0: 禁止串口接收数据

1: 允许串口接收数据

TB8: 当串口 1 使用模式 2 或模式 3 时, TB8 为要发送的第 9 位数据, 按需要由软件置位或清 0。在模式 0 和模式 1 中, 该位不用。

RB8: 当串口 1 使用模式 2 或模式 3 时, RB8 为接收到的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位。在模式 0 和模式 1 中, 该位不用。

TI: 串口 1 发送中断请求标志位。在模式 0 中, 当串口发送数据第 8 位结束时, 由硬件自动将 TI 置 1, 向主机请求中断, 响应中断后 TI 必须用软件清零。在其他模式中, 则在停止位开始发送时由硬件自动将 TI 置 1, 向 CPU 发请求中断, 响应中断后 TI 必须用软件清零。

RI: 串口 1 接收中断请求标志位。在模式 0 中, 当串口接收第 8 位数据结束时, 由硬件自动将 RI 置 1, 向主机请求中断, 响应中断后 RI 必须用软件清零。在其他模式中, 串行接收到停止位的中间时刻由硬件自动将 RI 置 1, 向 CPU 发中断申请, 响应中断后 RI 必须由软件清零。

18.3.2 串口 1 数据寄存器 (SBUF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SBUF	99H								

SBUF: 串口 1 数据接收/发送缓冲区。SBUF 实际是 2 个缓冲器, 读缓冲器和写缓冲器, 两个操作分别对应两个不同的寄存器, 1 个是只写寄存器 (写缓冲器), 1 个是只读寄存器 (读缓冲器)。对 SBUF 进行读操作, 实际是读取串口接收缓冲区, 对 SBUF 进行写操作则是触发串口开始发送数据。

18.3.3 电源管理寄存器 (PCON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

SMOD: 串口 1 波特率控制位

- 0: 串口 1 的各个模式的波特率都不加倍
- 1: 串口 1 模式 1 (定时器 1 的模式 2 作为波特率发生器时有效)、模式 2、模式 3 (定时器 1 的模式 2 作为波特率发生器时有效) 的波特率加倍

SMOD0: 帧错误检测控制位

- 0: 无帧错检测功能
- 1: 使能帧错误检测功能。此时 SCON 的 SM0/FE 为 FE 功能, 即为帧错误检测标志位。

18.3.4 辅助寄存器 1 (AUXR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	T0x12	T1x12	UART_M0x6	T2R	T2_C/T	T2x12	EXTRAM	S1ST2

UART_M0x6: 串口 1 模式 0 的通讯速度控制

- 0: 串口 1 模式 0 的波特率不加倍, 固定为 Fosc/12
- 1: 串口 1 模式 0 的波特率 6 倍速, 即固定为 $Fosc/12 \times 6 = Fosc/2$

S1ST2: 串口 1 波特率发生器选择位

- 0: 选择定时器 1 作为波特率发生器
- 1: 选择定时器 2 作为波特率发生器

18.3.5 串口 1 同步模式控制寄存器 2 (USARTCR2)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0	
USARTCR2	FDC1H	-	-	-	-	-	-	PCEN	PS	PE

PCEN: 硬件自动产生校验位控制使能

- 0: 禁止硬件自动产生校验位 (串口的校验位为 TB8 设置的值)
- 1: 使能硬件自动产生校验位

PS: 硬件校验位模式选择

- 0: 硬件根据 SBUF 的值自动产生偶校验位
- 1: 硬件根据 SBUF 的值自动产生奇校验位

PE: 校验位错误标志 (必须软件清零)

- 0: 无检验错误
- 1: 有校验错误 (串口接收 DMA 过程中如果发生接收数据校验位错误, DMA 不会停止, 但校验位错误标志会一直保持直到 DMA 完成)

18.3.6 串口 1 接收超时中断控制寄存器 (UR1TOCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR1TOCR	FD70H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTO: 串口 1 接收超时中断功能控制位

0: 禁止串口 1 接收超时中断功能

1: 使能串口 1 接收超时中断功能

ENTOI: 串口 1 接收超时中断中断控制位

0: 禁止串口 1 接收超时中断

1: 使能串口 1 接收超时中断

SCALE: 串口 1 超时计数时钟源选择

0: 串口数据位率 (波特率)

1: 系统时钟

18.3.7 串口 1 超时状态寄存器 (UR1TOSR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR1TOSR	FD71H	CTOIF	-	-	-	-	-	-	TOIF

CTOIF: 写“1”清除串口 1 超时中断标志位 TOIF。**(只写)**

TOIF: 串口 1 超时中断请求标志位。**(只读)**

当发生串口 1 超时时, TOIF 会被硬件置“1”。如果 ENTOI 为 1 时会产生串口中断, 中断入口地址为原串口 1 的中断入口地址。**标志位需要软件向 CTOIF 位写“1”清零**

18.3.8 串口 1 超时长度控制寄存器 (UR1TOTE/H/L)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR1TOTE	FD88H	TM[23:16]							
UR1TOTH	FD72H	TM[15:8]							
UR1TOTL	FD73H	TM[7:0]							

TM[23:0]: 串口 1 超时时间控制位。

当串口 1 处于接收空闲状态时, 内部超时计数器根据 SCALE 寄存器所选择的时钟源进行计数, 当计数时间达到 TM 所设置的超时时间时, 便会产生超时中断。当串口 1 接收数据完成时, 复位内部超时计数器, 重新进行超时计数。

注:

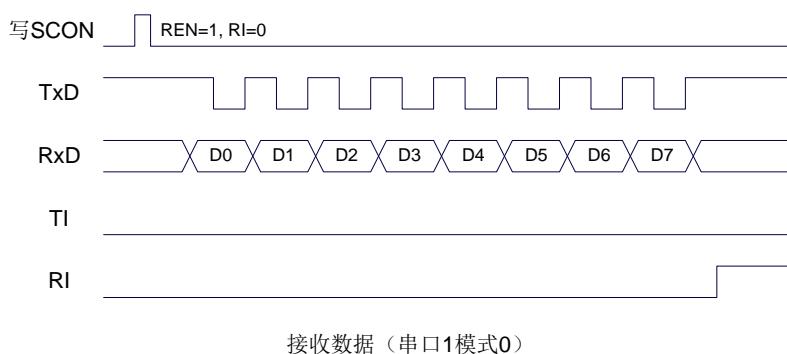
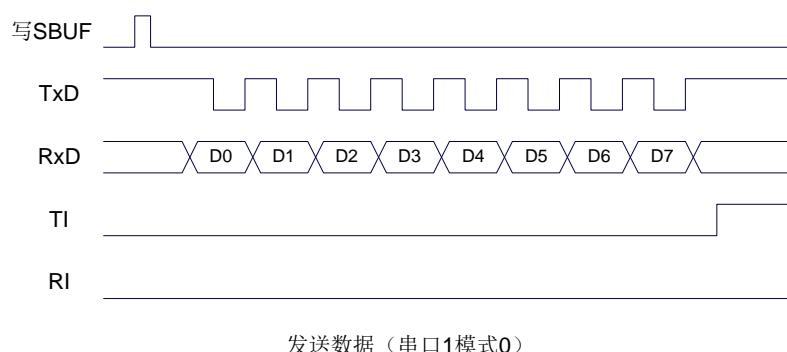
- 1、如果需要使能接收超时中断功能, 则 TM 不可设置为 0, 且 UR1TOTL、UR1TOTH、UR1TOTE 寄存器的设置必须先设置 UR1TOTL 和 UR1TOTH, 最后设置 UR1TOTE
- 2、必须使能串口接收才有接收超时功能
- 3、接收超时功能必须在正确接收到一字节数据后才能触发
- 4、正在接收过程中, 无接收超时功能

18.3.9 串口1模式0，模式0波特率计算公式

当串口1选择工作模式为模式0时，串行通信接口工作在同步移位寄存器模式，当串行口模式0的通信速度设置位 `UART_M0x6` 为0时，其波特率固定为系统时钟时钟的12分频(`SYSclk/12`)；当设置`UART_M0x6`为1时，其波特率固定为系统时钟频率的2分频(`SYSclk/2`)。`RxD`为串行通讯的数据口，`TxD`为同步移位脉冲输出脚，发送、接收的是8位数据，低位在先。

模式0的发送过程：当主机执行将数据写入发送缓冲器 `SBUF` 指令时启动发送，串行口即将8位数据以 `SYSclk/12` 或 `SYSclk/2` (由 `UART_M0x6` 确定是12分频还是2分频) 的波特率从 `RxD` 管脚输出(从低位到高位)，发送完中断标志 `TI` 置1，`TxD` 管脚输出同步移位脉冲信号。当写信号有效后，相隔一个时钟，发送控制端 `SEND` 有效(高电平)，允许 `RxD` 发送数据，同时允许 `TxD` 输出同步移位脉冲。一帧(8位)数据发送完毕时，各控制端均恢复原状态，只有 `TI` 保持高电平，呈中断申请状态。在再次发送数据前，必须用软件将 `TI` 清0。

模式0的接收过程：首先将接收中断请求标志 `RI` 清零并置位允许接收控制位 `REN` 时启动模式0接收过程。启动接收过程后，`RxD` 为串行数据输入端，`TxD` 为同步脉冲输出端。串行接收的波特率为 `SYSclk/12` 或 `SYSclk/2` (由 `UART_M0x6` 确定是12分频还是2分频)。当接收完成一帧数据(8位)后，控制信号复位，中断标志 `RI` 被置1，呈中断申请状态。当再次接收时，必须通过软件将 `RI` 清0



工作于模式0时，必须清0多机通信控制位 `SM2`，使之不影响 `TB8` 位和 `RB8` 位。由于波特率固定为 `SYSclk/12` 或 `SYSclk/2`，无需定时器提供，直接由单片机的时钟作为同步移位脉冲。

串口 1 模式 0 的波特率计算公式如下表所示 (SYSclk 为系统工作频率):

UART_M0x6	波特率计算公式
0	波特率 = $\frac{\text{SYSclk}}{12}$
1	波特率 = $\frac{\text{SYSclk}}{2}$

18.3.10 串口 1 模式 1，模式 1 波特率计算公式

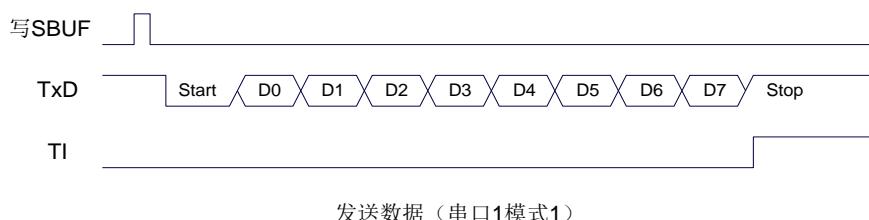
当软件设置 SCON 的 SM0、SM1 为“01”时，串行口 1 则以模式 1 进行工作。此模式为 8 位 UART 格式，一帧信息为 10 位：1 位起始位，8 位数据位（低位在先）和 1 位停止位。波特率可变，即可根据需要进行设置波特率。TxD 为数据发送口，RxD 为数据接收口，串行口全双工接受/发送。

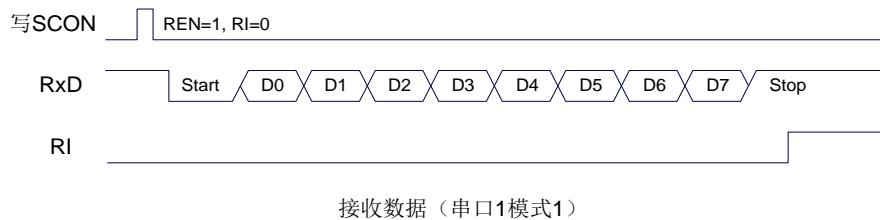
模式 1 的发送过程：串行通信模式发送时，数据由串行发送端 TxD 输出。当主机执行一条写 SBUF 的指令就启动串行通信的发送，写“SBUF”信号还把“1”装入发送移位寄存器的第 9 位，并通知 TX 控制单元开始发送。移位寄存器将数据不断右移送 TxD 端口发送，在数据的左边不断移入“0”作补充。当数据的最高位移到移位寄存器的输出位置，紧跟其后的是第 9 位“1”，在它的左边各位全为“0”，这个状态条件，使 TX 控制单元作最后一次移位输出，然后使允许发送信号“SEND”失效，完成一帧信息的发送，并置位中断请求位 TI，即 TI=1，向主机请求中断处理。

模式 1 的接收过程：当软件置位接收允许标志位 REN，即 REN=1 时，接收器便对 RxD 端口的信号进行检测，当检测到 RxD 端口发送从“1”→“0”的下降沿跳变时就启动接收器准备接收数据，并立即复位波特率发生器的接收计数器，将 1FFH 装入移位寄存器。接收的数据从接收移位寄存器的右边移入，已装入的 1FFH 向左边移出，当起始位“0”移到移位寄存器的最左边时，使 RX 控制器作最后一次移位，完成一帧的接收。若同时满足以下两个条件：

- RI=0；
- SM2=0 或接收到的停止位为 1。

则接收到的数据有效，实现装载入 SBUF，停止位进入 RB8，RI 标志位被置 1，向主机请求中断，若上述两条件不能同时满足，则接收到的数据作废并丢失，无论条件满足与否，接收器重又检测 RxD 端口上的“1”→“0”的跳变，继续下一帧的接收。接收有效，在响应中断后，RI 标志位必须由软件清 0。通常情况下，串行通信工作于模式 1 时，SM2 设置为“0”。





串口 1 的波特率是可变的，其波特率可由定时器 1 或者定时器 2 产生。当定时器采用 1T 模式时 (12 倍速)，相应的波特率的速度也会相应提高 12 倍。

串口 1 模式 1 的波特率计算公式如下表所示: (SYSclk 为系统工作频率)

选择定时器	定时器速度	重装值计算公式	波特率
定时器2	1T	定时器2重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器2重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$
定时器1 模式0	1T	定时器1重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器1重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$
定时器1 模式2	1T	定时器1重载值 = $256 - \frac{2^{\text{SMOD}} \times SYSclk}{32 \times \text{波特率}}$	波特率 = $\frac{2^{\text{SMOD}} \times SYSclk}{32 \times (256 - \text{定时器重装数})}$
	12T	定时器1重载值 = $256 - \frac{2^{\text{SMOD}} \times SYSclk}{12 \times 32 \times \text{波特率}}$	波特率 = $\frac{2^{\text{SMOD}} \times SYSclk}{12 \times 32 \times (256 - \text{定时器重装数})}$

下面为常用频率与常用波特率所对应定时器的重载值

频率 (MHz)	波特率	定时器 2		定时器 1 模式 0		定时器 1 模式 2			
		1T 模式	12T 模式	1T 模式	12T 模式	SMOD=1		SMOD=0	
						1T 模式	12T 模式	1T 模式	12T 模式
11.0592	115200	FFE8H	FFF8H	FFE8H	FFF8H	FAH	-	FDH	-
	57600	FFD0H	FFFCH	FFD0H	FFFCH	F4H	FFH	FAH	-
	38400	FFB8H	FFF8H	FFB8H	FFF8H	EEH	-	F7H	-
	19200	FF70H	FFF4H	FF70H	FFF4H	DCH	FDH	EEH	-
	9600	FEE0H	FFE8H	FEE0H	FFE8H	B8H	FAH	DCH	FDH
18.432	115200	FFD8H	-	FFD8H	-	F6H	-	FBH	-
	57600	FFB0H	-	FFB0H	-	ECH	-	F6H	-
	38400	FF88H	FFF6H	FF88H	FFF6H	E2H	-	F1H	-
	19200	FF10H	FFECH	FF10H	FFECH	C4H	FBH	E2H	-
	9600	FE20H	FFD8H	FE20H	FFD8H	88H	F6H	C4H	FBH
22.1184	115200	FFD0H	FFFCH	FFD0H	FFFCH	F4H	FFH	FAH	-
	57600	FFA0H	FFF8H	FFA0H	FFF8H	E8H	FEH	F4H	FFH
	38400	FF70H	FFF4H	FF70H	FFF4H	DCH	FDH	EEH	-
	19200	FEE0H	FFE8H	FEE0H	FFE8H	B8H	FAH	DCH	FDH
	9600	FDC0H	FFD0H	FDC0H	FFD0H	70H	F4H	B8H	FAH

18.3.11 串口 1 模式 2，模式 2 波特率计算公式

当 SM0、SM1 两位为 10 时，串行口 1 工作在模式 2。串行口 1 工作模式 2 为 9 位数据异步通信 UART 模式，其一帧的信息由 11 位组成：1 位起始位，8 位数据位（低位在先），1 位可编程位（第 9 位数据）和 1 位停止位。发送时可编程位（第 9 位数据）由 SCON 中的 TB8 提供，可软件设置为 1 或 0，或者可将 PSW 中的奇/偶校验位 P 值装入 TB8（TB8 既可作为多机通信中的地址数据标志位，又可作为数据的奇偶校验位）。接收时第 9 位数据装入 SCON 的 RB8。TxD 为发送端口，RxD 为接收端口，以全双工模式进行接收/发送。

模式 2 的波特率固定为系统时钟的 64 分频或 32 分频（取决于 PCON 中 SMOD 的值）

串口 1 模式 2 的波特率计算公式如下表所示（SYSclk 为系统工作频率）：

SMOD	波特率计算公式
0	波特率 = $\frac{\text{SYSclk}}{64}$
1	波特率 = $\frac{\text{SYSclk}}{32}$

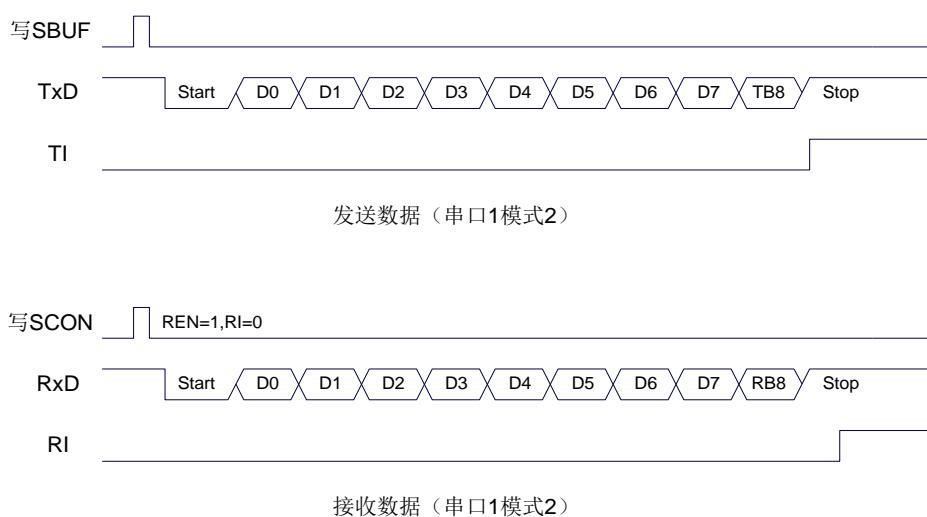
模式 2 和模式 1 相比，除波特率发生源略有不同，发送时由 TB8 提供给移位寄存器第 9 数据位不同外，其余功能结构均基本相同，其接收/发送操作过程及时序也基本相同。

当接收器接收完一帧信息后必须同时满足下列条件：

- RI=0
- SM2=0 或者 SM2=1 且接收到的第 9 数据位 RB8=1。

当上述两条条件同时满足时，才将接收到的移位寄存器的数据装入 SBUF 和 RB8 中，RI 标志位被置 1，并向主机请求中断处理。如果上述条件有一个不满足，则刚接收到移位寄存器中的数据无效而丢失，也不置位 RI。无论上述条件满足与否，接收器又重新开始检测 RxD 输入端口的跳变信息，接收下一帧的输入信息。在模式 2 中，接收到的停止位与 SBUF、RB8 和 RI 无关。

通过软件对 SCON 中的 SM2、TB8 的设置以及通信协议的约定，为多机通信提供了方便。



18.3.12 串口 1 模式 3，模式 3 波特率计算公式

当 SM0、SM1 两位为 11 时，串行口 1 工作在模式 3。串行通信模式 3 为 9 位数据异步通信 UART 模式，其一帧的信息由 11 位组成：1 位起始位，8 位数据位（低位在先），1 位可编程位（第 9 位数据）和 1 位停止位。发送时可编程位（第 9 位数据）由 SCON 中的 TB8 提供，可软件设置为 1 或 0，或者可将 PSW 中的奇/偶校验位 P 值装入 TB8（TB8 既可作为多机通信中的地址数据标志位，又可作为数据的奇偶校验位）。接收时第 9 位数据装入 SCON 的 RB8。TxD 为发送端口，RxD 为接收端口，以全双工模式进行接收/发送。

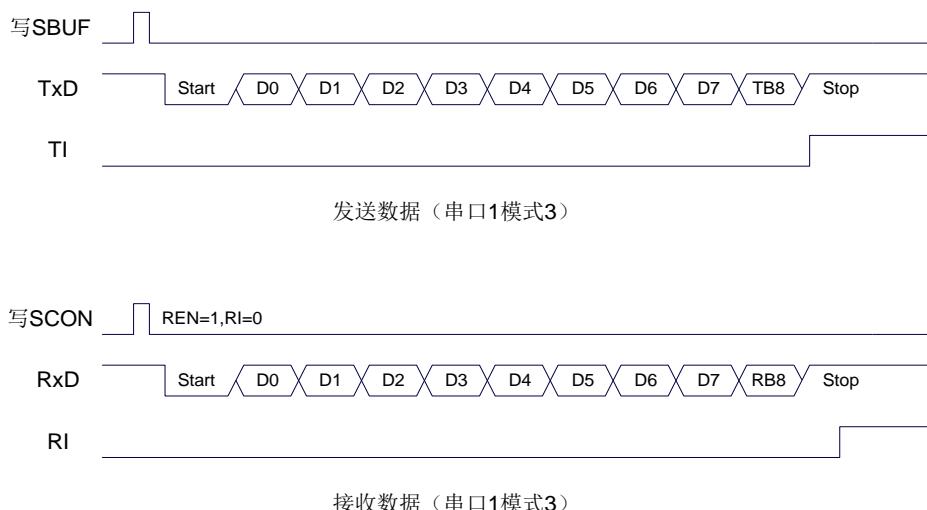
模式 3 和模式 1 相比，除发送时由 TB8 提供给移位寄存器第 9 数据位不同外，其余功能结构均基本相同，其接收‘发送操作过程及时序也基本相同。

当接收器接收完一帧信息后必须同时满足下列条件：

- RI=0
- SM2=0 或者 SM2=1 且接收到的第 9 数据位 RB8=1。

当上述两条条件同时满足时，才将接收到的移位寄存器的数据装入 SBUF 和 RB8 中，RI 标志位被置 1，并向主机请求中断处理。如果上述条件有一个不满足，则刚接收到移位寄存器中的数据无效而丢失，也不置位 RI。无论上述条件满足与否，接收器又重新开始检测 RxD 输入端口的跳变信息，接收下一帧的输入信息。在模式 3 中，接收到的停止位与 SBUF、RB8 和 RI 无关。

通过软件对 SCON 中的 SM2、TB8 的设置以及通信协议的约定，为多机通信提供了方便。



串口 1 模式 3 的波特率计算公式与模式 1 是完全相同的。请参考模式 1 的波特率计算公式。

18.3.13 自动地址识别，从机地址控制寄存器（SADDR，SADEN）

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SADDR	A9H								
SADEN	B9H								

SADDR: 从机地址寄存器

SADEN: 从机地址屏蔽位寄存器

自动地址识别功能典型应用在多机通讯领域，其主要原理是从机系统通过硬件比较功能来识别来自于主机串口数据流中的地址信息，通过寄存器 SADDR 和 SADEN 设置的本机的从机地址，硬件自动对从机地址进行过滤，当来自于主机的从机地址信息与本机所设置的从机地址相匹配时，硬件产生串口中断；否则硬件自动丢弃串口数据，而不产生中断。当众多处于空闲模式的从机链接在一起时，只有从机地址相匹配的从机才会从空闲模式唤醒，从而可以大大降低从机 MCU 的功耗，即使从机处于正常工作状态也可避免不停地进入串口中断而降低系统执行效率。

要使用串口的自动地址识别功能，首先需要将参与通讯的 MCU 的串口通讯模式设置为模式 2 或者模式 3（通常都选择波特率可变的模式 3，因为模式 2 的波特率是固定的，不便于调节），并开启从机的 SCON 的 SM2 位。对于串口模式 2 或者模式 3 的 9 位数据位中，第 9 位数据（存放在 RB8 中）为地址/数据的标志位，当第 9 位数据为 1 时，表示前面的 8 位数据（存放在 SBUF 中）为地址信息。当 SM2 被设置为 1 时，从机 MCU 会自动过滤掉非地址数据（第 9 位为 0 的数据），而对 SBUF 中的地址数据（第 9 位为 1 的数据）自动与 SADDR 和 SADEN 所设置的本机地址进行比较，若地址相匹配，则会将 RI 置“1”，并产生中断，否则不予处理本次接收的串口数据。

从机地址的设置是通过 SADDR 和 SADEN 两个寄存器进行设置的。SADDR 为从机地址寄存器，里面存放本机的从机地址。SADEN 为从机地址屏蔽位寄存器，用于设置地址信息中的忽略位，设置方法如下：

例如

SADDR = 11001010

SADEN = 10000001

则匹配地址为 1xxxxxx0

即，只要主机送出的地址数据中的 bit0 为 0 且 bit7 为 1 就可以和本机地址相匹配

再例如

SADDR = 11001010

SADEN = 00001111

则匹配地址为 xxxx1010

即，只要主机送出的地址数据中的低 4 位为 1010 就可以和本机地址相匹配，而高 4 为被忽略，可以为任意值。

主机可以使用广播地址（FFH）同时选中所有的从机来进行通讯。

18.4 串口 2

18.4.1 串口 2 控制寄存器 (S2CON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S2CON	9AH	S2SM0	-	S2SM2	S2REN	S2TB8	S2RB8	S2TI	S2RI

S2SM0: 指定串口2的通信工作模式, 如下表所示:

S2SM0	串口2工作模式	功能说明
0	模式0	可变波特率8位数据方式
1	模式1	可变波特率9位数据方式

S2SM2: 允许串口 2 在模式 1 时允许多机通信控制位。在模式 1 时, 如果 S2SM2 位为 1 且 S2REN 位为 1, 则接收机处于地址帧筛选状态。此时可以利用接收到的第 9 位 (即 S2RB8) 来筛选地址帧: 若 S2RB8=1, 说明该帧是地址帧, 地址信息可以进入 S2BUF, 并使 S2RI 为 1, 进而在中断服务程序中再进行地址号比较; 若 S2RB8=0, 说明该帧不是地址帧, 应丢掉且保持 S2RI=0。在模式 1 中, 如果 S2SM2 位为 0 且 S2REN 位为 1, 接收机处于地址帧筛选被禁止状态。不论收到的 S2RB8 为 0 或 1, 均可使接收到的信息进入 S2BUF, 并使 S2RI=1, 此时 S2RB8 通常为校验位。模式 0 为非多机通信方式, 在这种方式时, 要设置 S2SM2 应为 0。

S2REN: 允许/禁止串口接收控制位

0: 禁止串口接收数据

1: 允许串口接收数据

S2TB8: 当串口 2 使用模式 1 时, S2TB8 为要发送的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位, 按需要由软件置位或清 0。在模式 0 中, 该位不用。

S2RB8: 当串口 2 使用模式 1 时, S2RB8 为接收到的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位。在模式 0 中, 该位不用。

S2TI: 串口 2 发送中断请求标志位。在停止位开始发送时由硬件自动将 S2TI 置 1, 向 CPU 发请求中断, 响应中断后 S2TI 必须用软件清零。

S2RI: 串口 2 接收中断请求标志位。串行接收到停止位的中间时刻由硬件自动将 S2RI 置 1, 向 CPU 发中断申请, 响应中断后 S2RI 必须由软件清零。

18.4.2 串口 2 数据寄存器 (S2BUF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S2BUF	9BH								

S2BUF: 串口 2 数据接收/发送缓冲区。S2BUF 实际是 2 个缓冲器, 读缓冲器和写缓冲器, 两个操作分别对应两个不同的寄存器, 1 个是只写寄存器 (写缓冲器), 1 个是只读寄存器 (读缓冲器)。对 S2BUF 进行读操作, 实际是读取串口接收缓冲区, 对 S2BUF 进行写操作则是触发串口开始发送数据。

18.4.3 串口 2 同步模式控制寄存器 2 (USART2CR2)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
USART2CR2	FDC9H	-	-	-	-	-	PCEN	PS	PE

PCEN: 硬件自动产生校验位控制使能

0: 禁止硬件自动产生校验位 (串口的校验位为 S2TB8 设置的值)

1: 使能硬件自动产生校验位

PS: 硬件校验位模式选择

0: 硬件根据 S2BUF 的值自动产生偶校验位

1: 硬件根据 S2BUF 的值自动产生奇校验位

PE: 校验位错误标志 (必须软件清零)

0: 无检验错误

1: 有校验错误 (串口接收 DMA 过程中如果发生接收数据校验位错误, DMA 不会停止, 但校验位错误标志会一直保持直到 DMA 完成)

18.4.4 串口 2 接收超时中断控制寄存器 (UR2TOCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR2TOCR	FD74H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTO: 串口 2 接收超时中断功能控制位

0: 禁止串口 2 接收超时中断功能

1: 使能串口 2 接收超时中断功能

ENTOI: 串口 2 接收超时中断控制位

0: 禁止串口 2 接收超时中断

1: 使能串口 2 接收超时中断

SCALE: 串口 2 超时计数时钟源选择

0: 串口数据位率 (波特率)

1: 系统时钟

18.4.5 串口 2 超时状态寄存器 (UR2TOSR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR2TOSR	FD75H	CTOIF	-	-	-	-	-	-	TOIF

CTOIF: 写“1”清除串口超时中断标志位 TOIF。[\(只写\)](#)

TOIF: 串口 2 超时中断请求标志位。[\(只读\)](#)

当发生串口 2 超时时, TOIF 会被硬件置“1”。如果 ENTOI 为 1 时会产生串口中断, 中断入口地址为原串口 2 的中断入口地址。[标志位需要软件向 CTOIF 位写“1”清零](#)

18.4.6 串口 2 超时长度控制寄存器 (UR2TOTE/H/L)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
UR2TOTE	FD89H						TM[23:16]		
UR2TOTh	FD76H						TM[15:8]		
UR2TOTL	FD77H						TM[7:0]		

TM[23:0]: 串口 2 超时时间控制位。

当串口 2 处于接收空闲状态时，内部超时计数器根据 SCALE 寄存器所选择的时钟源进行计数，当计数时间达到 TM 所设置的超时时间时，便会产生超时中断。当串口 2 接收数据完成时，复位内部超时计数器，重新进行超时计数。

注:

- 1、如果需要使能接收超时中断功能，则 TM 不可设置为 0，且 UR1TOTL、UR1TOTh、UR1TOTE 寄存器的设置必须先设置 UR1TOTL 和 UR1TOTh，最后设置 UR1TOTE
- 2、必须使能串口接收才有接收超时功能
- 3、接收超时功能必须在正确接收到一字节数据后才能触发
- 4、正在接收过程中，无接收超时功能

18.4.7 串口 2 模式 0，模式 0 波特率计算公式

串行口 2 的模式 0 为 8 位数据位可变波特率 UART 工作模式。此模式一帧信息为 10 位: 1 位起始位, 8 位数据位 (低位在先) 和 1 位停止位。波特率可变, 可根据需要进行设置波特率。TxD2 为数据发送口, RxD2 为数据接收口, 串行口全双工接受/发送。



串口 2 的波特率是可变的, 其波特率由定时器 2 产生。当定时器采用 1T 模式时 (12 倍速), 相应的波特率的速度也会相应提高 12 倍。

串口 2 模式 0 的波特率计算公式如下表所示: (SYSclk 为系统工作频率)

选择定时器	定时器速度	重装值计算公式	波特率
定时器2	1T	定时器2重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器2重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$

18.4.8 串口 2 模式 1，模式 1 波特率计算公式

串行口 2 的模式 1 为 9 位数据位可变波特率 UART 工作模式。此模式一帧信息为 11 位：1 位起始位，9 位数据位（低位在先）和 1 位停止位。波特率可变，可根据需要进行设置波特率。TxD2 为数据发送口，RxD2 为数据接收口，串行口全双工接受/发送。



串口 2 模式 1 的波特率计算公式与模式 0 是完全相同的。请参考模式 0 的波特率计算公式。

18.5 串口 3

18.5.1 串口 3 控制寄存器 (S3CON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S3CON	ACH	S3SM0	S3ST3	S3SM2	S3REN	S3TB8	S3RB8	S3TI	S3RI

S3SM0: 指定串口3的通信工作模式, 如下表所示:

S3SM0	串口3工作模式	功能说明
0	模式0	可变波特率8位数据方式
1	模式1	可变波特率9位数据方式

S3ST3: 选择串口 3 的波特率发生器

- 0: 选择定时器 2 为串口 3 的波特率发生器
- 1: 选择定时器 3 为串口 3 的波特率发生器

S3SM2: 允许串口 3 在模式 1 时允许多机通信控制位。在模式 1 时, 如果 S3SM2 位为 1 且 S3REN 位为 1, 则接收机处于地址帧筛选状态。此时可以利用接收到的第 9 位 (即 S3RB8) 来筛选地址帧: 若 S3RB8=1, 说明该帧是地址帧, 地址信息可以进入 S3BUF, 并使 S3RI 为 1, 进而在中断服务程序中再进行地址号比较; 若 S3RB8=0, 说明该帧不是地址帧, 应丢掉且保持 S3RI=0。在模式 1 中, 如果 S3SM2 位为 0 且 S3REN 位为 1, 接收机处于地址帧筛选被禁止状态。不论收到的 S3RB8 为 0 或 1, 均可使接收到的信息进入 S3BUF, 并使 S3RI=1, 此时 S3RB8 通常为校验位。模式 0 为非多机通信方式, 在这种方式时, 要设置 S3SM2 应为 0。

S3REN: 允许/禁止串口接收控制位

- 0: 禁止串口接收数据
- 1: 允许串口接收数据

S3TB8: 当串口 3 使用模式 1 时, S3TB8 为要发送的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位, 按需要由软件置位或清 0。在模式 0 中, 该位不用。

S3RB8: 当串口 3 使用模式 1 时, S3RB8 为接收到的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位。在模式 0 中, 该位不用。

S3TI: 串口 3 发送中断请求标志位。在停止位开始发送时由硬件自动将 S3TI 置 1, 向 CPU 发请求中断, 响应中断后 S3TI 必须用软件清零。

S3RI: 串口 3 接收中断请求标志位。串行接收到停止位的中间时刻由硬件自动将 S3RI 置 1, 向 CPU 发中断申请, 响应中断后 S3RI 必须由软件清零。

18.5.2 串口 3 数据寄存器 (S3BUF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S3BUF	ADH								

S3BUF: 串口 3 数据接收/发送缓冲区。S3BUF 实际是 2 个缓冲器, 读缓冲器和写缓冲器, 两个操作分别对应两个不同的寄存器, 1 个是只写寄存器 (写缓冲器), 1 个是只读寄存器 (读缓冲器)。对 S3BUF 进行读操作, 实际是读取串口接收缓冲区, 对 S3BUF 进行写操作则是触发串口开始发送数据。

18.5.3 串口3模式0，模式0波特率计算公式

串行口3的模式0为8位数据位可变波特率UART工作模式。此模式一帧信息为10位：1位起始位，8位数据位（低位在先）和1位停止位。波特率可变，可根据需要进行设置波特率。TxD3为数据发送口，RxD3为数据接收口，串行口全双工接受/发送。



串口3的波特率是可变的，其波特率可由定时器2或定时器3产生。当定时器采用1T模式时（12倍速），相应的波特率的速度也会相应提高12倍。

串口3模式0的波特率计算公式如下表所示：(SYSclk为系统工作频率)

选择定时器	定时器速度	重装值计算公式	波特率
定时器2	1T	定时器2重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器2重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$
定时器3	1T	定时器3重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器3重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$

18.5.4 串口3模式1，模式1波特率计算公式

串行口3的模式1为9位数据位可变波特率UART工作模式。此模式一帧信息为11位：1位起始位，9位数据位（低位在先）和1位停止位。波特率可变，可根据需要进行设置波特率。TxD3为数据发送口，RxD3为数据接收口，串行口全双工接受/发送。



串口3模式1的波特率计算公式与模式0是完全相同的。请参考模式0的波特率计算公式。

18.6 串口 4

18.6.1 串口 4 控制寄存器 (S4CON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S4CON	84H	S4SM0	S4ST4	S4SM2	S4REN	S4TB8	S4RB8	S4TI	S4RI

S4SM0: 指定串口4的通信工作模式, 如下表所示:

S4SM0	串口4工作模式	功能说明
0	模式0	可变波特率8位数据方式
1	模式1	可变波特率9位数据方式

S4ST4: 选择串口 4 的波特率发生器

- 0: 选择定时器 2 为串口 4 的波特率发生器
- 1: 选择定时器 4 为串口 4 的波特率发生器

S4SM2: 允许串口 4 在模式 1 时允许多机通信控制位。在模式 1 时, 如果 S4SM2 位为 1 且 S4REN 位为 1, 则接收机处于地址帧筛选状态。此时可以利用接收到的第 9 位 (即 S4RB8) 来筛选地址帧: 若 S4RB8=1, 说明该帧是地址帧, 地址信息可以进入 S4BUF, 并使 S4RI 为 1, 进而在中断服务程序中再进行地址号比较; 若 S4RB8=0, 说明该帧不是地址帧, 应丢掉且保持 S4RI=0。在模式 1 中, 如果 S4SM2 位为 0 且 S4REN 位为 1, 接收机处于地址帧筛选被禁止状态。不论收到的 S4RB8 为 0 或 1, 均可使接收到的信息进入 S4BUF, 并使 S4RI=1, 此时 S4RB8 通常为校验位。模式 0 为非多机通信方式, 在这种方式时, 要设置 S4SM2 应为 0。

S4REN: 允许/禁止串口接收控制位

- 0: 禁止串口接收数据
- 1: 允许串口接收数据

S4TB8: 当串口 4 使用模式 1 时, S4TB8 为要发送的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位, 按需要由软件置位或清 0。在模式 0 中, 该位不用。

S4RB8: 当串口 4 使用模式 1 时, S4RB8 为接收到的第 9 位数据, 一般用作校验位或者地址帧/数据帧标志位。在模式 0 中, 该位不用。

S4TI: 串口 4 发送中断请求标志位。在停止位开始发送时由硬件自动将 S4TI 置 1, 向 CPU 发请求中断, 响应中断后 S4TI 必须用软件清零。

S4RI: 串口 4 接收中断请求标志位。串行接收到停止位的中间时刻由硬件自动将 S4RI 置 1, 向 CPU 发中断申请, 响应中断后 S4RI 必须由软件清零。

18.6.2 串口 4 数据寄存器 (S4BUF)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
S4BUF	85H								

S4BUF: 串口 4 数据接收/发送缓冲区。S4BUF 实际是 2 个缓冲器, 读缓冲器和写缓冲器, 两个操作分别对应两个不同的寄存器, 1 个是只写寄存器 (写缓冲器), 1 个是只读寄存器 (读缓冲器)。对 S4BUF 进行读操作, 实际是读取串口接收缓冲区, 对 S4BUF 进行写操作则是触发串口开始发送数据。

18.6.3 串口 4 模式 0，模式 0 波特率计算公式

串行口 4 的模式 0 为 8 位数据位可变波特率 UART 工作模式。此模式一帧信息为 10 位: 1 位起始位, 8 位数据位 (低位在先) 和 1 位停止位。波特率可变, 可根据需要进行设置波特率。TxD4 为数据发送口, RxD4 为数据接收口, 串行口全双工接受/发送。



串口 4 的波特率是可变的, 其波特率可由定时器 2 或定时器 4 产生。当定时器采用 1T 模式时 (12 倍速), 相应的波特率的速度也会相应提高 12 倍。

串口 4 模式 0 的波特率计算公式如下表所示: (SYSclk 为系统工作频率)

选择定时器	定时器速度	重装值计算公式	波特率
定时器2	1T	定时器2重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器2重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$
定时器4	1T	定时器4重载值 = $65536 - \frac{SYSclk}{4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{4 \times (65536 - \text{定时器重装数})}$
	12T	定时器4重载值 = $65536 - \frac{SYSclk}{12 \times 4 \times \text{波特率}}$	波特率 = $\frac{SYSclk}{12 \times 4 \times (65536 - \text{定时器重装数})}$

18.6.4 串口 4 模式 1，模式 1 波特率计算公式

串行口 4 的模式 1 为 9 位数据位可变波特率 UART 工作模式。此模式一帧信息为 11 位：1 位起始位，9 位数据位（低位在先）和 1 位停止位。波特率可变，可根据需要进行设置波特率。TxD4 为数据发送口，RxD4 为数据接收口，串行口全双工接受/发送。

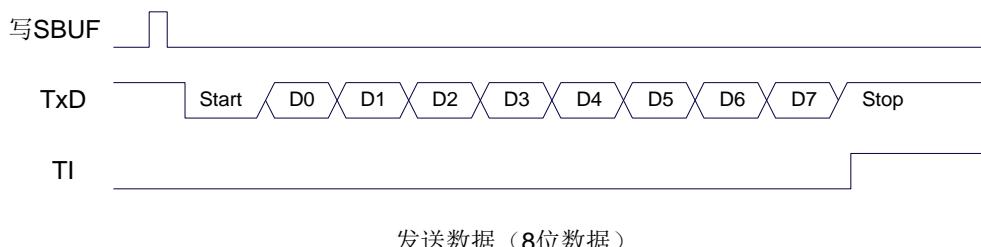


串口 4 模式 1 的波特率计算公式与模式 0 是完全相同的。请参考模式 0 的波特率计算公式。

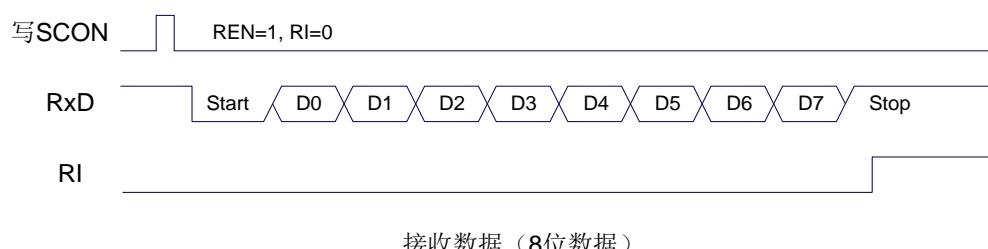
18.7 串口注意事项

关于串口中断请求有如下问题需要注意: (串口 1、串口 2、串口 3、串口 4 均类似, 下面以串口 1 为例进行说明)

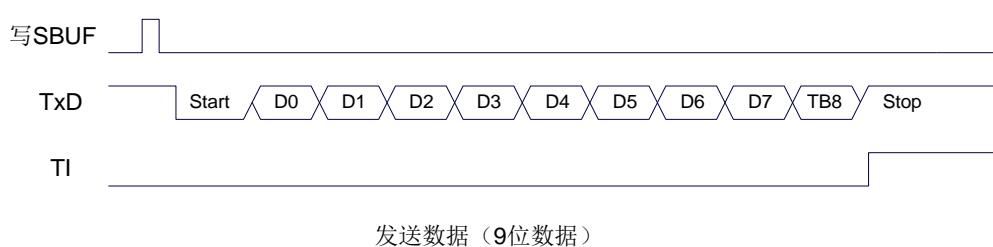
8 位数据模式时, 发送完成约 1/3 个停止位后产生 TI 中断请求, 如下图所示:



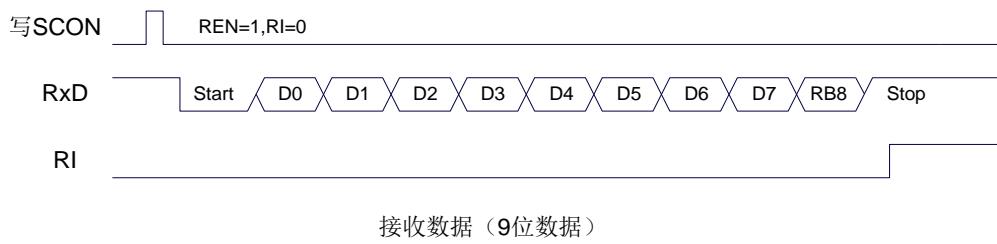
8 位数据模式时, 接收完成一半个停止位后产生 RI 中断请求, 如下图所示:



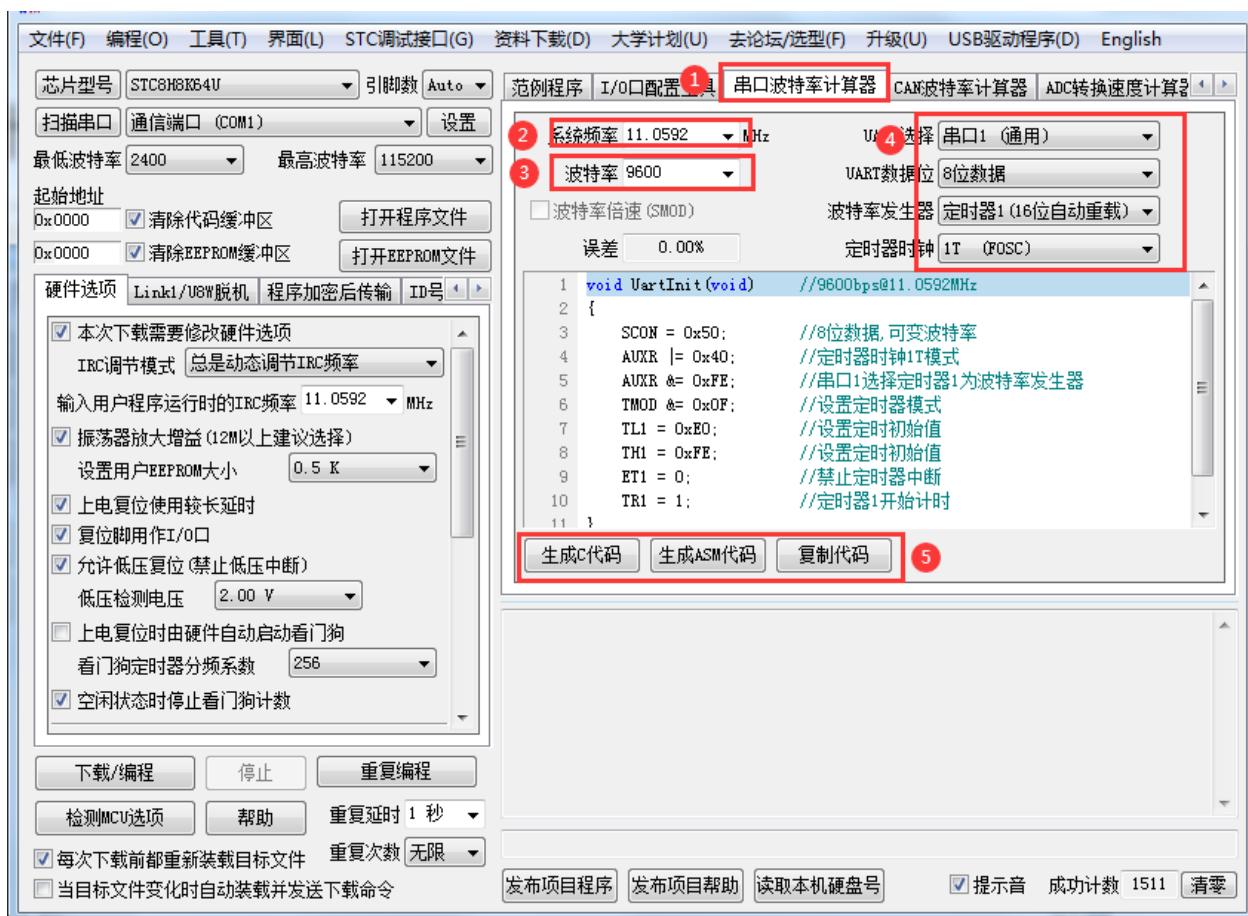
9 位数据模式时, 发送完成约 1/3 个停止位后产生 TI 中断请求:



9 位数据模式时, 一半个停止位后产生 RI 中断请求, 如下图所示:

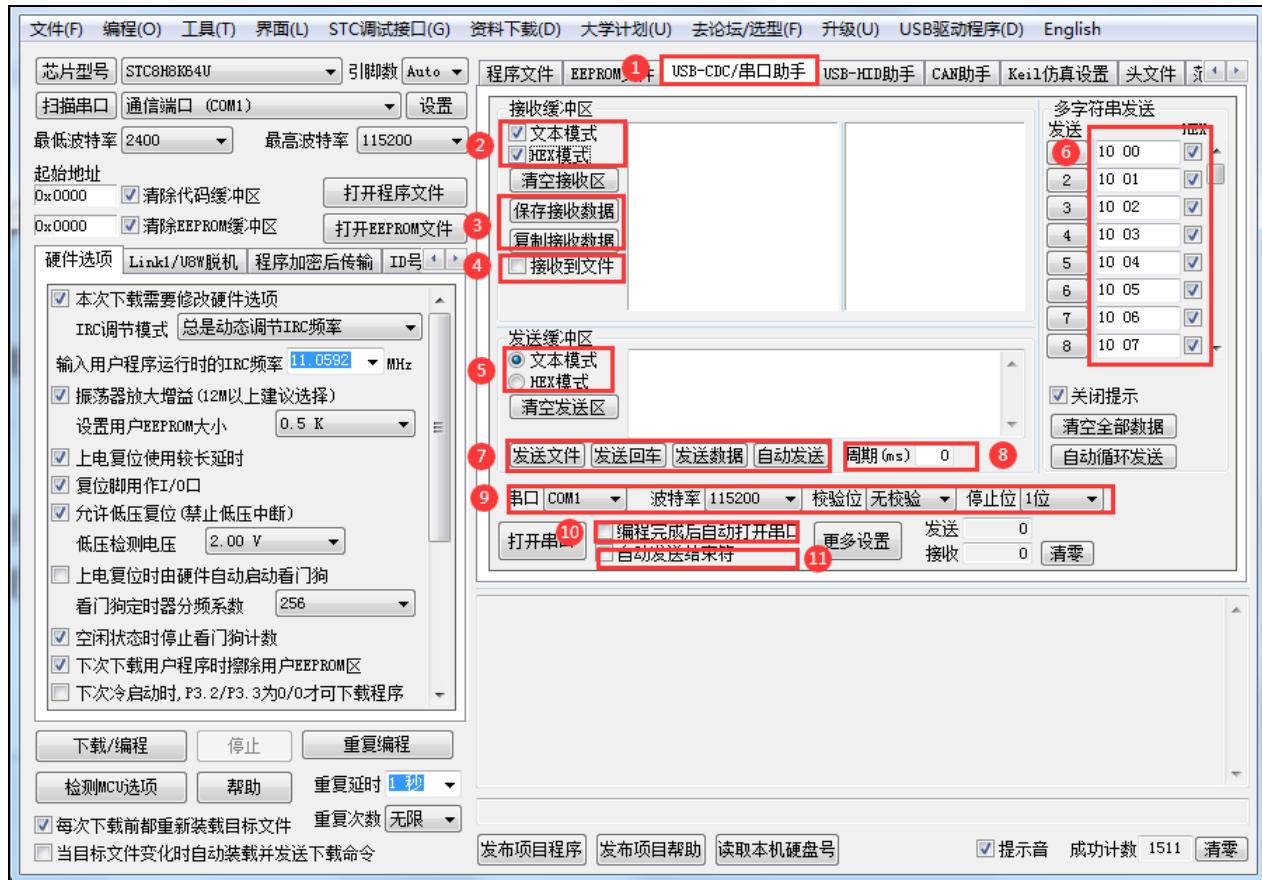


18.8 AiCube-ISP | 串口波特率计算器工具



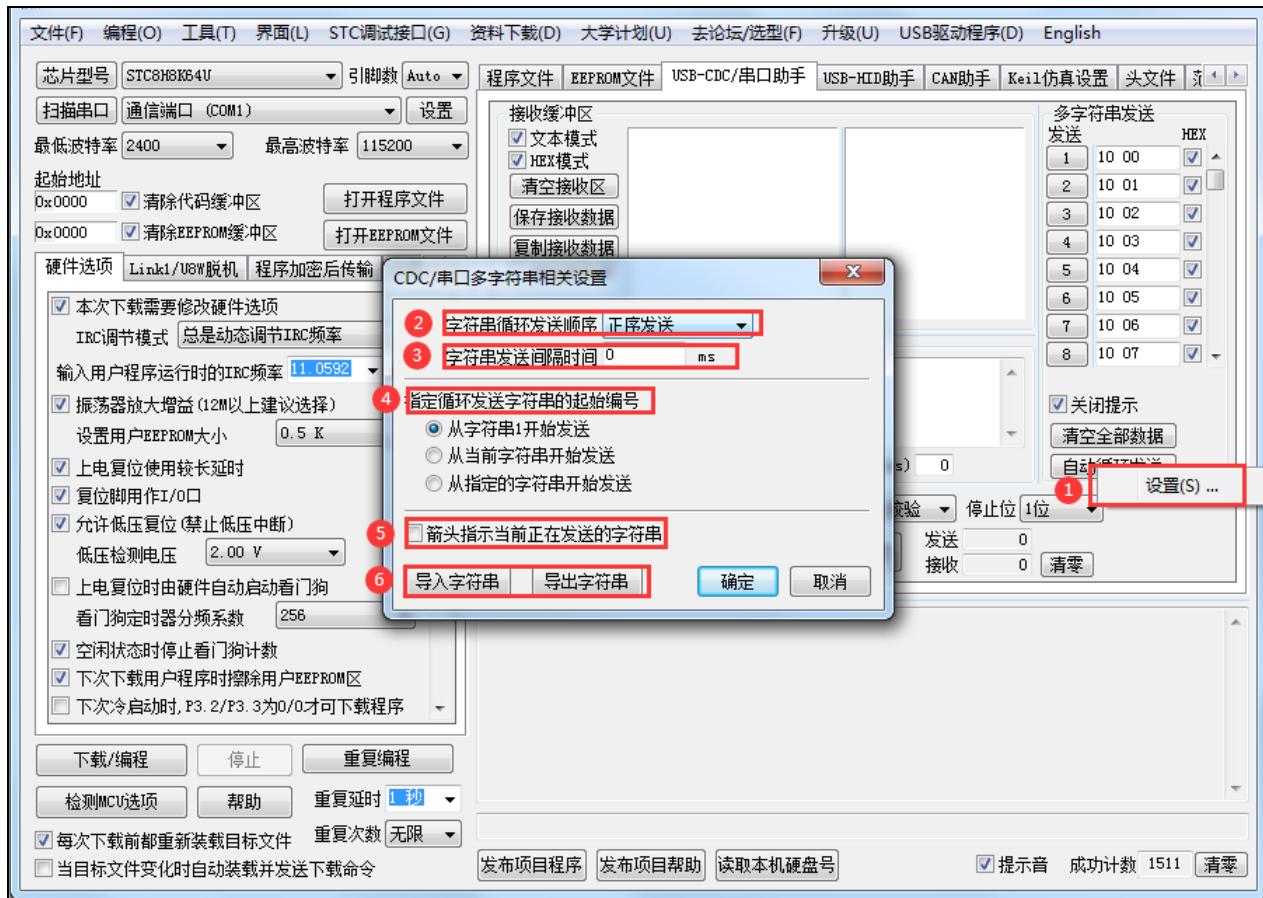
18.9 AiCube-ISP | 串口助手/USB-CDC

串口助手主界面



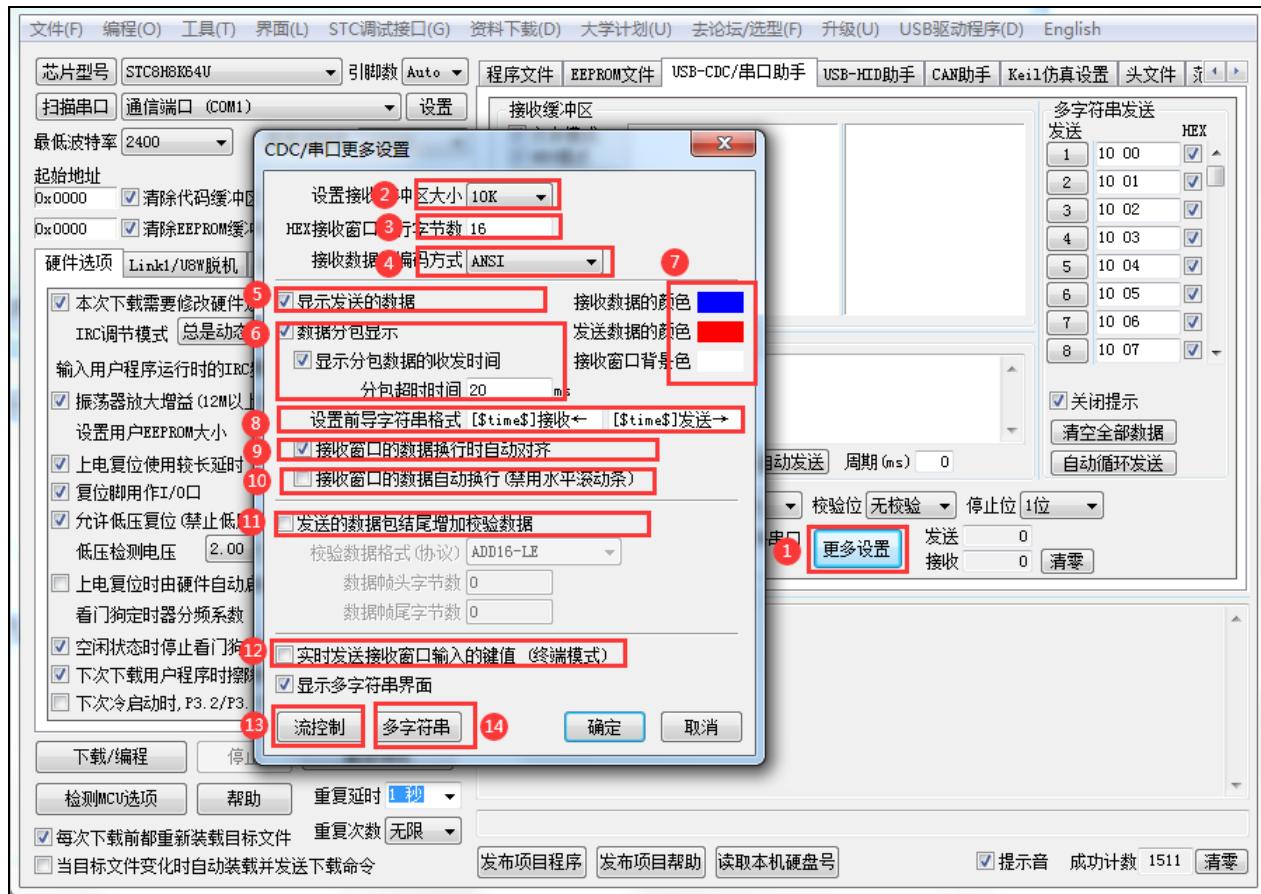
- ①: 在下载软件中选择“USB-CDC/串口助手”功能页，进入串口助手界面
- ②: 选择接收数据的显示格式
- ③: 保存/复制接收的数据
- ④: 设置接收数据自动存储到文件
- ⑤: 选择发送数据的格式
- ⑥: “多字符串”控制界面
- ⑦: 数据/文件发送按钮
- ⑧: 设置重复下载的周期
- ⑨: 选择串口号，设置串口参数
- ⑩: 设置 ISP 下载完成后是否自动打开串口
- ⑪: 设置发送完成数据后，是否自动发送结束符

多字符串设置界面



- ①: 鼠标右键点击“自动循环发送”按钮，点击右键菜单“设置...”进入多字符串设置界面
- ②: 设置多字符串循环发送顺序
- ③: 设置多字符串循环发送间隔时间
- ④: 设置多字符串循环发送的起始编号
- ⑤: 设置是否需要用箭头指示当前正在发送的字符串
- ⑥: 多字符串导出到文件或从文件导入

串口助手更多设置



①: 点击“更多设置”按钮，进入串口助手更多设置界面

②: 设置接收缓存大小（缓存越大，软件反应越慢）

③: 设置 HEX 接收数据每行显示的数据个数

④: 设置接收数据的汉字编码格式
支持如下汉字编码格式:

ANSI: GB2312 汉字编码

UTF8: UNICODE 互联网常用编码

UTF16-LE: 小端 UTF16 编码

UTF16-BE: 大端 UTF16 编码

⑤: 设置是否显示发送的数据

⑥: 设置数据是否自动分包显示

⑦: 设置界面的显示颜色

⑧: 设置接收窗口数据显示的前导字符

⑨: 设置接收数据的换行显示模式

⑩: 设置接收窗口是否自动换行

⑪: 设置发送数据是否自动追加校验数据
支持如下校验格式:

ADD8: 字节校验和

ADD8N: 字节校验和补码

ADD16-LE: 小端双字节校验和

ADD16-BE: 大端双字节校验和

XOR8: 字节异或

CRC16-MODBUS: MODBUS 协议的 CRC16

CRC16-USB: USB 协议的 CRC16

CRC16-XMODEM: XMODEM 协议的 CRC16

CRC32: 32 位 CRC 校验

⑫: 设置是否使能终端模式（终端模式：将光标定位到接收窗口，按下键盘按键实时发送相应的键码）

⑬: 进入流控制设置界面

⑭: 进入多字符串界面设置界面

18.10 范例程序

18.10.1 串口 1 使用定时器 2 做波特率发生器

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
        buffer[wptr++] = SBUF;
        wptr &= 0x0f;
    }
}

void UartInit()
{
    SCON = 0x50;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x15;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
```

```

{
    UartSend(*p++);
}
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    ES = 1;
    EA = 1;
    UartSendStr("Uart Test !r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            UartSend(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
AUXR	DATA	8EH
T2H	DATA	0D6H
T2L	DATA	0D7H
BUSY	BIT	20H.0
WPTR	DATA	21H
RPTR	DATA	22H
BUFFER	DATA	23H ;16 bytes
P0M1	DATA	093H
P0M0	DATA	094H
P1M1	DATA	091H
P1M0	DATA	092H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H

<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>

<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>
<i>ORG</i>	<i>0023H</i>
<i>LJMP</i>	<i>UART_ISR</i>
<i>ORG</i>	<i>0100H</i>

UART_ISR:

<i>PUSH</i>	<i>ACC</i>
<i>PUSH</i>	<i>PSW</i>
<i>MOV</i>	<i>PSW,#08H</i>

<i>JNB</i>	<i>TI,CHKRI</i>
<i>CLR</i>	<i>TI</i>
<i>CLR</i>	<i>BUSY</i>

CHKRI:

<i>JNB</i>	<i>RI,UARTISR_EXIT</i>
<i>CLR</i>	<i>RI</i>
<i>MOV</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>@R0,SBUF</i>
<i>INC</i>	<i>WPTR</i>

UARTISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART_INIT:

<i>MOV</i>	<i>SCON,#50H</i>
<i>MOV</i>	<i>T2L,#0E8H</i>
<i>MOV</i>	<i>T2H,#0FFH</i>
<i>MOV</i>	<i>AUXR,#15H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

;65536-11059200/115200/4=0FFE8H

UART_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>SBUF,A</i>
<i>RET</i>	

UART_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SENDEND</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART_SENDSTR</i>

SENDEND:

RET***START:***

<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>LCALL</i>	<i>UART_INIT</i>	
<i>SETB</i>	<i>ES</i>	
<i>SETB</i>	<i>EA</i>	
<i>MOV</i>	<i>DPTR,#STRING</i>	
<i>LCALL</i>	<i>UART_SENDSTR</i>	

LOOP:

<i>MOV</i>	<i>A,RPTR</i>	
<i>XRL</i>	<i>A,WPTR</i>	
<i>ANL</i>	<i>A,#0FH</i>	
<i>JZ</i>	<i>LOOP</i>	
<i>MOV</i>	<i>A,RPTR</i>	
<i>ANL</i>	<i>A,#0FH</i>	
<i>ADD</i>	<i>A,#BUFFER</i>	
<i>MOV</i>	<i>R0,A</i>	
<i>MOV</i>	<i>A,@R0</i>	
<i>LCALL</i>	<i>UART_SEND</i>	
<i>INC</i>	<i>RPTR</i>	
<i>JMP</i>	<i>LOOP</i>	

STRING: DB 'Uart Test !',0DH,0AH,00H***END***

18.10.2 串口 1 使用定时器 1（模式 0）做波特率发生器

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

#define FOSC 11059200UL
#define BRT (65536 - (FOSC / 115200+2) / 4)
//加2 操作是为了让 Keil 编译器
```

//自动实现四舍五入运算

```
bit      busy;
char     wptr;
char     rptr;
char     buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
        buffer[wptr++] = SBUF;
        wptr &= 0x0f;
    }
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x00;
    TLI = BRT;
    TH1 = BRT >> 8;
    TR1 = 1;
    AUXR = 0x40;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
    {
        UartSend(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
```

```

P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

UartInit();
ES = 1;
EA = 1;
UartSendStr("Uart Test !\r\n");

while (1)
{
    if (rptr != wptr)
    {
        UartSend(buffer[rptr++]);
        rptr &= 0x0f;
    }
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		;16 bytes
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0023H</i>
	<i>LJMP</i>	<i>UART_ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>UART_ISR:</i>		
<i>PUSH</i>		<i>ACC</i>
<i>PUSH</i>		<i>PSW</i>

MOV	PSW,#08H
JNB	TI,CHKRI
CLR	TI
CLR	BUSY
CHKRI:	
JNB	RI,UARTISR_EXIT
CLR	RI
MOV	A,WPTR
ANL	A,#0FH
ADD	A,#BUFFER
MOV	R0,A
MOV	@R0,SBUF
INC	WPTR
UARTISR_EXIT:	
POP	PSW
POP	ACC
RETI	
UART_INIT:	
MOV	SCON,#50H
MOV	TMOD,#00H
MOV	TL1,#0E8H
MOV	TH1,#0FFH
SETB	TR1
MOV	AUXR,#40H
CLR	BUSY
MOV	WPTR,#00H
MOV	RPTR,#00H
RET	
UART_SEND:	
JB	BUSY,\$
SETB	BUSY
MOV	SBUF,A
RET	
UART_SENDSTR:	
CLR	A
MOVC	A,@A+DPTR
JZ	SENDEND
LCALL	UART_SEND
INC	DPTR
JMP	UART_SENDSTR
SENDEND:	
RET	
START:	
MOV	SP, #5FH
ORL	P_SW2,#80H
	<i>;使能访问 XFR，没有冲突不用关闭</i>
MOV	P0M0, #00H
MOV	P0M1, #00H
MOV	P1M0, #00H
MOV	P1M1, #00H
MOV	P2M0, #00H
MOV	P2M1, #00H
MOV	P3M0, #00H
MOV	P3M1, #00H

```

MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL    UART_INIT
SETB    ES
SETB    EA

MOV      DPTR,#STRING
LCALL    UART_SENDSTR

LOOP:
MOV      A,RPTR
XRL      A,WPTR
ANL      A,#0FH
JZ       LOOP
MOV      A,RPTR
ANL      A,#0FH
ADD      A,#BUFFER
MOV      R0,A
MOV      A,@R0
LCALL    UART_SEND
INC      RPTR
JMP      LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

```

END

18.10.3 串口 1 使用定时器 1（模式 2）做波特率发生器

C 语言代码

```

//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC     11059200UL
#define BRT      (256 - (FOSC / 115200+16) / 32)
                                         //加16 操作是为了让Keil 编译器
                                         //自动实现四舍五入运算

bit      busy;
char    wptr;
char    rptr;
char    buffer[16];

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
}

```

```
if (RI)
{
    RI = 0;
    buffer[wptr++] = SBUF;
    wptr &= 0x0f;
}
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x20;
    TLI = BRT;
    TH1 = BRT;
    TR1 = 1;
    AUXR = 0x40;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void UartSendStr(char *p)
{
    while (*p)
    {
        UartSend(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    ES = 1;
    EA = I;
    UartSendStr("Uart Test !\r\n");

    while (1)
```

```

{
    if (rptr != wptr)
    {
        UartSend(buffer[rptr++]);
        rptr &= 0x0f;
    }
}

```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		<i>;16 bytes</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0023H</i>
	<i>LJMP</i>	<i>UART_ISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>UART_ISR:</i>		
	<i>PUSH</i>	<i>ACC</i>
	<i>PUSH</i>	<i>PSW</i>
	<i>MOV</i>	<i>PSW,#08H</i>
	<i>JNB</i>	<i>TI,CHKRI</i>
	<i>CLR</i>	<i>TI</i>
	<i>CLR</i>	<i>BUSY</i>
<i>CHKRI:</i>		
	<i>JNB</i>	<i>RI,UARTISR_EXIT</i>
	<i>CLR</i>	<i>RI</i>
	<i>MOV</i>	<i>A,WPTR</i>
	<i>ANL</i>	<i>A,#0FH</i>
	<i>ADD</i>	<i>A,#BUFFER</i>
	<i>MOV</i>	<i>R0,A</i>
	<i>MOV</i>	<i>@R0,SBUF</i>
	<i>INC</i>	<i>WPTR</i>

UARTISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART_INIT:

<i>MOV</i>	<i>SCON,#50H</i>
<i>MOV</i>	<i>TMOD,#20H</i>
<i>MOV</i>	<i>TL1,#0FDH</i>
<i>MOV</i>	<i>TH1,#0FDH</i>
<i>SETB</i>	<i>TR1</i>
<i>MOV</i>	<i>AUXR,#40H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

UART_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>SBUF,A</i>
<i>RET</i>	

UART_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SENDEND</i>
<i>LCALL</i>	<i>UART_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART_SENDSTR</i>

SENDEND:

<i>RET</i>

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	<i>;使能访问 XFR，没有冲突不用关闭</i>
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART_INIT</i>
<i>SETB</i>	<i>ES</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
------------	---------------

<i>XRL</i>	A,WPTR
<i>ANL</i>	A,#0FH
<i>JZ</i>	LOOP
<i>MOV</i>	A,RPTR
<i>ANL</i>	A,#0FH
<i>ADD</i>	A,#BUFFER
<i>MOV</i>	R0,A
<i>MOV</i>	A,@R0
<i>LCALL</i>	UART_SEND
<i>INC</i>	RPTR
<i>JMP</i>	LOOP

STRING: *DB* 'Uart Test !',0DH,0AH,00H

END

18.10.4 串口 2 使用定时器 2 做波特率发生器

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC 11059200UL
#define BRT (65536 - (FOSC / 115200+2) / 4)
                                //加2 操作是为了让Keil 编译器
                                //自动实现四舍五入运算

bit busy;
char wptr;
char rptr;
char buffer[16];

void Uart2Isr() interrupt 8
{
    if (S2CON & 0x02)
    {
        S2CON &= ~0x02;
        busy = 0;
    }
    if (S2CON & 0x01)
    {
        S2CON &= ~0x01;
        buffer[wptr++] = S2BUF;
        wptr &= 0x0f;
    }
}

void Uart2Init()
{
    S2CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
```

```

wptr = 0x00;
rptr = 0x00;
busy = 0;
}

void Uart2Send(char dat)
{
    while (busy);
    busy = 1;
    S2BUF = dat;
}

void Uart2SendStr(char *p)
{
    while (*p)
    {
        Uart2Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    Uart2Init();
    IE2 = 0x01;
    EA = 1;
    Uart2SendStr("Uart Test !\r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            Uart2Send(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
AUXR	DATA	8EH

<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>S2CON</i>	<i>DATA</i>	<i>9AH</i>
<i>S2BUF</i>	<i>DATA</i>	<i>9BH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		<i>;16 bytes</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>
<i>ORG</i>		<i>0043H</i>
<i>LJMP</i>		<i>UART2_ISR</i>
<i>ORG</i>		<i>0100H</i>
<i>UART2_ISR:</i>		
<i>PUSH</i>		<i>ACC</i>
<i>PUSH</i>		<i>PSW</i>
<i>MOV</i>		<i>PSW,#08H</i>
<i>MOV</i>		<i>A,S2CON</i>
<i>JNB</i>		<i>ACC.1,CHKRI</i>
<i>ANL</i>		<i>S2CON,#NOT 02H</i>
<i>CLR</i>		<i>BUSY</i>
<i>CHKRI:</i>		
<i>JNB</i>		<i>ACC.0,UART2ISR_EXIT</i>
<i>ANL</i>		<i>S2CON,#NOT 01H</i>
<i>MOV</i>		<i>A,WPTR</i>
<i>ANL</i>		<i>A,#0FH</i>
<i>ADD</i>		<i>A,#BUFFER</i>
<i>MOV</i>		<i>R0,A</i>
<i>MOV</i>		<i>@R0,S2BUF</i>
<i>INC</i>		<i>WPTR</i>
<i>UART2ISR_EXIT:</i>		
<i>POP</i>		<i>PSW</i>
<i>POP</i>		<i>ACC</i>
<i>RETI</i>		
<i>UART2_INIT:</i>		
<i>MOV</i>		<i>S2CON,#10H</i>
<i>MOV</i>		<i>T2L,#0E8H</i>
<i>MOV</i>		<i>T2H,#0FFH</i>
<i>MOV</i>		<i>AUXR,#14H</i>

<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

UART2_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>S2BUFA</i>
<i>RET</i>	

UART2_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SEND2END</i>
<i>LCALL</i>	<i>UART2_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART2_SENDSTR</i>

SEND2END:

<i>RET</i>

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	;使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART2_INIT</i>
<i>MOV</i>	<i>IE2,#01H</i>
<i>SETB</i>	<i>EA</i>
<i>MOV</i>	<i>DPTR,#STRING</i>
<i>LCALL</i>	<i>UART2_SENDSTR</i>

LOOP:

<i>MOV</i>	<i>A,RPTR</i>
<i>XRL</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>JZ</i>	<i>LOOP</i>
<i>MOV</i>	<i>A,RPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>A,@R0</i>
<i>LCALL</i>	<i>UART2_SEND</i>
<i>INC</i>	<i>RPTR</i>
<i>JMP</i>	<i>LOOP</i>

STRING: DB 'Uart Test !,0DH,0AH,00H

END

18.10.5 串口 3 使用定时器 2 做波特率发生器

C 语言代码

//测试工作频率为11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2操作是为了让Keil编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void Uart3Isr() interrupt 17
{
    if(S3CON & 0x02)
    {
        S3CON &= ~0x02;
        busy = 0;
    }
    if(S3CON & 0x01)
    {
        S3CON &= ~0x01;
        buffer[wptr++] = S3BUF;
        wptr &= 0x0f;
    }
}

void Uart3Init()
{
    S3CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart3Send(char dat)
{
    while (busy);
    busy = 1;
    S3BUF = dat;
}
```

```

void Uart3SendStr(char *p)
{
    while (*p)
    {
        Uart3Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    Uart3Init();
    IE2 = 0x08;
    EA = 1;
    Uart3SendStr("Uart Test !r\n");

    while (1)
    {
        if (rptr != wptr)
        {
            Uart3Send(buffer[rptr++]);
            rptr &= 0x0f;
        }
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
AUXR	DATA	8EH
T2H	DATA	0D6H
T2L	DATA	0D7H
S3CON	DATA	0ACh
S3BUF	DATA	0ADH
IE2	DATA	0AFH
BUSY	BIT	20H.0
WPTR	DATA	21H
RPTR	DATA	22H
BUFFER	DATA	23H
		;16 bytes
P0M1	DATA	093H

P0M0	DATA	094H
P1M1	DATA	091H
P1M0	DATA	092H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH

ORG	0000H
LJMP	START
ORG	008BH
LJMP	UART3_ISR
ORG	0100H

UART3_ISR:

PUSH	ACC
PUSH	PSW
MOV	PSW,#08H
MOV	A,S3CON
JNB	ACC.I,CHKRI
ANL	S3CON,#NOT 02H
CLR	BUSY

CHKRI:

JNB	ACC.0,UART3ISR_EXIT
ANL	S3CON,#NOT 01H
MOV	A,WPTR
ANL	A,#0FH
ADD	A,#BUFFER
MOV	R0,A
MOV	@R0,S3BUF
INC	WPTR

UART3ISR_EXIT:

POP	PSW
POP	ACC
RETI	

UART3_INIT:

MOV	S3CON,#10H
MOV	T2L,#0E8H
MOV	T2H,#0FFH
MOV	AUXR,#14H
CLR	BUSY
MOV	WPTR,#00H
MOV	RPTR,#00H
RET	

UART3_SEND:

JB	BUSY,\$
SETB	BUSY
MOV	S3BUFA
RET	

UART3_SENDSTR:

```


CLR      A
MOVC    A,@A+DPTR
JZ      SEND3END
LCALL   UART3_SEND
INC     DPTR
JMP     UART3_SENDSTR
SEND3END:
RET

START:
MOV     SP, #5FH
ORL     P_SW2,#80H           ;使能访问 XFR，没有冲突不用关闭

MOV     P0M0, #00H
MOV     P0M1, #00H
MOV     P1M0, #00H
MOV     P1M1, #00H
MOV     P2M0, #00H
MOV     P2M1, #00H
MOV     P3M0, #00H
MOV     P3M1, #00H
MOV     P4M0, #00H
MOV     P4M1, #00H
MOV     P5M0, #00H
MOV     P5M1, #00H

LCALL   UART3_INIT
MOV     IE2,#08H
SETB   EA

MOV     DPTR,#STRING
LCALL   UART3_SENDSTR

LOOP:
MOV     A,RPTR
XRL     A,WPTR
ANL     A,#0FH
JZ      LOOP
MOV     A,RPTR
ANL     A,#0FH
ADD     A,#BUFFER
MOV     R0,A
MOV     A,@R0
LCALL   UART3_SEND
INC     RPT
JMP     LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

END


```

18.10.6 串口 3 使用定时器 3 做波特率发生器

C 语言代码

```
//测试工作频率为11.0592MHz
```

```
#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit    busy;
char   wptr;
char   rptr;
char   buffer[16];

void Uart3Isr() interrupt 17
{
    if(S3CON & 0x02)
    {
        S3CON &= ~0x02;
        busy = 0;
    }
    if(S3CON & 0x01)
    {
        S3CON &= ~0x01;
        buffer[wptr++] = S3BUF;
        wptr &= 0x0f;
    }
}

void Uart3Init()
{
    S3CON = 0x50;
    T3L = BRT;
    T3H = BRT >> 8;
    T4T3M = 0x0a;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart3Send(char dat)
{
    while (busy);
    busy = 1;
    S3BUF = dat;
}

void Uart3SendStr(char *p)
{
    while (*p)
    {
        Uart3Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭
```

```

P0M0 = 0x00;
P0M1 = 0x00;
P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

Uart3Init();
IE2 = 0x08;
EA = 1;
Uart3SendStr("Uart Test !\r\n");

while (1)
{
    if (rptr != wptr)
    {
        Uart3Send(buffer[rptr++]);
        rptr &= 0x0f;
    }
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
T4T3M	DATA	0DIH
T4L	DATA	0D3H
T4H	DATA	0D2H
T3L	DATA	0D5H
T3H	DATA	0D4H
T2L	DATA	0D7H
T2H	DATA	0D6H
S3CON	DATA	0ACH
S3BUF	DATA	0ADH
IE2	DATA	0AFH
 BUSY	 BIT	 20H.0
WPTR	DATA	21H
RPTR	DATA	22H
BUFFER	DATA	23H
		;16 bytes
 P0M1	 DATA	 093H
P0M0	DATA	094H
P1M1	DATA	091H
P1M0	DATA	092H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H

<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>008BH</i>
	<i>LJMP</i>	<i>UART3_ISR</i>
	<i>ORG</i>	<i>0100H</i>

UART3_ISR:

<i>PUSH</i>	<i>ACC</i>
<i>PUSH</i>	<i>PSW</i>
<i>MOV</i>	<i>PSW,#08H</i>
<i>MOV</i>	<i>A,S3CON</i>
<i>JNB</i>	<i>ACC.I,CHKRI</i>
<i>ANL</i>	<i>S3CON,#NOT 02H</i>
<i>CLR</i>	<i>BUSY</i>

CHKRI:

<i>JNB</i>	<i>ACC.0,UART3ISR_EXIT</i>
<i>ANL</i>	<i>S3CON,#NOT 01H</i>
<i>MOV</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>@R0,S3BUF</i>
<i>INC</i>	<i>WPTR</i>

UART3ISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART3_INIT:

<i>MOV</i>	<i>S3CON,#50H</i>
<i>MOV</i>	<i>T3L,#0E8H</i>
<i>MOV</i>	<i>T3H,#0FFH</i>
<i>MOV</i>	<i>T4T3M,#0AH</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

UART3_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>S3BUFA</i>
<i>RET</i>	

UART3_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SEND3END</i>
<i>LCALL</i>	<i>UART3_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART3_SENDSTR</i>

SEND3END:

<i>RET</i>	
------------	--

START:

```

MOV      SP, #5FH
ORL      P_SW2,#80H           ;使能访问 XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL   UART3_INIT
MOV      IE2,#08H
SETB    EA

MOV      DPTR,#STRING
LCALL   UART3_SENDSTR

```

LOOP:

```

MOV      A,RPTR
XRL      A,WPTR
ANL      A,#0FH
JZ       LOOP
MOV      A,RPTR
ANL      A,#0FH
ADD      A,#BUFFER
MOV      R0,A
MOV      A,@R0
LCALL   UART3_SEND
INC      RPTR
JMP      LOOP

```

STRING: DB 'Uart Test !',0DH,0AH,00H

END

18.10.7 串口 4 使用定时器 2 做波特率发生器

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

#define FOSC     11059200UL
#define BRT      (65536 - (FOSC / 115200+2) / 4)           //加2 操作是为了让 Keil 编译器
                                                               //自动实现四舍五入运算

```

```
bit      busy;
char     wptr;
char     rptr;
char     buffer[16];

void Uart4Isr() interrupt 18
{
    if (S4CON & 0x02)
    {
        S4CON &= ~0x02;
        busy = 0;
    }
    if (S4CON & 0x01)
    {
        S4CON &= ~0x01;
        buffer[wptr++] = S4BUF;
        wptr &= 0x0f;
    }
}

void Uart4Init()
{
    S4CON = 0x10;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x14;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart4Send(char dat)
{
    while (busy);
    busy = 1;
    S4BUF = dat;
}

void Uart4SendStr(char *p)
{
    while (*p)
    {
        Uart4Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
```

```

P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

Uart4Init();
IE2 = 0x10;
EA = 1;
Uart4SendStr("Uart Test !\r\n");

while (1)
{
    if (rptr != wptr)
    {
        Uart4Send(buffer[rptr++]);
        rptr &= 0x0f;
    }
}
}

```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>S4CON</i>	<i>DATA</i>	<i>84H</i>
<i>S4BUF</i>	<i>DATA</i>	<i>85H</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		<i>;16 bytes</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>
<i>ORG</i>		<i>0093H</i>
<i>LJMP</i>		<i>UART4_ISR</i>
<i>ORG</i>		<i>0100H</i>

UART4_ISR:

<i>PUSH</i>	<i>ACC</i>	
<i>PUSH</i>	<i>PSW</i>	
<i>MOV</i>	<i>PSW,#08H</i>	
<i>MOV</i>	<i>A,S4CON</i>	
<i>JNB</i>	<i>ACC.I,CHKRI</i>	
<i>ANL</i>	<i>S4CON,#NOT 02H</i>	
<i>CLR</i>	<i>BUSY</i>	
 <i>CHKRI:</i>		
<i>JNB</i>	<i>ACC.0,UART4ISR_EXIT</i>	
<i>ANL</i>	<i>S4CON,#NOT 01H</i>	
<i>MOV</i>	<i>A,WPTR</i>	
<i>ANL</i>	<i>A,#0FH</i>	
<i>ADD</i>	<i>A,#BUFFER</i>	
<i>MOV</i>	<i>R0,A</i>	
<i>MOV</i>	<i>@R0,S4BUF</i>	
<i>INC</i>	<i>WPTR</i>	
 <i>UART4ISR_EXIT:</i>		
<i>POP</i>	<i>PSW</i>	
<i>POP</i>	<i>ACC</i>	
<i>RETI</i>		
 <i>UART4_INIT:</i>		
<i>MOV</i>	<i>S4CON,#10H</i>	
<i>MOV</i>	<i>T2L,#0E8H</i>	<i>;65536-11059200/115200/4=0FFE8H</i>
<i>MOV</i>	<i>T2H,#0FFH</i>	
<i>MOV</i>	<i>AUXR,#14H</i>	
<i>CLR</i>	<i>BUSY</i>	
<i>MOV</i>	<i>WPTR,#00H</i>	
<i>MOV</i>	<i>RPTR,#00H</i>	
<i>RET</i>		
 <i>UART4_SEND:</i>		
<i>JB</i>	<i>BUSY,\$</i>	
<i>SETB</i>	<i>BUSY</i>	
<i>MOV</i>	<i>S4BUFA,A</i>	
<i>RET</i>		
 <i>UART4_SENDSTR:</i>		
<i>CLR</i>	<i>A</i>	
<i>MOVC</i>	<i>A,@A+DPTR</i>	
<i>JZ</i>	<i>SEND4END</i>	
<i>LCALL</i>	<i>UART4_SEND</i>	
<i>INC</i>	<i>DPTR</i>	
<i>JMP</i>	<i>UART4_SENDSTR</i>	
 <i>SEND4END:</i>		
<i>RET</i>		
 <i>START:</i>		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	<i>;使能访问 XFR，没有冲突不用关闭</i>
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	

```

MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

LCALL   UART4_INIT
MOV      IE2,#10H
SETB    EA

MOV      DPTR,#STRING
LCALL   UART4_SENDSTR

LOOP:
MOV      A,RPTR
XRL      A,WPTR
ANL      A,#0FH
JZ       LOOP
MOV      A,RPTR
ANL      A,#0FH
ADD      A,#BUFFER
MOV      R0,A
MOV      A,@R0
LCALL   UART4_SEND
INC      RPT
JMP      LOOP

STRING: DB      'Uart Test !',0DH,0AH,00H

END

```

18.10.8 串口 4 使用定时器 4 做波特率发生器

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

bit      busy;
char    wptr;
char    rptr;
char    buffer[16];

void Uart4Isr() interrupt 18
{
    if (S4CON & 0x02)
    {
        S4CON &= ~0x02;
        busy = 0;

```

```
}

if (S4CON & 0x01)
{
    S4CON &= ~0x01;
    buffer[wptr++] = S4BUF;
    wptr &= 0x0f;
}
}

void Uart4Init()
{
    S4CON = 0x50;
    T4L = BRT;
    T4H = BRT >> 8;
    T4T3M = 0xa0;
    wptr = 0x00;
    rptr = 0x00;
    busy = 0;
}

void Uart4Send(char dat)
{
    while (busy);
    busy = 1;
    S4BUF = dat;
}

void Uart4SendStr(char *p)
{
    while (*p)
    {
        Uart4Send(*p++);
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    Uart4Init();
    IE2 = 0x10;
    EA = 1;
    Uart4SendStr("Uart Test !\r\n");

    while (1)
{
```

```

if (rptr != wptr)
{
    Uart4Send(buffer[rptr++]);
    rptr &= 0x0f;
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>T4T3M</i>	<i>DATA</i>	<i>0DIH</i>
<i>T4L</i>	<i>DATA</i>	<i>0D3H</i>
<i>T4H</i>	<i>DATA</i>	<i>0D2H</i>
<i>T3L</i>	<i>DATA</i>	<i>0D5H</i>
<i>T3H</i>	<i>DATA</i>	<i>0D4H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>S4CON</i>	<i>DATA</i>	<i>84H</i>
<i>S4BUF</i>	<i>DATA</i>	<i>85H</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>WPTR</i>	<i>DATA</i>	<i>21H</i>
<i>RPTR</i>	<i>DATA</i>	<i>22H</i>
<i>BUFFER</i>	<i>DATA</i>	<i>23H</i>
		<i>;16 bytes</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>		<i>0000H</i>
<i>LJMP</i>		<i>START</i>
<i>ORG</i>		<i>0093H</i>
<i>LJMP</i>		<i>UART4_ISR</i>
<i>ORG</i>		<i>0100H</i>
<i>UART4_ISR:</i>		
<i>PUSH</i>		<i>ACC</i>
<i>PUSH</i>		<i>PSW</i>
<i>MOV</i>		<i>PSW,#08H</i>
<i>MOV</i>		<i>A,S4CON</i>
<i>JNB</i>		<i>ACC.I,CHKRI</i>
<i>ANL</i>		<i>S4CON,#NOT 02H</i>
<i>CLR</i>		<i>BUSY</i>

CHKRI:

<i>JNB</i>	<i>ACC.0,UART4ISR_EXIT</i>
<i>ANL</i>	<i>S4CON,#NOT 01H</i>
<i>MOV</i>	<i>A,WPTR</i>
<i>ANL</i>	<i>A,#0FH</i>
<i>ADD</i>	<i>A,#BUFFER</i>
<i>MOV</i>	<i>R0,A</i>
<i>MOV</i>	<i>@R0,S4BUF</i>
<i>INC</i>	<i>WPTR</i>

UART4ISR_EXIT:

<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

UART4_INIT:

<i>MOV</i>	<i>S4CON,#50H</i>
<i>MOV</i>	<i>T4L,#0E8H</i>
<i>MOV</i>	<i>T4H,#0FFH</i>
<i>MOV</i>	<i>T4T3M,#0A0H</i>
<i>CLR</i>	<i>BUSY</i>
<i>MOV</i>	<i>WPTR,#00H</i>
<i>MOV</i>	<i>RPTR,#00H</i>
<i>RET</i>	

;65536-11059200/115200/4=0FFE8H

UART4_SEND:

<i>JB</i>	<i>BUSY,\$</i>
<i>SETB</i>	<i>BUSY</i>
<i>MOV</i>	<i>S4BUFA,A</i>
<i>RET</i>	

UART4_SENDSTR:

<i>CLR</i>	<i>A</i>
<i>MOVC</i>	<i>A,@A+DPTR</i>
<i>JZ</i>	<i>SEND4END</i>
<i>LCALL</i>	<i>UART4_SEND</i>
<i>INC</i>	<i>DPTR</i>
<i>JMP</i>	<i>UART4_SENDSTR</i>

SEND4END:

<i>RET</i>	
------------	--

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>ORL</i>	<i>P_SW2,#80H</i>
	<i>;使能访问 XFR，没有冲突不用关闭</i>
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>LCALL</i>	<i>UART4_INIT</i>
<i>MOV</i>	<i>IE2,#10H</i>

SETB **EA**

MOV **DPTR,#STRING**
LCALL **UART4_SENDSTR**

LOOP:

MOV **A,RPTR**
XRL **A,WPTR**
ANL **A,#0FH**
JZ **LOOP**
MOV **A,RPTR**
ANL **A,#0FH**
ADD **A,#BUFFER**
MOV **R0,A**
MOV **A,@R0**
LCALL **UART4_SEND**
INC **RPTR**
JMP **LOOP**

STRING: **DB** **'Uart Test !',0DH,0AH,00H**

END

18.10.9 串口多机通讯（主机模式）

***** 功能说明 *****

本例程基于 STC8H8K64U 核心实验板（开天斧）进行编写测试。

串口多机通信-主机参考程序。

串口1(P1.6,P1.7) 设置为可变波特率9位数据模式。第9位数据作为地址帧(1), 数据帧(0)的标志。

通过 USB-CDC 接口向 MCU 发送数据, MCU 将收到的数据从串口1发送出去, 其中第一个字节数据作为从机地址, 其它作为数据内容。

SM2 置1后, 只能接收地址帧内容, 自动过滤数据帧内容。如果需要接收指定地址从机返回的数据, 需要在收到指定地址帧后将 SM2 置0。

用定时器做波特率发生器, 建议使用 IT 模式(除非低波特率用 12T), 并选择可被波特率整除的时钟频率, 以提高精度。

此外程序演示两种复位进入 USB 下载模式的方法:

1. 通过每1毫秒执行一次“KeyResetScan”函数, 实现长按P3.2口按键触发 MCU 复位, 进入 USB 下载模式。

(如果不希望复位进入 USB 下载模式的话, 可在复位代码里将 IAP_CONTR 的 bit6 清0, 选择复位进用户程序区)

2. 通过加载“stc_usb_cdc_8h.lib”库函数, 实现使用 AiCube-ISP 软件发送指令触发 MCU 复位, 进入 USB 下载模式并自动下载。

3. 如果 data 空间不够, 可将 Memory Model 设为 Large 模式, 然后使用“stc_usb_cdc_8h_xdata.lib”库函数。

下载时, 选择时钟 22.1184MHZ (用户可自行修改频率).

```
#include "../../comm/STC8h.h"                                //包含此头文件后, 不需要再包含"reg51.h"头文件
#include "../../comm/usb.h"                                    //USB 调试及复位所需头文件

#define MAIN_Fosc 22118400L                                     //定义主时钟 (精确计算 115200 波特率)
#define Timer0_Reload (65536UL -(MAIN_Fosc / 1000))          //Timer 0 中断频率, 1000 次/秒

#define Baudrate1           115200L
#define UART1_BUFLLENGTH   64

bit B_Ims;                                                 //Ims 标志
bit B_TX1_Busy;                                           //发送忙标志
u8 TX1_Cnt;                                               //发送计数
u8 RX1_Cnt;                                               //接收计数
u8 RX1_TimeOut;                                           //接收缓冲

u8 xdata RX1_Buffer[UART1_BUFLLENGTH];                   //接收缓冲

//USB 调试及复位所需定义
char *USER_DEVICEDESC = NULL;
char *USER_PRODUCTDESC = NULL;
char *USER_STCISPCMD = "@STCISP#";                        //设置自动复位到 ISP 区的用户接口命令

//P3.2 口按键复位所需变量
bit Key_Flag;
u16 Key_cnt;

void Timer0_config(void);
void UART1_config(u8 brt);                                 //选择波特率,2: 使用 Timer2 做波特率,
                                                               //其它值: 使用 Timer1 做波特率

void UART1_TxByte(u8 dat);
void UART1_Send(u8 addr,u8 *dat,u8 len);
void KeyResetScan(void);

//=====
// 函数: void main(void)
// 描述: 主函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0

```

```

// 备注:
//=====
void main(void)
{
    P_SW2 |= 0x80;                                // 扩展寄存器(XFR)访问使能

    RSTFLAG |= 0x04;                               // 设置硬件复位后需要检测 P3.2 的状态选择运行区域,
                                                    // 否则硬件复位后进入 USB 下载模式

    P0M1 = 0x00;   P0M0 = 0x00;                    // 设置为准双向口
    P1M1 = 0x00;   P1M0 = 0x00;                    // 设置为准双向口
    P2M1 = 0x00;   P2M0 = 0x00;                    // 设置为准双向口
    P3M1 = 0x00;   P3M0 = 0x00;                    // 设置为准双向口
    P4M1 = 0x00;   P4M0 = 0x00;                    // 设置为准双向口
    P5M1 = 0x00;   P5M0 = 0x00;                    // 设置为准双向口
    P6M1 = 0x00;   P6M0 = 0x00;                    // 设置为准双向口
    P7M1 = 0x00;   P7M0 = 0x00;                    // 设置为准双向口

    usb_init();
    Timer0_config();
    UART1_config(1);                             // 选择波特率, 2: 使用 Timer2 做波特率,
                                                // 其它值: 使用 Timer1 做波特率
    IE2 |= 0x80;                                 // IE2 相关的中断位操作使能后, 需要重新设置 EUSB
    EA = I;                                     // 允许总中断

    while (1)
    {
        if(B_Ims)
        {
            B_Ims = 0;
            KeyResetScan();                         // P3.2 口按键触发软件复位, 进入 USB 下载模式,
                                                    // 不需要此功能可删除本行代码

            if(RXI_TimeOut > 0)                   // 超时计数
            {
                if(--RXI_TimeOut == 0)
                {
                    USB_SendData(RXI_Buffer,RXI_Cnt); // 把收到的数据通过 CDC 串口输出
                    RXI_Cnt = 0;                      // 清除字节数
                }
            }
        }

        if(DeviceState != DEVSTATE_CONFIGURED)      // 等待 USB 完成配置
            continue;

        if (bUsbOutReady)
        {
            //UART1_Send(0x53,UsbOutBuffer,OutNumber); // 地址, 数据, 长度
            UART1_Send(UsbOutBuffer[0],&UsbOutBuffer[1],OutNumber-1); // 地址, 数据, 长度

            //USB_SendData(UsbOutBuffer,OutNumber);     // 发送数据缓冲区, 长度

            usb_OUT_done();                          // 接应收应答 (固定格式)
        }
    }
}

void timer0(void) interrupt 1

```

```
{  
    B_Ims = 1;  
}  
  
=====  
// 函数: void Timer0_config(void)  
// 描述: Timer0 初始化函数。  
// 参数: none.  
// 返回: none.  
// 版本: VER1.0  
// 备注:  
=====  
void Timer0_config(void)  
{  
    //Timer0 初始化  
    TMOD &= 0xf0;                                //16 bits timer auto-reload  
    AUXR |= 0x80;                                 //Timer0 set as IT  
    TH0 = (u8)(Timer0_Reload / 256);  
    TL0 = (u8)(Timer0_Reload % 256);  
    ET0 = 1;                                     //Timer0 interrupt enable  
    TR0 = 1;                                     //Tiner0 run  
}  
  
=====  
// 函数: void UART1_TxByte(u8 dat)  
// 描述: 发送一个字节。  
// 参数: 无  
// 返回: 无  
// 版本: V1.0  
=====  
void UART1_TxByte(u8 dat)  
{  
    B_TX1_Busy = 1;  
    SBUF = dat;  
    while(B_TX1_Busy);  
}  
  
=====  
// 函数: void UART1_Send(u8 addr,u8 *dat,u8 len)  
// 描述: 发送一个字节。  
// 参数: 无  
// 返回: 无  
// 版本: V1.0  
=====  
void UART1_Send(u8 addr,u8 *dat,u8 len)  
{  
    u8 i;  
  
    TB8 = 1;                                       //地址帧  
    UART1_TxByte(addr);  
  
    TB8 = 0;                                       //数据帧  
    For (i=0;i<len;i++)  
    {  
        UART1_TxByte(dat[i]);  
    }  
}
```

```

// 函数: SetTimer2Baudrate(u16 dat)
// 描述: 设置 Timer2 做波特率发生器。
// 参数: dat: Timer2 的重装值。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void SetTimer2Baudrate(u16 dat)                                //选择波特率, 2: 使用 Timer2 做波特率,
                                                               //其它值: 使用 Timer1 做波特率
{
    AUXR &= ~(1<<4);                                         //Timer stop
    AUXR &= ~(1<<3);                                         //Timer2 set As Timer
    AUXR |= (1<<2);                                          //Timer2 set as IT mode
    T2H = dat / 256;
    T2L = dat % 256;
    IE2 &= ~(1<<2);                                         //禁止中断
    AUXR |= (1<<4);                                         //Timer run enable
}

//=====
// 函数: void UART1_config(u8 brt)
// 描述: UART1 初始化函数。
// 参数: brt: 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 使用 Timer1 做波特率
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void UART1_config(u8 brt)                                //选择波特率, 2: 使用 Timer2 做波特率,
                                                               //其它值: 使用 Timer1 做波特率
{
    /****** 波特率使用定时器 2 *****/
    if(brt == 2)                                              //SI BRT Use Timer2;
    {
        AUXR |= 0x01;                                         //SI BRT Use Timer1;
        SetTimer2Baudrate(65536UL - (MAIN_Fosc / 4) / Baudrate1);
    }

    /****** 波特率使用定时器 1 *****/
    else
    {
        TRI = 0;                                               //SI BRT Use Timer1;
        AUXR &= ~0x01;                                         //Timer1 set as IT mode
        AUXR |= (1<<6);                                       //Timer1 set As Timer
        TMOD &= ~(1<<6);                                      //Timer1_16bitAutoReload;
        TMOD |= ~0x30;                                         //Timer1_16bitAutoReload;
        TH1 = (u8)((65536UL - (MAIN_Fosc / 4) / Baudrate1) / 256);
        TL1 = (u8)((65536UL - (MAIN_Fosc / 4) / Baudrate1) % 256);
        ET1 = 0;                                                 //禁止中断
        INTCLKO &= ~0x02;                                       //不输出时钟
        TRI = 1;
    }
    /****** *****/
}

SCON = (SCON & 0x3f) / 0xe0;                                //UART1 模式 0x00: 同步移位输出,
                                                               //0x40: 8 位数据, 可变波特率
                                                               //0x80: 9 位数据, 固定波特率,
                                                               //0xc0: 9 位数据, 可变波特率
SM2 = 1;                                                       //允许多机通信
// PS = 1;                                                       //高优先级中断

```

```

ES = 1;                                //允许中断
REN = 1;                                //允许接收
P_SWI &= 0x3f;
P_SWI |= 0x80;                           //UART1 switch to, 0x00: P3.0 P3.1,
                                         //0x40: P3.6 P3.7,
                                         //0x80: P1.6 P1.7,
                                         //0xC0: P4.3 P4.4

RXI_TimeOut = 0;
B_TXI_Busy = 0;
TXI_Cnt = 0;
RXI_Cnt = 0;
}

//=====================================================================
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void UART1_int (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        RXI_Buffer[RXI_Cnt] = SBUF;
        if(++RXI_Cnt >= UART1_BUF_LENGTH)
            RXI_Cnt = 0;                                //防溢出
        RXI_TimeOut = 5;
    }

    if(TI)
    {
        TI = 0;
        B_TXI_Busy = 0;
    }
}

//=====================================================================
// 函数: void delay_ms(u8 ms)
// 描述: 延时函数。
// 参数: ms, 要延时的ms 数, 这里只支持1~255ms. 自动适应主时钟。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void delay_ms(u8 ms)
{
    u16 i;
    do{
        i = MAIN_Fosc / 10000;
        while(--i);                                //10T per loop
    }while(--ms);
}

```

```

// 函数: void KeyResetScan(void)
// 描述: P3.2 口按键长按1秒触发软件复位, 进入 USB 下载模式。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void KeyResetScan(void)
{
    if(!P32)
    {
        if(!Key_Flag)
        {
            Key_cnt++;
            if(Key_cnt >= 1000) //连续 1000ms 有效按键检测
            {
                Key_Flag = 1; //设置按键状态, 防止重复触发

                USBCON = 0x00; //清除 USB 设置
                USBCLK = 0x00;
                IRC48MCR = 0x00;

                delay_ms(10);
                IAP_CONTR = 0x60; //触发软件复位, 从 ISP 开始执行
                while (1);
            }
        }
    }
    else
    {
        Key_cnt = 0;
        Key_Flag = 0;
    }
}

```

18.10.10 串口多机通讯（从机模式）

***** 功能说明 *****

本例程基于 STC8H8K64U 核心实验板（开天斧）进行编写测试。

串口多机通信-从机参考程序。

串口1(P1.6,P1.7)设置为可变波特率9位数据模式。第9位数据作为地址帧(1), 数据帧(0)的标志。

通过SADDR 从机地址寄存器设置本机的从机地址, 其中0xFF 是广播地址。

对SALEN 从机屏蔽地址寄存器进行配置, SALEN 置1 的位所对应的从机地址位与主机发送的地址帧进行对比, 只有匹配的地址帧才能触发串口中断。

例如: SADDR=0x53, SALEN=0xf0, 那么只有高4位是"5"的地址帧才会触发从机的串口接收中断。

从机将串口接收到的内容通过USB-CDC 接口对外发送, 可通过串口助手打开CDC 串口打印接收到的数据。

SM2 置1 后, 只能接收地址帧内容, 自动过滤数据帧内容。需要接收数据时需要将SM2 置0, 收完后再置1。

用定时器做波特率发生器, 建议使用1T 模式(除非低波特率用12T), 并选择可被波特率整除的时钟频率, 以提高精度。

此外程序演示两种复位进入USB 下载模式的方法:

1. 通过每1毫秒执行一次“KeyResetScan”函数, 实现长按P3.2 口按键触发MCU 复位, 进入USB 下载模式。
(如果不希望复位进入USB 下载模式的话, 可在复位代码里将 IAP_CONTR 的bit6 清0, 选择复位进用户程序区)
2. 通过加载“stc_usb_cdc_8h.lib”库函数, 实现使用AiCube-ISP 软件发送指令触发MCU 复位, 进入USB 下载模式并自动下载。
3. 如果data 空间不够, 可将Memory Model 设为Large 模式, 然后使用“stc_usb_cdc_8h_xdata.lib”库函数。

下载时, 选择时钟 22.1184MHZ (用户可自行修改频率).

```

#include "./comm/STC8h.h"                                //包含此头文件后，不需要再包含"reg51.h"头文件
#include "./comm/usb.h"                                  //USB 调试及复位所需头文件

#define MAIN_Fosc 22118400L                            //定义主时钟（精确计算 115200 波特率）
#define Timer0_Reload (65536UL -(MAIN_Fosc / 1000))    //Timer 0 中断频率, 1000 次/秒

#define Baudrate1           115200L                    //1ms 标志
#define UART1_BUFL_LENGTH 64                           //发送忙标志
                                                       //发送计数
                                                       //接收计数
                                                       //接收缓冲

bit B_1ms;                                              //接收缓冲
bit B_TX1_Busy;
u8 TX1_Cnt;
u8 RX1_Cnt;
u8 RX1_TimeOut;

u8 xdata RX1_Buffer[UART1_BUFL_LENGTH];                //接收缓冲

//USB 调试及复位所需定义
char *USER_DEVICEDESC = NULL;
char *USER_PRODUCTDESC = NULL;
char *USER_STCISPCMD = "@STCISP#";                     //设置自动复位到ISP 区的用户接口命令

//P3.2 口按键复位所需变量
bit Key_Flag;
u16 Key_cnt;

void Timer0_config(void);
void UART1_config(u8 brt);                             //选择波特率,2: 使用 Timer2 做波特率,
                                                       //其它值: 使用 Timer1 做波特率
void UART1_TxByte(u8 dat);
void UART1_Send(u8 addr,u8 *dat,u8 len);
void KeyResetScan(void);

//=====================================================================
// 函数: void main(void)
// 描述: 主函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void main(void)
{
    P_SW2 |= 0x80;                                      //扩展寄存器(XFR)访问使能

    RSTFLAG |= 0x04;                                     //设置硬件复位后需要检测 P3.2 的状态选择运行区域,
                                                       //否则硬件复位后进入 USB 下载模式

    P0M1 = 0x00;   P0M0 = 0x00;                          //设置为准双向口
    P1M1 = 0x00;   P1M0 = 0x00;                          //设置为准双向口
    P2M1 = 0x00;   P2M0 = 0x00;                          //设置为准双向口
    P3M1 = 0x00;   P3M0 = 0x00;                          //设置为准双向口
    P4M1 = 0x00;   P4M0 = 0x00;                          //设置为准双向口
    P5M1 = 0x00;   P5M0 = 0x00;                          //设置为准双向口
    P6M1 = 0x00;   P6M0 = 0x00;                          //设置为准双向口
    P7M1 = 0x00;   P7M0 = 0x00;                          //设置为准双向口

    usb_init();
}

```

```

Timer0_config();
UART1_config(1); //选择波特率, 2: 使用 Timer2 做波特率,
//其它值: 使用 Timer1 做波特率
IE2 |= 0x80; //IE2 相关的中断位操作使能后, 需要重新设置 EUSB
EA = 1; //允许总中断

while (1)
{
    if(B_ImS)
    {
        B_ImS = 0;
        KeyResetScan(); //P3.2 口按键触发软件复位, 进入 USB 下载模式,
        //不需要此功能可删除本行代码

        if(RXI_TimeOut > 0) //超时计数
        {
            if(--RXI_TimeOut == 0)
            {
                SM2 = 1; //数据接收完毕, 重新使能地址匹配
                USB_SendData(RXI_Buffer,RXI_Cnt); //把收到的数据通过 CDC 串口输出
                RXI_Cnt = 0; //清除字节数
            }
        }
    }

    if(DeviceState != DEVSTATE_CONFIGURED) //等待 USB 完成配置
        continue;

    if (bUsbOutReady)
    {
        //USB_SendData(UsbOutBuffer,OutNumber); //发送数据缓冲区, 长度
        UART1_Send(UsbOutBuffer[0],&UsbOutBuffer[1],OutNumber-1); //地址, 数据, 长度

        usb_OUT_done(); //接收应答 (固定格式)
    }
}

void timer0(void) interrupt 1
{
    B_ImS = 1;
}

//=====================================================================
// 函数: void Timer0_config(void)
// 描述: Timer0 初始化函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void Timer0_config(void)
{
    //Timer0 初始化
    TMOD &= 0xf0; //16 bits timer auto-reload
    AUXR |= 0x80; //Timer0 set as IT
    TH0 = (u8)(Timer0_Reload / 256);
    TL0 = (u8)(Timer0_Reload % 256);
    ET0 = 1; //Timer0 interrupt enable
}

```

```

    TR0 = 1;                                //Tiner0 run
}

//=====================================================================
// 函数: void UART1_TxByte(u8 dat)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0
//=====================================================================
void UART1_TxByte(u8 dat)
{
    B_TX1_Busy = 1;
    SBUF = dat;
    while(B_TX1_Busy);
}

//=====================================================================
// 函数: void UART1_Send(u8 addr,u8 *dat,u8 len)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0
//=====================================================================
void UART1_Send(u8 addr,u8 *dat,u8 len)
{
    u8 i;

    TB8 = 1;                                //地址帧
    UART1_TxByte(addr);

    TB8 = 0;                                //数据帧
    For (i=0;i<len;i++)
    {
        UART1_TxByte(dat[i]);
    }
}

//=====================================================================
// 函数: SetTimer2Baudrate(u16 dat)
// 描述: 设置 Timer2 做波特率发生器.
// 参数: dat: Timer2 的重装值.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void SetTimer2Baudrate(u16 dat)           //选择波特率, 2: 使用 Timer2 做波特率,
                                         //其它值: 使用 Timer1 做波特率.
{
    AUXR &= ~(1<<4);                    //Timer stop
    AUXR &= ~(1<<3);                    //Timer2 set As Timer
    AUXR |= (1<<2);                     //Timer2 set as IT mode
    T2H = dat / 256;
    T2L = dat % 256;
    IE2 &= ~(1<<2);                   //禁止中断
    AUXR |= (1<<4);                     //Timer run enable
}

```

```

// 函数: void UART1_config(u8 brt)
// 描述: UART1 初始化函数。
// 参数: brt: 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 使用 Timer1 做波特率
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void UART1_config(u8 brt)                                //选择波特率, 2: 使用 Timer2 做波特率,
                                                        //其它值: 使用 Timer1 做波特率
{
    /***** 波特率使用定时器 2 *****/
    if(brt == 2)
    {
        AUXR |= 0x01;                                     //SI BRT Use Timer2;
        SetTimer2Baudrate(65536UL - (MAIN_Fosc / 4) / Baudrate1);
    }

    /***** 波特率使用定时器 1 *****/
    else
    {
        TRI = 0;                                         //SI BRT Use Timer1;
        AUXR &= ~0x01;                                    //Timer1 set as IT mode
        AUXR |= (1<<6);                                 //Timer1 set As Timer
        TMOD &= ~(1<<6);                               //Timer1_16bitAutoReload;
        TMOD &= ~0x30;

        TH1 = (u8)((65536UL - (MAIN_Fosc / 4) / Baudrate1) / 256);
        TL1 = (u8)((65536UL - (MAIN_Fosc / 4) / Baudrate1) % 256);
        ET1 = 0;                                         //禁止中断
        INTCLKO &= ~0x02;                                //不输出时钟
        TRI = 1;
    }
    *****/
}

SCON = (SCON & 0x3f) | 0xc0;                           //UART1 模式 0x00: 同步移位输出,
                                                        //0x40: 8 位数据, 可变波特率
                                                        //0x80: 9 位数据, 固定波特率
                                                        //0xc0: 9 位数据, 可变波特率
SM2 = 1;                                              //允许多机通信
// PS = 1;                                              //高优先级中断
ES = 1;                                                //允许中断
REN = 1;                                                //允许接收
SADDR = 0x53;                                         //从机地址
SADEN = 0xf0;                                         //高 4 位匹配
P_SW1 &= 0x3f;                                         //UART1 switch to, 0x00: P3.0 P3.1,
P_SW1 |= 0x80;                                         //0x40: P3.6 P3.7, 0x80: P1.6 P1.7,
                                                        //0xC0: P4.3 P4.4

RXI_TimeOut = 0;
B_TXI_Busy = 0;
TXI_Cnt = 0;
RXI_Cnt = 0;
}

//=====
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: nine.

```

```

// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void UART1_int (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        if(RB8) SM2 = 0;                                //收到匹配地址, SM2 置 0 后接收后续数据

        RX1_Buffer[RX1_Cnt] = SBUF;
        if(++RX1_Cnt >= UART1_BUFL_LENGTH)
            RX1_Cnt = 0;                                //防溢出
        RX1_TimeOut = 5;
    }

    if(TI)
    {
        TI = 0;
        B_TX1_Busy = 0;
    }
}

//=====
// 函数: void delay_ms(u8 ms)
// 描述: 延时函数。
// 参数: ms, 要延时的ms 数, 这里只支持1~255ms. 自动适应主时钟.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void delay_ms(u8 ms)
{
    u16 i;
    do{
        i = MAIN_Fosc / 10000;
        while(--i);                                    //10T per loop
    }while(--ms);
}

//=====
// 函数: void KeyResetScan(void)
// 描述: P3.2 口按键长按1秒触发软件复位, 进入 USB 下载模式。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void KeyResetScan(void)
{
    if(!P32)
    {
        if(!Key_Flag)
        {
            Key_cnt++;
            if(Key_cnt >= 1000)                         //连续 1000ms 有效按键检测
            {
                Key_Flag = 1;                            //设置按键状态, 防止重复触发
            }
        }
    }
}

```

```
USBCON = 0x00;           //清除 USB 设置
USBCLK = 0x00;
IRC48MCR = 0x00;

delay_ms(10);
IAP_CONTR = 0x60;        //触发软件复位, 从 ISP 开始执行
while (1);

}

}

else
{
    Key_cnt = 0;
    Key_Flag = 0;
}

}
```

18.10.11 串口中断收发—MODBUS 协议

C 语言代码

//测试工作频率为 11.0592MHz

```
#include <reg52.h>
#define MAIN_Fosc 11059200L //定义主时钟
/******************* 功能说明 *****/

```

请先别修改程序，直接下载“08-串口1 中断收发-C 语言-MODBUS 协议”里的“UART1.hex”测试，主频选择 11.0592MHz。测试正常后再修改移植。

串口1 按 MODBUS-RTU 协议通信。本例为从机程序，主机一般是电脑端。

本例程只支持多寄存器读和多寄存器写，寄存器长度为 64 个，别的命令用户可以根据需要按 MODBUS-RTU 协议自行添加。

本例子数据使用大端模式(与 C51 一致), CRC16 使用小端模式(与 PC 一致)。

默认参数:

串口1 设置均为 1 位起始位, 8 位数据位, 1 位停止位, 无校验。
串口1(P3.0 P3.1): 9600bps.

定时器0 用于超时计时。串口每收到一个字节都会重置超时计数，当串口空闲超过 35bit 时间时(9600bps 对应 3.6ms)则接收完成。

用户修改波特率时注意要修改这个超时时间。

本例程只是一个应用例子，科普 MODBUS-RTU 协议并不在本例子职责范围，用户可以上网搜索相关协议文本参考。
本例定义了 64 个寄存器，访问地址为 0x1000~0x103f。

命令例子:

写入 4 个寄存器(8 个字节):

10 10 1000 0004 08 1234 5678 90AB CDEF 4930

返回:

10 10 10 00 00 04 4B C6

读出 4 个寄存器:

10 03 1000 0004 4388

返回:

10 03 08 12 34 56 78 90 AB CD EF 3D D5

命令错误返回信息(自定义):

0x90: 功能码错误。收到了不支持的功能码。

0x91: 命令长度错误。

0x92: 写入或读出寄存器个数或字节数错误。

0x93: 寄存器地址错误。

注意：收到广播地址 0x00 时要处理信息，但不返回应答。

```
typedef unsigned char u8;
typedef unsigned int u16;
typedef unsigned long u32;
```

```
sfr P1M1 = 0x91;
```

```
sfr P1M0 = 0x92;
sfr P3M1 = 0xB1;
sfr P3M0 = 0xB2;
sfr P4M1 = 0xB3;
sfr P4M0 = 0xB4;
sfr P_SW1 = 0xA2;
sfr AUXR = 0x8E;
sfr IE2 = 0xAF;
```

```
*****本地常量声明*****
#define RX1_Length 128 /* 接收缓冲长度 */
#define TX1_Length 128 /* 发送缓冲长度 */
```

```
*****本地变量声明*****
u8 xdataRX1_Buffer[RX1_Length]; //接收缓冲
u8 xdataTX1_Buffer[TX1_Length]; //发送缓冲

u8 RX1_cnt; //接收字节计数
u8 TX1_cnt; //发送字节计数
u8 TX1_number; //要发送的字节数
u8 RX1_TimeOut; //接收超时计时器

bit B_RX1_OK; // 接收数据标志
bit B_TX1_Busy; // 发送忙标志
```

```
*****本地函数声明*****
```

```
void UART1_config(u32 brt, u8 timer, u8 io); // brt: 通信波特率, timer=2: 波特率使用定时器 2, 其它值: 使用
Timer1 做波特率. io=0: 串口1 切换到P3.0 P3.1, =1: 切换到P3.6 P3.7, =2: 切换到P1.6 P1.7, =3: 切换到 P4.3
P4.4.

u8 Timer0_Config(u8 t, u32 reload); //t=0: reload 值是主时钟周期数, t=1: reload 值是时间(单位us), 返回0 正
确, 返回1 装载值过大错误

u16 MODBUS_CRC16(u8 *p, u8 n);
u8 MODBUS_RTU(void);
```

```
#define SL_ADDR 0x10 /* 本从机站号地址 */
#define REG_ADDRESS 0x1000 /* 寄存器首地址 */
#define REG_LENGTH 64 /* 寄存器长度 */
u16 xdata modbus_reg[REG_LENGTH]; /* 寄存器地址 */
```

```
=====

// 函数: void main(void)
// 描述: 主函数
```

```

// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void main(void)
{
    u8 i;
    u16 crc;

    Timer0_Config(0, MAIN_Fosc / 10000); //t=0: reload 值是主时钟周期数, (中断频率, 20000 次/秒)
    UART1_config(9600UL, 1, 0); //brt:通信波特率      timer=2: 波特率使用定时器 2, 其它值: 使用 Timer1 做波特率. io=0: 串口1 切换到P3.0 P3.1, =1: 切换到P3.6 P3.7, =2: 切换到P1.6 P1.7, =3: 切换到P4.3      P4.4.

    EA = I;

    while (1)
    {
        if(B_RXI_OK && !B_TXI_Busy) //收到数据,      进行MODBUS-RTU 协议解析
        {
            if(MODBUS_CRC16(RXI_Buffer, RXI_cnt) == 0) //首先判断 CRC16 是否正确, 不正确则忽略,
            不处理也不返回信息
            {
                if(RXI_Buffer[0] == 0x00) // (RXI_Buffer[0] == SL_ADDR) //然后判断站号地址是否正确, 或
                者是否广播地址(不返回信息)
                {
                    if(RXI_cnt > 2) RXI_cnt -= 2; //去掉CRC16 校验字节
                    i = MODBUS_RTU(); //MODBUS-RTU 协议解析
                    if(i != 0) //错误处理
                    {
                        TXI_Buffer[0]=SL_ADDR; //站号地址
                        TXI_Buffer[1]=i; //错误代码
                        crc = MODBUS_CRC16(TXI_Buffer, 2);
                        TXI_Buffer[2]=(u8)(crc>>8); //CRC 是小端模式
                        TXI_Buffer[3]=(u8)crc;
                        B_TXI_Busy = 1; //标志发送忙
                        TXI_cnt = 0; //发送字节计数
                        TXI_number = 4; //要发送的字节数
                        TI = 1; //启动发送
                    }
                }
            }
            RXI_cnt = 0;
            B_RXI_OK = 0;
        }
    }
}

```

}

```

***** MODBUS_CRC (shift) ***** past test 06-11-27 *****
计算CRC，调用方式MODBUS_CRC16(&CRC,8); &CRC 为首地址, 8 为字节数
CRC-16 for MODBUS
CRC16=X16+X15+X2+1
TEST: ---> ABCDEFGHIJ CRC16=0x0BEE 1627T
*/
//=====
// 函数: u16 MODBUS_CRC16(u8 *p, u8 n)
// 描述: 计算CRC16 函数.
// 参数: *p: 要计算的数据指针.
//        n: 要计算的字节数.
// 返回: CRC16 值.
// 版本: V1.0
//=====

u16 MODBUS_CRC16(u8 *p, u8 n)
{
    u8 i;
    u16 crc16;

    crc16 = 0xffff; //预置16 位CRC 寄存器为0xffff (即全为1)
    do
    {
        crc16 ^= (u16)*p; //把8 位数据与16 位CRC 寄存器的低位相异或, 把结果放于CRC 寄存器
        for(i=0; i<8; i++) //8 位数据
        {
            if(crc16 & 1) crc16 = (crc16 >> 1) ^ 0xA001; //如果最低位为0, 把CRC 寄存器的内容右移一位
            //朝低位), 用0 填补最高位,
            //再异或多项式 0xA001
            else crc16 >>= 1; //如果最低位为0, 把CRC 寄存器的内容右移一位
            //朝低位), 用0 填补最高位
        }
        p++;
    }while(--n != 0);
    return (crc16);
}

***** modbus 协议 *****
*****写多寄存器*****
数据: 地址 功能码 寄存地址 寄存器个数 写入字节数 写入数据 CRC16
偏移: 0 1 2 3 4 5 6 7~ 最后2 字节
字节: 1 byte 1 byte 2 byte 2 byte Ibyte 2*n byte 2 byte
      addr 0x10 xxxx xxxx xx xx....xx xxxx

```

返回

数据:	地址	功能码	寄存地址	寄存器个数	CRC16
偏移:	0	1	2 3	4 5	6 7
字节:	1 byte	1 byte	2 byte	2 byte	2 byte
	addr		xxxx	xxxx	xxxx

读多寄存器

数据: 站号(地址)	功能码	寄存地址	寄存器个数	CRC16
偏移:	0 1	2 3	4 5	6 7
字节:	1 byte	1 byte	2 byte	2 byte
	addr	0x03	xxxx	xxxx

返回

数据: 站号(地址)	功能码	读出字节数	读出数据	CRC16
偏移:	0 1	2	3~	最后2字节
字节:	1 byte	1 byte	1byte	2*n byte
	addr	0x03	xx	xx....xx xxxx

返回错误代码

数据: 站号(地址)	错误码	CRC16
偏移:	0 1	最后2字节
字节:	1 byte	1 byte
	addr	0x03

u8 MODBUS_RTU(void)

{

```
u8 i,j,k;
u16 reg_addr; //寄存器地址
u8 reg_len; //写入寄存器个数
u16 crc;
```

```
if(RX1_Buffer[1] == 0x10)//写多寄存器
{
    if(RX1_cnt < 9)      return 0x91;          //命令长度错误
    if((RX1_Buffer[4] != 0) || ((RX1_Buffer[5]*2) != RX1_Buffer[6])) return 0x92; //写入寄存器个数与字
    节数错误
    if((RX1_Buffer[5]==0) || (RX1_Buffer[5] > REG_LENGTH)) return 0x92; //写入寄存器个数错误

    reg_addr = ((u16)RX1_Buffer[2]<<8) + RX1_Buffer[3]; //寄存器地址
    reg_len = RX1_Buffer[5]; //写入寄存器个数
    if((reg_addr+(u16)RX1_Buffer[5]) > (REG_ADDRESS+REG_LENGTH)) return 0x93; //寄存器地址错误
    if(reg_addr< REG_ADDRESS)      return 0x93; //寄存器地址错误
    if((reg_len*2+7) != RX1_cnt) return 0x91; //命令长度错误
```

```

j = reg_addr - REG_ADDRESS; //寄存器数据下标
for(k=7, i=0; i<reg_len; i++,j++)
{
    modbus_reg[j] = ((u16)RXI_Buffer[k] << 8) + RXI_Buffer[k+1]; //写入数据, 大端模式
    k += 2;
}

if(RXI_Buffer[0] != 0) //非广播地址则应答
{
    for(i=0; i<6; i++) TXI_Buffer[i] = RXI_Buffer[i]; //要返回的应答
    crc = MODBUS_CRC16(TXI_Buffer, 6);
    TXI_Buffer[6] = (u8)(crc>>8); //CRC 是小端模式
    TXI_Buffer[7] = (u8)crc;
    B_TXI_Busy = 1; //标志发送忙
    TXI_cnt = 0; //发送字节计数
    TXI_number = 8; //要发送的字节数
    TI = 1; //启动发送
}
}

else if(RXI_Buffer[1] == 0x03) //读多寄存器
{
    if(RXI_Buffer[0] != 0) //非广播地址则应答
    {
        if(RXI_cnt != 6) return 0x91; //命令长度错误
        if(RXI_Buffer[4] != 0) return 0x92; //读出寄存器个数错误
        if((RXI_Buffer[5]==0) || (RXI_Buffer[5] > REG_LENGTH)) return 0x92; //读出寄存器个数错误

        reg_addr = ((u16)RXI_Buffer[2]<<8) + RXI_Buffer[3]; //寄存器地址
        reg_len = RXI_Buffer[5]; //读出寄存器个数
        if((reg_addr+(u16)RXI_Buffer[5]) > (REG_ADDRESS+REG_LENGTH)) return 0x93; //寄存器地址错误
    }

    if(reg_addr < REG_ADDRESS) return 0x93; //寄存器地址错误

    j = reg_addr - REG_ADDRESS; //寄存器数据下标
    TXI_Buffer[0] = SL_ADDR; //站号地址
    TXI_Buffer[1] = 0x03; //读功能码
    TXI_Buffer[2] = reg_len*2; //返回字节数

    for(k=3, i=0; i<reg_len; i++,j++)
    {
        TXI_Buffer[k++] = (u8)(modbus_reg[j] >> 8); //数据为大端模式
        TXI_Buffer[k++] = (u8)modbus_reg[j];
    }

    crc = MODBUS_CRC16(TXI_Buffer, k);
    TXI_Buffer[k++] = (u8)(crc>>8); //CRC 是小端模式
    TXI_Buffer[k++] = (u8)crc;
}

```

```

        B_TX1_Busy = 1;          //标志发送忙
        TX1_cnt      = 0;          //发送字节计数
        TX1_number = k;          //要发送的字节数
        TI = 1;                  //启动发送
    }
}

else return 0x90; //功能码错误

return 0; //解析正确
}

//=====
// 函数:u8 Timer0_Config(u8 t, u32 reload)
// 描述: timer0 初始化函数.
// 参数:   t: 重装值类型, 0 表示重装的是系统时钟数, 其余值表示重装的是时间(us).
//         reload: 重装值.
// 返回: 0: 初始化正确, 1: 重装值过大, 初始化错误.
// 版本: V1.0
//=====

u8 Timer0_Config(u8 t, u32 reload) //t=0: reload 值是主时钟周期数, t=1: reload 值是时间(单位us)
{
    TR0 = 0; //停止计数

    if(t != 0) reload = (u32)((float)MAIN_Fosc * (float)reload)/1000000UL; //重装的是时间(us), 计算所需要的系统时钟数.

    if(reload >= (65536UL * 12)) return 1; //值过大, 返回错误
    if(reload < 65536UL) AUXR |= 0x80; //IT mode
    else
    {
        AUXR &= ~0x80; //12T mode
        reload = reload / 12;
    }
    reload = 65536UL - reload;
    TH0 = (u8)(reload >> 8);
    TL0 = (u8)(reload);

    ET0 = 1; //允许中断
    TMOD &= 0xf0;
    TMOD |= 0; //工作模式, 0: 16 位自动重装, 1: 16 位定时/计数, 2: 8 位自动重装, 3: 16 位自动重装, 不可屏蔽中断
    TR0 = 1; //开始运行
    return 0;
}
//=====

```

```

// 函数: void timer0_ISR (void) interrupt TIMER0_VECTOR
// 描述: timer0 中断函数。
// 参数: none.
// 返回: none.
// 版本: V1.0
//=====
void timer0_ISR (void) interrupt 1
{
    if(RXI_TimeOut != 0)
    {
        if(--RXI_TimeOut == 0) //超时
        {
            if(RXI_cnt != 0)//接收有数据
            {
                B_RXI_OK = 1; //标志已收到数据块
            }
        }
    }
}

//=====
// 函数: SetTimer2Baudrate(u16 dat)
// 描述: 设置 Timer2 做波特率发生器。
// 参数: dat: Timer2 的重装值。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void SetTimer2Baudrate(u16 dat) // 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 使用 Timer1 做波特率
{
    AUXR &= ~(1<<4); //Timer stop
    AUXR &= ~(1<<3); //Timer2 set As Timer
    AUXR |= (1<<2); //Timer2 set as IT mode
    T2H = (u8)(dat >> 8);
    T2L = (u8)dat;
    IE2 &= ~(1<<2); //禁止中断
    AUXR |= (1<<4); //Timer run enable
}

//=====
// 函数: void UART1_config(u32 brt, u8 timer, u8 io)
// 描述: UART1 初始化函数。
// 参数: brt: 通信波特率
//         timer: 波特率使用的定时器, timer=2: 波特率使用定时器2, 其它值: 使用 Timer1 做波特率
//         io: 串口1 切换到的IO, io=0: 串口1 切换到 P3.0 P3.1, =1: 切换到 P3.6 P3.7, =2: 切换到
//              P1.6 P1.7, =3: 切换到 P4.3 P4.4.

```

```
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void UART1_config(u32 brt, u8 timer, u8 io)      // brt: 通信波特率,      timer=2: 波特率使用定时器 2, 其它值: 使用
                                                // Timer1 做波特率. io=0: 串口1 切换到 P3.0 P3.1, =1: 切换到 P3.6 P3.7, =2: 切换到 P1.6 P1.7, =3: 切换到 P4.3
                                                // P4.4.
{
    brt = 65536UL - (MAIN_Fosc / 4) / brt;
    if(timer == 2) // 波特率使用定时器 2
    {
        AUXR |= 0x01; // S1 BRT Use Timer2;
        SetTimer2Baudrate((u16)brt);
    }
    else // 波特率使用定时器 1
    {
        TRI = 0;
        AUXR &= ~0x01; // S1 BRT Use Timer1;
        AUXR |= (1<<6); // Timer1 set as IT mode
        TMOD &= ~(1<<6); // Timer1 set As Timer
        TMOD |= ~0x30; // Timer1_16bitAutoReload;
        TH1 = (u8)(brt >> 8);
        TL1 = (u8)brt;
        ET1 = 0; // 禁止 Timer1 中断
        TRI = 1; // 运行 Timer1
    }
    P_SW1 &= ~0xc0; // 默认切换到 P3.0 P3.1
    if(io == 1)
    {
        P_SW1 |= 0x40; // 切换到 P3.6 P3.7
        P3M1 &= ~0xc0;
        P3M0 &= ~0xc0;
    }
    else if(io == 2)
    {
        P_SW1 |= 0x80; // 切换到 P1.6 P1.7
        P1M1 &= ~0xc0;
        P1M0 &= ~0xc0;
    }
    else if(io == 3)
    {
        P_SW1 |= 0xc0; // 切换到 P4.3 P4.4
        P4M1 &= ~0x18;
        P4M0 &= ~0x18;
    }
}
```

```
else
{
    P3M1 &= ~0x03;
    P3M0 &= ~0x03;
}

SCON = (SCON & 0x3f) | (1<<6);      // 8 位数据, 1 位起始位, 1 位停止位, 无校验
// PS = 1; // 高优先级中断
// ES = 1; // 允许中断
// REN = 1; // 允许接收
}

//=====================================================================
// 函数: void UART1_ISR (void) interrupt UART1_VECTOR
// 描述: 串口1 中断函数
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void UART1_ISR (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        if(!B_RX1_OK) // 接收缓冲空闲
        {
            if(RX1_cnt >= RX1_Length)    RX1_cnt = 0;
            RX1_Buffer[RX1_cnt++] = SBUF;
            RX1_TimeOut = 36; // 接收超时计时器, 35 个位时间
        }
    }

    if(TI)
    {
        TI = 0;
        if(TX1_number != 0) // 有数据要发
        {
            SBUF = TX1_Buffer[TX1_cnt++];
            TX1_number--;
        }
        else B_TX1_Busy = 0;
    }
}
```

18.10.12 串口转 LIN 总线

C 语言代码

//测试工作频率为22.1184MHz

```
***** 功能说明 *****
本例程基于STC8H8K64U为主控芯片的实验箱8进行编写测试, STC8G、STC8H 系列芯片可通用参考。
通过UART 接口连接LIN 收发器实现LIN 总线信号收发测试例程。
UART1 通过串口工具连接电脑。
UART2 外接LIN 收发器(TJA1020/I), 连接LIN 总线。
将电脑串口发送的数据转发到LIN 总线; 从LIN 总线接收到的数据转发到电脑串口。
默认传输速率: 9600 波特率, 发送LIN 数据前切换波特率, 发送13 个显性间隔信号。
下载时, 选择时钟 22.1184MHz(用户可自行修改频率)。
*****/
```

```
#include "stc8h.h"
#include "intrins.h"

#define MAIN_Fosc 22118400L

typedef unsigned char u8;
typedef unsigned int u16;
typedef unsigned long u32;

sbit SLP_N = P2^4; //: Sleep

***** 用户定义宏 *****
#define Baudrate1 (65536UL - (MAIN_Fosc / 4) / 9600UL)
#define Baudrate2 (65536UL - (MAIN_Fosc / 4) / 9600UL)

#define Baudrate_Break (65536UL - (MAIN_Fosc / 4) / 6647UL) //发送显性间隔信号波特率

#define UART1_BUF_LENGTH 32
#define UART2_BUF_LENGTH 32

#define LIN_ID 0x31

u8 TX1_Cnt; //发送计数
u8 RX1_Cnt; //接收计数
u8 TX2_Cnt; //发送计数
u8 RX2_Cnt; //接收计数
bit B_TX1_Busy; //发送忙标志
bit B_TX2_Busy; //发送忙标志
u8 RX1_TimeOut;
u8 RX2_TimeOut;

u8 xdata RX1_Buffer[UART1_BUF_LENGTH]; //接收缓冲
u8 xdata RX2_Buffer[UART2_BUF_LENGTH]; //接收缓冲

void UART1_config(u8 brt);
void UART2_config(u8 brt);
void PrintString1(u8 *puts);
void delay_ms(u8 ms);
void UART1_TxByte(u8 dat);
void UART2_TxByte(u8 dat);
void Lin_Send(u8 *puts);
```

```

void SetTimer2Baudrate(u16 dat);

//=====
// 函数: void main(void)
// 描述: 主函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void main(void)
{
    u8 i;

    P0M1 = 0; P0M0 = 0;                                //设置为准双向口
    P1M1 = 0; P1M0 = 0;                                //设置为准双向口
    P2M1 = 0; P2M0 = 0;                                //设置为准双向口
    P3M1 = 0; P3M0 = 0;                                //设置为准双向口
    P4M1 = 0; P4M0 = 0;                                //设置为准双向口
    P5M1 = 0; P5M0 = 0;                                //设置为准双向口
    P6M1 = 0; P6M0 = 0;                                //设置为准双向口
    P7M1 = 0; P7M0 = 0;                                //设置为准双向口

    UART1_config(1);
    UART2_config(2);
    EA = 1;                                         //允许全局中断
    SLP_N = 1;

    PrintStringI("STC8H8K64U UART1 Test Programme!\r\n"); //UART1 发送一个字符串

    while (1)
    {
        delay_ms(1);
        if(RX1_TimeOut > 0)
        {
            if(--RX1_TimeOut == 0)                      //超时,则串口接收结束
            {
                if(RX1_Cnt > 0)                         //将 UART1 收到的数据发送到 LIN 总线上
                {
                    Lin_Send(RX1_Buffer);
                }
                RX1_Cnt = 0;
            }
        }

        if(RX2_TimeOut > 0)
        {
            if(--RX2_TimeOut == 0)                      //超时,则串口接收结束
            {
                if(RX2_Cnt > 0)
                {
                    for (i=0; i < RX2_Cnt; i++)          //遇到停止符 0 结束
                    {
                        UART1_TxByte(RX2_Buffer[i]); //从 LIN 总线收到的数据发送到 UART1
                    }
                }
                RX2_Cnt = 0;
            }
        }
    }
}

```

```

        }

//=====
// 函数: void delay_ms(unsigned char ms)
// 描述: 延时函数。
// 参数: ms,要延时的ms 数, 这里只支持1~255ms. 自动适应主时钟。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void delay_ms(u8 ms)
{
    u16 i;
    do{
        i = MAIN_Fosc / 10000;
        while(--i);                                //10T per loop
    }while(--ms);
}

//=====
// 函数: u8          Lin_CheckPID(u8      id)
// 描述: ID 码加上校验符, 转成 PID 码。
// 参数: ID 码。
// 返回: PID 码。
// 版本: VER1.0
// 备注:
//=====

u8 Lin_CheckPID(u8 id)
{
    u8 returnpid ;
    u8 P0 ;
    u8 PI ;

    P0 = (((id)^((id>>1)^((id>>2)^((id>>4)&0x01)<<6;
    PI = ((~((id>>1)^((id>>3)^((id>>4)^((id>>5))))&0x01)<<7;

    returnpid = id|P0|PI ;

    return returnpid ;
}

//=====
// 函数: u8 LINCalcChecksum(u8 *dat)
// 描述: 计算校验码。
// 参数: 数据场传输的数据。
// 返回: 校验码。
// 版本: VER1.0
// 备注:
//=====

static u8 LINCalcChecksum(u8 *dat)
{
    u16 sum = 0;
    u8 i;

    for(I = 0; i < 8; i++)
    {
        sum += dat[i];
        if(sum & 0xFF00)

```

```

{
    sum = (sum & 0x00FF) + 1;
}
}
sum ^= 0x00FF;
return (u8)sum;
}

//=====================================================================
// 函数: void Lin_SendBreak(void)
// 描述: 发送显性间隔信号。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void Lin_SendBreak(void)
{
    SetTimer2Baudrate(Baudrate_Break);
    UART2_TxByte(0);
    SetTimer2Baudrate(Baudrate2);
}

//=====================================================================
// 函数: void Lin_Send(u8 *puts)
// 描述: 发送LIN总线报文。
// 参数: 待发送的数据场内容.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void Lin_Send(u8 *puts)
{
    u8 i;

    Lin_SendBreak();                                //Break
    UART2_TxByte(0x55);                            //SYNC
    UART2_TxByte(Lin_CheckPID(LIN_ID));           //LIN ID
    for(i=0;i<8;i++)
    {
        UART2_TxByte(puts[i]);
    }
    UART2_TxByte(LINCalcChecksum(puts));
}

//=====================================================================
// 函数: void UART1_TxByte(u8 dat)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0
//=====================================================================
void UART1_TxByte(u8 dat)
{
    SBUF = dat;
    B_TX1_Busy = 1;
    while(B_TX1_Busy);
}

```

```
=====
// 函数: void UART2_TxByte(u8 dat)
// 描述: 发送一个字节.
// 参数: 无.
// 返回: 无.
// 版本: V1.0
=====

void UART2_TxByte(u8 dat)
{
    S2BUF = dat;
    B_TX2_Busy = 1;
    while(B_TX2_Busy);
}

=====
// 函数: void PrintString1(u8 *puts)
// 描述: 串口1发送字符串函数。
// 参数: puts:          字符串指针.
// 返回: none.
// 版本: VER1.0
// 备注:
=====

void PrintString1(u8 *puts)
{
    for ( ; *puts != 0; puts++)           //遇到停止符0 结束
    {
        SBUF = *puts;
        B_TX1_Busy = 1;
        while(B_TX1_Busy);
    }
}

=====
// 函数: void PrintString2(u8 *puts)
// 描述: 串口2发送字符串函数。
// 参数: puts:          字符串指针.
// 返回: none.
// 版本: VER1.0
// 备注:
=====

//void PrintString2(u8 *puts)
//{
//    for ( ; *puts != 0;  puts++)           //遇到停止符0 结束
//    {
//        S2BUF = *puts;
//        B_TX2_Busy = 1;
//        while(B_TX2_Busy);
//    }
//}

=====
// 函数: SetTimer2Baudrate(u16 dat)
// 描述: 设置 Timer2 做波特率发生器。
// 参数: dat: Timer2 的重装值.
// 返回: none.
// 版本: VER1.0
// 备注:
=====

void SetTimer2Baudrate(u16 dat)
```

```

{
    AUXR &= ~(1<<4);                                //Timer stop
    AUXR &= ~(1<<3);                                //Timer2 set As Timer
    AUXR |= (1<<2);                                 //Timer2 set as IT mode
    T2H   = dat / 256;
    T2L = dat % 256;
    IE2 &= ~(1<<2);                                //禁止中断
    AUXR |= (1<<4);                                //Timer run enable
}

//=====
// 函数: void UART1_config(u8 brt)
// 描述: UART1 初始化函数。
// 参数: brt: 选择波特率, 2: 使用 Timer2 做波特率, 其它值: 使用 Timer1 做波特率
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void UART1_config(u8 brt)
{
    /****** 波特率使用定时器 2 *****/
    if(brt == 2)
    {
        AUXR |= 0x01;                                //SI BRT Use Timer2;
        SetTimer2Baudrate(Baudrate1);
    }

    /****** 波特率使用定时器 1 *****/
    else
    {
        TRI = 0;
        AUXR &= ~0x01;                                //SI BRT Use Timer1;
        AUXR |= (1<<6);                            //Timer1 set as IT mode
        TMOD &= ~(1<<6);                           //Timer1 set As Timer
        TMOD &= ~0x30;                               //Timer1_16bitAutoReload;
        TH1 = (u8)(Baudrate1 / 256);
        TL1 = (u8)(Baudrate1 % 256);
        ET1 = 0;                                     //禁止中断
        INTCLKO &= ~0x02;                            //不输出时钟
        TRI = 1;
    }
}

//=====
SCON = (SCON & 0x3f) / 0x40;                      //UART1 模式: 0x00: 同步移位输出,
//                                              //0x40: 8 位数据, 可变波特率,
//                                              //0x80: 9 位数据, 固定波特率,
//                                              //0xc0: 9 位数据, 可变波特率
// PS    = 1;                                       //高优先级中断
// ES    = 1;                                       //允许中断
// REN   = 1;                                       //允许接收
P_SW1 &= 0x3f;                                    //UART1switch to: 0x00: P3.0 P3.1,
//                                              //0x40: P3.6 P3.7,
//                                              //0x80: P1.6 P1.7,
//                                              //0xC0: P4.3 P4.4
// P_SW1 |= 0x80;

B_TX1_Busy = 0;
TX1_Cnt = 0;
RX1_Cnt = 0;

```

```

}

//=====
// 函数: void UART2_config(u8 brt)
// 描述: UART2 初始化函数。
// 参数: brt: 选择波特率, 2: 使用Timer2 做波特率, 其它值: 无效
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void UART2_config(u8 brt)
{
    if(brt == 2)
    {
        SetTimer2Baudrate(Baudrate2);

        S2CON &= ~(1<<7);                                //8 位数据, 1 位起始位, 1 位停止位, 无校验
        IE2 |= 1;                                         //允许中断
        S2CON |= (1<<4);                                //允许接收
        P_SW2 &= ~0x01;
        P_SW2 |= 1;                                       //UART2 switch to: 0: P1.0/P1.1, 1: P4.6/P4.7

        B_TX2_Busy = 0;
        TX2_Cnt = 0;
        RX2_Cnt = 0;
    }
}

//=====
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: nine.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void UART1_int (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        if(RX1_Cnt >= UART1_BUF_LENGTH) RX1_Cnt = 0;
        RX1_Buffer[RX1_Cnt] = SBUF;
        RX1_Cnt++;
        RX1_TimeOut = 5;
    }

    if(TI)
    {
        TI = 0;
        B_TX1_Busy = 0;
    }
}

//=====
// 函数: void UART2_int (void) interrupt UART2_VECTOR
// 描述: UART2 中断函数。
// 参数: nine.
// 返回: none.
//=====
```

```
// 版本: VER1.0
// 备注:
//=====
void UART2_int (void) interrupt 8
{
    if((S2CON & 1) != 0)
    {
        S2CON &= ~1;                                //Clear Rx flag
        if(RX2_Cnt >= UART2_BUF_LENGTH) RX2_Cnt = 0;
        RX2_Buffer[RX2_Cnt] = S2BUF;
        RX2_Cnt++;
        RX2_TimeOut = 5;
    }

    if((S2CON & 2) != 0)
    {
        S2CON &= ~2;                                //Clear Tx flag
        B_TX2_Busy = 0;
    }
}
```

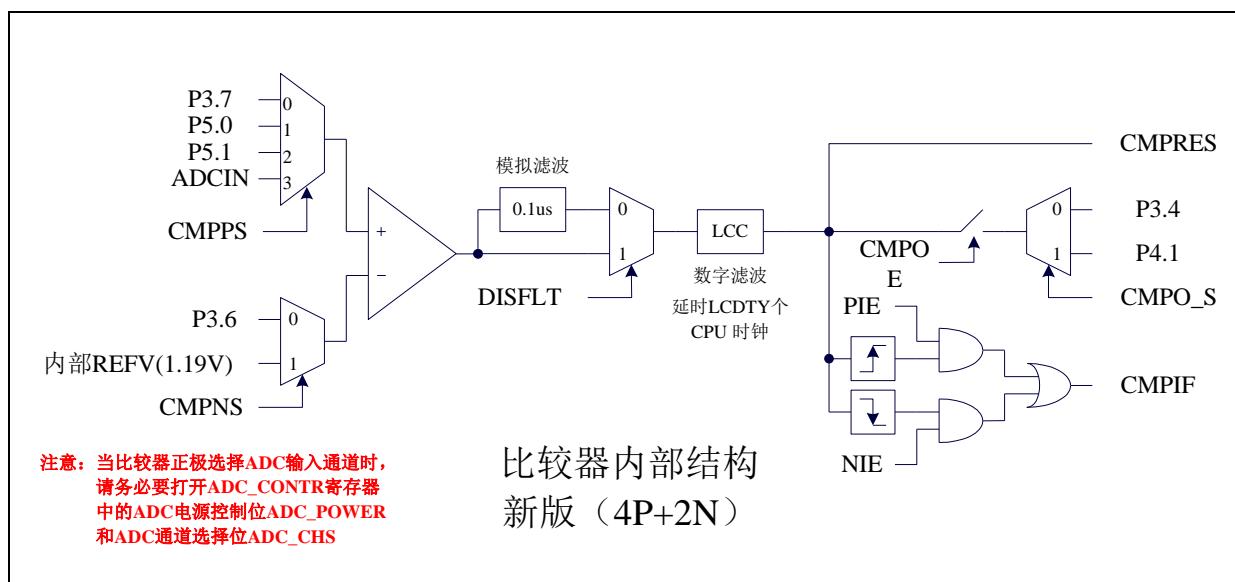
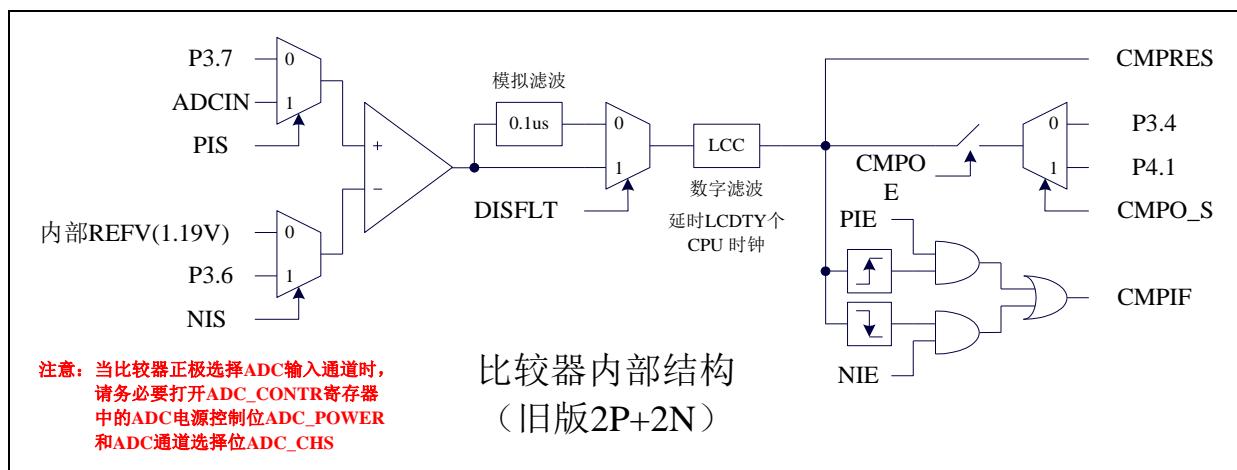
19 比较器，掉电检测，内部 1.19V 参考信号源 (BGV)

产品线	比较器		CMPEXCFG
	旧版 (2P+2N)	新版 (4P+2N)	
STC8H1K08 系列	●		
STC8H1K28 系列	●		
STC8H3K64S4 系列	●		
STC8H3K64S2 系列	●		
STC8H8K64U 系列 A 版本	●		
STC8H8K64U 系列 B/C/D 版本		●	●
STC8H4K64TL 系列		●	●
STC8H4K64LCD 系列		●	●
STC8H1K08T 系列		●	●
STC8H2K12U-A/B 系列		●	●
STC8H2K32U 系列		●	●
STC8G1K08-SOP8 系列			
STC8G1K08A-SOP8 系列			

STC8H 系列单片机内部集成了一个比较器。比较器的正极可以是 P3.7 端口或者 ADC 的模拟输入通道（新版比较器的正极可以是 P3.7 端口、P5.0 端口、P5.1 端口或者 ADC 的模拟输入通道），而负极可以 P3.6 端口或者是内部 BandGap 经过 OP 后的 REFV 电压（内部固定比较电压）。**通过多路选择器和分时复用可实现多个比较器的应用。**

比较器内部有可程序控制的两级滤波：模拟滤波和数字滤波。模拟滤波可以过滤掉比较输入信号中的毛刺信号，数字滤波可以等待输入信号更加稳定后再进行比较。比较结果可直接通过读取内部寄存器位获得，也可将比较器结果正向或反向输出到外部端口。将比较结果输出到外部端口可用作外部事件的触发信号和反馈信号，可扩大比较的应用范围。

19.1 比较器内部结构图



19.2 比较器输出功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW2	BAH	EAXFR	-	I2C_S[1:0]		CMPO_S	S4_S	S3_S	S2_S

CMPO_S: 比较器输出脚选择位

CMPO_S	CMPO
0	P3.4
1	P4.1

19.3 比较器相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
CMPCR1	比较器控制寄存器 1	E6H	CMPEN	CMPIF	PIE	NIE	PIS	NIS	CMPOE	CMPRES	0000,0000
CMPCR2	比较器控制寄存器 2	E7H	INVCMPO	DISFLT	LCDTY[5:0]						0000,0000
CMPEXCFG	比较器扩展配置寄存器	FEAEH	CHYS[1:0]		-	-	-	CMPNS	CMPPS[1:0]		00xx,x000

19.3.1 比较器控制寄存器 1 (CMPCR1)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPCR1	E6H	CMPEN	CMPIF	PIE	NIE	PIS	NIS	CMPOE	CMPRES

CMPEN: 比较器模块使能位

0: 关闭比较功能

1: 使能比较功能

CMPIF: 比较器中断标志位。当 PIE 或 NIE 被使能后, 若产生相应的中断信号, 硬件自动将 CMPIF 置 1, 并向 CPU 提出中断请求。此标志位必须用户软件清零。

(注意: 没有使能比较器中断时, 硬件不会设置此中断标志, 即使用查询方式访问比较器时, 不能查询此中断标志)

PIE: 比较器上升沿中断使能位。

0: 禁止比较器上升沿中断。

1: 使能比较器上升沿中断。使能比较器的比较结果由 0 变成 1 时产生中断请求。

NIE: 比较器下降沿中断使能位。

0: 禁止比较器下降沿中断。

1: 使能比较器下降沿中断。使能比较器的比较结果由 1 变成 0 时产生中断请求。

PIS: 比较器的正极选择位 (适用于旧版比较器, 新版比较器使用 CMPEXCFG 中的 CMPPS 进行选择)

0: 选择外部端口 P3.7 为比较器正极输入源。

1: 通过 ADC_CONTR 中的 ADC_CHS 位选择 ADC 的模拟输入端作为比较器正极输入源。

(注意 1: 当比较器正极选择 ADC 输入通道时, 请务必要打开 ADC_CONTR 寄存器中的 ADC 电源控制位 ADC_POWER 和 ADC 通道选择位 ADC_CHS)

(注意 2: 当需要使用比较器中断唤醒掉电模式/主时钟停振/省电模式时, 比较器正极必须选择 P3.7, 不能使用 ADC 输入通道)

NIS: 比较器的负极选择位 (适用于旧版比较器, 新版比较器使用 CMPEXCFG 中的 CMPNS 进行选择)

0: 选择内部 BandGap 经过 OP 后的电压 REFV 作为比较器负极输入源。 (芯片在出厂时, 内部参考信号源调整为 1.19V)

1: 选择外部端口 P3.6 为比较器负极输入源。

CMPOE: 比较器结果输出控制位

0: 禁止比较器结果输出

1: 使能比较器结果输出。比较器结果输出到 P3.4 或者 P4.1 (由 P_SW2 中的 CMPO_S 进行设定)

CMPRES: 比较器的比较结果。此位为只读。

0: 表示 CMP+的电平低于 CMP-的电平

1: 表示 CMP+的电平高于 CMP-的电平

CMPRES 是经过数字滤波后的输出信号, 而不是比较器的直接输出结果。

19.3.2 比较器控制寄存器 2 (CMPCR2)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPCR2	E7H	INVCMPO	DISFLT						LCDTY[5:0]

INVCMPO: 比较器结果输出控制

0: 比较器结果正向输出。若 CMPRES 为 0, 则 P3.4/P4.1 输出低电平, 反之输出高电平。

1: 比较器结果反向输出。若 CMPRES 为 0, 则 P3.4/P4.1 输出高电平, 反之输出低电平。

DISFLT: 模拟滤波功能控制

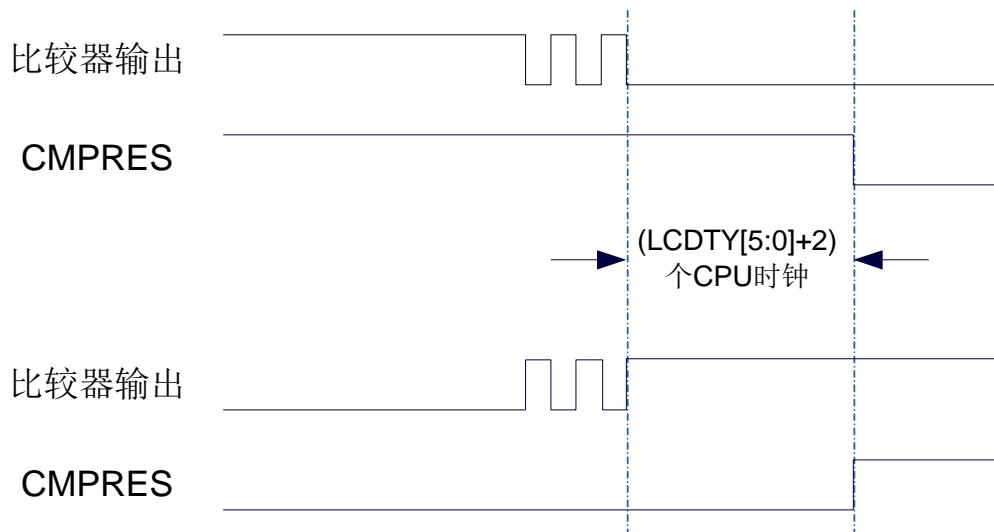
0: 使能 0.1us 模拟滤波功能

1: 关闭 0.1us 模拟滤波功能, 可略微提高比较器的比较速度。

LCDTY[5:0]: 数字滤波功能控制

数字滤波功能即为数字信号去抖动功能。当比较结果发生上升沿或者下降沿变化时, 比较器侦测变化后的信号必须维持 LCDTY 所设置的 CPU 时钟数不发生变化, 才认为数据变化是有效的; 否则将视同信号无变化。

注意: 当使能数字滤波功能后, 芯片内部实际的等待时钟需额外增加两个状态机切换时间, 即若 LCDTY 设置为 0 时, 为关闭数字滤波功能; 若 LCDTY 设置为非 0 值 n (n=1~63) 时, 则实际的数字滤波时间为 (n+2) 个系统时钟



19.3.3 比较器扩展配置寄存器 (CMPEXCFG)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMPEXCFG	FEAEH	CHYS[1:0]	-	-	-	CMPNS	CMPPS[1:0]		

CHYS[1:0]: 比较器 DC 迟滞输入选择

CHYS [1:0]	比较器 DC 迟滞输入选择
00	0mV
01	10mV
10	20mV
11	30mV

CMPNS: 比较器负端输入选择

0: P3.6

1: 选择内部 BandGap 经过 OP 后的电压 REFV 作为比较器负极输入源 (芯片在出厂时, 内部参考电压调整为 1.19V)

CMPPS[1:0]: 比较器正端输入选择

CMPPS[1:0]	比较器正端
00	P3.7
01	P5.0
10	P5.1
11	ADCIN

(STC8H1K08T 系列 / STC8H2K12U 系列 / STC8H2K32U 系列)

CMPPS[1:0]	比较器正端
00	P3.7
01	P1.0
10	P1.1
11	ADCIN

19.4 范例程序

19.4.1 旧版比较器的使用（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void CMP_Isr() interrupt 21
{
    CMPCRI &= ~0x40; //清中断标志
    if(CMPCRI & 0x01)
    {
        P10 = !P10; //上升沿中断测试端口
    }
    else
    {
        P11 = !P11; //下降沿中断测试端口
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CMPCR2 = 0x00;
    CMPCR2 &= ~0x80; //比较器正向输出
    // CMPCR2 |= 0x80; //比较器反向输出
    CMPCR2 &= ~0x40; //使能 0.1us 滤波
    // CMPCR2 |= 0x40; //禁止 0.1us 滤波
    CMPCR2 &= ~0x3f; //比较器结果直接输出
    CMPCR2 |= 0x10; //比较器结果经过 16 个去抖时钟后输出
    CMPCRI = 0x00;
    CMPCRI |= 0x30; //使能比较器边沿中断
    // CMPCRI &= ~0x20; //禁止比较器上升沿中断
    // CMPCRI |= 0x20; //使能比较器上升沿中断
    // CMPCRI &= ~0x10; //禁止比较器下降沿中断
    // CMPCRI |= 0x10; //使能比较器下降沿中断
    CMPCRI &= ~0x08; //P3.7 为 CMP+ 输入脚
    // CMPCRI |= 0x08; //ADC 输入脚为 CMP+ 输入脚
    // CMPCRI &= ~0x04; //内部 1.19V 参考信号源为 CMP- 输入脚
}
```

```

CMPCRI /= 0x04;                                //P3.6 为 CMP- 输入脚
//  CMPCRI &= ~0x02;                            //禁止比较器输出
CMPCRI /= 0x02;                                //使能比较器输出
CMPCRI /= 0x80;                                //使能比较器模块

EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH	
CMPCRI	DATA	0E6H	
CMPCR2	DATA	0E7H	
P1M1	DATA	091H	
P1M0	DATA	092H	
P0M1	DATA	093H	
P0M0	DATA	094H	
P2M1	DATA	095H	
P2M0	DATA	096H	
P3M1	DATA	0B1H	
P3M0	DATA	0B2H	
P4M1	DATA	0B3H	
P4M0	DATA	0B4H	
P5M1	DATA	0C9H	
P5M0	DATA	0CAH	
	ORG	0000H	
	LJMP	START	
	ORG	00ABH	
	LJMP	CMPISR	
	ORG	0100H	
CMPISR:	PUSH	ACC	
	ANL	CMPCRI,#NOT 40H	; 清中断标志
	MOV	A,CMPCRI	
	JB	ACC.0,RSING	
FALLING:	CPL	P1.0	; 下降沿中断测试端口
	POP	ACC	
	RETI		
RSING:	CPL	P1.1	; 上升沿中断测试端口
	POP	ACC	
	RETI		
START:	MOV	SP, #5FH	
	ORL	P_SW2,#80H	; 使能访问 XFR，没有冲突不用关闭
	MOV	P0M0, #00H	
	MOV	P0M1, #00H	
	MOV	P1M0, #00H	

```

MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      CMPCR2,#00H
ANL      CMPCR2,#NOT 80H ; 比较器正向输出
; ORL      CMPCR2,#80H ; 比较器反向输出
; ANL      CMPCR2,#NOT 40H ; 使能 0.1us 滤波
; ORL      CMPCR2,#40H ; 禁止 0.1us 滤波
; ANL      CMPCR2,#NOT 3FH ; 比较器结果直接输出
; ORL      CMPCR2,#10H ; 比较器结果经过 16 个去抖时钟后输出
MOV      CMPCR1,#00H
ORL      CMPCR1,#30H ; 使能比较器边沿中断
; ANL      CMPCR1,#NOT 20H ; 禁止比较器上升沿中断
; ORL      CMPCR1,#20H ; 使能比较器上升沿中断
; ANL      CMPCR1,#NOT 10H ; 禁止比较器下降沿中断
; ORL      CMPCR1,#10H ; 使能比较器下降沿中断
; ANL      CMPCR1,#NOT 08H ; P3.7 为 CMP+ 输入脚
; ORL      CMPCR1,#08H ; ADC 输入脚为 CMP+ 输入脚
; ANL      CMPCR1,#NOT 04H ; 内部 1.19V 参考信号源为 CMP- 输入脚
; ORL      CMPCR1,#04H ; P3.6 为 CMP- 输入脚
; ANL      CMPCR1,#NOT 02H ; 禁止比较器输出
; ORL      CMPCR1,#02H ; 使能比较器输出
; ORL      CMPCR1,#80H ; 使能比较器模块
SETB    EA

JMP     $

END

```

19.4.2 旧版比较器的使用（查询方式）

C 语言代码

// 测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80; // 使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;

```

```

P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

CMPCR2 = 0x00;
CMPCR2 &= ~0x80;                                // 比较器正向输出
// CMPCR2 |= 0x80;                                // 比较器反向输出
CMPCR2 &= ~0x40;                                // 使能 0.1us 滤波
// CMPCR2 |= 0x40;                                // 禁止 0.1us 滤波
// CMPCR2 &= ~0x3f;                                // 比较器结果直接输出
CMPCR2 |= 0x10;                                // 比较器结果经过 16 个去抖时钟后输出
CMPCRI = 0x00;
CMPCRI |= 0x30;                                // 使能比较器边沿中断
// CMPCRI &= ~0x20;                                // 禁止比较器上升沿中断
// CMPCRI |= 0x20;                                // 使能比较器上升沿中断
// CMPCRI &= ~0x10;                                // 禁止比较器下降沿中断
// CMPCRI |= 0x10;                                // 使能比较器下降沿中断
CMPCRI &= ~0x08;                                // P3.7 为 CMP+ 输入脚
// CMPCRI |= 0x08;                                // ADC 输入脚为 CMP+ 输入脚
// CMPCRI &= ~0x04;                                // 内部 1.19V 参考信号源为 CMP- 输入脚
CMPCRI |= 0x04;                                // P3.6 为 CMP- 输入脚
// CMPCRI &= ~0x02;                                // 禁止比较器输出
CMPCRI |= 0x02;                                // 使能比较器输出
CMPCRI |= 0x80;                                // 使能比较器模块

while (1)
{
    P10 = CMPCRI & 0x01;                          // 读取比较器比较结果
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
CMPCRI	DATA	0E6H
CMPCR2	DATA	0E7H
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG	DATA	0000H
LJMP	DATA	START
ORG	DATA	0100H

START:

```

MOV      SP, #5FH
ORL      P_SW2,#80H ;使能访问 XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      CMPCR2,#00H
ANL      CMPCR2,#NOT 80H ;比较器正向输出
; ORL      CMPCR2,#80H ;比较器反向输出
; ANL      CMPCR2,#NOT 40H ;使能 0.1us 滤波
; ORL      CMPCR2,#40H ;禁止 0.1us 滤波
; ANL      CMPCR2,#NOT 3FH ;比较器结果直接输出
; ORL      CMPCR2,#10H ;比较器结果经过 16 个去抖时钟后输出

MOV      CMPCRI,#00H
ORL      CMPCRI,#30H ;使能比较器边沿中断
; ANL      CMPCRI,#NOT 20H ;禁止比较器上升沿中断
; ORL      CMPCRI,#20H ;使能比较器上升沿中断
; ANL      CMPCRI,#NOT 10H ;禁止比较器下降沿中断
; ORL      CMPCRI,#10H ;使能比较器下降沿中断
; ANL      CMPCRI,#NOT 08H ;P3.7 为 CMP+ 输入脚
; ORL      CMPCRI,#08H ;ADC 输入脚为 CMP+ 输入脚
; ANL      CMPCRI,#NOT 04H ;内部 1.19V 参考信号源为 CMP- 输入脚
; ORL      CMPCRI,#04H ;P3.6 为 CMP- 输入脚
; ANL      CMPCRI,#NOT 02H ;禁止比较器输出
; ORL      CMPCRI,#02H ;使能比较器输出
; ORL      CMPCRI,#80H ;使能比较器模块

```

LOOP:

```

MOV      A,CMPCRI
MOV      C,ACC.0
MOV      PI.0,C ;读取比较器比较结果
JMP      LOOP

```

END

19.4.3 新版比较器的使用（中断方式）

C 语言代码

//测试工作频率为 11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

void CMP_Isr() interrupt 21
{

```

```

CMPCRI &= ~0x40; //清中断标志
if (CMPCRI & 0x01)
{
    P10 = !P10; //上升沿中断测试端口
}
else
{
    P11 = !P11; //下降沿中断测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CMPEXCFG = 0x00;
//    CMPEXCFG |= 0x40; //比较器DC 迟滞输入选择
//    //0:0mV; 0x40:10mV; 0x80:20mV; 0xc0:30mV

    CMPEXCFG &= ~0x03; //P3.7 为 CMP+ 输入脚
//    CMPEXCFG |= 0x01; //P5.0 为 CMP+ 输入脚
//    CMPEXCFG |= 0x02; //P5.1 为 CMP+ 输入脚
//    CMPEXCFG |= 0x03; //ADC 输入脚为 CMP+ 输入脚
    CMPEXCFG &= ~0x04; //P3.6 为 CMP- 输入脚
//    CMPEXCFG |= 0x04; //内部 1.19V 参考电压为 CMP- 输入脚

    CMPCR2 = 0x00; //比较器正向输出
    CMPCR2 &= ~0x80; //比较器反向输出
//    CMPCR2 |= 0x80; //使能 0.1us 滤波
//    CMPCR2 &= ~0x40; //禁止 0.1us 滤波
//    CMPCR2 |= 0x40; //比较器结果直接输出
//    CMPCR2 &= ~0x3f; //比较器结果经过 16 个去抖时钟后输出
    CMPCR2 |= 0x10;
    CMPCRI = 0x00;
    CMPCRI |= 0x30; //使能比较器边沿中断
//    CMPCRI &= ~0x20; //禁止比较器上升沿中断
//    CMPCRI |= 0x20; //使能比较器上升沿中断
//    CMPCRI &= ~0x10; //禁止比较器下降沿中断
//    CMPCRI |= 0x10; //使能比较器下降沿中断
//    CMPCRI &= ~0x02; //禁止比较器输出
//    CMPCRI |= 0x02; //使能比较器输出
//    CMPCRI |= 0x80; //使能比较器模块

EA = I;
while (1);

```

}

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>CMPCR1</i>	<i>DATA</i>	<i>0E6H</i>
<i>CMPCR2</i>	<i>DATA</i>	<i>0E7H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>CMPEXCFG</i>	<i>XDATA</i>	<i>0FEAEH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>00ABH</i>
	<i>LJMP</i>	<i>CMPISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>CMPISR:</i>		
	<i>PUSH</i>	<i>ACC</i>
	<i>ANL</i>	<i>CMPCR1,#NOT 40H</i>
		; 清中断标志
	<i>MOV</i>	<i>A,CMPCR1</i>
	<i>JB</i>	<i>ACC.0,RSING</i>
<i>FALLING:</i>		
	<i>CPL</i>	<i>P1.0</i>
		; 下降沿中断测试端口
	<i>POP</i>	<i>ACC</i>
	<i>RETI</i>	
<i>RSING:</i>		
	<i>CPL</i>	<i>P1.1</i>
		; 上升沿中断测试端口
	<i>POP</i>	<i>ACC</i>
	<i>RETI</i>	
<i>START:</i>		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>

```

MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      DPTR,# CMPEXCFG
CLR      A
ANL      A,#NOT 03H ; P3.7 为 CMP+ 输入脚
; ORL    A,#01H ; P5.0 为 CMP+ 输入脚
; ORL    A,#02H ; P5.1 为 CMP+ 输入脚
; ORL    A,#03H ; ADC 输入脚为 CMP+ 输入脚
; ANL    A,#NOT 04H ; P3.6 为 CMP- 输入脚
; ORL    A,# 04H ; 内部 1.19V 参考信号源为 CMP- 输入脚
MOVX   @DPTR,A

MOV      CMPCR2,#00H
ANL      CMPCR2,#NOT 80H ; 比较器正向输出
; ORL    CMPCR2,#80H ; 比较器反向输出
; ANL    CMPCR2,#NOT 40H ; 使能 0.1us 滤波
; ORL    CMPCR2,#40H ; 禁止 0.1us 滤波
; ANL    CMPCR2,#NOT 3FH ; 比较器结果直接输出
; ORL    CMPCR2,#10H ; 比较器结果经过 16 个去抖时钟后输出
MOV      CMPCRI,#00H
ORL      CMPCRI,#30H ; 使能比较器边沿中断
; ANL    CMPCRI,#NOT 20H ; 禁止比较器上升沿中断
; ORL    CMPCRI,#20H ; 使能比较器上升沿中断
; ANL    CMPCRI,#NOT 10H ; 禁止比较器下降沿中断
; ORL    CMPCRI,#10H ; 使能比较器下降沿中断
; ANL    CMPCRI,#NOT 02H ; 禁止比较器输出
; ORL    CMPCRI,#02H ; 使能比较器输出
; ORL    CMPCRI,#80H ; 使能比较器模块
SETB   EA

JMP     $

END

```

19.4.4 新版比较器的使用（查询方式）

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
}
```

```

P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

CMPEXCFG = 0x00;
// CMPEXCFG |= 0x40; // 比较器DC 迟滞输入选择, 0:0mV; 0x40:10mV; 0x80:20mV; 0xc0:30mV

CMPEXCFG &= ~0x03; // P3.7 为 CMP+ 输入脚
// CMPEXCFG |= 0x01; // P5.0 为 CMP+ 输入脚
// CMPEXCFG |= 0x02; // P5.1 为 CMP+ 输入脚
// CMPEXCFG |= 0x03; // ADC 输入脚为 CMP+ 输入脚
CMPEXCFG &= ~0x04; // P3.6 为 CMP- 输入脚
// CMPEXCFG |= 0x04; // 内部 1.19V 参考电压为 CMP- 输入脚

CMPCR2 = 0x00; // 比较器正向输出
CMPCR2 &= ~0x80; // 比较器反向输出
// CMPCR2 |= 0x80; // 使能 0.1us 滤波
CMPCR2 &= ~0x40; // 禁止 0.1us 滤波
// CMPCR2 |= 0x40; // 比较器结果直接输出
// CMPCR2 &= ~0x3f; // 比较器结果经过 16 个去抖时钟后输出
CMPCR2 |= 0x10; // CMPCR1 = 0x00;
CMPCR1 = 0x00; // CMPCR1 |= 0x30; // 使能比较器边沿中断
// CMPCR1 &= ~0x20; // 禁止比较器上升沿中断
// CMPCR1 |= 0x20; // 使能比较器上升沿中断
// CMPCR1 &= ~0x10; // 禁止比较器下降沿中断
// CMPCR1 |= 0x10; // 使能比较器下降沿中断
// CMPCR1 &= ~0x02; // 禁止比较器输出
CMPCR1 |= 0x02; // 使能比较器输出
CMPCR1 |= 0x80; // 使能比较器模块

while (1)
{
    P10 = CMPCR1 & 0x01; // 读取比较器比较结果
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
CMPCR1	DATA	0E6H
CMPCR2	DATA	0E7H
PIM1	DATA	091H
PIM0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH

```

CMPEXCFG XDATA 0FEAEH

ORG 0000H
LJMP START

ORG 0100H
START:
MOV SP, #5FH
ORL P_SW2,#80H ;使能访问 XFR，没有冲突不用关闭

MOV P0M0, #00H
MOV P0M1, #00H
MOV P1M0, #00H
MOV P1M1, #00H
MOV P2M0, #00H
MOV P2M1, #00H
MOV P3M0, #00H
MOV P3M1, #00H
MOV P4M0, #00H
MOV P4M1, #00H
MOV P5M0, #00H
MOV P5M1, #00H

MOV DPTR, # CMPEXCFG
CLR A
ANL A, #NOT 03H ; P3.7 为 CMP+ 输入脚
ORL A, #01H ; P5.0 为 CMP+ 输入脚
ORL A, #02H ; P5.1 为 CMP+ 输入脚
ORL A, #03H ; ADC 输入脚为 CMP+ 输入脚
ANL A, #NOT 04H ; P3.6 为 CMP- 输入脚
ORL A, #04H ; 内部 1.19V 参考信号源为 CMP- 输入脚
MOVX @DPTR, A

MOV CMPCR2, #00H
ANL CMPCR2, #NOT 80H ; 比较器正向输出
ORL CMPCR2, #80H ; 比较器反向输出
ANL CMPCR2, #NOT 40H ; 使能 0.1us 滤波
ORL CMPCR2, #40H ; 禁止 0.1us 滤波
ANL CMPCR2, #NOT 3FH ; 比较器结果直接输出
ORL CMPCR2, #10H ; 比较器结果经过 16 个去抖时钟后输出
MOV CMPCRI, #00H
ORL CMPCRI, #30H ; 使能比较器边沿中断
ANL CMPCRI, #NOT 20H ; 禁止比较器上升沿中断
ORL CMPCRI, #20H ; 使能比较器上升沿中断
ANL CMPCRI, #NOT 10H ; 禁止比较器下降沿中断
ORL CMPCRI, #10H ; 使能比较器下降沿中断
ANL CMPCRI, #NOT 02H ; 禁止比较器输出
ORL CMPCRI, #02H ; 使能比较器输出
ORL CMPCRI, #80H ; 使能比较器模块

LOOP:
MOV A, CMPCRI
MOV C, ACC.0
MOV P1.0, C ; 读取比较器比较结果
JMP LOOP

END

```

19.4.5 旧版比较器的多路复用应用（比较器+ADC 输入通道）

由于比较器的正极可以选择 ADC 的模拟输入通道，因此可以通过多路选择器和分时复用可实现多个比较器的应用。

注意：当比较器正极选择 ADC 输入通道时，请务必要打开 ADC_CONTR 寄存器中的 ADC 电源控制位 ADC_POWER 和 ADC 通道选择位 ADC_CHS

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 &= 0xfe;                                //设置 P1.0 为输入口
    P1M1 |= 0x01;                                //使能 ADC 模块并选择 P1.0 为 ADC 输入脚
    ADC_CONTR = 0x80;

    CMPCR2 = 0x00;
    CMPCR1 = 0x00;

    CMPCR1 |= 0x08;                                //ADC 输入脚为 CMP+ 输入脚
    CMPCR1 |= 0x04;                                //P3.6 为 CMP- 输入脚
    CMPCR1 |= 0x02;                                //使能比较器输出
    CMPCR1 |= 0x80;                                //使能比较器模块

    while (1);
}
```

汇编代码

;测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
CMPCR1	DATA	0E6H
CMPCR2	DATA	0E7H
ADC_CONTR	DATA	0BCH
P1M1	DATA	091H

```

P1M0      DATA      092H
P0M1      DATA      093H
P0M0      DATA      094H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP    START

        ORG      0100H
START:
        MOV      SP, #5FH
        ORL      P_SW2,#80H           ;使能访问 XFR, 没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        ANL      PIM0,#0FEH          ;设置 P1.0 为输入口
        ORL      PIM1,#01H
        MOV      ADC_CONTR,#80H       ;使能 ADC 模块并选择 P1.0 为 ADC 输入脚

        MOV      CMPCR2,#00H
        MOV      CMPCR1,#00H

        ORL      CMPCR1,#08H          ;ADC 输入脚为 CMP+ 输入脚
        ORL      CMPCR1,#04H          ;P3.6 为 CMP- 输入脚
        ORL      CMPCR1,#02H          ;使能比较器输出
        ORL      CMPCR1,#80H          ;使能比较器模块

LOOP:
        JMP      LOOP

END

```

19.4.6 新版比较器的多路复用应用（比较器+ADC 输入通道）

由于比较器的正极可以选择 ADC 的模拟输入通道，因此可以通过多路选择器和分时复用可实现多个比较器的应用。

注意：当比较器正极选择 ADC 输入通道时，请务必要打开 ADC_CONTR 寄存器中的 ADC 电源控制位 ADC_POWER 和 ADC 通道选择位 ADC_CHS

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR， 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    PIM0 &= 0xfe;                                //设置 P1.0 为输入口
    P1M1 |= 0x01;                                //使能 ADC 模块并选择 P1.0 为 ADC 输入脚

    CMPEXCFG = 0x00;
//    CMPEXCFG &= ~0x03;                          //P3.7 为 CMP+ 输入脚
//    CMPEXCFG |= 0x01;                            //P5.0 为 CMP+ 输入脚
//    CMPEXCFG |= 0x02;                            //P5.1 为 CMP+ 输入脚
//    CMPEXCFG |= 0x03;                            //ADC 输入脚为 CMP+ 输入脚
//    CMPEXCFG &= ~0x04;                          //P3.6 为 CMP- 输入脚
//    CMPEXCFG |= 0x04;                            //内部 1.19V 参考电压为 CMP- 输入脚

    CMPCR2 = 0x00;
    CMPCRI = 0x00;                                //使能比较器输出
    CMPCRI |= 0x02;                                //使能比较器模块
    CMPCRI |= 0x80;

    while (1);
}

```

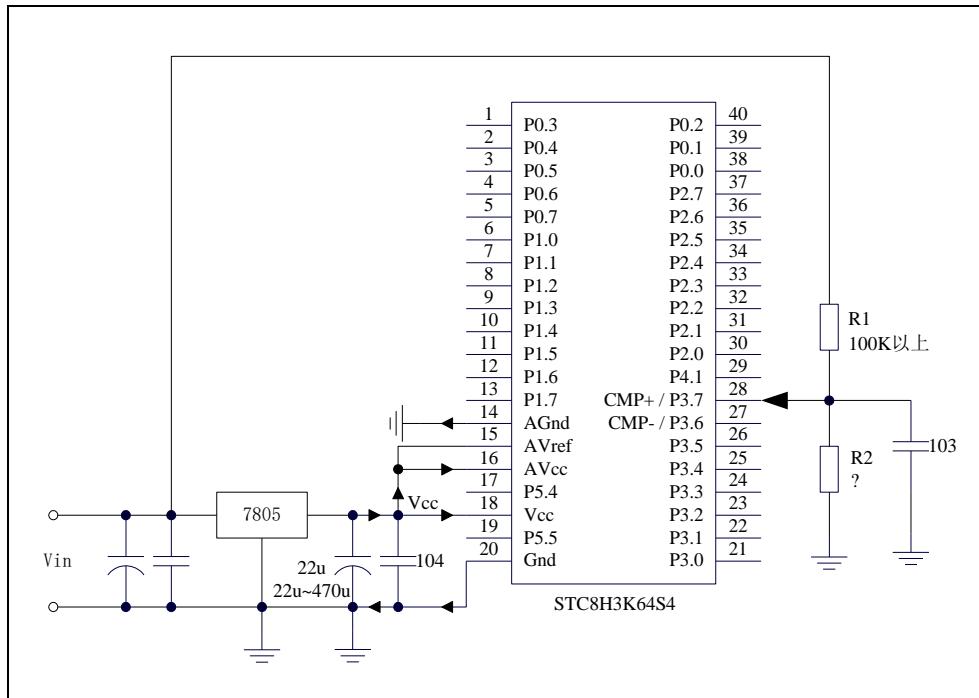
汇编代码

;测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
CMPCRI	DATA	0E6H
CMPCR2	DATA	0E7H
ADC_CONTR	DATA	0BCH
PIM1	DATA	091H
PIM0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H

<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
<i>CMPEXCFG</i>	<i>XDATA</i>	<i>0FEAEH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>START:</i>	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>ANL</i>	<i>P1M0,#0FEH</i>	;设置 P1.0 为输入口
	<i>ORL</i>	<i>P1M1,#01H</i>	
	<i>MOV</i>	<i>ADC_CONTR,#80H</i>	;使能 ADC 模块并选择 P1.0 为 ADC 输入脚
	<i>MOV</i>	<i>DPTR,# CMPEXCFG</i>	
	<i>CLR</i>	<i>A</i>	
;	<i>ANL</i>	<i>A,#NOT 03H</i>	; P3.7 为 CMP+ 输入脚
;	<i>ORL</i>	<i>A,#01H</i>	; P5.0 为 CMP+ 输入脚
;	<i>ORL</i>	<i>A,#02H</i>	; P5.1 为 CMP+ 输入脚
	<i>ORL</i>	<i>A,#03H</i>	;ADC 输入脚为 CMP+ 输入脚
	<i>ANL</i>	<i>A,#NOT 04H</i>	; P3.6 为 CMP- 输入脚
;	<i>ORL</i>	<i>A,# 04H</i>	; 内部 1.19V 参考信号源为 CMP- 输入脚
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>CMPCR2,#00H</i>	
	<i>MOV</i>	<i>CMPCR1,#00H</i>	
	<i>ORL</i>	<i>CMPCR1,#02H</i>	; 使能比较器输出
	<i>ORL</i>	<i>CMPCR1,#80H</i>	; 使能比较器模块
<i>LOOP:</i>	<i>JMP</i>	<i>LOOP</i>	
	<i>END</i>		

19.4.7 比较器作外部掉电检测(掉电过程中应及时保存用户数据到 EEPROM 中)

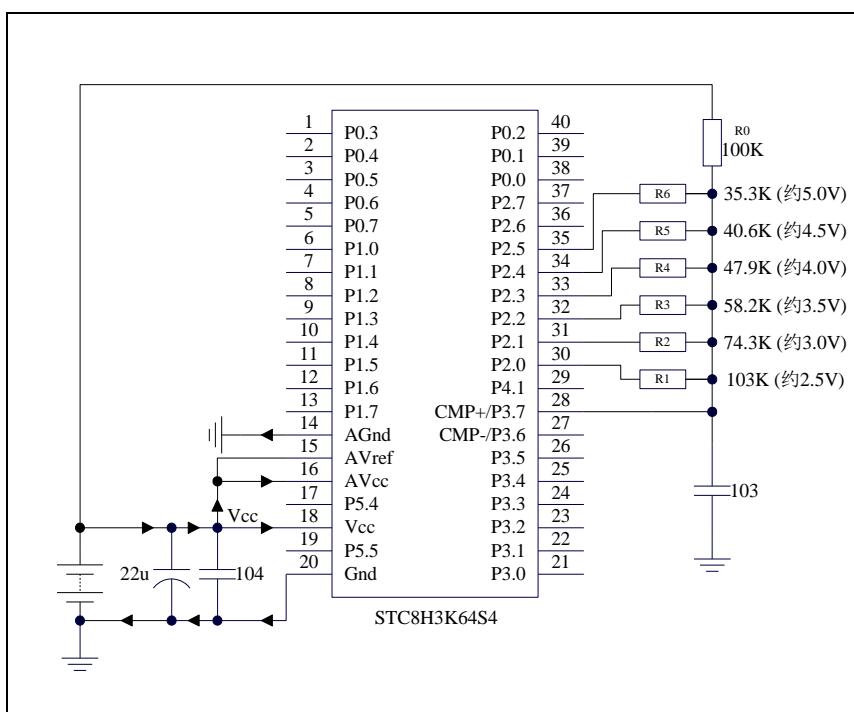


上图中电阻 R1 和 R2 对稳压块 7805 的前端电压进行分压，分压后的电压作为比较器 CMP+ 的外部输入与内部 1.19V 参考信号源进行比较。

一般当交流电在 220V 时，稳压块 7805 前端的直流电压为 11V，但当交流电压降到 160V 时，稳压块 7805 前端的直流电压为 8.5V。当稳压块 7805 前端的直流电压低于或等于 8.5V 时，该前端输入的直流电压被电阻 R1 和 R2 分压到比较器正极输入端 CMP+，CMP+ 端输入电压低于内部 1.19V 参考信号源，此时可产生比较器中断，这样在掉电检测时就有充足的时间将数据保存到 EEPROM 中。当稳压块 7805 前端的直流电压高于 8.5V 时，该前端输入的直流电压被电阻 R1 和 R2 分压到比较器正极输入端 CMP+，CMP+ 端输入电压高于内部 1.19V 参考信号源，此时 CPU 可继续正常工作。

内部 1.19V 参考信号源即为内部 BandGap 经过 OP 后的电压 REFV (芯片在出厂时，**内部参考信号源调整为 1.19V**)。具体的数值要通过读取内部 1.19V 参考信号源在内部 RAM 区或者 Flash 程序存储器 (ROM) 区所占用的地址的值获得。对于 STC8 系列，内部 1.19V 参考信号源值在 RAM 和 Flash 程序存储器 (ROM) 中的存储地址请参考 “[存储器中的特殊参数](#)” 章节

19.4.8 比较器检测工作电压（电池电压）



上图中，利用电阻分压的原理可以近似的测量出 MCU 的工作电压（选通的通道，MCU 的 IO 口输出低电平，端口电压值接近 Gnd，未选通的通道，MCU 的 IO 口输出开漏模式的高，不影响其他通道）。

比较器的负端选择内部 1.19V 参考信号源，正端选择通过电阻分压后输入到 CMP+管脚的电压值。

初始化时 P2.5~P2.0 口均设置为开漏模式，并输出高。首先 P2.0 口输出低电平，此时若 Vcc 电压低于 2.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 2.5V 则比较器的比较值为 1；

若确定 Vcc 高于 2.5V，则将 P2.0 口输出高，P2.1 口输出低电平，此时若 Vcc 电压低于 3.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 3.0V 则比较器的比较值为 1；

若确定 Vcc 高于 3.0V，则将 P2.1 口输出高，P2.2 口输出低电平，此时若 Vcc 电压低于 3.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 3.5V 则比较器的比较值为 1；

若确定 Vcc 高于 3.5V，则将 P2.2 口输出高，P2.3 口输出低电平，此时若 Vcc 电压低于 4.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 4.0V 则比较器的比较值为 1；

若确定 Vcc 高于 4.0V，则将 P2.3 口输出高，P2.4 口输出低电平，此时若 Vcc 电压低于 4.5V 则比较器的比较值为 0，反之若 Vcc 电压高于 4.5V 则比较器的比较值为 1；

若确定 Vcc 高于 4.5V，则将 P2.4 口输出高，P2.5 口输出低电平，此时若 Vcc 电压低于 5.0V 则比较器的比较值为 0，反之若 Vcc 电压高于 5.0V 则比较器的比较值为 1。

C 语言代码

//测试工作频率为11.0592MHz

```
#include "stc8h.h"
```

```
#include "intrins.h"
```

```
void delay()
```

```
{
```

```
    char i;
```

```
    for (i=0; i<20; i++);
```

}

```

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    unsigned char v;

    P2M0 = 0x3f;                                  //P2.5~P2.0 初始化为开漏模式
    P2M1 = 0x3f;
    P2 = 0xff;

    CMPCR2 = 0x10;                                //比较器结果经过 16 个去抖时钟后输出
    CMPCR1 = 0x00;
    CMPCR1 &= ~0x08;                            //P3.7 为 CMP+ 输入脚
    CMPCR1 &= ~0x04;                            //内部 1.19V 参考信号源为 CMP- 输入脚
    CMPCR1 &= ~0x02;                            //禁止比较器输出
    CMPCR1 |= 0x80;                             //使能比较器模块

    while (1)
    {
        v = 0x00;                                //电压<2.5V
        P2 = 0xfe;                               //P2.0 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x01;                                //电压>2.5V
        P2 = 0xfd;                               //P2.1 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x03;                                //电压>3.0V
        P2 = 0xfb;                               //P2.2 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x07;                                //电压>3.5V
        P2 = 0xf7;                               //P2.3 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x0f;                                //电压>4.0V
        P2 = 0xef;                               //P2.4 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x1f;                                //电压>4.5V
        P2 = 0xdf;                               //P2.5 输出 0
        delay();
        if (!(CMPCR1 & 0x01)) goto ShowVol;
        v = 0x3f;                                //电压>5.0V
    }
}

```

ShowVol:

```

P2 = 0xff;
P0 = ~v;
}
}
```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH
CMPCR1     DATA      0E6H
CMPCR2     DATA      0E7H

P2M0       DATA      096H
P2M1       DATA      095H
P1M1       DATA      091H
P1M0       DATA      092H
P0M1       DATA      093H
P0M0       DATA      094H
P2M1       DATA      095H
P2M0       DATA      096H
P3M1       DATA      0B1H
P3M0       DATA      0B2H
P4M1       DATA      0B3H
P4M0       DATA      0B4H
P5M1       DATA      0C9H
P5M0       DATA      0CAH

ORG        0000H
LJMP       START

ORG        0100H
START:
MOV        SP, #5FH
ORL        P_SW2, #80H           ; 使能访问 XFR，没有冲突不用关闭

MOV        P0M0, #00H
MOV        P0M1, #00H
MOV        P1M0, #00H
MOV        P1M1, #00H
MOV        P2M0, #00H
MOV        P2M1, #00H
MOV        P3M0, #00H
MOV        P3M1, #00H
MOV        P4M0, #00H
MOV        P4M1, #00H
MOV        P5M0, #00H
MOV        P5M1, #00H

MOV        P2M0, #00111111B      ; P2.5~P2.0 初始化为开漏模式
MOV        P2M1, #00111111B
MOV        P2, #0FFH
MOV        CMPCR2, #10H          ; 比较器结果经过 16 个去抖时钟后输出
MOV        CMPCR1, #00H
ANL        CMPCR1, #NOT 08H      ; P3.7 为 CMP+ 输入脚
ANL        CMPCR1, #NOT 04H      ; 内部 1.19V 参考信号源为 CMP- 输入脚
ANL        CMPCR1, #NOT 02H      ; 禁止比较器输出

```

ORL	CMPCR1,#80H	;使能比较器模块
LOOP:		
MOV	R0,#0000000B	;电压<2.5V
MOV	P2,#1111110B	;P2.0 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00000001B	;电压>2.5V
MOV	P2,#11111101B	;P2.1 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00000011B	;电压>3.0V
MOV	P2,#11111011B	P2.2 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00000111B	;电压>3.5V
MOV	P2,#11110111B	;P2.3 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00001111B	;电压>4.0V
MOV	P2,#11101111B	;P2.4 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00011111B	;电压>4.5V
MOV	P2,#11011111B	;P2.5 输出0
CALL	DELAY	
MOV	A,CMPCR1	
JNB	ACC.0,SKIP	
MOV	R0,#00111111B	;电压>5.0V
SKIP:		
MOV	P2,#1111111B	
MOV	A,R0	
CPL	A	
MOV	P0,A	;P0.5~P0.0 口显示电压
JMP	LOOP	
DELAY:		
MOV	R0,#20	
DJNZ	R0,\$	
RET		
END		

20 IAP/EEPROM/DATA-FLASH

STC8H 系列单片机内部集成了大容量的 EEPROM。利用 ISP/IAP 技术可将内部 Data Flash 当 EEPROM，擦写次数在 10 万次以上。EEPROM 可分为若干个扇区，每个扇区包含 512 字节。

注意：EEPROM 的写操作只能将字节中的 1 写为 0，当需要将字节中的 0 写为 1，则必须执行扇区擦除操作。EEPROM 的读/写操作是以 1 字节为单位进行，而 EEPROM 擦除操作是以 1 扇区（512 字节）为单位进行，在执行擦除操作时，如果目标扇区中有需要保留的数据，则必须预先将这些数据读取到 RAM 中暂存，待擦除完成后再将保存的数据和需要更新的数据一起再写回 EEPROM/DATA-FLASH。

所以在使用 EEPROM 时，建议同一次修改的数据放在同一个扇区，不是同一次修改的数据放在不同的扇区，不一定要用满。数据存储器的擦除操作是按扇区进行的（每扇区 512 字节）。

EEPROM 可用于保存一些需要在应用过程中修改并且掉电不丢失的参数数据。在用户程序中，可以对 EEPROM 进行字节读/字节编程/扇区擦除操作。在工作电压偏低时，建议不要进行 EEPROM 操作，以免发送数据丢失的情况。

20.1 EEPROM 操作时间

- 读取 1 字节：4 个系统时钟（使用 MOVC 指令读取更方便快捷）
- 编程 1 字节：约 30~40us（实际的编程时间为 6~7.5us，但还需要加上状态转换时间和各种控制信号的 SETUP 和 HOLD 时间）
- 擦除 1 扇区（512 字节）：约 4~6ms

EEPROM 操作所需时间是硬件自动控制的，用户只需要正确设置 IAP_TPS 寄存器即可。

IAP_TPS=系统工作频率/1000000（小数部分四舍五入进行取整）

例如：系统工作频率为 12MHz，则 IAP_TPS 设置为 12

又例如：系统工作频率为 22.1184MHz，则 IAP_TPS 设置为 22

再例如：系统工作频率为 5.5296MHz，则 IAP_TPS 设置为 6

20.2 EEPROM 相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
IAP_DATA	IAP 数据寄存器	C2H									1111,1111
IAP_ADDRH	IAP 高地址寄存器	C3H									0000,0000
IAP_ADDRL	IAP 低地址寄存器	C4H									0000,0000
IAP_CMD	IAP 命令寄存器	C5H	-	-	-	-	-	-	CMD[1:0]	xxxx,xx00	
IAP_TRIG	IAP 触发寄存器	C6H									0000,0000
IAP_CONTR	IAP 控制寄存器	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	-	-	-	0000,xxxx
IAP_TPS	IAP 等待时间控制寄存器	F5H	-	-	IAPTPS[5:0]						xx00,0000

20.2.1 EEPROM 数据寄存器 (IAP_DATA)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_DATA	C2H								

在进行 EEPROM 的读操作时，命令执行完成后读出的 EEPROM 数据保存在 IAP_DATA 寄存器中。在进行 EEPROM 的写操作时，在执行写命令前，必须将待写入的数据存放在 IAP_DATA 寄存器中，再发送写命令。擦除 EEPROM 命令与 IAP_DATA 寄存器无关。

20.2.2 EEPROM 地址寄存器 (IAP_ADDR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_ADDRH	C3H								
IAP_ADDRL	C4H								

EEPROM 进行读、写、擦除操作的目标地址寄存器。IAP_ADDRH 保存地址的高字节，IAP_ADDRL 保存地址的低字节

20.2.3 EEPROM 命令寄存器 (IAP_CMD)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CMD	C5H	-	-	-	-	-	-	CMD[1:0]	

CMD[1:0]: 发送EEPROM操作命令

00: 空操作

01: 读 EEPROM 命令。读取目标地址所在的 1 字节。

10: 写 EEPROM 命令。写目标地址所在的 1 字节。**注意：写操作只能将目标字节中的 1 写为 0，而不能将 0 写为 1。一般当目标字节不为 FFH 时，必须先擦除。**

11: 擦除 EEPROM。擦除目标地址所在的 1 页（1 扇区/512 字节）。**注意：擦除操作会一次擦除 1 个扇区（512 字节），整个扇区的内容全部变成 FFH。**

20.2.4 EEPROM 触发寄存器 (IAP_TRIG)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_TRIG	C6H								

设置完成 EEPROM 读、写、擦除的命令寄存器、地址寄存器、数据寄存器以及控制寄存器后，需要向触发寄存器 IAP_TRIG 依次写入 5AH、A5H（顺序不能交换）两个触发命令来触发相应的读、写、擦除操作。操作完成后，EEPROM 地址寄存器 IAP_ADDRH、IAP_ADDRL 和 EEPROM 命令寄存器 IAP_CMD 的内容不变。如果接下来要对下一个地址的数据进行操作，需手动更新地址寄存器 IAP_ADDRH 和寄存器 IAP_ADDRL 的值。

注意：每次 EEPROM 操作时，都要对 IAP_TRIG 先写入 5AH，再写入 A5H，相应的命令才会生效。写完触发命令后，CPU 会处于 IDLE 等待状态，直到相应的 IAP 操作执行完成后 CPU 才会从 IDLE 状态返回正常状态继续执行 CPU 指令。

如果主循环代码和中断代码中都有使用 IAP 方式对 EEPROM 进行操作，则 EEPROM 操作时必须关闭中断。因为如果在主循环代码中设置完成 IAP 相关寄存器后在 IAP 触发前或 IAP 触发一半时，产生了中断，而在中断中进行 EEPROM 操作时，又对 IAP 相关寄存器进行重新设置，中断返回后 IAP 相关寄存器已经发生了改变，回到主循环中继续之前的 IAP 操作必然会出错。

建议程序代码按照如下方式编写：

```

...                                ;设置 ISP/IAP 的其他相关寄存器
CLR EA                            ;触发 ISP/IAP 前，先关闭总中断
MOV IAP_TRIG, #5AH                ;先送 5Ah，到 ISP/IAP 触发寄存器
MOV IAP_TRIG, #0A5H                ;再送 A5h，到 ISP/IAP 触发寄存器，每次都需如此
                                    ;送完 A5h 后，ISP/IAP 命令立即被触发启动
                                    ;IAP 命令执行完成前，CPU 不会向下继续执行程序
SETB EA                            ;IAP 完成后再开总中断，防止【IAP 触发命令序列】被任意中断打断
                                    ;而这个任意中断服务程序中又有【IAP 触发命令序列】
                                    ;确保操作的【原子性】

```

20.2.5 EEPROM 控制寄存器 (IAP_CONTR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CONTR	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	-	-	-

IAPEN: EEPROM 操作使能控制位

- 0: 禁止 EEPROM 操作
- 1: 使能 EEPROM 操作

SWBS: 软件复位选择控制位，(需要与 SWRST 配合使用)

- 0: 软件复位后从用户代码开始执行程序
- 1: 软件复位后从系统 ISP 监控代码区开始执行程序

SWRST: 软件复位控制位

- 0: 无动作
- 1: 产生软件复位

CMD_FAIL: EEPROM 操作失败状态位，需要软件清零

- 0: EEPROM 操作正确
- 1: EEPROM 操作失败

20.2.6 EEPROM 等待时间控制寄存器 (IAP_TPS)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
IAP_TPS	F5H	-	-						IAPTPS[5:0]

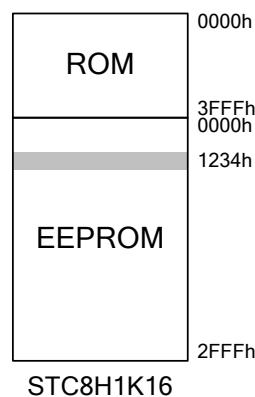
需要根据工作频率进行设置

若工作频率为12MHz，则需要将IAP_TPS设置为12；若工作频率为24MHz，则需要将IAP_TPS设置为24，其他频率以此类推。

20.3 EEPROM 大小及地址

STC8H 系列单片机内部均有用于保存用户数据的 EEPROM。内部的 EEPROM 有 3 操作方式：读、写和擦除，其中擦除操作是以扇区为单位进行操作，每扇区为 512 字节，即每执行一次擦除命令就会擦除一个扇区，而读数据和写数据都是以字节为单位进行操作的，即每执行一次读或者写命令时只能读出或者写入一个字节。

STC8H 系列单片机内部的 EEPROM 的访问方式有两种：IAP 方式和 MOVC 方式。IAP 方式可对 EEPROM 执行读、写、擦除操作，但 MOVC 只能对 EEPROM 进行读操作，而不能进行写和擦除操作。无论是使用 IAP 方式还是使用 MOVC 方式访问 EEPROM，首先都需要设置正确的目标地址。IAP 方式时，目标地址与 EEPROM 实际的物理地址是一致的，均是从地址 0000H 开始访问，但若要使用 MOVC 指令进行读取 EEPROM 数据时，目标地址必须是在 EEPROM 实际的物理地址的基础上还有加上程序大小的偏移。下面以 STC8H1K16 这个型号为例，对目标地址进行详细说明：



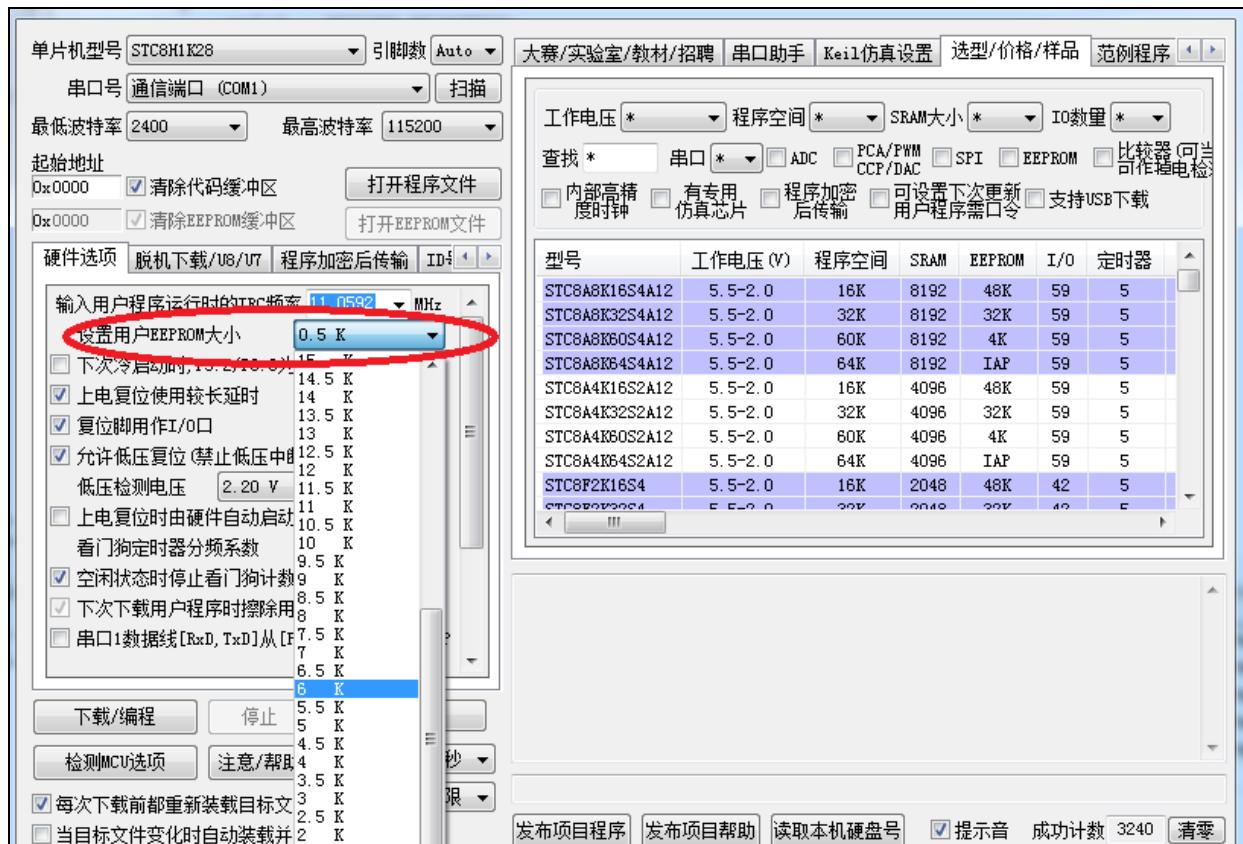
STC8H1K16 的程序空间为 16K 字节 (0000h~3FFFh)，EEPROM 空间为 12K (0000h~2FFFh)。当需要对 EEPROM 物理地址 1234h 的单元进行读、写、擦除时，若使用 IAP 方式进行访问时，设置的目标地址为 1234h，即 IAP_ADDRH 设置 12h，IAP_ADDRL 设置 34h，然后设置相应的触发命令即可对 1234h 单元进行正确操作了。但若是使用 MOVC 方式读取 EEPROM 的 1234h 单元，则必须在 1234h 的基础上还有加上 ROM 空间的大小 4000h，即必须将 DPTR 设置为 5234h，然后才能使用 MOVC 指令进行读取。

注意：由于擦除是以 512 字节为单位进行操作的，所以执行擦除操作时所设置的目标地址的低 9 位是无意义的。例如：执行擦除命令时，设置地址 1234H/1200H/1300H/13FFH，最终执行擦除的动作都是相同的，都是擦除 1200H~13FFH 这 512 字节。

不同型号内部 EEPROM 的大小及访问地址会存在差异，针对各个型号 EEPROM 的详细大小和地址请参考下表：

型号	大小	扇区	IAP方式读/写/擦除		MOVC读取	
			起始地址	结束地址	起始地址	结束地址
STC8H1K16	12K	24	0000h	2FFFh	4000h	6FFFh
STC8H1K24	4K	8	0000h	0FFFh	6000h	6FFFh
STC8H1K28	用户自定义 ^[1]					
STC8H1K33	用户自定义 ^[1]					
STC8H1K08	4K	8	0000h	0FFFh	2000h	2FFFh
STC8H1K12	用户自定义 ^[1]					
STC8H1K17	用户自定义 ^[1]					
STC8H3K32S4	32K	64	0000h	7FFFh	8000h	FFFFh
STC8H3K48S4	16K	32	0000h	3FFFh	C000h	FFFFh
STC8H3K60S4	4K	8	0000h	0FFFh	F000h	FFFFh
STC8H3K64S4	用户自定义 ^[1]					
STC8H3K32S2	32K	64	0000h	7FFFh	8000h	FFFFh
STC8H3K48S2	16K	32	0000h	3FFFh	C000h	FFFFh
STC8H3K60S2	4K	8	0000h	0FFFh	F000h	FFFFh
STC8H3K64S2	用户自定义 ^[1]					
STC8H8K32U	32K	64	0000h	7FFFh	8000h	FFFFh
STC8H8K48U	16K	32	0000h	3FFFh	C000h	FFFFh
STC8H8K60U	4K	8	0000h	0FFFh	F000h	FFFFh
STC8H8K64U	用户自定义 ^[1]					
STC8H4K32TL	32K	64	0000h	7FFFh	8000h	FFFFh
STC8H4K48TL	16K	32	0000h	3FFFh	C000h	FFFFh
STC8H4K60TL	4K	8	0000h	0FFFh	F000h	FFFFh
STC8H4K64TL	用户自定义 ^[1]					
STC8H4K32LCD	32K	64	0000h	7FFFh	8000h	FFFFh
STC8H4K48LCD	16K	32	0000h	3FFFh	C000h	FFFFh
STC8H4K60LCD	4K	8	0000h	0FFFh	F000h	FFFFh
STC8H4K64LCD	用户自定义 ^[1]					
STC8H1K08T	4K	8	0000h	0FFFh	2000h	2FFFh
STC8H1K12T	用户自定义 ^[1]					
STC8H1K17T	用户自定义 ^[1]					
STC8H2K08U	4K	8	0000h	0FFFh	2000h	2FFFh
STC8H2K12U	用户自定义 ^[1]					
STC8H2K17U	用户自定义 ^[1]					
STC8H2K16U	16K	32	0000h	3FFFh	4000h	7FFFh
STC8H2K24U	8K	16	0000h	1FFFh	2000h	7FFFh
STC8H2K32U	用户自定义 ^[1]					

^[1]: 这个为特殊型号，这个型号的 EEPROM 大小是可用在 ISP 下载时用户自己设置的。如下图所示：



用户可根据自己的需要在整个 FLASH 空间中规划出任意不超过 FLASH 大小的 EEPROM 空间，但需要注意：**EEPROM 总是从后向前进行规划的。**

例如：STC8H1K28 这个型号的 FLASH 为 28K，此时若用户想分出其中的 8K 作为 EEPROM 使用，则 EEPROM 的物理地址则为 28K 的最后 8K，物理地址为 5000h~6FFFh，当然，用户若使用 IAP 的方式进行访问，目标地址仍然从 0000h 开始，到 1FFFh 结束，当使用 MOVC 读取则需要从 5000h 开始，到 6FFFh 结束。**注意：**STC8H1K28 这个型号的程序空间为 28K，即整个 28K 的范围均可运行程序，即使在 ISP 下载时将 28K 最后的 8K 设置为 EEPROM 使用，但这分配的 8K 空间仍然可以运行程序。其它可自定义 EEPROM 大小的型号与此类似。

20.4 范例程序

20.4.1 EEPROM 基本操作

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"
```

```
void IapIdle()
{
    IAP_CONTR = 0;                                //关闭IAP 功能
    IAP_CMD = 0;                                   //清除命令寄存器
    IAP_TRIG = 0;                                 //清除触发寄存器
    IAP_ADDRH = 0x80;                             //将地址设置到非IAP 区域
    IAP_ADDRL = 0;
}

char IapRead(int addr)
{
    char dat;

    EA = 0;                                         //关闭总中断
    IAP_CONTR = 0x80;                            //使能IAP
    IAP_TPS = 11;                                //设置等待参数 11MHz
    IAP_CMD = 1;                                 //设置IAP 读命令
    IAP_ADDRL = addr;                           //设置IAP 低地址
    IAP_ADDRH = addr >> 8;                      //设置IAP 高地址
    IAP_TRIG = 0x5a;                            //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                            //写触发命令(0xa5)
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    dat = IAP_DATA;                            //读IAP 数据
    IapIdle();                                  //关闭IAP 功能
    EA = 1;                                     //再开总中断,防止【IAP 触发命令序列】被任意中断打断
                                                //而这个任意中断服务程序中又有【IAP 触发命令序列】
                                                //确保操作的【原子性】

    return dat;
}

void IapProgram(int addr, char dat)
{
    EA = 0;                                         //关闭总中断
    IAP_CONTR = 0x80;                            //使能IAP
    IAP_TPS = 11;                                //设置等待参数 11MHz
    IAP_CMD = 2;                                 //设置IAP 写命令
    IAP_ADDRL = addr;                           //设置IAP 低地址
    IAP_ADDRH = addr >> 8;                      //设置IAP 高地址
    IAP_DATA = dat;                            //写IAP 数据
    IAP_TRIG = 0x5a;                            //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                            //写触发命令(0xa5)
    _nop_();
```

```

_nop_();
_nop_();
_nop_();
IapIdle();
EA = I;
} //关闭IAP 功能
//再开总中断,防止【IAP 触发命令序列】被任意中断打断
//而这个任意中断服务程序中又有 【IAP 触发命令序列】
//确保操作的 【原子性】

void IapErase(int addr)
{
    EA = 0; //关闭总中断
    IAP_CONTR = 0x80; //使能IAP
    IAP_TPS = 11; //设置等待参数 11MHz
    IAP_CMD = 3; //设置IAP 擦除命令
    IAP_ADDRL = addr; //设置IAP 低地址
    IAP_ADDRH = addr >> 8; //设置IAP 高地址
    IAP_TRIG = 0x5a; //写触发命令(0x5a)
    IAP_TRIG = 0xa5; //写触发命令(0xa5)

    _nop_();
    _nop_();
    _nop_();
    _nop_();
    IapIdle(); //关闭IAP 功能
    EA = I; //再开总中断,防止【IAP 触发命令序列】被任意中断打断
} //而这个任意中断服务程序中又有 【IAP 触发命令序列】
//确保操作的 【原子性】

}

void main()
{
    P_SW2 |= 0x80; //使能访问XFR · 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IapErase(0x0400);
    P0 = IapRead(0x0400); //P0=0xff
    IapProgram(0x0400, 0x12);
    P1 = IapRead(0x0400); //P1=0x12

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2 DATA 0BAH

<i>IAP_DATA</i>	DATA	0C2H
<i>IAP_ADDRH</i>	DATA	0C3H
<i>IAP_ADDRL</i>	DATA	0C4H
<i>IAP_CMD</i>	DATA	0C5H
<i>IAP_TRIG</i>	DATA	0C6H
<i>IAP CONTR</i>	DATA	0C7H
<i>IAP_TPS</i>	DATA	0F5H

<i>PIMI</i>	DATA	091H
<i>PIM0</i>	DATA	092H
<i>P0M1</i>	DATA	093H
<i>P0M0</i>	DATA	094H
<i>P2M1</i>	DATA	095H
<i>P2M0</i>	DATA	096H
<i>P3M1</i>	DATA	0B1H
<i>P3M0</i>	DATA	0B2H
<i>P4M1</i>	DATA	0B3H
<i>P4M0</i>	DATA	0B4H
<i>P5M1</i>	DATA	0C9H
<i>P5M0</i>	DATA	0CAH

<i>ORG</i>	0000H
<i>LJMP</i>	START
<i>ORG</i>	0100H

IAP_IDLE:

<i>MOV</i>	<i>IAP CONTR,#0</i>	;关闭 IAP 功能
<i>MOV</i>	<i>IAP CMD,#0</i>	;清除命令寄存器
<i>MOV</i>	<i>IAP TRIG,#0</i>	;清除触发寄存器
<i>MOV</i>	<i>IAP ADDRH,#80H</i>	;将地址设置到非 IAP 区域
<i>MOV</i>	<i>IAP ADDRL,#0</i>	
<i>RET</i>		

IAP_READ:

<i>CLR</i>	EA	;关闭总中断
<i>MOV</i>	<i>IAP CONTR,#80H</i>	;使能 IAP
<i>MOV</i>	<i>IAP TPS,#12</i>	;设置等待参数 12MHz
<i>MOV</i>	<i>IAP CMD,#1</i>	;设置 IAP 读命令
<i>MOV</i>	<i>IAP ADDRL,DPL</i>	;设置 IAP 低地址
<i>MOV</i>	<i>IAP ADDRH,DPH</i>	;设置 IAP 高地址
<i>MOV</i>	<i>IAP TRIG,#5AH</i>	;写触发命令(0x5a)
<i>MOV</i>	<i>IAP TRIG,#0A5H</i>	;写触发命令(0xa5)
<i>NOP</i>		
<i>MOV</i>	<i>A,IAP DATA</i>	;读取 IAP 数据
<i>LCALL</i>	<i>IAP_IDLE</i>	;关闭 IAP 功能
<i>SETB</i>	EA	;再开总中断,防止【IAP 触发命令序列】被任意中断打断 ;而这个任意中断服务程序中又有【IAP 触发命令序列】 ;确保操作的【原子性】
<i>RET</i>		

IAP_PROGRAM:

<i>CLR</i>	EA	;关闭总中断
<i>MOV</i>	<i>IAP CONTR,#80H</i>	;使能 IAP
<i>MOV</i>	<i>IAP TPS,#12</i>	;设置等待参数 12MHz

<i>MOV</i>	<i>IAP_CMD,#2</i>	; 设置 IAP 写命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	; 设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	; 设置 IAP 高地址
<i>MOV</i>	<i>IAP_DATA,A</i>	; 写 IAP 数据
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	; 写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	; 写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	; 关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	; 再开总中断, 防止【IAP 触发命令序列】被任意中断打断 ; 而这个任意中断服务程序中又有【IAP 触发命令序列】 ; 确保操作的【原子性】
<i>RET</i>		

IAP_ERASE:

<i>CLR</i>	<i>EA</i>	; 关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	; 使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	; 设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#3</i>	; 设置 IAP 擦除命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	; 设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	; 设置 IAP 高地址
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	; 写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	; 写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	; 关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	; 再开总中断, 防止【IAP 触发命令序列】被任意中断打断 ; 而这个任意中断服务程序中又有【IAP 触发命令序列】 ; 确保操作的【原子性】
<i>RET</i>		

START:

<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>DPTR, #0400H</i>	
<i>LCALL</i>	<i>IAP_ERASE</i>	
<i>MOV</i>	<i>DPTR, #0400H</i>	
<i>LCALL</i>	<i>IAP_READ</i>	
<i>MOV</i>	<i>P0,A</i>	; P0=0FFH
<i>MOV</i>	<i>DPTR, #0400H</i>	
<i>MOV</i>	<i>A, #12H</i>	

```

LCALL      IAP_PROGRAM
MOV        DPTR,#0400H
LCALL      IAP_READ
MOV        PI,A           ;PI=12H
SJMP      $
END

```

20.4.2 使用 MOVC 读取 EEPROM

C 语言代码

//测试工作频率为11.0592MHz

```

#include "stc8h.h"
#include "intrins.h"

#define IAP_OFFSET      0x4000H          //STC8H1K16

void IapIdle()
{
    IAP_CONTR = 0;                   //关闭IAP 功能
    IAP_CMD = 0;                    //清除命令寄存器
    IAP_TRIG = 0;                  //清除触发寄存器
    IAP_ADDRH = 0x80;              //将地址设置到非IAP 区域
    IAP_ADDRL = 0;
}

unsigned char IapRead(unsigned int addr)
{
    addr += IAP_OFFSET;            //使用 MOVC 读取 EEPROM 需要加上相应的偏移
    return *(unsigned char code *)(addr); //使用 MOVC 读取数据
}

void IapProgram(unsigned int addr, unsigned char dat)
{
    EA = 0;                        //关闭总中断
    IAP_CONTR = 0x80;              //使能IAP
    IAP_TPS = 11;                  //设置等待参数 11MHz
    IAP_CMD = 2;                  //设置IAP 写命令
    IAP_ADDRL = addr;              //设置IAP 低地址
    IAP_ADDRH = addr >> 8;        //设置IAP 高地址
    IAP_DATA = dat;                //写IAP 数据
    IAP_TRIG = 0x5a;                //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                //写触发命令(0xa5)
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    IapIdle();                     //关闭IAP 功能
    EA = 1;                        //再开总中断,防止【IAP 触发命令序列】被任意中断打断
                                    //而这个任意中断服务程序中又有【IAP 触发命令序列】
                                    //确保操作的【原子性】
}

void IapErase(unsigned int addr)

```

```

{
    EA = 1;                                //再开总中断,防止【IAP 触发命令序列】被任意中断打断
    IAP_CONTR = 0x80;                      //使能IAP
    IAP_TPS = 11;                           //设置等待参数 11MHz
    IAP_CMD = 3;                            //设置IAP 擦除命令
    IAP_ADDRL = addr;                      //设置IAP 低地址
    IAP_ADDRH = addr >> 8;                 //设置IAP 高地址
    IAP_TRIG = 0x5a;                        //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                        //写触发命令(0xa5)
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    IapIdle();                            //关闭IAP 功能
    EA = 0;                                //关闭总中断
                                            //而这个任意中断服务程序中又有 【IAP 触发命令序列】
                                            //确保操作的 【原子性】
}

void main()
{
    P_SW2 |= 0x80;                         //使能访问XFR · 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    IapErase(0x0400);
    P0 = IapRead(0x0400);                  //P0=0xff
    IapProgram(0x0400, 0x12);
    P1 = IapRead(0x0400);                  //P1=0x12

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
IAP_DATA	DATA	0C2H
IAP_ADDRH	DATA	0C3H
IAP_ADDRL	DATA	0C4H
IAP_CMD	DATA	0C5H
IAP_TRIG	DATA	0C6H
IAP CONTR	DATA	0C7H
IAP_TPS	DATA	0F5H
IAP_OFFSET EQU	4000H	;STC8H1K16

<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>

<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>
<i>ORG</i>	<i>0100H</i>

IAP_IDLE:

<i>MOV</i>	<i>IAP_CONTR,#0</i>	;关闭 IAP 功能
<i>MOV</i>	<i>IAP_CMD,#0</i>	;清除命令寄存器
<i>MOV</i>	<i>IAP_TRIG,#0</i>	;清除触发寄存器
<i>MOV</i>	<i>IAP_ADDRH,#80H</i>	;将地址设置到非 IAP 区域
<i>MOV</i>	<i>IAP_ADDRL,#0</i>	
<i>RET</i>		

IAP_READ:

<i>MOV</i>	<i>A,#LOW IAP_OFFSET</i>	;使用 MOVC 读取 EEPROM 需要加上相应的偏移
<i>ADD</i>	<i>A,DPL</i>	
<i>MOV</i>	<i>DPL,A</i>	
<i>MOV</i>	<i>A,@HIGH IAP_OFFSET</i>	
<i>ADDC</i>	<i>A,DPH</i>	
<i>MOV</i>	<i>DPH,A</i>	
<i>CLR</i>	<i>A</i>	
<i>MOVC</i>	<i>A,@A+DPTR</i>	;使用 MOVC 读取数据
<i>RET</i>		

IAP_PROGRAM:

<i>CLR</i>	<i>EA</i>	;关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	;使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	;设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#2</i>	;设置 IAP 写命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	;设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	;设置 IAP 高地址
<i>MOV</i>	<i>IAP_DATA,A</i>	;写 IAP 数据
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	;写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	;写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	;关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	;再开总中断,防止【IAP 触发命令序列】被任意中断打断 ;而这个任意中断服务程序中又有【IAP 触发命令序列】 ;确保操作的【原子性】
<i>RET</i>		

IAP_ERASE:

<i>CLR</i>	<i>EA</i>	;关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	;使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	;设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#3</i>	;设置 IAP 擦除命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	;设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	;设置 IAP 高地址
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	;写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	;写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	;关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	;再开总中断,防止【IAP 触发命令序列】被任意中断打断 ;而这个任意中断服务程序中又有【IAP 触发命令序列】 ;确保操作的【原子性】
<i>RET</i>		
START:		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>DPTR,#0400H</i>	
<i>LCALL</i>	<i>IAP_ERASE</i>	
<i>MOV</i>	<i>DPTR,#0400H</i>	
<i>LCALL</i>	<i>IAP_READ</i>	
<i>MOV</i>	<i>P0,A</i>	;P0=0FFH
<i>MOV</i>	<i>DPTR,#0400H</i>	
<i>MOV</i>	<i>A,#12H</i>	
<i>LCALL</i>	<i>IAP_PROGRAM</i>	
<i>MOV</i>	<i>DPTR,#0400H</i>	
<i>LCALL</i>	<i>IAP_READ</i>	
<i>MOV</i>	<i>P1,A</i>	;P1=12H
<i>SJMP</i>	\$	
<i>END</i>		

20.4.3 使用串口送出 EEPROM 数据

C 语言代码

//测试工作频率为11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

void UartInit()
{
    SCON = 0x5a;
    T2L = BRT;
    T2H = BRT >> 8;
    AUXR = 0x15;
}

void UartSend(char dat)
{
    while (!TI);
    TI = 0;
    SBUF = dat;
}

void IapIdle()
{
    IAP_CONTR = 0;                                //关闭IAP 功能
    IAP_CMD = 0;                                  //清除命令寄存器
    IAP_TRIG = 0;                                 //清除触发寄存器
    IAP_ADDRH = 0x80;                            //将地址设置到非IAP 区域
    IAP_ADDRL = 0;
}

char IapRead(int addr)
{
    char dat;

    EA = 0;                                      //关闭总中断
    IAP_CONTR = 0x80;                            //使能IAP
    IAP_TPS = 11;                                //设置等待参数 11MHz
    IAP_CMD = 1;                                 //设置IAP 读命令
    IAP_ADDRL = addr;                            //设置IAP 低地址
    IAP_ADDRH = addr >> 8;                      //设置IAP 高地址
    IAP_TRIG = 0x5a;                            //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                            //写触发命令(0xa5)

    _nop_();
    _nop_();
    _nop_();
    _nop_();

    dat = IAP_DATA;                            //读IAP 数据
    IapIdle();                                //关闭IAP 功能
    EA = 1;                                    //再开总中断,防止【IAP 触发命令序列】被任意中断打断
                                                //而这个任意中断服务程序中又有 【IAP 触发命令序列】
                                                //确保操作的 【原子性】

    return dat;
}

void IapProgram(int addr, char dat)
{
```

```

EA = 0;                                //关闭总中断
IAP_CONTR = 0x80;                      //使能IAP
IAP_TPS = 11;                           //设置等待参数 11MHz
IAP_CMD = 2;                            //设置IAP 写命令
IAP_ADDRL = addr;                      //设置IAP 低地址
IAP_ADDRH = addr >> 8;                 //设置IAP 高地址
IAP_DATA = dat;                         //写IAP 数据
IAP_TRIG = 0x5a;                        //写触发命令(0x5a)
IAP_TRIG = 0xa5;                        //写触发命令(0xa5)
_nop_();
_nop_();
_nop_();
_nop_();
IapIdle();                             //关闭IAP 功能
EA = I;                                 //再开总中断,防止【IAP 触发命令序列】被任意中断打断
                                         //而这个任意中断服务程序中又有【IAP 触发命令序列】
                                         //确保操作的【原子性】
}

void IapErase(int addr)
{
    EA = 0;                                //关闭总中断
    IAP_CONTR = 0x80;                      //使能IAP
    IAP_TPS = 11;                           //设置等待参数 11MHz
    IAP_CMD = 3;                            //设置IAP 擦除命令
    IAP_ADDRL = addr;                      //设置IAP 低地址
    IAP_ADDRH = addr >> 8;                 //设置IAP 高地址
    IAP_TRIG = 0x5a;                        //写触发命令(0x5a)
    IAP_TRIG = 0xa5;                        //写触发命令(0xa5)
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    IapIdle();                             //关闭IAP 功能
    EA = I;                                 //再开总中断,防止【IAP 触发命令序列】被任意中断打断
                                         //而这个任意中断服务程序中又有【IAP 触发命令序列】
                                         //确保操作的【原子性】
}

void main()
{
    P_SW2 |= 0x80;                         //使能访问XFR · 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    UartInit();
    IapErase(0x0400);
    UartSend(IapRead(0x0400));
}

```

```

    IapProgram(0x0400, 0x12);
    UartSend(IapRead(0x0400));

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>AUXR</i>	<i>DATA</i>	<i>8EH</i>
<i>T2H</i>	<i>DATA</i>	<i>0D6H</i>
<i>T2L</i>	<i>DATA</i>	<i>0D7H</i>
<i>IAP_DATA</i>	<i>DATA</i>	<i>0C2H</i>
<i>IAP_ADDRH</i>	<i>DATA</i>	<i>0C3H</i>
<i>IAP_ADDRL</i>	<i>DATA</i>	<i>0C4H</i>
<i>IAP_CMD</i>	<i>DATA</i>	<i>0C5H</i>
<i>IAP_TRIG</i>	<i>DATA</i>	<i>0C6H</i>
<i>IAP_CONTR</i>	<i>DATA</i>	<i>0C7H</i>
<i>IAP_TPS</i>	<i>DATA</i>	<i>0F5H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>DATA</i>	<i>0000H</i>
<i>LJMP</i>	<i>DATA</i>	<i>START</i>
<i>ORG</i>	<i>DATA</i>	<i>0100H</i>
<i>UART_INIT:</i>		
<i>MOV</i>	<i>DATA</i>	<i>SCON,#5AH</i>
<i>MOV</i>	<i>DATA</i>	<i>T2L,#0E8H</i>
<i>MOV</i>	<i>DATA</i>	<i>T2H,#0FFH</i>
<i>MOV</i>	<i>DATA</i>	<i>AUXR,#15H</i>
<i>RET</i>		
<i>UART_SEND:</i>		
<i>JNB</i>	<i>DATA</i>	<i>TI,\$</i>
<i>CLR</i>	<i>DATA</i>	<i>TI</i>
<i>MOV</i>	<i>DATA</i>	<i>SBUF,A</i>
<i>RET</i>		
<i>IAP_IDLE:</i>		
<i>MOV</i>	<i>DATA</i>	<i>IAP_CONTR,#0</i> ; 关闭 IAP 功能
<i>MOV</i>	<i>DATA</i>	<i>IAP_CMD,#0</i> ; 清除命令寄存器
<i>MOV</i>	<i>DATA</i>	<i>IAP_TRIG,#0</i> ; 清除触发寄存器

<i>MOV</i>	<i>IAP_ADDRH,#80H</i>	; 将地址设置到非 IAP 区域
<i>MOV</i>	<i>IAP_ADDRL,#0</i>	
<i>RET</i>		

IAP_READ:

<i>CLR</i>	<i>EA</i>	; 关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	; 使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	; 设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#1</i>	; 设置 IAP 读命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	; 设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	; 设置 IAP 高地址
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	; 写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	; 写触发命令(0xa5)
<i>NOP</i>		
<i>MOV</i>	<i>A,IAP_DATA</i>	; 读取 IAP 数据
<i>LCALL</i>	<i>IAP_IDLE</i>	; 关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	; 再开总中断, 防止【IAP 触发命令序列】被任意中断打断 ; 而这个任意中断服务程序中又有【IAP 触发命令序列】 ; 确保操作的【原子性】
<i>RET</i>		

IAP_PROGRAM:

<i>CLR</i>	<i>EA</i>	; 关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	; 使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	; 设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#2</i>	; 设置 IAP 写命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	; 设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	; 设置 IAP 高地址
<i>MOV</i>	<i>IAP_DATA,A</i>	; 写 IAP 数据
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	; 写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	; 写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	; 关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	; 再开总中断, 防止【IAP 触发命令序列】被任意中断打断 ; 而这个任意中断服务程序中又有【IAP 触发命令序列】 ; 确保操作的【原子性】
<i>RET</i>		

IAP_ERASE:

<i>CLR</i>	<i>EA</i>	; 关闭总中断
<i>MOV</i>	<i>IAP_CONTR,#80H</i>	; 使能 IAP
<i>MOV</i>	<i>IAP_TPS,#12</i>	; 设置等待参数 12MHz
<i>MOV</i>	<i>IAP_CMD,#3</i>	; 设置 IAP 擦除命令
<i>MOV</i>	<i>IAP_ADDRL,DPL</i>	; 设置 IAP 低地址
<i>MOV</i>	<i>IAP_ADDRH,DPH</i>	; 设置 IAP 高地址
<i>MOV</i>	<i>IAP_TRIG,#5AH</i>	; 写触发命令(0x5a)
<i>MOV</i>	<i>IAP_TRIG,#0A5H</i>	; 写触发命令(0xa5)
<i>NOP</i>		
<i>LCALL</i>	<i>IAP_IDLE</i>	; 关闭 IAP 功能
<i>SETB</i>	<i>EA</i>	; 再开总中断, 防止【IAP 触发命令序列】被任意中断打断

;而这个任意中断服务程序中又有 【IAP 触发命令序列】
;确保操作的 【原子性】

RET

START:

MOV	SP, #5FH	
ORL	P_SW2,#80H	;使能访问 XFR，没有冲突不用关闭
MOV	P0M0, #00H	
MOV	P0M1, #00H	
MOV	P1M0, #00H	
MOV	P1M1, #00H	
MOV	P2M0, #00H	
MOV	P2M1, #00H	
MOV	P3M0, #00H	
MOV	P3M1, #00H	
MOV	P4M0, #00H	
MOV	P4M1, #00H	
MOV	P5M0, #00H	
MOV	P5M1, #00H	
LCALL	UART_INIT	
MOV	DPTR,#0400H	
LCALL	IAP_ERASE	
MOV	DPTR,#0400H	
LCALL	IAP_READ	
LCALL	UART_SEND	
MOV	DPTR,#0400H	
MOV	A,#12H	
LCALL	IAP_PROGRAM	
MOV	DPTR,#0400H	
LCALL	IAP_READ	
LCALL	UART_SEND	
SJMP	\$	

END

20.4.4 串口 1 读写 EEPROM-带 MOVC 读

C 语言代码 (main.c)

//测试工作频率为 11.0592MHz

/* 本程序经过测试完全正常, 不提供电话技术支持, 如不能理解, 请自行补充相关基础。 */

***** 本程序功能说明 *****

STC8G 系列 EEPROM 通用测试程序。

请先别修改程序, 直接下载测试。下载时选择主频 11.0592MHZ。

PC 串口设置: 波特率 115200,8,n,1。

对 EEPROM 做扇区擦除、写入 64 字节、读出 64 字节的操作。

命令例子:

E 0 对 EEPROM 进行扇区擦除操作 · E 表示擦除, 数字 0 为 0 扇区(十进制, 0~126, 看具体 IC)。

W 0 对 EEPROM 进行写入操作 · W 表示写入, 数字 0 为 0 扇区(十进制, 0~126, 看具体 IC). 从扇区的开始地址连续写 64 字节。

R 0 对 EEPROM 进行 IAP 读出操作 R 表示读出, 数字 0 为 0 扇区(十进制, 0~126, 看具体 IC). 从扇区的开始地址连

续读 64 字节

M 0 对 EEPROM 进行 MOVC 读出操作(操作地址为扇区*512+偏移地址) 数字 0 为 0 扇区(十进制, 0~126, 看具体 IC). 从扇区的开始地址连续读 64 字节。

注意: 为了通用, 程序不识别扇区是否有效, 用户自己根据具体的型号来决定。

```
*****
#include "config.H"
#include "EEPROM.h"

#define Baudrate1          115200L
#define UART1_BUF_LENGTH   10
#define EEADDR_OFFSET       (8 * 1024)      //定义 EEPROM 用 MOVC 访问时加的偏移量,
                                         //等于 FLASH ROM 的大小对于 IAP 或 IRC 开头的,
                                         //则偏移量必须为 0
#define TimeOutSet1         5

***** 本地常量声明 *****/
u8 code T.Strings[]={"去年今日此门中，人面桃花相映红。人面不知何处去，桃花依旧笑春风。"};

***** 本地变量声明 *****/
u8 xdata tmp[70];
u8 xdata RX1_Buffer[UART1_BUF_LENGTH];
u8 RX1_Cnt;
u8 RX1_TimeOut;
bit B_RX1_Busy;

***** 本地函数声明 *****/
void UART1_config(void);
void TX1_write2buff(u8 dat);           //写入发送缓冲
void PrintString1(u8 *puts);          //发送一个字符串

***** 外部函数和变量声明 *****/
*****



u8 CheckData(u8 dat)
{
    if((dat >= '0') && (dat <= '9')) return (dat-'0');
    if((dat >= 'A') && (dat <= 'F')) return (dat-'A'+10);
    if((dat >= 'a') && (dat <= 'f')) return (dat-'a'+10);
    return 0xff;
}

u16 GetAddress(void)
{
    u16 address;
    u8 i;

    address = 0;
    if(RX1_Cnt < 3) return 65535;           //error
    if(RX1_Cnt <= 5)                         //5 个字节以内是扇区操作, 十进制
                                              //支持命令: E 0, E 12, E 120
                                              //          W 0, W 12, W 120
                                              //          R 0, R 12, R 120
    {
        for(i=2; i<RX1_Cnt; i++)

```

```

{
    if(CheckData(RX1_Buffer[i]) > 9)
        return 65535; //error
    address = address * 10 + CheckData(RX1_Buffer[i]);
}
if(address < 124) //限制在 0~123 扇区
{
    address <<= 9;
    return (address);
}
else if(RX1_Cnt == 8) //8 个字节直接地址操作 · 十六进制
//支持命令: E 0x1234, W 0x12b3, R 0xA00
{
    if((RX1_Buffer[2] == '0') && ((RX1_Buffer[3] == 'x') || (RX1_Buffer[3] == 'X')))
    {
        for(i=4; i<8; i++)
        {
            if(CheckData(RX1_Buffer[i]) > 0x0F)
                return 65535; //error
            address = (address << 4) + CheckData(RX1_Buffer[i]);
        }
        if(address < 63488)
            return (address); //限制在 0~123 扇区
    }
}
return 65535; //error
}

//=====================================================================
// 函数: void delay_ms(u8 ms)
// 描述: 延时函数。
// 参数: ms, 要延时的ms 数, 这里只支持 1~255ms. 自动适应主时钟。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void delay_ms(u8 ms)
{
    u16 i;
    do
    {
        i = MAIN_Fosc / 10000;
        while(--i) ;
    }while(--ms);
}

//使用 MOVC 读 EEPROM
void EEPROM_MOVC_read_n(u16 EE_address, u8 *DataAddress, u16 number)
{
    u8 code *pc;

    pc = EE_address + EEADDR_OFFSET;
    do
    {
        *DataAddress = *pc; //读出的数据
        DataAddress++;
        pc++;
    }
}

```

```

        }while(--number);
    }

/***** 主函数 *****/
void main(void)
{
    u8 i;
    u16 addr;

    UART1_config();                                // 选择波特率, 2: 使用 Timer2 做波特率,
                                                    // 其它值: 使用 Timer1 做波特率
    EA = 1;                                         //允许总中断

    PrintString1("STC8 系列MCU 用串口1 测试EEPROM 程序!\r\n"); //UART1 发送一个字符串

    while(1)
    {
        delay_ms(1);
        if(RXI_TimeOut > 0)                         //超时计数
        {
            if(--RXI_TimeOut == 0)
            {
                if(RXI_Buffer[1] == ' ')
                {
                    addr = GetAddress();
                    if(addr < 63488)                      //限制在0~123 扇区
                    {
                        if(RXI_Buffer[0] == 'E') //PC 请求擦除一个扇区
                        {
                            EEPROM_SectorErase(addr);
                            PrintString1("扇区擦除完成!\r\n");
                        }

                        else if(RXI_Buffer[0] == 'W')      //PC 请求写入 EEPROM 64 字节数据
                        {
                            EEPROM_write_n(addr,T_Strings,64);
                            PrintString1("写入操作完成!\r\n");
                        }

                        else if(RXI_Buffer[0] == 'R')      //PC 请求返回 64 字节 EEPROM 数据
                        {
                            PrintString1("IAP 读出的数据如下 :\r\n");
                            EEPROM_read_n(addr,tmp,64);
                            for(i=0; i<64; i++)
                                TX1_write2buff(tmp[i]); //将数据返回给串口
                            TX1_write2buff(0x0d);
                            TX1_write2buff(0x0a);
                        }

                        else if(RXI_Buffer[0] == 'M')      //PC 请求返回 64 字节 EEPROM 数据
                        {
                            PrintString1("MOVC 读出的数据如下 :\r\n");
                            EEPROM_MOVC_read_n(addr,tmp,64);
                            for(i=0; i<64; i++)
                                TX1_write2buff(tmp[i]); //将数据返回给串口
                            TX1_write2buff(0x0d);
                            TX1_write2buff(0x0a);
                        }

                        else PrintString1("命令错误!\r\n");
                    }
                }
            }
        }
    }
}

```

```

        else PrintStringI("命令错误!\r\n");
    }

    RXI_Cnt = 0;
}
}

} //*****发送一个字节*****/



void TXI_write2buff(u8 dat) //写入发送缓冲
{
    B_TXI_Busy = 1; //标志发送忙
    SBUF = dat; //发送一个字节
    while(B_TXI_Busy); //等待发送完毕
}

//=====

// 函数: void PrintStringI(u8 *puts)
// 描述: 串口1 发送字符串函数。
// 参数: puts: 字符串指针。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void PrintStringI(u8 *puts) //发送一个字符串
{
    for (; *puts != 0; puts++) //遇到停止符0 结束
    {
        TXI_write2buff(*puts);
    }
}

//=====

// 函数: void UART1_config(void)
// 描述: UART1 初始化函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void UART1_config(void)
{
    TR1 = 0;
    AUXR &= ~0x01; //SI BRT Use Timer1;
    AUXR |= (1<<6); //Timer1 set as 1T mode
    TMOD &= ~(1<<6); //Timer1 set As Timer
    TMOD &= ~0x30; //Timer1_16bitAutoReload;
    TH1 = (u8)((65536L-(MAIN_Fosc / 4) / Baudrate1) >> 8);
    TL1 = (u8)(65536L-(MAIN_Fosc / 4) / Baudrate1);
    ET1 = 0; // 禁止Timer1 中断
    INTCLKO &= ~0x02; // Timer1 不输出高速时钟
    TR1 = 1; // 运行Timer1

    S1_USE_P30P31(); P3n_standard(0x03); //切换到 P3.0 P3.1
    //S1_USE_P36P37(); P3n_standard(0xc0); //切换到 P3.6 P3.7
    //S1_USE_P16P17(); P1n_standard(0xc0); //切换到 P1.6 P1.7
}

```

```

    SCON = (SCON & 0x3f) / 0x00;           //UART1 模式, 0x00: 同步移位输出,
                                            //0x40: 8 位数据, 可变波特率,
                                            //0x80: 9 位数据, 固定波特率,
                                            //0xc0: 9 位数据, 可变波特率
// PS = 1;                                //高优先级中断
// ES = 1;                                 //允许中断
// REN = 1;                                //允许接收

    B_TX1_Busy = 0;
    RX1_Cnt = 0;
}

//=====================================================================
// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====================================================================
void UART1_int (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        if(RX1_Cnt >= UART1_BUF_LENGTH)
            RX1_Cnt = 0;                      //防溢出
        RX1_Buffer[RX1_Cnt++] = SBUF;
        RX1_TimeOut = TimeOutSet1;
    }

    if(TI)
    {
        TI = 0;
        B_TX1_Busy = 0;
    }
}

```

C 语言代码 (EEPROM.c)

```

//测试工作频率为11.0592MHz

// 本程序是STC 系列的内置EEPROM 读写程序。

#include "config.h"
#include "eeprom.h"

//=====================================================================
// 函数: void IAP_Disable(void)
// 描述: 禁止访问ISP/IAP.
// 参数: non.
// 返回: non.
// 版本: V1.0
//=====================================================================
void DisableEEPROM(void)
{
    IAP CONTR = 0;                         //禁止ISP/IAP 操作
    IAP TPS = 0;                           //去除ISP/IAP 命令
    IAP CMD = 0;
}

```

```

IAP_TRIG = 0;                                //防止ISP/IAP 命令误触发
IAP_ADDRH = 0xff;                            //清0 地址高字节
IAP_ADDRL = 0xff;                            //清0 地址低字节，指向非EEPROM 区，防止误操作
}

//=====
// 函数: void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 从指定EEPROM 首地址读出n 个字节放指定的缓冲。
// 参数: EE_address: 读出EEPROM 的首地址。
//        DataAddress: 读出数据放缓冲的首地址。
//        number:      读出的字节长度。
// 返回: non.
// 版本: V1.0
//=====

void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    EA = 0;                                    //禁止中断
    IAP_CONTR = IAP_EN;                      //允许ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L);    //工作频率设置
    IAP_READ();                                //送字节读命令，命令不需改变时，不需重新送命令
    do
    {
        IAP_ADDRH = EE_address / 256;          //送地址高字节 ( 地址需要改变时才需重新送地址 )
        IAP_ADDRL = EE_address % 256;          //送地址低字节
        IAP_TRIG();                            //先送5AH，再送A5H 到ISP/IAP 触发寄存器
                                                //每次都需要如此
                                                //送完A5H 后，ISP/IAP 命令立即被触发启动
                                                //CPU 等待IAP 完成后，才会继续执行程序。
        _nop_();
        _nop_();
        _nop_();
        *DataAddress = IAP_DATA;                //读出的数据送往
        EE_address++;
        DataAddress++;
    }while(--number);

    DisableEEPROM();
    EA = 1;                                    //重新允许中断
}

//*****扇区擦除函数 *****/
//=====

// 函数: void EEPROM_SectorErase(u16 EE_address)
// 描述: 把指定地址的EEPROM 扇区擦除。
// 参数: EE_address: 要擦除的扇区EEPROM 的地址。
// 返回: non.
// 版本: V1.0
//=====

void EEPROM_SectorErase(u16 EE_address)
{
    EA = 0;                                    //禁止中断
                                                //只有扇区擦除，没有字节擦除，512 字节/扇区。
                                                //扇区中任意一个字节地址都是扇区地址。
    IAP_ADDRH = EE_address / 256;              //送扇区地址高字节 ( 地址需要改变时才需重新送地址 )
    IAP_ADDRL = EE_address % 256;              //送扇区地址低字节
    IAP_CONTR = IAP_EN;                      //允许ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L);    //工作频率设置
    IAP_ERASE();                            //送扇区擦除命令，命令不需改变时，不需重新送命令
    IAP_TRIG();
```

```

_nop_();
_nop_();
_nop_();
DisableEEPROM();
EA = 1; //重新允许中断
}

//=====
// 函数: void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 把缓冲的n个字节写入指定首地址的EEPROM.
// 参数: EE_address: 写入EEPROM 的首地址.
//         DataAddress: 写入源数据的缓冲的首地址.
//         number:      写入的字节长度.
// 返回: non.
// 版本: V1.0
//=====
void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    EA = 0; //禁止中断

    IAP_CONTR = IAP_EN; //允许ISP/IAP 操作
    IAP_TPS = (u8)(MAIN_Fosc / 1000000L); //工作频率设置
    IAP_WRITE(); //送字节写命令，命令不需改变时，不需重新送命令
    do
    {
        IAP_ADDRH = EE_address / 256; //送地址高字节(地址需要改变时才需重新送地址)
        IAP_ADDRL = EE_address % 256; //送地址低字节
        IAP_DATA = *DataAddress; //送数据到IAP_DATA，只有数据改变时才需重新送
        IAP_TRIG(); //送命令
        _nop_();
        _nop_();
        _nop_();
        EE_address++;
        DataAddress++;
    }while(--number);

    DisableEEPROM();
    EA = 1; //重新允许中断
}

```

20.4.5 口令擦除写入-多扇区备份-串口 1 操作

C 语言代码 (main.c)

//测试工作频率为11.0592MHz

/* 本程序经过测试完全正常，不提供电话技术支持，如不能理解，请自行补充相关基础。 */

***** 本程序功能说明 *****

STC8G 系列 STC8H 系列 STC8C 系列 EEPROM 通用测试程序，演示多扇区备份，有扇区错误则用正确扇区数据写入，全部扇区错误(比如第一次运行程序)则写入默认值。

每次写都写入 3 个扇区，即冗余备份。

每个扇区写一条记录，写入完成后读出保存的数据和校验值跟源数据和校验值比较，并从串口 1(P3.0 P3.1)返回结果(正确或错误提示)。

每条记录自校验，64 字节数据，2 字节校验值，校验值 = 64 字节数据累加和 ^ 0x5555. ^0x5555 是为了保证写入的 66 个数据不全部为 0。

如果有扇区错误，则将正确扇区的数据写入错误扇区，如果 3 个扇区都错误，则均写入默认值。

擦除、写入、读出操作前均需要设置口令，如果口令不对则退出操作，并且每次退出操作都会清除口令。

下载时选择主频 11.0592MHZ.

PC 串口设置：波特率 115200,8,n,1.

对 EEPROM 做扇区擦除、写入 64 字节、读出 64 字节的操作。

命令例子：

使用串口助手发单个字符，大小写均可。

发 E 或 e： 对 EEPROM 进行扇区擦除操作，E 表示擦除，会擦除扇区 0、1、2。

发 W 或 w： 对 EEPROM 进行写入操作，W 表示写入，会写入扇区 0、1、2，每个扇区连续写 64 字节，扇区 0 写入 0x0000~0x003f，扇区 1 写入 0x0200~0x023f，扇区 0 写入 0x0400~0x043f。

发 R 或 r： 对 EEPROM 进行读出操作，R 表示读出，会读出扇区 0、1、2，每个扇区连续读 64 字节，扇区 0 读出 0x0000~0x003f，扇区 1 读出 0x0200~0x023f，扇区 0 读出 0x0400~0x043f。

注意：为了通用，程序不识别扇区是否有效，用户自己根据具体的型号来决定。

```
#include "config.H"
#include "EEPROM.h"

#define Baudrate1 115200L

*****本地常量声明 *****
u8 code T_StringD[]={"去年今日此门中，人面桃花相映红。人面不知何处去，桃花依旧笑春风。"};
u8 code T_StringW[]={"横看成岭侧成峰，远近高低各不同。不识庐山真面目，只缘身在此山中。"};
```

```
*****本地变量声明 *****
u8 xdatatmp[70]; //通用数据
u8 xdataSaveTmp[70]; //要写入的数组

bit B_TX1_Busy;
u8 cmd; //串口单字符命令
```

```
*****本地函数声明 *****
void UART1_config(void);
void TX1_write2buff(u8 dat); //写入发送缓冲
void PrintString1(u8 *puts); //发送一个字符串
```

***** 外部函数和变量声明 *****

```
***** 读取 EEPROM 记录，并且校验，返回校验结果，0 为正确，1 为错误 *****
u8 ReadRecord(u16 addr)
{
    u8 i;
    u16 ChckSum; //计算的累加和
    u16 j; //读取的累加和

    for(i=0; i<66; i++) tmp[i] = 0; //清除缓冲
    PassWord = D_PASSWORD; //给定口令
    EEPROM_read_n(addr,tmp,66); //读出扇区0
    for(ChckSum=0, i=0; i<64; i++)
        ChckSum += tmp[i];
}
```

```

        ChckSum += tmp[i];                                //计算累加和
        j = ((u16)tmp[64]<<8) + (u16)tmp[65];          //读取记录的累加和
        j ^= 0x5555;                                     //隔位取反, 避免全0
        if(ChckSum != j)      return 1;                  //累加和错误, 返回1
        return 0;                                         //累加和正确, 返回0
    }

/***** 写入 EEPROM 记录, 并且校验, 返回校验结果, 0 为正确, 1 为错误 *****/
u8 SaveRecord(u16 addr)
{
    u8   i;
    u16 ChckSum;                                       //计算的累加和

    for(ChckSum=0, i=0; i<64; i++)
        ChckSum += SaveTmp[i];                         //计算累加和
        ChckSum ^= 0x5555;                            //隔位取反, 避免全0
    SaveTmp[64] = (u8)(ChckSum >> 8);
    SaveTmp[65] = (u8)ChckSum;

    PassWord = D_PASSWORD;                           //给定口令
    EEPROM_SectorErase(addr);                      //擦除一个扇区
    PassWord = D_PASSWORD;                           //给定口令
    EEPROM_write_n(addr, SaveTmp, 66);              //写入扇区

    for(i=0; i<66; i++)
        tmp[i] = 0;                                 //清除缓冲
    PassWord = D_PASSWORD;                           //给定口令
    EEPROM_read_n(addr,tmp,66);                     //读出扇区0
    for(i=0; i<66; i++)                           //数据比较
    {
        if(SaveTmp[i] != tmp[i])                  //数据有错误, 返回1
            return 1;
    }
    return 0;                                         //累加和正确, 返回0
}

/***** 主函数 *****/
void main(void)
{
    u8   i;
    u8   status;                                       //状态

    UART1_config();                                    // 选择波特率 2: 使用 Timer2 做波特率,
    // 其它值: 使用 Timer1 做波特率.
    EA = 1;                                           //允许总中断

    PrintString1("STC8G-8H-8C 系列 MCU 用串口1 测试 EEPROM 程序!\r\n"); //UART1 发送一个字符串

    //上电读取3个扇区并校验, 如果有扇区错误则将正确的
    //扇区写入错误扇区, 如果3个扇区都错误, 则写入默认值
    status = 0;
    if(ReadRecord(0x0000) == 0)                       //读扇区0
    {
        status |= 0x01;                               //正确则标记status.0=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                    //保存在写缓冲
    }
    if(ReadRecord(0x0200) == 0)                       //读扇区1
    {

```

```

        status |= 0x02;                                //正确则标记status.1=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                      //保存在写缓冲
    }
    if(ReadRecord(0x0400) == 0)                      //读扇区2
    {
        status |= 0x04;                                //正确则标记status.2=1
        for(i=0; i<64; i++)
            SaveTmp[i] = tmp[i];                      //保存在写缓冲
    }

    if(status == 0)                                    //所有扇区都错误，则写入默认值
    {
        for(i=0; i<64; i++)
            SaveTmp[i] = T_StringD[i];                //读取默认值
    }
    else PrintStringI("上电读取3个扇区数据均正确!\r\n"); //UART1 发送一个字符串提示

    if((status & 0x01) == 0)                          //扇区0 错误，则写入默认值
    {
        if(SaveRecord(0x0000) == 0)
            PrintStringI("写入扇区0 正确!\r\n");        //写入记录0 扇区正确
        else
            PrintStringI("写入扇区0 错误!\r\n");        //写入记录0 扇区错误
    }
    if((status & 0x02) == 0)                          //扇区1 错误，则写入默认值
    {
        if(SaveRecord(0x0200) == 0)
            PrintStringI("写入扇区1 正确!\r\n");        //写入记录1 扇区正确
        else
            PrintStringI("写入扇区1 错误!\r\n");        //写入记录1 扇区错误
    }
    if((status & 0x04) == 0)                          //扇区2 错误，则写入默认值
    {
        if(SaveRecord(0x0400) == 0)
            PrintStringI("写入扇区2 正确!\r\n");        //写入记录2 扇区正确
        else
            PrintStringI("写入扇区2 错误!\r\n");        //写入记录2 扇区错误
    }

    while(1)
    {
        if(cmd != 0)                                  //有串口命令
        {
            if((cmd >= 'a') && (cmd <= 'z'))
                cmd -= 0x20;                            //小写转大写

            if(cmd == 'E')                            //PC 请求擦除一个扇区
            {
                PassWord = D_PASSWORD;                //给定口令
                EEPROM_SectorErase(0x0000);           //擦除一个扇区
                PassWord = D_PASSWORD;                //给定口令
                EEPROM_SectorErase(0x0200);           //擦除一个扇区
                PassWord = D_PASSWORD;                //给定口令
                EEPROM_SectorErase(0x0400);           //擦除一个扇区
                PrintStringI("扇区擦除完成!\r\n");
            }
        }
    }

```

```

        else if(cmd == 'W')                                //PC 请求写入 EEPROM 64 字节数据
        {
            for(i=0; i<64; i++)
                SaveTmp[i] = T_StringW[i];                  //写入数值
            if(SaveRecord(0x0000) == 0)
                PrintStringI("写入扇区0 正确|r\n");          //写入记录0 扇区正确
            else
                PrintStringI("写入扇区0 错误|r\n");          //写入记录0 扇区错误
            if(SaveRecord(0x0200) == 0)
                PrintStringI("写入扇区1 正确|r\n");          //写入记录1 扇区正确
            else
                PrintStringI("写入扇区1 错误|r\n");          //写入记录1 扇区错误
            if(SaveRecord(0x0400) == 0)
                PrintStringI("写入扇区2 正确|r\n");          //写入记录2 扇区正确
            else
                PrintStringI("写入扇区2 错误|r\n");          //写入记录2 扇区错误
        }

        else if(cmd == 'R')                                //PC 请求返回 64 字节 EEPROM 数据
        {
            if(ReadRecord(0x0000) == 0)                      //读出扇区0 的数据
            {
                PrintStringI("读出扇区0 的数据如下 :|r\n");
                for(i=0; i<64; i++)
                    TX1_write2buff(tmp[i]);                  //将数据返回给串口
                TX1_write2buff(0xd);                        //回车换行
                TX1_write2buff(0xa);
            }
            else PrintStringI("读出扇区0 的数据错误|r\n");

            if(ReadRecord(0x0200) == 0)                      //读出扇区1 的数据
            {
                PrintStringI("读出扇区1 的数据如下 :|r\n");
                for(i=0; i<64; i++)
                    TX1_write2buff(tmp[i]);                  //将数据返回给串口
                TX1_write2buff(0xd);                        //回车换行
                TX1_write2buff(0xa);
            }
            else PrintStringI("读出扇区1 的数据错误|r\n");

            if(ReadRecord(0x0400) == 0)                      //读出扇区2 的数据
            {
                PrintStringI("读出扇区2 的数据如下 :|r\n");
                for(i=0; i<64; i++)
                    TX1_write2buff(tmp[i]);                  //将数据返回给串口
                TX1_write2buff(0xd);                        //回车换行
                TX1_write2buff(0xa);
            }
            else PrintStringI("读出扇区2 的数据错误|r\n");
        }
        else PrintStringI("命令错误|r\n");
        cmd = 0;
    }
}
*******/

***** 发送一个字节 *****/
void TX1_write2buff(u8 dat)                         //写入发送缓冲

```

```

{
    B_TX1_Busy = 1;                                //标志发送忙
    SBUF = dat;                                     //发送一个字节
    while(B_TX1_Busy);                             //等待发送完毕
}

//=====
// 函数: void PrintString1(u8 *puts)
// 描述: 串口1 发送字符串函数。
// 参数: puts: 字符串指针。
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void PrintString1(u8 *puts)                         //发送一个字符串
{
    for (; *puts != 0; puts++)                      //遇到停止符0 结束
    {
        TX1_write2buff(*puts);
    }
}

//=====
// 函数: void    UART1_config(void)
// 描述: UART1 初始化函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====

void    UART1_config(void)
{
    TR1 = 0;
    AUXR &= ~0x01;                                 //SI BRT Use Timer1;
    AUXR |= (1<<6);                            //Timer1 set as IT mode
    TMOD &= ~(1<<6);                           //Timer1 set As Timer
    TMOD &= ~0x30;                               //Timer1_16bitAutoReload;
    TH1 = (u8)((65536L-(MAIN_Fosc / 4) / Baudrate1) >> 8);
    TL1 = (u8)(65536L-(MAIN_Fosc / 4) / Baudrate1);
    ET1 = 0;                                      // 禁止 Timer1 中断
    INTCLKO &= ~0x02;                            // Timer1 不输出高速时钟
    TR1   = 1;                                    // 运行 Timer1

    S1_USE_P30P31();    P3n_standard(0x03);      //切换到 P3.0 P3.1
    //S1_USE_P36P37(); P3n_standard(0xc0);      //切换到 P3.6 P3.7
    //S1_USE_P16P17(); P1n_standard(0xc0);      //切换到 P1.6 P1.7

    SCON = (SCON & 0x3f) / 0x40;                  //UART1 模式, 0x00: 同步移位输出,
                                                //          0x40: 8 位数据, 可变波特率,
                                                //          0x80: 9 位数据, 固定波特率,
                                                //          0xc0: 9 位数据, 可变波特率
    // PS   = 1;                                  //高优先级中断
    // ES   = 1;                                  //允许中断
    // REN  = 1;                                  //允许接收

    B_TX1_Busy = 0;
}
//=====

```

```

// 函数: void UART1_int (void) interrupt UART1_VECTOR
// 描述: UART1 中断函数。
// 参数: none.
// 返回: none.
// 版本: VER1.0
// 备注:
//=====
void UART1_int (void) interrupt 4
{
    if(RI)
    {
        RI = 0;
        cmd = SBUF;
    }

    if(TI)
    {
        TI = 0;
        B_TX1_Busy = 0;
    }
}

```

C 语言代码 (EEPROM.c)

```

// 测试工作频率为 11.0592MHz

// 本程序是 STC 系列的内置 EEPROM 读写程序。

#include "config.h"
#include "EEPROM.h"

u32      PassWord;                      // 擦除 写入时需要的口令

//=====
// 函数: void IAP_Disable(void)
// 描述: 禁止访问ISP/IAP.
// 参数: non.
// 返回: non.
// 版本: V1.0
//=====
void DisableEEPROM(void)
{
    IAP_CONTR = 0;                         // 禁止 ISP/IAP 操作
    IAP_TPS  = 0;                          // 去除 ISP/IAP 命令
    IAP_CMD   = 0;                         // 防止 ISP/IAP 命令误触发
    IAP_TRIG  = 0;                          // 清 0 地址高字节
    IAP_ADDRH = 0xff;                      // 清 0 地址低字节，指向非 EEPROM 区，防止误操作
    IAP_ADDRL = 0xff;
}

//=====
// 函数: void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 从指定 EEPROM 首地址读出 n 个字节放指定的缓冲。
// 参数: EE_address: 读出 EEPROM 的首地址。
//        DataAddress: 读出数据放缓冲的首地址。
//        number:       读出的字节长度。
// 返回: non.
// 版本: V1.0
//=====

```

```

void EEPROM_read_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    if(PassWord == D_PASSWORD)                                //口令正确才会操作EEPROM
    {
        EA = 0;                                              //禁止中断
        IAP_CONTR = IAP_EN;                                    //允许ISP/IAP 操作
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L);                //工作频率设置
        IAP_READ();                                           //送字节读命令，命令不需改变时，不需重新送命令
        do
        {
            IAP_ADDRH = EE_address / 256;                     //送地址高字节 ( 地址需要改变时才需重新送地址 )
            IAP_ADDRL = EE_address % 256;                      //送地址低字节
            if(PassWord == D_PASSWORD)                          //口令口令正确才触发操作
            {
                IAP_TRIG = 0x5A;                             //先送5AH，再送A5H 到ISP/IAP 触发寄存器，每次都需要如此
                IAP_TRIG = 0xA5;                            //送完A5H 后，ISP/IAP 命令立即被触发启动
                _nop_();                                     //CPU 等待IAP 完成后，才会继续执行程序。
                _nop_();
                _nop_();
                *DataAddress = IAP_DATA;                    //读出的数据送往
                EE_address++;                           //EE_address++;
                DataAddress++;                         //DataAddress++;
            }while(--number);

            DisableEEPROM();
            EA = 1;                                         //重新允许中断
        }
        PassWord = 0;                                       //清除口令
    }

/**************************************** 扇区擦除函数 *****/
//=====
// 函数: void EEPROM_SectorErase(u16 EE_address)
// 描述: 把指定地址的EEPROM 扇区擦除。
// 参数: EE_address: 要擦除的扇区EEPROM 的地址
// 返回: non.
// 版本: V1.0
//=====
void EEPROM_SectorErase(u16 EE_address)
{
    if(PassWord == D_PASSWORD)                                //口令正确才会操作EEPROM
    {
        EA = 0;                                              //禁止中断
        IAP_ADDRH = EE_address / 256;                        //只有扇区擦除，没有字节擦除，512 字节/扇区。
        IAP_ADDRL = EE_address % 256;                        //扇区中任意一个字节地址都是扇区地址。
        IAP_CONTR = IAP_EN;                                    //送扇区地址高字节 ( 地址需要改变时才需重新送地址 )
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L);                //送扇区地址低字节
        IAP_ERASE();                                         //允许ISP/IAP 操作
        IAP_ERASE();                                         //工作频率设置
        if(PassWord == D_PASSWORD)                          //送扇区擦除命令，命令不需改变时，不需重新送命令
        {
            IAP_TRIG = 0x5A;                             //口令口令正确才触发操作
            IAP_TRIG = 0xA5;                            //先送5AH，再送A5H 到ISP/IAP 触发寄存器，每次都需要如此
            IAP_TRIG = 0xA5;                            //送完A5H 后，ISP/IAP 命令立即被触发启动
            _nop_();                                     //CPU 等待IAP 完成后，才会继续执行程序。
        }
    }
}

```

```

    _nop_();
    _nop_();
    DisableEEPROM();
    EA = 1;                                //重新允许中断
}
PassWord = 0;                            //清除口令
}

//=====================================================================
// 函数: void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
// 描述: 把缓冲的n个字节写入指定首地址的EEPROM.
// 参数: EE_address: 写入EEPROM 的首地址.
//        DataAddress: 写入源数据的缓冲的首地址.
//        number:      写入的字节长度.
// 返回: non.
// 版本: V1.0
//=====================================================================
void EEPROM_write_n(u16 EE_address,u8 *DataAddress,u16 number)
{
    if(PassWord == D_PASSWORD)                //口令正确才会操作EEPROM
    {
        EA = 0;                            //禁止中断

        IAP_CONTR = IAP_EN;                  //允许ISP/IAP 操作
        IAP_TPS = (u8)(MAIN_Fosc / 1000000L); //工作频率设置
        IAP_WRITE();                        //送字节写命令，命令不需改变时，不需重新送命令
        do
        {
            IAP_ADDRH = EE_address / 256;   //送地址高字节 ( 地址需要改变时才需重新送地址 )
            IAP_ADDRL = EE_address % 256;   //送地址低字节
            IAP_DATA = *DataAddress;        //送数据到IAP_DATA，只有数据改变时才需重新送
            if(PassWord == D_PASSWORD)      //口令正确才触发操作
            {
                IAP_TRIG = 0x5A;           //先送5AH，再送A5H 到ISP/IAP 触发寄存器
                IAP_TRIG = 0xA5;           //每次都需要如此
                _nop_();                  //送完A5H 后，ISP/IAP 命令立即被触发启动
                _nop_();                  //CPU 等待IAP 完成后，才会继续执行程序。
                _nop_();
                _nop_();
                _nop_();
                EE_address++;             //地址加1
                DataAddress++;            //数据指针加1
            }while(--number);
        }
        DisableEEPROM();                    //重新允许中断
        EA = 1;                            //重新允许中断
    }
    PassWord = 0;                            //清除口令
}

```

21 ADC 模数转换，内部 1.19V 参考信号源 (BGV)

产品线	ADC 分辨率	ADC 通道数	ADCEXCFG
STC8H1K08 系列	10 位	9 通道	
STC8H1K28 系列	10 位	12 通道	
STC8H3K64S4 系列	12 位	12 通道	
STC8H3K64S2 系列	12 位	12 通道	
STC8H8K64U 系列 A 版本	12 位	15 通道	
STC8H8K64U 系列 B/C/D 版本	12 位	15 通道	●
STC8H4K64TL 系列	12 位	15 通道	●
STC8H4K64TLC 系列	12 位	15 通道	●
STC8H1K08T 系列	12 位	15 通道	●
STC8H2K12U-A/B 系列	12 位	15 通道	●
STC8H2K32U 系列	12 位	15 通道	●
STC8G1K08-SOP8 系列			
STC8G1K08A-SOP8 系列	10 位	6 通道	

STC8H 系列单片机内部集成了一个 10 位/12 位高速 A/D 转换器。ADC 的时钟频率为系统频率 2 分频再经过用户设置的分频系数进行再次分频 (ADC 的工作时钟频率范围为 SYSclk/2/1 到 SYSclk/2/16)。

STC8H 系列的 ADC 最快速度: **12 位 ADC 为 800K (每秒进行 80 万次 ADC 转换)** , **10 位 ADC 为 500K (每秒进行 50 万次 ADC 转换)**

ADC 转换结果的数据格式有两种: 左对齐和右对齐。可方便用户程序进行读取和引用。

注意: ADC 的第 15 通道是专门测量内部 1.19V 参考信号源的通道, 参考信号源值出厂时校准为 1.19V, 由于制造误差以及测量误差, 导致实际的内部参考信号源相比 1.19V, 大约有±1%的误差。如果用户需要知道每一颗芯片的准确内部参考信号源值, 可外接精准参考信号源, 然后利用 ADC 的第 15 通道进行测量标定。ADC_VRef+脚外接参考电源时, 可利用 ADC 的第 15 通道可以反推 ADC_VRef+ 脚外接参考电源的电压; 如将 ADC_VREF+短接到 MCU-VCC, 就可以反推 MCU-VCC 的电压。

如果芯片有 ADC 的外部参考电源管脚 ADC_VRef+, 则一定不能浮空, 必须接外部参考电源或者直接连到 VCC

21.1 ADC 相关的寄存器

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
ADC_CONTR	ADC 控制寄存器	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]				000x,0000	
ADC_RES	ADC 转换结果高位寄存器	BDH										0000,0000
ADC_RESL	ADC 转换结果低位寄存器	BEH										0000,0000
ADCCFG	ADC 配置寄存器	DEH	-	-	RESFMT	-	SPEED[3:0]				xx0x,0000	

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
ADCTIM	ADC 时序控制寄存器	FEA8H	CSSETUP	CSHOLD[1:0]		SMPDUTY[4:0]					0010,1010

21.1.1 ADC 控制寄存器 (ADC_CONTR) , PWM 触发 ADC 控制

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADC_CONTR	BCH	ADC_POWER	ADC_START	ADC_FLAG	ADC_EPWMT	ADC_CHS[3:0]			

ADC_POWER: ADC 电源控制位

0: 关闭 ADC 电源

1: 打开 ADC 电源。

建议进入空闲模式和掉电模式前将 ADC 电源关闭, 以降低功耗

特别注意:

1、给 MCU 的 内部 ADC 模块电源打开后, 需等待约 1ms, 等 MCU 内部的 ADC 电源稳定后再让 ADC 工作;

2、适当加长对外部信号的采样时间, 就是对 ADC 内部采样保持电容的充电或放电时间, 时间够, 内部才能和外部电势相等。

ADC_START: ADC 转换启动控制位。写入 1 后开始 ADC 转换, 转换完成后硬件自动将此位清零。

0: 无影响。即使 ADC 已经开始转换工作, 写 0 也不会停止 A/D 转换。

1: 开始 ADC 转换, 转换完成后硬件自动将此位清零。

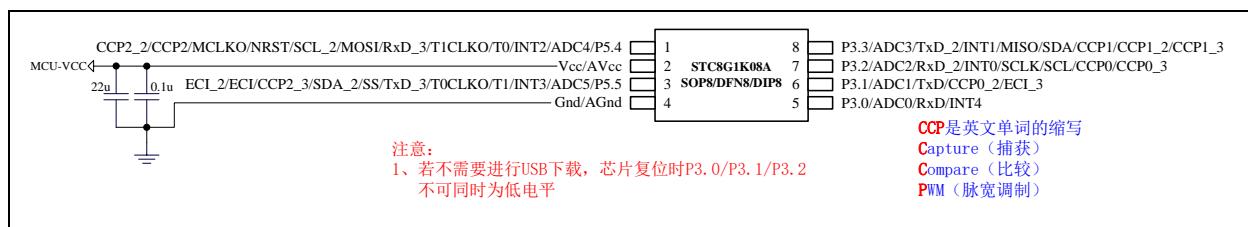
ADC_FLAG: ADC 转换结束标志位。当 ADC 完成一次转换后, 硬件会自动将此位置 1, 并向 CPU 提出中断请求。此标志位必须软件清零。

ADC_EPWMT: 使能 PWM 实时触发 ADC 功能。详情请参考 16 位高级 PWM 定时器章节

ADC_CHS[3:0]: ADC 模拟通道选择位

(注意: 被选择为 ADC 输入通道的 I/O 口, 必须设置 PxM0/PxM1 寄存器将 I/O 口模式设置为高阻输入模式。另外如果 MCU 进入掉电模式/主时钟停振/省电模式后, 仍需要使能 ADC 通道, 则需要设置 PxIE 寄存器关闭数字输入通道, 以防止外部模拟输入信号忽高忽低而产生额外的功耗)

ADC_CHS	ADC 通道	STC8H1K28 系列	STC8H1K08 系列	STC8H3K64S4 STC8H3K64S2 系列	STC8H8K64U STC8H4K64TL 系列	STC8H4K64TLCD 系列	STC8H1K08T STC8H2K12U STC8H2K32U 系列
0000	ADC0	P1.0	P1.0	P1.0	P1.0	P1.0	P1.0
0001	ADC1	P1.1	P1.1	P1.1	P1.1	P1.1	P1.1
0010	ADC2	P1.2	无此通道	P1.2	P5.4	P5.4	P5.4
0011	ADC3	P1.3	无此通道	无此通道	P1.3	P1.3	P1.3
0100	ADC4	P1.4	无此通道	无此通道	P1.4	P1.4	P1.4
0101	ADC5	P1.5	无此通道	无此通道	P1.5	P1.5	P1.5
0110	ADC6	P1.6	无此通道	P1.6	P1.6	P6.2	P1.6
0111	ADC7	P1.7	无此通道	P1.7	P1.7	P6.3	P1.7
1000	ADC8	P0.0	P3.0	P0.0	P0.0	P0.0	P3.0
1001	ADC9	P0.1	P3.1	P0.1	P0.1	P0.1	P3.1
1010	ADC10	P0.2	P3.2	P0.2	P0.2	P0.2	P3.2
1011	ADC11	P0.3	P3.3	P0.3	P0.3	P0.3	P3.3
1100	ADC12	无此通道	P3.4	P0.4	P0.4	P0.4	P3.4
1101	ADC13	无此通道	P3.5	P0.5	P0.5	P0.5	P3.5
1110	ADC14	无此通道	P3.6	P0.6	P0.6	P0.6	P3.6
1111	测试内部 1.19V	有	有	有	有	有	有



(STC8G1K08A 系列)

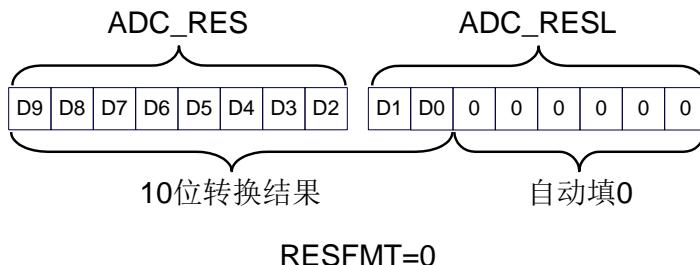
ADC_CHS[3:0]	ADC 通道
0000	P3.0/ADC0
0001	P3.1/ADC1
0010	P3.2/ADC2
0011	P3.3/ADC3
0100	P5.4/ADC4
0101	P5.5/ADC5
0110	无此通道
0111	无此通道
1000	无此通道
1001	无此通道
1010	无此通道
1011	无此通道
1100	无此通道
1101	无此通道
1110	无此通道
1111	测试内部 1.19V

21.1.2 ADC 配置寄存器 (ADCCFG)

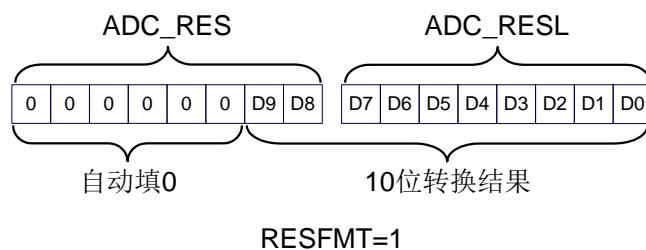
符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCCFG	DEH	-	-	RESFMT	-	SPEED[3:0]			

RESFMT: ADC 转换结果格式控制位 (**STC8H1K28 系列、STC8H1K08 系列**)

0: 转换结果左对齐。ADC_RES 保存结果的高 8 位, ADC_RESL 保存结果的低 2 位。格式如下:

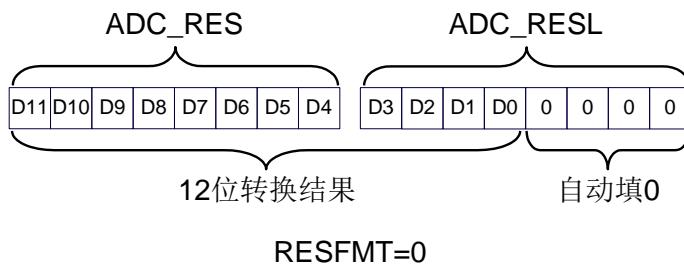


1: 转换结果右对齐。ADC_RES 保存结果的高 2 位, ADC_RESL 保存结果的低 8 位。格式如下:

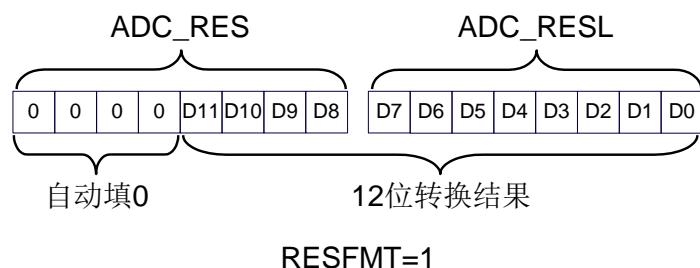


RESFMT: ADC 转换结果格式控制位 (**STC8H3K64S4 系列、STC8H3K64S2 系列、STC8H8K64U 系列、STC8H4K64TL 系列、STC8H4K64LCD 系列、STC8H1K08T 系列、STC8H2K12U 系列、STC8H2K32U 系列**)

0: 转换结果左对齐。ADC_RES 保存结果的高 8 位, ADC_RESL 保存结果的低 4 位。格式如下:



1: 转换结果右对齐。ADC_RES 保存结果的高 4 位, ADC_RESL 保存结果的低 8 位。格式如下:



SPEED[3:0]: 设置 ADC 工作时钟频率 { $F_{ADC} = SYSclk/2/(SPEED+1)$ }

SPEED[3:0]	给 ADC 的工作时钟频率
0000	SYSclk/2/1
0001	SYSclk/2/2
0010	SYSclk/2/3
...	...
1101	SYSclk/2/14
1110	SYSclk/2/15
1111	SYSclk/2/16

21.1.3 ADC 转换结果寄存器 (ADC_RES, ADC_RESL)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADC_RES	BDH								
ADC_RESL	BEH								

当 A/D 转换完成后, 10 位/12 位的转换结果会自动保存到 ADC_RES 和 ADC_RESL 中。保存结果的数据格式请参考 ADC_CFG 寄存器中的 RESFMT 设置。

21.1.4 ADC 时序控制寄存器 (ADCTIM)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCTIM	FEA8H	CSSETUP		CSHOLD[1:0]		SMPDUTY[4:0]			

CSSETUP: ADC 通道选择时间控制 T_{setup}

CSSETUP	占用 ADC 工作时钟数
0	1 (默认值)
1	2

CSHOLD[1:0]: ADC 通道选择保持时间控制 T_{hold}

CSHOLD[1:0]	占用 ADC 工作时钟数
00	1
01	2 (默认值)
10	3
11	4

SMPDUTY[4:0]: ADC 模拟信号采样时间控制 T_{duty} (注意: SMPDUTY 一定不能设置小于 01010B)

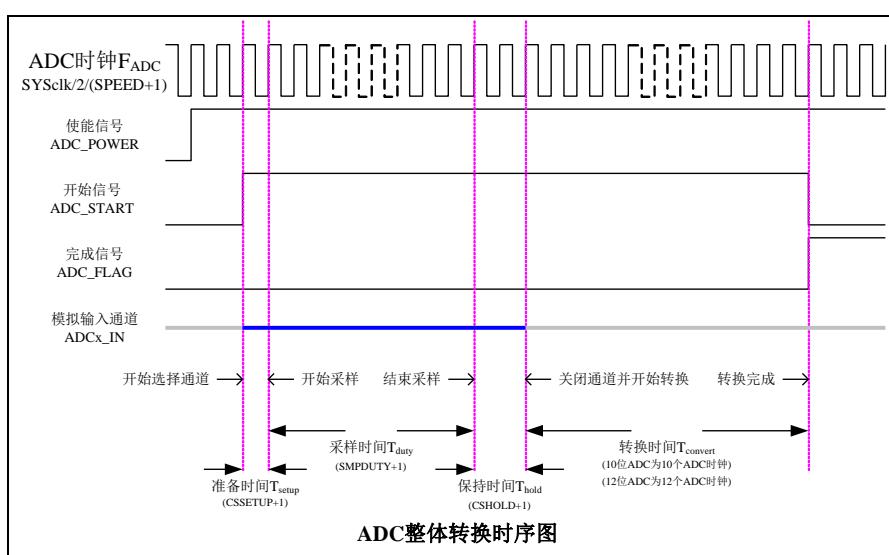
SMPDUTY[4:0]	占用 ADC 工作时钟数
00000	1
00001	2
...	...
01010	11 (默认值)
...	...
11110	31
11111	32

ADC 数模转换时间: T_{convert}

10 位 ADC 的转换时间为固定为 10 个 ADC 工作时钟

12 位 ADC 的转换时间为固定为 12 个 ADC 工作时钟

一个完整的 ADC 转换时间为: $T_{\text{setup}} + T_{\text{duty}} + T_{\text{hold}} + T_{\text{convert}}$, 如下图所示



21.1.5 非 DMA 模式下自动多次转换并取平均值

扩展配置寄存器 (ADCEXCFG)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
ADCEXCFG	FEADH	-	-	ADCETRS[1:0]	-	-	CVTIMESEL[2:0]	-	-

ADCETRS[1:0]: ADC 外部触发脚 ADC_ETR 控制位

ADCETRS[1:0]	ADC_ETR 设置
0x	禁止 ETR 功能
10	使能 ADC_ETR 的上升沿触发 ADC
11	使能 ADC_ETR 的下降沿触发 ADC

注: 使用此功能前, 必须打开 ADC_CONTR 中的 ADC 电源开关, 并设置好相应的 ADC 通道

CVTIMESEL[2:0]: ADC 自动转换次数选择

CVTIMESEL [2:0]	ADC 自动转换次数
0xx	转换 1 次
100	转换 2 次并取平均值
101	转换 4 次并取平均值
110	转换 8 次并取平均值
111	转换 16 次并取平均值

注:

- 1、当使能 ADC 自动转换多次功能后, ADC 中断标志只会在 ADC 自动转换到设置的次数后, 才会被置 1 (例如: 设置 CVTIMESEL 为 101B, 即 ADC 自动转换 4 次并取平均值, 则 ADC 中断标志位每完成 4 次 ADC 转换才会被置 1)
- 2、当 ADC 处于 DMA 模式下时, ADCEXCFG 设置的多次转换次数无效。
ADCEXCFG 设置的重复转换次数只有在非 DMA 模式下才有效。

21.2 ADC 相关计算公式

21.2.1 ADC 速度计算公式

ADC 的转换速度由 ADCCFG 寄存器中的 SPEED 和 ADCTIM 寄存器共同控制。转换速度的计算公式如下所示:

$$\text{10位ADC转换速度} = \frac{\text{MCU工作频率SYSclk}}{2 \times (\text{SPEED}[3:0] + 1) \times [(\text{CSSETUP} + 1) + (\text{CSHOLD} + 1) + (\text{SMPDUTY} + 1) + 10]}$$

$$\text{12位ADC转换速度} = \frac{\text{MCU工作频率SYSclk}}{2 \times (\text{SPEED}[3:0] + 1) \times [(\text{CSSETUP} + 1) + (\text{CSHOLD} + 1) + (\text{SMPDUTY} + 1) + 12]}$$

注意:

- 10 位 ADC 的速度不能高于 500KHz
- 12 位 ADC 的速度不能高于 800KHz
- SMPDUTY 的值不能小于 10, 建议设置为 15
- CSSETUP 可使用上电默认值 0
- CHOLD 可使用上电默认值 1 (ADCTIM 建议设置为 3FH)

21.2.2 ADC 转换结果计算公式

$$\text{10位ADC转换结果} = 1024 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{MCU工作电压} V_{cc}} \quad (\text{无独立ADC_Vref+管脚})$$

$$\text{10位ADC转换结果} = 1024 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{ADC外部参考源的电压}} \quad (\text{有独立ADC_Vref+管脚})$$

$$\text{12位ADC转换结果} = 4096 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{MCU工作电压} V_{cc}} \quad (\text{无独立ADC_Vref+管脚})$$

$$\text{12位ADC转换结果} = 4096 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{ADC外部参考源的电压}} \quad (\text{有独立ADC_Vref+管脚})$$

21.2.3 反推 ADC 输入电压计算公式

$$\text{ADC被转换通道的输入电压} V_{in} = \text{MCU工作电压} V_{cc} \times \frac{\text{10位ADC转换结果}}{1024} \quad (\text{无独立ADC_Vref+管脚})$$

$$\text{ADC被转换通道的输入电压} V_{in} = \text{ADC外部参考源的电压} \times \frac{\text{10位ADC转换结果}}{1024} \quad (\text{有独立ADC_Vref+管脚})$$

$$\text{ADC被转换通道的输入电压} V_{in} = \text{MCU工作电压} V_{cc} \times \frac{\text{12位ADC转换结果}}{4096} \quad (\text{无独立ADC_Vref+管脚})$$

$$\text{ADC被转换通道的输入电压} V_{in} = \text{ADC外部参考源的电压} \times \frac{\text{12位ADC转换结果}}{4096} \quad (\text{有独立ADC_Vref+管脚})$$

21.2.4 反推工作电压计算公式

当需要使用 ADC 输入电压和 ADC 转换结果反推工作电压时, 若目标芯片无独立的 ADC_Vref+管脚, 则可直接测量并使用下面公式, 若目标芯片有独立 ADC_Vref+管脚时, 则必须将 ADC_Vref+管脚连接到 Vcc 管脚。

$$\text{MCU工作电压} V_{cc} = 1024 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{10位ADC转换结果}}$$

$$\text{MCU工作电压} V_{cc} = 4096 \times \frac{\text{ADC被转换通道的输入电压} V_{in}}{\text{12位ADC转换结果}}$$

21.3 10 位 ADC 静态特性

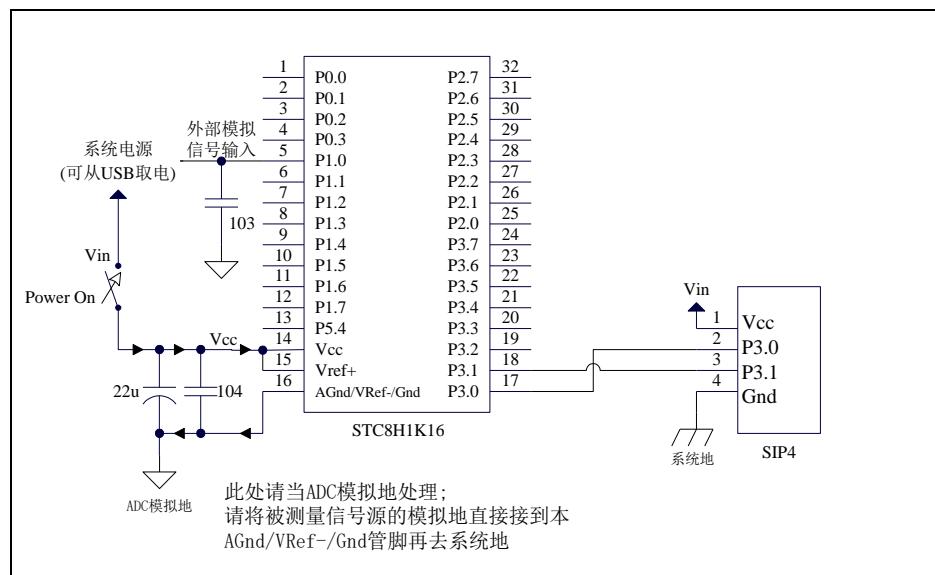
符号	描述	最小值	典型值	最大值	单位
RES	分辨率	-	10	-	Bits
E _T	整体误差	-	1.3	3	LSB
E _O	偏移误差	-	0.3	1	LSB
E _G	增益误差	-	0	1	LSB
E _D	微分非线性误差	-	0.7	1.5	LSB
E _I	积分非线性误差	-	1	2	LSB
R _{AIN}	通道等效电阻	-	∞	-	ohm
R _{ESD}	采样保持电容前串接的抗静电电阻	-	700	-	ohm
C _{ADC}	内部采样保持电容	-	16.5	-	pF

21.4 12 位 ADC 静态特性

符号	描述	最小值	典型值	最大值	单位
RES	分辨率	-	12	-	Bits
E _T	整体误差	-	0.5	1	LSB
E _O	偏移误差	-	-0.1	1	LSB
E _G	增益误差	-	0	1	LSB
E _D	微分非线性误差	-	0.7	1.5	LSB
E _I	积分非线性误差	-	1	2	LSB
R _{AIN}	通道等效电阻	-	∞	-	ohm
R _{ESD}	采样保持电容前串接的抗静电电阻	-	700	-	ohm
C _{ADC}	内部采样保持电容	-	16.5	-	pF

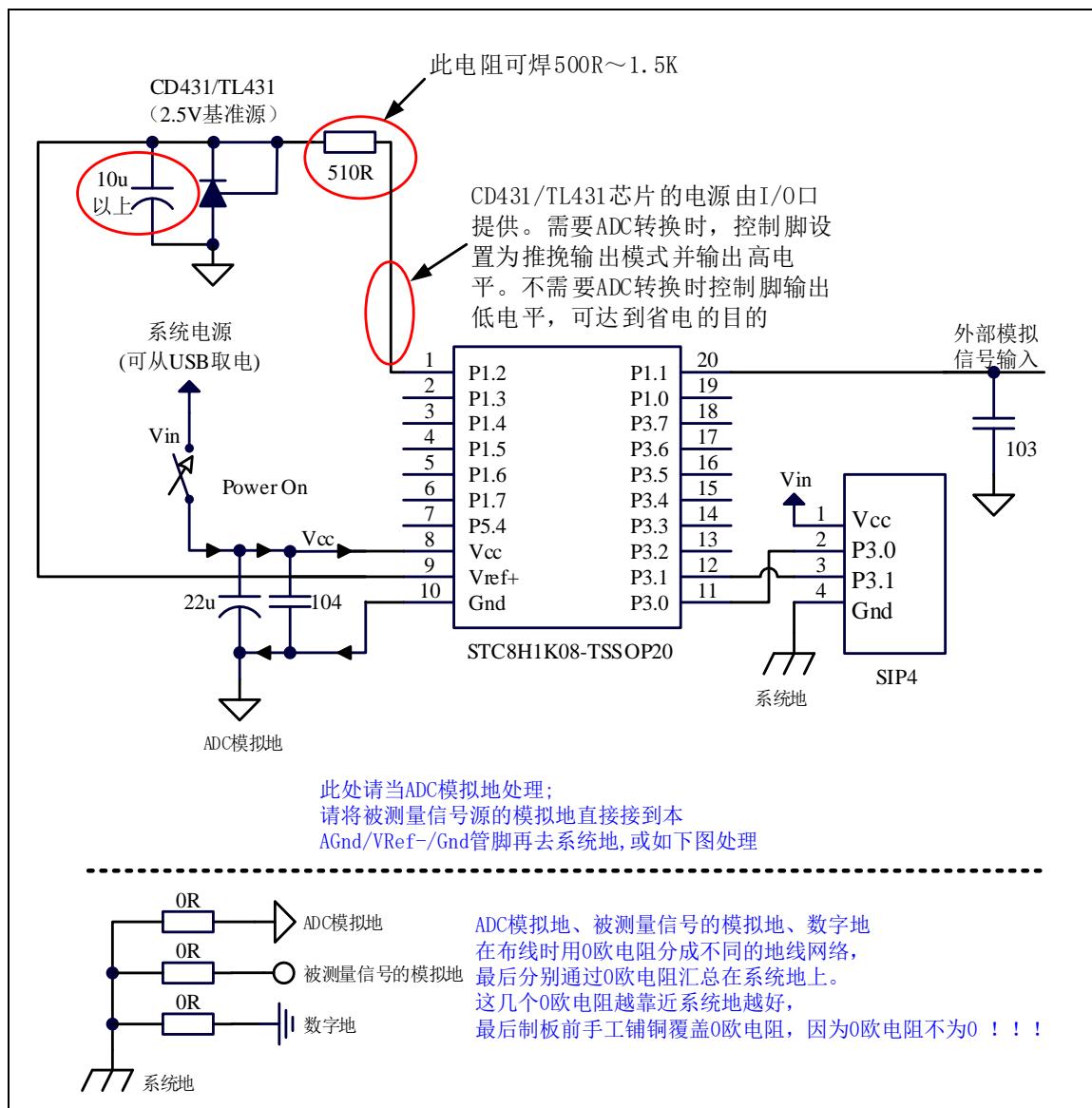
21.5 ADC 应用参考线路图

21.5.1 一般精度 ADC 参考线路图

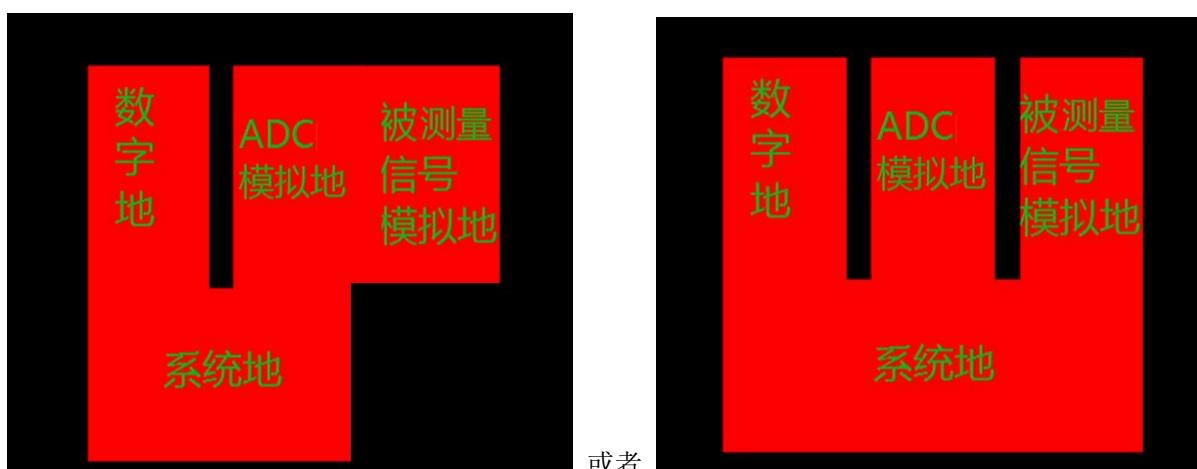


21.5.2 高精度 ADC 参考线路图

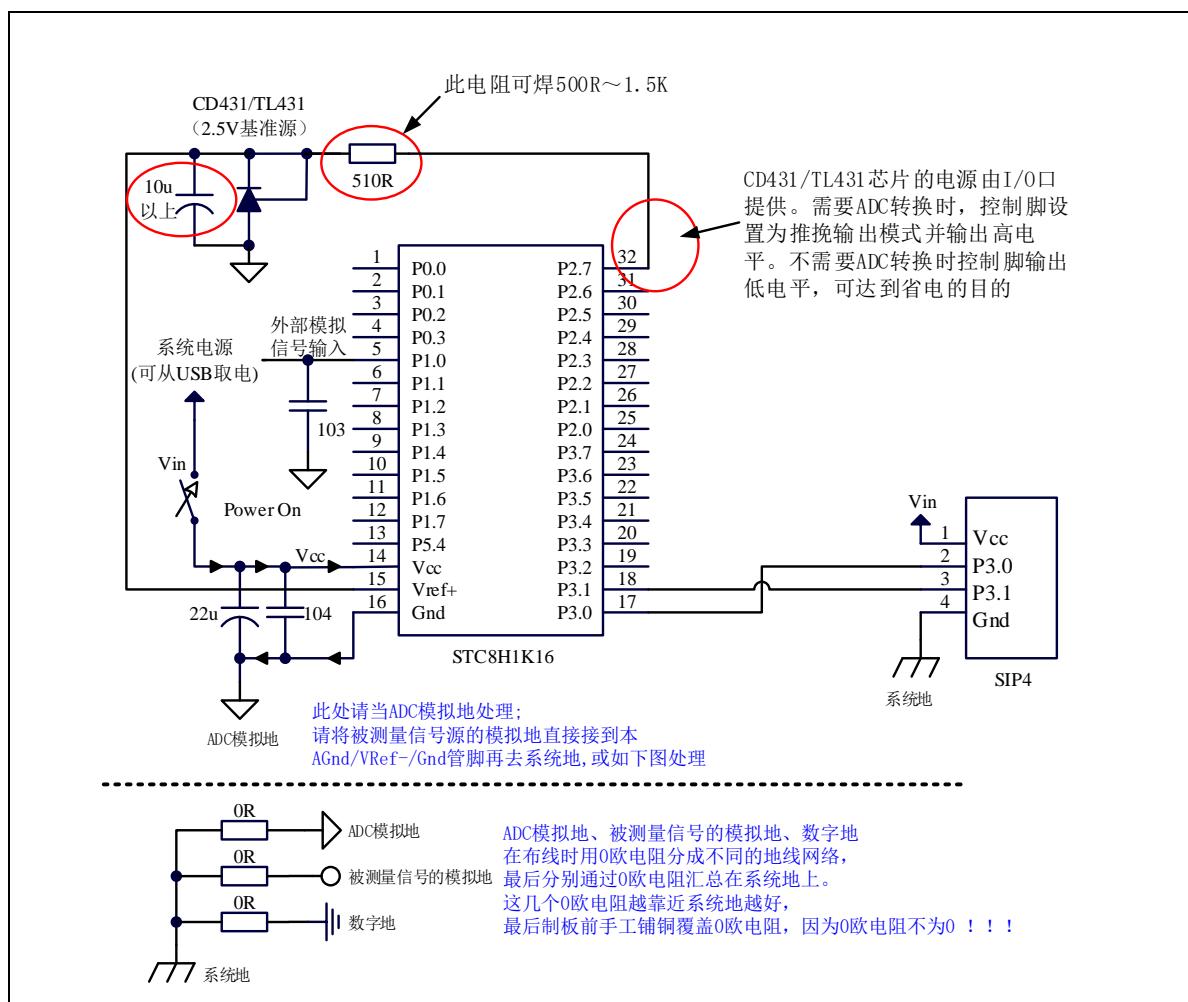
a) 高精度 ADC 参考线路图 -- STC8H1K08-TSSOP20



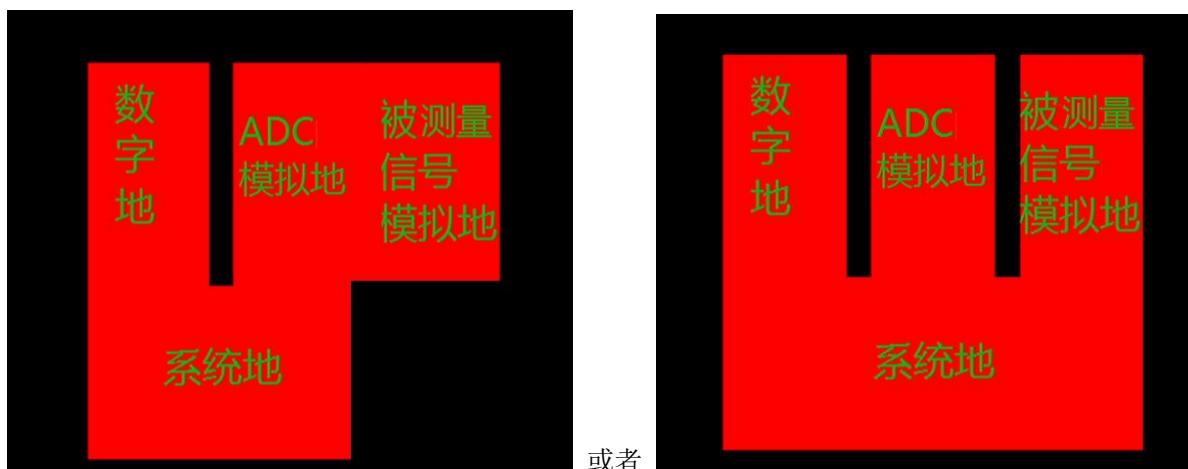
PCB 布线示意图



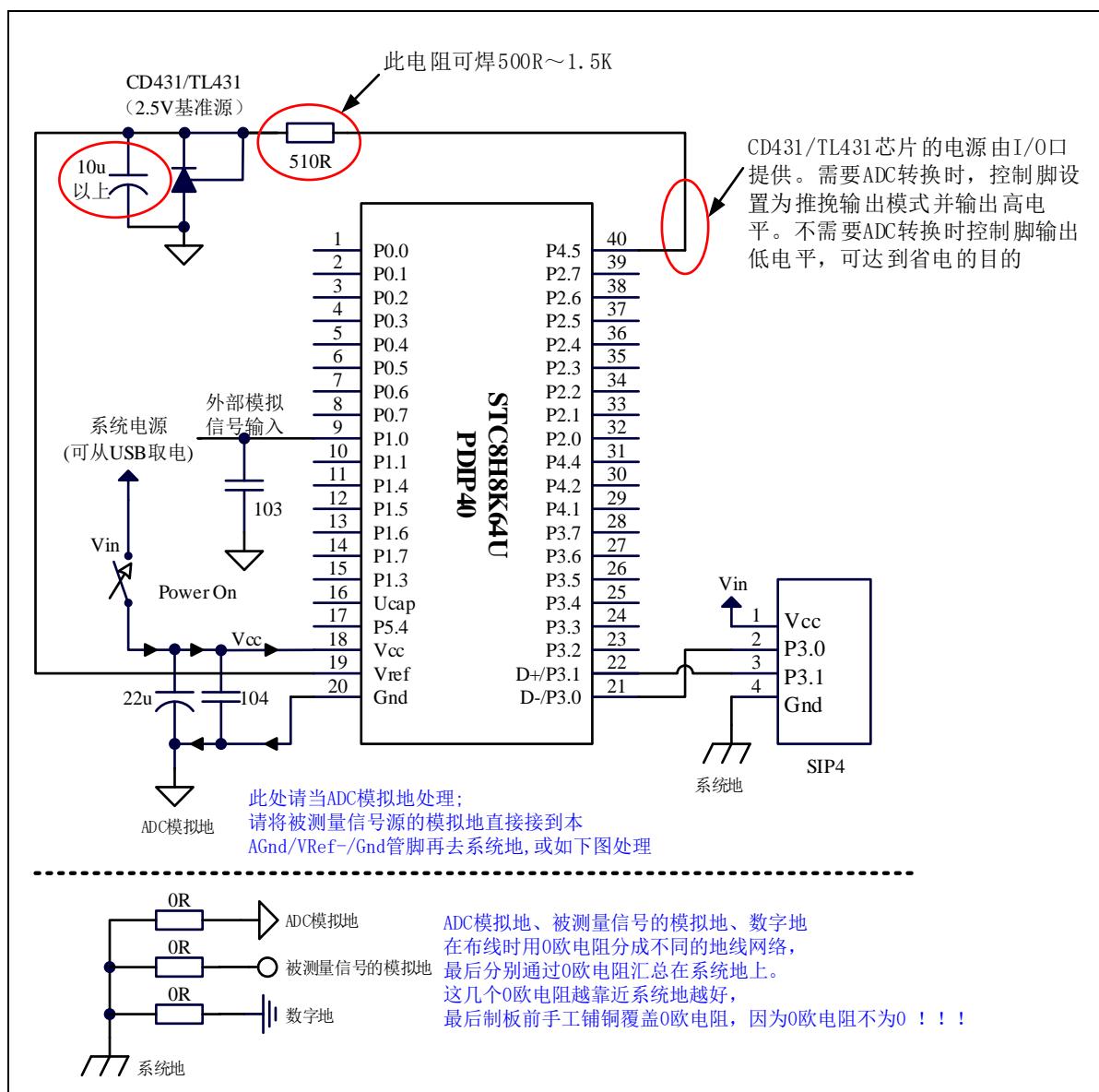
b) 高精度 ADC 参考线路图 -- STC8H1K16



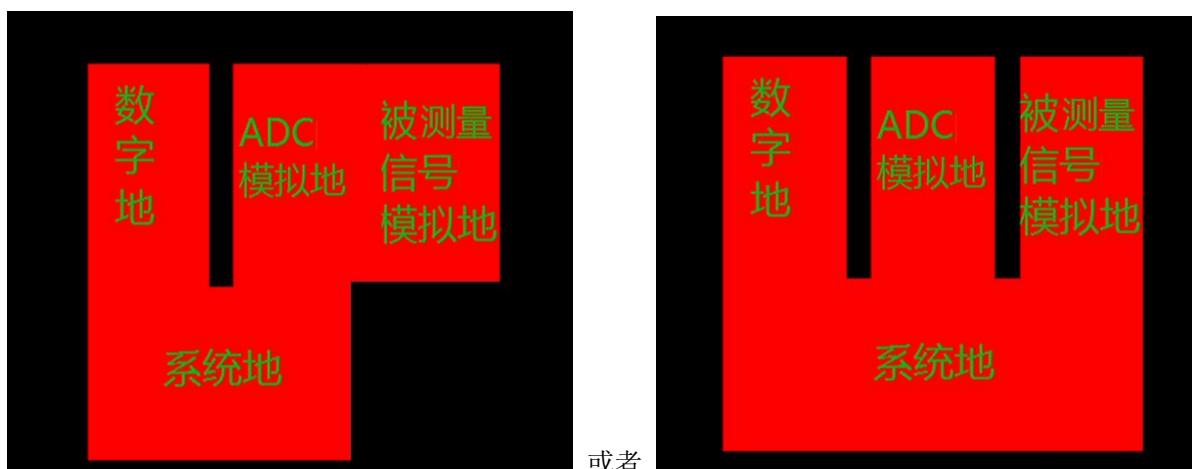
PCB 布线示意图



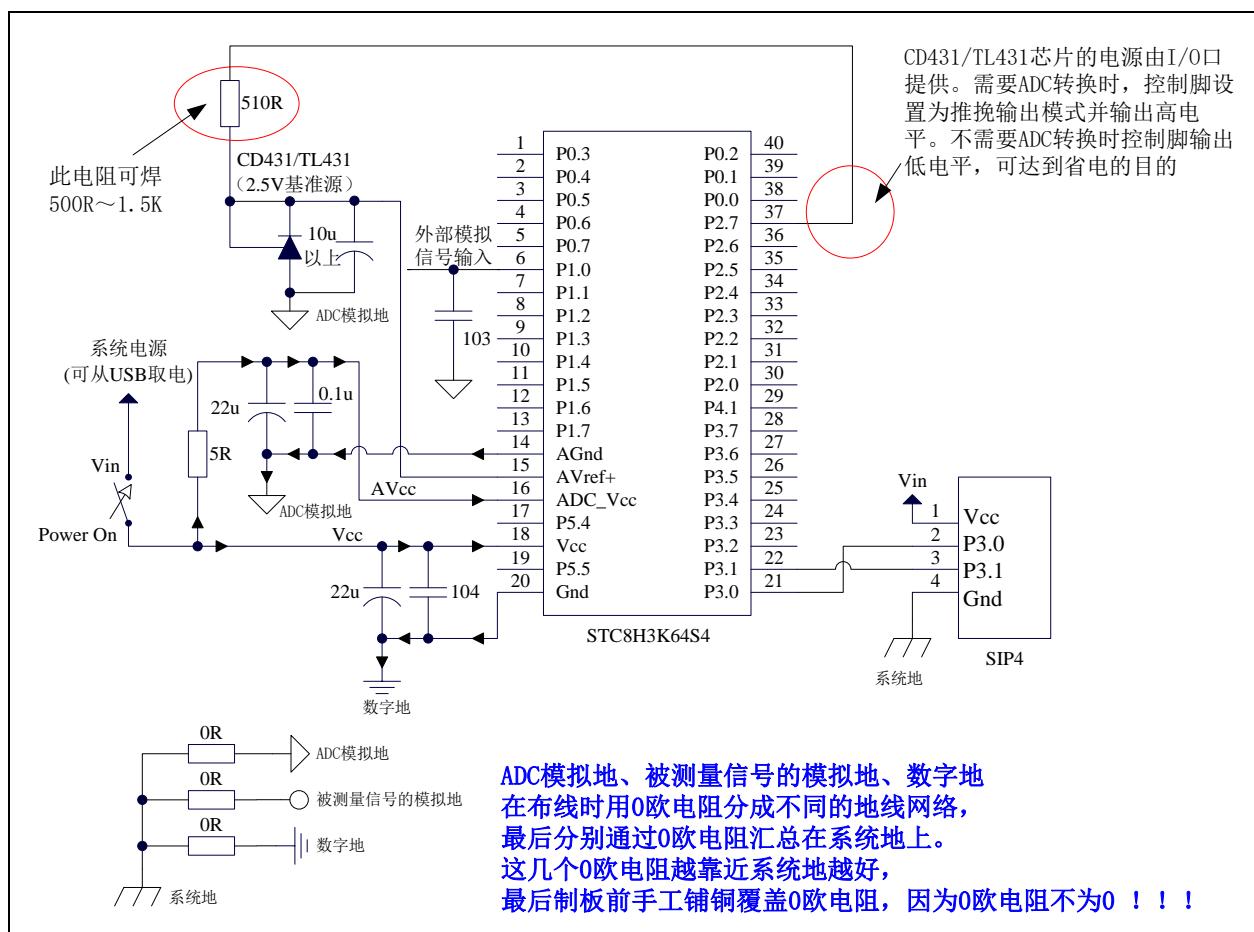
c) 高精度 ADC 参考线路图 -- STC8H8K64U-PDIP40



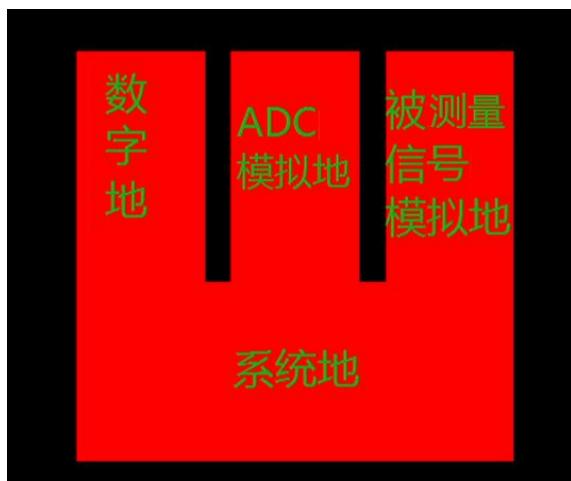
PCB 布线示意图



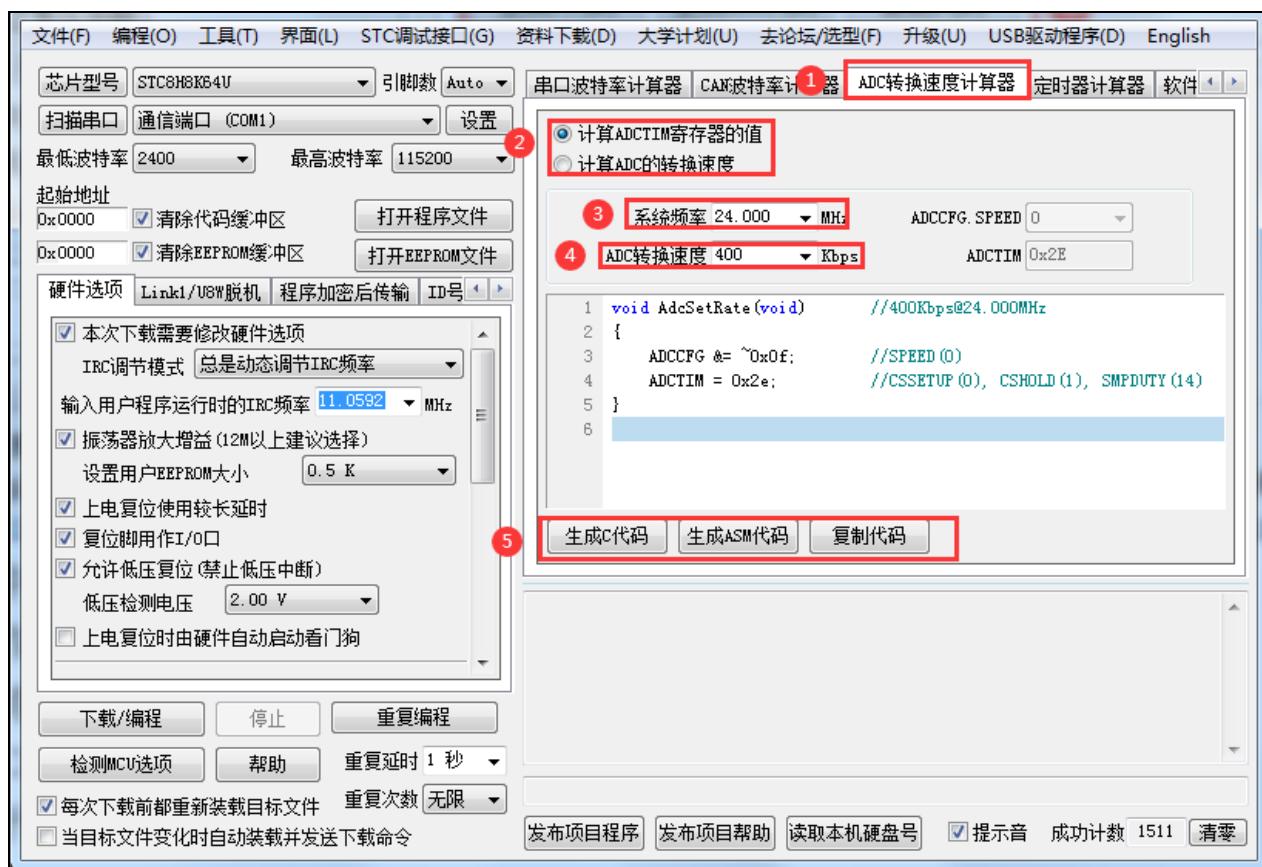
21.5.3 高精度 ADC 参考线路图 (有独立 AVcc 和 AGnd)



PCB 布线示意图



21.6 AiCube-ISP | ADC 转换速度计算器工具



- ①: 在下载软件中选择“ADC 转换速度计算器”功能页，进入 ADC 代码生成界面
- ②: 选择“根据速度计算配置寄存器功能”还是“根据寄存器配置反推转换速度”
- ③: 设置系统工作频率
- ④: 设置 ADC 转换速度
- ⑤: 手动生成 C 代码或者 ASM 代码，复制范例

21.7 范例程序

21.7.1 ADC 基本操作（查询方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    PIM0 = 0x00;                                  //设置 P1.0 为 ADC 口
    P1M1 = 0x01;

    ADCTIM = 0x3f;                                //设置 ADC 内部时序
    ADCCFG = 0x0f;                                //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;                            //使能 ADC 模块

    while (1)
    {
        ADC_CONTR |= 0x40;                          //启动 AD 转换
        _nop_();
        _nop_();
        while (!(ADC_CONTR & 0x20));            //查询 ADC 完成标志
        ADC_CONTR &= ~0x20;                      //清完成标志
        P2 = ADC_RES;                            //读取 ADC 结果
    }
}
```

汇编代码

```
; 测试工作频率为 11.0592MHz
```

ADC_CONTR	DATA	0BCH
ADC_RES	DATA	0BDH
ADC_RESL	DATA	0BEH
ADCCFG	DATA	0DEH
P_SW2	DATA	0BAH
ADCTIM	XDATA	0FEA8H

```

P1M1      DATA      091H
P1M0      DATA      092H
P0M1      DATA      093H
P0M0      DATA      094H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP    START

        ORG      0100H
START:
        MOV      SP, #5FH
        ORL      P_SW2,#80H      ;使能访问 XFR，没有冲突不用关闭

        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        MOV      P1M0,#00H      ;设置 P1.0 为 ADC 口
        MOV      P1M1,#01H
        MOV      DPTR,#ADCTIM      ;设置 ADC 内部时序
        MOV      A,#3FH
        MOVX    @DPTR,A
        MOV      ADCCFG,#0FH      ;设置 ADC 时钟为系统时钟/2/16
        MOV      ADC_CONTR,#80H      ;使能 ADC 模块

LOOP:
        ORL      ADC_CONTR,#40H      ;启动 AD 转换
        NOP
        NOP
        MOV      A,ADC_CONTR      ;查询 ADC 完成标志
        JNB      ACC.5,$-2
        ANL      ADC_CONTR,#NOT 20H      ;清完成标志
        MOV      P2,ADC_RES      ;读取 ADC 结果

        SJMP    LOOP

        END

```

21.7.2 ADC 基本操作（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

void ADC_Isr() interrupt 5
{
    ADC_CONTR &= ~0x20;                                //清中断标志
    P2 = ADC_RES;                                     //读取ADC 结果
    ADC_CONTR |= 0x40;                                //继续AD 转换
}

void main()
{
    P_SW2 |= 0x80;                                    //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    PIM0 = 0x00;                                      //设置 P1.0 为 ADC 口
    PIM1 = 0x01;

    ADCTIM = 0x3f;                                    //设置 ADC 内部时序
    ADCCFG = 0x0f;                                    //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;                                //使能 ADC 模块
    EADC = 1;                                         //使能 ADC 中断
    EA = 1;                                           //启动 AD 转换
}

while (1);
}
```

汇编代码

```
;测试工作频率为 11.0592MHz
```

ADC_CONTR	DATA	0BCH
ADC_RES	DATA	0BDH
ADC_RESL	DATA	0BEH
ADCCFG	DATA	0DEH
P_SW2	DATA	0BAH
ADCTIM	XDATA	0FEA8H

EADC	BIT	IE.5	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>002BH</i>	
	<i>LJMP</i>	<i>ADCISR</i>	
	<i>ORG</i>	<i>0100H</i>	
ADCISR:	<i>ANL</i>	<i>ADC_CONTR,#NOT 20H</i>	;清完成标志
	<i>MOV</i>	<i>P2,ADC_RES</i>	;读取 ADC 结果
	<i>ORL</i>	<i>ADC_CONTR,#40H</i>	;继续 AD 转换
	<i>RETI</i>		
START:	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>MOV</i>	<i>PIM0,#00H</i>	;设置 P1.0 为 ADC 口
	<i>MOV</i>	<i>PIM1,#01H</i>	
	<i>MOV</i>	<i>DPTR,#ADCTIM</i>	;设置 ADC 内部时序
	<i>MOV</i>	<i>A,#3FH</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>ADCCFG,#0FH</i>	;设置 ADC 时钟为系统时钟/2/16
	<i>MOV</i>	<i>ADC_CONTR,#80H</i>	;使能 ADC 模块
	<i>SETB</i>	<i>EADC</i>	;使能 ADC 中断
	<i>SETB</i>	<i>EA</i>	
	<i>ORL</i>	<i>ADC_CONTR,#40H</i>	;启动 AD 转换
	<i>SJMP</i>	\$	
	END		

21.7.3 格式化 ADC 转换结果

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    P1M0 = 0x00;                                  //设置 P1.0 为 ADC 口
    P1M1 = 0x01;
    ADCTIM = 0x3f;                                //设置 ADC 内部时序
    ADCCFG = 0x0f;                                //设置 ADC 时钟为系统时钟/2/16
    ADC_CONTR = 0x80;                            //使能 ADC 模块
    ADC_CONTR |= 0x40;                           //启动 AD 转换
    _nop_();
    _nop_();
    while (!(ADC_CONTR & 0x20));                //查询 ADC 完成标志
    ADC_CONTR &= ~0x20;                          //清完成标志

    ADCCFG = 0x00;                                //设置结果左对齐
    ACC = ADC_RES;                               //A 存储 ADC 的 10 位结果的高 8 位
    B = ADC_RESL;                                //B[7:6]存储 ADC 的 10 位结果的低 2 位,B[5:0]为 0

//    ADCCFG = 0x20;                                //设置结果右对齐
//    ACC = ADC_RES;                               //A[1:0]存储 ADC 的 10 位结果的高 2 位,A[7:2]为 0
//    B = ADC_RESL;                               //B 存储 ADC 的 10 位结果的低 8 位

    while (1);
}
```

汇编代码

;测试工作频率为 11.0592MHz

ADC CONTR	DATA	0BCH
ADC RES	DATA	0BDH
ADC RESL	DATA	0BEH
ADCCFG	DATA	0DEH

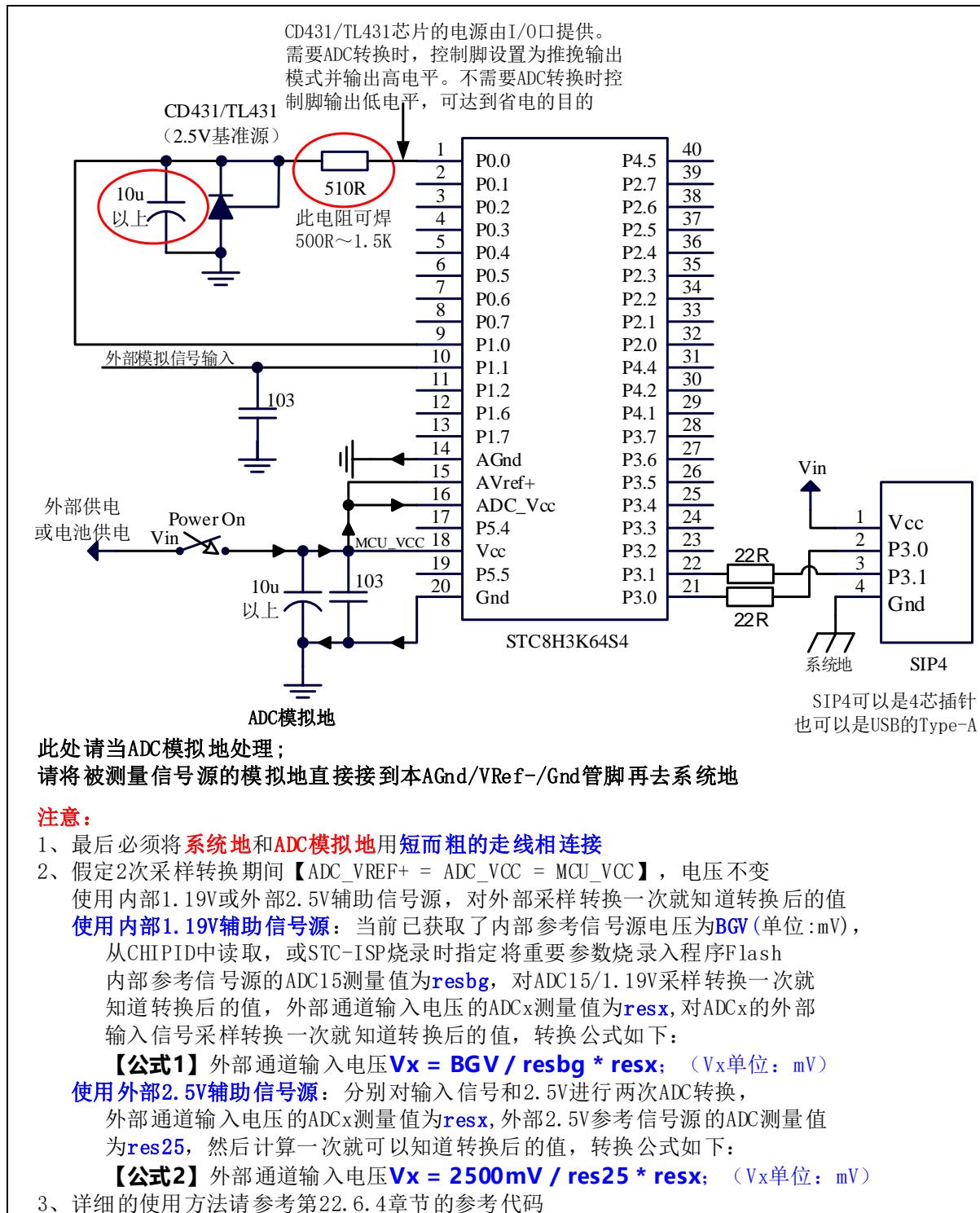
<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>	
<i>ADCTIM</i>	<i>XDATA</i>	<i>0FEA8H</i>	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>START:</i>			
	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>MOV</i>	<i>P1M0,#00H</i>	;设置 P1.0 为 ADC 口
	<i>MOV</i>	<i>P1M1,#01H</i>	
	<i>MOV</i>	<i>DPTR,#ADCTIM</i>	;设置 ADC 内部时序
	<i>MOV</i>	<i>A,#3FH</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>ADCCFG,#0FH</i>	;设置 ADC 时钟为系统时钟/2/16
	<i>MOV</i>	<i>ADC_CONTR,#80H</i>	;使能 ADC 模块
	<i>ORL</i>	<i>ADC_CONTR,#40H</i>	;启动 AD 转换
	<i>NOP</i>		
	<i>NOP</i>		
	<i>MOV</i>	<i>A,ADC_CONTR</i>	;查询 ADC 完成标志
	<i>JNB</i>	<i>ACC.5,\$-2</i>	
	<i>ANL</i>	<i>ADC_CONTR,#NOT 20H</i>	;清完成标志
	<i>MOV</i>	<i>ADCCFG,#00H</i>	;设置结果左对齐
	<i>MOV</i>	<i>A,ADC_RES</i>	;A 存储 ADC 的 10 位结果的高 8 位
	<i>MOV</i>	<i>B,ADC_RESL</i>	;B[7:6]存储 ADC 的 10 位结果的低 2 位,B[5:0]为0
;	<i>MOV</i>	<i>ADCCFG,#20H</i>	;设置结果右对齐
;	<i>MOV</i>	<i>A,ADC_RES</i>	;A[3:0]存储 ADC 的 10 位结果的高 2 位,A[7:2]为0

```
;           MOV      B,ADC_RESL      ;B 存储ADC 的10 位结果的低8 位
SJMP      $  
END
```

21.7.4 利用 ADC15 通道在内部固定接的 1.19V 辅助固定信号源，反推其他通道的输入电压或 VCC

STC8H 系列 ADC 的第 15 通道用于测量内部参考信号源，由于内部参考信号源很稳定，约为 1.19V，且不会随芯片的工作电压的改变而变化，所以可以通过测量内部 1.19V 参考信号源，然后通过 ADC 的值便可反推出外部电压或外部电池电压。

下图为参考线路图:



利用 ADC15 通道在内部固定接的 1.19V 辅助固定信号源！

反推其他 ADCx 通道的外部输入电压，ADC0 ~ ADC14

反推 VCC, 【ADC_VREF+/ADC_AVCC/MCU_VCC】

采样转换二次，只需要计算一次

====1, 假定 ADC 采样转换足够快

====2, 假定 2 次 ADC 转换期间, 【ADC_VREF+/ADC_VCC/MCU_VCC】

不变，变的误差也可以接受

====3, 应用场景, 【ADC_VREF+/ADC_AVCC/MCU_VCC】 这 3 条重要的电源线直接接在一起

Ai8051U 系列单片机内部集成了一个 10 位/12 位高速 A/D 转换器。ADC 的时钟频率为系统频率 2 分频再经过用户设置的分频系数进行再次分频（ADC 的工作时钟频率范围为 SYSclk/2/1 到 SYSclk/2/16）。

STC8H 系列/Ai8 系列的 ADC 最快速度：**12 位 ADC 为 800K**（每秒进行 **80** 万次 ADC 转换），**10 位 ADC 为 500K**（每秒进行 **50** 万次 ADC 转换）

ADC 转换结果的数据格式有两种：左对齐和右对齐。可方便用户程序进行读取和引用。

注意：ADC 的第 15 通道是专门测量内部 **1.19V** 参考信号源的通道，参考信号源值出厂时校准为 **1.19V**，由于制造误差以及测量误差，导致实际的内部参考信号源相比 **1.19V**，大约有 **±1%** 的误差。如果用户需要知道每一颗芯片的准确内部参考信号源值，可外接精准参考信号源，然后利用 ADC 的第 15 通道进行测量标定。**ADC_VRef+**脚外接参考电源时，可利用 ADC 的第 15 通道可以反推 **ADC_VRef+**脚外接参考电源的电压；如将 **ADC_VREF+**短接到 **MCU-VCC**，就可以反推 **MCU-VCC** 的电压。

如果芯片有 ADC 的外部参考电源管脚 **ADC_VRef+**，则一定不能浮空，必须接外部参考电源或者直接连到 **VCC**

=====

使用 ADC 的第 15 通道固定接的 1.19V 辅助信号源，反推外部通道输入电压，假设：当前已获取了内部参考信号源电压为 **BGV**，从 CHIP 中读取，或 AiCube-ISP 烧录时指定将重要参数烧录入程序 Flash，

内部参考信号源 ADC15 测量值为 **resbg**，对 ADC15/1.19V 采样转换一次就知道转换后的值；外部通道输入电压 ADCx 测量值为 **resx**，对 ADCx 的外部输入信号采样转换一次就知道转换后的值

则外部通道输入电压 **Vx=BGV / resbg * resx;**

采样转换二次，只需要计算一次

注意，是假定 2 次采样转换期间 【ADC_VREF+ = ADC_VCC = MCU_VCC】 不变

==所以对外部采样转换一次，也要对内部 ADC15 接的信号源立即采样转换一次

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

#define FOSC      11059200UL
#define BRT       (65536 - (FOSC / 115200+2) / 4)
                                         //加2 操作是为了让 Keil 编译器
                                         //自动实现四舍五入运算

int      *BGV;                         //内部参考信号源值存放在idata 中
                                         //idata 的EFH 地址存放高字节
                                         //idata 的F0H 地址存放低字节
                                         //电压单位为毫伏(mV)

bit      busy;                         //忙标志位

void UartIsr() interrupt 4
{
    if (TI)
    {
        TI = 0;
        busy = 0;
    }
    if (RI)
    {
        RI = 0;
    }
}

void UartInit()
{
    SCON = 0x50;
    TMOD = 0x00;
    TLI = BRT;
    TH1 = BRT >> 8;
    TR1 = 1;
    AUXR = 0x40;
    busy = 0;
}

void UartSend(char dat)
{
    while (busy);
    busy = 1;
    SBUF = dat;
}

void ADCInit()
{
    ADCTIM = 0x3f;                      //设置ADC 内部时序
                                         //ADCCFG = 0x2f;
                                         //ADC_CONTR = 0x8f;
                                         //使能ADC 模块,并选择第15 通道
}
```

```

int ADCRead()
{
    int res;

    ADC_CONTR |= 0x40; //启动AD 转换
    _nop_();
    _nop_();
    while (!(ADC_CONTR & 0x20)); //查询ADC 完成标志
    ADC_CONTR &= ~0x20; //清完成标志
    res = (ADC_RES << 8) / ADC_RESL; //读取ADC 结果

    return res;
}

void main()
{
    int res;
    int vcc;
    int i;

    P_SW2 |= 0x80; //使能访问XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    BGV = (int iodata *)0xef;
    ADCInit(); //ADC 初始化
    UartInit(); //串口初始化

    ES = I;
    EA = I;

//    ADCRead();
//    ADCRead(); //前两个数据建议丢弃

    res = 0;
    for (i=0; i<8; i++)
    {
        res += ADCRead(); //读取 8 次数据
    }
    res >= 3; //取平均值

    vcc = (int)(4096L * *BGV / res); //12 位ADC 算法计算VREF 管脚电压,即电池电压
//    vcc = (int)(1024L * *BGV / res); //10 位ADC 算法计算VREF 管脚电压,即电池电压
//    注意,此电压的单位为毫伏(mV)
    UartSend(vcc >> 8); //输出电压值到串口
    UartSend(vcc);
}

```

```

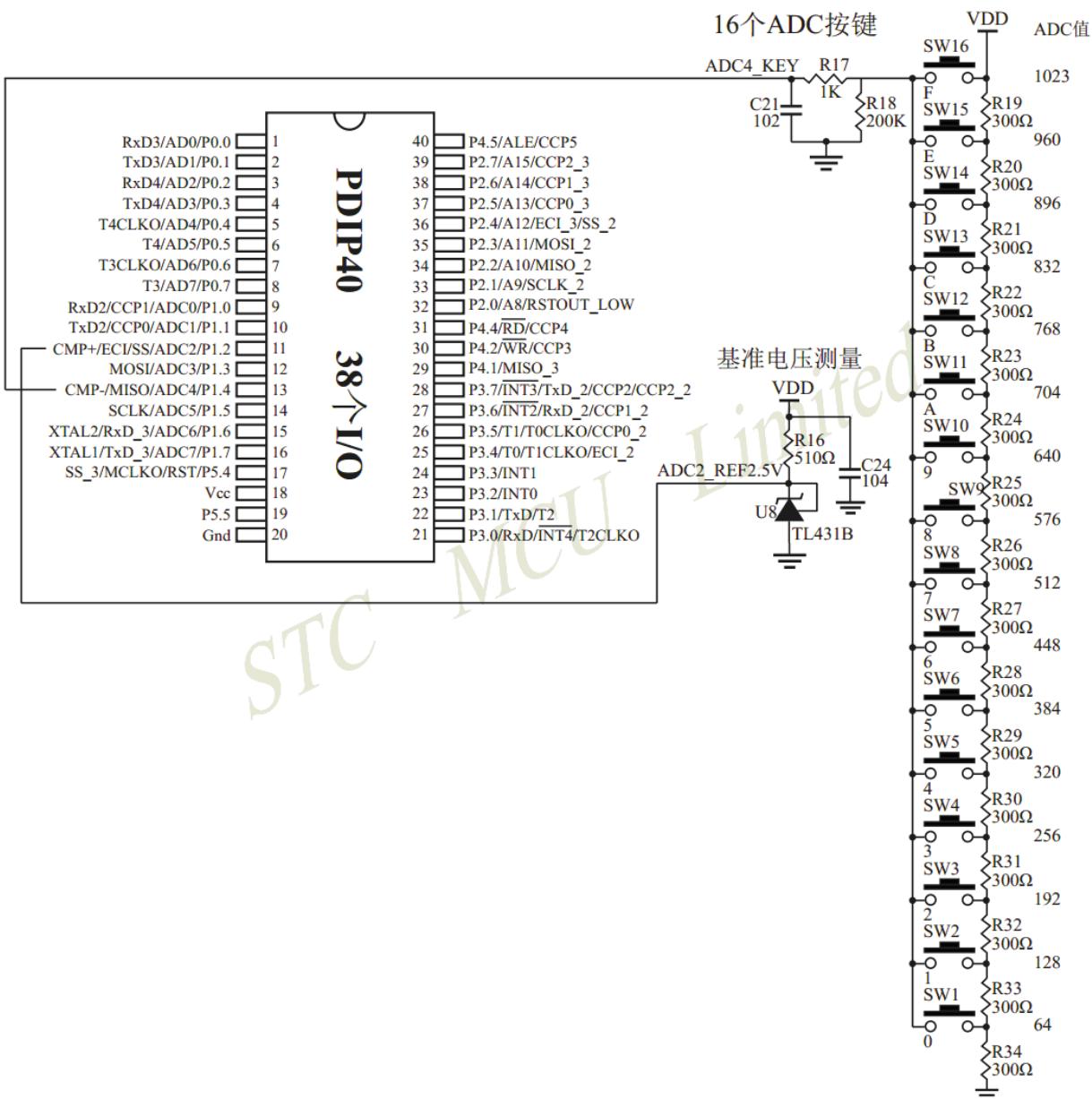
while (1);
}

```

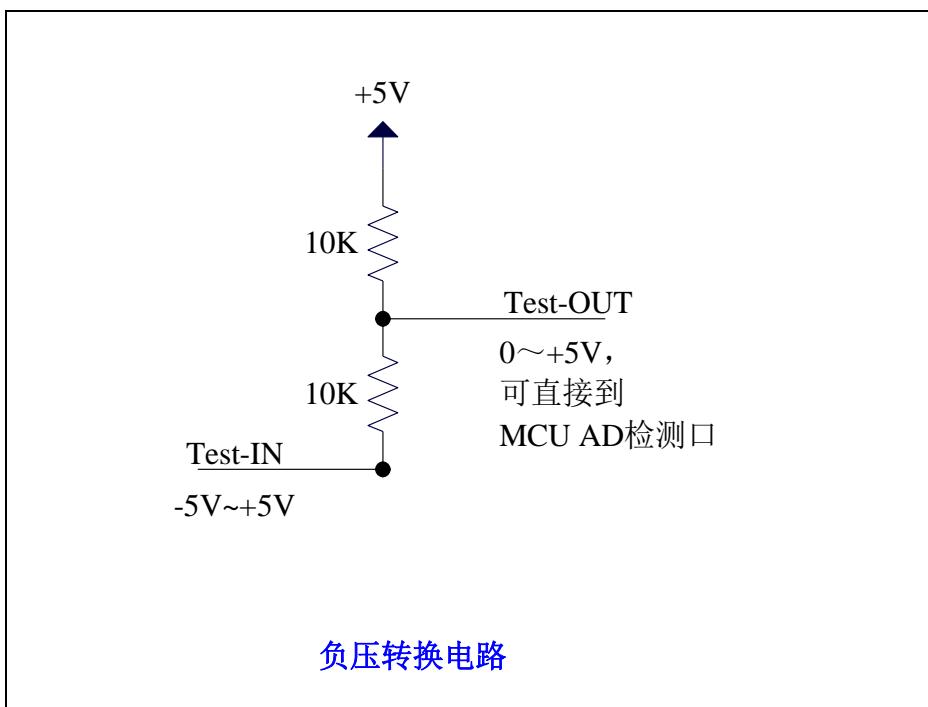
上面的方法是使用 ADC 的第 15 通道反推外部电池电压的。在 ADC 测量范围内，ADC 的外部测量电压与 ADC 的测量值是成正比例的，所以也可以使用 ADC 的第 15 通道反推外部通道输入电压，假设当前已获取了内部参考信号源电压为 BGV，内部参考信号源的 ADC 测量值为 res_{bg}，外部通道输入电压的 ADC 测量值为 res_x，则外部通道输入电压 $V_x = BGV / res_{bg} * res_x$ ；

21.7.5 ADC 作按键扫描应用线路图

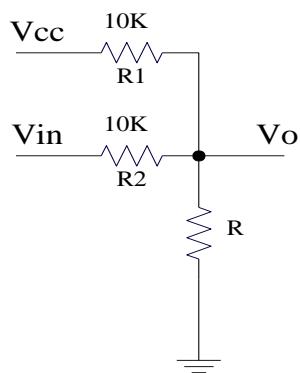
读 ADC 键的方法：每隔 10ms 左右读一次 ADC 值，并且保存最后 3 次的读数，其变化比较小时再判断键。判断键有效时，允许一定的偏差，比如±16 个字的偏差。



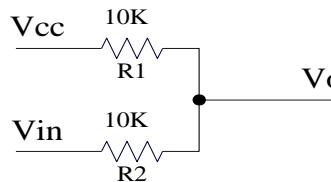
21.7.6 检测负电压参考线路图



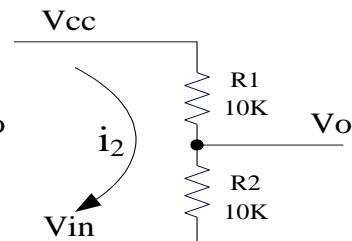
21.7.7 常用加法电路在 ADC 中的应用



常用加法电路



简化加法电路



变形成为分压电路形式

参照分压电路得到公式 1

$$\text{公式 1: } V_o = V_{in} + i_2 \cdot R_2$$

$$\text{公式 2: } i_2 = (V_{cc} - V_{in}) / (R_1 + R_2) \quad \{\text{条件: 流向 } V_o \text{ 的电流 } \approx 0\}$$

将 $R_1=R_2$ 代入公式 2 得公式 3

$$\text{公式 3: } i_2 = (V_{cc} - V_{in}) / 2R_2$$

将公式 3 代入公式 1 得公式 4

$$\text{公式 4: } V_o = (V_{cc} + V_{in}) / 2$$

根据公式 4, 可以将以上电路看成加法电路。

在单片机的模数转换测量中, 要求被测电压大于 0 并且小于 V_{cc} 。如果被测电压小于 0V, 可以利用加法电路将被测电压提升到 0V 以上。此时对被测电压的变化范围有一定的要求:

把上述条件代入公式 4 可得到下面 2 式

$$(V_{cc} + V_{in}) / 2 > 0 \quad \text{即 } V_{in} > -V_{cc}$$

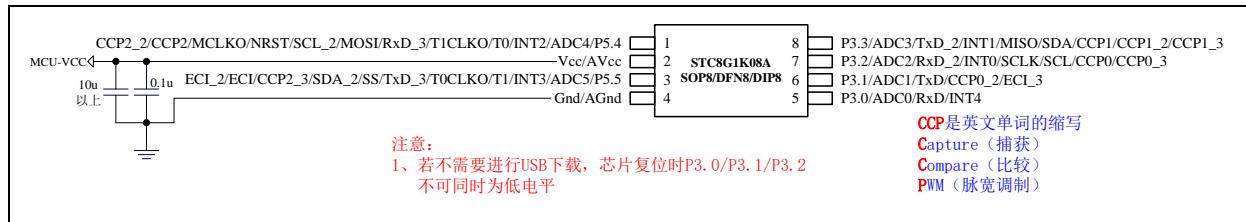
$$(V_{cc} + V_{in}) / 2 < V_{cc} \quad \text{即 } V_{in} < V_{cc}$$

$$\text{上面 2 式可以合起来: } -V_{cc} < V_{in} < V_{cc}$$

22 STC8G1K08A 专有的传统 PCA/CCP/PWM 应用

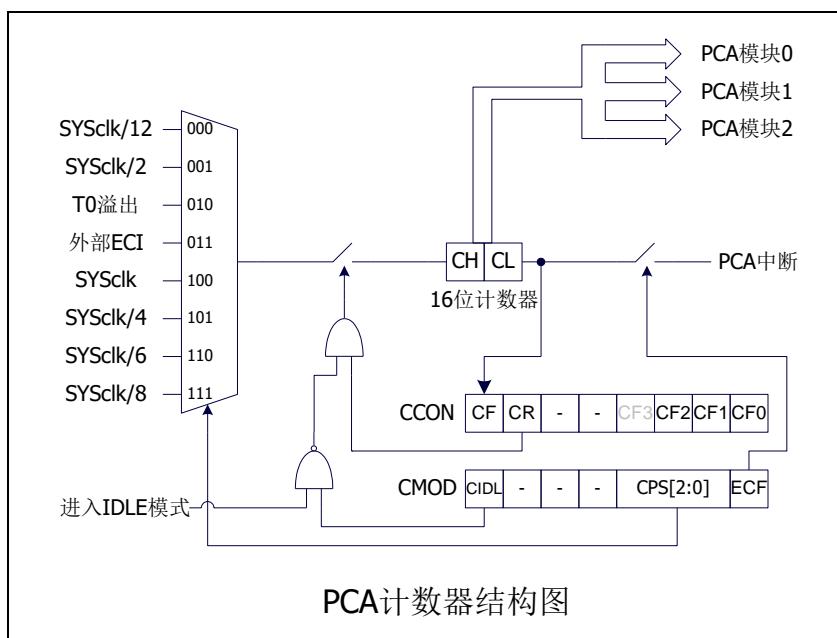
产品线	PCA
STC8G1K08-8Pin 系列	
STC8G1K08A 系列	●

STC8G1K08A-36I-SOP8、STC8G1K17A-36I-SOP8 才有传统的 PCA/CCP/PWM，管脚图如下：



STC8G 系列单片机内部集成了 3 组可编程计数器阵列 (PCA/CCP/PWM) 模块，可用于软件定时器、外部脉冲捕获、高速脉冲输出和 PWM 脉宽调制输出。

PCA 内部含有一个特殊的 16 位计数器，3 组 PCA 模块均与之相连接。PCA 计数器的结构图如下：



22.1 PCA 功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW1	A2H	S1_S[1:0]		CCP_S[1:0]		SPI_S[1:0]		0	-

CCP_S[1:0]: PCA 功能脚选择位

CCP_S[1:0]	ECI	CCP0	CCP1	CCP2
00	P1.2	P1.1	P1.0	P3.7
01	P3.4	P3.5	P3.6	P3.7
10	P2.4	P2.5	P2.6	P2.7
11	-	-	-	-

CCP_S[1:0]: PCA 功能脚选择位 ([STC8G1K08A 系列](#))

CCP_S[1:0]	ECI	CCP0	CCP1	CCP2
00	P5.5	P3.2	P3.3	P5.4
01	P5.5	P3.1	P3.3	P5.4
10	P3.1	P3.2	P3.3	P5.5
11	-	-	-	-

22.2 PCA 相关的寄存器

符号	描述	地址	位地址与符号								复位值				
			B7	B6	B5	B4	B3	B2	B1	B0					
CCON	PCA 控制寄存器	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0	00xx,x000				
CMOD	PCA 模式寄存器	D9H	CIDL	-	-	-	CPS[2:0]			ECF	0xxx,0000				
CCAPM0	PCA 模块 0 模式控制寄存器	DAH	-	ECOM0	CCAPPO	CCAPN0	MAT0	TOG0	PWM0	ECCF0	x000,0000				
CCAPM1	PCA 模块 1 模式控制寄存器	DBH	-	ECOM1	CCAPP1	CCAPN1	MAT1	TOG1	PWM1	ECCF1	x000,0000				
CCAPM2	PCA 模块 2 模式控制寄存器	DCH	-	ECOM2	CCAPP2	CCAPN2	MAT2	TOG2	PWM2	ECCF2	x000,0000				
CL	PCA 计数器低字节	E9H									0000,0000				
CCAP0L	PCA 模块 0 低字节	EAH									0000,0000				
CCAP1L	PCA 模块 1 低字节	EBH									0000,0000				
CCAP2L	PCA 模块 2 低字节	ECH									0000,0000				
PCA_PWM0	PCA0 的 PWM 模式寄存器	F2H	EBS0[1:0]		XCCAP0H[1:0]		XCCAP0L[1:0]		EPC0H	EPC0L	0000,0000				
PCA_PWM1	PCA1 的 PWM 模式寄存器	F3H	EBS1[1:0]		XCCAP1H[1:0]		XCCAP1L[1:0]		EPC1H	EPC1L	0000,0000				
PCA_PWM2	PCA2 的 PWM 模式寄存器	F4H	EBS2[1:0]		XCCAP2H[1:0]		XCCAP2L[1:0]		EPC2H	EPC2L	0000,0000				
CH	PCA 计数器高字节	F9H									0000,0000				
CCAP0H	PCA 模块 0 高字节	FAH									0000,0000				
CCAP1H	PCA 模块 1 高字节	FBH									0000,0000				
CCAP2H	PCA 模块 2 高字节	FCH									0000,0000				

22.2.1 PCA 控制寄存器 (CCON)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCON	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0

CF: PCA 计数器溢出中断标志。当 PCA 的 16 位计数器计数发生溢出时，硬件自动将此位置 1，并向 CPU 提出中断请求。此标志位需要软件清零。

CR: PCA 计数器允许控制位。

0: 停止 PCA 计数

1: 启动 PCA 计数

CCFn (n=0,1,2): PCA 模块中断标志。当 PCA 模块发生匹配或者捕获时，硬件自动将此位置 1，并向 CPU 提出中断请求。此标志位需要软件清零。

22.2.2 PCA 模式寄存器 (CMOD)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CMOD	D9H	CIDL	-	-	-	CPS[2:0]			ECF

CIDL: 空闲模式下是否停止 PCA 计数。

0: 空闲模式下 PCA 继续计数

1: 空闲模式下 PCA 停止计数

CPS[2:0]: PCA 计数脉冲源选择位

CPS[2:0]	PCA 的输入时钟源	注意事项
000	系统时钟/12	
001	系统时钟/2	
010	定时器 0 的溢出脉冲	
011	ECI 脚的外部输入时钟	外部时钟频率不能高于系统频率的 1/2
100	系统时钟	
101	系统时钟/4	
110	系统时钟/6	
111	系统时钟/8	

ECF: PCA 计数器溢出中断允许位。

0: 禁止 PCA 计数器溢出中断

1: 使能 PCA 计数器溢出中断

22.2.3 PCA 计数器寄存器 (CL, CH)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CL	E9H								
CH	F9H								

由 CL 和 CH 两个字节组合成一个 16 位计数器，CL 为低 8 位计数器，CH 为高 8 位计数器。每个 PCA 时钟 16 位计数器自动加 1。

22.2.4 PCA 模块模式控制寄存器 (CCAPM_n)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCAPM0	DAH	-	ECOM0	CCAPP0	CCAPN0	MAT0	TOG0	PWM0	ECCF0
CCAPM1	DBH	-	ECOM1	CCAPP1	CCAPN1	MAT1	TOG1	PWM1	ECCF1
CCAPM2	DCH	-	ECOM2	CCAPP2	CCAPN2	MAT2	TOG2	PWM2	ECCF2

ECOM_n: 允许 PCA 模块 n 的比较功能

CCAPP_n: 允许 PCA 模块 n 进行上升沿捕获

CCAPN_n: 允许 PCA 模块 n 进行下降沿捕获

MAT_n: 允许 PCA 模块 n 的匹配功能

TOG_n: 允许 PCA 模块 n 的高速脉冲输出功能

PWM_n: 允许 PCA 模块 n 的脉宽调制输出功能

ECCF_n: 允许 PCA 模块 n 的匹配/捕获中断

22.2.5 PCA 模块模式捕获值/比较值寄存器 (CCAPnL, CCAPnH)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
CCAP0L	EAH								
CCAP1L	EBH								
CCAP2L	ECH								
CCAP0H	FAH								
CCAP1H	FBH								
CCAP2H	FCH								

当 PCA 模块捕获功能使能时, CCAPnL 和 CCAPnH 用于保存发生捕获时的 PCA 的计数值 (CL 和 CH);

当 PCA 模块比较功能使能时, PCA 控制器会将当前 CL 和 CH 中的计数值与保存在 CCAPnL 和 CCAPnH 中的值进行比较, 并给出比较结果; 当 PCA 模块匹配功能使能时, PCA 控制器会将当前 CL 和 CH 中的计数值与保存在 CCAPnL 和 CCAPnH 中的值进行比较, 看是否匹配 (相等), 并给出匹配结果。

22.2.6 PCA 模块 PWM 模式控制寄存器 (PCA_PWMn)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM0	F2H	EBS0[1:0]		XCCAP0H[1:0]		XCCAP0L[1:0]		EPC0H	EPC0L
PCA_PWM1	F3H	EBS1[1:0]		XCCAP1H[1:0]		XCCAP1L[1:0]		EPC1H	EPC1L
PCA_PWM2	F4H	EBS2[1:0]		XCCAP2H[1:0]		XCCAP2L[1:0]		EPC2H	EPC2L

EBSn[1:0]: PCA 模块 n 的 PWM 位数控制

EBSn[1:0]	PWM 位数	重载值	比较值
00	8 位 PWM	{EPCnH, CCAPnH[7:0]}	{EPCnL, CCAPnL[7:0]}
01	7 位 PWM	{EPCnH, CCAPnH[6:0]}	{EPCnL, CCAPnL[6:0]}
10	6 位 PWM	{EPCnH, CCAPnH[5:0]}	{EPCnL, CCAPnL[5:0]}
11	10 位 PWM	{EPCnH, XCCAPnH[1:0], CCAPnH[7:0]}	{EPCnL, XCCAPnL[1:0], CCAPnL[7:0]}

XCCAPnH[1:0]: 10 位 PWM 的第 9 位和第 10 位的重载值

XCCAPnL[1:0]: 10 位 PWM 的第 9 位和第 10 位的比较值

EPCnH: PWM 模式下, 重载值的最高位 (8 位 PWM 的第 9 位, 7 位 PWM 的第 8 位, 6 位 PWM 的第 7 位, 10 位 PWM 的第 11 位)

EPCnL: PWM 模式下, 比较值的最高位 (8 位 PWM 的第 9 位, 7 位 PWM 的第 8 位, 6 位 PWM 的第 7 位, 10 位 PWM 的第 11 位)

注意: 在更新 10 位 PWM 的重载值时, 必须先写高两位 XCCAPnH[1:0], 再写低 8 位 CCAPnH[7:0]。

22.3 PCA 工作模式

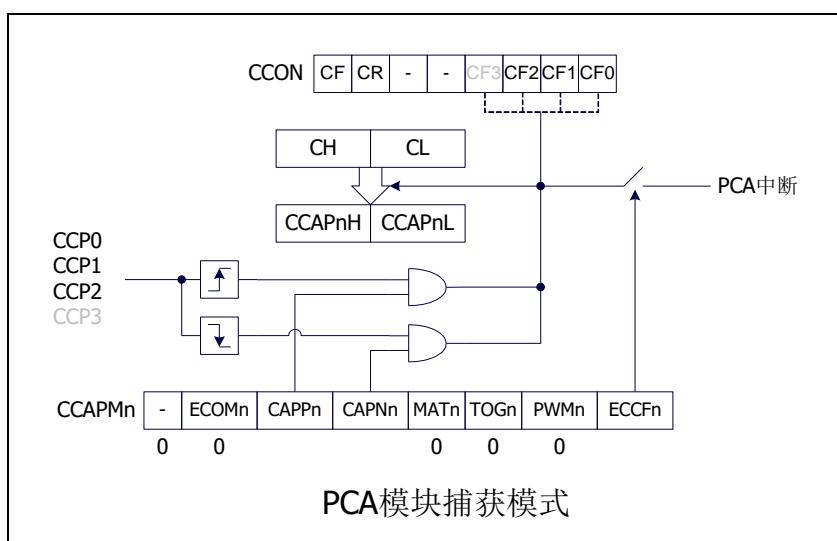
STC8 系列单片机共有 4 组 PCA 模块，每组模块都可独立设置工作模式。模式设置如下所示：

CCAPMn							模块功能	
-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	
-	0	0	0	0	0	0	0	无操作
-	1	0	0	0	0	1	0	6/7/8/10 位 PWM 模式, 无中断
-	1	1	0	0	0	1	1	6/7/8/10 位 PWM 模式, 产生上升沿中断
-	1	0	1	0	0	1	1	6/7/8/10 位 PWM 模式, 产生下降沿中断
-	1	1	1	0	0	1	1	6/7/8/10 位 PWM 模式, 产生边沿中断
-	0	1	0	0	0	0	x	16 位上升沿捕获
-	0	0	1	0	0	0	x	16 位下降沿捕获
-	0	1	1	0	0	0	x	16 位边沿捕获
-	1	0	0	1	0	0	x	16 位软件定时器
-	1	0	0	1	1	0	x	16 位高速脉冲输出

22.3.1 捕获模式

要使一个 PCA 模块工作于捕获模式，寄存器 CCAPMn 中的 CAPNn 和 CAPPn 至少有一位必须置 1（也可两位都置 1）。PCA 模块工作于捕获模式时，对模块的外部 CCP0/CCP1/CCP2 管脚的输入跳变进行采样。当采样到有效跳变时，PCA 控制器立即将 PCA 计数器 CH 和 CL 中的计数值装载到模块的捕获寄存器中 CCAPnL 和 CCAPnH，同时将 CCON 寄存器中相应的 CCFn 置 1。若 CCAPMn 中的 ECCFn 位被设置为 1，将产生中断。由于所有 PCA 模块的中断入口地址是共享的，所以在中断服务程序中需要判断是哪一个模块产生了中断，并注意中断标志位需要软件清零。

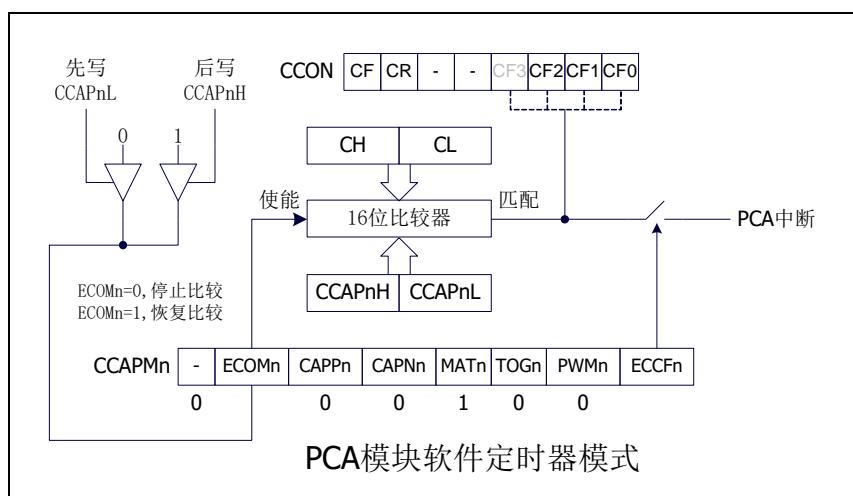
PCA 模块工作于捕获模式的结构图如下图所示：



22.3.2 软件定时器模式

通过置位 CCAPMn 寄存器的 ECOM 和 MAT 位，可使 PCA 模块用作软件定时器。PCA 计数器值 CL 和 CH 与模块捕获寄存器的值 CCAPnL 和 CCAPnH 相比较，当两者相等时，CCON 中的 CCFn 会被置 1，若 CCAPMn 中的 ECCFn 被设置为 1 时将产生中断。CCFn 标志位需要软件清零。

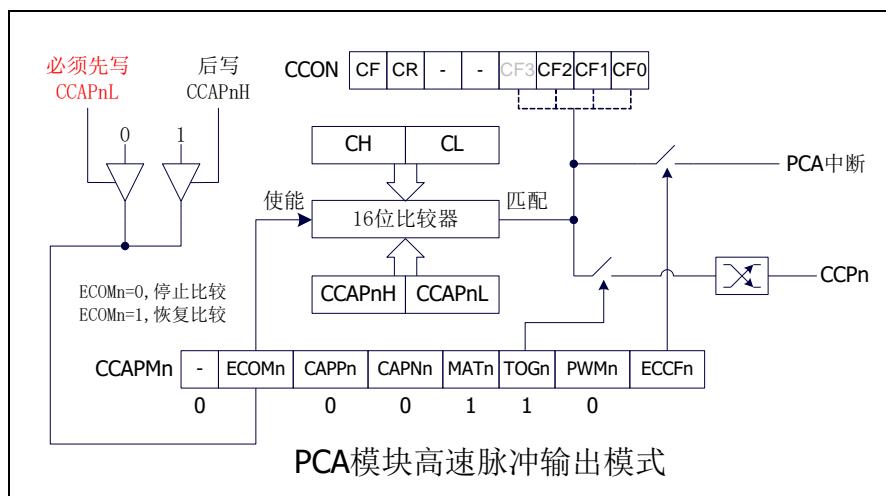
PCA 模块工作于软件定时器模式的结构图如下图所示：



22.3.3 高速脉冲输出模式

当 PCA 计数器的计数值与模块捕获寄存器的值相匹配时,PCA 模块的 CCPn 输出将发生翻转。要激活高速脉冲输出模式，CCAPMn 寄存器的 TOGn、MATn 和 ECOMn 位必须都置 1。

PCA 模块工作于高速脉冲输出模式的结构图如下图所示：

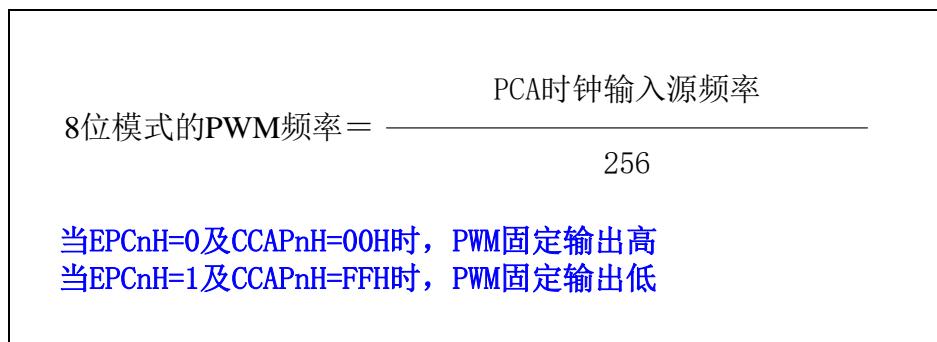


22.3.4 PWM 脉宽调制模式及频率计算公式

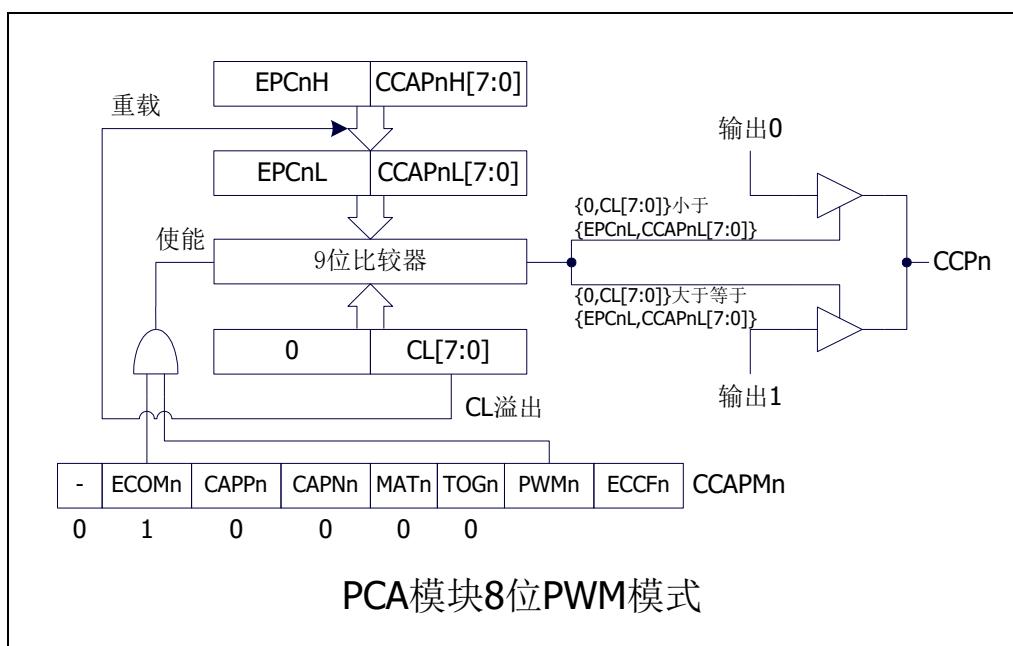
22.3.4.1 8 位 PWM 模式

脉宽调制是使用程序来控制波形的占空比、周期、相位波形的一种技术，在三相电机驱动、D/A 转换等场合有广泛的应用。STC8 系列单片机的 PCA 模块可以通过设定各自的 PCA_PWMn 寄存器使其工作于 8 位 PWM 或 7 位 PWM 或 6 位 PWM 或 10 位 PWM 模式。要使能 PCA 模块的 PWM 功能，模块寄存器 CCAPMn 的 PWMn 和 ECOMn 位必须置 1。

PCA_PWMn 寄存器中的 EBSn[1:0]设置为 00 时，PCA 模块 n 工作于 8 位 PWM 模式，此时将 {0,CL[7:0]} 与捕获寄存器 {EPCnL,CCAPnL[7:0]} 进行比较。当 PCA 模块工作于 8 位 PWM 模式时，由于所有模块共用一个 PCA 计数器，所有它们的输出频率相同。各个模块的输出占空比使用寄存器 {EPCnL,CCAPnL[7:0]} 进行设置。当 {0,CL[7:0]} 的值小于 {EPCnL,CCAPnL[7:0]} 时，输出为低电平；当 {0,CL[7:0]} 的值等于或大于 {EPCnL,CCAPnL[7:0]} 时，输出为高电平。当 CL[7:0] 的值由 FF 变为 00 溢出时，{EPCnH,CCAPnH[7:0]} 的内容重新装载到 {EPCnL,CCAPnL[7:0]} 中。这样就可实现无干扰地更新 PWM。

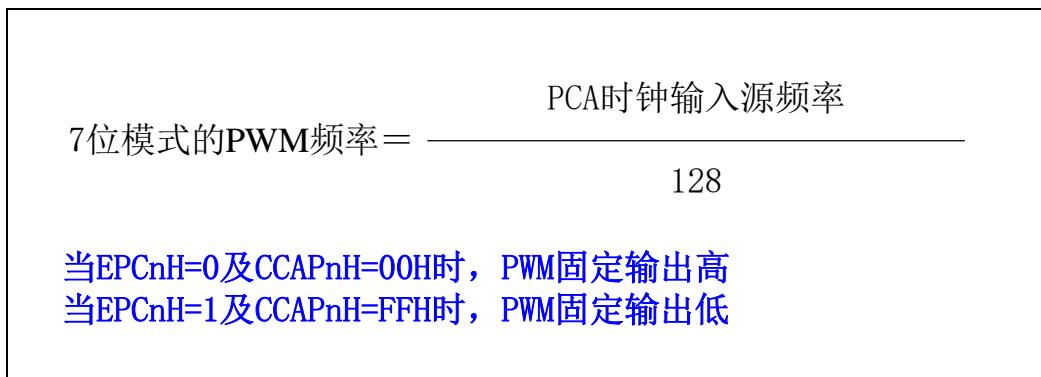


PCA 模块工作于 8 位 PWM 模式的结构图如下图所示：

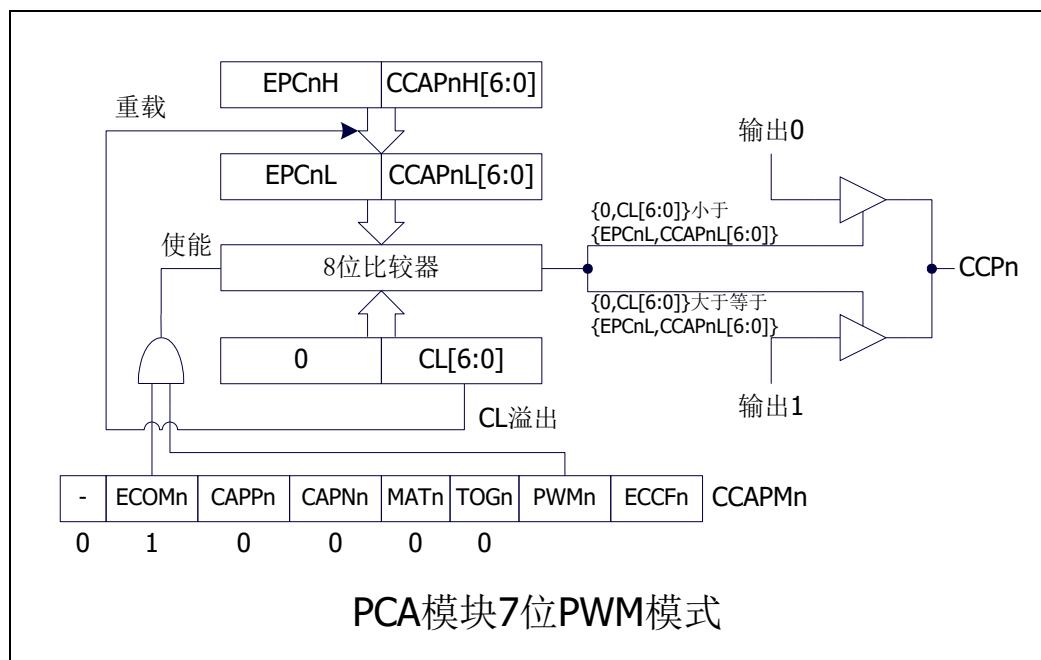


22.3.4.2 7 位 PWM 模式

PCA_PWMn 寄存器中的 EBSn[1:0] 设置为 01 时, PCA 模块 n 工作于 7 位 PWM 模式, 此时将 {0, CL[6:0]} 与捕获寄存器 {EPCnL, CCAPnL[6:0]} 进行比较。当 PCA 模块工作于 7 位 PWM 模式时, 由于所有模块共用一个 PCA 计数器, 所有它们的输出频率相同。各个模块的输出占空比使用寄存器 {EPCnL, CCAPnL[6:0]} 进行设置。当 {0, CL[6:0]} 的值小于 {EPCnL, CCAPnL[6:0]} 时, 输出为低电平; 当 {0, CL[6:0]} 的值等于或大于 {EPCnL, CCAPnL[6:0]} 时, 输出为高电平。当 CL[6:0] 的值由 7F 变为 00 溢出时, {EPCnH, CCAPnH[6:0]} 的内容重新装载到 {EPCnL, CCAPnL[6:0]} 中。这样就可实现无干扰地更新 PWM。

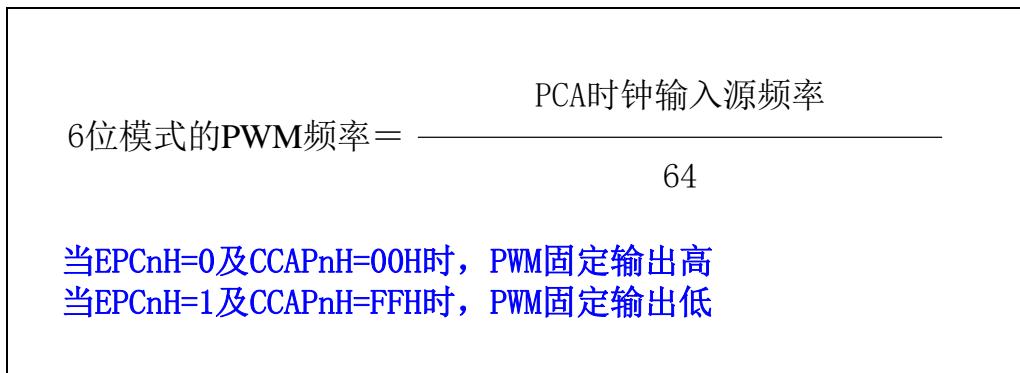


PCA 模块工作于 7 位 PWM 模式的结构图如下图所示:

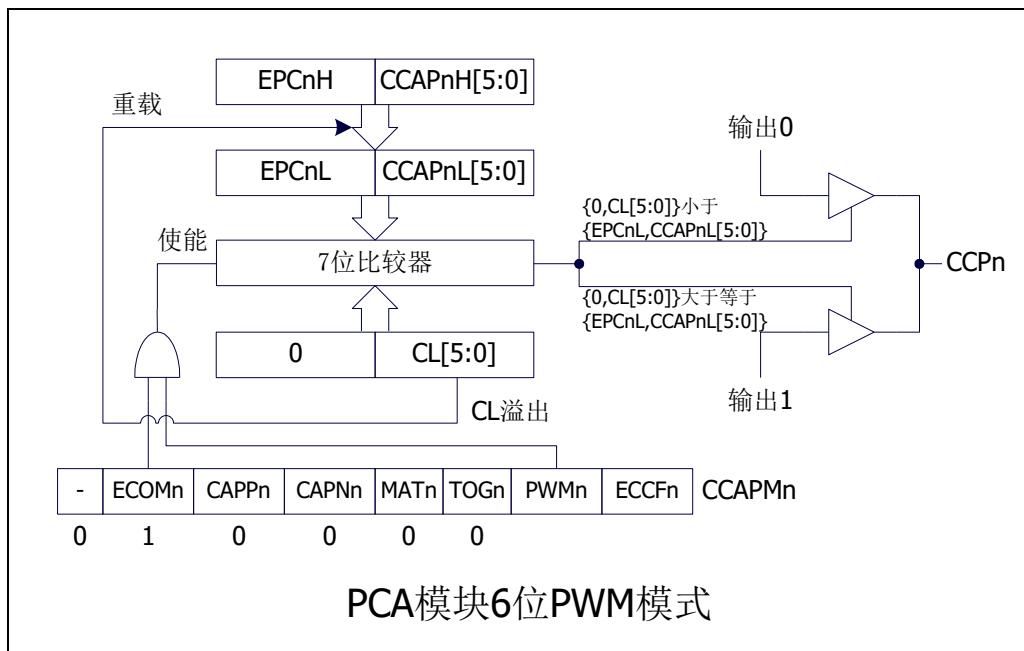


22.3.4.3 6 位 PWM 模式

PCA_PWMn 寄存器中的 EBSn[1:0]设置为 10 时, PCA 模块 n 工作于 6 位 PWM 模式, 此时将 {0,CL[5:0]} 与捕获寄存器 {EPCnL,CCAPnL[5:0]} 进行比较。当 PCA 模块工作于 6 位 PWM 模式时, 由于所有模块共用一个 PCA 计数器, 所有它们的输出频率相同。各个模块的输出占空比使用寄存器 {EPCnL,CCAPnL[5:0]} 进行设置。当 {0,CL[5:0]} 的值小于 {EPCnL,CCAPnL[5:0]} 时, 输出为低电平; 当 {0,CL[5:0]} 的值等于或大于 {EPCnL,CCAPnL[5:0]} 时, 输出为高电平。当 CL[5:0] 的值由 3F 变为 00 溢出时, {EPCnH,CCAPnH[5:0]} 的内容重新装载到 {EPCnL,CCAPnL[5:0]} 中。这样就可实现无干扰地更新 PWM。

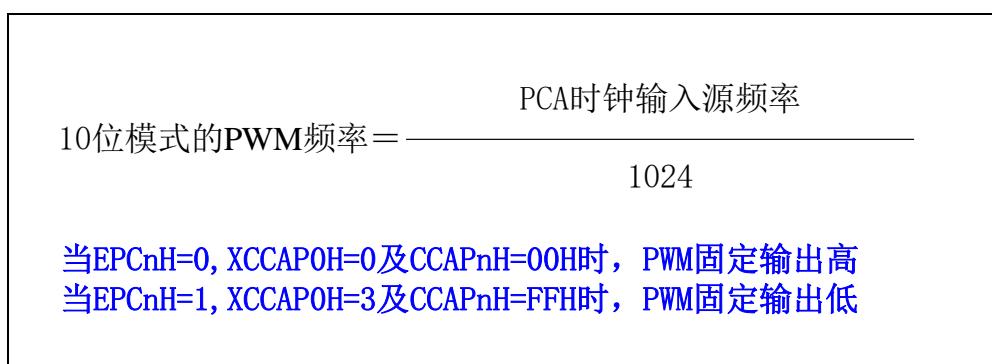


PCA 模块工作于 6 位 PWM 模式的结构图如下图所示:

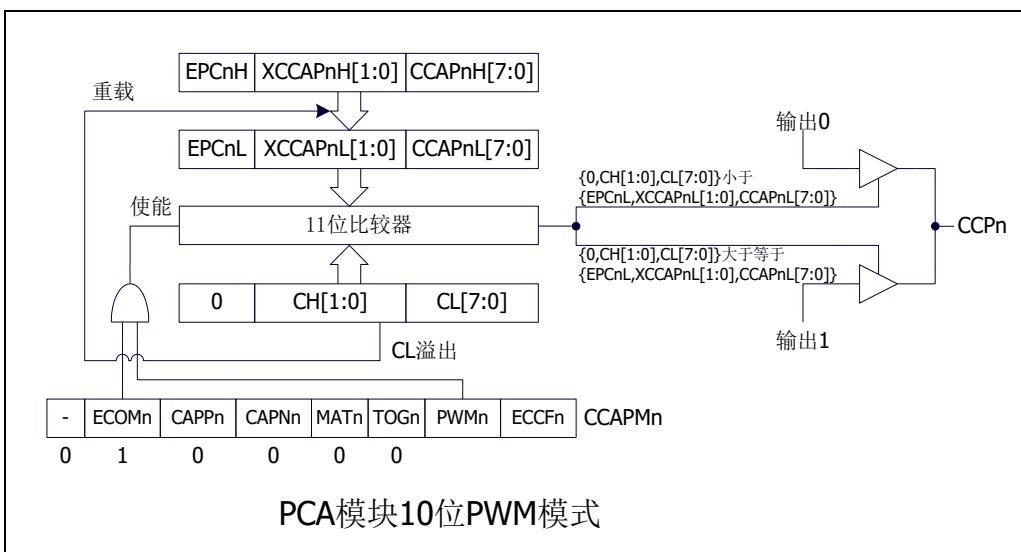


22.3.4.4 10 位 PWM 模式

PCA_PWMn 寄存器中的 EBSn[1:0]设置为 11 时，PCA 模块 n 工作于 10 位 PWM 模式，此时将 {CH[1:0],CL[7:0]} 与捕获寄存器 {EPCnL,XCCAPnL[1:0],CCAPnL[7:0]} 进行比较。当 PCA 模块工作于 10 位 PWM 模式时，由于所有模块共用一个 PCA 计数器，所有它们的输出频率相同。各个模块的输出占空比使用寄存器 {EPCnL,XCCAPnL[1:0],CCAPnL[7:0]} 进行设置。当 {CH[1:0],CL[7:0]} 的值小于 {EPCnL,XCCAPnL[1:0],CCAPnL[7:0]} 时，输出为低电平；当 {CH[1:0],CL[7:0]} 的值等于或大于 {EPCnL,XCCAPnL[1:0],CCAPnL[7:0]} 时，输出为高电平。当 {CH[1:0],CL[7:0]} 的值由 3FF 变为 00 溢出时，{EPCnH,XCCAPnH[1:0],CCAPnH[7:0]} 的内容重新装载到 {EPCnL,XCCAPnL[1:0],CCAPnL[7:0]} 中。这样就可实现无干扰地更新 PWM。



PCA 模块工作于 10 位 PWM 模式的结构图如下图所示：

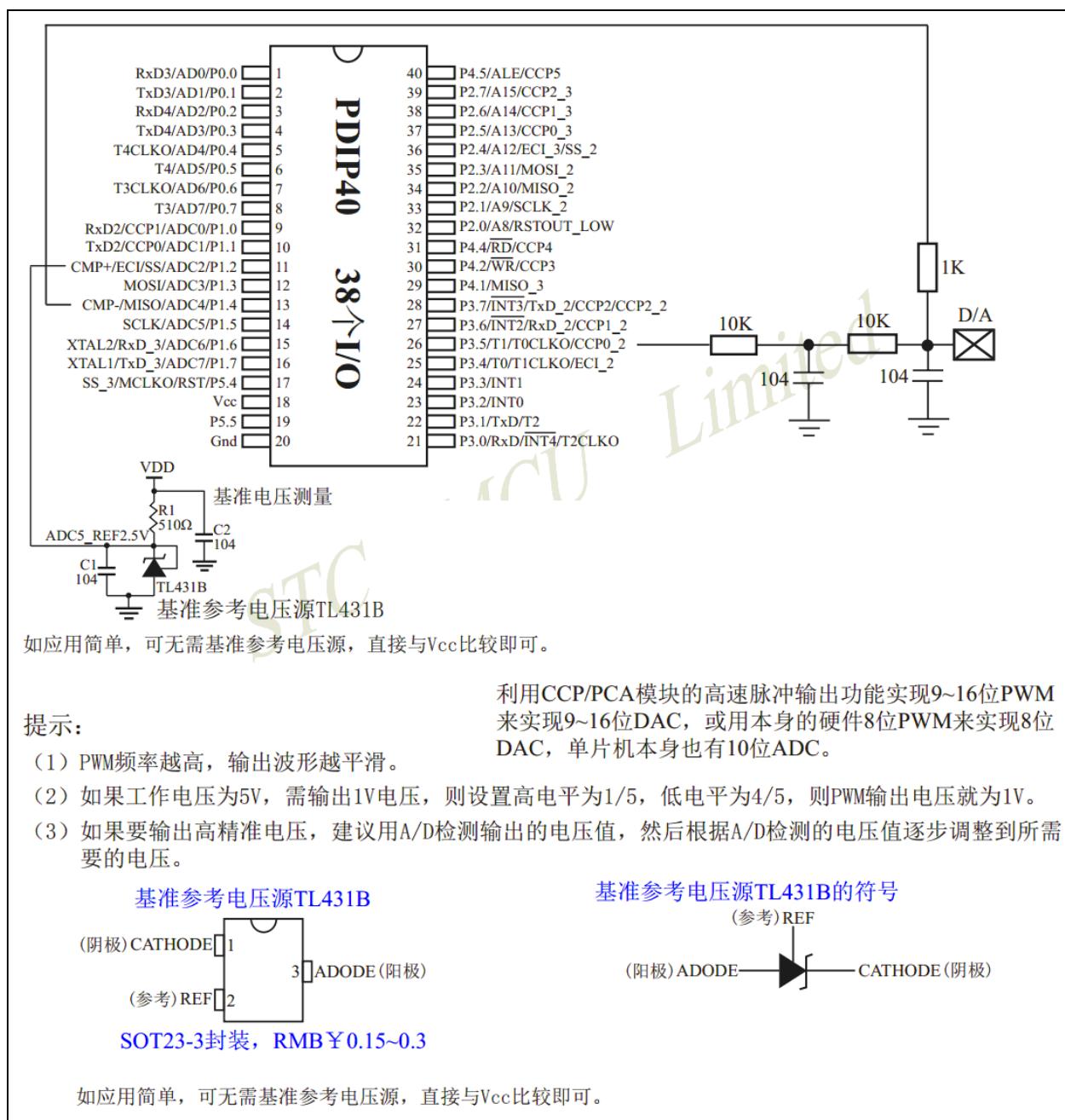


22.3.4.5 如何控制 PWM 固定输出高电平/低电平

当 PCA_PWMn &= 0xC0, CCAPnH = 0x00 时， PWM 固定输出高电平

当 PCA_PWMn |= 0x3F, CCAPnH = 0xFF 时， PWM 固定输出低电平

22.4 利用 CCP/PCA/PWM 模块实现 8~16 位 DAC 的参考线路图



22.5 范例程序

22.5.1 PCA 输出 PWM (6/7/8/10 位)

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "reg51.h"
#include "intrins.h"

sfr CCON      = 0xd8;
sbit CF       = CCON^7;
sbit CR       = CCON^6;
sbit CCF2     = CCON^2;
sbit CCF1     = CCON^1;
sbit CCF0     = CCON^0;
sfr CMOD     = 0xd9;
sfr CL       = 0xe9;
sfr CH       = 0xf9;
sfr CCAPM0   = 0xda;
sfr CCAP0L   = 0xea;
sfr CCAP0H   = 0xfa;
sfr PCA_PWM0 = 0xf2;
sfr CCAPM1   = 0xdb;
sfr CCAP1L   = 0xeb;
sfr CCAP1H   = 0xfb;
sfr PCA_PWM1 = 0xf3;
sfr CCAPM2   = 0xdc;
sfr CCAP2L   = 0xec;
sfr CCAP2H   = 0xfc;
sfr PCA_PWM2 = 0xf4;

sfr P0M1     = 0x93;
sfr P0M0     = 0x94;
sfr P1M1     = 0x91;
sfr P1M0     = 0x92;
sfr P2M1     = 0x95;
sfr P2M0     = 0x96;
sfr P3M1     = 0xb1;
sfr P3M0     = 0xb2;
sfr P4M1     = 0xb3;
sfr P4M0     = 0xb4;
sfr P5M1     = 0xc9;
sfr P5M0     = 0xca;

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
```

```

P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

CCON = 0x00;
CMOD = 0x08;                                //PCA 时钟为系统时钟
CL = 0x00;
CH = 0x00;

//--6 位 PWM --
CCAPM0 = 0x42;                            //PCA 模块0 为 PWM 工作模式
PCA_PWM0 = 0x80;                           //PCA 模块0 输出6 位 PWM
CCAP0L = 0x20;                             //PWM 占空比为50%[(40H-20H)/40H]
CCAP0H = 0x20;

//--7 位 PWM --
CCAPM1 = 0x42;                            //PCA 模块1 为 PWM 工作模式
PCA_PWM1 = 0x40;                           //PCA 模块1 输出7 位 PWM
CCAP1L = 0x20;                             //PWM 占空比为75%[(80H-20H)/80H]
CCAP1H = 0x20;

//--8 位 PWM --
CCAPM2 = 0x42;                            //PCA 模块2 为 PWM 工作模式
PCA_PWM2 = 0xc0;                           //PCA 模块2 输出8 位 PWM
CCAP2L = 0x20;                             //PWM 占空比为87.5%[(100H-20H)/100H]
CCAP2H = 0x20;

//--10 位 PWM --
CCAPM2 = 0x42;                            //PCA 模块2 为 PWM 工作模式
PCA_PWM2 = 0xc0;                           //PCA 模块2 输出10 位 PWM
CCAP2L = 0x20;                            //PWM 占空比为96.875%[(400H-20H)/400H]
CCAP2H = 0x20;

CR = 1;                                    //启动 PCA 计时器

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

CCON	DATA	0D8H
CF	BIT	CCON.7
CR	BIT	CCON.6
CCF2	BIT	CCON.2
CCF1	BIT	CCON.1
CCF0	BIT	CCON.0
CMOD	DATA	0D9H
CL	DATA	0E9H
CH	DATA	0F9H
CCAPM0	DATA	0DAH
CCAP0L	DATA	0EAH
CCAP0H	DATA	0FAH
PCA_PWM0	DATA	0F2H
CCAPM1	DATA	0DBH
CCAP1L	DATA	0EBH
CCAP1H	DATA	0FBH
PCA_PWM1	DATA	0F3H
CCAPM2	DATA	0DCH
CCAP2L	DATA	0ECH
CCAP2H	DATA	0FCH
PCA_PWM2	DATA	0F4H

```

P0M1      DATA      093H
P0M0      DATA      094H
P1M1      DATA      091H
P1M0      DATA      092H
P2M1      DATA      095H
P2M0      DATA      096H
P3M1      DATA      0B1H
P3M0      DATA      0B2H
P4M1      DATA      0B3H
P4M0      DATA      0B4H
P5M1      DATA      0C9H
P5M0      DATA      0CAH

        ORG      0000H
        LJMP    START

        ORG      0100H
START:
        MOV      SP, #5FH
        MOV      P0M0, #00H
        MOV      P0M1, #00H
        MOV      P1M0, #00H
        MOV      P1M1, #00H
        MOV      P2M0, #00H
        MOV      P2M1, #00H
        MOV      P3M0, #00H
        MOV      P3M1, #00H
        MOV      P4M0, #00H
        MOV      P4M1, #00H
        MOV      P5M0, #00H
        MOV      P5M1, #00H

        MOV      CCON, #00H
        MOV      CMOD, #08H ;PCA 时钟为系统时钟
        MOV      CL, #00H
        MOV      CH, #0H

;--6 位 PWM --
        MOV      CCAPM0, #42H ;PCA 模块0 为 PWM 工作模式
        MOV      PCA_PWM0, #80H ;PCA 模块0 输出6位 PWM
        MOV      CCAP0L, #20H ;PWM 占空比为 50%[(40H-20H)/40H]
        MOV      CCAP0H, #20H

;--7 位 PWM --
        MOV      CCAPM1, #42H ;PCA 模块1 为 PWM 工作模式
        MOV      PCA_PWM1, #40H ;PCA 模块1 输出7位 PWM
        MOV      CCAP1L, #20H ;PWM 占空比为 75%[(80H-20H)/80H]
        MOV      CCAP1H, #20H

;--8 位 PWM --
        ;       MOV      CCAPM2, #42H ;PCA 模块2 为 PWM 工作模式
        ;       MOV      PCA_PWM2, #00H ;PCA 模块2 输出8位 PWM
        ;       MOV      CCAP2L, #20H ;PWM 占空比为 87.5%[(100H-20H)/100H]
        ;       MOV      CCAP2H, #20H

;--10 位 PWM --
        MOV      CCAPM2, #42H ;PCA 模块2 为 PWM 工作模式
        MOV      PCA_PWM2, #0C0H ;PCA 模块2 输出10位 PWM
        MOV      CCAP2L, #20H ;PWM 占空比为 96.875%[(400H-20H)/400H]
        MOV      CCAP2H, #20H
        SETB    CR ;启动 PCA 计时器

        JMP      $

```

END

22.5.2 PCA 捕获测量脉冲宽度

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "reg51.h"
#include "intrins.h"

sfr CCON      = 0xd8;
sbit CF        = CCON^7;
sbit CR        = CCON^6;
sbit CCF2      = CCON^2;
sbit CCF1      = CCON^1;
sbit CCF0      = CCON^0;
sfr CMOD      = 0xd9;
sfr CL         = 0xe9;
sfr CH         = 0xf9;
sfr CCAPM0    = 0xda;
sfr CCAP0L    = 0xea;
sfr CCAP0H    = 0xfa;
sfr PCA_PWM0  = 0xf2;
sfr CCAPMI    = 0xdb;
sfr CCAP1L    = 0xeb;
sfr CCAP1H    = 0xfb;
sfr PCA_PWM1  = 0xf3;
sfr CCAPM2    = 0xdc;
sfr CCAP2L    = 0xec;
sfr CCAP2H    = 0xfc;
sfr PCA_PWM2  = 0xf4;

sfr P0M1      = 0x93;
sfr P0M0      = 0x94;
sfr P1M1      = 0x91;
sfr P1M0      = 0x92;
sfr P2M1      = 0x95;
sfr P2M0      = 0x96;
sfr P3M1      = 0xb1;
sfr P3M0      = 0xb2;
sfr P4M1      = 0xb3;
sfr P4M0      = 0xb4;
sfr P5M1      = 0xc9;
sfr P5M0      = 0xca;

unsigned char cnt;           //存储PCA 计时溢出次数
unsigned long count0;        //记录上一次的捕获值
unsigned long count1;        //记录本次的捕获值
unsigned long length;        //存储信号的时间长度

void PCA_Isr() interrupt 7
{
    if (CF)
    {
```

```

    CF = 0;
    cnt++;
}
if (CCF0)
{
    CCF0 = 0;
    count0 = count1;                                //备份上一次的捕获值
    ((unsigned char *)&count1)[3] = CCAP0L;
    ((unsigned char *)&count1)[2] = CCAP0H;
    ((unsigned char *)&count1)[1] = cnt;
    ((unsigned char *)&count1)[0] = 0;
    length = count1 - count0;                      //length 保存的即为捕获的脉冲宽度
}
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    cnt = 0;                                         //用户变量初始化
    count0 = 0;
    count1 = 0;
    length = 0;
    CCON = 0x00;
    CMOD = 0x09;                                     //PCA 时钟为系统时钟,使能PCA 计时中断
    CL = 0x00;
    CH = 0x00;
    CCAPM0 = 0x11;                                    //PCA 模块0 为16 位捕获模式 (下降沿捕获)
//    CCAPM0 = 0x21;                                    //PCA 模块0 为16 位捕获模式 (上升沿捕获)
//    CCAPM0 = 0x31;                                    //PCA 模块0 为16 位捕获模式 (边沿捕获)
    CCAP0L = 0x00;
    CCAP0H = 0x00;
    CR = 1;                                         //启动PCA 计时器
    EA = 1;

    while (1);
}

```

汇编代码

;测试工作频率为 11.0592MHz

CCON	DATA	0D8H
CF	BIT	CCON.7
CR	BIT	CCON.6
CCF2	BIT	CCON.2
CCF1	BIT	CCON.1
CCF0	BIT	CCON.0

<i>CMOD</i>	<i>DATA</i>	<i>0D9H</i>	
<i>CL</i>	<i>DATA</i>	<i>0E9H</i>	
<i>CH</i>	<i>DATA</i>	<i>0F9H</i>	
<i>CCAPM0</i>	<i>DATA</i>	<i>0DAH</i>	
<i>CCAP0L</i>	<i>DATA</i>	<i>0EAH</i>	
<i>CCAP0H</i>	<i>DATA</i>	<i>0FAH</i>	
<i>PCA_PWM0</i>	<i>DATA</i>	<i>0F2H</i>	
<i>CCAPM1</i>	<i>DATA</i>	<i>0DBH</i>	
<i>CCAP1L</i>	<i>DATA</i>	<i>0EBH</i>	
<i>CCAPIH</i>	<i>DATA</i>	<i>0FBH</i>	
<i>PCA_PWM1</i>	<i>DATA</i>	<i>0F3H</i>	
<i>CCAPM2</i>	<i>DATA</i>	<i>0DCH</i>	
<i>CCAP2L</i>	<i>DATA</i>	<i>0ECH</i>	
<i>CCAP2H</i>	<i>DATA</i>	<i>0FCH</i>	
<i>PCA_PWM2</i>	<i>DATA</i>	<i>0F4H</i>	
<i>CNT</i>	<i>DATA</i>	<i>20H</i>	
<i>COUNT0</i>	<i>DATA</i>	<i>21H</i>	<i>;3 bytes</i>
<i>COUNT1</i>	<i>DATA</i>	<i>24H</i>	<i>;3 bytes</i>
<i>LENGTH</i>	<i>DATA</i>	<i>27H</i>	<i>;3 bytes, (COUNT1-COUNT0)</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
<i>ORG</i>		<i>0000H</i>	
<i>LJMP</i>		<i>START</i>	
<i>ORG</i>		<i>003BH</i>	
<i>LJMP</i>		<i>PCAiSR</i>	
<i>ORG</i>		<i>0100H</i>	
PCAiSR:			
<i>PUSH</i>		<i>ACC</i>	
<i>PUSH</i>		<i>PSW</i>	
<i>JNB</i>		<i>CF,CHECKCCF0</i>	
<i>CLR</i>		<i>CF</i>	<i>;清中断标志</i>
<i>INC</i>		<i>CNT</i>	<i>; PCA 计时溢出次数+1</i>
CHECKCCF0:			
<i>JNB</i>		<i>CCF0,ISREXIT</i>	
<i>CLR</i>		<i>CCF0</i>	
<i>MOV</i>		<i>COUNT0,COUNT1</i>	<i>;备份上一次的捕获值</i>
<i>MOV</i>		<i>COUNT0+1,COUNT1+1</i>	
<i>MOV</i>		<i>COUNT0+2,COUNT1+2</i>	
<i>MOV</i>		<i>COUNT1,CNT</i>	<i>;保存本次的捕获值</i>
<i>MOV</i>		<i>COUNT1+1,CCAP0H</i>	
<i>MOV</i>		<i>COUNT1+2,CCAP0L</i>	
<i>CLR</i>		<i>C</i>	<i>;计算两次的捕获差值</i>
<i>MOV</i>		<i>A,COUNT1+2</i>	
<i>SUBB</i>		<i>A,COUNT0+2</i>	
<i>MOV</i>		<i>LENGTH+2,A</i>	

```

MOV      A,COUNT1+1
SUBB    A,COUNT0+1
MOV      LENGTH+1,A
MOV      A,COUNT1
SUBB    A,COUNT0
MOV      LENGTH,A           ;LENGTH 保存的即为捕获的脉冲宽度

ISREXIT:
POP      PSW
POP      ACC
RETI

START:
MOV      SP,#5FH
MOV      P0M0,#00H
MOV      P0M1,#00H
MOV      P1M0,#00H
MOV      P1M1,#00H
MOV      P2M0,#00H
MOV      P2M1,#00H
MOV      P3M0,#00H
MOV      P3M1,#00H
MOV      P4M0,#00H
MOV      P4M1,#00H
MOV      P5M0,#00H
MOV      P5M1,#00H

CLR      A
MOV      CNT,A             ;用户变量初始化
MOV      COUNT0,A
MOV      COUNT0+1,A
MOV      COUNT0+2,A
MOV      COUNT1,A
MOV      COUNT1+1,A
MOV      COUNT1+2,A
MOV      LENGTH,A
MOV      LENGTH+1,A
MOV      LENGTH+2,A

MOV      CCON,#00H
MOV      CMOD,#09H          ;PCA 时钟为系统时钟,使能PCA 计时中断
MOV      CL,#00H
MOV      CH,#0H
MOV      CCAPM0,#11H         ;PCA 模块0 为16 位捕获模式 (下降沿捕获)
;                   ;PCA 模块0 为16 位捕获模式 (上升沿捕获)
;                   ;PCA 模块0 为16 位捕获模式 (边沿捕获)
MOV      CCAPM0,#21H
MOV      CCAPM0,#31H
MOV      CCAP0L,#00H
MOV      CCAP0H,#00H
SETB    CR                  ;启动PCA 计时器
SETB    EA

JMP      $

END

```

22.5.3 PCA 实现 16 位软件定时

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "reg51.h"
#include "intrins.h"

#define T50HZ (11059200L / 12 / 2 / 50)

sfr CCON = 0xd8;
sbit CF = CCON^7;
sbit CR = CCON^6;
sbit CCF2 = CCON^2;
sbit CCF1 = CCON^1;
sbit CCF0 = CCON^0;
sfr CMOD = 0xd9;
sfr CL = 0xe9;
sfr CH = 0xf9;
sfr CCAPM0 = 0xda;
sfr CCAP0L = 0xea;
sfr CCAP0H = 0xfa;
sfr PCA_PWM0 = 0xf2;
sfr CCAPM1 = 0xdb;
sfr CCAP1L = 0xeb;
sfr CCAP1H = 0xfb;
sfr PCA_PWM1 = 0xf3;
sfr CCAPM2 = 0xdc;
sfr CCAP2L = 0xec;
sfr CCAP2H = 0xfc;
sfr PCA_PWM2 = 0xf4;

sfr P0M1 = 0x93;
sfr P0M0 = 0x94;
sfr P1M1 = 0x91;
sfr P1M0 = 0x92;
sfr P2M1 = 0x95;
sfr P2M0 = 0x96;
sfr P3M1 = 0xb1;
sfr P3M0 = 0xb2;
sfr P4M1 = 0xb3;
sfr P4M0 = 0xb4;
sfr P5M1 = 0xc9;
sfr P5M0 = 0xca;

sbit P10 = P1^0;

unsigned int value;

void PCA_Isr() interrupt 7
{
    CCF0 = 0;
    CCAP0L = value;
    CCAP0H = value >> 8;
    value += T50HZ;

    P10 = !P10; // 测试端口
}

void main()
{
```

```

P0M0 = 0x00;
P0M1 = 0x00;
P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

CCON = 0x00;
CMOD = 0x00; //PCA 时钟为系统时钟/12
CL = 0x00;
CH = 0x00;
CCAPM0 = 0x49; //PCA 模块0 为16 位定时器模式
value = T50HZ;
CCAP0L = value;
CCAP0H = value >> 8;
value += T50HZ;
CR = 1; //启动PCA 计时器
EA = 1;

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>CCON</i>	<i>DATA</i>	<i>0D8H</i>
<i>CF</i>	<i>BIT</i>	<i>CCON.7</i>
<i>CR</i>	<i>BIT</i>	<i>CCON.6</i>
<i>CCF2</i>	<i>BIT</i>	<i>CCON.2</i>
<i>CCF1</i>	<i>BIT</i>	<i>CCON.1</i>
<i>CCF0</i>	<i>BIT</i>	<i>CCON.0</i>
<i>CMOD</i>	<i>DATA</i>	<i>0D9H</i>
<i>CL</i>	<i>DATA</i>	<i>0E9H</i>
<i>CH</i>	<i>DATA</i>	<i>0F9H</i>
<i>CCAPM0</i>	<i>DATA</i>	<i>0DAH</i>
<i>CCAP0L</i>	<i>DATA</i>	<i>0EAH</i>
<i>CCAP0H</i>	<i>DATA</i>	<i>0FAH</i>
<i>PCA_PWM0</i>	<i>DATA</i>	<i>0F2H</i>
<i>CCAPM1</i>	<i>DATA</i>	<i>0DBH</i>
<i>CCAP1L</i>	<i>DATA</i>	<i>0EBH</i>
<i>CCAPIH</i>	<i>DATA</i>	<i>0FBH</i>
<i>PCA_PWM1</i>	<i>DATA</i>	<i>0F3H</i>
<i>CCAPM2</i>	<i>DATA</i>	<i>0DCH</i>
<i>CCAP2L</i>	<i>DATA</i>	<i>0ECH</i>
<i>CCAP2H</i>	<i>DATA</i>	<i>0FCB</i>
<i>PCA_PWM2</i>	<i>DATA</i>	<i>0F4H</i>
<i>T50HZ</i>	<i>EQU</i>	<i>2400H</i>
		<i>;11059200/12/2/50</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>

<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>

<i>ORG</i>	<i>0000H</i>
<i>LJMP</i>	<i>START</i>
<i>ORG</i>	<i>003BH</i>
<i>LJMP</i>	<i>PCAI SR</i>

<i>ORG</i>	<i>0100H</i>
------------	--------------

PCAI SR:

<i>PUSH</i>	<i>ACC</i>
<i>PUSH</i>	<i>PSW</i>
<i>CLR</i>	<i>CCF0</i>
<i>MOV</i>	<i>A,CCAP0L</i>
<i>ADD</i>	<i>A,#LOW T50HZ</i>
<i>MOV</i>	<i>CCAP0L,A</i>
<i>MOV</i>	<i>A,CCAP0H</i>
<i>ADDC</i>	<i>A,#HIGH T50HZ</i>
<i>MOV</i>	<i>CCAP0H,A</i>
<i>CPL</i>	<i>PI.0</i>
	; 测试端口, 闪烁频率为 50Hz
<i>POP</i>	<i>PSW</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

START:

<i>MOV</i>	<i>SP, #5FH</i>
<i>MOV</i>	<i>P0M0, #00H</i>
<i>MOV</i>	<i>P0M1, #00H</i>
<i>MOV</i>	<i>P1M0, #00H</i>
<i>MOV</i>	<i>P1M1, #00H</i>
<i>MOV</i>	<i>P2M0, #00H</i>
<i>MOV</i>	<i>P2M1, #00H</i>
<i>MOV</i>	<i>P3M0, #00H</i>
<i>MOV</i>	<i>P3M1, #00H</i>
<i>MOV</i>	<i>P4M0, #00H</i>
<i>MOV</i>	<i>P4M1, #00H</i>
<i>MOV</i>	<i>P5M0, #00H</i>
<i>MOV</i>	<i>P5M1, #00H</i>
<i>MOV</i>	<i>CCON,#00H</i>
<i>MOV</i>	<i>CMOD,#00H</i>
	; PCA 时钟为系统时钟/12
<i>MOV</i>	<i>CL,#00H</i>
<i>MOV</i>	<i>CH,#0H</i>
<i>MOV</i>	<i>CCAPM0,#49H</i>
	; PCA 模块 0 为 16 位定时器模式
<i>MOV</i>	<i>CCAP0L,#LOW T50HZ</i>
<i>MOV</i>	<i>CCAP0H,#HIGH T50HZ</i>
<i>SETB</i>	<i>CR</i>
	; 启动 PCA 计时器
<i>SETB</i>	<i>EA</i>
<i>JMP</i>	\$

END

22.5.4 PCA 实现 16 位软件定时 (ECI 外部时钟模式)

注意: 外部时钟频率不能高于系统频率的 1/2

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "reg51.h"
#include "intrins.h"

#define T50HZ (11059200L / 12 / 2 / 50)

sfr CCON = 0xd8;
sbit CF = CCON^7;
sbit CR = CCON^6;
sbit CCF2 = CCON^2;
sbit CCF1 = CCON^1;
sbit CCF0 = CCON^0;
sfr CMOD = 0xd9;
sfr CL = 0xe9;
sfr CH = 0xf9;
sfr CCAPM0 = 0xda;
sfr CCAP0L = 0xea;
sfr CCAP0H = 0xfa;
sfr PCA_PWM0 = 0xf2;
sfr CCAPM1 = 0xdb;
sfr CCAP1L = 0xeb;
sfr CCAP1H = 0xfb;
sfr PCA_PWM1 = 0xf3;
sfr CCAPM2 = 0xdc;
sfr CCAP2L = 0xec;
sfr CCAP2H = 0xfc;
sfr PCA_PWM2 = 0xf4;

sfr P0M1 = 0x93;
sfr P0M0 = 0x94;
sfr P1M1 = 0x91;
sfr P1M0 = 0x92;
sfr P2M1 = 0x95;
sfr P2M0 = 0x96;
sfr P3M1 = 0xb1;
sfr P3M0 = 0xb2;
sfr P4M1 = 0xb3;
sfr P4M0 = 0xb4;
sfr P5M1 = 0xc9;
sfr P5M0 = 0xca;

sbit P10 = PI^0;

unsigned int value;

void PCA_Isr() interrupt 7
{
    CCF0 = 0;
    CCAP0L = value;
    CCAP0H = value >> 8;
```

```

value += T50HZ;

P10 = !P10;                                //测试端口
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CCON = 0x00;
    CMOD = 0x06;                            //PCA 时钟为从 ECI 端口输入的外部时钟
    CL = 0x00;
    CH = 0x00;
    CCAPM0 = 0x49;                          //PCA 模块 0 为 16 位定时器模式
    value = T50HZ;
    CCAP0L = value;
    CCAP0H = value >> 8;
    value += T50HZ;
    CR = 1;                                //启动 PCA 计时器
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

CCON	DATA	0D8H
CF	BIT	CCON.7
CR	BIT	CCON.6
CCF2	BIT	CCON.2
CCF1	BIT	CCON.1
CCF0	BIT	CCON.0
CMOD	DATA	0D9H
CL	DATA	0E9H
CH	DATA	0F9H
CCAPM0	DATA	0DAH
CCAP0L	DATA	0EAH
CCAP0H	DATA	0FAH
PCA_PWM0	DATA	0F2H
CCAPMI	DATA	0DBH
CCAP1L	DATA	0EBH
CCAP1H	DATA	0FBH
PCA_PWM1	DATA	0F3H
CCAPM2	DATA	0DCH
CCAP2L	DATA	0ECH
CCAP2H	DATA	0FCH

<i>PCA_PWM2</i>	<i>DATA</i>	<i>0F4H</i>	
<i>T50HZ</i>	<i>EQU</i>	<i>2400H</i>	<i>;11059200/12/2/50</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>003BH</i>	
	<i>LJMP</i>	<i>PCAI SR</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>PCAI SR:</i>			
	<i>PUSH</i>	<i>ACC</i>	
	<i>PUSH</i>	<i>PSW</i>	
	<i>CLR</i>	<i>CCF0</i>	
	<i>MOV</i>	<i>A,CCAP0L</i>	
	<i>ADD</i>	<i>A,#LOW T50HZ</i>	
	<i>MOV</i>	<i>CCAP0L,A</i>	
	<i>MOV</i>	<i>A,CCAP0H</i>	
	<i>ADDC</i>	<i>A,#HIGH T50HZ</i>	
	<i>MOV</i>	<i>CCAP0H,A</i>	
	<i>CPL</i>	<i>PI.0</i>	<i>; 测试端口, 闪烁频率为 50Hz</i>
	<i>POP</i>	<i>PSW</i>	
	<i>POP</i>	<i>ACC</i>	
	<i>RETI</i>		
<i>START:</i>			
	<i>MOV</i>	<i>SP, #5FH</i>	
	<i>MOV</i>	<i>P0M0, #00H</i>	
	<i>MOV</i>	<i>P0M1, #00H</i>	
	<i>MOV</i>	<i>P1M0, #00H</i>	
	<i>MOV</i>	<i>P1M1, #00H</i>	
	<i>MOV</i>	<i>P2M0, #00H</i>	
	<i>MOV</i>	<i>P2M1, #00H</i>	
	<i>MOV</i>	<i>P3M0, #00H</i>	
	<i>MOV</i>	<i>P3M1, #00H</i>	
	<i>MOV</i>	<i>P4M0, #00H</i>	
	<i>MOV</i>	<i>P4M1, #00H</i>	
	<i>MOV</i>	<i>P5M0, #00H</i>	
	<i>MOV</i>	<i>P5M1, #00H</i>	
	<i>MOV</i>	<i>CCON,#00H</i>	
	<i>MOV</i>	<i>CMOD,#06H</i>	<i>; PCA 时钟为从 ECI 端口输入的外部时钟</i>
	<i>MOV</i>	<i>CL,#00H</i>	
	<i>MOV</i>	<i>CH,#0H</i>	
	<i>MOV</i>	<i>CCAPM0,#49H</i>	<i>; PCA 模块 0 为 16 位定时器模式</i>
	<i>MOV</i>	<i>CCAP0L,#LOW T50HZ</i>	

```

MOV      CCAP0H,#HIGH T50HZ
SETB    CR          ;启动 PCA 计时器
SETB    EA

JMP      $

END

```

22.5.5 PCA 输出高速脉冲

C 语言代码

//测试工作频率为11.0592MHz

```

#include "reg51.h"
#include "intrins.h"

#define T38K4HZ      (11059200L / 2 / 38400)

sfr CCON      = 0xd8;
sbit CF        = CCON^7;
sbit CR        = CCON^6;
sbit CCF2      = CCON^2;
sbit CCF1      = CCON^1;
sbit CCF0      = CCON^0;
sfr CMOD      = 0xd9;
sfr CL         = 0xe9;
sfr CH         = 0xf9;
sfr CCAPM0    = 0xda;
sfr CCAP0L    = 0xea;
sfr CCAP0H    = 0xfa;
sfr PCA_PWM0  = 0xf2;
sfr CCAPMI    = 0xdb;
sfr CCAP1L    = 0xeb;
sfr CCAP1H    = 0xfb;
sfr PCA_PWM1  = 0xf3;
sfr CCAPM2    = 0xdc;
sfr CCAP2L    = 0xec;
sfr CCAP2H    = 0xfc;
sfr PCA_PWM2  = 0xf4;

sfr P0M1      = 0x93;
sfr P0M0      = 0x94;
sfr P1M1      = 0x91;
sfr P1M0      = 0x92;
sfr P2M1      = 0x95;
sfr P2M0      = 0x96;
sfr P3M1      = 0xb1;
sfr P3M0      = 0xb2;
sfr P4M1      = 0xb3;
sfr P4M0      = 0xb4;
sfr P5M1      = 0xc9;
sfr P5M0      = 0xca;

unsigned int value;

```

```

void PCA_Isr() interrupt 7
{
    CCF0 = 0;
    CCAP0L = value;
    CCAP0H = value >> 8;
    value += T38K4HZ;
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CCON = 0x00;
    CMOD = 0x08; //PCA 时钟为系统时钟
    CL = 0x00;
    CH = 0x00;
    CCAPM0 = 0x4d; //PCA 模块0 为16 位定时器模式并使能脉冲输出
    value = T38K4HZ;
    CCAP0L = value;
    CCAP0H = value >> 8;
    value += T38K4HZ;
    CR = 1; //启动PCA 计时器
    EA = 1;

    while (1);
}

```

汇编代码

;测试工作频率为 11.0592MHz

<i>CCON</i>	<i>DATA</i>	<i>0D8H</i>
<i>CF</i>	<i>BIT</i>	<i>CCON.7</i>
<i>CR</i>	<i>BIT</i>	<i>CCON.6</i>
<i>CCF2</i>	<i>BIT</i>	<i>CCON.2</i>
<i>CCF1</i>	<i>BIT</i>	<i>CCON.1</i>
<i>CCF0</i>	<i>BIT</i>	<i>CCON.0</i>
<i>CMOD</i>	<i>DATA</i>	<i>0D9H</i>
<i>CL</i>	<i>DATA</i>	<i>0E9H</i>
<i>CH</i>	<i>DATA</i>	<i>0F9H</i>
<i>CCAPM0</i>	<i>DATA</i>	<i>0DAH</i>
<i>CCAP0L</i>	<i>DATA</i>	<i>0EAH</i>
<i>CCAP0H</i>	<i>DATA</i>	<i>0FAH</i>
<i>PCA_PWM0</i>	<i>DATA</i>	<i>0F2H</i>
<i>CCAPMI</i>	<i>DATA</i>	<i>0DBH</i>
<i>CCAPIL</i>	<i>DATA</i>	<i>0EBH</i>
<i>CCAPIH</i>	<i>DATA</i>	<i>0FBH</i>
<i>PCA_PWM1</i>	<i>DATA</i>	<i>0F3H</i>

<i>CCAPM2</i>	<i>DATA</i>	<i>0DCH</i>	
<i>CCAP2L</i>	<i>DATA</i>	<i>0ECH</i>	
<i>CCAP2H</i>	<i>DATA</i>	<i>0FCH</i>	
<i>PCA_PWM2</i>	<i>DATA</i>	<i>0F4H</i>	
<i>T38K4HZ</i>	<i>EQU</i>	<i>90H</i>	<i>;11059200/2/38400</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
<i>ORG</i>		<i>0000H</i>	
<i>LJMP</i>		<i>START</i>	
<i>ORG</i>		<i>003BH</i>	
<i>LJMP</i>		<i>PCAI SR</i>	
<i>ORG</i>		<i>0100H</i>	
PCAI SR:			
<i>PUSH</i>		<i>ACC</i>	
<i>PUSH</i>		<i>PSW</i>	
<i>CLR</i>		<i>CCF0</i>	
<i>MOV</i>		<i>A,CCAP0L</i>	
<i>ADD</i>		<i>A,#LOW T38K4HZ</i>	
<i>MOV</i>		<i>CCAP0L,A</i>	
<i>MOV</i>		<i>A,CCAP0H</i>	
<i>ADDC</i>		<i>A,#HIGH T38K4HZ</i>	
<i>MOV</i>		<i>CCAP0H,A</i>	
<i>POP</i>		<i>PSW</i>	
<i>POP</i>		<i>ACC</i>	
<i>RETI</i>			
START:			
<i>MOV</i>		<i>SP, #5FH</i>	
<i>MOV</i>		<i>P0M0, #00H</i>	
<i>MOV</i>		<i>P0M1, #00H</i>	
<i>MOV</i>		<i>P1M0, #00H</i>	
<i>MOV</i>		<i>P1M1, #00H</i>	
<i>MOV</i>		<i>P2M0, #00H</i>	
<i>MOV</i>		<i>P2M1, #00H</i>	
<i>MOV</i>		<i>P3M0, #00H</i>	
<i>MOV</i>		<i>P3M1, #00H</i>	
<i>MOV</i>		<i>P4M0, #00H</i>	
<i>MOV</i>		<i>P4M1, #00H</i>	
<i>MOV</i>		<i>P5M0, #00H</i>	
<i>MOV</i>		<i>P5M1, #00H</i>	
<i>MOV</i>		<i>CCON,#00H</i>	
<i>MOV</i>		<i>CMOD,#08H</i>	<i>;PCA 时钟为系统时钟</i>
<i>MOV</i>		<i>CL,#00H</i>	
<i>MOV</i>		<i>CH,#0H</i>	

```

MOV      CCAPM0,#4DH          ;PCA 模块0为16位定时器模式并使能脉冲输出
MOV      CCAP0L,#LOW T38K4HZ
MOV      CCAP0H,#HIGH T38K4HZ
SETB    CR                   ;启动PCA 计时器
SETB    EA
JMP     $

END

```

22.5.6 PCA 扩展外部中断

C 语言代码

//测试工作频率为11.0592MHz

```

#include "reg51.h"
#include "intrins.h"

sfr    CCON      = 0xd8;
sbit   CF        = CCON^7;
sbit   CR        = CCON^6;
sbit   CCF2      = CCON^2;
sbit   CCF1      = CCON^1;
sbit   CCF0      = CCON^0;
sfr    CMOD      = 0xd9;
sfr    CL        = 0xe9;
sfr    CH        = 0xf9;
sfr    CCAPM0    = 0xda;
sfr    CCAP0L    = 0xea;
sfr    CCAP0H    = 0xfa;
sfr    PCA_PWM0  = 0xf2;
sfr    CCAPMI    = 0xdb;
sfr    CCAP1L    = 0xeb;
sfr    CCAP1H    = 0xfb;
sfr    PCA_PWM1  = 0xf3;
sfr    CCAPM2    = 0xdc;
sfr    CCAP2L    = 0xec;
sfr    CCAP2H    = 0xfc;
sfr    PCA_PWM2  = 0xf4;

sfr    P0M1      = 0x93;
sfr    P0M0      = 0x94;
sfr    P1M1      = 0x91;
sfr    P1M0      = 0x92;
sfr    P2M1      = 0x95;
sfr    P2M0      = 0x96;
sfr    P3M1      = 0xb1;
sfr    P3M0      = 0xb2;
sfr    P4M1      = 0xb3;
sfr    P4M0      = 0xb4;
sfr    P5M1      = 0xc9;
sfr    P5M0      = 0xca;

sbit  P10       = P1^0;

```

```

void PCA_Isr() interrupt 7
{
    CCF0 = 0;
    P10 = !P10;
}

void main()
{
    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    CCON = 0x00;
    CMOD = 0x08;                                //PCA 时钟为系统时钟
    CL = 0x00;
    CH = 0x00;
    CCAPM0 = 0x11;                               //扩展外部端口 CCP0 为下降沿中断口
//    CCAPM0 = 0x21;                             //扩展外部端口 CCP0 为上升沿中断口
//    CCAPM0 = 0x31;                             //扩展外部端口 CCP0 为边沿中断口
    CCAP0L = 0;
    CCAP0H = 0;
    CR = 1;                                     //启动 PCA 计时器
    EA = 1;

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

CCON	DATA	0D8H
CF	BIT	CCON.7
CR	BIT	CCON.6
CCF2	BIT	CCON.2
CCF1	BIT	CCON.1
CCF0	BIT	CCON.0
CMOD	DATA	0D9H
CL	DATA	0E9H
CH	DATA	0F9H
CCAPM0	DATA	0DAH
CCAP0L	DATA	0EAH
CCAP0H	DATA	0FAH
PCA_PWM0	DATA	0F2H
CCAPM1	DATA	0DBH
CCAP1L	DATA	0EBH
CCAP1H	DATA	0FBH
PCA_PWM1	DATA	0F3H
CCAPM2	DATA	0DCH
CCAP2L	DATA	0ECH

CCAP2H DATA *0FCH*
PCA_PWM2 DATA *0F4H*

P0M1 DATA *093H*
P0M0 DATA *094H*
P1M1 DATA *091H*
P1M0 DATA *092H*
P2M1 DATA *095H*
P2M0 DATA *096H*
P3M1 DATA *0B1H*
P3M0 DATA *0B2H*
P4M1 DATA *0B3H*
P4M0 DATA *0B4H*
P5M1 DATA *0C9H*
P5M0 DATA *0CAH*

ORG *0000H*
LJMP START
ORG *003BH*
LJMP PCAiSR

ORG *0100H*
PCAiSR:
CLR CCF0
CPL PI.0
RETI

START:

MOV SP, #5FH
MOV P0M0, #00H
MOV P0M1, #00H
MOV P1M0, #00H
MOV P1M1, #00H
MOV P2M0, #00H
MOV P2M1, #00H
MOV P3M0, #00H
MOV P3M1, #00H
MOV P4M0, #00H
MOV P4M1, #00H
MOV P5M0, #00H
MOV P5M1, #00H

MOV CCON, #00H
MOV CMOD, #08H ;PCA 时钟为系统时钟
MOV CL, #00H
MOV CH, #0H
MOV CCAPM0, #11H ;扩展外部端口 CCP0 为下降沿中断口
; MOV CCAPM0, #21H ;扩展外部端口 CCP0 为上升沿中断口
; MOV CCAPM0, #31H ;扩展外部端口 CCP0 为边沿中断口
MOV CCAP0L, #0
MOV CCAP0H, #0
SETB CR ;启动 PCA 计时器
SETB EA

JMP \$

END

23 同步串行外设接口 SPI

产品线	SPI	快速 SPI	接收超时中断	MISO 和 MOSI 可切换
STC8H1K08 系列	●			
STC8H1K28 系列	●			
STC8H3K64S4 系列 A 版本	●			
STC8H3K64S4 系列 A 版本	●			
STC8H3K64S2 系列 B 版本		●		
STC8H3K64S4 系列 B 版本		●		
STC8H8K64U 系列 A 版本	●			
STC8H8K64U 系列 B/C/D 版本		●		
STC8H4K64TL 系列		●		
STC8H4K64TLCD 系列		●		
STC8H1K08T 系列		●		
STC8H2K12U-A/B 系列		●	●	●
STC8H2K32U 系列		●	●	●
STC8G1K08-SOP8 系列	●			
STC8G1K08A-SOP8 系列	●			

STC8H 系列单片机内部集成了一种高速串行通信接口——SPI 接口。SPI 是一种全双工的高速同步通信总线。STC8H 系列集成的 SPI 接口提供了两种操作模式：主模式和从模式。

23.1 SPI 功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW1	A2H	S1_S[1:0]	-	-	SPI_S[1:0]	0	-		

SPI_S[1:0]: SPI 功能脚选择位

SPI_S[1:0]	SS	MOSI	MISO	SCLK
00	P1.2/P5.4 ^[1]	P1.3	P1.4	P1.5
01	P2.2	P2.3	P2.4	P2.5
10	P5.4	P4.0	P4.1	P4.3
11	P3.5	P3.4	P3.3	P3.2

注^[1]：对于部分没有 P1.2 口的单片机型号，此功能在 P5.4 口上

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
HSSPI_CFG2	FBF9H	-	IOSW	-	-	-	-	-	-

IOSW: 交换 MOSI 和 MISO 脚位

0: 不交换，维持上电默认脚位。

1: 交换 MOSI 和 MISO 的脚位。

23.2 SPI 相关的寄存器

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
SPSTAT	SPI 状态寄存器	CDH	SPIF	WCOL	-	-	-	-	-	-	00xx,xxxx
SPCTL	SPI 控制寄存器	CEH	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR[1:0]		0000,0100
SPDAT	SPI 数据寄存器	CFH									0000,0000

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
SPITOCSR	SPI 从机超时控制寄存器	FD80H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx	
SPITOSR	SPI 从机超时状态寄存器	FD81H	CTOIF	-	-	-	-	-	-	-	TOIF	
SPITOTH	SPI 从机超时长度寄存器	FD82H				TM[15:8]						0000,0000
SPITOTL	SPI 从机超时长度寄存器	FD83H				TM[7:0]						0000,0000
SPITOTE	SPI 从机超时长度寄存器	FD8CH				TM[23:16]						0000,0000
HSSPI_CFG2	高速 SPI 配置寄存器 2	FBF9H	-	IOSW	-	-	-	-	-	-	-	x0xx,xxxx

23.2.1 SPI 状态寄存器 (SPSTAT)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPSTAT	CDH	SPIF	WCOL	-	-	-	-	-	-

SPIF: SPI 中断标志位。

当发送/接收完成 1 字节的数据后, 硬件自动将此位置 1, 并向 CPU 提出中断请求。当 SPI 处于主机模式且 SSIG 位被设置为 0 时, 如果 SS 管脚的输入电平被驱动为低电平时, 此标志位也会被硬件自动置 1, 以标志设备模式发生变化。

注意: 此标志位必须用户通过软件方式向此位写 1 进行清零。

WCOL: SPI 写冲突标志位。

当 SPI 在进行数据传输的过程中写 SPDAT 寄存器时, 硬件将此位置 1。

注意: 此标志位必须用户通过软件方式向此位写 1 进行清零。

23.2.2 SPI 控制寄存器 (SPCTL), SPI 速度控制

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPCTL	CEH	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR[1:0]	

SSIG: SS 引脚功能控制位

0: SS 引脚确定器件是主机还是从机

1: 忽略 SS 引脚功能, 使用 MSTR 确定器件是主机还是从机

SPEN: SPI 使能控制位

0: 关闭 SPI 功能

1: 使能 SPI 功能

DORD: SPI 数据位发送/接收的顺序

0: 先发送/接收数据的高位 (MSB)

1: 先发送/接收数据的低位 (LSB)

MSTR: 器件主/从模式选择位

设置主机模式:

若 SSIG=0, 则 SS 管脚必须为高电平且设置 MSTR 为 1

若 SSIG=1, 则只需要设置 MSTR 为 1 (忽略 SS 管脚的电平)

设置从机模式:

若 SSIG=0, 则 SS 管脚必须为低电平 (与 MSTR 位无关)

若 SSIG=1, 则只需要设置 MSTR 为 0 (忽略 SS 管脚的电平)

CPOL: SPI 时钟极性控制

0: SCLK 空闲时为低电平, SCLK 的前时钟沿为上升沿, 后时钟沿为下降沿

1: SCLK 空闲时为高电平, SCLK 的前时钟沿为下降沿, 后时钟沿为上升沿

CPHA: SPI 时钟相位控制

0: 数据 SS 管脚为低电平驱动第一位数据并在 SCLK 的后时钟沿改变数据, 前时钟沿采样数据 (必须 SSIG=0)

1: 数据在 SCLK 的前时钟沿驱动, 后时钟沿采样

SPR[1:0]: SPI 时钟频率选择

SPR[1:0]	SCLK 频率	快速 SPI 的 SCLK 频率
00	SYSlk/4	SYSlk/4
01	SYSlk/8	SYSlk/8
10	SYSlk/16	SYSlk/16
11	SYSlk/32	SYSlk/2

23.2.3 SPI 数据寄存器 (SPDAT)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPDAT	CFH								

SPI 发送/接收数据缓冲器。

23.2.4 SPI 从机超时控制寄存器 (SPITOCSR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPITOCSR	FD80H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTO: SPI 从机超时功能控制位

0: 禁止 SPI 从机超时功能

1: 使能 SPI 从机超时功能 (注: SPI 的主机模式无接收超时功能)

ENTOI: SPI 从机超时中断控制位

0: 禁止 SPI 从机超时中断

1: 使能 SPI 从机超时中断

SCALE: SPI 从机计数时钟源选择

0: 1us 时钟 (1MHz 时钟)。(注意: 如需要使用此时钟源, 用户必须先正确设置 IAP_TPS 寄存器。

例如: 若系统时钟为 12MHz, 则需要将 IAP_TPS 设置为 12; 若系统时钟为 22.1184MHz, 则需要将 IAP_TPS 设置为 22; 其他频率以此类推。特别注意, 此 1us 的时钟并不是精准时间)

1: 系统时钟

23.2.5 SPI 从机超时状态寄存器 (SPITOSR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPITOSR	FD81H	CTOIF	-	-	-	-	-	-	TOIF

CTOIF: 写“1”清除超时中断标志位 TOIF。(只写)

TOIF: SPI 超时中断请求标志位。(只读)

当发生 SPI 超时时, TOIF 会被硬件置“1”。如果 ENTOI 为 1 时会产生 SPI 中断, 中断入口地址为原 SPI 的中断入口地址。标志位需要软件向 CTOIF 位写“1”清零

23.2.6 SPI 从机超时长度控制寄存器 (SPITOTE/H/L)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
SPITOTE	FD8CH				TM[23:16]				
SPITOTH	FD82H				TM[15:8]				
SPITOTL	FD83H				TM[7:0]				

TM[23:0]: SPI 超时时间控制位。

当从机模式的 SPI 处于空闲状态时 (未检测到来自于主机的 SPI 时钟信号), 内部超时计数器根据 SCALE 寄存器所选择的时钟源进行计数, 当计数时间达到 TM 所设置的超时时间时, 便会产生超时中断。当 SPI 数据传输完成时, 复位内部超时计数器, 重新进行超时计数。

注:

- 1、如果需要使能接收超时中断功能, 则 TM 不可设置为 0, 且 SPITOTL、SPITOTH、SPITOTE 寄存器的设置必须先设置 SPITOTL 和 SPITOTH, 最后设置 SPITOTE
- 2、只有 SPI 从机模式才有接收超时功能
- 3、接收超时功能必须在从机正确接收到一字节数据后才能触发
- 4、正在接收过程中, 无接收超时功能

23.3 SPI 通信方式

SPI 的通信方式通常有 3 种：单主单从（一个主机设备连接一个从机设备）、互为主从（两个设备连接，设备和互为主机和从机）、单主多从（一个主机设备连接多个从机设备）

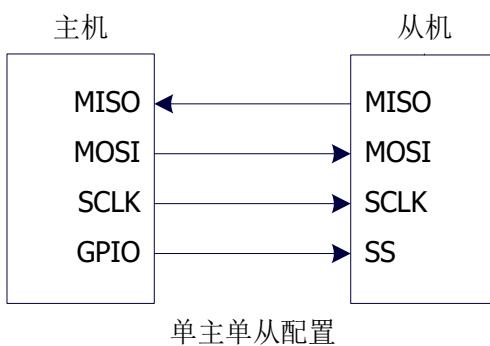
23.3.1 单主单从

两个设备相连，其中一个设备固定作为主机，另外一个固定作为从机。

主机设置：SSIG 设置为 1，MSTR 设置为 1，固定为主机模式。主机可以使用任意端口连接从机的 SS 管脚，拉低从机的 SS 脚即可使能从机

从机设置：SSIG 设置为 0，SS 管脚作为从机的片选信号。

单主单从连接配置图如下所示：



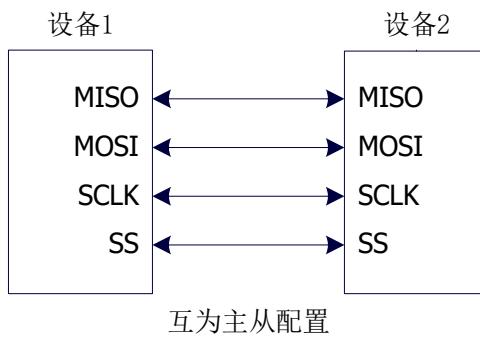
23.3.2 互为主从

两个设备相连，主机和从机不固定。

设置方法 1：两个设备初始化时都设置为 SSIG 设置为 0，MSTR 设置为 1，且将 SS 脚设置为双向口模式输出高电平。此时两个设备都是不忽略 SS 的主机模式。当其中一个设备需要启动传输时，可将自己的 SS 脚设置为输出模式并输出低电平，拉低对方的 SS 脚，这样另一个设备就被强行设置为从机模式了。

设置方法 2：两个设备初始化时都将自己设置成忽略 SS 的从机模式，即将 SSIG 设置为 1，MSTR 设置为 0。当其中一个设备需要启动传输时，先检测 SS 管脚的电平，如果时候高电平，就将自己设置成忽略 SS 的主模式，即可进行数据传输了。

互为主从连接配置图如下所示：



23.3.3 单主多从

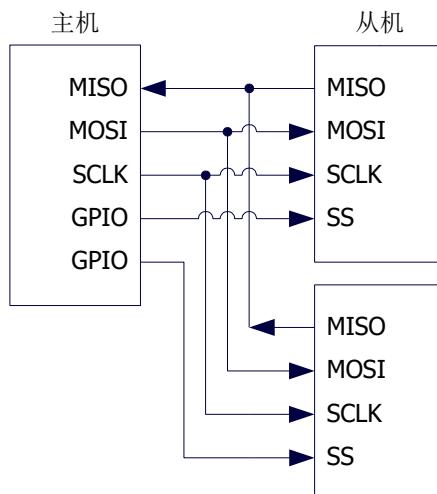
多个设备相连，其中一个设备固定作为主机，其他设备固定作为从机。

主机设置：SSIG 设置为 1，MSTR 设置为 1，固定为主机模式。主机可以使用任意端口分别连接各

个从机的 SS 管脚，拉低其中一个从机的 SS 脚即可使能相应的从机设备

从机设置：SSIG 设置为 0，SS 管脚作为从机的片选信号。

单主多从连接配置图如下所示：



单主多从配置

23.4 配置 SPI

控制位			通信端口				说明
SPEN	SSIG	MSTR	SS	MISO	MOSI	SCLK	
0	x	x	x	输入	输入	输入	关闭 SPI 功能, SS/MOSI/MISO/SCLK 均为普通 IO
1	0	0	0	输出	输入	输入	从机模式, 且被选中
1	0	0	1	高阻	输入	输入	从机模式, 但未被选中
1	0	1→0	0	输出	输入	输入	从机模式, 不忽略 SS 且 MSTR 为 1 的主机模式, 当 SS 管脚被拉低时, MSTR 将被硬件自动清零, 工作模式将被被动设置为从机模式
1	0	1	1	输入	高阻	高阻	主机模式, 空闲状态
					输出	输出	主机模式, 激活状态
1	1	0	x	输出	输入	输入	从机模式
1	1	1	x	输入	输出	输出	主机模式

从机模式的注意事项:

当 CPHA=0 时, SSIG 必须为 0 (即不能忽略 SS 脚)。在每次串行字节开始还发送前 SS 脚必须拉低, 并且在串行字节发送完后须重新设置为高电平。SS 管脚为低电平时不能对 SPDAT 寄存器执行写操作, 否则将导致一个写冲突错误。CPHA=0 且 SSIG=1 时的操作未定义。

当 CPHA=1 时, SSIG 可以置 1 (即可以忽略脚)。如果 SSIG=0, SS 脚可在连续传输之间保持低有效 (即一直固定为低电平)。这种方式适用于固定单主单从的系统。

主机模式的注意事项:

在 SPI 中, 传输总是由主机启动的。如果 SPI 使能 (SPEN=1) 并选择作为主机时, 主机对 SPI 数据寄存器 SPDAT 的写操作将启动 SPI 时钟发生器和数据的传输。在数据写入 SPDAT 之后的半个到一个 SPI 位时间后, 数据将出现在 MOSI 脚。写入主机 SPDAT 寄存器的数据从 MOSI 脚移出发送到从机的 MOSI 脚。同时从机 SPDAT 寄存器的数据从 MISO 脚移出发送到主机的 MISO 脚。

传输完一个字节后, SPI 时钟发生器停止, 传输完成标志 (SPIF) 置位, 如果 SPI 中断使能则会产生一个 SPI 中断。主机和从机 CPU 的两个移位寄存器可以看作是一个 16 位循环移位寄存器。当数据从主机移位传送到从机的同时, 数据也以相反的方向移入。这意味着在一个移位周期中, 主机和从机的数据相互交换。

通过 SS 改变模式

如果 SPEN=1, SSIG=0 且 MSTR=1, SPI 使能为主机模式, 并将 SS 脚可配置为输入模式化或准双向模式。这种情况下, 另外一个主机可将该脚驱动为低电平, 从而将该器件选择为 SPI 从机并向其发送数据。为了避免争夺总线, SPI 系统将该从机的 MSTR 清零, MOSI 和 SCLK 强制变为输入模式, 而 MISO 则变为输出模式, 同时 SPSTAT 的 SPIF 标志位置 1。

用户软件必须一直对 MSTR 位进行检测, 如果该位被一个从机选择动作而被动清零, 而用户想继续将 SPI 作为主机, 则必须重新设置 MSTR 位, 否则将一直处于从机模式。

写冲突

SPI 在发送时为单缓冲, 在接收时为双缓冲。这样在前一次发送尚未完成之前, 不能将新的数据写

入移位寄存器。当发送过程中对数据寄存器 SPDAT 进行写操作时，WCOL 位将被置 1 以指示发生数据写冲突错误。在这种情况下，当前发送的数据继续发送，而新写入的数据将丢失。

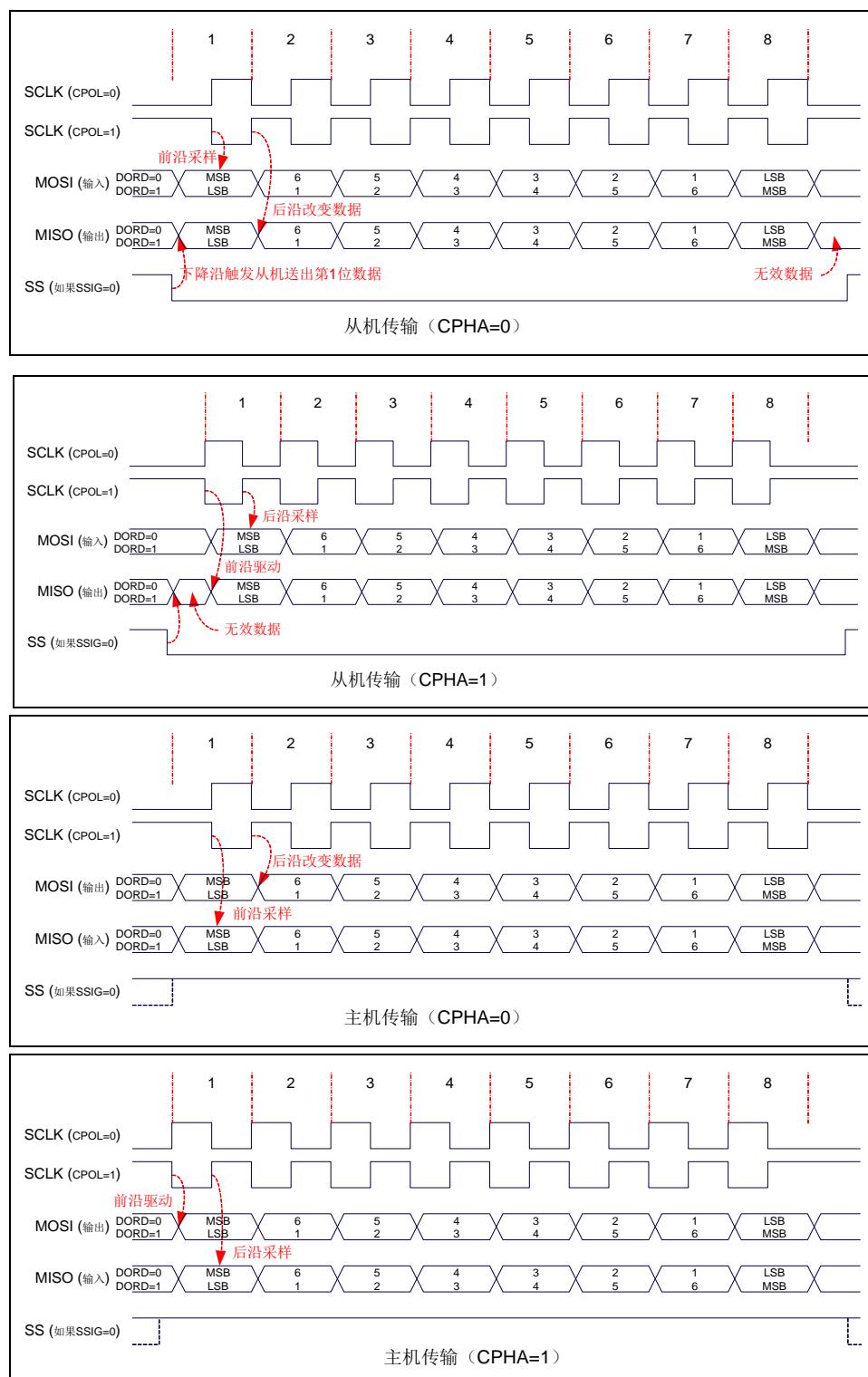
当对主机或从机进行写冲突检测时，主机发生写冲突的情况是很罕见的，因为主机拥有数据传输的完全控制权。但从机有可能发生写冲突，因为当主机启动传输时，从机无法进行控制。

接收数据时，接收到的数据传送到一个并行读数据缓冲区，这样将释放移位寄存器以进行下一个数据的接收。但必须在下个字符完全移入之前从数据寄存器中读出接收到的数据，否则，前一个接收数据将丢失。

WCOL 可通过软件向其写入“1”清零。

23.5 数据模式

SPI 的时钟相位控制位 CPHA 可以让用户设定数据采样和改变时的时钟沿。时钟极性位 CPOL 可以让用户设定时钟极性。下面图例显示了不同时钟相位、极性设置下 SPI 通讯时序。



23.6 范例程序

23.6.1 SPI 单主单从系统主机程序（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit SS = P1^0;
sbit LED = P1^1;

bit busy;

void SPI_Isr() interrupt 9
{
    SPSTAT = 0xc0; //清中断标志
    SS = 1; //拉高从机的 SS 管脚
    busy = 0;
    LED = !LED; //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    LED = 1;
    SS = 1;
    busy = 0;

    SPCTL = 0x50; //使能 SPI 主机模式
    SPSTAT = 0xc0; //清中断标志
    IE2 = ESPI; //使能 SPI 中断
    EA = 1;

    while (1)
    {
        while (busy);
        busy = 1;
        SS = 0; //拉低从机 SS 管脚
        SPDAT = 0x5a; //发送测试数据
    }
}
```

}

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>SPSTAT</i>	<i>DATA</i>	<i>0CDH</i>
<i>SPCTL</i>	<i>DATA</i>	<i>0CEH</i>
<i>SPDAT</i>	<i>DATA</i>	<i>0CFH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>ESPI</i>	<i>EQU</i>	<i>02H</i>
<i>BUSY</i>	<i>BIT</i>	<i>20H.0</i>
<i>SS</i>	<i>BIT</i>	<i>P1.0</i>
<i>LED</i>	<i>BIT</i>	<i>P1.1</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>004BH</i>
	<i>LJMP</i>	<i>SPIISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>SPIISR:</i>	 	
	<i>MOV</i>	<i>SPSTAT,#0C0H</i>
	<i>SETB</i>	<i>SS</i>
	<i>CLR</i>	<i>BUSY</i>
	<i>CPL</i>	<i>LED</i>
	<i>RETI</i>	
<i>START:</i>		
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>

<i>MOV</i>	<i>P5M1, #00H</i>	
<i>SETB</i>	<i>LED</i>	
<i>SETB</i>	<i>SS</i>	
<i>CLR</i>	<i>BUSY</i>	
<i>MOV</i>	<i>SPCTL,#50H</i>	;使能 SPI 主机模式
<i>MOV</i>	<i>SPSTAT,#0C0H</i>	;清中断标志
<i>MOV</i>	<i>IE2,#ESPI</i>	;使能 SPI 中断
<i>SETB</i>	<i>EA</i>	
 <i>LOOP:</i>		
<i>JB</i>	<i>BUSY,\$</i>	
<i>SETB</i>	<i>BUSY</i>	
<i>CLR</i>	<i>SS</i>	;拉低从机 SS 管脚
<i>MOV</i>	<i>SPDAT,#5AH</i>	;发送测试数据
<i>JMP</i>	<i>LOOP</i>	
 <i>END</i>		

23.6.2 SPI 单主单从系统从机程序（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit LED = P1^1;

void SPI_Isr() interrupt 9
{
    SPSTAT = 0xc0;           //清中断标志
    SPDAT = SPDAT;          //将接收到的数据回传给主机
    LED = !LED;              //测试端口
}

void main()
{
    P_SW2 |= 0x80;           //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    SPCTL = 0x40;            //使能 SPI 从机模式
```

```

SPSTAT = 0xc0;           //清中断标志
IE2 = ESPI;             //使能SPI 中断
EA = I;

while (1);
}

```

汇编代码

;*测试工作频率为 11.0592MHz*

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>SPSTAT</i>	<i>DATA</i>	<i>0CDH</i>
<i>SPCTL</i>	<i>DATA</i>	<i>0CEH</i>
<i>SPDAT</i>	<i>DATA</i>	<i>0CFH</i>
<i>IE2</i>	<i>DATA</i>	<i>0AFH</i>
<i>ESPI</i>	<i>EQU</i>	<i>02H</i>
<i>LED</i>	<i>BIT</i>	<i>P1.1</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>DATA</i>	<i>0000H</i>
<i>LJMP</i>	<i>DATA</i>	<i>START</i>
<i>ORG</i>	<i>DATA</i>	<i>004BH</i>
<i>LJMP</i>	<i>DATA</i>	<i>SPIISR</i>
<i>ORG</i>	<i>DATA</i>	<i>0100H</i>
<i>SPIISR:</i>	<i>MOV</i>	<i>SPSTAT,#0C0H</i> ;清中断标志
	<i>MOV</i>	<i>SPDAT,SPDAT</i> ;将接收到的数据回传给主机
	<i>CPL</i>	<i>LED</i>
	<i>RETI</i>	
<i>START:</i>	<i>MOV</i>	<i>SP,#5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i> ;使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>

```

MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      SPCTL,#40H          ;使能 SPI 从机模式
MOV      SPSTAT,#0C0H         ;清中断标志
MOV      IE2,#ESPI           ;使能 SPI 中断
SETB    EA

JMP      $

END

```

23.6.3 SPI 单主单从系统主机程序（查询方式）

C 语言代码

```

//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit    SS      =  P1^0;
sbit    LED     =  P1^1;

void main()
{
    P_SW2 |= 0x80;                      //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    LED = 1;
    SS = 1;

    SPCTL = 0x50;                      //使能 SPI 主机模式
    SPSTAT = 0xc0;                     //清中断标志

    while (1)
    {
        SS = 0;                         //拉低从机 SS 管脚
        SPDAT = 0x5a;                   //发送测试数据
        while (!(SPSTAT & 0x80));      //查询完成标志
        SPSTAT = 0xc0;                  //清中断标志
        SS = 1;                         //拉高从机的 SS 管脚
        LED = !LED;                    //测试端口
    }
}

```

```

    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH
SPSTAT     DATA      0CDH
SPCTL      DATA      0CEH
SPDAT      DATA      0CFH
IE2        DATA      0AFH
ESPI       EQU       02H

SS          BIT       P1.0
LED         BIT       P1.1

P1M1       DATA      091H
P1M0       DATA      092H
P0M1       DATA      093H
P0M0       DATA      094H
P2M1       DATA      095H
P2M0       DATA      096H
P3M1       DATA      0B1H
P3M0       DATA      0B2H
P4M1       DATA      0B3H
P4M0       DATA      0B4H
P5M1       DATA      0C9H
P5M0       DATA      0CAH

ORG         0000H
LJMP        START

ORG         0100H
START:
MOV         SP, #5FH
ORL         P_SW2, #80H           ; 使能访问 XFR，没有冲突不用关闭

MOV         P0M0, #00H
MOV         P0M1, #00H
MOV         P1M0, #00H
MOV         P1M1, #00H
MOV         P2M0, #00H
MOV         P2M1, #00H
MOV         P3M0, #00H
MOV         P3M1, #00H
MOV         P4M0, #00H
MOV         P4M1, #00H
MOV         P5M0, #00H
MOV         P5M1, #00H

SETB        LED
SETB        SS

MOV         SPCTL, #50H          ; 使能 SPI 主机模式
MOV         SPSTAT, #0C0H        ; 清中断标志

LOOP:

```

<i>CLR</i>	<i>SS</i>	; 拉低从机 SS 管脚
<i>MOV</i>	<i>SPDAT,#5AH</i>	; 发送测试数据
<i>MOV</i>	<i>A,SPSTAT</i>	; 查询完成标志
<i>JNB</i>	<i>ACC.7,\$-2</i>	
<i>MOV</i>	<i>SPSTAT,#0C0H</i>	; 清中断标志
<i>SETB</i>	<i>SS</i>	
<i>CPL</i>	<i>LED</i>	
<i>JMP</i>	<i>LOOP</i>	
<i>END</i>		

23.6.4 SPI 单主单从系统从机程序（查询方式）

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

sbit LED = P1^1;

void SPI_Isr() interrupt 9
{
    SPSTAT = 0xc0; // 清中断标志
}

void main()
{
    P_SW2 |= 0x80; // 使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    SPCTL = 0x40; // 使能 SPI 从机模式
    SPSTAT = 0xc0; // 清中断标志

    while (1)
    {
        while (!(SPSTAT & 0x80)); // 查询完成标志
        SPSTAT = 0xc0; // 清中断标志
        SPDAT = SPDAT; // 将接收到的数据回传给主机
        LED = !LED; // 测试端口
    }
}
```

汇编代码

; 测试工作频率为 11.0592MHz

```

P_SW2      DATA      0BAH

SPSTAT     DATA      0CDH
SPCTL      DATA      0CEH
SPDAT      DATA      0CFH
IE2        DATA      0AFH
ESPI       EQU       02H

LED        BIT       P1.1

P1M1       DATA      091H
P1M0       DATA      092H
P0M1       DATA      093H
P0M0       DATA      094H
P2M1       DATA      095H
P2M0       DATA      096H
P3M1       DATA      0B1H
P3M0       DATA      0B2H
P4M1       DATA      0B3H
P4M0       DATA      0B4H
P5M1       DATA      0C9H
P5M0       DATA      0CAH

ORG        0000H
LJMP      START

ORG        0100H

START:
MOV        SP, #5FH
ORL        P_SW2,#80H           ; 使能访问 XFR，没有冲突不用关闭

MOV        P0M0, #00H
MOV        P0M1, #00H
MOV        P1M0, #00H
MOV        P1M1, #00H
MOV        P2M0, #00H
MOV        P2M1, #00H
MOV        P3M0, #00H
MOV        P3M1, #00H
MOV        P4M0, #00H
MOV        P4M1, #00H
MOV        P5M0, #00H
MOV        P5M1, #00H

MOV        SPCTL,#40H          ; 使能 SPI 从机模式
MOV        SPSTAT,#0C0H         ; 清中断标志

LOOP:
MOV        A,SPSTAT            ; 查询完成标志
JNB        ACC.7,$-2
MOV        SPSTAT,#0C0H         ; 清中断标志
MOV        SPDAT,SPDAT         ; 将接收到的数据回传给主机
CPL        LED
JMP        LOOP

```

END

23.6.5 SPI 互为主从系统程序（中断方式）

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit SS          = P1^0;
sbit LED         = P1^1;
sbit KEY         = P0^0;

void SPI_Isr() interrupt 9
{
    SPSTAT = 0xc0;                                //清中断标志
    if(SPCTL & 0x10)
    {
        SS = 1;                                    //主机模式
        //拉高从机的SS 管脚
        SPCTL = 0x40;                             //重新设置为从机待机
    }
    else
    {
        SPDAT = SPDAT;                           //从机模式
        //将接收到的数据回传给主机
    }
    LED = !LED;                                  //测试端口
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    LED = 1;
    KEY = 1;
    SS = 1;

    SPCTL = 0x40;                                //使能 SPI 从机模式进行待机
    SPSTAT = 0xc0;                                //清中断标志
    IE2 = ESPI;                                   //使能 SPI 中断
    EA = 1;
}
```

```

while (1)
{
    if (!KEY)                                //等待按键触发
    {
        SPCTL = 0x50;                         //使能 SPI 主机模式
        SS = 0;                                //拉低从机 SS 管脚
        SPDAT = 0x5a;                          //发送测试数据
        while (!KEY);                      //等待按键释放
    }
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
SPSTAT	DATA	0CDH
SPCTL	DATA	0CEH
SPDAT	DATA	0CFH
IE2	DATA	0AFH
ESPI	EQU	02H
SS	BIT	P1.0
LED	BIT	P1.1
KEY	BIT	P0.0
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H
P4M0	DATA	0B4H
P5M1	DATA	0C9H
P5M0	DATA	0CAH
ORG 0000H		
LJMP START		
ORG 004BH		
LJMP SPIISR		
ORG 0100H		
SPIISR:	PUSH	ACC
	MOV	SPSTAT,#0C0H ; 清中断标志
	MOV	A,SPCTL
	JB	ACC.4,MASTER
SLAVE:	MOV	SPDAT,SPDAT ; 将接收到的数据回传给主机
	JMP	ISREXIT
MASTER:	SETB	SS ; 拉高从机的 SS 管脚
	MOV	SPCTL,#40H ; 重新设置为从机待机
ISREXIT:		

<i>CPL</i>	<i>LED</i>
<i>POP</i>	<i>ACC</i>
<i>RETI</i>	

START:

<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	;使能访问 XFR, 没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>SETB</i>	<i>SS</i>	
<i>SETB</i>	<i>LED</i>	
<i>SETB</i>	<i>KEY</i>	
<i>MOV</i>	<i>SPCTL,#40H</i>	;使能 SPI 从机模式进行待机
<i>MOV</i>	<i>SPSTAT,#0C0H</i>	;清中断标志
<i>MOV</i>	<i>IE2,#ESPI</i>	;使能 SPI 中断
<i>SETB</i>	<i>EA</i>	

LOOP:

<i>JB</i>	<i>KEY,LOOP</i>	;等待按键触发
<i>MOV</i>	<i>SPCTL,#50H</i>	;使能 SPI 主机模式
<i>CLR</i>	<i>SS</i>	;拉低从机 SS 管脚
<i>MOV</i>	<i>SPDAT,#5AH</i>	;发送测试数据
<i>JNB</i>	<i>KEY,\$</i>	;等待按键释放
<i>JMP</i>	<i>LOOP</i>	

END

23.6.6 SPI 互为主从系统程序（查询方式）

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

sbit SS = P1^0;
sbit LED = P1^1;
sbit KEY = P0^0;

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭
```

```

P0M0 = 0x00;
P0M1 = 0x00;
P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

LED = I;
KEY = I;
SS = I;

SPCTL = 0x40;                                //使能 SPI 从机模式进行待机
SPSTAT = 0xc0;                                //清中断标志

while (1)
{
    if (!KEY)                                    //等待按键触发
    {
        SPCTL = 0x50;                            //使能 SPI 主机模式
        SS = 0;                                  //拉低从机 SS 管脚
        SPDAT = 0x5a;                            //发送测试数据
        while (!KEY);                           //等待按键释放
    }
    if (SPSTAT & 0x80)
    {
        SPSTAT = 0xc0;                            //清中断标志
        if (SPCTL & 0x10)
        {
            SS = I;                             //拉高从机的 SS 管脚
            SPCTL = 0x40;                        //重新设置为从机待机
        }
        else
        {
            SPDAT = SPDAT;                     //从机模式
            //将接收到的数据回传给主机
        }
        LED = !LED;                            //测试端口
    }
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
SPSTAT	DATA	0CDH
SPCTL	DATA	0CEH
SPDAT	DATA	0CFH
IE2	DATA	0AFH
ESPI	EQU	02H

<i>SS</i>	<i>BIT</i>	<i>P1.0</i>
<i>LED</i>	<i>BIT</i>	<i>P1.1</i>
<i>KEY</i>	<i>BIT</i>	<i>P0.0</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0100H</i>
<i>START:</i>	 	
	<i>MOV</i>	<i>SP, #5FH</i>
	<i>ORL</i>	<i>P_SW2,#80H</i>
		; 使能访问 XFR，没有冲突不用关闭
	<i>MOV</i>	<i>P0M0, #00H</i>
	<i>MOV</i>	<i>P0M1, #00H</i>
	<i>MOV</i>	<i>P1M0, #00H</i>
	<i>MOV</i>	<i>P1M1, #00H</i>
	<i>MOV</i>	<i>P2M0, #00H</i>
	<i>MOV</i>	<i>P2M1, #00H</i>
	<i>MOV</i>	<i>P3M0, #00H</i>
	<i>MOV</i>	<i>P3M1, #00H</i>
	<i>MOV</i>	<i>P4M0, #00H</i>
	<i>MOV</i>	<i>P4M1, #00H</i>
	<i>MOV</i>	<i>P5M0, #00H</i>
	<i>MOV</i>	<i>P5M1, #00H</i>
	<i>SETB</i>	<i>SS</i>
	<i>SETB</i>	<i>LED</i>
	<i>SETB</i>	<i>KEY</i>
	<i>MOV</i>	<i>SPCTL,#40H</i>
	<i>MOV</i>	<i>SPSTAT,#0C0H</i>
		; 使能 SPI 从机模式进行待机
		; 清中断标志
<i>LOOP:</i>	 	
	<i>JB</i>	<i>KEY,SKIP</i>
	<i>MOV</i>	<i>SPCTL,#50H</i>
	<i>CLR</i>	<i>SS</i>
	<i>MOV</i>	<i>SPDAT,#5AH</i>
	<i>JNB</i>	<i>KEY,\$</i>
		; 等待按键触发
		; 使能 SPI 主机模式
		; 拉低从机 SS 管脚
		; 发送测试数据
		; 等待按键释放
<i>SKIP:</i>	 	
	<i>MOV</i>	<i>A,SPSTAT</i>
	<i>JNB</i>	<i>ACC.7,LOOP</i>
	<i>MOV</i>	<i>SPSTAT,#0C0H</i>
	<i>MOV</i>	<i>A,SPCTL</i>
	<i>JB</i>	<i>ACC.4,MASTER</i>
		; 清中断标志
<i>SLAVE:</i>	 	
	<i>MOV</i>	<i>SPDAT,SPDAT</i>
		; 将接收到的数据回传给主机

CPL *LED*
JMP *LOOP*

MASTER:

SETB *SS* ;拉高从机的 SS 管脚
MOV *SPCTL,#40H* ;重新设置为从机待机
CPL *LED*
JMP *LOOP*

END

24 I²C 总线

产品线	I ² C	接收超时 中断
STC8H1K08 系列	●	
STC8H1K28 系列	●	
STC8H3K64S4 系列	●	
STC8H3K64S2 系列	●	
STC8H8K64U 系列	●	
STC8H4K64TL 系列	●	
STC8H4K64TLC 系列	●	
STC8H1K08T 系列	●	
STC8H2K12U-A/B 系列	●	●
STC8H2K32U 系列	●	●
STC8G1K08-SOP8 系列	●	
STC8G1K08A-SOP8 系列	●	

STC8H 系列的单片机内部集成了一个 I²C 串行总线控制器。I²C 是一种高速同步通讯总线，通讯使用 SCL（时钟线）和 SDA（数据线）两线进行同步通讯。对于 SCL 和 SDA 的端口分配，STC8 系列的单片机提供了切换模式，可将 SCL 和 SDA 切换到不同的 I/O 口上，以方便用户将一组 I²C 总线当作多组进行分时复用。

与标准 I²C 协议相比较，忽略了如下两种机制：

- 发送起始信号（START）后不进行仲裁
- 时钟信号（SCL）停留在低电平时不进行超时检测

STC8 系列的 I²C 总线提供了两种操作模式：主机模式（SCL 为输出口，发送同步时钟信号）和从机模式（SCL 为输入口，接收同步时钟信号）

STC 创新：STC 的 I²C 串行总线控制器工作在从机模式时，SDA 管脚的下降沿信号可以唤醒进入掉电模式的 MCU。（注意：由于 I²C 传输速度比较快，MCU 唤醒后第一包数据一般是不正确的）

24.1 I²C 功能脚切换

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P_SW2	BAH	EAXFR	-	I2C_S[1:0]	CMPO_S	S4_S	S3_S	S2_S	

I2C_S[1:0]: I²C 功能脚选择位

I2C_S[1:0]	SCL	SDA
00	P1.5	P1.4
01	P2.5	P2.4
10	P7.7	P7.6
11	P3.2	P3.3

24.2 I²C 相关的寄存器

符号	描述	地址	位地址与符号								复位值	
			B7	B6	B5	B4	B3	B2	B1	B0		
I2CCFG	I ² C 配置寄存器	FE80H	ENI2C	MSSL	MSSPEED[5:0]							
I2CMSCR	I ² C 主机控制寄存器	FE81H	EMSI	-	-	-	MSCMD[3:0]				0xxx,0000	
I2CMSST	I ² C 主机状态寄存器	FE82H	MSBUSY	MSIF	-	-	-	-	MSACKI	MSACKO	0xx,xx10	
I2CSLCR	I ² C 从机控制寄存器	FE83H	-	ESTAI	ERXI	ETXI	ESTOI	-	-	SLRST	x000,0xx0	
I2CSLST	I ² C 从机状态寄存器	FE84H	SLBUSY	STAIF	RXIF	TXIF	STOIF	TXING	SLACKI	SLACKO	0000,0000	
I2CSLADR	I ² C 从机地址寄存器	FE85H	I2CSLADR[7:1]								MA	0000,0000
I2CTXD	I ² C 数据发送寄存器	FE86H										0000,0000
I2CRXD	I ² C 数据接收寄存器	FE87H										0000,0000
I2CMSAUX	I ² C 主机辅助控制寄存器	FE88H	-	-	-	-	-	-	-	-	WDTA	xxxx,xxx0

符号	描述	地址	位地址与符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
I2CTOCR	I ² C 从机超时控制寄存器	FD84H	ENTO	ENTOI	SCALE	-	-	-	-	-	000x,xxxx
I2CTOSR	I ² C 从机超时状态寄存器	FD85H	CTOIF	-	-	-	-	-	-	TOIF	0xxx,xxx0
I2CTOTH	I ² C 从机超时长度寄存器	FD86H	TM[15:8]								0000,0000
I2CTOTL	I ² C 从机超时长度寄存器	FD87H	TM[7:0]								0000,0000
I2CTOTE	I ² C 从机超时长度寄存器	FD8DH	TM[23:16]								0000,0000

24.3 I²C 主机模式

24.3.1 I²C 配置寄存器 (I2CCFG) , 总线速度控制

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CCFG	FE80H	ENI2C	MSSL	MSSPEED[5:0]					

ENI2C: I²C 功能使能控制位

0: 禁止 I²C 功能

1: 允许 I²C 功能

MSSL: I²C 工作模式选择位

0: 从机模式

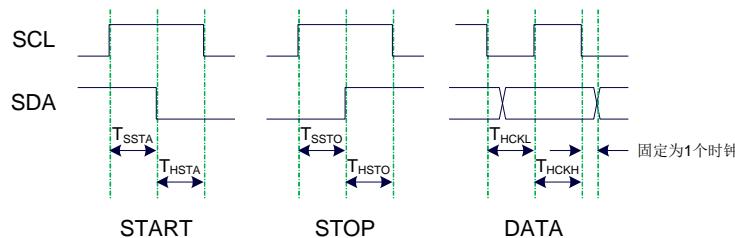
1: 主机模式

MSSPEED[5:0]: I²C 总线速度 (等待时钟数) 控制, I²C 总线速度 = Fosc / 2 / (MSSPEED * 2 + 4)

MSSPEED[5:0]	对应的时钟数
0	4
1	6
2	8
...	...
X	2x+4
...	...
62	128

只有当 I²C 模块工作在主机模式时，MSSPEED 参数设置的等待参数才有效。此等待参数主要用于主机模式的以下几个信号：

- T_{SSTA}: 起始信号的建立时间（Setup Time of START）
- T_{HSTA}: 起始信号的保持时间（Hold Time of START）
- T_{SSTO}: 停止信号的建立时间（Setup Time of STOP）
- T_{HSTO}: 停止信号的保持时间（Hold Time of STOP）
- T_{HCKL}: 时钟信号的低电平保持时间（Hold Time of SCL Low）
- T_{HCKH}: 时钟信号的高电平保持时间（Hold Time of SCL High）



例 1：当 MSSPEED=10 时，T_{SSTA}=T_{HSTA}=T_{SSTO}=T_{HSTO}=T_{HCKL}=24/FOSC

例 2：当 24MHz 的工作频率下需要 400K 的 I²C 总线速度时，

$$\text{MSSPEED} = (24\text{M} / 400\text{K} / 2 - 4) / 2 = 13$$

24.3.2 I²C 主机控制寄存器 (I2CMSCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CMSCR	FE81H	EMSI	-	-	-	MSCMD[3:0]			

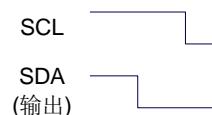
EMSI: 主机模式中断使能控制位

- 0: 关闭主机模式的中断
- 1: 允许主机模式的中断

MSCMD[3:0]: 主机命令

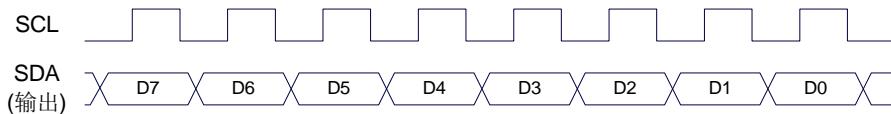
- 0000: 待机, 无动作。
- 0001: 起始命令。

发送 START 信号。如果当前 I²C 控制器处于空闲状态, 即 MSBUSY (I2CMSST.7) 为 0 时, 写此命令会使控制器进入忙状态, 硬件自动将 MSBUSY 状态位置 1, 并开始发送 START 信号; 若当前 I²C 控制器处于忙状态, **写此命令可触发发送 START 信号**。发送 START 信号的波形如下图所示:



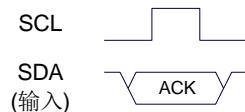
0010: 发送数据命令。

写此命令后, I²C 总线控制器会在 SCL 管脚上产生 8 个时钟, 并将 I2CTXD 寄存器里面数据按位送到 SDA 管脚上 (先发送高位数据)。发送数据的波形如下图所示:



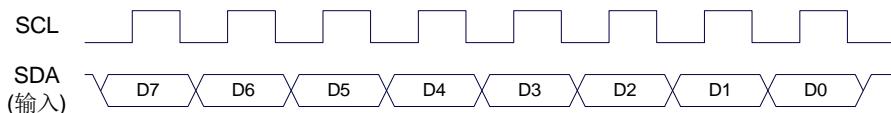
0011: 接收 ACK/NAK 命令。

写此命令后, I²C 总线控制器会在 SCL 管脚上产生 1 个时钟, 并将从 SDA 端口上读取的数据保存到 MSACKI (I2CMSST.1)。接收 ACK/NAK 的波形如下图所示:



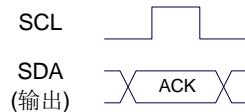
0100: 接收数据命令。

写此命令后, I²C 总线控制器会在 SCL 管脚上产生 8 个时钟, 并将从 SDA 端口上读取的数据依次左移到 I2CRXD 寄存器 (先接收高位数据)。接收数据的波形如下图所示:



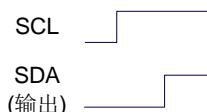
0101: 发送 ACK/NAK 命令。

写此命令后, I²C 总线控制器会在 SCL 管脚上产生 1 个时钟, 并将 MSACKO (I2CMSST.0) 中的数据发送到 SDA 端口。发送 ACK/NAK 的波形如下图所示:



0110: 停止命令。

发送 STOP 信号。写此命令后, I²C 总线控制器开始发送 STOP 信号。信号发送完成后, 硬件自动将 MSBUSY 状态位清零。STOP 信号的波形如下图所示:



0111: 保留。

1000: 保留。

1001: 起始命令+发送数据命令+接收 ACK/NAK 命令。

此命令为命令 0001、命令 0010、命令 0011 三个命令的组合, 下此命令后控制器会依次执行这三个命令。

1010: 发送数据命令+接收 ACK/NAK 命令。

此命令为命令 0010、命令 0011 两个命令的组合, 下此命令后控制器会依次执行这两个命令。

1011: 接收数据命令+发送 ACK(0)命令。

此命令为命令 0100、命令 0101 两个命令的组合, 下此命令后控制器会依次执行这两个命令。

注意: 此命令所返回的应答信号固定为 ACK (0), 不受 MSACKO 位的影响。

1100: 接收数据命令+发送 NAK(1)命令。

此命令为命令 0100、命令 0101 两个命令的组合, 下此命令后控制器会依次执行这两个命令。

注意: 此命令所返回的应答信号固定为 NAK (1), 不受 MSACKO 位的影响。

24.3.3 I²C 主机辅助控制寄存器 (I2CMSAUX)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CMSAUX	FE88H	-	-	-	-	-	-	-	WDTA

WDTA: 主机模式时 I²C 数据自动发送允许位

0: 禁止自动发送

1: 使能自动发送

若自动发送功能被使能, 当 MCU 执行完成对 I2CTXD 数据寄存器的写操作后, I²C 控制器会自动触发“1010”命令, 即自动发送数据并接收 ACK/NAK 信号。

24.3.4 I²C 主机状态寄存器 (I2CMSST)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CMSST	FE82H	MSBUSY	MSIF	-	-	-	-	MSACKI	MSACKO

MSBUSY: 主机模式时 I²C 控制器状态位 (只读位)

0: 控制器处于空闲状态

1: 控制器处于忙碌状态

当 I²C 控制器处于主机模式时, 在空闲状态下, 发送完成 START 信号后, 控制器便进入到忙碌状态, 忙碌状态会一直维持到成功发送完成 STOP 信号, 之后状态会再次恢复到空闲状态。

MSIF: 主机模式的中断请求位 (中断标志位)。当处于主机模式的 I²C 控制器执行完成寄存器 I2CMSCR 中 MSCMD 命令后产生中断信号, 硬件自动将此位 1, 向 CPU 发请求中断, 响应中断后 MSIF 位必须用软件清零。

MSACKI: 主机模式时, 发送“0011”命令到 I2CMSCR 的 MSCMD 位后所接收到的 ACK/NAK 数据。
(只读位)

MSACKO: 主机模式时, 准备将要发送出去的 ACK/NAK 信号。当发送“0101”命令到 I2CMSCR 的 MSCMD 位后, 控制器会自动读取此位的数据当作 ACK/NAK 发送到 SDA。

24.4 I²C 从机模式

24.4.1 I²C 从机控制寄存器 (I2CSLCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CSLCR	FE83H	-	ESTAI	ERXI	ETXI	ESTOI	-	-	SLRST

ESTAI: 从机模式时接收到 START 信号中断允许位

0: 禁止从机模式时接收到 START 信号时发生中断

1: 使能从机模式时接收到 START 信号时发生中断

ERXI: 从机模式时接收到 1 字节数据后中断允许位

0: 禁止从机模式时接收到数据后发生中断

1: 使能从机模式时接收到 1 字节数据后发生中断

ETXI: 从机模式时发送完成 1 字节数据后中断允许位

0: 禁止从机模式时发送完成数据后发生中断

1: 使能从机模式时发送完成 1 字节数据后发生中断

ESTOI: 从机模式时接收到 STOP 信号中断允许位

0: 禁止从机模式时接收到 STOP 信号时发生中断

1: 使能从机模式时接收到 STOP 信号时发生中断

SLRST: 复位从机模式

24.4.2 I²C 从机状态寄存器 (I2CSLST)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CSLST	FE84H	SLBUSY	STAIF	RXIF	TXIF	STOIF	-	SLACKI	SLACKO

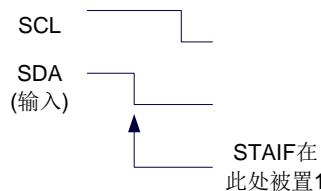
SLBUSY: 从机模式时 I²C 控制器状态位 (只读位)

0: 控制器处于空闲状态

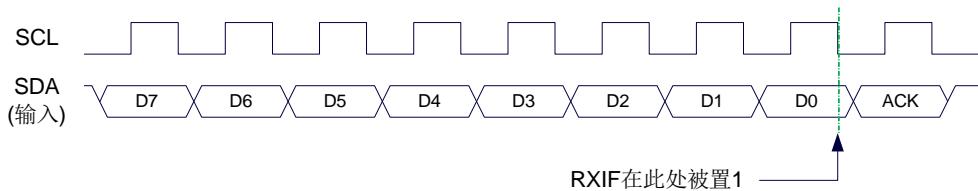
1: 控制器处于忙碌状态

当 I²C 控制器处于从机模式时, 在空闲状态下, 接收到主机发送 START 信号后, 控制器会继续检测之后的设备地址数据, 若设备地址与当前 I2CSLADR 寄存器中所设置的从机地址相同时, 控制器便进入到忙碌状态, 忙碌状态会一直维持到成功接收到主机发送 STOP 信号, 之后状态会再次恢复到空闲状态。

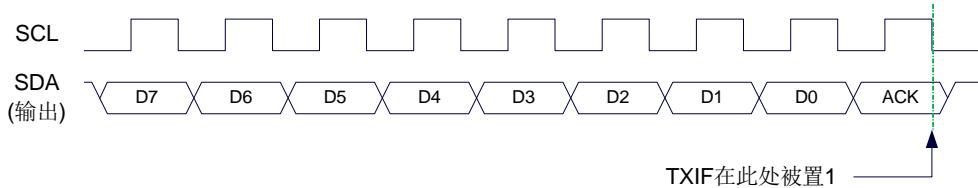
STAIF: 从机模式时接收到 START 信号后的中断请求位。从机模式的 I²C 控制器接收到 START 信号后, 硬件会自动将此位置 1, 并向 CPU 发请求中断, 响应中断后 STAIF 位必须用软件清零。STAIF 被置 1 的时间点如下图所示:



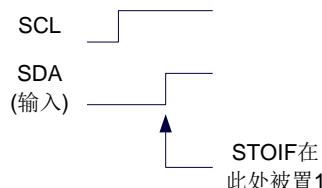
RXIF: 从机模式时接收到 1 字节的数据后的中断请求位。从机模式的 I²C 控制器接收到 1 字节的数据后, 在第 8 个时钟的下降沿时硬件会自动将此位置 1, 并向 CPU 发请求中断, 响应中断后 RXIF 位必须用软件清零。RXIF 被置 1 的时间点如下图所示:



TXIF: 从机模式时发送完成 1 字节的数据后的中断请求位。从机模式的 I²C 控制器发送完成 1 字节的数据并成功接收到 1 位 ACK/NAK 信号后, 在第 9 个时钟的下降沿时硬件会自动将此位置 1, 并向 CPU 发请求中断, 响应中断后 TXIF 位必须用软件清零。TXIF 被置 1 的时间点如下图所示:

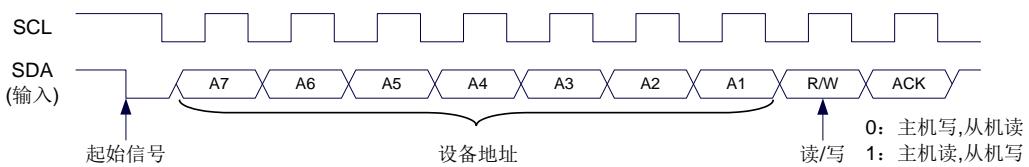


STOIF: 从机模式时接收到 STOP 信号后的中断请求位。从机模式的 I²C 控制器接收到 STOP 信号后, 硬件会自动将此位置 1, 并向 CPU 发请求中断, 响应中断后 STOIF 位必须用软件清零。STOIF 被置 1 的时间点如下图所示:



SLACKI: 从机模式时, 接收到的 ACK/NAK 数据。

SLACKO: 从机模式时, 准备将要发送出去的 ACK/NAK 信号。



24.4.3 I²C 从机地址寄存器 (I2CSLADR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CSLADR	FE85H								MA

I2CSLADR[7:1]: 从机设备地址

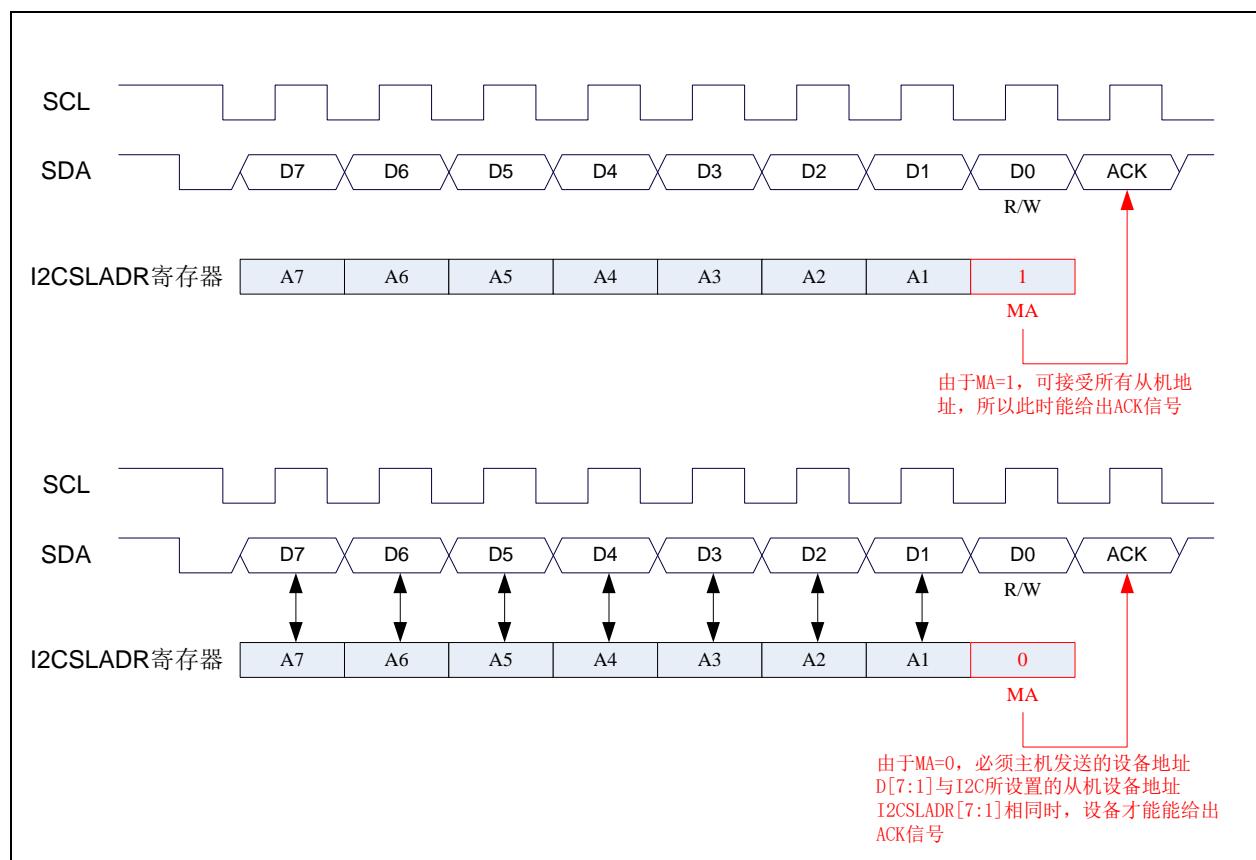
当 I²C 控制器处于从机模式时, 控制器在接收到 START 信号后, 会继续检测接下来主机发送出的设备地址数据以及读/写信号。当主机发送出的设备地址与 I2CSLADR[7:1]中所设置的从机设备地址相同时, 控制器才会向 CPU 发出中断求, 请求 CPU 处理 I²C 事件; 否则若设备地址不同, I²C 控制器继续监控, 等待下一个起始信号, 对下一个设备地址继续比较。

MA: 从机设备地址比较控制

0: 设备地址必须与 I2CSLADR[7:1]相同

1: 忽略 I2CSLADR[7:1]中的设置, 接受所有的设备地址

说明: I²C 总线协议规定 I²C 总线上最多可挂载 128 个 I²C 设备 (理论值), 不同的 I²C 设备用不同的 I²C 从机设备地址进行识别。I²C 主机发送完成起始信号后, 发送的第一个数据 (DATA0) 的高 7 位即为从机设备地址 (DATA0[7:1]为 I²C 设备地址), 最低位为读写信号。当 I²C 设备从机地址寄存器 MA (I2CSLADR.0) 为 1 时, 表示 I²C 从机能够接受所有的设备地址, 此时主机发送的任何设备地址, 即 DATA0[7:1]为任何值, 从机都能响应。当 I²C 设备从机地址寄存器 MA (I2CSLADR.0) 为 0 时, 主机发送的设备地址 DATA0[7:1]必须与从机的设备地址 I2CSLADR[7:1]相同时才能访问此从机设备



24.4.4 I²C 数据寄存器 (I2CTXD, I2CRXD)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTXD	FE86H								
I2CRXD	FE87H								

I2CTXD 是 I²C 发送数据寄存器，存放将要发送的 I²C 数据

I2CRXD 是 I²C 接收数据寄存器，存放接收完成的 I²C 数据

24.4.5 I2C 从机超时控制寄存器 (I2CTOCR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTOCR	7EFD84H	ENTO	ENTOI	SCALE	-	-	-	-	-

ENTO: I2C 从机超时功能控制位

0: 禁止 I2C 从机超时功能

1: 使能 I2C 从机超时功能 (注: I2C 的主机模式无接收超时功能)

ENTOI: I2C 从机超时中断控制位

0: 禁止 I2C 从机超时中断

1: 使能 I2C 从机超时中断

SCALE: I2C 超时计数时钟源选择

0: 1us 时钟 (1MHz 时钟)。(注意: 如需要使用此时钟源, 用户必须先正确设置 IAP_TPS 寄存器。

例如: 若系统时钟为 12MHz, 则需要将 IAP_TPS 设置为 12; 若系统时钟为 22.1184MHz, 则需要将 IAP_TPS 设置为 22; 其他频率以此类推。特别注意, 此 1us 的时钟并不是精准时间)

1: 系统时钟

24.4.6 I2C 从机超时状态寄存器 (I2CTOSR)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTOSR	7EFD85H	CTOIF	-	-	-	-	-	-	TOIF

CTOIF: 写“1”清除超时中断标志位 TOIF。(只写)

TOIF: I2C 超时中断请求标志位。(只读)

当发生 I2C 超时时, TOIF 会被硬件置“1”。如果 ENTOI 为 1 时会产生 I2C 中断, 中断入口地址为原 I2C 的中断入口地址。标志位需要软件向 CTOIF 位写“1”清零

24.4.7 I2C 从机超时长度控制寄存器 (I2CTOTE/H/L)

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
I2CTOTE	7EFD8DH					TM[23:16]			
I2CTOTH	7EFD86H					TM[15:8]			
I2CTOTL	7EFD87H					TM[7:0]			

TM[23:0]: I2C 超时时间控制位。

当从机模式的 I2C 处于空闲状态时 (起始信号后的地址消息与软件所设置的从机地址未发生匹配被认定为空闲状态, 若地址已经发生匹配, 则不会发生超时), 内部超时计数器根据 SCALE 寄存器所选择的时钟源进行计数, 当计数时间达到 TM 所设置的超时时间时, 便会产生超时中断。

当 I2C 数据传输完成时, 复位内部超时计数器, 重新进行超时计数。

注:

- 1、如果需要使能接收超时中断功能, 则 TM 不可设置为 0, 且 SPITOTL、SPITOTH、SPITOTE 寄存器的设置必须先设置 SPITOTL 和 SPITOTH, 最后设置 SPITOTE
- 2、只有 I2C 从机模式才有接收超时功能
- 3、接收超时功能必须在从机正确接收到一字节数据后才能触发
- 4、正在接收过程中, 无接收超时功能

24.5 范例程序

24.5.1 I²C 主机模式访问 AT24C256 (中断方式)

C 语言代码

```
//测试工作频率为 11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit SDA = P1^4;
sbit SCL = P1^5;

bit busy;

void I2C_Isr() interrupt 24
{
    if (I2CMSST & 0x40)
    {
        I2CMSST &= ~0x40; //清中断标志
        busy = 0;
    }
}

void Start()
{
    busy = 1;
    I2CMSCR = 0x81; //发送 START 命令
    while (busy);
}

void SendData(char dat)
{
    I2CTXD = dat; //写数据到数据缓冲区
    busy = 1;
    I2CMSCR = 0x82; //发送 SEND 命令
    while (busy);
}

void RecvACK()
{
    busy = 1;
    I2CMSCR = 0x83; //发送读 ACK 命令
    while (busy);
}

char RecvData()
{
    busy = 1;
    I2CMSCR = 0x84; //发送 RECV 命令
    while (busy);
    return I2CRXD;
}

void SendACK()
{
```

```

I2CMSST = 0x00;                                //设置ACK 信号
busy = 1;
I2CMSCR = 0x85;                                //发送ACK 命令
while (busy);
}

void SendNAK()
{
    I2CMSST = 0x01;                                //设置NAK 信号
    busy = 1;
    I2CMSCR = 0x85;                                //发送ACK 命令
    while (busy);
}

void Stop()
{
    busy = 1;                                     //发送STOP 命令
    I2CMSCR = 0x86;
    while (busy);
}

void Delay()
{
    int i;

    for (i=0; i<3000; i++)
    {
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    I2CCFG = 0xe0;                                //使能I2C 主机模式
    I2CMSST = 0x00;
    EA = 1;

    Start();           //发送起始命令
    SendData(0xa0);                                //发送设备地址+写命令
    RecvACK();
    SendData(0x00);                                //发送存储地址高字节
}

```

```

RecvACK();                                //发送存储地址低字节
SendData(0x00);
RecvACK();
SendData(0x12);                        //写测试数据1
RecvACK();
SendData(0x78);                        //写测试数据2
RecvACK();
Stop();                                 //发送停止命令

Delay();                               //等待设备写数据

Start();                                //发送起始命令
SendData(0xa0);                        //发送设备地址+写命令
RecvACK();
SendData(0x00);                        //发送存储地址高字节
RecvACK();
SendData(0x00);                        //发送存储地址低字节
RecvACK();
Start();                                //发送起始命令
SendData(0xa1);                        //发送设备地址+读命令
RecvACK();
P0 = RecvData();                      //读取数据1
SendACK();
P2 = RecvData();                      //读取数据2
SendNAK();
Stop();                                 //发送停止命令

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
I2CCFG	XDATA	0FE80H
I2CMSCR	XDATA	0FE81H
I2CMSST	XDATA	0FE82H
I2CSLCR	XDATA	0FE83H
I2CSLST	XDATA	0FE84H
I2CSLADR	XDATA	0FE85H
I2CTXD	XDATA	0FE86H
I2CRXD	XDATA	0FE87H
SDA	BIT	P1.4
SCL	BIT	P1.5
BUSY	BIT	20H.0
P1M1	DATA	091H
P1M0	DATA	092H
P0M1	DATA	093H
P0M0	DATA	094H
P2M1	DATA	095H
P2M0	DATA	096H
P3M1	DATA	0B1H
P3M0	DATA	0B2H
P4M1	DATA	0B3H

<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>00C3H</i>	
	<i>LJMP</i>	<i>I2CISR</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>I2CISR:</i>	<i>PUSH</i>	<i>ACC</i>	
	<i>PUSH</i>	<i>DPL</i>	
	<i>PUSH</i>	<i>DPH</i>	
	<i>MOV</i>	<i>DPTR,#I2CMSST</i>	;清中断标志
	<i>MOVX</i>	<i>A,@DPTR</i>	
	<i>ANL</i>	<i>A,#NOT 40H</i>	
	<i>MOV</i>	<i>DPTR,#I2CMSST</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>CLR</i>	<i>BUSY</i>	;复位忙标志
	<i>POP</i>	<i>DPH</i>	
	<i>POP</i>	<i>DPL</i>	
	<i>POP</i>	<i>ACC</i>	
	<i>RETI</i>		
<i>SENDSTART:</i>			
	<i>SETB</i>	<i>BUSY</i>	
	<i>MOV</i>	<i>A,#10000001B</i>	;发送 START 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>SENDDATA:</i>			
	<i>MOV</i>	<i>DPTR,#I2CTXD</i>	;写数据到数据缓冲区
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>SETB</i>	<i>BUSY</i>	
	<i>MOV</i>	<i>A,#10000010B</i>	;发送 SEND 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>RECVACK:</i>			
	<i>SETB</i>	<i>BUSY</i>	
	<i>MOV</i>	<i>A,#10000011B</i>	;发送读 ACK 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>RECVDATA:</i>			
	<i>SETB</i>	<i>BUSY</i>	
	<i>MOV</i>	<i>A,#10000100B</i>	;发送 RECV 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>CALL</i>	<i>WAIT</i>	
	<i>MOV</i>	<i>DPTR,#I2CRXD</i>	;从数据缓冲区读取数据
	<i>MOVX</i>	<i>A,@DPTR</i>	
	<i>RET</i>		
<i>SENDACK:</i>			
	<i>MOV</i>	<i>A,#00000000B</i>	;设置 ACK 信号
	<i>MOV</i>	<i>DPTR,#I2CMSST</i>	

```

MOVX      @DPTR,A
SETB      BUSY
MOV      A,#10000101B          ;发送ACK 命令
MOV      DPTR,#I2CMSCR
MOVX      @DPTR,A
JMP      WAIT

SENDNAK:
MOV      A,#00000001B          ;设置NAK信号
MOV      DPTR,#I2CMSST
MOVX      @DPTR,A
SETB      BUSY
MOV      A,#10000101B          ;发送ACK 命令
MOV      DPTR,#I2CMSCR
MOVX      @DPTR,A
JMP      WAIT

SENDSTOP:
SETB      BUSY
MOV      A,#10000110B          ;发送STOP 命令
MOV      DPTR,#I2CMSCR
MOVX      @DPTR,A
JMP      WAIT

WAIT:
JB      BUSY,$              ;等待命令发送完成
RET

DELAY:
MOV      R0,#0
MOV      R1,#0

DELAYI:
NOP
NOP
NOP
NOP
DJNZ      R1,DELAYI
DJNZ      R0,DELAYI
RET

START:
MOV      SP, #5FH
ORL      P_SW2,#80H          ;使能访问XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      A,#11100000B          ;设置I2C模块为主机模式
MOV      DPTR,#I2CCFG
MOVX      @DPTR,A
MOV      A,#00000000B
MOV      DPTR,#I2CMSST

```

<i>MOVX</i>	<i>@DPTR,A</i>	
<i>SETB</i>	<i>EA</i>	
<i>CALL</i>	<i>SENDSTART</i>	;发送起始命令
<i>MOV</i>	<i>A,#0A0H</i>	
<i>CALL</i>	<i>SENDATA</i>	;发送设备地址+写命令
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#000H</i>	;发送存储地址高字节
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#000H</i>	;发送存储地址低字节
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#12H</i>	;写测试数据 1
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#78H</i>	;写测试数据 2
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>SENDSTOP</i>	;发送停止命令
<i>CALL</i>	<i>DELAY</i>	;等待设备写数据
<i>CALL</i>	<i>SENDSTART</i>	;发送起始命令
<i>MOV</i>	<i>A,#0A0H</i>	;发送设备地址+写命令
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#000H</i>	;发送存储地址高字节
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#000H</i>	;发送存储地址低字节
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>SENDSTART</i>	;发送起始命令
<i>MOV</i>	<i>A,#0A1H</i>	;发送设备地址+读命令
<i>CALL</i>	<i>SENDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>RECVDATA</i>	;读取数据 1
<i>MOV</i>	<i>P0,A</i>	
<i>CALL</i>	<i>SENDACK</i>	
<i>CALL</i>	<i>RECVDATA</i>	;读取数据 2
<i>MOV</i>	<i>P2,A</i>	
<i>CALL</i>	<i>SENDNAK</i>	
<i>CALL</i>	<i>SENDSTOP</i>	;发送停止命令
<i>JMP</i>	\$	
<i>END</i>		

24.5.2 I²C 主机模式访问 AT24C256 (查询方式)

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
```

```
#include "intrins.h"

sbit      SDA      =  P1^4;
sbit      SCL      =  P1^5;

void Wait()
{
    while (!(I2CMSST & 0x40));
    I2CMSST &= ~0x40;
}

void Start()
{
    I2CMSCR = 0x01;                      //发送 START 命令
    Wait();
}

void SendData(char dat)
{
    I2CTXD = dat;                        //写数据到数据缓冲区
    I2CMSCR = 0x02;                      //发送 SEND 命令
    Wait();
}

void RecvACK()
{
    I2CMSCR = 0x03;                      //发送读 ACK 命令
    Wait();
}

char RecvData()
{
    I2CMSCR = 0x04;                      //发送 RECV 命令
    Wait();
    return I2CRXD;
}

void SendACK()
{
    I2CMSST = 0x00;                      //设置 ACK 信号
    I2CMSCR = 0x05;                      //发送 ACK 命令
    Wait();
}

void SendNAK()
{
    I2CMSST = 0x01;                      //设置 NAK 信号
    I2CMSCR = 0x05;                      //发送 ACK 命令
    Wait();
}

void Stop()
{
    I2CMSCR = 0x06;                      //发送 STOP 命令
    Wait();
}

void Delay()
{
```

```

int i;

for (i=0; i<3000; i++)
{
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR， 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    I2CCFG = 0xe0;                                //使能 I2C 主机模式
    I2CMSST = 0x00;

    Start();                                         //发送起始命令
    SendData(0xa0);                                //发送设备地址+写命令
    RecvACK();
    SendData(0x00);                                //发送存储地址高字节
    RecvACK();
    SendData(0x00);                                //发送存储地址低字节
    RecvACK();
    SendData(0x12);                                //写测试数据 1
    RecvACK();
    SendData(0x78);                                //写测试数据 2
    RecvACK();
    Stop();                                         //发送停止命令

    Delay();                                         //等待设备写数据

    Start();                                         //发送起始命令
    SendData(0xa0);                                //发送设备地址+写命令
    RecvACK();
    SendData(0x00);                                //发送存储地址高字节
    RecvACK();
    SendData(0x00);                                //发送存储地址低字节
    RecvACK();
    Start();                                         //发送起始命令
    SendData(0xa1);                                //发送设备地址+读命令
    RecvACK();
    P0 = RecvData();                                //读取数据 1
    SendACK();
    P2 = RecvData();                                //读取数据 2
}

```

```

SendNAK();
Stop();                                //发送停止命令

while (1);
}

```

汇编代码

;测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>I2CCFG</i>	<i>XDATA</i>	<i>0FE80H</i>
<i>I2CMSCR</i>	<i>XDATA</i>	<i>0FE81H</i>
<i>I2CMSST</i>	<i>XDATA</i>	<i>0FE82H</i>
<i>I2CSLCR</i>	<i>XDATA</i>	<i>0FE83H</i>
<i>I2CSLST</i>	<i>XDATA</i>	<i>0FE84H</i>
<i>I2CSLADR</i>	<i>XDATA</i>	<i>0FE85H</i>
<i>I2CTXD</i>	<i>XDATA</i>	<i>0FE86H</i>
<i>I2CRXD</i>	<i>XDATA</i>	<i>0FE87H</i>
<i>SDA</i>	<i>BIT</i>	<i>P1.4</i>
<i>SCL</i>	<i>BIT</i>	<i>P1.5</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>0100H</i>
<i>SENDSTART:</i>		
	<i>MOV</i>	<i>A,#00000001B</i> ;发送 START 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>JMP</i>	<i>WAIT</i>
<i>SENDDATA:</i>		
	<i>MOV</i>	<i>DPTR,#I2CTXD</i> ;写数据到数据缓冲区
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>MOV</i>	<i>A,#00000010B</i> ;发送 SEND 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>JMP</i>	<i>WAIT</i>
<i>RECVACK:</i>		
	<i>MOV</i>	<i>A,#00000011B</i> ;发送读 ACK 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>JMP</i>	<i>WAIT</i>
<i>RECVDATA:</i>		

```

MOV      A,#00000100B          ;发送RECV 命令
MOV      DPTR,#I2CMSCR
MOVX    @DPTR,A
CALL    WAIT
MOV      DPTR,#I2CRXD        ;从数据缓冲区读取数据
MOVX    A,@DPTR
RET

SENDACK:
MOV      A,#00000000B          ;设置ACK 信号
MOV      DPTR,#I2CMSST
MOVX    @DPTR,A
MOV      A,#00000101B          ;发送ACK 命令
MOV      DPTR,#I2CMSCR
MOVX    @DPTR,A
JMP     WAIT

SENDNAK:
MOV      A,#00000001B          ;设置NAK 信号
MOV      DPTR,#I2CMSST
MOVX    @DPTR,A
MOV      A,#00000101B          ;发送ACK 命令
MOV      DPTR,#I2CMSCR
MOVX    @DPTR,A
JMP     WAIT

SENDSTOP:
MOV      A,#00000110B          ;发送STOP 命令
MOV      DPTR,#I2CMSCR
MOVX    @DPTR,A
JMP     WAIT

WAIT:
MOV      DPTR,#I2CMSST        ;清中断标志
MOVX    A,@DPTR
JNB    ACC.6,WAIT
ANL     A,#NOT 40H
MOVX    @DPTR,A
RET

DELAY:
MOV      R0,#0
MOV      R1,#0

DELAY1:
NOP
NOP
NOP
NOP
DJNZ   R1,DELAY1
DJNZ   R0,DELAY1
RET

START:
MOV      SP, #5FH
ORL    P_SW2,#80H          ;使能访问XFR, 没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H

```

```

MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      A,#III00000B          ;设置 I2C 模块为主机模式
MOV      DPTR,#I2CCFG
MOVX    @DPTR,A
MOV      A,#00000000B
MOV      DPTR,#I2CMSST
MOVX    @DPTR,A

CALL    SENDSTART            ;发送起始命令
MOV      A,#0A0H
CALL    SENDDATA              ;发送设备地址+写命令
CALL    RECVACK
MOV      A,#000H                ;发送存储地址高字节
CALL    SENDDATA
CALL    RECVACK
MOV      A,#000H                ;发送存储地址低字节
CALL    SENDDATA
CALL    RECVACK
MOV      A,#I2H                 ;写测试数据 1
CALL    SENDDATA
CALL    RECVACK
MOV      A,#78H                 ;写测试数据 2
CALL    SENDDATA
CALL    RECVACK
CALL    SENDSTOP               ;发送停止命令

CALL    DELAY                  ;等待设备写数据

CALL    SENDSTART            ;发送起始命令
MOV      A,#0A0H                ;发送设备地址+写命令
CALL    SENDDATA
CALL    RECVACK
MOV      A,#000H                ;发送存储地址高字节
CALL    SENDDATA
CALL    RECVACK
MOV      A,#000H                ;发送存储地址低字节
CALL    SENDDATA
CALL    RECVACK
CALL    SENDSTART            ;发送起始命令
MOV      A,#0AIH                 ;发送设备地址+读命令
CALL    SENDDATA
CALL    RECVACK
CALL    RECVDATA               ;读取数据 1
MOV      P0,A
CALL    SENDACK
CALL    RECVDATA               ;读取数据 2
MOV      P2,A
CALL    SENDNAK
CALL    SENDSTOP               ;发送停止命令

JMP      $
END

```

24.5.3 I²C 主机模式访问 PCF8563

C 语言代码

```
//测试工作频率为11.0592MHz

#include "stc8h.h"
#include "intrins.h"

sbit SDA = P1^4;
sbit SCL = P1^5;

void Wait()
{
    while (!(I2CMSST & 0x40));
    I2CMSST &= ~0x40;
}

void Start()
{
    I2CMSCR = 0x01; //发送START命令
    Wait();
}

void SendData(char dat)
{
    I2CTXD = dat; //写数据到数据缓冲区
    I2CMSCR = 0x02; //发送SEND命令
    Wait();
}

void RcvACK()
{
    I2CMSCR = 0x03; //发送读ACK命令
    Wait();
}

char RcvData()
{
    I2CMSCR = 0x04; //发送RECV命令
    Wait();
    return I2CRXD;
}

void SendACK()
{
    I2CMSST = 0x00; //设置ACK信号
    I2CMSCR = 0x05; //发送ACK命令
    Wait();
}

void SendNAK()
{
    I2CMSST = 0x01; //设置NAK信号
    I2CMSCR = 0x05; //发送ACK命令
    Wait();
}
```

```

}

void Stop()
{
    I2CMSCR = 0x06; //发送 STOP 命令
    Wait();
}

void Delay()
{
    int i;

    for (i=0; i<3000; i++)
    {
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR，没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    I2CCFG = 0xe0; //使能 I2C 主机模式
    I2CMSST = 0x00;

    Start(); //发送起始命令
    SendData(0xa2); //发送设备地址+写命令
    RecvACK();
    SendData(0x02); //发送存储地址
    RecvACK();
    SendData(0x00); //设置秒值
    RecvACK();
    SendData(0x00); //设置分钟值
    RecvACK();
    SendData(0x12); //设置小时值
    RecvACK();
    Stop(); //发送停止命令

    while (1)
    {
        Start(); //发送起始命令
        SendData(0xa2); //发送设备地址+写命令
        RecvACK();
    }
}

```

```

SendData(0x02);                                //发送存储地址
RecvACK();
Start();                                         //发送起始命令
SendData(0xa3);                                //发送设备地址+读命令
RecvACK();
P0 = RecvData();                                //读取秒值
SendACK();
P2 = RecvData();                                //读取分钟值
SendACK();
P3 = RecvData();                                //读取小时值
SendNAK();
Stop();                                         //发送停止命令

Delay();
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

<i>P_SW2</i>	<i>DATA</i>	<i>0BAH</i>
<i>I2CCFG</i>	<i>XDATA</i>	<i>0FE80H</i>
<i>I2CMSCR</i>	<i>XDATA</i>	<i>0FE81H</i>
<i>I2CMSST</i>	<i>XDATA</i>	<i>0FE82H</i>
<i>I2CSLCR</i>	<i>XDATA</i>	<i>0FE83H</i>
<i>I2CSLST</i>	<i>XDATA</i>	<i>0FE84H</i>
<i>I2CSLADR</i>	<i>XDATA</i>	<i>0FE85H</i>
<i>I2CTXD</i>	<i>XDATA</i>	<i>0FE86H</i>
<i>I2CRXD</i>	<i>XDATA</i>	<i>0FE87H</i>
<i>SDA</i>	<i>BIT</i>	<i>P1.4</i>
<i>SCL</i>	<i>BIT</i>	<i>P1.5</i>
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
<i>ORG</i>	<i>0000H</i>	
<i>LJMP</i>	<i>START</i>	
<i>ORG</i>	<i>0100H</i>	
<i>SENDSTART:</i>		
<i>MOV</i>	<i>A,#00000001B</i>	;发送 START 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
<i>SENDDATA:</i>		
<i>MOV</i>	<i>DPTR,#I2CTXD</i>	;写数据到数据缓冲区

<i>MOVX</i>	<i>@DPTR,A</i>	
<i>MOV</i>	<i>A,#00000010B</i>	;发送 SEND 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
RECVACK:		
<i>MOV</i>	<i>A,#00000011B</i>	;发送读 ACK 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
RECVDATA:		
<i>MOV</i>	<i>A,#00000100B</i>	;发送 RECV 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>CALL</i>	<i>WAIT</i>	
<i>MOV</i>	<i>DPTR,#I2CRXD</i>	;从数据缓冲区读取数据
<i>MOVX</i>	<i>A,@DPTR</i>	
<i>RET</i>		
SENDACK:		
<i>MOV</i>	<i>A,#00000000B</i>	;设置 ACK 信号
<i>MOV</i>	<i>DPTR,#I2CMSST</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>MOV</i>	<i>A,#00000101B</i>	;发送 ACK 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
SENDNAK:		
<i>MOV</i>	<i>A,#00000001B</i>	;设置 NAK 信号
<i>MOV</i>	<i>DPTR,#I2CMSST</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>MOV</i>	<i>A,#00000101B</i>	;发送 ACK 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
SENDSTOP:		
<i>MOV</i>	<i>A,#00000110B</i>	;发送 STOP 命令
<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>WAIT</i>	
WAIT:		
<i>MOV</i>	<i>DPTR,#I2CMSST</i>	;清中断标志
<i>MOVX</i>	<i>A,@DPTR</i>	
<i>JNB</i>	<i>ACC.6, WAIT</i>	
<i>ANL</i>	<i>A,#NOT 40H</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>RET</i>		
DELAY:		
<i>MOV</i>	<i>R0,#0</i>	
<i>MOV</i>	<i>RI,#0</i>	
DELAYI:		
<i>NOP</i>		
<i>DJNZ</i>	<i>RI,DELAYI</i>	
<i>DJNZ</i>	<i>R0,DELAYI</i>	
<i>RET</i>		

START:

```

MOV      SP, #5FH
ORL      P_SW2,#80H          ;使能访问 XFR，没有冲突不用关闭

MOV      P0M0, #00H
MOV      P0M1, #00H
MOV      P1M0, #00H
MOV      P1M1, #00H
MOV      P2M0, #00H
MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      A,#III00000B        ;设置 I2C 模块为主机模式
MOV      DPTR,#I2CCFG
MOVX    @DPTR,A
MOV      A,#00000000B
MOV      DPTR,#I2CMSST
MOVX    @DPTR,A

CALL    SENDSTART         ;发送起始命令
MOV      A,#0A2H
CALL    SENDATA           ;发送设备地址+写命令
CALL    RECVACK
MOV      A,#002H           ;发送存储地址
CALL    SENDATA
CALL    RECVACK
MOV      A,#00H            ;设置秒值
CALL    SENDATA
CALL    RECVACK
MOV      A,#00H            ;设置分钟值
CALL    SENDATA
CALL    RECVACK
MOV      A,#12H            ;设置小时值
CALL    SENDATA
CALL    RECVACK
CALL    SENDSTOP          ;发送停止命令

```

LOOP:

```

CALL    SENDSTART         ;发送起始命令
MOV      A,#0A2H           ;发送设备地址+写命令
CALL    SENDATA
CALL    RECVACK
MOV      A,#002H           ;发送存储地址
CALL    SENDATA
CALL    RECVACK
CALL    SENDSTART         ;发送起始命令
MOV      A,#0A3H           ;发送设备地址+读命令
CALL    SENDATA
CALL    RECVACK
CALL    RECVDATA          ;读取秒值
MOV      P0,A
CALL    SENDACK
CALL    RECVDATA          ;读取分钟值
MOV      P2,A
CALL    SENDACK

```

CALL	RECVDATA	;读取小时值
MOV	P3,A	
CALL	SENDNAK	
CALL	SENDSTOP	;发送停止命令
CALL	DELAY	
JMP	LOOP	
END		

24.5.4 I²C 从机模式（中断方式）

C 语言代码

//测试工作频率为11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

sbit SDA = P1^4;
sbit SCL = P1^5;

bit isda; //设备地址标志
bit isma; //存储地址标志
unsigned char addr; //接收地址
unsigned char xdata buffer[256]; //接收缓冲区

void I2C_Isr() interrupt 24
{
    if (I2CSLST & 0x40)
    {
        I2CSLST &= ~0x40; //处理 START 事件
        isda = 1; //若为重复起始信号时必须作此设置
    }
    else if (I2CSLST & 0x20)
    {
        I2CSLST &= ~0x20; //处理 RECV 事件
        if (isda) //处理 RECV 事件 (RECV DEVICE ADDR)
        {
            isda = 0; //处理 RECV 事件 (RECV MEMORY ADDR)
            addr = I2CRXD;
            I2CTXD = buffer[addr];
        }
        else
        {
            buffer[addr++] = I2CRXD; //处理 RECV 事件 (RECV DATA)
        }
    }
    else if (I2CSLST & 0x10)
    {
        I2CSLST &= ~0x10; //处理 SEND 事件
    }
}
```

```

if (I2CSLST & 0x02)
{
    I2CTXD = 0xff;                                //接收到NAK 则停止读取数据
}
else
{
    I2CTXD = buffer[addr];                         //接收到ACK 则继续读取数据
}
}
else if (I2CSLST & 0x08)
{
    I2CSLST &= ~0x08;                            //处理STOP 事件
    isda = 1;
    isma = 1;
}
}

void main()
{
    P_SW2 |= 0x80;                               //使能访问XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
    P2M0 = 0x00;
    P2M1 = 0x00;
    P3M0 = 0x00;
    P3M1 = 0x00;
    P4M0 = 0x00;
    P4M1 = 0x00;
    P5M0 = 0x00;
    P5M1 = 0x00;

    I2CCFG = 0x81;                             //使能I2C 从机模式
    I2CSLADR = 0x5a;                           //设置从机设备地址寄存器I2CSLADR=0101_1010B
                                                //即I2CSLADR[7:1]=010_1101B,MA=0B。
                                                //由于MA 为0,主机发送的设备地址必须与
                                                //I2CSLADR[7:1]相同才能访问此I2C 从机设备。
                                                //主机若需要写数据则要发送5AH(0101_1010B)
                                                //主机若需要读数据则要发送5BH(0101_1011B)

    I2CSLST = 0x00;
    I2CSLCR = 0x78;                           //使能从机模式中断
    EA = 1;

    isda = 1;                                  //用户变量初始化
    isma = 1;
    addr = 0;
    I2CTXD = buffer[addr];

    while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2 DATA 0BAH

<i>I2CCFG</i>	<i>XDATA</i>	<i>0FE80H</i>
<i>I2CMSCR</i>	<i>XDATA</i>	<i>0FE81H</i>
<i>I2CMSST</i>	<i>XDATA</i>	<i>0FE82H</i>
<i>I2CSLCR</i>	<i>XDATA</i>	<i>0FE83H</i>
<i>I2CSLST</i>	<i>XDATA</i>	<i>0FE84H</i>
<i>I2CSLADR</i>	<i>XDATA</i>	<i>0FE85H</i>
<i>I2CTXD</i>	<i>XDATA</i>	<i>0FE86H</i>
<i>I2CRXD</i>	<i>XDATA</i>	<i>0FE87H</i>
<i>SDA</i>	<i>BIT</i>	<i>P1.4</i>
<i>SCL</i>	<i>BIT</i>	<i>P1.5</i>
<i>ISDA</i>	<i>BIT</i>	<i>20H.0</i>
<i>ISMA</i>	<i>BIT</i>	<i>20H.1</i>
		;设备地址标志
		;存储地址标志
<i>ADDR</i>	<i>DATA</i>	<i>21H</i>
<i>PIM1</i>	<i>DATA</i>	<i>091H</i>
<i>PIM0</i>	<i>DATA</i>	<i>092H</i>
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>
	<i>ORG</i>	<i>0000H</i>
	<i>LJMP</i>	<i>START</i>
	<i>ORG</i>	<i>00C3H</i>
	<i>LJMP</i>	<i>I2CISR</i>
	<i>ORG</i>	<i>0100H</i>
<i>I2CISR:</i>		
	<i>PUSH</i>	<i>ACC</i>
	<i>PUSH</i>	<i>PSW</i>
	<i>PUSH</i>	<i>DPL</i>
	<i>PUSH</i>	<i>DPH</i>
	<i>MOV</i>	<i>DPTR,#I2CSLST</i>
		;检测从机状态
	<i>MOVX</i>	<i>A,@DPTR</i>
	<i>JB</i>	<i>ACC.6,STARTIF</i>
	<i>JB</i>	<i>ACC.5,RXIF</i>
	<i>JB</i>	<i>ACC.4,TXIF</i>
	<i>JB</i>	<i>ACC.3,STOPIF</i>
<i>ISRExit:</i>		
	<i>POP</i>	<i>DPH</i>
	<i>POP</i>	<i>DPL</i>
	<i>POP</i>	<i>PSW</i>
	<i>POP</i>	<i>ACC</i>
	<i>RETI</i>	
<i>STARTIF:</i>		
	<i>ANL</i>	<i>A,#NOT 40H</i>
		;处理 START 事件
	<i>MOVX</i>	<i>@DPTR,A</i>
	<i>SETB</i>	<i>ISDA</i>
	<i>JMP</i>	<i>ISRExit</i>
<i>RXIF:</i>		
	<i>ANL</i>	<i>A,#NOT 20H</i>
		;处理 RECV 事件

<i>MOVX</i>	<i>@DPTR,A</i>	
<i>MOV</i>	<i>DPTR,#I2CRXD</i>	
<i>MOVX</i>	<i>A,@DPTR</i>	
<i>JBC</i>	<i>ISDA,RXDA</i>	
<i>JBC</i>	<i>ISMA,RXMA</i>	
<i>MOV</i>	<i>R0,ADDR</i>	; 处理 RECV 事件 (RECV DATA)
<i>MOVX</i>	<i>@R0,A</i>	
<i>INC</i>	<i>ADDR</i>	
<i>JMP</i>	<i>ISREXIT</i>	
RXDA:		
<i>JMP</i>	<i>ISREXIT</i>	; 处理 RECV 事件 (RECV DEVICE ADDR)
RXMA:		
<i>MOV</i>	<i>ADDR,A</i>	; 处理 RECV 事件 (RECV MEMORY ADDR)
<i>MOV</i>	<i>R0,A</i>	
<i>MOVX</i>	<i>A,@R0</i>	
<i>MOV</i>	<i>DPTR,#I2CTXD</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>ISREXIT</i>	
TXIF:		
<i>ANL</i>	<i>A,#NOT 10H</i>	; 处理 SEND 事件
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JB</i>	<i>ACC.I,RXNAK</i>	
<i>INC</i>	<i>ADDR</i>	
<i>MOV</i>	<i>R0,ADDR</i>	
<i>MOVX</i>	<i>A,@R0</i>	
<i>MOV</i>	<i>DPTR,#I2CTXD</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>ISREXIT</i>	
RXNAK:		
<i>MOVX</i>	<i>A,#0FFH</i>	
<i>MOV</i>	<i>DPTR,#I2CTXD</i>	
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>JMP</i>	<i>ISREXIT</i>	
STOPIF:		
<i>ANL</i>	<i>A,#NOT 08H</i>	; 处理 STOP 事件
<i>MOVX</i>	<i>@DPTR,A</i>	
<i>SETB</i>	<i>ISDA</i>	
<i>SETB</i>	<i>ISMA</i>	
<i>JMP</i>	<i>ISREXIT</i>	
START:		
<i>MOV</i>	<i>SP, #5FH</i>	
<i>ORL</i>	<i>P_SW2,#80H</i>	; 使能访问 XFR，没有冲突不用关闭
<i>MOV</i>	<i>P0M0, #00H</i>	
<i>MOV</i>	<i>P0M1, #00H</i>	
<i>MOV</i>	<i>P1M0, #00H</i>	
<i>MOV</i>	<i>P1M1, #00H</i>	
<i>MOV</i>	<i>P2M0, #00H</i>	
<i>MOV</i>	<i>P2M1, #00H</i>	
<i>MOV</i>	<i>P3M0, #00H</i>	
<i>MOV</i>	<i>P3M1, #00H</i>	
<i>MOV</i>	<i>P4M0, #00H</i>	
<i>MOV</i>	<i>P4M1, #00H</i>	
<i>MOV</i>	<i>P5M0, #00H</i>	
<i>MOV</i>	<i>P5M1, #00H</i>	
<i>MOV</i>	<i>A,#10000001B</i>	; 使能 I2C 从机模式
<i>MOV</i>	<i>DPTR,#I2CCFG</i>	

MOVX	@DPTR,A	
MOV	A,#01011010B	;设置从机设备地址寄存器 I2CSLADR=0101_1010B
		;即 I2CSLADR[7:1]=010_1101B,MA=0B。
		;由于 MA 为 0, 主机发送的设备地址必须与
		I2CSLADR[7:1]相同才能访问此 I2C 从机设备。
		;主机若需要写数据则要发送 5AH(0101_1010B)
		;主机若需要读数据则要发送 5BH(0101_1011B)
MOV	DPTR,#I2CSLADR	
MOVX	@DPTR,A	
MOV	A,#00000000B	
MOV	DPTR,#I2CSLST	
MOVX	@DPTR,A	
MOV	A,#01111000B	;使能从机模式中断
MOV	DPTR,#I2CSLCR	
MOVX	@DPTR,A	
SETB	ISDA	;用户变量初始化
SETB	ISMA	
CLR	A	
MOV	ADDR,A	
MOV	R0,A	
MOVX	A,@R0	
MOV	DPTR,#I2CTXD	
MOVX	@DPTR,A	
SETB	EA	
SJMP	\$	
END		

24.5.5 I²C 从机模式（查询方式）

C 语言代码

//测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

sbit SDA = P1^4;
sbit SCL = P1^5;

bit isda; //设备地址标志
bit isma; //存储地址标志
unsigned char addr;
unsigned char xdata buffer[256];

void main()
{
    P_SW2 |= 0x80; //使能访问 XFR, 没有冲突不用关闭

    P0M0 = 0x00;
    P0M1 = 0x00;
    P1M0 = 0x00;
    P1M1 = 0x00;
```

```

P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

I2CCFG = 0x81;           //使能 I2C 从机模式
I2CSLADR = 0x5a;         //设置从机设备地址寄存器 I2CSLADR=0101_1010B
                          //即 I2CSLADR[7:1]=010_1101B,MA=0B。
                          //由于 MA 为 0, 主机发送的设备地址必须与
                          //I2CSLADR[7:1]相同才能访问此 I2C 从机设备。
                          //主机若需要写数据则要发送 5AH(0101_1010B)
                          //主机若需要读数据则要发送 5BH(0101_1011B)

I2CSLST = 0x00;
I2CSLCR = 0x00;          //禁止从机模式中断

isda = 1;                 //用户变量初始化
isma = 1;
addr = 0;
I2CTXD = buffer[addr];

while (1)
{
    if (I2CSLST & 0x40)
    {
        I2CSLST &= ~0x40;           //处理 START 事件
        isda = 1;                  //若为重复起始信号时必须作此设置
    }
    else if (I2CSLST & 0x20)
    {
        I2CSLST &= ~0x20;           //处理 RECV 事件
        if (isda)                  //处理 RECV 事件 (RECV DEVICE ADDR)
        {
            isda = 0;              //处理 RECV 事件 (RECV MEMORY ADDR)
            addr = I2CRXD;
            I2CTXD = buffer[addr];
        }
        else
        {
            buffer[addr++] = I2CRXD; //处理 RECV 事件 (RECV DATA)
        }
    }
    else if (I2CSLST & 0x10)
    {
        I2CSLST &= ~0x10;           //处理 SEND 事件
        if (I2CSLST & 0x02)
        {
            I2CTXD = 0xff;          //接收到 NAK 则停止读取数据
        }
        else
        {
            I2CTXD = buffer[++addr]; //接收到 ACK 则继续读取数据
        }
    }
}

```

```

        }
    }
    else if (I2CSLST & 0x08)
    {
        I2CSLST &= ~0x08;                                //处理 STOP 事件
        isda = 1;
        isma = 1;
    }
}
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH	
<i>I2CCFG</i>	<i>XDATA</i>	<i>0FE80H</i>	
<i>I2CMSCR</i>	<i>XDATA</i>	<i>0FE81H</i>	
<i>I2CMSST</i>	<i>XDATA</i>	<i>0FE82H</i>	
<i>I2CSLCR</i>	<i>XDATA</i>	<i>0FE83H</i>	
<i>I2CSLST</i>	<i>XDATA</i>	<i>0FE84H</i>	
<i>I2CSLADR</i>	<i>XDATA</i>	<i>0FE85H</i>	
<i>I2CTXD</i>	<i>XDATA</i>	<i>0FE86H</i>	
<i>I2CRXD</i>	<i>XDATA</i>	<i>0FE87H</i>	
SDA	BIT	P1.4	
SCL	BIT	P1.5	
ISDA	BIT	20H.0	; 设备地址标志
ISMA	BIT	20H.1	; 存储地址标志
ADDR	DATA	21H	
PIMI	DATA	091H	
PIM0	DATA	092H	
P0M1	DATA	093H	
P0M0	DATA	094H	
P2M1	DATA	095H	
P2M0	DATA	096H	
P3M1	DATA	0B1H	
P3M0	DATA	0B2H	
P4M1	DATA	0B3H	
P4M0	DATA	0B4H	
P5M1	DATA	0C9H	
P5M0	DATA	0CAH	
	ORG	0000H	
	LJMP	START	
	ORG	0100H	
START:	MOV	SP, #5FH	
	ORL	P_SW2,#80H	; 使能访问 XFR，没有冲突不用关闭
	MOV	P0M0, #00H	
	MOV	P0M1, #00H	
	MOV	P1M0, #00H	
	MOV	P1M1, #00H	
	MOV	P2M0, #00H	

```

MOV      P2M1, #00H
MOV      P3M0, #00H
MOV      P3M1, #00H
MOV      P4M0, #00H
MOV      P4M1, #00H
MOV      P5M0, #00H
MOV      P5M1, #00H

MOV      A,#10000001B          ;使能 I2C 从机模式
MOV      DPTR,#I2CCFG
MOVX    @DPTR,A
MOV      A,#01011010B          ;设置从机设备地址寄存器 I2CSLADR=0101_1010B
                                ;即 I2CSLADR[7:1]=010_1101B,MA=0B。
                                ;由于 MA 为 0, 主机发送的设备地址必须与
                                ;I2CSLADR[7:1]相同才能访问此 I2C 从机设备。
                                ;主机若需要写数据则要发送 5AH(0101_1010B)
                                ;主机若需要读数据则要发送 5BH(0101_1011B)

MOV      DPTR,#I2CSLADR
MOVX    @DPTR,A
MOV      A,#00000000B
MOV      DPTR,#I2CSLST
MOVX    @DPTR,A
MOV      A,#00000000B          ;禁止从机模式中断
MOV      DPTR,#I2CSLCR
MOVX    @DPTR,A

SETB    ISDA                  ;用户变量初始化
SETB    ISMA
CLR     A
MOV     ADDR,A
MOV     R0,A
MOVX   A,@R0
MOV     DPTR,#I2CTXD
MOVX    @DPTR,A

LOOP:
MOV      DPTR,#I2CSLST          ;检测从机状态
MOVX    A,@DPTR
JB      ACC.6,STARTIF
JB      ACC.5,RXIF
JB      ACC.4,TXIF
JB      ACC.3,STOPIF
JMP     LOOP

STARTIF:
ANL     A,#NOT 40H            ;处理 START 事件
MOVX    @DPTR,A
SETB    ISDA
JMP     LOOP

RXIF:
ANL     A,#NOT 20H            ;处理 RECV 事件
MOVX    @DPTR,A
MOV     DPTR,#I2CRXD
MOVX    A,@DPTR
JBC    ISDA,RXDA
JBC    ISMA,RXMA
MOV     R0,ADDR                ;处理 RECV 事件 (RECV DATA)
MOVX   @R0,A
INC    ADDR
JMP     LOOP

```

RXDA:

JMP	LOOP	; 处理 RECV 事件 (RECV DEVICE ADDR)
------------	-------------	---------------------------------

RXMA:

MOV	ADDR,A	; 处理 RECV 事件 (RECV MEMORY ADDR)
MOV	R0,A	
MOVX	A,@R0	
MOV	DPTR,#I2CTXD	
MOVX	@DPTR,A	
JMP	LOOP	

TXIF:

ANL	A,#NOT 10H	; 处理 SEND 事件
MOVX	@DPTR,A	
JB	ACC.I,RXNAK	
INC	ADDR	
MOV	R0,ADDR	
MOVX	A,@R0	
MOV	DPTR,#I2CTXD	
MOVX	@DPTR,A	
JMP	LOOP	

RXNAK:

MOVX	A,#0FFH	
MOV	DPTR,#I2CTXD	
MOVX	@DPTR,A	
JMP	LOOP	

STOPIF:

ANL	A,#NOT 08H	; 处理 STOP 事件
MOVX	@DPTR,A	
SETB	ISDA	
SETB	ISMA	
JMP	LOOP	

END

24.5.6 测试 I²C 从机模式代码的主机代码

C 语言代码

// 测试工作频率为 11.0592MHz

```
#include "stc8h.h"
#include "intrins.h"

sbit SDA = P1^4;
sbit SCL = P1^5;

void Wait()
{
    while (!(I2CMSST & 0x40));
    I2CMSST &= ~0x40;
}

void Start()
{
    I2CMSCR = 0x01; // 发送 START 命令
    Wait();
}
```

```
void SendData(char dat)
{
    I2CTXD = dat;                                //写数据到数据缓冲区
    I2CMSCR = 0x02;                                //发送 SEND 命令
    Wait();
}

void RecvACK()
{
    I2CMSCR = 0x03;                                //发送读 ACK 命令
    Wait();
}

char RecvData()
{
    I2CMSCR = 0x04;                                //发送 RECV 命令
    Wait();
    return I2CRXD;
}

void SendACK()
{
    I2CMSST = 0x00;                                //设置 ACK 信号
    I2CMSCR = 0x05;                                //发送 ACK 命令
    Wait();
}

void SendNAK()
{
    I2CMSST = 0x01;                                //设置 NAK 信号
    I2CMSCR = 0x05;                                //发送 ACK 命令
    Wait();
}

void Stop()
{
    I2CMSCR = 0x06;                                //发送 STOP 命令
    Wait();
}

void Delay()
{
    int i;

    for (i=0; i<3000; i++)
    {
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}

void main()
{
    P_SW2 |= 0x80;                                //使能访问 XFR, 没有冲突不用关闭
    P0M0 = 0x00;
```

```

P0M1 = 0x00;
P1M0 = 0x00;
P1M1 = 0x00;
P2M0 = 0x00;
P2M1 = 0x00;
P3M0 = 0x00;
P3M1 = 0x00;
P4M0 = 0x00;
P4M1 = 0x00;
P5M0 = 0x00;
P5M1 = 0x00;

I2CCFG = 0xe0; //使能 I2C 主机模式
I2CMSST = 0x00;

Start(); //发送起始命令
SendData(0x5a); //发送设备地址(010_1101B)+写命令(0B)
RecvACK();
SendData(0x00); //发送存储地址
RecvACK();
SendData(0x12); //写测试数据 1
RecvACK();
SendData(0x78); //写测试数据 2
RecvACK();
Stop(); //发送停止命令

Start(); //发送起始命令
SendData(0x5a); //发送设备地址(010_1101B)+写命令(0B)
RecvACK();
SendData(0x00); //发送存储地址高字节
RecvACK();
Start(); //发送起始命令
SendData(0x5b); //发送设备地址(010_1101B)+读命令(1B)
RecvACK();
P0 = RecvData(); //读取数据 1
SendACK();
P2 = RecvData(); //读取数据 2
SendNAK();
Stop(); //发送停止命令

while (1);
}

```

汇编代码

; 测试工作频率为 11.0592MHz

P_SW2	DATA	0BAH
I2CCFG	XDATA	0FE80H
I2CMSCR	XDATA	0FE81H
I2CMSST	XDATA	0FE82H
I2CSLCR	XDATA	0FE83H
I2CSLST	XDATA	0FE84H
I2CSLADR	XDATA	0FE85H
I2CTXD	XDATA	0FE86H
I2CRXD	XDATA	0FE87H
SDA	BIT	P1.4

<i>SCL</i>	<i>BIT</i>	<i>P1.5</i>	
<i>P1M1</i>	<i>DATA</i>	<i>091H</i>	
<i>P1M0</i>	<i>DATA</i>	<i>092H</i>	
<i>P0M1</i>	<i>DATA</i>	<i>093H</i>	
<i>P0M0</i>	<i>DATA</i>	<i>094H</i>	
<i>P2M1</i>	<i>DATA</i>	<i>095H</i>	
<i>P2M0</i>	<i>DATA</i>	<i>096H</i>	
<i>P3M1</i>	<i>DATA</i>	<i>0B1H</i>	
<i>P3M0</i>	<i>DATA</i>	<i>0B2H</i>	
<i>P4M1</i>	<i>DATA</i>	<i>0B3H</i>	
<i>P4M0</i>	<i>DATA</i>	<i>0B4H</i>	
<i>P5M1</i>	<i>DATA</i>	<i>0C9H</i>	
<i>P5M0</i>	<i>DATA</i>	<i>0CAH</i>	
	<i>ORG</i>	<i>0000H</i>	
	<i>LJMP</i>	<i>START</i>	
	<i>ORG</i>	<i>0100H</i>	
<i>SENDSTART:</i>			
	<i>MOV</i>	<i>A,#00000001B</i>	;发送 START 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>SENDDATA:</i>			
	<i>MOV</i>	<i>DPTR,#I2CTXD</i>	;写数据到数据缓冲区
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>A,#00000010B</i>	;发送 SEND 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>RECVACK:</i>			
	<i>MOV</i>	<i>A,#00000011B</i>	;发送读 ACK 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>RECVDATA:</i>			
	<i>MOV</i>	<i>A,#00000100B</i>	;发送 RECV 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>CALL</i>	<i>WAIT</i>	
	<i>MOV</i>	<i>DPTR,#I2CRXD</i>	;从数据缓冲区读取数据
	<i>MOVX</i>	<i>A,@DPTR</i>	
	<i>RET</i>		
<i>SENDACK:</i>			
	<i>MOV</i>	<i>A,#00000000B</i>	;设置 ACK 信号
	<i>MOV</i>	<i>DPTR,#I2CMSST</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>A,#00000101B</i>	;发送 ACK 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>JMP</i>	<i>WAIT</i>	
<i>SENDNAK:</i>			
	<i>MOV</i>	<i>A,#00000001B</i>	;设置 NAK 信号
	<i>MOV</i>	<i>DPTR,#I2CMSST</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	
	<i>MOV</i>	<i>A,#00000101B</i>	;发送 ACK 命令
	<i>MOV</i>	<i>DPTR,#I2CMSCR</i>	
	<i>MOVX</i>	<i>@DPTR,A</i>	

JMP	WAIT	
SENDSTOP:		
MOV	A,#00000110B	;发送 STOP 命令
MOV	DPTR,#I2CMSCR	
MOVX	@DPTR,A	
JMP	WAIT	
WAIT:		
MOV	DPTR,#I2CMSST	;清中断标志
MOVX	A,@DPTR	
JNB	ACC.6, WAIT	
ANL	A,#NOT 40H	
MOVX	@DPTR,A	
RET		
DELAY:		
MOV	R0,#0	
MOV	R1,#0	
DELAYI:		
NOP		
DJNZ	RI,DELAYI	
DJNZ	R0,DELAYI	
RET		
START:		
MOV	SP, #5FH	
ORL	P_SW2,#80H	;使能访问 XFR，没有冲突不用关闭
MOV	P0M0, #00H	
MOV	P0M1, #00H	
MOV	P1M0, #00H	
MOV	P1M1, #00H	
MOV	P2M0, #00H	
MOV	P2M1, #00H	
MOV	P3M0, #00H	
MOV	P3M1, #00H	
MOV	P4M0, #00H	
MOV	P4M1, #00H	
MOV	P5M0, #00H	
MOV	P5M1, #00H	
MOV	A,#11100000B	;设置 I2C 模块为主机模式
MOV	DPTR,#I2CCFG	
MOVX	@DPTR,A	
MOV	A,#00000000B	
MOV	DPTR,#I2CMSST	
MOVX	@DPTR,A	
CALL	SENDSTART	;发送起始命令
MOV	A,#5AH	;
CALL	SENDDATA	;发送设备地址(010_1101B)+写命令(0B)
CALL	RECVACK	
MOV	A,#000H	;发送存储地址
CALL	SENDDATA	
CALL	RECVACK	
MOV	A,#I2H	;写测试数据 1
CALL	SENDDATA	

<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#78H</i>	;写测试数据2
<i>CALL</i>	<i>SENDDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>SENDSTOP</i>	;发送停止命令
<i>CALL</i>	<i>DELAY</i>	;等待设备写数据
<i>CALL</i>	<i>SENDSTART</i>	;发送起始命令
<i>MOV</i>	<i>A,#5AH</i>	;发送设备地址(010_1101B)+写命令(0B)
<i>CALL</i>	<i>SENDDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>MOV</i>	<i>A,#000H</i>	;发送存储地址
<i>CALL</i>	<i>SENDDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>SENDSTART</i>	;发送起始命令
<i>MOV</i>	<i>A,#5BH</i>	;发送设备地址(010_1101B)+读命令(1B)
<i>CALL</i>	<i>SENDDATA</i>	
<i>CALL</i>	<i>RECVACK</i>	
<i>CALL</i>	<i>RECVDATA</i>	;读取数据1
<i>MOV</i>	<i>P0,A</i>	
<i>CALL</i>	<i>SENDACK</i>	
<i>CALL</i>	<i>RECVDATA</i>	;读取数据2
<i>MOV</i>	<i>P2,A</i>	
<i>CALL</i>	<i>SENDNAK</i>	
<i>CALL</i>	<i>SENDSTOP</i>	;发送停止命令
<i>JMP</i>	\$	
<i>END</i>		
