STC12C5410AD系列单片机器件手册

---1个时钟/机器周期8051

---超强加密，有全球唯一ID号

---高速，高可靠

---低功耗,超低价

---强抗静电，强抗干扰

STC12C5401AD,	STC12C5401
STC12C5402AD,	STC12C5402
STC12C5404AD,	STC12C5404
STC12C5406AD,	STC12C5406
STC12C5408AD,	STC12C5408
STC12C5410AD,	STC12C5410
STC12C5412AD,	STC12C5412

Update date: 2011/10/30

目录

第1章 STC12C5410AD系列单片机总体介绍

1.1 STC12C5410AD系列单片机简介

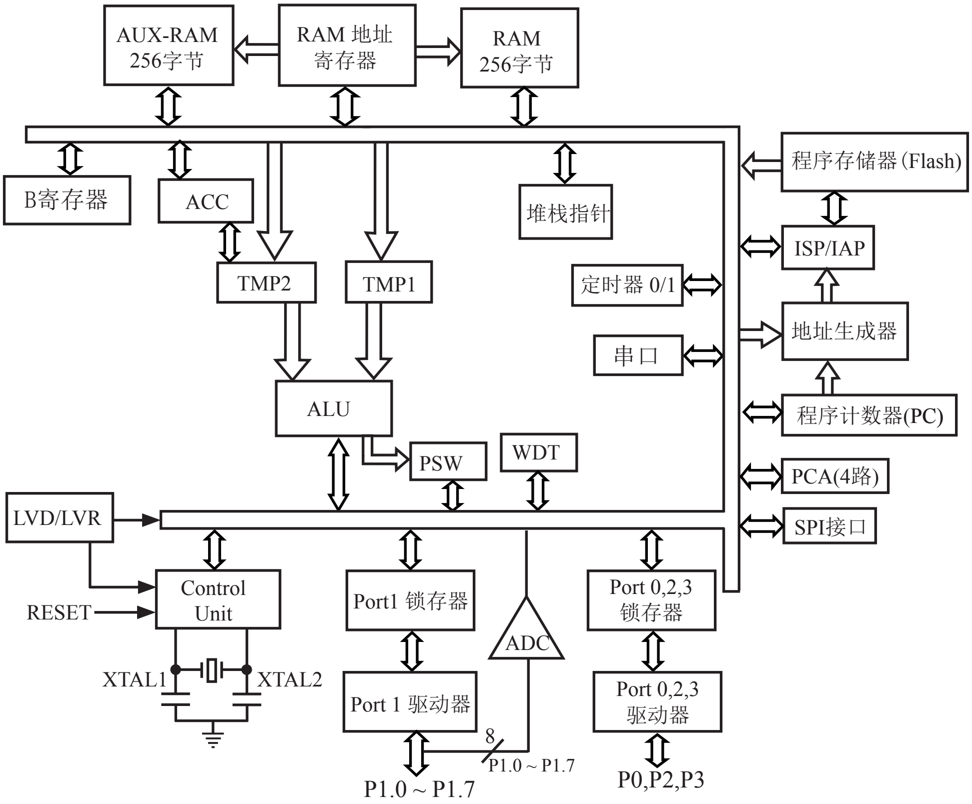
STC12C5410AD系列单片机是STC生产的单时钟/机器周期(1T)的单片机，是高速/低功耗/超强抗干扰的新一代8051单片机，指令代码完全兼容传统8051,但速度快8-12倍。内部集成MAX810专用复位电路,4路PWM,8路高速10位A/D转换,针对电机控制，强干扰场合。

1. 增强型 8051 CPU，1T，单时钟/机器周期，指令代码完全兼容传统8051
2. 工作电压：
   STC12C5410AD　系列工作电压：　5.5V - 3.5V（5V单片机）
   STC12LE5410AD 系列工作电压：　3.6V - 2.2V（3V单片机）
3. 工作频率范围：0～35MHz，相当于普通8051的 0～420MHz
4. 用户应用程序空间 12K /10K / 8K / 6K / 4K / 2K / 1K 字节......
5. 片上集成512字节 RAM
6. 通用I/O口（27/23/15个），复位后为：准双向口/弱上拉（普通8051传统I/O口）
   可设置成四种模式：准双向口/弱上拉，强推挽/强上拉，仅为输入/高阻，开漏
   每个I/O口驱动能力均可达到20mA，但整个芯片最大不要超过55mA
7. ISP（在系统可编程）/ IAP（在应用可编程），无需专用编程器，无需专用仿真器
   可通过串口（P3.0/P3.1）直接下载用户程序，数秒即可完成一片
8. 有EEPROM功能
9. 看门狗
10. 内部集成MAX810专用复位电路（外部晶体12M以下时，可省外部复位电路）
11. 时钟源：外部高精度晶体/时钟，内部R/C振荡器
    用户在下载用户程序时，可选择是使用内部R/C振荡器还是外部晶体/ 时钟
    常温下内部R/C振荡器频率为：5.2MHz ～ 6.8MHz
    精度要求不高时，可选择使用内部时钟，但因为有制造误差和温漂，以实际测试为准
12. 共6个16位定时器
    两个与传统8051兼容的定时器/计数器,16位定时器T0 和T1，没有定时器2，
    PCA模块可再实现4个16位定时器
13. 2个时钟输出口，可由T0的溢出在P1.0输出时钟，可由T1的溢出在P1.1输出时钟
14. 外部中断9路,下降沿中断或低电平触发中断,PCA模式可分别或同时支持上升沿中断/下降沿中断，Power Down模式可由外部中断唤醒，$\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, T0/P3.4, T1/P3.5, RxD/P3.0, PCA0/P3.7, PCA1/P3.5, PCA2/P2.0, PCA3/P2.4
15. PWM(4路）/PCA（可编程计数器阵列,4路）
    --- 也可用来当4路D/A使用
    --- 也可用来再实现4个定时器
    --- 也可用来再实现4个外部中断(上升沿中断/下降沿中断均可分别或同时支持)

16. A/D转换, 10位精度ADC，共8路
17. 通用全双工异步串行口(UART)，由于STC12系列是高速的8051，也可再用定时器软件实现多串口
18. SPI同步通信口，主模式/从模式
19. 工作温度范围：-40～+85℃(工业级) / 0～75℃(商业级)
21. 封装：LQFP-32, SOP-32/28/20, SKDIP-28, PDIP-20, TSSOP-20 (超小封装6.4mm×6.4mm), LQFP32/SOP32有27个I/O口，SOP28/SKDIP28有23个I/O口，SOP20/TSSOP20/PDIP20有15个I/O口，I/O口不够时，可用2到3根普通I/O口线外接74HC595/164/165（均可级联）来扩展I/O口，还可用A/D做按键扫描来节省I/O口，或用双CPU, 三线通信，还多了串口。

1.2 STC12C5410AD系列单片机的内部结构

STC12C5410AD系列单片机的内部结构框图如下图所示。STC12C5410AD单片机中包含中央处理器(CPU)、程序存储器(Flash)、数据存储器(SRAM)、定时/计数器、UART串口、I/O接口、高速A/D转换、SPI接口、PCA、看门狗及片内R/C振荡器和外部晶体振荡电路等模块。STC12C5410AD系列单片机几乎包含了数据采集和控制中所需的所有单元模块，可称得上一个片上系统。

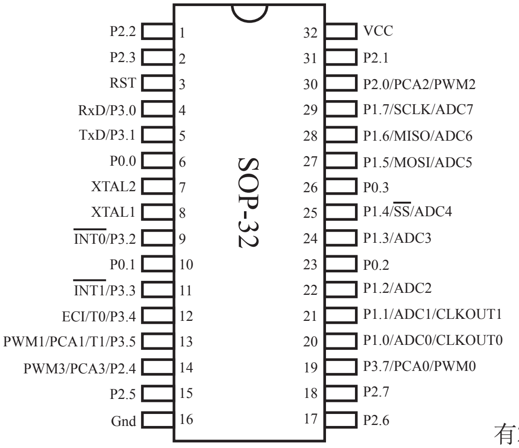


STC12C5410AD系列内部结构框图

1.3 STC12C5410AD系列单片机管脚图

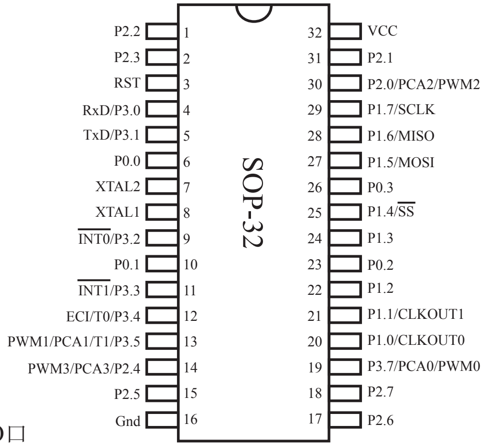
所有封装形式均满足欧盟RoHS要求，LQFP-32采用Green标准生产

强烈推荐选择SOP-32/28/20及LQFP-32贴片封装，尽量不选落后的插件DIP封装



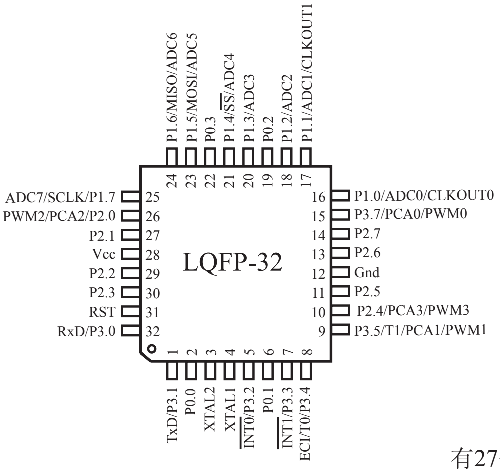
STC12C5410AD系列(有A/D转换)，SOP-32

有27个I/O口

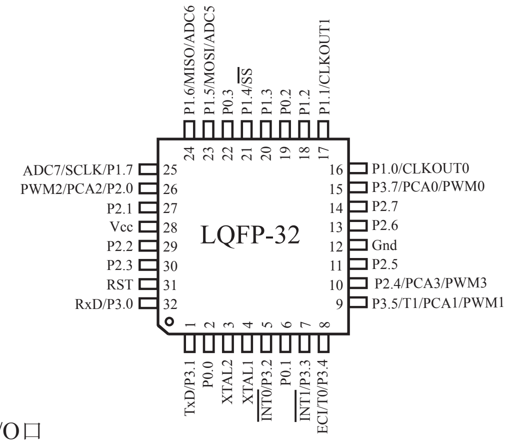


STC12C5410AD系列(无A/D转换)，SOP-32

LQFP-32封装，长×宽 = 9mm×9mm，高 < 1.6mm

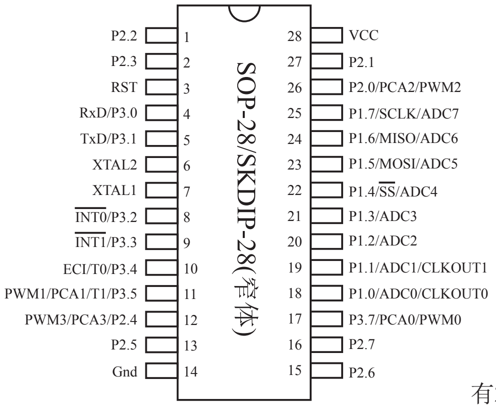


有27个I/O口

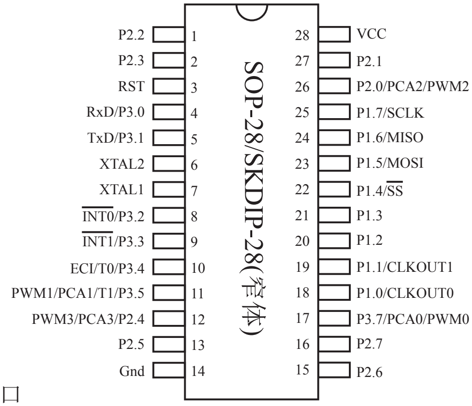


STC12C5410AD系列(有A/D转换)，LQFP-32

STC12C5410AD系列(无A/D转换)，LQFP-32

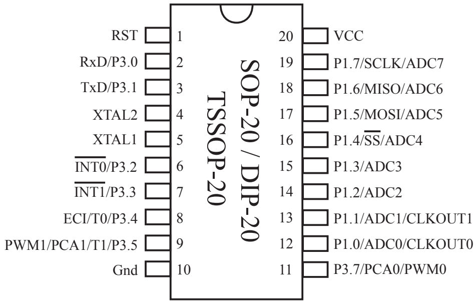


有23个I/O口

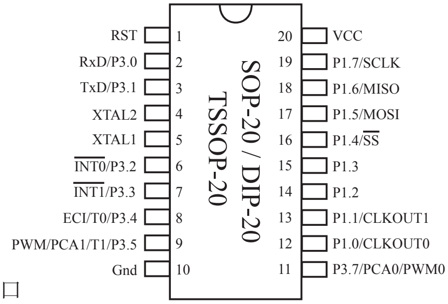


STC12C5410AD系列(有A/D转换)，28-Pin

STC12C5410AD系列(无A/D转换)，28-Pin



有15个I/O口



STC12C5410AD系列(有A/D转换)，20-Pin

STC12C5410AD系列(无A/D转换)，20-Pin

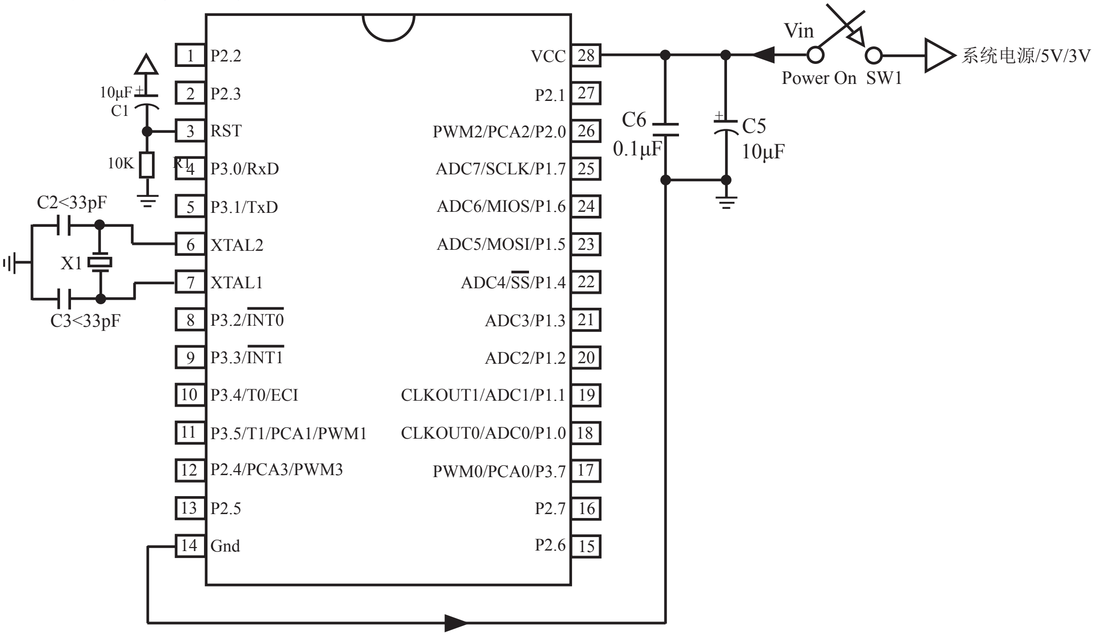
超小封装TSSOP-20, 6.4mm×6.4mm

1.4 STC12C5410AD系列单片机选型一览表

型号	工作电压（V）	Flash程序存储器字节	SRAM字节	定时器T0 T1	PCA定时器	时钟输出	UART串口	EEPROM	PCA 16位PWM 8位	A/D 8路	I/O	看门狗	内置复位	SPI	封装32-Pin	封装28-Pin	封装20-Pin
STC12C5410AD系列单片机选型一览																	
STC12C5401	5.5-3.5	1K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5401AD	5.5-3.5	1K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5402	5.5-3.5	2K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5402AD	5.5-3.5	2K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5404	5.5-3.5	4K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5404AD	5.5-3.5	4K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5406	5.5-3.5	6K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5406AD	5.5-3.5	6K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5408	5.5-3.5	8K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5408AD	5.5-3.5	8K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5410	5.5-3.5	10K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5410AD	5.5-3.5	10K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5412	5.5-3.5	12K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12C5412AD	5.5-3.5	12K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5410AD系列单片机选型一览																	
STC12LE5401	3.6-2.2	1K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5401AD	3.6-2.2	1K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5402	3.6-2.2	2K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5402AD	3.6-2.2	2K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5404	3.6-2.2	4K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5404AD	3.6-2.2	4K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5406	3.6-2.2	6K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5406AD	3.6-2.2	6K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5408	3.6-2.2	8K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5408AD	3.6-2.2	8K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5410	3.6-2.2	10K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5410AD	3.6-2.2	10K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5412	3.6-2.2	12K	512	有	4	有	有	有	4路		27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP
STC12LE5412AD	3.6-2.2	12K	512	有	4	有	有	有	4路	10位	27/23/15	有	有	有	SOP/LQFP	SOP/SKDIP	SOP/TSSOP/DIP

1.5 STC12C5410AD系列单片机最小应用系统

晶振频率在12MHz以下时：
可以不用C1, R1接1K电阻到地
不过建议保留R/C复位电路



关于晶振电路:

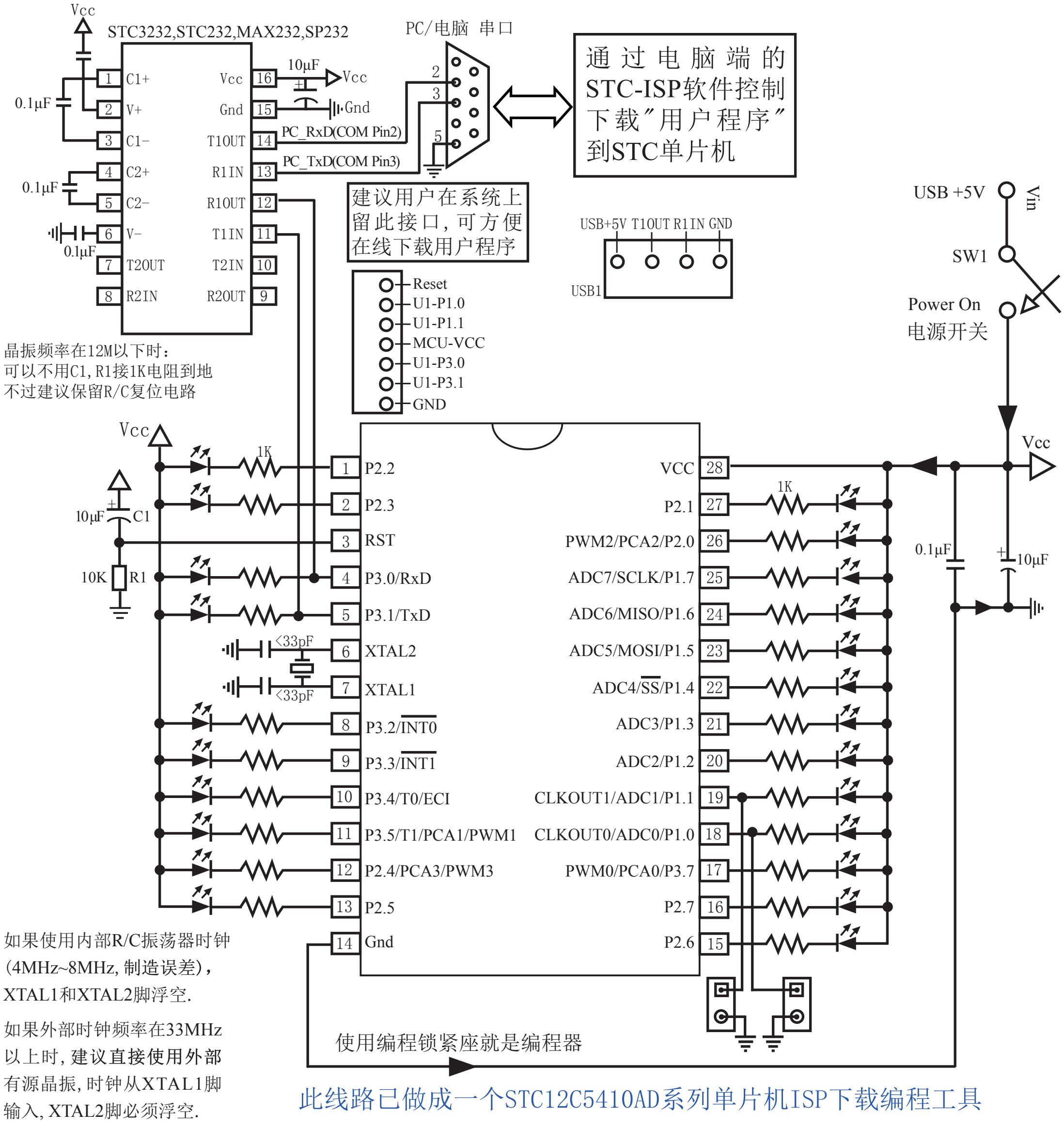
如果使用内部R/C振荡器时钟(4MHz～8MHz, 制造误差), XTAL1和XTAL2脚浮空.

如果外部时钟频率在33MHz以上时, 建议直接使用外部有源晶振, 时钟从XTAL1脚输入, XTAL2脚必须浮空.

1.6 STC12C5410AD系列在系统可编程(ISP)典型应用线路图

----通过RS-232转换器连接电脑就可以下载程序

1.6.1 STC12C5410AD系列单片机28脚典型应用电路

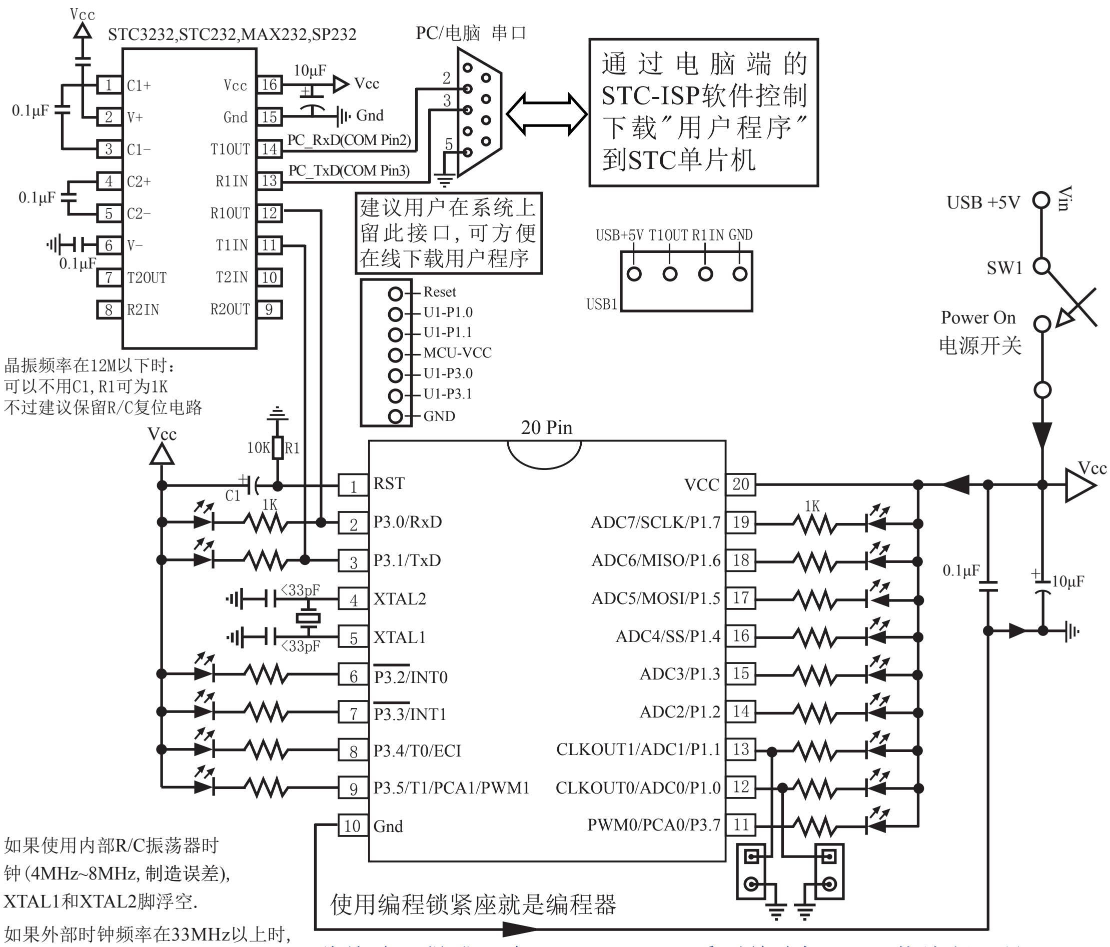


用户在自己的目标系统上，如将P3.0/P3.1经过RS-232电平转换器转换后连接到电脑的普通RS-232串口，就可以在系统编程/升级用户软件。建议如果用户板上无RS-232电平转换器，应引出一个插座，含Gnd/P3.1/P3.0/Vcc四个信号线，这样就可以在用户系统上直接编程了。当然如能引出Gnd/P3.1/P3.0/Vcc/P1.1/P1.0六个信号线为好，因为可以通过P1.0/P1.1禁止ISP下载程序。如果能将Gnd/P3.1/P3.0/Vcc/P1.1/P1.0/Reset七个信号线引出就更好了，这样可以很方便的使用“脱机下载板（ 无需电脑）”。

关于ISP编程的原理及应用指南详见“STC12C5410AD系列单片机开发/编程工具说明”部分。另外我们有标准化的编程下载工具，用户可以在上面编程后再插到目标系统上，也可以借用它上面的RS-232电平转换器连接到电脑，以做下载编程之用。编程一个芯片大致需几秒钟，速度比普通的通用编程器快很多，故无须买第三方的高价编程器。

电脑端STC-ISP软件从网站 STC官方网站下载

1.6.2 STC12C5410AD系列单片机20脚典型应用电路



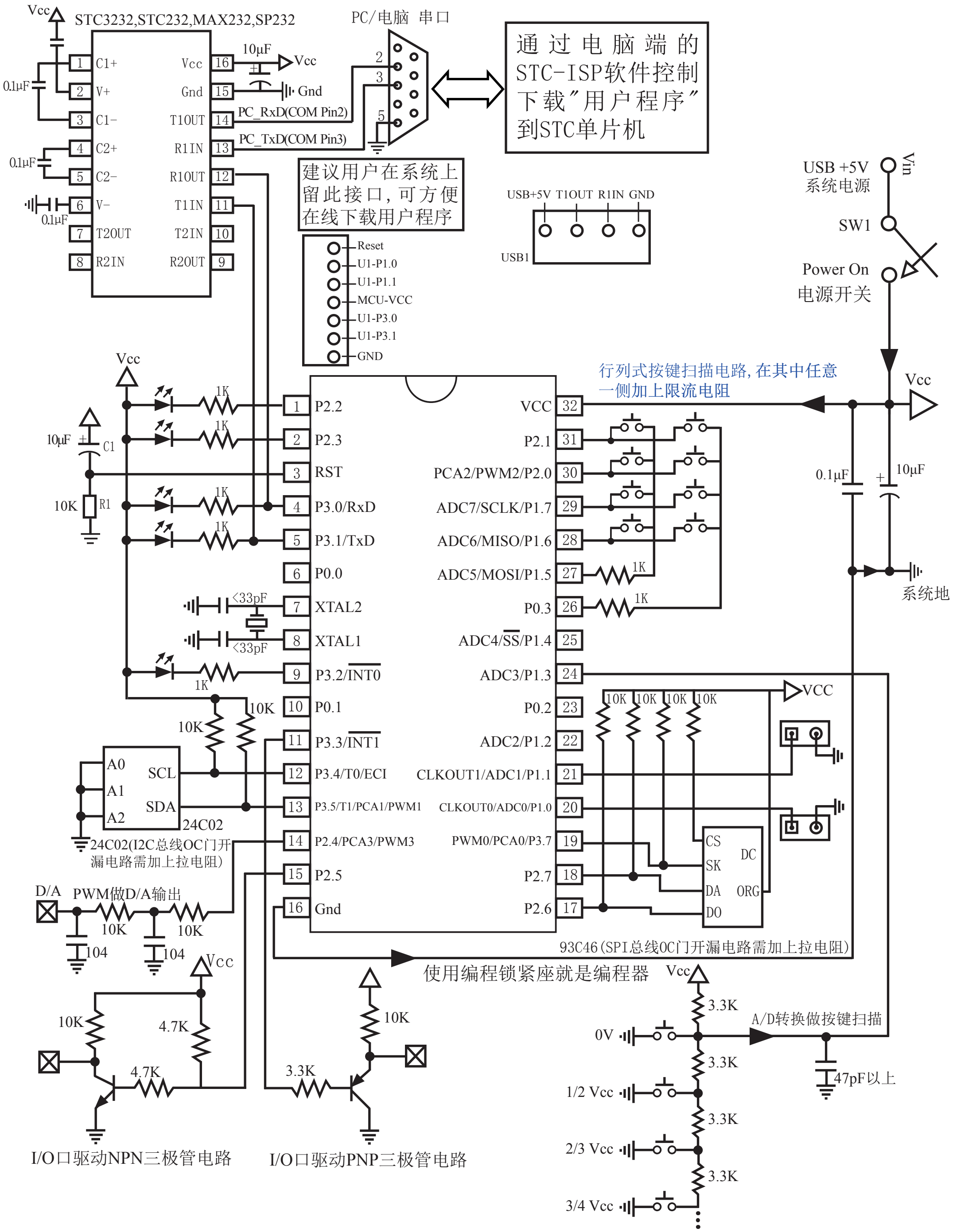
此线路已做成一个STC12C5410AD系列单片机ISP下载编程工具，
可直接赠送给客户

用户在自己的目标系统上，如将P3.0/P3.1经过RS-232电平转换器转换后连接到电脑的普通RS-232串口，就可以在系统编程/升级用户软件。建议如果用户板上无RS-232电平转换器，应引出一个插座，含Gnd/P3.1/P3.0/Vcc四个信号线，这样就可以在用户系统上直接编程了。当然如能引出Gnd/P3.1/P3.0/Vcc/P1.1/P1.0六个信号线为好，因为可以通过P1.0/P1.1禁止ISP下载程序。如果能将Gnd/P3.1/P3.0/Vcc/P1.1/P1.0/Reset七个信号线引出就更好了，这样可以很方便的使用“脱机下载板（ 无需电脑）”。

关于ISP编程的原理及应用指南详见“STC12C5410AD系列单片机开发/编程工具说明”部分。另外我们有标准化的编程下载工具，用户可以在上面编程后再插到目标系统上，也可以借用它上面的RS-232电平转换器连接到电脑，以做下载编程之用。编程一个芯片大致需几秒钟，速度比普通的通用编程器快很多，故无须买第三方的高价编程器。

电脑端STC-ISP软件从网站 STC官方网站下载

1.6.3 STC12C5410AD系列单片机32脚典型应用电路



1.7 STC12C5410AD系列管脚说明

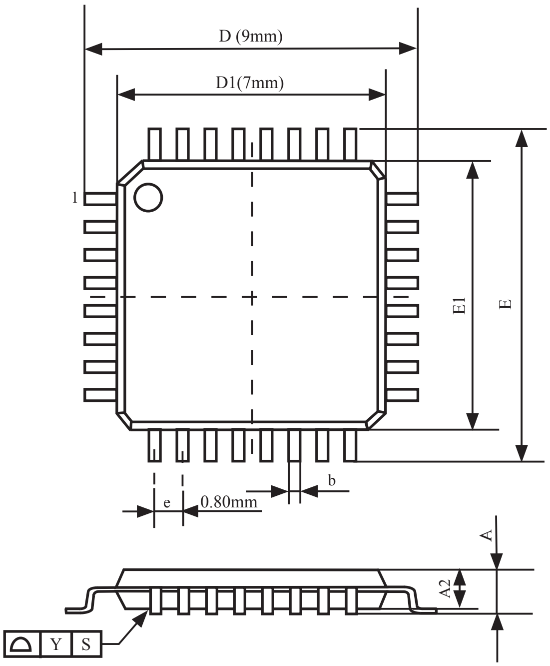
管脚	管脚编号				说明	
	LQFP32	SOP32	SOP28/SKDIP28	SOP20/DIP20/TSSOP20		
P0.0	2	6			标准I/O口 PORT0[0]	
P0.1	6	10			标准I/O口 PORT0[1]	
P0.2	19	23			标准I/O口 PORT0[2]	
P0.3	22	26			标准I/O口 PORT0[3]	
P1.0/ADC0/CLKOUT0	16	20	18	12	P1.0	标准I/O口 PORT1[0]
					ADC0	ADC 输入通道-0
					CLKOUT0	定时器/计数器0的时钟输出 可通过设置WAKE_CLKO[0]位/T0CLKO将该管脚配置为CLKOUT0
P1.1/ADC1/CLKOUT1	17	21	19	13	P1.1	标准I/O口 PORT1[1]
					ADC1	ADC 输入通道-1
					CLKOUT1	定时器/计数器1的时钟输出 可通过设置WAKE_CLKO[1]位/T1CLKO将该管脚配置为CLKOUT1
P1.2/ADC2	18	22	20	14	P1.2	标准I/O口 PORT1[2]
					ADC2	ADC 输入通道-2
P1.3/ADC3	20	24	21	15	P1.3	标准I/O口 PORT1[3]
					ADC3	ADC 输入通道-3
P1.4/ADC4/$\overline{SS}$	21	25	22	16	P1.4	标准I/O口 PORT1[4]
					ADC4	ADC 输入通道-4
					$\overline{SS}$	SPI同步串行接口的从机选择信号
P1.5/ADC5/MOSI	23	27	23	17	P1.5	标准I/O口 PORT1[5]
					ADC5	ADC 输入通道-5
					MOSI	SPI同步串行接口的主出从入(主器件的输出和从器件的输入)
P1.6/ADC6/MISO	24	28	24	18	P1.6	标准I/O口 PORT1[6]
					ADC5	ADC 输入通道-6
					MISO	SPI同步串行接口的主入从出(主器件的输入和从器件的输出)
P1.7/ADC7/SCLK	15	29	25	19	P1.7	标准I/O口 PORT1[7]
					ADC7	ADC 输入通道-7
					SCLK	SPI同步串行接口的时钟信号
P2.0/PCA2/PWM2	26	30	26		P2.0	标准I/O口 PORT2[0]
					PCA2	可编程阵列输出2
					PWM2	脉宽调制输出2

管脚	管脚编号				说明	
	LQFP32	SOP32	SOP28/SKDIP28	SOP20/DIP20/TSSOP-20		
P2.1	27	31	27		标准I/O口 PORT2[1]	
P2.2	29	1	1		标准I/O口 PORT2[2]	
P2.3	30	2	2		标准I/O口 PORT2[3]	
P2.4/PCA3/PWM3	10	14	12		P2.4	标准I/O口 PORT2[4]
					PCA3	可编程阵列输出3
					PWM3	脉宽调制输出3
P2.5	11	15	13		标准I/O口 PORT2[5]	
P2.6	13	17	15		标准I/O口 PORT2[6]	
P2.7	14	18	16		标准I/O口 PORT2[7]	
P3.0/RxD	32	4	4	2	P3.0	标准I/O口 PORT3[0]
					RxD	串口数据接收端
P3.1/TxD	1	5	5	3	P3.1	标准I/O口 PORT3[1]
					TxD	串口数据发送端
P3.2/$\overline{INT0}$	5	9	8	6	P3.2	标准I/O口 PORT3[2]
					$\overline{INT0}$	外部中断0, 下降沿中断或低电平中断
P3.3/$\overline{INT1}$	7	11	9	7	P3.3	标准I/O口 PORT3[3]
					$\overline{INT1}$	外部中断1, 下降沿中断或低电平中断
P3.4/T0/ECI	8	12	10	8	P3.4	标准I/O口 PORT3[4]
					T0	定时器/计数器0的外部输入
					ECI	PCA计数器的外部脉冲输入脚
P3.5/T1/PCA1/PWM1	9	13	11	9	P3.5	标准I/O口 PORT3[5]
					T1	定时器/计数器1的外部输入
					PCA1	可编程阵列输出1
					PWM1	脉宽调制输出1
P3.7/PCA0/PWM0	15	19	17	11	P3.7	标准I/O口 PORT3[7]
					PCA0	可编程阵列输出0
					PWM0	脉宽调制输出0
RST	31	3	3	1	复位脚	
XTAL1	4	8	7	5	内部时钟电路反相放大器输入端，接外部晶振的一个引脚。当直接使用外部时钟源时，此引脚是外部时钟源的输入端。	
XTAL2	3	7	6	4	内部时钟电路反相放大器的输出端，接外部晶振的另一端。当直接使用外部时钟源时，此引脚可浮空，此时XTAL2实际将XTAL1输入的时钟进行输出。	
VCC	28	32	28	20	电源正极	
Gnd	12	16	14	10	电源负极，接地	

1.8 STC12C5410AD系列单片机封装尺寸图

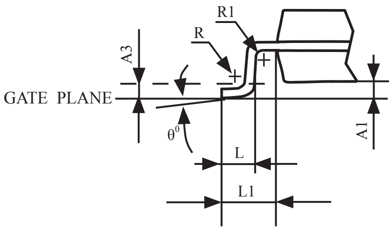
LQFP-32 封装尺寸图

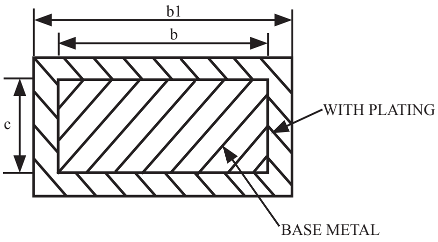
LQFP-32 OUTLINE PACKAGE



VARIATIONS (ALL DIMENSIONS SHOWN IN MM)

SYMBOLS	MIN.	NOM	MAX.
A	1.45	1.55	1.65
A1	0.01	-	0.21
A2	1.35	1.40	1.45
A3	-	0.254	-
D	8.80	9.00	9.20
D1	6.90	7.00	7.10
E	8.80	9.00	9.20
E1	6.90	7.00	7.10
e	0.80		
b	0.3	0.35	0.4
b1	0.31	0.37	0.43
c	-	0.127	-
L	0.43	-	0.71
L1	0.90	1.00	1.10
R	0.1	-	0.25
R1	0.1	-	-
θ^0	0^0	-	10^0





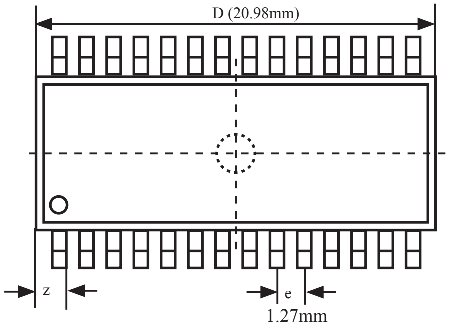
NOTES:
1. All dimensions are in mm
2. Dim D1 AND E1 does not include plastic flash.

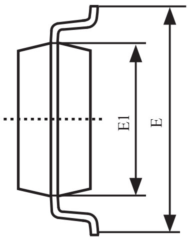
Flash:Plastic residual around body edge after de junk/singulation

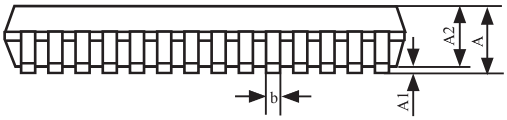
3. Dim b does not include dambar protrusion/ intrusion.
4. Plating thickness 0.05~0.015 mm.

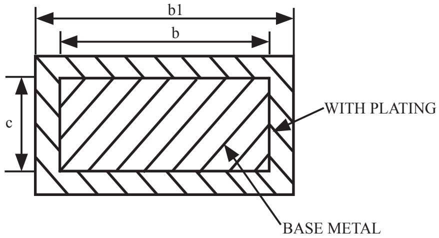
SOP-32 封装尺寸图

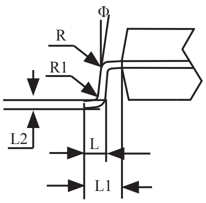
32-Pin Small Outline Package (SOP-32)
Dimensions in Millimeters







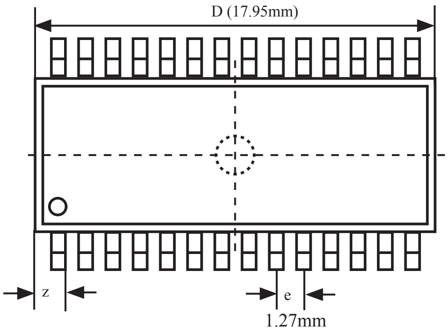


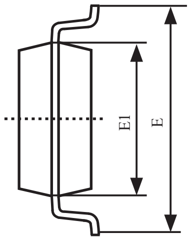


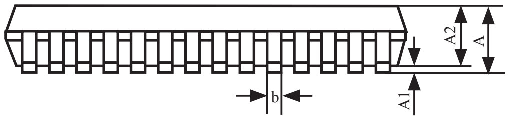
COMMON DIMENSIONS (UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b	0.356	0.406	0.456
b1	0.366	0.426	0.486
c	-	0.254	-
D	20.88	20.98	21.08
E	9.980	10.180	10.380
E1	7.390	7.500	7.600
e		1.27	
L	0.700	0.800	0.900
L1	1.303	1.403	1.503
L2	-	0.274	-
R	-	0.200	-
R1	-	0.300	-
Φ	0^0	-	10^0
z	-	0.745	-

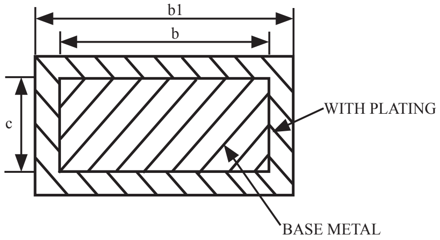
SOP-28 封装尺寸图

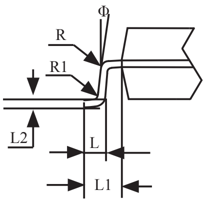
28-Pin Small Outline Package (SOP-28)
Dimensions in Millimeters







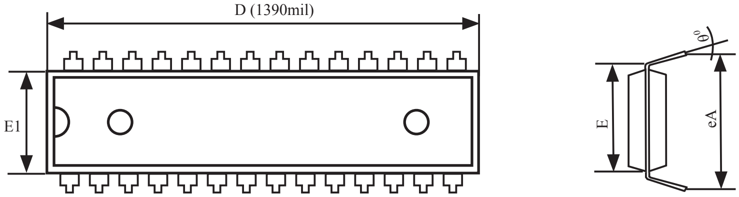


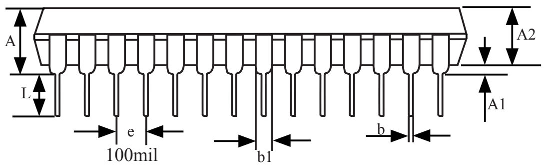


COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b	0.356	0.406	0.456
b1	0.366	0.426	0.486
c	-	0.254	-
D	17.750	17.950	18.150
E	10.100	10.300	10.500
E1	7.424	7.500	7.624
e		1.27	
L	0.764	0.864	0.964
L1	1.303	1.403	1.503
L2	-	0.274	-
R	-	0.200	-
R1	-	0.300	-
Φ	0^0	-	10^0
z	-	0.745	-

SKDIP-28 封装尺寸图

28-Pin Plastic Dual-In-line Package (SKDIP-28)
Dimensions in Inches and Millmeters



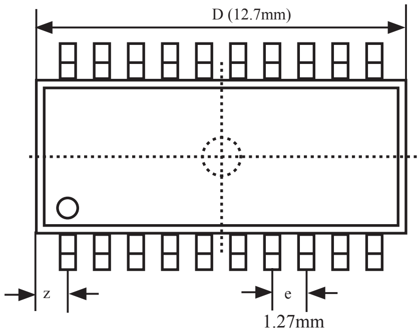


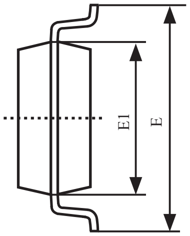
COMMON DIMENSIONS			
(UNITS OF MEASURE = INCH)			
SYMBOL	MIN	NOM	MAX
A	-	-	0.210
A1	0.015	-	-
A2	0.125	0.130	0.135
b	-	0.018	-
b1	-	0.060	-
D	1.385	1.390	1.40
E	-	0.310	-
E1	0.283	0.288	0.293
e	-	0.100	-
L	0.115	0.130	0.150
θ^0	0	7	15
eA	0.330	0.350	0.370

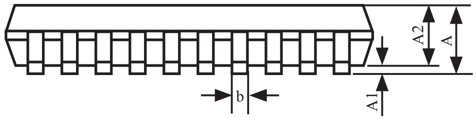
UNIT: INCH 1inch = 1000mil

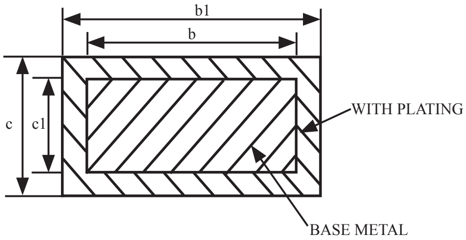
SOP-20 封装尺寸图

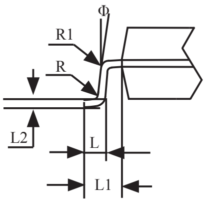
20-Pin Small Outline Package (SOP-20)
Dimensions in Inches and (Millimeters)







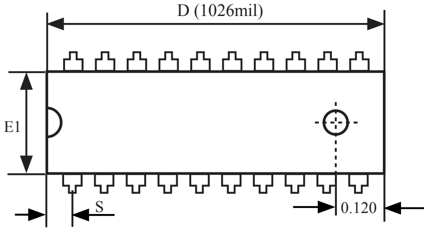


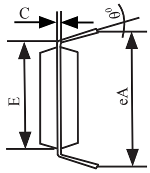


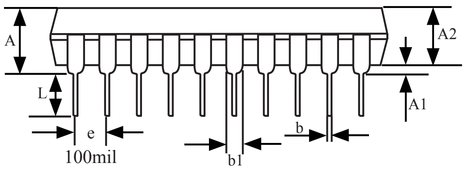
COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b1	0.366	0.426	0.486
b	0.356	0.406	0.456
c	0.234	-	0.274
c1	0.224	0.254	0.274
D	12.500	12.700	12.900
E	10.206	10.306	10.406
E1	7.450	7.500	7.550
e	1.270		
L	0.800	0.864	0.900
L1	1.303	1.403	1.503
L2	-	0.274	-
R	-	0.300	-
R1	-	0.200	-
Φ	0^0	-	10^0
z	-	0.660	-

PDIP-20 封装尺寸图

20-Pin Plastic Dual Inline Package (PDIP-20)
Dimensions in Inches







COMMON DIMENSIONS			
(UNITS OF MEASURE = INCH)			
SYMBOL	MIN	NOM	MAX
A	-	-	0.175
A1	0.015	-	-
A2	0.125	0.13	0.135
b	0.016	0.018	0.020
b1	0.058	0.060	0.064
C	0.008	0.010	0.11
D	1.012	1.026	1.040
E	0.290	0.300	0.310
E1	0.245	0.250	0.255
e	0.090	0.100	0.110
L	0.120	0.130	0.140
θ^0	0	-	15
eA	0.355	0.355	0.375
S	-	-	0.075

UNIT: INCH   1 inch = 1000 mil

1.9 STC12C5410AD系列单片机命名规则

STC12 xx 54 xx xx -- 35 x - xxxx xx

- xx：管脚数
  如 32、28、20
- xxxx：封装类型：
  如 PDIP，SOP，LQFP
- x：工作温度范围：
  I : 工业级, -40℃ ~ 85℃
  C : 商业级, 0℃ ~ 70℃
- 35：工作频率：
  35 : 工作频率可到35MHz
- xx：
  有AD字样 ： 有A/D转换功能
  无AD字样 ： 无A/D转换功能
- xx：程序空间大小，如:
  02是2K字节， 04是4K字节， 06是6K字节,
  08是8K字节， 10是10K字节，12是12K字节,
- 54：RAM是512字节，PCA/PWM是4路
- xx：工作电压
  C ： 5.5V~3.5V
  LE: 2.2V~3.6V
- STC12：STC 1T 8051，同样的工作频率时，速度是普通8051的8~12倍

1.10 每个单片机具有全球唯一身份证号码(ID号)

STC最新一代STC12C5410AD系列每一个单片机出厂时都具有全球唯一身份证号码(ID号)，用户可以在单片机上电后读取内部RAM单元从F1H - F7H 连续7个单元的值来获取此单片机的唯一身份证号码(ID号)，使用“ MOV @Ri” 指令来读取。如果用户需要用全球唯一ID号进行用户自己的软件加密，建议用户在程序的多个地方有技巧地判断自己的用户程序有无被非法修改，提高解密的难度，防止解密者修改程序，绕过对全球唯一ID号的判断。

//读内部ID号的C语言参考程序

```
/* ---   STC MCU Limited -------------------------------------------------- */
/* ---   STC   姚永平   2009/2/7   V1.0 ----------------------------------- */
/* ---   STC12C5201AD 系列单片机,软件实现自定义下载程序------------------ */
/* ---   本演示程序在STC-ISP Ver 3.0A.PCB的下载编程工具上测试通过 ----------- */
/* ---   如果要在程序中使用该程序,请在程序中注明使用了STC的资料及程序 - */
/* ---   如果要在文章中引用该程序,请在文章中注明使用了STC的资料及程序-- */
#include<reg51.h>
#include<intrins.h>
sfr      ISP_CONTR      = 0xE7;

sbit     MCU_Start_Led  = P1^7;
//unsigned        char     self_command_array[4] = {0x22,0x33,0x44,0x55};
#define  Self_Define_ISP_Download_Command          0x22
#define  RELOAD_COUNT          0xfb               //18.432MHz,12T,SMOD=0,9600bps

void     serial_port_initial();
void     send_UART(unsigned char);
void     UART_Interrupt_Receive(void);
void     soft_reset_to_ISP_Monitor(void);
void     delay(void);
void     display_MCU_Start_Led(void);

void main(void)
{
         unsigned  char i = 0;
         unsigned  char j = 0;

         unsigned char idata *idata_point;
```

```
	serial_port_initial();				//串口初始化
//	display_MCU_Start_Led();			//点亮发光二极管表示单片机开始工作
//	send_UART(0x34);				//串口发送数据表示单片机串口正常工作
//	send_UART(0xa7);				//串口发送数据表示单片机串口正常工作

	idata_point = 0xF1;
	for(j=0;j<=6; j++)
	{
		i = *idata_point;
		send_UART(i);
		idata_point++;
	}

	while(1);
}

void serial_port_initial()
{
	SCON	=	0x50;			//0101,0000 8位可变波特率，无奇偶校验位
	TMOD	=	0x21;			//0011,0001  设置顶时器1为8位自动重装计数器
	TH1	=	RELOAD_COUNT;		//设置定时器1自动重装数
	TL1	=	RELOAD_COUNT;
	TR1	=	1;			//开定时器1
	ES	=	1;			//允许串口中断
	EA	=	1;			//开总中断
}

void send_UART(unsigned char i)
{
	ES	=	0;			//关串口中断
	TI	=	0;			//清零串口发送完成中断请求标志
	SBUF	=	i;
	while(TI ==0);				//等待发送完成
	TI	=	0;			//清零串口发送完成中断请求标志
	ES	=	1;			//允许串口中断
}

void UART_Interrupt_Receive(void) interrupt 4
{
	unsigned char	k	=	0;
	if(RI==1)
	{
		RI	=	0;
		k	=	SBUF;
```

```
                if(k==Self_Define_ISP_Download_Command)          //是自定义下载命令
                {
                        delay();                                 //延时1秒就足够了
                        delay();                                 //延时1秒就足够了
                        soft_reset_to_ISP_Monitor();             //软复位到系统ISP监控区
                }
                send_UART(k);
        }
        else
        {
                TI  =  0;
        }
}


void soft_reset_to_ISP_Monitor(void)
{
        ISP_CONTR   =    0x60;                           //0110,0000   软复位到系统ISP监控区
}

void delay(void)
{
        unsigned int j  =   0;
        unsigned int g  =   0;
        for(j=0;j<5;j++)
        {
                for(g=0;g<60000;g++)
                {
                        _nop_();
                        _nop_();
                        _nop_();
                        _nop_();
                        _nop_();
                }
        }
}

void display_MCU_Start_Led(void)
{
        unsigned  char i = 0;
        for(i=0;i<3;i++)
        {
                MCU_Start_Led   =   0;                   //顶亮MCU开始工作指示灯
                delay();
                MCU_Start_Led   =   1;                   //熄灭MCU开始工作指示灯
                delay();
                MCU_Start_Led   =   0;                   //顶亮MCU开始工作指示灯
        }
}
```

第2章 时钟，省电模式及复位

2.1 STC12C5410AD系列单片机的时钟

2.1.1 STC12C5410AD系列单片机内部/外部工作时钟可选

STC12C5410AD系列是1T的8051单片机，系统时钟兼容传统8051。

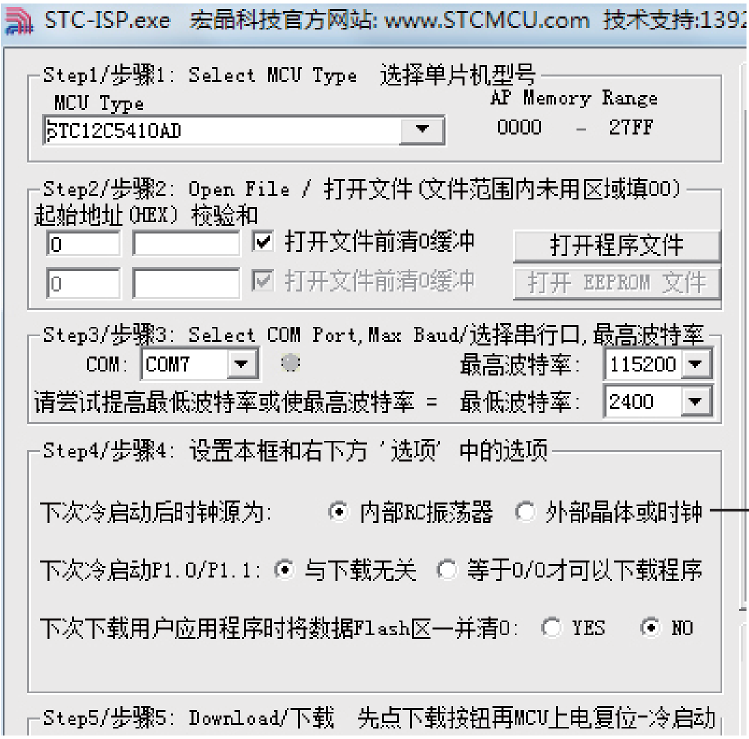
STC12C5410AD系列单片机有两个时钟源：内部R/C振荡时钟和外部晶体时钟。现STC-12C5410AD系列出厂标准配置是使用芯片外部晶体或时钟. 片内R/C振荡时钟在5V单片机常温下频率是5MHz~6.9MHz, 因为随着温度的变化，内部R/C振荡时钟的频率会有一些温飘, 再加上制造误差，故内部R/C振荡时钟只适用于对时钟频率要求不敏感的场合.

在对STC12C5410AD系列单片机进行ISP下载用户程序时，可以在选项中选择：

“下次冷启动后时钟源为外部晶体或时钟”

这样下载完用户程序后，停电，再冷启动后单片机的工作时钟使用的就不是内部R/C振荡器，而是外部晶体振荡后产生的高精度时钟了(接在XTAL1/XTAL2管脚上)，也可以直接从XTAL1脚输入外部时钟，XTAL2脚浮空。用户以后外部必须接晶体或时钟单片机才可以工作。

如果已被设置成用外部晶体或时钟工作的单片机，还要再设回使用内部R/C振荡器工作，则需给单片机外接晶体或时钟，再对STC12C5410AD系列单片机进行ISP下载用户程序时在选项中选择：



选择下次冷启动后时钟源为：
1. 内部R/C振荡器
2. 外部晶体或时钟

下载用户程序成功后，新的设置就设置进单片机内部了，但必须停电后再上电单片机才会用新的设置工作

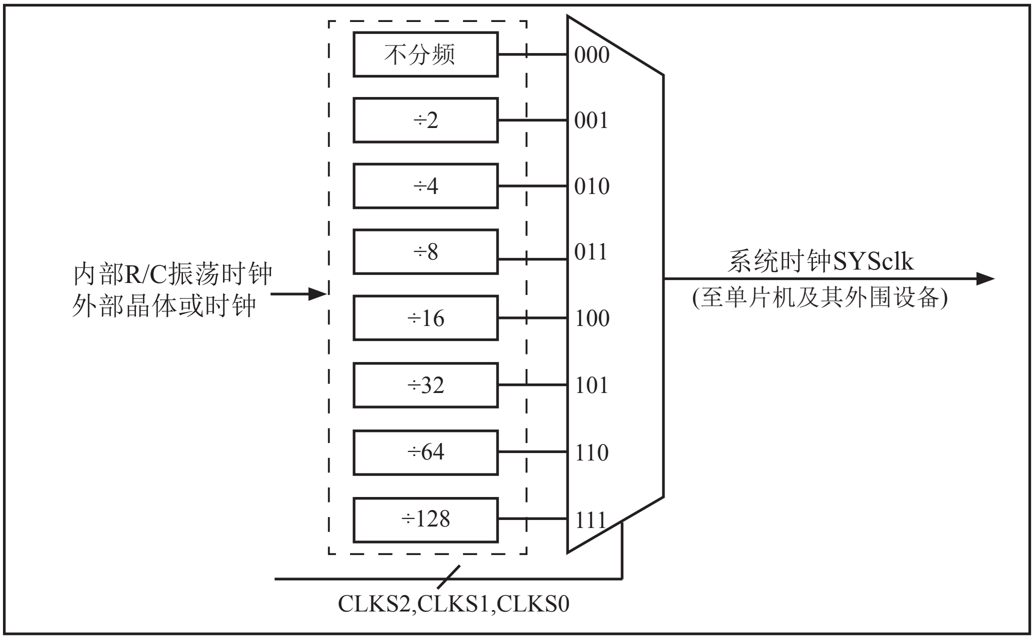
2.1.2 时钟分频及分频寄存器

如果希望降低系统功耗，可对时钟进行分频。利用时钟分频控制寄存器CLK_DIV可进行时钟分频，从而使单片机在较低频率下工作。

时钟分频寄存器CLK_DIV各位的定义如下：

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CLK_DIV	C7H	name	-	-	-	-	-	CLKS2	CLKS1	CLKS0

CLKS2	CLKS1	CLKS0	分频后CPU的实际工作时钟
0	0	0	外部晶体时钟或内部R/C振荡时钟
0	0	1	(外部晶体时钟或内部R/C振荡时钟)/2
0	1	0	(外部晶体时钟或内部R/C振荡时钟)/4
0	1	1	(外部晶体时钟或内部R/C振荡时钟)/8
1	0	0	(外部晶体时钟或内部R/C振荡时钟)/16
1	0	1	(外部晶体时钟或内部R/C振荡时钟)/32
1	1	0	(外部晶体时钟或内部R/C振荡时钟)/64
1	1	1	(外部晶体时钟或内部R/C振荡时钟)/128



时钟结构

STC12C5410AD系列单片机可以在空闲模式时分频工作，也可以在正常工作时分频。

2.1.3 如何知道单片机内部R/C振荡频率(内部时钟频率)

STC最新一代STC12C5410AD系列单片机除了可以使用传统的外部时钟外，还可以选择内部R/C振荡器时钟源(内部时钟). 如果选择单片机工作在内部R/C振荡器频率(内部时钟频率)，则可以省掉外部晶振。这时XTAL1/XTAL2浮空. 但由于使用内部时钟源误差较大, 所以在对时序要求较高或者有串行通信的情况下不建议使用内部R/C时钟源。在上电初始化程序时, 我们可以通过读取内部RAM单元（FCH, FDH, FEH, FFH连续四个单元)的值来获取单片机出厂时的内部R/C振荡器频率(内部时钟频率）。可以通过读取内部RAM单元（F8H,F9H,FAH,FBH连续四个单元)的值来获取用户最后一次使用内部R/C振荡器时钟下载程序时的频率(内部时钟频率), 使用“MOV @Ri”指令来读取。

```
//读内部R/C时钟频率的C语言参考程序
/* ---   STC MCU Limited -------------------------------------------------- */
/* ---   STC12C5201AD 系列单片机,软件实现自定义下载程序------------------ */
/* ---   本演示程序在STC-ISP Ver 3.0A.PCB的下载编程工具上测试通过 ----------- */
/* ---   如果要在程序中使用该程序,请在程序中注明使用了STC的资料及程序 - */
/* ---   如果要在文章中引用该程序,请在文章中注明使用了STC的资料及程序-- */
#include<reg51.h>
#include<intrins.h>
sfr     IAP_CONTR      = 0xC7;

sbit    MCU_Start_Led  = P1^7;
//unsigned        char     self_command_array[4] = {0x22,0x33,0x44,0x55};
#define Self_Define_ISP_Download_Command         0x22
#define RELOAD_COUNT             0xfb            //18.432MHz,12T,SMOD=0,9600bps

void    serial_port_initial();
void    send_UART(unsigned char);
void    UART_Interrupt_Receive(void);
void    soft_reset_to_ISP_Monitor(void);
void    delay(void);
void    display_MCU_Start_Led(void);

void main(void)
{
        unsigned  char i = 0;
        unsigned  char j = 0;

        unsigned char idata *idata_point;
```

```
        serial_port_initial();                          //串口初始化
//      display_MCU_Start_Led();                        //点亮发光二极管表示单片机开始工作
//      send_UART(0x34);                                //串口发送数据表示单片机串口正常工作
//      send_UART(0xa7);                                //串口发送数据表示单片机串口正常工作

        idata_point = 0xF8;
        for(j=0;j<=3;j++)
        {
                i = *idata_point;
                send_UART(i);
                idata_point++;
        }

        while(1);
}

void serial_port_initial()
{
        SCON    =  0x50;                                //0101,0000 8位可变波特率，无奇偶校验位
        TMOD    =  0x21;                                //0011,0001  设置顶时器1为8位自动重装计数器
        TH1     =  RELOAD_COUNT;                        //设置定时器1自动重装数
        TL1     =  RELOAD_COUNT;
        TR1     =  1;                                   //开定时器1
        ES      =  1;                                   //允许串口中断
        EA      =  1;                                   //开总中断
}

void send_UART(unsigned char i)
{
        ES      =  0;                                   //关串口中断
        TI      =  0;                                   //清零串口发送完成中断请求标志
        SBUF    =  i;
        while(TI ==0);                                  //等待发送完成
        TI      =  0;                                   //清零串口发送完成中断请求标志
        ES      =  1;                                   //允许串口中断
}

void UART_Interrupt_Receive(void) interrupt 4
{
        unsigned char   k   =   0;
        if(RI==1)
        {
                RI  =   0;
                k   =   SBUF;
```

```
                    if(k==Self_Define_ISP_Download_Command)            //是自定义下载命令
                    {
                              delay();                                  //延时1秒就足够了
                              delay();                                  //延时1秒就足够了
                              soft_reset_to_ISP_Monitor();              //软复位到系统ISP监控区
                    }
                    send_UART(k);
          }
          else
          {
                    TI  =  0;
          }
}

void soft_reset_to_ISP_Monitor(void)
{
          IAP_CONTR   =   0x60;                              //0110,0000   软复位到系统ISP监控区
}

void delay(void)
{
          unsigned int j  =   0;
          unsigned int g  =   0;
          for(j=0;j<5;j++)
          {
                    for(g=0;g<60000;g++)
                    {
                              _nop_();
                              _nop_();
                              _nop_();
                              _nop_();
                              _nop_();
                    }
          }
}

void display_MCU_Start_Led(void)
{
          unsigned  char i = 0;
          for(i=0;i<3;i++)
          {
                    MCU_Start_Led   =   0;                     //顶亮MCU开始工作指示灯
                    delay();
                    MCU_Start_Led   =   1;                     //熄灭MCU开始工作指示灯
                    delay();
                    MCU_Start_Led   =   0;                     //顶亮MCU开始工作指示灯
          }
}
```

2.1.4 可编程时钟输出

STC12C5410AD系列单片机有2三路可编程时钟输出:CLKOUT0/P1.0, CLKOUT1/P1.1

与可编程时钟输出有关的特殊功能寄存器:

AUXR : Auxiliary register

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

WAKE_CLKO :Clock output and Power-down Wakeup Control register

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO

特殊功能寄存器AUXR/WAKE_CLKO/BRT的C语言声明:

```
sfr     AUXR         =   0x8E;      //特殊功能寄存器AUXR的地址声明
sfr     WAKE_CLKO    =   0x8F;      //新增加特殊功能寄存器WAKE_CLKO的地址声明
```

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的汇编语言声明:

```
AUXR          EQU    8EH          ;特殊功能寄存器AUXR的地址声明
WAKE_CLKO     EQU    8FH          ;新增加的特殊功能寄存器WAKE_CLKO的地址声明
```

如何利用CLKOUT0/P1.0和CLKOUT1/P1.1管脚输出时钟:

CLKOUT0/P1.0和CLKOUT1/P1.1的时钟输出控制由WAKE_CLKO寄存器的T0CLKO位和T1CLKO位控制。CLKOUT0的输出时钟频率由定时器0控制, CLKOUT1的输出时钟频率由定时器1控制, 相应的定时器需要工作在定时器的模式2方式(8位自动重装载模式), 不要允许相应的定时器中断, 免得CPU反复进中断.

新增加的特殊功能寄存器: WAKE_CLKO(地址: 0x8F)

WAKE_CLKO :Clock output and Power-down Wakeup Control register(不可位寻址)

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO

B7 - PCAWAKEUP: 在掉电模式下, 是否允许PCA上升沿/下降沿中断唤醒powerdown。

0: 禁止PCA上升沿/下降沿中断唤醒powerdown;

1: 允许PCA上升沿/下降沿中断唤醒powerdown。

B6 - RXD_PIN_IE: 掉电模式下, 允许P3.0(RXD)下降沿置RI, 也能使RXD唤醒powerdown.

0: 禁止P3.0(RXD)下降沿置RI, 也禁止RXD唤醒powerdown;

1: 允许P3.0(RXD)下降沿置RI, 也允许RXD唤醒powerdown。

B5 - T1_PIN_IE：掉电模式下, 允许T1/P3.5脚下降沿置T1中断标志, 也能使T1脚唤醒powerdown.
0：禁止T1/P3.5脚下降沿置T1中断标志, 也禁止T1脚唤醒powerdown；
1：允许T1/P3.5脚下降沿置T1中断标志, 也允许T1脚唤醒powerdown 。

B4 - T0_PIN_IE：掉电模式下, 允许T0/P3.4脚下降沿置T0中断标志, 也能使T0脚唤醒powerdown.
0：禁止T0/P3.4脚下降沿置T0中断标志, 也禁止T0脚唤醒powerdown；
1：允许T0/P3.4脚下降沿置T0中断标志, 也允许T0脚唤醒powerdown 。

B1 - T1CLKO：是否允许将P1.1/ADC1脚配置为定时器T1的时钟输出CLKOUT1
1：允许将P1.1脚配置为定时器T1的时钟输出CLKOUT1，此时定时器T1只能工作在模式2(8位自动重装模式), CLKOUT1输出时钟频率= T1溢出率/2
如果$C/\overline{T}$=0，定时器/计数器T1是对内部系统时钟计数，则：
T1工作在1T模式时的输出频率 = SYSclk / (256 - TH1) / 2
T1工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH1) / 2
如果$C/\overline{T}$=1，定时器/计数器T1是对外部脉冲输入(P3.5/T1)计数，则：
输出时钟频率 = (T1_Pin_CLK) /(256 - TH1) / 2
0：不允许将P1.1脚配置为定时器T1的时钟输出CLKOUT1

B0 - T0CLKO：是否允许将P1.0/ADC0脚配置为定时器T0的时钟输出CLKOUT0
1：允许将P1.0脚配置为定时器T0的时钟输出CLKOUT0，此时定时器T0只能工作在模式2(8位自动重装模式), CLKOUT0输出时钟频率 = T0溢出率 / 2
如果$C/\overline{T}$=0，定时器/计数器T0是对内部系统时钟计数，则：
T0工作在1T 模式时的输出频率 = SYSclk / (256 - TH0) / 2
T0工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH0) / 2
如果$C/\overline{T}$=1，定时器/计数器T0是对外部脉冲输入(P3.4/T0)计数，则：
输出时钟频率 = (T0_Pin_CLK) / (256-TH0) / 2
0：不允许将P1.0脚配置为定时器T0的时钟输出CLKOUT0

特殊功能寄存器：AUXR(地址：0x8E)

AUXR : Auxiliary register(不可位寻址)

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

B7 - T0x12：定时器0速度控制位。

0：定时器0速度是8051单片机定时器的速度，即12分频；

1：定时器0速度是8051单片机定时器速度的12倍，即不分频。

B6 - T1x12：定时器1速度控制位。

0：定时器1速度是8051单片机定时器的速度，即12分频；

1：定时器1速度是8051单片机定时器速度的12倍，即不分频。

如果UART串口用T1作为波特率发生器，则由T1x12位决定UART串口是12T还是1T。

B5 - UART_M0x6：串口模式0的通信速度设置位。

0：UART串口模式0的速度是传统8051单片机串口的速度，即12分频；

1：UART串口模式0的速度是传统8051单片机串口速度的6倍，即2分频。

STC12C5201AD系列是 1T 的8051单片机，为了兼容传统8051，UART串口复位后是兼容传统8051的

B4 - EADCI：A/D中断允许控制位。

0：禁止A/D中断；

1：允许A/D中断。

B3 - ESPI：SPI中断允许控制位。

0：禁止SPI中断；

1：允许SPI中断。

B2 - ELVDI：低压检测中断允许控制位。

0：禁止低压检测中断；

1：允许低压检测中断。

5V单片机，3.7V以下为低压，3V单片机，2.4V以下为低压。如ELVDI=1(允许低压检测中断)，则会产生低压检测中断。

```
/* 本程序演示CLKOUT0/INT/T0/P3.4，CLKOUT1/INT/T1/P3.5，CLKOUT2/P1.0输出时钟演示程序*/
/* 时钟频率 SYSclk = 18.432MHz,T0, T1,独立波特率发生器均工作在12T 模式*/
#include"reg51.h"
sfr WAKE_CLKO = 0x8F;
sfr AUXR = 0x8E;

main()
{
/* 附加的 SFR WAKE_CLKO (地址：0x8F)
B7 - PCAWAKEUP : 允许 PCA 上升沿 / 下降沿中断 唤醒 powerdown。
B6 - RXD_PIN_IE: 1, 允许 RxD/P3.0(或RxD/P1.6)下降沿置RI，也能使RxD脚唤醒 powerdown。
B5 - T1_PIN_IE : 1, 允许 T1/P3.5脚下降沿置T1中断标志，也能使T1脚唤醒 powerdown。
B4 - T0_PIN_IE : 1, 允许 T0/P3.4脚下降沿置T0中断标志，也能使T0脚唤醒 powerdown。
B3 - N/A
B2 - N/A:
B1 - T1CLKO :
        1, 允许 P1.1 脚输出 T1(P3.5) 的溢出脉冲，输出时钟频率 = 1/2 T1 溢出率
                T1 工作在1T 模式时的输出频率CLKOUT1 =( SYSclk / 2 ) / ( 256 - TH1 )
                T1 工作在12T 模式时的输出频率CLKOUT1 =( SYSclk / 2 ) / 12 / ( 256 - TH1 )
        0, 不允许平P1.1脚输出 T1(P3.5) 溢出脉冲
B0 - T0CLKO :
        1, 允许P1.0脚输出 T0(P3.4) 溢出脉冲，输出时钟频率 = 1/2 T0 溢出率
                T0 工作在1T 模式时的输出频率CLKOUT0 =( SYSclk / 2 ) / ( 256 - TH0 )
                T0 工作在12T 模式时的输出频率CLKOUT0 =( SYSclk / 2 ) / 12 / ( 256 - TH0 )
        0，不允许P1.0脚输出 T0(P3.4) 溢出脉冲
*/

//      AUXR = 0xC0;                    //1T 模式
        WAKE_CLKO = (WAKE_CLKO|0x03);   //允许T0,T1
        TMOD = 0x22;                    //T0，T1 工作在模式2，8 位自动重装计数器
        TL0 = 0xFF;
        TH0 = 0xFF;
        TR0 = 1;                        //启动T0开始计数工作，对系统时钟进行分频输出
        TH1 = 0xFE;
        TL1 = 0xFE;
        TR1 = 1;                        //启动T1开始计数工作，对系统时钟进行分频输出
        while(1);
}
```

2.2 STC12C5410AD系列单片机的省电模式

STC12C5410AD系列单片机可以运行3种省电模式以降低功耗，它们分别是：空闲模式，低速模式和掉电模式。正常工作模式下，STC12C5410AD系列单片机的典型功耗是2.7mA ~ 7mA，而掉电模式下的典型功耗是<0.1uA，空闲模式下的典型功耗是<1.8mA.

低速模式由时钟分频器CLK_DIV控制，而空闲模式和掉电模式的进入由电源控制寄存器PCON的相应位控制。PCON寄存器定义如下：

PCON (Power Control Register) （不可位寻址）

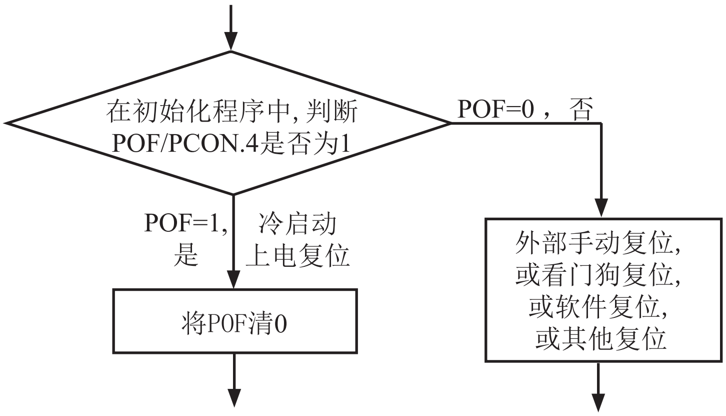
SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

LVDF : 低压检测标志位，同时也是低压检测中断请求标志位。

如果内部工作电压Vcc低于低压检测门槛电压，该位自动置1，与低压检测中断是否被允许无关。即在内部工作电压Vcc低于低压检测门槛电压时，不管有没有允许低压检测中断，该位都自动为1。该位要用软件清0，清0后，如内部工作电压Vcc继续低于低压检测门槛电压，该位又被自动设置为1。

POF : 上电复位标志位，单片机停电后，上电复位标志位为1 ，可由软件清0 。

实际应用：要判断是上电复位（冷启动），还是外部复位脚输入复位信号产生的复位，还是内部看门狗复位，还是软件复位或者其他复位，可通过如下方法来判断：



判断复位种类流程图

PD：将其置1时，进入Power Down模式，可由外部中断低电平触发或下降沿触发唤醒, 进入掉电模式时，内部时钟停振, 由于无时钟CPU、定时器、串行口等功能部件停止工作, 只有外部中断继续工作。可将CPU从掉电模式唤醒的外部管脚有：$\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, ECI/T0/P3.4, PWM1/PCA1/T1/P3.5, RxD/P3.0, PWM0/PCA0/P3.7, PWM2/PCA2/P2.0, PWM3/PCA3/P2.4。掉电模式也叫停机模式，此时功耗<0.1uA

IDL：将其置1，进入IDLE模式(空闲)，除系统不给CPU供时钟，CPU不执行指令外，其余功能部件仍可继续工作，可由外部中断、定时器中断、低压检测中断及A/D转换中断中的任何一个中断唤醒。可将CPU从空闲模式(IDLE 模式)唤醒的外部中断脚有:

$\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, T0/P3.4,T1/P3.5, RxD/P3.0,
内部定时器Timer0, Timer1也可以将单片机从空闲模式唤醒，
串行口中断(UART)也可以将单片机从空闲模式唤醒

GF1,GF0：两个通用工作标志位, 用户可以任意使用。

SMOD, SMOD0：与电源控制无关，与串口有关，在此不作介绍。

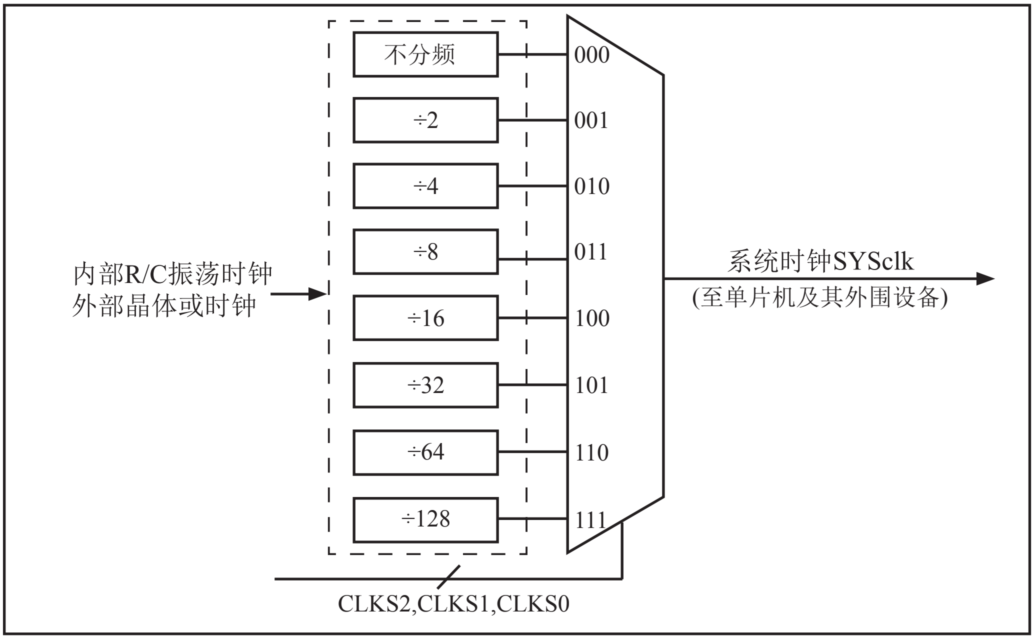
2.2.1 低速模式

时钟分频器可以对系统时钟(外部晶体时钟或内部R/C振荡时钟)进行分频，从而降低工作时钟频率，降低功耗，降低EMI。

时钟分频寄存器CLK_DIV各位的定义如下：

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CLK_DIV	C7H	name	-	-	-	-	-	CLKS2	CLKS1	CLKS0

CLKS2	CLKS1	CLKS0	分频后CPU的实际工作时钟
0	0	0	外部晶体时钟或内部R/C振荡时钟
0	0	1	(外部晶体时钟或内部R/C振荡时钟)/2
0	1	0	(外部晶体时钟或内部R/C振荡时钟)/4
0	1	1	(外部晶体时钟或内部R/C振荡时钟)/8
1	0	0	(外部晶体时钟或内部R/C振荡时钟)/16
1	0	1	(外部晶体时钟或内部R/C振荡时钟)/32
1	1	0	(外部晶体时钟或内部R/C振荡时钟)/64
1	1	1	(外部晶体时钟或内部R/C振荡时钟)/128



时钟结构

2.2.2 空闲模式

将IDL/PCON.0置为1，单片机将进入IDLE(空闲)模式。在空闲模式下，仅CPU无时钟停止工作，但是外部中断、外部低压检测电路、定时器、A/D转换、串行口等仍正常运行。而看门狗在空闲模式下是否工作取决于其自身有一个“IDLE ”模式位：IDLE_WDT(WDT_CONTR.3)。当IDLE_WDT位被设置为“1”时， 看门狗定时器在“空闲模式”计数，即正常工作。当IDLE_WDT位被清“0”时， 看门狗定时器在“空闲模式”时不计数，即停止工作。在空闲模式下，RAM、堆栈指针(SP)、程序计数器(PC)、程序状态字(PSW)、累加器(A)等寄存器都保持原有数据。I/O口保持着空闲模式被激活前那一刻的逻辑状态。空闲模式下单片机的所有外围设备都能正常运行(除CPU无时钟不工作外)。当任何一个中断产生时，它们都可以将单片机唤醒，单片机被唤醒后，CPU将继续执行进入空闲模式语句的下一条指令。

有两种方式可以退出空闲模式。任何一个中断的产生都会引起IDL/PCON.0被硬件清除，从而退出空闲模式。另一个退出空闲模式的方法是：外部RST引脚复位，将复位脚拉高，产生复位。这种拉高复位引脚来产生复位的信号源需要被保持24个时钟加上10us，才能产生复位，再将RST引脚拉低，结束复位，单片机从用户程序的0000H处开始正常工作。

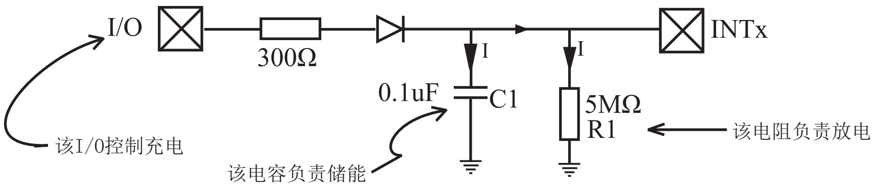
2.2.3 掉电模式/停机模式

将PD/PCON.1置为1，单片机将进入Power Down(掉电)模式，掉电模式也叫停机模式。进入掉电模式后，内部时钟停振，由于无时钟源，CPU、定时器、看门狗、A/D转换、串行口等停止工作，外部中断继续工作。如果低压检测电路被允许可产生中断，则低压检测电路也可继续工作，否则将停止工作。进入掉电模式后，所有I/O口、SFRs(特殊功能寄存器)维持进入掉电模式前那一刻的状态不变。

可将CPU从掉电模式唤醒的外部管脚有：$\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, ECI/T0/P3.4, PWM1/PCA1/T1/P3.5, RxD/P3.0, PWM0/PCA0/P3.7, PWM2/PCA2/P2.0, PWM3/PCA3/P2.4。

另外，外部复位也将MCU从掉电模式中唤醒，复位唤醒后的MCU将从用户程序的0000H处开始正常工作。

当用户系统无外部中断源将单片机从掉电模式唤醒时，下面的电路能够定时唤醒掉电模式。



控制充电的I/O口首先配置为推挽/强上拉模式并置高，上面的电路会给储能电容C1充电。在单片机进入掉电模式之前，将控制充电的I/O口拉低，上面电路通过电阻R1给储能电容C1放电。当电容C1的电被放到小于0.8V时，外部中断INTx会产生一个下降沿中断,从而自动地将单片机从掉电模式中唤醒。

2.2.4 由外部中断0唤醒掉电模式的测试程序(C程序和汇编程序)

1. C程序

```
/*由外部中断0唤醒掉电模式的示例程序 ---------------------------------------- */
/*---------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由外部中断0唤醒掉电模式 ---------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ----------------------------*/
/*---------------------------------------------------------------------------------------------------*/
#include "reg51.h"
#include "intrins.h"

//External interrupt0 service routine
void exint0() interrupt 0                    //interrupt 0 (location at 0003H)
{
}

void main()
{
        IT0 = 1;                             //set INT0 int type (1:Falling 0:Low level)
        EX0 = 1;                             //enable INT0 interrupt
        EA = 1;                              //open global interrupt switch

        while (1)
        {
                INT0 = 1;                    //ready read INT0 port
                while (!INT0);               //check INT0
                _nop_();
                _nop_();
                PCON = 0x02;                 //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序

```
/*由外部中断0唤醒掉电模式的示例程序 ---------------------------------------- */
/*--------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由外部中断0唤醒掉电模式 ---------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ----------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*--------------------------------------------------------------------------------------------------*/
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0003H                   ;interrupt 0 (location at 0003H)
        LJMP    EXINT0

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,#7FH                 ;initial SP
        SETB    IT0                     ;set INT0 int type (1:Falling 0:Low level)
        SETB    EX0                     ;enable INT0 interrupt
        SETB    EA                      ;open global interrupt switch
LOOP:
        SETB    INT0                    ;ready read INT0 port
        JNB     INT0,$                  ;check INT0
        NOP
        NOP
        MOV     PCON,#02H               ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;External interrupt0 service routine

EXINT0:
        RETI

;-----------------------------------------

        END
```

2.2.5 由外部中断1唤醒掉电模式的测试程序（C程序和汇编程序）

1. C程序

```
/*由外部中断1唤醒掉电模式的示例程序 ---------------------------------------- */
/*-------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由外部中断1唤醒掉电模式 ----------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ----------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ----------------------------*/
/*-------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

//External interrupt0 service routine
void exint1() interrupt 2                          //interrupt 2 (location at 0013H)
{
}

void main()
{
        IT1 = 1;                                   //set INT1 int type (1:Falling 0:Low level)
        EX1 = 1;                                   //enable INT1 interrupt
        EA = 1;                                    //open global interrupt switch

        while (1)
        {
                INT1 = 1;                          //ready read INT1 port
                while (!INT1);                     //check INT1
                _nop_();
                _nop_();
                PCON = 0x02;                       //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序

```
/*由外部中断1唤醒掉电模式的示例程序 ---------------------------------------- */
/*--------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由外部中断1唤醒掉电模式 ---------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*--------------------------------------------------------------------------------------------------*/

;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0013H                   ;interrupt 2 (location at 0013H)
        LJMP    EXINT1

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,#7FH                 ;initial SP
        SETB    IT1                     ;set INT1 int type (1:Falling 0:Low level)
        SETB    EX1                     ;enable INT1 interrupt
        SETB    EA                      ;open global interrupt switch
LOOP:
        SETB    INT1                    ;ready read INT1 port
        JNB     INT1,   $               ;check INT1
        NOP
        NOP
        MOV     PCON,   #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;External interrupt1 service routine

EXINT1:
        RETI
;-----------------------------------------

        END
```

2.2.6 由定时器0唤醒掉电模式的测试程序(C程序和汇编程序)

1. C程序

```
/*由定时器0唤醒掉电模式的示例程序 -------------------------------------------- */
/*---------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由定时器0唤醒掉电模式 -------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ----------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*---------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

sfr WAKE_CLKO = 0x8f;

//External interrupt0 service routine
void t0int() interrupt 1                        //interrupt 1 (location at 000BH)
{
}

void main()
{
        WAKE_CLKO = 0x10;                       //enable T0 falling edge wakeup MCU from power-down mode
        ET0 = 1;                                //enable T0 interrupt
        EA = 1;                                 //open global interrupt switch

        while (1)
        {
                T0 = 1;                         //ready read T0 port
                while (!T0);                    //check T0
                _nop_();
                _nop_();
                PCON = 0x02;                    //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序

```
/*由定时器0唤醒掉电模式的示例程序 ------------------------------------------- */
/*-----------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited --------------- --------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由定时器0唤醒掉电模式 -----------------------------*/
/*如果要在程序中使用或在文章中引用该程序 --------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 --------------------------*/
/*-------------------------------------------------------------------------------------------------*/

WAKE_CLKO      EQU       8FH
;-----------------------------------------
;interrupt vector table
            ORG      0000H
            LJMP     MAIN

            ORG      000BH                      ;interrupt 1 (location at 000BH)
            LJMP     T0INT
;-----------------------------------------
            ORG    0100H
MAIN:
            MOV      SP,#7FH                    ;initial SP
            MOV      WAKE_CLKO,    #10H         ;enable T0 falling edge wakeup MCU from power-down mode
            SETB     ET0                        ;enable T0 interrupt
            SETB     EA                         ;open global interrupt switch
LOOP:
            SETB     T0                         ;ready read T0 port
            JNB      T0,       $                ;check T0
            NOP
            NOP
            MOV      PCON,    #02H              ;MCU power down
            NOP
            NOP
            CPL      P1.0
            SJMP     LOOP
;-----------------------------------------
;T0 interrupt service routine

T0INT:
            RETI
;-----------------------------------------
            END
```

2.2.7 由定时器1唤醒掉电模式的测试程序(C程序和汇编程序)

1. C程序

```
/*由定时器1唤醒掉电模式的示例程序 ------------------------------------------- */
/*----------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited --------------- ---------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由定时器1唤醒掉电模式 ------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*----------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

sfr WAKE_CLKO = 0x8f;

//External interrupt0 service routine
void t1int() interrupt 3                    //interrupt 3 (location at 001BH)
{
}

void main()
{
        WAKE_CLKO = 0x20;                   //enable T1 falling edge wakeup MCU from power-down mode
        ET1 = 1;                            //enable T1 interrupt
        EA = 1;                             //open global interrupt switch

        while (1)
        {
                T1 = 1;                     //ready read T1 port
                while (!T1);                //check T1
                _nop_();
                _nop_();
                PCON = 0x02;                //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序

```
/*由定时器1唤醒掉电模式的示例程序 ------------------------------------------- */
/*-------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由定时器1唤醒掉电模式 ------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 --------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*-------------------------------------------------------------------------------------------------*/

WAKE_CLKO   EQU 8FH
;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     001BH                   ;interrupt 3 (location at 001BH)
        LJMP    T1INT
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,#7FH                 ;initial SP
        MOV     WAKE_CLKO,   #20H       ;enable T1 falling edge wakeup MCU from power-down mode
        SETB    ET1                     ;enable T1 interrupt
        SETB    EA                      ;open global interrupt switch
LOOP:
        SETB    T1                      ;ready read T1 port
        JNB     T1,$                    ;check T1
        NOP
        NOP
        MOV     PCON,#02H               ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP
;-----------------------------------------
;T1 interrupt service routine

T1INT:
        RETI
;-----------------------------------------
        END
```

2.2.8 由串行口RxD唤醒掉电模式的测试程序(C程序和汇编程序)

1. C程序

```
/*由串行口RxD唤醒掉电模式的示例程序 ---------------------------------------- */
/*------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由RxD唤醒掉电模式 ------------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ----------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the PCA */
sfr WAKE_CLKO   =   0x8F;

void uart_isr() interrupt 4 using 1
{
        if (RI)
        {
                RI = 0;
        }
}

void main()
{
        WAKE_CLKO = 0x40;             //enable RXD falling edge wakeup MCU from power-down mode
        ES = 1;
        EA = 1;

        while (1)
        {
                RXD = 1;                        //ready read RXD port
                while (!RXD);                   //check RXD
                _nop_();
                _nop_();
                PCON = 0x02;                    //MCU power down
                _nop_();
                _nop_();
                P2++;
        }
}
```

2. 汇编程序

```
/*由串行口RxD唤醒掉电模式的示例程序 --------------------------------------- */
/*-----------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited --------------- ---------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由RxD唤醒掉电模式 ------------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ----------------------------*/
/*-----------------------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
          WAKE_CLKO      EQU      8FH

;-------------------------------------------
          ORG       0000H
          LJMP      MAIN

          ORG       0023H
UART_ISR:
          JBC       RI,       EXIT                   ;clear RI flag
EXIT:
          RETI
;-------------------------------------------
          ORG       0100H
MAIN:
          MOV       WAKE_CLKO,     #40H              ;enable RXD falling edge wakeup MCU
                                                     ;from power-down mode
          SETB      ES
          SETB      EA

LOOP:
          SETB      RXD                              ;ready read RXD port
          JNB       RXD,      $                      ;check RXD
          NOP
          NOP
          MOV       PCON,     #02H                   ;MCU power down
          NOP
          NOP
          CPL       P1.0
          SJMP      LOOP
;-------------------------------------------
          END
```

2.2.9 由PCA唤醒掉电模式的测试程序(C程序和汇编程序)

1. C程序

```
/*由PCA唤醒掉电模式的示例程序 ------------------------------------------------- */
/*---------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由PCA唤醒掉电模式  -----------------------------------*/
/*如果要在程序中使用或在文章中引用该程序  ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*---------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the PCA */
sbit    EPCAI       =   IE^6;
sfr     WAKE_CLKO   =   0x8F;

sfr     CCON        =   0xD8;           //PCA control register
sbit    CCF0        =   CCON^0;         //PCA module-0 interrupt flag
sbit    CCF1        =   CCON^1;         //PCA module-1 interrupt flag
sbit    CR          =   CCON^6;         //PCA timer run control bit
sbit    CF          =   CCON^7;         //PCA timer overflow flag
sfr     CMOD        =   0xD9;           //PCA mode register
sfr     CL          =   0xE9;           //PCA base timer LOW
sfr     CH          =   0xF9;           //PCA base timer HIGH
sfr     CCAPM0      =   0xDA;           //PCA module-0 mode register
sfr     CCAP0L      =   0xEA;           //PCA module-0 capture register LOW
sfr     CCAP0H      =   0xFA;           //PCA module-0 capture register HIGH
sfr     CCAPM1      =   0xDB;           //PCA module-1 mode register
sfr     CCAP1L      =   0xEB;           //PCA module-1 capture register LOW
sfr     CCAP1H      =   0xFB;           //PCA module-1 capture register HIGH
sfr     CCAPM2      =   0xDC;           //PCA module-2 mode register
sfr     CCAP2L      =   0xEC;           //PCA module-2 capture register LOW
sfr     CCAP2H      =   0xFC;           //PCA module-2 capture register HIGH
sfr     CCAPM3      =   0xDD;           //PCA module-3 mode register
sfr     CCAP3L      =   0xED;           //PCA module-3 capture register LOW
sfr     CCAP3H      =   0xFD;           //PCA module-3 capture register HIGH
```

```
sfr     PCAPWM0   =  0xF2;
sfr     PCAPWM1   =  0xF3;
sfr     PCAPWM2   =  0xF4;
sfr     PCAPWM3   =  0xF5;

sbit    PCA_LED   =  P1^0;               //PCA test LED
sbit    CEX0      =  P3^7;

void PCA_isr() interrupt 7 using 1
{
        CCF0 = 0;                        //Clear interrupt flag
        PCA_LED = !PCA_LED;              //toggle the test pin while CEX0(P3.7) have a falling edge
}

void main()
{
        CCON = 0;                        //Initial PCA control register
                                         //PCA timer stop running
                                         //Clear CF flag
                                         //Clear all module interrupt flag
        CL = 0;                          //Reset PCA base timer
        CH = 0;
        CMOD = 0x00;                     //Set PCA timer clock source as Fosc/12
                                         //Disable PCA timer overflow interrupt
        CCAPM0 = 0x11;                   //PCA module-0 capture by a negative tigger on CEX0(P3.7)
                                         //and enable PCA interrupt
//      CCAPM0 = 0x21;                   //PCA module-0 capture by a rising edge on CEX0(P3.7)
                                         //and enable PCA interrupt
//      CCAPM0 = 0x31;                   //PCA module-0 capture by a transition (falling/rising edge)
                                         //on CEX0(P3.7) and enable PCA interrupt

        WAKE_CLKO = 0x80;                //enable PCA falling/raising edge wakeup MCU
                                         //from power-down mode
        CR = 1;                          //PCA timer start run
        EPCAI = 1;
        EA = 1;

        while (1)
        {
                CEX0 = 1;                //ready read CEX0 port
                while (!CEX0);           //check CEX0
                _nop_();
                _nop_();
                PCON = 0x02;             //MCU power down
                _nop_();
                _nop_();
                P2++;
        }
}
```

2. 汇编程序

```
/*由PCA唤醒掉电模式的示例程序 ------------------------------------------------- */
/*---------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------------*/
/* --- 演示STC 1T 系列单片机由PCA唤醒掉电模式 -----------------------------------*/
/*如果要在程序中使用或在文章中引用该程序 ---------------------------------------- */
/*请在程序中或文章中注明使用了STC的资料及程序 ---------------------------*/
/*---------------------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
EPCAI           BIT     IE.6
WAKE_CLKO       EQU     8FH

CCON            EQU     0D8H            ;PCA control register
CCF0            BIT     CCON.0          ;PCA module-0 interrupt flag
CCF1            BIT     CCON.1          ;PCA module-1 interrupt flag
CR              BIT     CCON.6          ;PCA timer run control bit
CF              BIT     CCON.7          ;PCA timer overflow flag
CMOD            EQU     0D9H            ;PCA mode register
CL              EQU     0E9H            ;PCA base timer LOW
CH              EQU     0F9H            ;PCA base timer HIGH
CCAPM0          EQU     0DAH            ;PCA module-0 mode register
CCAP0L          EQU     0EAH            ;PCA module-0 capture register LOW
CCAP0H          EQU     0FAH            ;PCA module-0 capture register HIGH
CCAPM1          EQU     0DBH            ;PCA module-1 mode register
CCAP1L          EQU     0EBH            ;PCA module-1 capture register LOW
CCAP1H          EQU     0FBH            ;PCA module-1 capture register HIGH
CCAPM2          EQU     0DCH            ;PCA module-2 mode register
CCAP2L          EQU     0ECH            ;PCA module-2 capture register LOW
CCAP2H          EQU     0FCH            ;PCA module-2 capture register HIGH
CCAPM3          EQU     0DDH            ;PCA module-3 mode register
CCAP3L          EQU     0EDH            ;PCA module-3 capture register LOW
CCAP3H          EQU     0FDH            ;PCA module-3 capture register HIGH

PCA_LED         BIT     P1.1            ;PCA test LED
CEX0            BIT     P3.7

;-----------------------------------------
                ORG     0000H
                LJMP    MAIN
```

```
        ORG     0033H
PCA_ISR:
        CLR     CCF0                    ;Clear interrupt flag
        CPL     PCA_LED                 ;toggle the test pin while CEX0(P3.7) have a falling edge
        RETI
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
        CLR     A                       ;
        MOV     CL,     A               ;Reset PCA base timer
        MOV     CH,     A               ;
        MOV     CMOD,   #00H            ;Set PCA timer clock source as Fosc/12
                                        ;Disable PCA timer overflow interrupt
        MOV     CCAPM0,#11H             ;PCA module-0 capture by a falling edge on CEX0(P3.7)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,#21H             ;PCA module-0 capture by a rising edge on CEX0(P3.7)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,#31H             ;PCA module-0 capture by a transition (falling/rising edge)
                                        ;on CEX0(P3.7) and enable PCA interrupt
;-------------------------------
        MOV     WAKE_CLKO,      #80H    ;enable PCA falling/raising edge wakeup MCU
                                        ;from power-down mode
        SETB    CR                      ;PCA timer start run
        SETB    EPCAI
        SETB    EA

LOOP:
        SETB    CEX0                    ;ready read CEX0 port
        JNB     CEX0,   $               ;check CEX0
        NOP
        NOP
        MOV     PCON,   #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP
;-----------------------------------------
        END
```

2.3 复位

STC12C5410AD系列单片机有5种复位方式：外部RST引脚复位，软件复位，上电复位，内部低压检测复位，MAX810专用复位电路复位，看门狗复位。

2.3.1 外部RST引脚复位

外部RST引脚复位就是从外部向RST引脚施加一定宽度的复位脉冲，从而实现单片机的复位。将RST复位管脚拉高并维持至少24个时钟加10us后，单片机会进入复位状态，将RST复位管脚拉回低电平后，单片机结束复位状态并从用户程序区的0000H处开始正常工作。

2.3.2 软件复位

用户应用程序在运行过程当中，有时会有特殊需求，需要实现单片机系统软复位（热启动之一），传统的8051单片机由于硬件上未支持此功能，用户必须用软件模拟实现，实现起来较麻烦。现STC新推出的增强型8051根据客户要求增加了IAP_CONTR特殊功能寄存器，实现了此功能。用户只需简单的控制IAP_CONTR特殊功能寄存器的其中两位SWBS/SWRST就可以系统复位了。

IAP_CONTR: ISP/IAP 控制寄存器

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
ISP_CONTR	E7H	name	ISPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0

ISPEN：ISP/IAP功能允许位。
　　0：禁止IAP读/写/擦除Data Flash/EEPROM；
　　1：允许读/写/擦除Data Flash/EEPROM。
SWBS：软件选择从用户应用程序区启动(0)，还是从ISP程序区启动(1)。要与SWRST直接配合才可以实现
SWRST：0：不操作；1：产生软件系统复位，硬件自动清零。
CMD_FAIL：如果送了ISP/IAP命令，并对ISP_TRIG送46h/B9h触发失败，则为1，需由软件清零.

```
;从用户应用程序区(AP区)软件复位并切换到用户应用程序区(AP区)开始执行程序
MOV ISP_CONTR, #00100000B ;SWBS = 0(选择AP区), SWRST = 1(软复位)
;从系统ISP监控程序区软件复位并切换到用户应用程序区(AP区)开始执行程序
MOV ISP_CONTR, #00100000B ;SWBS = 0(选择AP区), SWRST = 1(软复位)
;从用户应用程序区(AP区)软件复位并切换到系统ISP监控程序区开始执行程序
MOV ISP_CONTR, #01100000B ;SWBS = 1(选择ISP区), SWRST = 1(软复位)
;从系统ISP监控程序区软件复位并切换到系统ISP监控程序区开始执行程序
MOV ISP_CONTR, #01100000B ;SWBS = 1(选择ISP区), SWRST = 1(软复位)
```

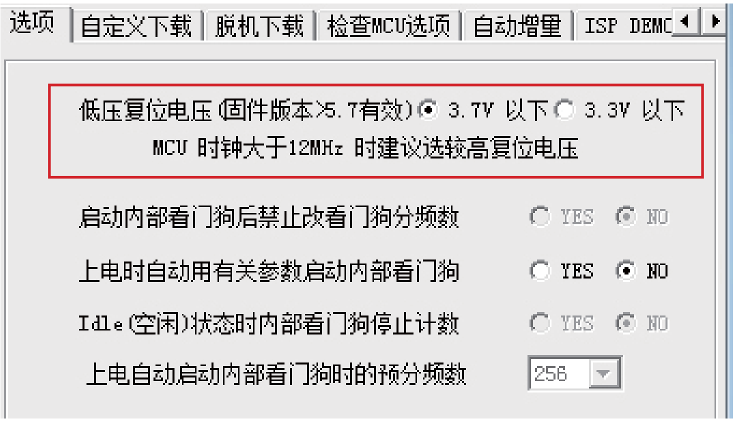
本复位是整个系统复位，所有的特殊功能寄存器都会复位到初始值，I/O口也会初始化

2.3.3 上电复位/掉电复位

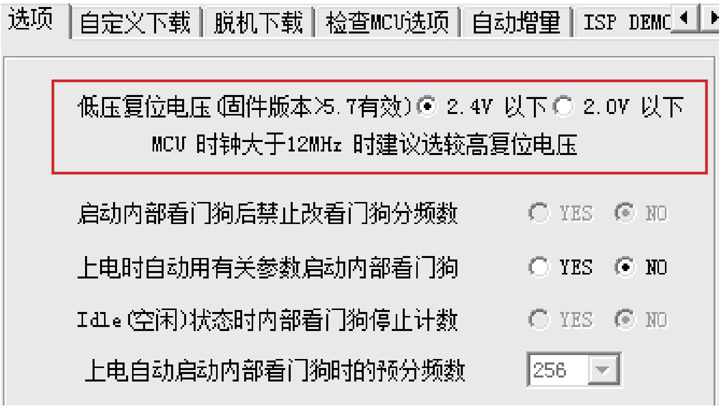
当电源电压VCC低于上电复位/掉电复位电路的检测门槛电压时，所有的逻辑电路都会复位。当VCC重新恢复正常电压时，延迟32768个时钟后，上电复位/掉电复位结束。。进入掉电模式时，上电复位/掉电复位功能被关闭。

2.3.4 内部低压检测复位

STC12C5410AD系列单片机内置低压检测复位，并且复位门槛电压可选（在STC-ISP编程器中选择）。5伏单片机内部低压检测复位门槛电压的选择如下图所示，低压复位电压可选3.7V和3.3V。



3伏单片机内部低压检测复位门槛电压的选择如下图所示，低压复位电压可选2.4V和2.0V。



当电源电压过低时，无法保证单片机正常工作。此时，可以利用单片机的内部部低压检测复位功能；当检测到电源电压低于门槛电压时，单片机复位，从而保证系统正常工作。

上电复位后低压检测标志位(LVDF/PCON.5)是1，要由软件清零（注意该位不可位寻址），建议清零后，再读一次该位是否为零，如为零，电源电压高于检测门槛电压。

与外部低压检测LVD有关的特殊功能寄存器表

Mnemonic	Add	Name	B7	B6	B5	B4	B3	B2	B1	B0	Reset Value
PCON	87H	Power Control	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011,0000
IE	A8H	Interrupt Enable	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000,0000
AUXR	8EH	Auxiliary Register	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000,00xx
IP	B8H	Interrupt Priority Low	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000,0000
IPH	B7H	Interrupt Priority High	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000,0000

内部低压检测相应的中断控制允许位是：EA/EPCA_LVD，ELVDI。
EPCA_LVD是PCA模块和低压检测中断的总中断允许位，此位打不开，时无法产生低压检测中断的。
ELVDI时低压检测中断的单独允许位。

内部低压检测相应的中断优先级控制位是：PPCA_LVDH/PPCA_LVD，0/0,0/1,1/0,1/1,四级中断优先级

内部低压检测相应的中断请求标志位是 ：LVDF，要由软件清零

与低压检测相关的一些寄存器：

PCON：电源控制寄存器(不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

LVDF：低压检测标志位，同时也是低压检测中断请求标志位。
如果内部工作电压Vcc低于低压检测门槛电压，该位自动置1，与低压检测中断是否被允许无关。即在内部工作电压Vcc低于低压检测门槛电压时，不管有没有允许低压检测中断，该位都自动为1。该位要用软件清0，清0后，如内部工作电压Vcc继续低于低压检测门槛电压，该位又被自动设置为1。

IE：中断允许寄存器(可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA：中断允许总控制位。
EA=0，屏蔽了所有的中断请求；
EA=1，开放总中断，但每个中断源还有自己的独立允许控制位。

EPCA_LVD：PCA模块中断和低压检测中断允许位。
EPCA_LVD = 0，禁止PCA模块和低压检测中断；
EPCA_LVD = 1，允许PCA模块和低压检测中断。

AUXR： 辅助寄存器(不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

ELVDI：低压检测中断单独允许位。

ELVDI = 0，禁止低压检测中断；

ELVDI = 1，允许低压检测中断。

如果要允许低压中断则需要将几个相应的控制位置1:

1、将ELVDI置1，允许低压检测中断，这是低压中断的单独控制位。

2、将EPCA_LVD置1，允许PCA模块中断及低压检测中断，这是PCA模块中断及低压检测中断的总中断控制位，此位不打开，也是无法产生低压检测中断的。

3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生低压检测中断的

低压检测中断服务程序中要用软件清低压中断请求标志位LVDF。

5V单片机，3.7V（±0.1）以下为低压，3V单片机，2.4V（± 0.1）以下为低压，

如ELVDI=1（允许低压中断），则会产生低压中断.

IPH： 中断优先级控制寄存器高(不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IPH	B7H	name	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H

IP： 中断优先级控制寄存器低(可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	B8H	name	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0

PPCA_LVDH, PPCA_LVD：PCA模块中断和低压检测中断优先级控制位。

当PPCA_LVDH = 0且PPCA_LVD = 0时，PCA模块中断和低压检测中断为最低优先级中断(优先级0)

当PPCA_LVDH = 0且PPCA_LVD = 1时，PCA模块中断和低压检测中断为较低优先级中断(优先级1)

当PPCA_LVDH = 1且PPCA_LVD = 0时，PCA模块中断和低压检测中断为较高优先级中断(优先级2)

当PPCA_LVDH = 1且PPCA_LVD = 1时，PCA模块中断和低压检测中断为较高优先级中断(优先级3)

2.3.5 MAX810专用复位电路

STC12C5410AD系列单片机内部集成了MAX810专用复位电路。若MAX810专用复位电路在STC-ISP编程器中被允许，则以后上电复位后将再产生约200mS延迟，复位才能被解除。

2.3.6 看门狗(WDT)复位

在工业控制/ 汽车电子/ 航空航天等需要高可靠性的系统中, 为了防止“系统在异常情况下，受到干扰，MCU/CPU程序跑飞，导致系统长时间异常工作”, 通常是引进看门狗, 如果MCU/CPU 不在规定的时间内按要求访问看门狗, 就认为MCU/CPU处于异常状态, 看门狗就会强迫MCU/CPU复位, 使系统重新从头开始按规律执行用户程序。STC12C5410AD系列单片机内部也引进了此看门狗功能， 使单片机系统可靠性设计变得更加方便/简洁。为此功能，我们增加如下特殊功能寄存器WDT_CONTR：

WDT_CONTR：看门狗(Watch-Dog-Timer)控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WDT_CONTR	E1H	name	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0

Symbol符号Function功能

WDT_FLAG : When WDT overflows， this bit is set. It can be cleared by software.
看门狗溢出标志位, 当溢出时，该位由硬件置1，可用软件将其清0。

EN_WDT : Enable WDT bit. When set, WDT is started
看门狗允许位， 当设置为“1 ”时，看门狗启动。

CLR_WDT : WDT clear bit. If set, WDT will recount. Hardware will automatically clear this bit.
看门狗清“0”位, 当设为“1”时, 看门狗将重新计数。硬件将自动清“0 ”此位。

IDLE_WDT : When set, WDT is enabled in IDLE mode. When clear, WDT is disabled in IDLE
看门狗“IDLE ”模式位， 当设置为“1”时， 看门狗定时器在“空闲模式”计数
当清“0”该位时， 看门狗定时器在“空闲模式”时不计数

PS2,PS1,PS0 : Pre-scale value of Watchdog timer is shown as the bellowed table:
看门狗定时器预分频值，如下表所示

PS2	PS1	PS0	Pre-scale 预分频	WDT overflow Time @20MHz
0	0	0	2	39.3 mS
0	0	1	4	78.6 mS
0	1	0	8	157.3 mS
0	1	1	16	314.6 mS
1	0	0	32	629.1 mS
1	0	1	64	1.25 S
1	1	0	128	2.5 S
1	1	1	256	5 S

The WDT period is determined by the following equation 看门狗溢出时间计算
看门狗溢出时间 = (12 x Pre-scale x 32768) / Oscillator frequency

设时钟为12MHz：

看门狗溢出时间 = (12 × Pre-scale × 32768) / 12000000 = Pre-scale× 393216 / 12000000

PS2	PS1	PS0	Pre-scale 预分频	WDT overflow Time @12MHz
0	0	0	2	65.5 mS
0	0	1	4	131.0 mS
0	1	0	8	262.1 mS
0	1	1	16	524.2 mS
1	0	0	32	1.0485 S
1	0	1	64	2.0971 S
1	1	0	128	4.1943 S
1	1	1	256	8.3886 S

设时钟为11.0592MHz：

看门狗溢出时间= (12 x Pre-scale x 32768) / 11059200 = Pre-scale x 393216 / 11059200

PS2	PS1	PS0	Pre-scale	WDT overflow Time @11.0592MHz
0	0	0	2	71.1 mS
0	0	1	4	142.2 mS
0	1	0	8	284.4 mS
0	1	1	16	568.8 mS
1	0	0	32	1.1377 S
1	0	1	64	2.2755 S
1	1	0	128	4.5511 S
1	1	1	256	9.1022 S

看门狗测试程序，在STC的下载板上可以直接测试

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机 看门狗及其溢出时间计算公式-------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/
;本演示程序在STC-ISP Ver 4.86.PCB的下载编程工具上测试通过,相关的工作状态在P1口上显示
;看门狗及其溢出时间 = (12 * Pre_scale *32768)/Oscillator frequency
WDT_CONTR               EQU     0E1H            ;看门狗地址
WDT_TIME_LED            EQU     P1.5            ;用 P1.5 控制看门狗溢出时间指示灯,
                                ;看门狗溢出时间可由该指示灯亮的时间长度或熄灭的时间长度表示
WDT_FLAG_LED            EQU     P1.7
                                ;用P1.7控制看门狗溢出复位指示灯, 如点亮表示为看门狗溢出复位
Last_WDT_Time_LED_Status    EQU   00H           ;位变量,存储看门狗溢出时间指示灯的上一次状态位
;WDT复位时间(所用的Oscillator frequency = 18.432MHz):
;Pre_scale_Word         EQU     00111100B       ;清0,启动看门狗, 预分频数=32,    0.68S
Pre_scale_Word          EQU     00111101B       ;清0,启动看门狗, 预分频数=64,    1.36S
;Pre_scale_Word         EQU     00111110B       ;清0,启动看门狗, 预分频数=128,   2.72S
;Pre_scale_Word         EQU     00111111B       ;清0,启动看门狗, 预分频数=256,   5.44S
        ORG     0000H
        AJMP    MAIN
        ORG     0100H
MAIN:
        MOV     A,      WDT_CONTR               ;检测是否为看门狗复位
        ANL     A,      #10000000B
        JNZ     WDT_Reset               ;WDT_CONTR.7 = 1, 看门狗复位, 跳转到看门狗复位程序
;WDT_CONTR.7 = 0,上电复位, 冷启动,RAM单元内容为随机值
        SETB    Last_WDT_Time_LED_Status        ;上电复位,
                                                ;初始化看门狗溢出时间指示灯的状态位 = 1
        CLR     WDT_TIME_LED                    ;上电复位, 点亮看门狗溢出时间指示灯
        MOV     WDT_CONTR,      #Pre_scale_Word     ;启动看门狗
```

```
WAIT1:
        SJMP    WAIT1                               ;循环执行本语句(停机)，等待看门狗溢出复位
;WDT_CONTR.7 = 1, 看门狗复位， 热启动，RAM 单元内容不变， 为复位前的值
WDT_Reset:                                          ;看门狗复位， 热启动
        CLR     WDT_FLAG_LED                        ;是看门狗复位,点亮看门狗溢出复位指示灯

        JB      Last_WDT_Time_LED_Status,           Power_Off_WDT_TIME_LED
                                                    ;为1熄灭相应的灯,为0亮相应灯
;根据看门狗溢出时间指示灯的上一次状态位设置 WDT_TIME_LED 灯,
;若上次亮本次就熄灭， 若上次熄灭本次就亮
        CLR     WDT_TIME_LED                        ;上次熄灭本次点亮看门狗溢出时间指示灯
        CPL     Last_WDT_Time_LED_Status            ;将看门狗溢出时间指示灯的上一次状态位取反
WAIT2:
        SJMP    WAIT2                               ;循环执行本语句(停机)，等待看门狗溢出复位
Power_Off_WDT_TIME_LED:
        SETB    WDT_TIME_LED                        ;上次亮本次就熄灭看门狗溢出时间指示灯
        CPL     Last_WDT_Time_LED_Status            ;将看门狗溢出时间指示灯的上一次状态位取反
WAIT3:
        SJMP    WAIT3                               ;循环执行本语句(停机)，等待看门狗溢出复位
        END
```

2.3.7 冷启动复位和热启动复位

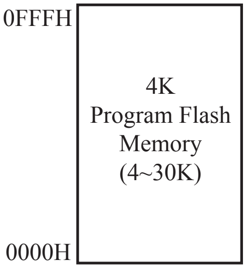
	复位源	现象
热启动复位	内部看门狗复位	会使单片机直接从用户程序区0000H处开始执行用户程序
	通过控制RESET脚产生的硬复位	会使系统 从用户程序区0000H处开始直接执行用户程序
	通过对ISP_CONTR寄存器送入20H产生的软复位	会使系统从用户程序区0000H处开始直接执行用户程序
	通过对ISP_CONTR寄存器送入60H产生的软复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用户程序区执行用户程序
冷启动复位	系统停电后再上电引起的硬复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用户程序区执行用户程序

第3章 片内存储器和特殊功能寄存器(SFRs)

STC12C5410AD系列单片机的程序存储器和数据存储器是各自独立编址的. STC-12C5410AD系列单片机的所有程序存储器都是片上Flash存储器，不能访问外部程序存储器，因为没有外部访问使能信号—$\overline{EA}$和程序存储启用信号—$\overline{PSEN}$。STC12C5410AD系列单片机内部有768字节的数据存储器，其在物理和逻辑上都分为两个地址空间:内部RAM(256字节)和内部扩展RAM(512字节)。同样，STC12C5410AD系列单片机也不可以访问在外部数据存储器，因为没有访问外部数据存储器的总线。

3.1 程序存储器

程序存储器用于存放用户程序、数据和表格等信息。STC12C5410AD系列单片机内部集成了1K~30K字节的Flash程序存储器。STC12C5410AD系列各种型号单片机的程序Flash存储器的地址如下表所示。



STC12C5404AD单片机程序存储器

Type	Program Memory
STC12C/LE5401AD	0000H~03FFH (1K)
STC12C/LE5402AD	0000H~07FFH (2K)
STC12C/LE5404AD	0000H~0FFFH (4K)
STC12C/LE5406AD	0000H~17FFH (6K)
STC12C/LE5408AD	0000H~1FFFH (8K)
STC12C/LE5410AD	0000H~27FFH (10K)
STC12C/LE5412AD	0000H~2FFFH (12K)

单片机复位后，程序计数器(PC)的内容为0000H，从0000H单元开始执行程序。另外中断服务程序的入口地址(又称中断向量)也位于程序存储器单元。在程序存储器中，每个中断都有一个固定的入口地址，当中断发生并得到响应后，单片机就会自动跳转到相应的中断入口地址去执行程序。外部中断0的中断服务程序的入口地址是0003H，定时器/计数器0中断服务程序的入口地址是000BH, 外部中断1的中断服务程序的入口地址是0013H，定时器/计数器1的中断服务程序的入口地址是001BH等。更多的中断服务程序的入口地址(中断向量)见单独的中断章节。由于相邻中断入口地址的间隔区间(8个字节)有限，一般情况下无法保存完整的中断服务程序，因此，一般在中断响应的地址区域存放一条无条件转移指令，指向真正存放中断服务程序的空间去执行。

程序Flash存储器可在线反复编程擦写10万次以上，提高了使用的灵活性和方便性。

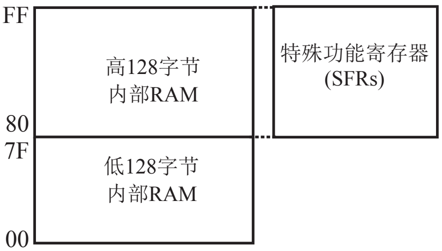
3.2 数据存储器(SRAM)

STC12C5410AD系列单片机内部集成了512字节RAM，可用于存放程序执行的中间结果和过程数据。内部数据存储器在物理和逻辑上都分为两个地址空间:内部RAM(256字节)和内部扩展RAM(256字节)。

3.2.1 内部RAM

内部RAM共256字节，可分为3个部分：**低128字节RAM(与传统8051兼容)、高128字节RAM(Intel在8052中扩展了高128字节RAM)**及特殊功能寄存器区。低128字节的数据存储器既可直接寻址也可间接寻址,通过“MOV”和“MOV @Ri”指令访问。高128字节RAM与特殊功能寄存器区貌似共用相同的地址范围，都使用80H~FFH,地址空间虽然貌似重叠，但物理上是独立的，使用时通过不同的寻址方式加以区分。高128字节RAM只能间接寻址，通过“MOV @Ri”指令访问。特殊功能寄存器区只可直接寻址，通过“MOV”指令访问。

内部RAM的结构如下图所示，地址范围是00H~FFH。



内部 RAM

低128字节的内部RAM

低128字节RAM也称通用RAM区。通用RAM区又可分为工作寄存器组区，可位寻址区，用户RAM区和堆栈区。工作寄存器组区地址从00H~1FH共32B(字节)单元，分为4组(每一组称为一个寄存器组)，每组包含8个8位的工作寄存器，编号均为R0 ~ R7，但属于不同的物理空间。通过使用工作寄存器组，可以提高运算速度。R0~R7是常用的寄存器，提供4组是因为1组往往不够用。程序状态字PSW寄存器中的RS1和RS0组合决定当前使用的工作寄存器组。见下面PSW寄存器的介绍。可位寻址区的地址从20H ~ 2FH共16个字节单元。20H~2FH单元既可向普通RAM单元一样按字节存取，也可以对单元中的任何一位单独存取，共128位，所对应的地址范围是00H~7FH。位地址范围是00H~7FH，内部RAM低128字节的地址也是00H~7FH；从外表看，二者地址是一样的，实际上二者具有本质的区别；位地址指向的是一个位，而字节地址指向的是一个字节单元，在程序中使用不同的指令区分。内部RAM中的30H~FFH单元是用户RAM和堆栈区。一个8位的堆栈指针(SP)，用于指向堆栈区。单片机复位后，堆栈指针SP为07H，指向了工作寄存器组0中的R7，因此，用户初始化程序都应对SP设置初值，一般设置在80H以后的单元为宜。

PSW：程序状态字寄存器(可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PSW	D0H	name	CY	AC	F0	RS1	RS0	OV	F1	P

CY ： 标志位。进行加法运算时，当最高位即B7位有进位，或执行减法运算最高位有借位时，CY为1；反之为0

AC ： 进位辅助位。进行加法运算时，当B3位有进位，或执行减法运算B3有借位时，AC为1；反之为0。设置辅助进位标志AC的目的是为了便于BCD码加法、减法运算的调整。

F0 ： 用户标志位0。

RS1、RS0: 工作寄存器组的选择位。如下表

RS1	RS0	当前使用的工作寄存器组(R0~R7)
0	0	0组(00H~07H)
0	1	1组(08H~0FH)
1	0	2组(10H~17H)
1	1	3组(18H~1FH)

OV : 溢出标志位.

B1 ： 保留位

F1 ： 用户标志位1。

P ： 奇偶标志位。该标志位始终体现累加器ACC中1的个数的奇偶性。如果累加器ACC中1的个数为奇数，则P置1；当累加器ACC中的个数为偶数(包括0个)时，P位为0

堆栈指针(SP):

堆栈指针是一个8位专用寄存器。它指示出堆栈顶部在内部RAM块中的位置。系统复位后，SP初始化位07H，使得堆栈事实上由08H单元开始，考虑08H~1FH单元分别属于工作寄存器组1~3，若在程序设计中用到这些区，则最好把SP值改变为80H或更大的值为宜。STC12C5410AD系列单片机的堆栈是向上生长的，即将数据压入堆栈后，SP内容增大。

;内部常规256字节RAM的测试程序(汇编)

```
;/* ---  STC International Limited ------------------------------------------- */
;/* ---  STC 姚永平  设计 2006/1/6   V1.0 ----------------------------- */
;/* ---  演示 STC12C5410AD/AD/PWM系列单片机 MCU 内部扩展RAM演示程序--------- */
;/* ---  本演示程序在STC-ISP Ver 3.0A.PCB的下载编程工具上测试通过 ---------- */
;/* ---  如果要在程序中使用该程序,请在程序中注明使用了STC的资料及程序 - */
;/* ---  如果要在文章中引用该程序,请在文章中注明使用了STC的资料及程序 - */

TEST_CONST      EQU     5AH
;TEST_RAM       EQU     03H
        ORG     0000H
        LJMP    INITIAL

        ORG     0050H
INITIAL:
        MOV     R0,     #253
        MOV     R1,     #3H
TEST_ALL_RAM:
        MOV     R2,     #0FFH
TEST_ONE_RAM:
        MOV     A,      R2
        MOV     @R1,    A
        CLR     A
        MOV     A,      @R1
        CJNE    A,      2H,     ERROR_DISPLAY
        DJNZ    R2,     TEST_ONE_RAM
        INC     R1
        DJNZ    R0,     TEST_ALL_RAM
OK_DISPLAY:
        MOV     P1,     #11111110B
Wait1:
        SJMP    Wait1
ERROR_DISPLAY:
        MOV     A,      R1
        MOV     P1,     A
Wait2:
        SJMP    Wait2

        END
```

3.2.2 内部扩展256字节RAM(物理上在内部，逻辑上在外部)

STC12C5410AD单片机片内除了集成256字节的内部RAM外，还集成了256字节的扩展RAM，地址范围是0000H~00FFH. 访问内部扩展RAM的方法和传统8051单片机访问外部扩展RAM的方法相同，但是不影响P0口、P2口、P3.6、P3.7和ALE。在汇编语言中，内部扩展RAM通过MOVX指令访问，即使用" MOVX @DPTR"或者"MOVX @Ri"指令访问。在C语言中，可使用xdata声明存储类型即可，如"unsigned char xdata i=0; "。

内部扩展256字节RAM的测试程序(物理上在内部，逻辑上在外部，C语言)

```
;/* ---  STC International Limited ------------------------------------------ */
;/* ---  STC 姚永平  设计 2006/1/6   V1.0 ------------------------------ */
;/* ---  演示 STC12C5410AD/AD/PWM系列单片机 MCU 内部扩展RAM演示程序--------- */
;/* ---  本演示程序在STC-ISP Ver 3.0A.PCB的下载编程工具上测试通过 ----------- */
;/* ---  如果要在程序中使用该程序,请在程序中注明使用了STC的资料及程序 --- */
;/* ---  如果要在文章中引用该程序,请在文章中注明使用了STC的资料及程序 --- */

#include <reg52.h>
#include <intrins.h>        /* use _nop_() function */

sfr AUXR = 0x8e;

sfr IPH       = 0xb7;
sfr WDT_CONTR = 0xe1;
sfr ISP_DATA  = 0xe2;
sfr ISP_ADDRH = 0xe3;
sfr ISP_ADDRL = 0xe4;
sfr ISP_CMD   = 0xe5;
sfr ISP_TRIG  = 0xe6;
sfr ISP_CONTR = 0xe7;

sbit ERROR_LED = P1^5;
sbit OK_LED = P1^7;

void main()
{
    unsigned int array_point = 0;

    /* 测试数组 Test_array_one[128],Test_array_two[128]*/
    unsigned char xdata Test_array_one[128]       =
    {
        0x00,     0x01,     0x02,     0x03,     0x04,     0x05,     0x06,     0x07,
        0x08,     0x09,     0x0a,     0x0b,     0x0c,     0x0d,     0x0e,     0x0f,
        0x10,     0x11,     0x12,     0x13,     0x14,     0x15,     0x16,     0x17,
        0x18,     0x19,     0x1a,     0x1b,     0x1c,     0x1d,     0x1e,     0x1f,
        0x20,     0x21,     0x22,     0x23,     0x24,     0x25,     0x26,     0x27,
        0x28,     0x29,     0x2a,     0x2b,     0x2c,     0x2d,     0x2e,     0x2f,
        0x30,     0x31,     0x32,     0x33,     0x34,     0x35,     0x36,     0x37,
        0x38,     0x39,     0x3a,     0x3b,     0x3c,     0x3d,     0x3e,     0x3f,
        0x40,     0x41,     0x42,     0x43,     0x44,     0x45,     0x46,     0x47,
        0x48,     0x49,     0x4a,     0x4b,     0x4c,     0x4d,     0x4e,     0x4f,
```

```
        0x50,       0x51,       0x52,       0x53,       0x54,       0x55,       0x56,       0x57,
        0x58,       0x59,       0x5a,       0x5b,       0x5c,       0x5d,       0x5e,       0x5f,
        0x60,       0x61,       0x62,       0x63,       0x64,       0x65,       0x66,       0x67,
        0x68,       0x69,       0x6a,       0x6b,       0x6c,       0x6d,       0x6e,       0x6f,
        0x70,       0x71,       0x72,       0x73,       0x74,       0x75,       0x76,       0x77,
        0x78,       0x79,       0x7a,       0x7b,       0x7c,       0x7d,       0x7e,       0x7f,
    };

 unsigned char xdata Test_array_two[128]        =
    {
        0x00,       0x01,       0x02,       0x03,       0x04,       0x05,       0x06,       0x07,
        0x08,       0x09,       0x0a,       0x0b,       0x0c,       0x0d,       0x0e,       0x0f,
        0x10,       0x11,       0x12,       0x13,       0x14,       0x15,       0x16,       0x17,
        0x18,       0x19,       0x1a,       0x1b,       0x1c,       0x1d,       0x1e,       0x1f,
        0x20,       0x21,       0x22,       0x23,       0x24,       0x25,       0x26,       0x27,
        0x28,       0x29,       0x2a,       0x2b,       0x2c,       0x2d,       0x2e,       0x2f,
        0x30,       0x31,       0x32,       0x33,       0x34,       0x35,       0x36,       0x37,
        0x38,       0x39,       0x3a,       0x3b,       0x3c,       0x3d,       0x3e,       0x3f,
        0x40,       0x41,       0x42,       0x43,       0x44,       0x45,       0x46,       0x47,
        0x48,       0x49,       0x4a,       0x4b,       0x4c,       0x4d,       0x4e,       0x4f,
        0x50,       0x51,       0x52,       0x53,       0x54,       0x55,       0x56,       0x57,
        0x58,       0x59,       0x5a,       0x5b,       0x5c,       0x5d,       0x5e,       0x5f,
        0x60,       0x61,       0x62,       0x63,       0x64,       0x65,       0x66,       0x67,
        0x68,       0x69,       0x6a,       0x6b,       0x6c,       0x6d,       0x6e,       0x6f,
        0x70,       0x71,       0x72,       0x73,       0x74,       0x75,       0x76,       0x77,
        0x78,       0x79,       0x7a,       0x7b,       0x7c,       0x7d,       0x7e,       0x7f,
    };
    ERROR_LED = 1;
    OK_LED = 1;
    for(array_point=0; array_point<128; array_point++)
    {
        if(Test_array_one[array_point]!=Test_array_two [array_point])
        {
                ERROR_LED = 0;
                OK_LED = 1;
                break;
         }
         else
         {
                OK_LED = 0;
                ERROR_LED = 1;
          }
        }
        while(1);
}
```

3.3 特殊功能寄存器(SFRs)

特殊功能寄存器(SFR)是用来对片内各功能模块进行管理、控制、监视的控制寄存器和状态寄存器，是一个特殊功能的RAM区。STC12C5410AD系列单片机内的特殊功能寄存器(SFR)与内部高128字节RAM貌似共用相同的地址范围，都使用80H~FFH, 但特殊功能寄存器(SFR)必须用直接寻址指令访问。

STC12C5410AD系列单片机的特殊功能寄存器名称及地址映象如下表所示

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	
0F8H		CH 0000,0000	CCAP0H 0000,0000	CCAP1H 0000,0000	CCAP2H 0000,0000	CCAP3H 0000,0000			0FFH
0F0H	B 0000,0000		PCA_PWM0 xxxx,xx00	PCA_PWM1 xxxx,xx00	PCA_PWM2 xxxx,xx00	PCA_PWM3 xxxx,xx00			0F7H
0E8H		CL 0000,0000	CCAP0L 0000,0000	CCAP1L 0000,0000	CCAP2L 0000,0000	CCAP3L 0000,0000			0EFH
0E0H	ACC 0000,0000	WDT_CONR 0x00,0000	ISP_DATA 1111,1111	ISP_ADDRH 0000,0000	ISP_ADDRL 0000,0000	ISP_CMD xxxx,xx00	ISP_TRIG xxxx,xxxx	ISP_CONTR 0000,1000	0E7H
0D8H	CCON 00xx,0000	CMOD 0xxx,x000	CCAPM0 x000,0000	CCAPM1 x000,0000	CCAPM2 x000,0000	CCAPM3 x000,0000			0DFH
0D0H	PSW 0000,0000								0D7H
0C8H									0CFH
0C0H						ADC_CONTR 0000,0000	ADC_DATA 0000,0000	CLK_DIV xxxx,x000	0C7H
0B8H	IP x000,0000	SADEN					ADC_LOW2 0000,0000		0BFH
0B0H	P3 1x11,1111	P3M0 0000,0000	P3M1 0000,0000					IPH x000,0000	0B7H
0A8H	IE 0000,0000	SADDR							0AFH
0A0H	P2 1111,1111							Don't use	0A7H
098H	SCON 0000,0000	SBUF xxxx,xxxx							09FH
090H	P1 1111,1111	P1M0 0000,0000	P1M1 0000,0000	P0M0 0000,0000	P0M1 0000,0000	P2M0 0000,0000	P2M1 0000,0000		097H
088H	TCON 0000,0000	TMOD 0000,0000	TL0 0000,0000	TL1 0000,0000	TH0 0000,0000	TH1 0000,0000	AUXR 0000,00xx	WAKE_CLKO 0000,xx00	08FH
080H	P0 1111,1111	SP 0000,0111	DPL 0000,0000	DPH 0000,0000	SPSTAT 00xx,xxxx	SPCTL 0000,0100	SPDAT 0000,0000	PCON 0011,0000	087H
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

可位寻址

不可位寻址

注意：寄存器地址能够被8整除的才可以进行位操作，不能够被8整除的不可以进行位操作

符号	描述	地址	位地址及符号 MSB							LSB	复位值
P0	Port 0	80H	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	1111 1111B
SP	堆栈指针	81H									0000 0111B
DPTR DPL	数据指针(低)	82H									0000 0000B
DPTR DPH	数据指针(高)	83H									0000 0000B
SPSTAT	SPI状态寄存器	84H	SPIF	WCOL	-	-	-	-	-	-	00xx xxxxB
SPCTL	SPI控制寄存器	85H	SSIG	SPEN	DORD	MSTR	CPOL	CAPHA	SPR1	SPR0	0000 0100B
SPDAT	SPI数据寄存器	86H									0000 0000B
PCON	电源控制寄存器	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
TCON	定时器控制寄存器	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	定时器工作方式寄存器	89H	GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0	0000 0000B
TL0	定时器0低8位寄存器	8AH									0000 0000B
TL1	定时器1低8位寄存器	8BH									0000 0000B
TH0	定时器0高8位寄存器	8CH									0000 0000B
TH1	定时器1高8位寄存器	8DH									0000 0000B
AUXR	辅助寄存器	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	掉电唤醒和时钟输出寄存器	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B
P1	Port 1	90H	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	1111 1111B
P1M0	P1口模式配置寄存器0	91H									0000 0000B
P1M1	P1口模式配置寄存器1	92H									0000 0000B
P0M0	P0口模式配置寄存器0	93H									0000 0000B
P0M1	P0口模式配置寄存器1	94H									0000 0000B
P2M0	P2口模式配置寄存器0	95H									0000 0000B
P2M1	P2口模式配置寄存器1	96H									0000 0000B
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数据缓冲器	99H									xxxx,xxxx
P2	Port 2	A0H	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	1111 1111B

符号	描述	地址	位地址及符号 MSB							LSB	复位值
IE	中断允许寄存器	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
SADDR	从机地址控制寄存器	A9H									0000 0000B
P3	Port 3	B0H	P3.7	-	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	1x11 1111B
P3M0	P3口模式配置寄存器0	B1H									0000 0000B
P3M1	P3口模式配置寄存器1	B2H									0000 0000B
IPH	中断优先级高字节寄存器	B7H	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
IP	中断优先级寄存器	B8H	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000 0000B
SADEN	从机地址掩模寄存器	B9H									0000 0000B
ADC_CONTR	A/D转换控制寄存器	C5H	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0	0000 0000B
ADC_DATA	A/D转换结果寄存器高	C6H									0000 0000B
ADC_LOW2	A/D转换结果寄存器低	BEH									0000 0000B
CLK_DIV	时钟分频寄存器	C7H	-	-	-	-	-	CLKS2	CLKS1	CLKS0	xxxx x000B
PSW	程序状态字寄存器	D0H	CY	AC	F0	RS1	RS0	OV	F1	P	0000 0000B
CCON	PCA控制寄存器	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0	00xx 0000B
CMOD	PCA模式寄存器	D9H	CIDL	-	-	-	-	CPS1	CPS0	ECF	0xxx x000B
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000 0000B
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000 0000B
CCAPM2	PCA Module 2 Mode Register	DCH	-	ECOM2	CAPP2	CAPN2	MAT2	TOG2	PWM2	ECCF2	x000 0000B
CCAPM3	PCA Module 3 Mode Register	DDH	-	ECOM3	CAPP3	CAPN3	MAT3	TOG3	PWM3	ECCF3	x000 0000B
ACC	累加器	E0H									0000 0000B
WDT_CONTR	看门狗控制寄存器	E1H	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	0x00 0000B
ISP_DATA	ISP/IAP 数据寄存器	E2H									1111 1111B
ISP_ADDRH	ISP/IAP 高8位地址寄存器	E3H									0000 0000B
ISP_ADDRL	ISP/IAP 低8位地址寄存器	E4H									0000 0000B
ISP_CMD	ISP/IAP 命令寄存器	E5H	-	-	-	-	-	-	MS1	MS0	xxxx xx00B
ISP_TRIG	ISP/IAP 命令触发寄存器	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP控制寄存器	E7H	ISPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 x000B

符号	描述	地址	位地址及符号 MSB							LSB	复位值
CL	PCA Base Timer Low	E9H									0000 0000B
CCAP0L	PCA Module-0 Capture Register Low	EAH									0000 0000B
CCAP1L	PCA Module-1 Capture Register Low	EBH									0000 0000B
CCAP2L	PCA Module-2 Capture Register Low	ECH									0000 0000B
CCAP3L	PCA Module-3 Capture Register Low	EDH									0000 0000B
B	B寄存器	F0H									0000 0000B
PCA_PWM0	PCA PWM Mode Auxiliary Register 0	F2H	-	-	-	-	-	-	EPC0H	EPC0L	xxxx xx00B
PCA_PWM1	PCA PWM Mode Auxiliary Register 1	F3H	-	-	-	-	-	-	EPC1H	EPC1L	xxxx xx00B
PCA_PWM2	PCA PWM Mode Auxiliary Register 2	F4H	-	-	-	-	-	-	EPC2H	EPC2L	xxxx xx00B
PCA_PWM3	PCA PWM Mode Auxiliary Register 3	F5H	-	-	-	-	-	-	EPC3H	EPC3L	xxxx xx00B
CH	PCA Base Timer High	F9H									0000 0000B
CCAP0H	PCA Module-0 Capture Register High	FAH									0000 0000B
CCAP1H	PCA Module-1 Capture Register High	FBH									0000 0000B
CCAP2H	PCA Module-2 Capture Register High	FCH									0000 0000B
CCAP3H	PCA Module-3 Capture Register High	FDH									0000 0000B

下面简单的介绍一下普通8051单片机常用的一些寄存器：

1. 程序计数器(PC)

程序计数器PC在物理上是独立的，不属于SFR之列。PC字长16位，是专门用来控制指令执行顺序的寄存器。单片机上电或复位后，PC=0000H，强制单片机从程序的零单元开始执行程序。

2. 累加器(ACC)

累加器ACC是8051单片机内部最常用的寄存器，也可写作A。常用于存放参加算术或逻辑运算的操作数及运算结果。

3. B寄存器

B寄存器在乘法和除法运算中须与累加器A配合使用。MUL AB指令把累加器A和寄存器B中的8位无符号数相乘，所得的16位乘积的低字节存放在A中，高字节存放在B中。DIV AB指令用B除以A，整数商存放在A中，余数存放在B中。寄存器B还可以用作通用暂存寄存器。

4. 程序状态字(PSW)寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PSW	D0H	name	CY	AC	F0	RS1	RS0	OV	F1	P

CY ： 标志位。进行加法运算时，当最高位即B7位有进位，或执行减法运算最高位有借位时，CY为1；反之为0

AC ： 进位辅助位。进行加法运算时，当B3位有进位，或执行减法运算B3有借位时，AC为1；反之为0。设置辅助进位标志AC的目的是为了便于BCD码加法、减法运算的调整。

F0 ： 用户标志位0。

RS1、RS0: 工作寄存器组的选择位。RS1、RS0: 工作寄存器组的选择位。如下表

RS1	RS0	当前使用的工作寄存器组(R0~R7)
0	0	0组(00H~07H)
0	1	1组(08H~0FH)
1	0	2组(10H~17H)
1	1	3组(18H~1FH)

OV： 溢出标志位.

F0 ： 用户标志位1。

B1 ： 保留位

P ： 奇偶标志位。该标志位始终体现累加器ACC中1的个数的奇偶性。如果累加器ACC中1的个数为奇数，则P置1；当累加器ACC中的个数为偶数(包括0个)时，P位为0

5. 堆栈指针(SP)

堆栈指针是一个8位专用寄存器。它指示出堆栈顶部在内部RAM块中的位置。系统复位后，SP初始化位07H，使得堆栈事实上由08H单元开始，考虑08H~1FH单元分别属于工作寄存器组1~3，若在程序设计中用到这些区，则最好把SP值改变为80H或更大的值为宜。STC12C5410AD系列单片机的堆栈是向上生长的，即将数据压入堆栈后，SP内容增大。

6. 数据指针(DPTR)

数据指针(DPTR)是一个16位专用寄存器，由DPL(低8位)和DPH(高8位)组成,地址是82H(DPL,低字节)和83H(DPH,高字节)。DPTR是传统8051机中唯一可以直接进行16位操作的寄存器也可分别对DPL河DPH按字节进行操作。

第4章 STC12C5410AD系列单片机的I/O口结构

4.1 I/O口各种不同的工作模式及配置介绍

I/O口配置

STC12C5410AD系列单片机所有I/O口均可由软件配置成4种工作类型之一，如下表所示。4种类型分别为：准双向口/弱上拉(标准8051输出模式)、强推挽输出/强上拉、仅为输入(高阻)或开漏输出功能。每个口由2个控制寄存器中的相应位控制每个引脚工作类型。STC-12C5410AD系列单片机上电复位后为准双向口/弱上拉（传统8051的I/O口）模式。2V以上时为高电平，0.8V以下时为低电平。每个I/O口驱动能力均可达到20mA，但整个芯片最大不得超过55mA。

P3口设定 <P3.7, x, P3.5, P3.4, P3.3, P3.2, P3.1, P3.0,无P3.6口>(P3口地址：B0H)

P3M0[7 : 0]	P3M1 [7 : 0]	I/O 口模式
0	0	准双向口(传统8051 I/O 口模式）， 灌电流可达20mA , 拉电流为230μA , 由于制造误差， 实际为250uA～ 150uA
0	1	强推挽输出（ 强上拉输出， 可达20mA, 要加限流电阻）
1	0	仅为输入（ 高阻）
1	1	开漏(Open Drain)， 内部上拉电阻断开， 要外加

举例: MOV P3M0, #10100000B
MOV P3M1, #10010000B
;P3.7为开漏, P3.5为高阻输入, P3.4为强推挽输出, P3.3/P3.2/P3.1/P3.0为准双向口/弱上拉

P2口设定 <P2.7, P2.6, P2.5, P2.4, P2.3, P2.2, P2.1, P2.0>(P2口地址：A0H)

P2M0 [7 : 0]	P2M1 [7 : 0]	I/O 口模式
0	0	准双向口(传统8051 I/O 口模式）， 灌电流可达20mA , 拉电流为230μA , 由于制造误差， 实际为250uA～ 150uA
0	1	强推挽输出（ 强上拉输出， 可达20mA, 要加限流电阻）
1	0	仅为输入（ 高阻）
1	1	开漏(Open Drain)， 内部上拉电阻断开， 要外加

举例: MOV P2M0, #10100000B
MOV P2M1, #11000000B
;P2.7为开漏,P2.6为强推挽输出,P2.5为高阻输入,P2.4/P2.3/P2.2/P2.1/P2.0为准双向口/弱上拉

P1口设定 <P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0口>(P1口地址：90H)

P1M0 [7 : 0]	P1M1 [7 : 0]	I/O 口模式（P1.x 如做A/D使用，需先将其设置成开漏或高阻输入）
0	0	准双向口(传统8051 I/O 口模式）， 灌电流可达20mA , 拉电流为230μA , 由于制造误差， 实际为250uA～ 150uA
0	1	推挽输出（ 强上拉输出， 可达20mA, 要加限流电阻）
1	0	仅为输入（ 高阻），如果该I/O口需作为A/D使用，可选此模式
1	1	开漏(Open Drain)，如果该I/O口需作为A/D使用，可选此模式

```
举例:  MOV    P1M0, #10100000B
       MOV    P1M1, #11000000B
```

;P1.7为开漏,P1.6为强推挽输出,P1.5为高阻输入,P1.4/P1.3/P1.2/P1.1/P1.0为准双向口/弱上拉

P0口设定 <x, x, x, x, P0.3, P0.2, P0.1, P0.0口, 无P0.7, P0.6, P0.5, P0.4口>(P0口地址：80H)

P0M0 [1 : 0]	P0M1 [1 : 0]	I/O 口模式
0	0	准双向口(传统8051 I/O 口模式）， 灌电流可达20mA , 拉电流为230μA , 由于制造误差， 实际为250uA～ 150uA
0	1	推挽输出（ 强上拉输出， 可达20mA, 要加限流电阻）
1	0	仅为输入（ 高阻）
1	1	开漏(Open Drain)， 内部上拉电阻断开， 要外加

```
举例:  MOV    P0M0, #00001010B
       MOV    P0M1, #00001100B
```

;P0.3为开漏, P0.2为强推挽输出, P0.1为高阻输入, P0.0为准双向口/弱上拉

注意:

虽然每个I/O口在弱上拉时都能承受20mA的灌电流(还是要加限流电阻,如1K，560Ω等),在强推挽输出时都能输出20mA的拉电流（也要加限流电阻）,但整个芯片的工作电流推荐不要超过55mA。即从MCU-VCC流入的电流不超过55mA,从MCU-Gnd流出电流不超过55mA,整体流入/流出电流都不能超过55mA.

下面将与I/O口相关的寄存器及其地址列于此处，以方便用户查询

P3 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3	B0H	name	P3.7	-	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0

P3M0 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M0	B1H	name	P3M0.7	-	P3M0.5	P3M0.4	P3M0.3	P3M0.2	P3M0.1	P3M0.0

P3M1 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M1	B2H	name	P3M1.7	-	P3M1.5	P3M1.4	P3M1.3	P3M1.2	P3M1.1	P3M1.0

P2 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P2	A0H	name	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0

P2M0 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P2M0	95H	name	P2M0.7	P2M0.6	P2M0.5	P2M0.4	P2M0.3	P2M0.2	P2M0.1	P2M0.0

P2M1 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P2M1	96H	name	P2M1.7	P2M1.6	P2M1.5	P2M1.4	P2M1.3	P2M1.2	P2M1.1	P2M1.0

P1 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1	90H	name	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0

P1M0 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1M0	91H	name	P1M0.7	P1M0.6	P1M0.5	P1M0.4	P1M0.3	P1M0.2	P1M0.1	P1M0.0

P1M1 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1M1	92H	name	P1M1.7	P1M1.6	P1M1.5	P1M1.4	P1M1.3	P1M1.2	P1M1.1	P1M1.0

P0 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P0	80H	name	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0

P0M0 register(不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P0M0	93H	name	P0M0.7	P0M0.6	P0M0.5	P0M0.4	P0M0.3	P0M0.2	P0M0.1	P0M0.0

P0M1 register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P0M1	94H	name	P0M1.7	P0M1.6	P0M1.5	P0M1.4	P0M1.3	P0M1.2	P0M1.1	P0M1.0

4.2 I/O口各种不同的工作模式结构框图

4.2.1 准双向口输出配置

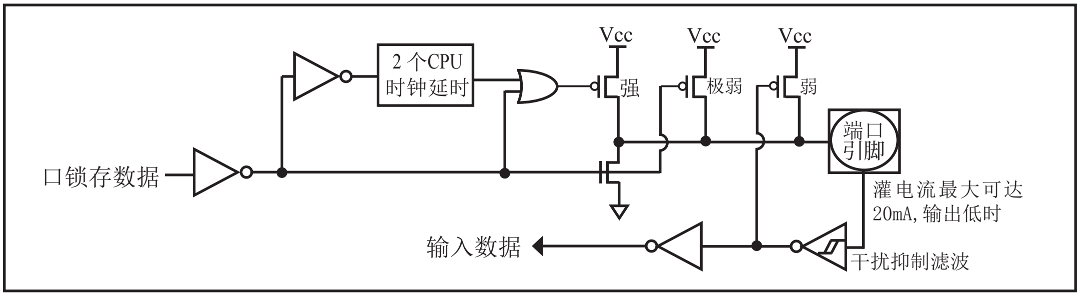
准双向口输出类型可用作输出和输入功能而不需重新配置口线输出状态。这是因为当口线输出为1时驱动能力很弱，允许外部装置将其拉低。当引脚输出为低时，它的驱动能力很强，可吸收相当大的电流。准双向口有3个上拉晶体管适应不同的需要。

在3个上拉晶体管中，有1个上拉晶体管称为“弱上拉”，当口线寄存器为1且引脚本身也为1时打开。此上拉提供基本驱动电流使准双向口输出为1。如果一个引脚输出为1而由外部装置下拉到低时，弱上拉关闭而“极弱上拉”维持开状态，为了把这个引脚强拉为低，外部装置必须有足够的灌电流能力使引脚上的电压降到门槛电压以下。

第2个上拉晶体管，称为“极弱上拉”，当口线锁存为1时打开。当引脚悬空时，这个极弱的上拉源产生很弱的上拉电流将引脚上拉为高电平。

第3个上拉晶体管称为“强上拉”。当口线锁存器由0到1跳变时，这个上拉用来加快准双向口由逻辑0到逻辑1转换。当发生这种情况时，强上拉打开约2个时钟以使引脚能够迅速地上拉到高电平。

准双向口输出如下图所示。



准双口输出

STC12LE5410AD系列单片机为3V器件，如果用户在引脚加上5V电压，将会有电流从引脚流向Vcc，这样导致额外的功率消耗。因此，建议不要在准双向口模式中向3V单片机引脚施加5V电压,如使用的话，要加限流电阻，或用二极管做输入隔离，或用三极管做输出隔离。

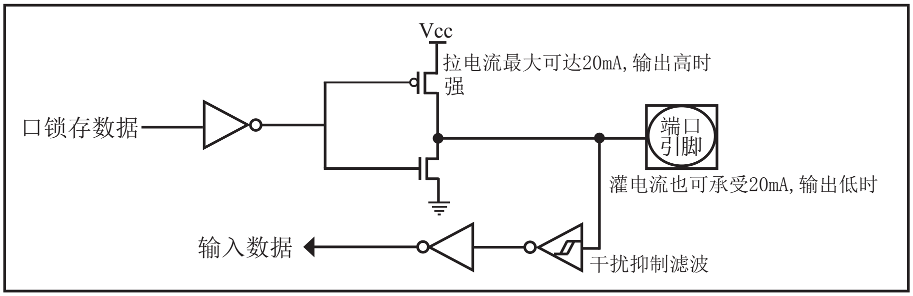
准双向口带有一个施密特触发输入以及一个干扰抑制电路。

准双向口读外部状态前,要先锁存为‘1’,才可读到外部正确的状态.

4.2.2 强推挽输出配置

强推挽输出配置的下拉结构与开漏输出以及准双向口的下拉结构相同，但当锁存器为1时提供持续的强上拉。推挽模式一般用于需要更大驱动电流的情况。

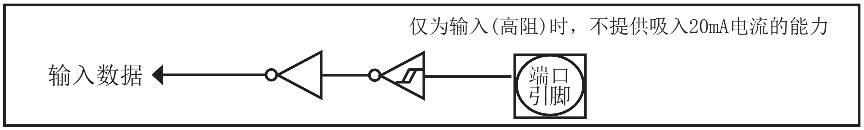
强推挽引脚配置如下图所示。



强推挽输出

4.2.3 仅为输入（高阻）配置

输入口配置如下图所示。



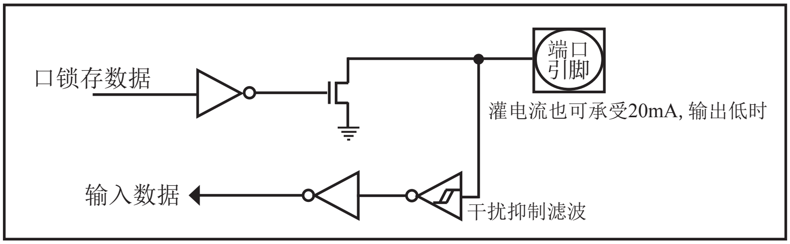
仅为输入(高阻)模式

输入口带有一个施密特触发输入以及一个干扰抑制电路。

4.2.4 开漏输出配置(若外加上拉电阻，也可读)

当口线锁存器为0时，开漏输出关闭所有上拉晶体管。当作为一个逻辑输出时，这种配置方式必须有外部上拉，一般通过电阻外接到Vcc。如果外部有上拉电阻，开漏的I/O口还可读外部状态，即此时被配置为开漏模式的I/O口还可作为输入I/O口。这种方式的下拉与准双向口相同。输出口线配置如下图所示。

开漏端口带有一个施密特触发输入以及一个干扰抑制电路。



开漏输出(如外部有上拉电阻，也可读)

关于I/O口应用注意事项：

少数用户反映I/O口有损坏现象,后发现是

有些是I/O口由低变高读外部状态时,读不对,实际没有损坏,软件处理一下即可。

因为1T的8051单片机速度太快了,软件执行由低变高指令后立即读外部状态,此时由于实际输出还没有变高,就有可能读不对,正确的方法是在软件设置由低变高后加1到2个空操作指令延时,再读就对了.

有些实际没有损坏,加上拉电阻就OK了

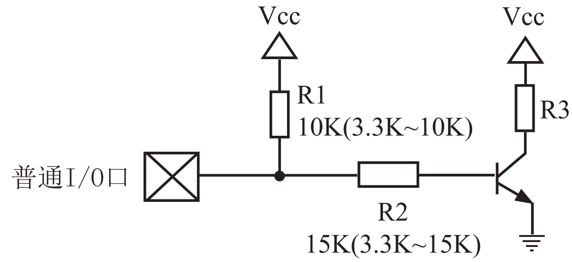
有些是外围接的是NPN三极管,没有加上拉电阻,其实基极串多大电阻,I/O口就应该上拉多大的电阻,或者将该I/O口设置为强推挽输出.

有些确实是损坏了,原因:

发现有些是驱动LED发光二极管没有加限流电阻,建议加1K以上的限流电阻,至少也要加470欧姆以上

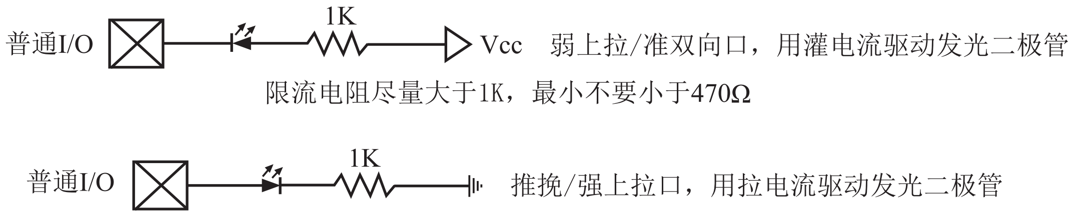
发现有些是做行列矩阵按键扫描电路时,实际工作时没有加限流电阻,实际工作时可能出现2个I/O口均输出为低,并且在按键按下时,短接在一起,我们知道一个CMOS电路的2个输出脚不应该直接短接在一起,按键扫描电路中,此时一个口为了读另外一个口的状态,必须先置高才能读另外一个口的状态,而8051单片机的弱上拉口在由0变为1时,会有2个时钟的强推挽高输出电流,输出到另外一个输出为低的I/O口,就有可能造成I/O口损坏.建议在两侧各加1K限流电阻,或者在软件处理上,不要出现按键两端的I/O口同时为低.

4.3 一种典型三极管控制电路



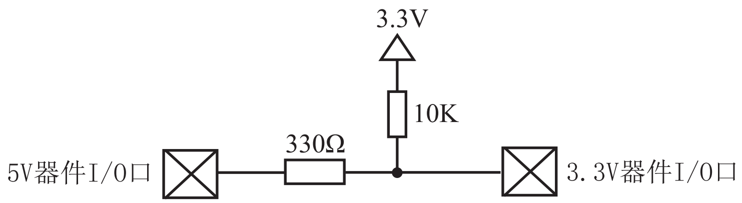
如果用弱上拉控制，建议加上拉电阻R1(3.3K～10K)，如果不加上拉电阻R1(3.3K～10K)，建议R2的值在15K以上，或用强推挽输出。

4.4 典型发光二极管控制电路

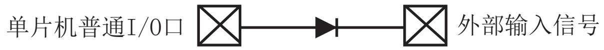


4.5 混合电压供电系统3V/5V器件I/O口互连

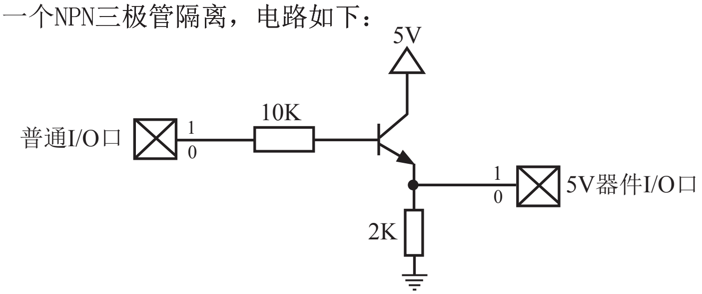
STC12C5410AD系列5V单片机连接3.3V器件时，为防止3.3V器件承受不了5V，可将相应的5V单片机I/O口先串一个330Ω的限流电阻到3.3V器件I/O口，程序初始化时将5V器件的I/O口设置成开漏配置，断开内部上拉电阻，相应的3.3V器件I/O口外部加10K上拉电阻到3.3V器件的Vcc，这样高电平是3.3V，低电平是0V，输入输出一切正常。



STC12LE5410AD系列3V单片机连接5V器件时，为防止3V器件承受不了5V，如果相应的I/O口是输入，可在该I/O口上串接一个隔离二极管，隔离高压部分。外部信号电压高于单片机工作电压时截止，I/O口因内部上拉到高电平，所以读I/O口状态是高电平；外部信号电压为低时导通，I/O口被钳位在0.7V，小于0.8V时单片机读I/O口状态是低电平。



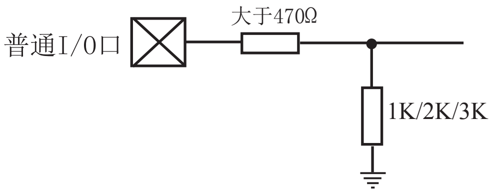
STC12LE5410AD系列3V单片机连接5V器件时，为防止3V器件承受不了5V，如果相应的I/O口是输出，可用一个NPN三极管隔离，电路如下：



4.6 如何让I/O口上电复位时为低电平

普通8051单片机上电复位时普通I/O口为弱上拉高电平输出,而很多实际应用要求上电时某些I/O口为低电平输出,否则所控制的系统(如马达)就会误动作,现STC12系列单片机由于既有弱上拉输出又有强推挽输出,就可以很轻松的解决此问题。

现可在STC12系列单片机I/O口上加一个下拉电阻(1K/2K/3K),这样上电复位时，虽然单片机内部I/O口是弱上拉/高电平输出，但由于内部上拉能力有限，而外部下拉电阻又较小,无法将其拉高,所以该I/O口上电复位时外部为低电平。如果要将此I/O口驱动为高电平，可将此I/O口设置为强推挽输出，而强推挽输出时，I/O口驱动电流可达20mA,故肯定可以将该口驱动为高电平输出。

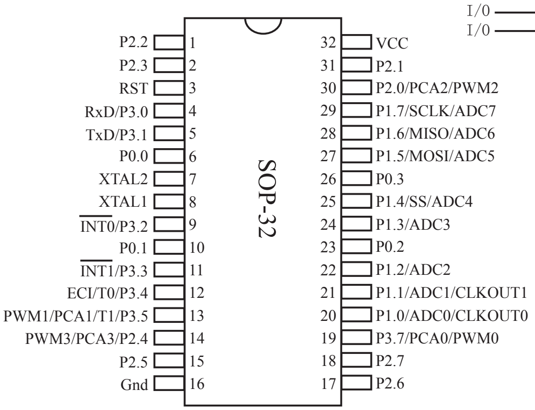


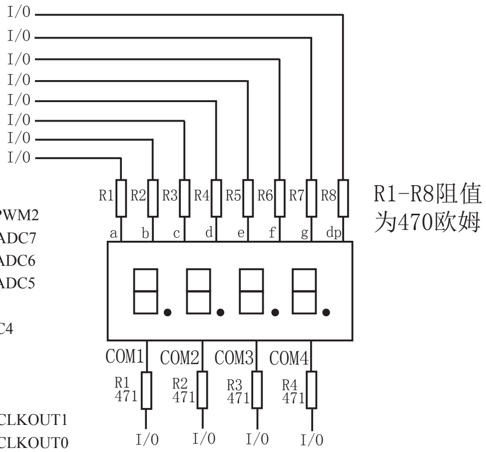
4.7 PWM输出时I/O口的状态

当某个I/O口作为PWM输出用时,该口的状态:

PWM 之前口的状态	PWM时口的状态
弱上拉/准双向口	强推挽输出/强上拉输出 要加输出限流电阻10K～1K
强推挽输出	强推挽输出/强上拉输出 要加输出限流电阻10K～1K
仅为输入/高阻	PWM无效
开漏	开漏

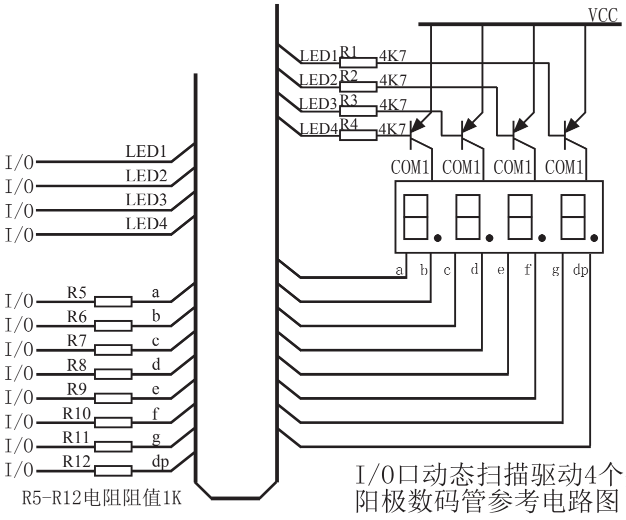
4.9 I/O口直接驱动LED数码管应用线路图





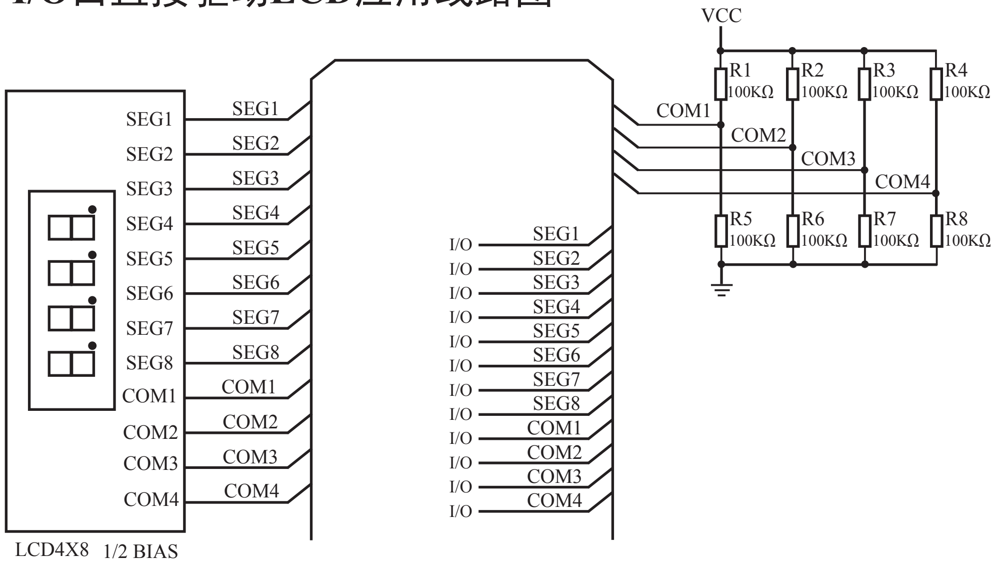
I/O口动态扫描驱动4个共阴极数码管参考电路图

I/O口动态扫描驱动数码管时，可以一次点亮一个数码管中的8段，但为降低功耗，建议可以一次只点亮其中的4段或者2段



I/O口动态扫描驱动4个共阳极数码管参考电路图

4.10 I/O口直接驱动LCD应用线路图



如何点亮相应的LCD像素:

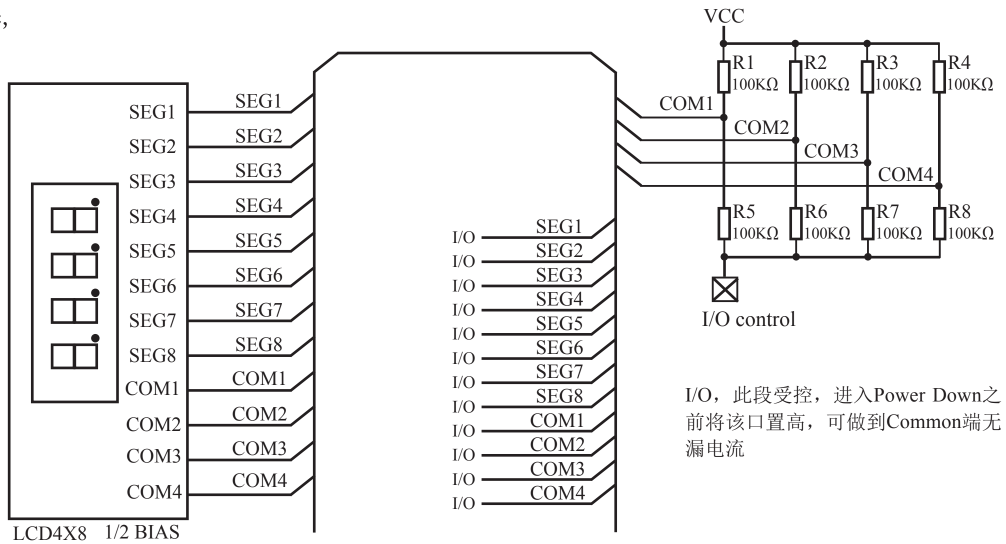
当相应的Common端和相应的Segment端压差大于1/2Vcc时,相应的像素就显示,当压差小于1/2Vcc时,相应的像素就不显示

I/O口如何控制Segment :

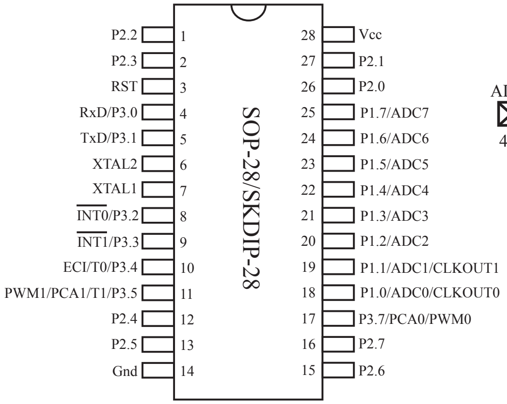
I/O口直接控制Segment,程序控制相应的口输出高或低时,对应的Segment就是Vcc或0V

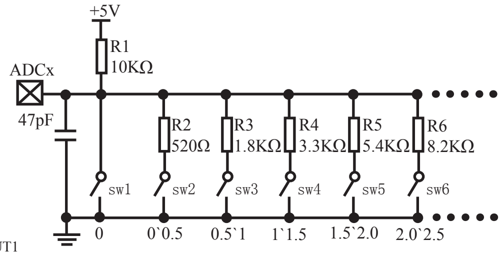
I/O口如何控制Common:

I/O口和2个100K的分压电阻组成Common,当I/O口输出为0时,相应的Common端为0V,当I/O口强推挽输出为1时,相应的Common端为Vcc,当I/O口为高阻输入时,相应的Common端为1/2Vcc,



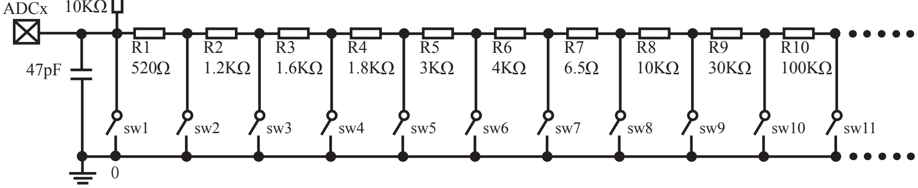
4.11 A/D做按键扫描应用线路图





此电路可以实现单个按键扫描和组合按键检测功能，但是具体电阻值应根据实际需要进行选择

本电路图采用10个按键等间隔分压，每个按键正负误差余量允许在±0.25V范围内变化，可以有效避免因为电阻误差或温度漂移等造成的按键检测失效，如果要求按键检测更加稳定可靠，可以减少按键数量放宽各个按键检测电压允许误差量。



第5章 指令系统

5.1 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

5.1.1 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

5.1.2 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。

5.1.3 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。

5.1.4 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

5.1.5 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中的当前值为基地址，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。

5.1.6 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。

5.1.7 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　；片内位单元位操作型指令

5.2 指令系统分类总结

----与普通8051指令代码完全兼容，但执行的时间效率大幅提升
----其中INC DPTR指令的执行速度大幅提升24倍
----共有12条指令，一个时钟就可以执行完成，平均速度快8~12倍

如果按功能分类，STC12C5410AD系列单片机指令系统可分为：
1. 数据传送类指令；
2. 算术操作类指令；
3. 逻辑操作类指令；
4. 控制转移类指令；
5. 布尔变量操作类指令。
按功能分类的指令系统表如下表所示。

传统 12T的8051
指令执行所需时钟

STC12C5410AD系列
指令执行所需时钟

算术操作类指令

助记符		功能说明	字节数	12时钟/机器周期所需时钟	1时钟/机器周期所需时钟	效率提升
ADD	A，Rn	寄存器内容加到累加器	1	12	2	6倍
ADD	A，direct	直接地址单元中的数据加到累加器	2	12	3	4倍
ADD	A，@Ri	间接RAM中的数据加到累加器	1	12	3	4倍
ADD	A，#data	立即数加到累加器	2	12	2	6倍
ADDC	A，Rn	寄存器带进位加到累加器	1	12	2	6倍
ADDC	A，direct	直接地址单元的内容带进位加到累加器	2	12	3	4倍
ADDC	A，@Ri	间接RAM内容带进位加到累加器	1	12	3	4倍
ADDC	A，#data	立即数带进位加到累加器	2	12	2	6倍
SUBB	A，Rn	累加器带借位减寄存器内容	1	12	2	6倍
SUBB	A，direct	累加器带借位减直接地址单元的内容	2	12	3	4倍
SUBB	A，@Ri	累加器带借位减间接RAM中的内容	1	12	3	4倍
SUBB	A，#data	累加器带借位减立即数	2	12	2	6倍
INC	A	累加器加1	1	12	2	6倍
INC	Rn	寄存器加1	1	12	3	4倍
INC	direct	直接地址单元加1	2	12	4	3倍
INC	@Ri	间接RAM单元加1	1	12	4	3倍
DEC	A	累加器减1	1	12	2	6倍
DEC	Rn	寄存器减1	1	12	3	4倍
DEC	direct	直接地址单元减1	2	12	4	3倍
DEC	@Ri	间接RAM单元减1	1	12	4	3倍
INC	DPTR	地址寄存器DPTR加1	1	24	1	24倍
MUL	AB	A乘以B	1	48	4	12倍
DIV	AB	A除以B	1	48	5	9.6倍
DA	A	累加器十进制调整	1	12	4	3倍

逻辑操作类指令

助记符	功能说明	字节数	12时钟/机器周期所需时钟	1时钟/机器周期所需时钟	效率提升
ANL A，Rn	累加器与寄存器相“与”	1	12	2	6倍
ANL A，direct	累加器与直接地址单元相“与”	2	12	3	4倍
ANL A，@Ri	累加器与间接RAM单元相“与”	1	12	3	4倍
ANL A，#data	累加器与立即数相“与”	2	12	2	6倍
ANL direct， A	直接地址单元与累加器相“与”	2	12	4	3倍
ANL direct，#data	直接地址单元与立即数相“与”	3	24	4	6倍
ORL A， Rn	累加器与寄存器相“或”	1	12	2	6倍
ORL A，direct	累加器与直接地址单元相“或”	2	12	3	4倍
ORL A, @Ri	累加器与间接RAM单元相“或”	1	12	3	4倍
ORL A，# data	累加器与立即数相“或”	2	12	2	6倍
ORL direct， A	直接地址单元与累加器相“或”	2	12	4	3倍
ORL direct， #data	直接地址单元与立即数相“或”	3	24	4	6倍
XRL A， Rn	累加器与寄存器相“异或”	1	12	2	6倍
XRL A， direct	累加器与直接地址单元相“异或”	2	12	3	4倍
XRL A， @Ri	累加器与间接RAM单元相“异或”	1	12	3	4倍
XRL A， # data	累加器与立即数相“异或”	2	12	2	6倍
XRL direct， A	直接地址单元与累加器相“异或”	2	12	4	3倍
XRL direct，#data	直接地址单元与立即数相“异或”	3	24	4	6倍
CLR A	累加器清“ 0”	1	12	1	12倍
CPL A	累加器求反	1	12	2	6倍
RL A	累加器循环左移	1	12	1	12倍
RLC A	累加器带进位位循环左移	1	12	1	12倍
RR A	累加器循环右移	1	12	1	12倍
RRC A	累加器带进位位循环右移	1	12	1	12倍
SWAP A	累加器内高低半字节交换	1	12	1	12倍

数据传送类指令

助记符	功能说明	字节数	12时钟/机器周期所需时钟	1时钟/机器周期所需时钟	效率提升
MOV A, Rn	寄存器内容送入累加器	1	12	1	12倍
MOV A, direct	直接地址单元中的数据送入累加器	2	12	2	6倍
MOV A, @Ri	间接RAM中的数据送入累加器	1	12	2	6倍
MOV A, #data	立即数送入累加器	2	12	2	6倍
MOV Rn, A	累加器内容送入寄存器	1	12	2	6倍
MOV Rn, direct	直接地址单元中的数据送入寄存器	2	24	4	6倍
MOV Rn, #data	立即数送入寄存器	2	12	2	6倍
MOV direct, A	累加器内容送入直接地址单元	2	12	3	4倍
MOV direct, Rn	寄存器内容送入直接地址单元	2	24	3	8倍
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	24	4	6倍
MOV direct, @Ri	间接RAM中的数据送入直接地址单元	2	24	4	6倍
MOV direct, #data	立即数送入直接地址单元	3	24	3	8倍
MOV @Ri, A	累加器内容送间接RAM单元	1	12	3	4倍
MOV @Ri, direct	直接地址单元数据送入间接RAM单元	2	24	4	6倍
MOV @Ri, #data	立即数送入间接RAM单元	2	12	3	4倍
MOV DPTR,#data16	16位立即数送入数据指针	3	24	3	8倍
MOVC A, @A+DPTR	以DPTR为基地址变址寻址单元中的数据送入累加器	1	24	4	6倍
MOVC A, @A+PC	以PC为基地址变址寻址单元中的数据送入累加器	1	24	4	6倍
MOVX A, @Ri	逻辑上在外部的片内扩展RAM,(8位地址）送入累加器	1	24	3	8倍
MOVX @Ri, A	累加器送入逻辑上在外部的片内扩展RAM（8位地址）	1	24	4	6倍
MOVX A, @DPTR	逻辑上在外部的片内扩展RAM,(16位地址)送入累加器	1	24	3	8倍
MOVX @DPTR, A	累加器送逻辑上在外部的片内扩展RAM（16位地址）	1	24	3	8倍
PUSH direct	直接地址单元中的数据压入堆栈	2	24	4	6倍
POP direcct	栈底数据弹出送入直接地址单元	2	24	3	8倍
XCH A, Rn	寄存器与累加器交换	1	12	3	4倍
XCH A,direct	直接地址单元与累加器交换	2	12	4	3倍
XCH A, @Ri	间接RAM与累加器交换	1	12	4	3倍
XCHD A, @Ri	间接RAM的低半字节与累加器交换	1	12	4	3倍

布尔变量操作类指令

助记符		功能说明	字节数	12时钟/机器周期所需时钟	1时钟/机器周期所需时钟	效率提升
CLR	C	清零进位位	1	12	1	12倍
CLR	bit	清0直接地址位	2	12	4	3倍
SETB	C	置1进位位	1	12	1	12倍
SETB	bit	置1直接地址位	2	12	4	3倍
CPL	C	进位位求反	1	12	1	12倍
CPL	bit	直接地址位求反	2	12	4	3倍
ANL	C, bit	进位位和直接地址位相“与”	2	24	3	8倍
ANL	C, /bit	进位位和直接地址位的反码相“与”	2	24	3	8倍
ORL	C, bit	进位位和直接地址位相“或”	2	24	3	8倍
ORL	C, /bit	进位位和直接地址位的反码相“或”	2	24	3	8倍
MOV	C, bit	直接地址位送入进位位	2	12	3	4倍
MOV	bit, C	进位位送入直接地址位	2	24	4	6倍
JC	rel	进位位为1则转移	2	24	3	8倍
JNC	rel	进位位为0则转移	2	24	3	8倍
JB	bit, rel	直接地址位为1则转移	3	24	4	6倍
JNB	bit, rel	直接地址位为0则转移	3	24	4	6倍
JBC	bit, rel	直接地址位为1则转移，该位清0	3	24	5	4.8倍

控制转移类指令

助记符		功能说明	字节数	12时钟/机器周期所需时钟	1时钟/机器周期所需时钟	效率提升
ACALL	addr11	绝对（短）调用子程序	2	24	6	4倍
LCALL	addr16	长调用子程序	3	24	6	4倍
RET		子程序返回	1	24	4	6倍
RETI		中断返回	1	24	4	6倍
AJMP	addr11	绝对（短）转移	2	24	3	8倍
LJMP	addr16	长转移	3	24	4	6倍
SJMP	re1	相对转移	2	24	3	8倍
JMP	@A+DPTR	相对于DPTR的间接转移	1	24	3	8倍
JZ	re1	累加器为零转移	2	24	3	8倍
JNZ	re1	累加器非零转移	2	24	3	8倍
CJNE	A，direct，re1	累加器与直接地址单元比较，不相等则转移	3	24	5	4.8倍
CJNE	A，#data，re1	累加器与立即数比较，不相等则转移	3	24	4	6倍
CJNE	Rn，#data，re1	寄存器与立即数比较，不相等则转移	3	24	4	6倍
CJNE	@Ri，#data，re1	间接RAM单元与立即数比较，不相等则转移	3	24	5	4.8倍
DJNZ	Rn，re1	寄存器减1，非零转移	2	24	4	6倍
DJNZ	direct，re1	直接地址单元减1，非零转移	3	24	5	4.8倍
NOP		空操作	1	12	1	12倍

指令执行速度效率提升总结：

指令系统共包括111条指令，其中：

执行速度快24倍的	共1条
执行速度快12倍的	共12条
执行速度快9.6倍的	共1条
执行速度快8倍的	共19条
执行速度快6倍的	共39条
执行速度快4.8倍的	共4条
执行速度快4倍的	共21条
执行速度快3倍的	共14条

根据对指令的使用频率分析统计，STC12系列 1T的8051单片机比普通的8051单片机在同样的工作频率下运行速度提升了8～12倍。

指令执行时钟数统计（供参考）：

指令系统共包括111条指令，其中：

1个时钟就可执行完成的指令	共12条
2个时钟就可执行完成的指令	共20条
3个时钟就可执行完成的指令	共39条
4个时钟就可执行完成的指令	共33条
5个时钟就可执行完成的指令	共5条
6个时钟就可执行完成的指令	共2条

5.3 传统8051单片机指令定义详解(中文&English)

5.3.1 传统8051单片机指令定义详解

ACALL addr 11

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：2

二进制编码：

a10 a9 a8 1	0 0 1 0	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)) ← (PC_{7-0})
(SP)←(SP) + 1
((SP))←(PC_{15-8})
(PC_{10-0})← 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(000011B)，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)←(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)←(A) + #data

ADDC　A, <src-byte>

功能：带进位的加法。

说明：执行ADDC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)，*R*0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)←(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)←(A) + (C) + (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)←(A) + (C) + ((Ri))

ADDC　A, #data

指令长度(字节)：2

执行周期：1

二进制编码：

0 0 1 1	0 1 0 0	immediate data

操作：ADDC

(A)←(A) + (C) + #data

AJMP　addr 11

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：2

二进制编码：

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)← (PC)+ 2

$(PC_{10\text{-}0})$← 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(010101011B)，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 Pl, #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

二进制编码： | 0 1 0 1 | 1 r r r |

操作： ANL

(A)←(A) ∧ (Rn)

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)←(A) ∧ (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)←(A) ∧ ((Ri))

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)←(A) ∧ #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

(direct)←(direct) ∧ (A)

ANL　direct, #data

指令长度(字节)：3

执行周期：2

二进制编码：

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

操作：ANL

(direct)←(direct) ∧ #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：2

二进制编码：

1 0 0 0	0 0 1 0	bit address

操作：ANL

(C) ← (C) ∧ (bit)

ANL　C, /bit

指令长度(字节)：2

执行周期：2

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

操作：ANL

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

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，*R*7包含的数据为56H。如下指令序列：

```
            CJNE    R7,#60H, NOT-EQ
;           ...     ......               ; R7 = 60H.
NOT_EQ:     JC      REQ_LOW              ; IF R7 < 60H.
;           ...     .....                ; 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*R*7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*P*l端口的数据是其他的值，那么程序在此不停地循环，直到*P*1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

二进制编码：| 1 0 1 1 | 0 1 0 1 | | direct address | | rel. address |

操作：(PC) ← (PC) + 3

IF (A) < > *(direct)*

THEN

　　(PC) ← (PC) + *relative offset*

IF (A) < *(direct)*

THEN

　　(C) ← 1

ELSE

　　(C) ← 0

CJNE A, #data, rel

指令长度(字节)： 3

执行周期： 2

二进制编码：

1 0 1 1	0 1 0 1	immediata data	rel. address

操作： (PC) ← (PC) + 3
IF (A) < > *(data)*
THEN
(PC) ← (PC) + *relative offset*
IF (A) < *(data)*
THEN
(C) ← 1
ELSE
(C) ← 0

CJNE Rn, #data, rel

指令长度(字节)： 3

执行周期： 2

二进制编码：

1 0 1 1	1 r r r	immediata data	rel. address

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

CJNE @Ri,#data,rel

指令长度(字节)： 3

执行周期： 2

二进制编码：

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

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

CLR A

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)← 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，Pl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) ← 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) ← 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。CLR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*P*1端口的数据为5BH(01011011B)，那么指令

CLR P1.1

CLR P1.2

执行完后，*P*l端口被设置为5BH(01011011B)。

CPL C

指令长度(字节)： 1

执行周期： 1

二进制编码：

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

二进制编码：

1 0 1 1	0 0 1 0	bit address

操作： CPL

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

DA A

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0lH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： 1

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0FFH。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0FFH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

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

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)←(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

((Ri))←((Ri)) - 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11111011B)，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0001000B)，正好符合251＝13×18＋17。进位和溢出标志都被清零。

指令长度(字节)：1

执行周期：4

二进制编码：

1 0 0 0	0 1 0 0

操作：DIV

$\begin{matrix}(A)_{15-8} \\ (B)_{7-0}\end{matrix} \leftarrow (A)/(B)$

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0FFH。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

二进制编码：

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

二进制编码：

操作：DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) – 1
IF (direct) > 0 or (direct) < 0
　THEN
　　(PC) ← (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，该指令步影响标志位。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0FFH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

二进制编码：

0 0 0 0	0 1 0 0

操作：INC
(A) ← (A)+1

INC　Rn

指令长度(字节)：1

执行周期：1

二进制编码：

0 0 0 0	1 r r r

操作：INC
(Rn) ← (Rn)+1

INC direct

指令长度(字节)： 2

执行周期： 1

二进制编码：

0 0 0 0	0 1 0 1	direct address

操作： INC

(direct)←(direct) + 1

INC @Ri

指令长度(字节)： 1

执行周期： 1

二进制编码：

0 0 0 0	0 1 1 i

操作： INC

((Ri))←((Ri)) + 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

```
INC 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： 2

二进制编码：

1 0 1 0	0 0 1 1

操作： INC

(DPTR) ← (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：2

二进制编码：

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

操作：JB

(PC) ← (PC)+ 3
IF (bit) = 1
　THEN
　　(PC) ← (PC) + 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：2

二进制编码：

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

操作：JBC

(PC) ← (PC)+ 3
IF (bit) = 1
　THEN
　　(bit) ← 0
　　(PC) ← (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：2

二进制编码：

0 1 0 0	0 0 0 0	rel. address

操作：JC

(PC) ← (PC)+ 2
IF (C) = 1
THEN
(PC) ← (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：2

二进制编码：

0 1 1 1	0 0 1 1

操作：JMP

(PC) ← (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110010108，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： 2

二进制编码：

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

操作： JNB
(PC) ← (PC)+ 3
IF (bit) = 0
THEN (PC) ← (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： 2

二进制编码：

0 1 0 1	0 0 0 0	rel. address

操作： JNC
(PC) ← (PC)+ 2
IF (C) = 0
THEN (PC) ← (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：2

二进制编码：

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：2

二进制编码：

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： 2

二进制编码：

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： 2

二进制编码：

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, @Rl      ;B < = 10H
MOV     @Rl, Pl     ;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： 2

二进制编码：

1 0 1 0	1 r r r	direct addr.

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： 2

二进制编码：

1 0 0 0	0 1 0 1	dir.addr. (src)

操作： MOV
(direct)← (direct)

MOV direct, @Ri

指令长度(字节)： 2

执行周期： 2

二进制编码：

1 0 0 0	0 1 1 i	direct addr.

操作： MOV
(direct)←((Ri))

MOV direct,#data

指令长度(字节)： 3

执行周期： 2

二进制编码：

0 1 1 1	0 1 0 1	direct address

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： 2

二进制编码：

1 0 1 0	0 1 1 i	direct addr.

操作： MOV
((Ri)) ← (direct)

MOV @Ri, #data

指令长度(字节)： 2

执行周期： 1

二进制编码：

0 1 1 1	0 1 1 i	immediate data

操作： MOV
((Ri)) ← #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P2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：

1 0 1 0	0 0 1 1	bit address

操作：MOV

(C) ← (bit)

MOV bit,C

指令长度(字节)：2

执行周期：2

二进制编码：

1 0 0 1	0 0 1 0	bit address

操作：MOV

(bit)← (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：2

二进制编码：

1 0 0 1	0 0 0 0	immediate data 15-8

操作：MOV

(DPTR) ← $\#data_{15\text{-}0}$

DPH DPL ← $\#data_{15\text{-}8}$ $\#data_{7\text{-}0}$

MOVC A , @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： 2

二进制编码：

1 0 0 1	0 0 1 1

操作： MOVC

(A) ← ((A)+(DPTR))

MOVC A,@A+PC

指令长度(字节)： 1

执行周期： 2

二进制编码：

1 0 0 0	0 0 1 1

操作： MOVC

(PC) ← (PC)+1

(A) ← ((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Rl间址寻址的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：Inte1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Pl和P2用作通用输入/输出端口。R0和Rl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： 2

二进制编码：

1 1 1 0	0 0 1 i

操作： MOVX

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

MOVX A,@DPTR

指令长度(字节)： 1

执行周期： 2

二进制编码：

1 1 1 0	0 0 0 0

操作： MOVX

$(A) \leftarrow ((DPTR))$

MOVX　@Ri, A

指令长度(字节)：1

执行周期：2

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

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12 800 (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：4

二进制编码：

1 0 1 0	0 1 0 0

操作：MUL

$(A)_{7-0}$ ← (A)×(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) ← (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*R*0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

二进制编码：

0	1	0	0	1	r	r	r

操作：ORL

(A) ← (A)∨(Rn)

ORL A,direct

指令长度(字节)：2

执行周期：1

二进制编码：

0	1	0	0	0	1	0	1	direct address

操作：ORL

(A)← (A)∨(direct)

ORL A,@Ri

指令长度(字节)：1

执行周期：1

二进制编码：

0	1	0	0	0	1	1	i

操作：ORL

(A)← (A)∨((Ri))

ORL A,#data

指令长度(字节)：2

执行周期：1

二进制编码：

0	1	0	0	0	1	0	0	immediate data

操作：ORL

(A)← (A)∨ #data

ORL direct, A

指令长度(字节)：2

执行周期：1

二进制编码：

0	1	0	0	0	0	1	0	direct address

操作：ORL

(direct)← (direct)∨(A)

ORL direct, #data

指令长度(字节)：3

执行周期：2

二进制编码：

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

操作：ORL

(direct) ← (direct)∨#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：2

二进制编码：

0 1 1 1	0 0 1 0	bit address

操作：ORL

$(C) \leftarrow (C) \vee (bit)$

ORL C, /bit

指令长度（字节）：2

执行周期：2

二进制编码：

1 0 1 0	0 0 0 0	bit address

操作：ORL

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

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：2

二进制编码：

1 1 0 1	0 0 0 0	direct address

操作：POP

(diect) $\leftarrow$ ((SP))

(SP) $\leftarrow$ (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：2

二进制编码：

1 1 0 0	0 0 0 0	direct address

操作：PUSH

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

$((SP)) \leftarrow (direct)$

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：2

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：2

二进制编码：

0 0 1 1	0 0 1 0

操作：RETI

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

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

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

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

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：设累加器的内容为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

二进制编码：

0 0 1 0	0 0 1 1

操作：RL

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

$(A0) \leftarrow (A7)$

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

二进制编码：

0 0 1 1	0 0 1 1

操作： RLC

(An+1) ← (An)　n = 0-6

(A0) ← (C)

(C) ← (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) ← (An+1)　n = 0 - 6

(A7) ← (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) ← (An) n = 0-6

(A7) ← (C)

(C) ← (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) ← 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

(bit) ← 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： 2

二进制编码：

1 0 0 0	0 0 0 0	rel. address

操作： SJMP

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

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

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*R*2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) ← (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) ← (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) ← (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) ← (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

$(A_{3-0}) \rightleftarrows (A_{7-4})$

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

二进制编码：

1 1 0 0	1 r r r

操作： XCH

(A) $\rightleftarrows$ (Rn)

XCH A, direct

指令长度(字节)： 2

执行周期： 1

二进制编码：

1 1 0 0	0 1 0 1	direct address

操作： XCH

(A) $\rightleftarrows$ (direct)

XCH A, @Ri

指令长度(字节)： 1

执行周期： 1

二进制编码：

1 1 0 0	0 1 1 i

操作： XCH

(A) $\rightleftarrows$ ((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 (011110101B)。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

二进制编码：

1 1 0 1	0 1 1 i

操作：XCHD

$(A_{3-0}) \rightleftarrows (Ri_{3-0})$

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

二进制编码：

0	1	1	0	1	r	r	r

操作：XRL

(A) ← (A) ⊻ (Rn)

XRL　A, direct

指令长度(字节)：2

执行周期：1

二进制编码：

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

二进制编码：

0	1	1	0	0	1	1	i

操作：XRL

(A) ← (A) ⊻ ((Ri))

XRL　A, #data

指令长度(字节)：2

执行周期：1

二进制编码：

0	1	1	0	0	1	0	0	immediate data

操作：XRL

(A) ← (A) ⊻ #data

XRL　direct, A

指令长度(字节)：2

执行周期：1

二进制编码：

0	1	1	0	0	0	1	0	direct address

操作：XRL

(direct) ← (direct) ⊻ (A)

XRL　direct, #dataw

指令长度(字节)：3

执行周期：2

二进制编码：

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

操作：XRL

(direct) ← (direct) ⊻ # data

5.3.2 Instruction Definitions of Traditional 8051 MCU

ACALL addr 11

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 suceesively 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 executingthe 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: 2

Encoding:

a10 a9 a8 1	0 0 1 0	a7 a6 a5 a4	a3 a2 a1 a0

Operation: ACALL
(PC)← (PC)+ 2
(SP)←(SP) + 1
((SP)) ← (PC_{7-0})
(SP)←(SP) + 1
((SP))←(PC_{15-8})
(PC_{10-0})← 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 occured.

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)←(A) + (Rn)

ADD A,direct

Bytes: 2

Cycles: 1

Encoding:

0 0 1 0	0 1 0 1	direct address

Operation: ADD
(A)←(A) + (direct)

ADD A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 1 0	0 1 1 i

Operation: ADD
(A)←(A) + ((Ri))

ADD A,#data

Bytes: 2

Cycles: 1

Encoding:

0 0 1 0	0 1 0 0	immediate data

Operation: ADD
(A)←(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 occured.
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 (01101101B) 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

Encoding:

0 0 1 1	1 r r r

Operation: ADDC

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

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 1	direct address

Operation: ADDC

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

ADDC A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 1 1	0 1 1 i

Operation: ADDC

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

ADDC A,#data

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 0	immediate data

Operation: ADDC

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

AJMP addr 11

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: 2

Encoding:

a10 a9 a8 0	0 0 0 1	a7 a6 a5 a4	a3 a2 a1 a0

Operation: AJMP

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

$(PC_{10\text{-}0}) \leftarrow$ 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.

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 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 Pl, #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)←(A) ∧ (Rn)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 1 0 1	direct address

Operation: ANL
(A)←(A) ∧ (direct)

ANL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 0 1	0 1 1 i

Operation: ANL
(A)←(A) ∧ ((Ri))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 1 0 0	immediate data

Operation: ANL
(A)←(A) ∧ #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 0 1 0	direct address

Operation: ANL
(direct)←(direct) ∧ (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:

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

Operation: ANL
(direct)←(direct) ∧ #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 affceted.* No other flsgs 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: 2

Encoding:

1 0 0 0	0 0 1 0	bit address

Operation: ANL
(C) ← (C) ∧ (bit)

ANL C, /bit

Bytes: 2

Cycles: 2

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 Pl, the program will loop at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation: (PC) ← (PC) + 3

IF (A) < > *(direct)*

THEN

(PC) ← (PC) + *relative offset*

IF (A) < *(direct)*

THEN

(C) ← 1

ELSE

(C) ← 0

CJNE A,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1 0 1 1	0 1 0 1	immediata data	rel. address

Operation: (PC) ← (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

Encoding:

1 0 1 1	1 r r r	immediata 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

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

CLR A

Function: Clear Accumulator

Description: The Aecunmlator 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)← 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) ← 0

CLR bit

Bytes: 2

Cycles: 1

Encoding:

1 1 0 0	0 0 1 0	bit address

Operation: CLR

(bit) ← 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 (101000011B).

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 5DH (01011101B). The instruction,

CLR P1.1

CLR P1.2

will leave the port set to 59H (01011001B).

CPL C

Bytes: 1

Cycles: 1

Encoding:

1 0 1 1	0 0 1 1

Operation: CPL

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

CPL bit

Bytes: 2

Cycles: 1

Encoding:

1 0 1 1	0 0 1 0	bit address

Operation: CPL

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

DA A

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-111xxxx), 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 twos-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: 1

Encoding:

1 1 0 1	0 1 0 0

Operation: DA

-contents of Accumulator are BCD

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

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

AND

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

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

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 (01111111B). 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)←(direct) −1

DEC @Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 0 1	0 1 1 i

Operation: DEC
((Ri))←((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 00010010B) in B, since 251 = (13×18) + 17. Carry and OV will both be cleared.

Bytes: 1

Cycles: 4

Encoding:

1 0 0 0	0 1 0 0

Operation: DIV
$(A)_{15\text{-}8}$
$(B)_{7\text{-}0}$ ← (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 0FFH. No flags are afected. 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

Encoding:

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

Encoding:

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

Operation: DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) – 1
IF (direct) > 0 or (direct) < 0
THEN
(PC) ← (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 (011111110B). 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) ← (A)+1

INC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	1 r r r

Operation: INC
(Rn) ← (Rn)+1

INC direct

Bytes: 2

Cycles: 1

Encoding:

0 0 0 0	0 1 0 1	direct address

Operation: INC
(direct)←(direct) + 1

INC @Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 1 i

Operation: INC
((Ri))←((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,
INC DPTR
INC DPTR
INC DPTR
will change DPH and DPL to 13H and 01H.

Bytes: 1

Cycles: 2

Encoding:

1 0 1 0	0 0 1 1

Operation: INC
(DPTR) ← (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: 2

Encoding:

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

Operation: JB
(PC) ← (PC)+ 3
IF (bit) = 1
THEN
(PC) ← (PC) + 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 wili 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: 2

Encoding:

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

Operation: JBC
(PC) ← (PC)+ 3
IF (bit) = 1
THEN
(bit) ← 0
(PC) ← (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      LABEL2s
```

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

Bytes: 2

Cycles: 2

Encoding:

0 1 0 0	0 0 0 0	rel. address

Operation: JC
(PC) ← (PC)+ 2
IF (C) = 1
THEN
(PC) ← (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 2^{16}): 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: 2

Encoding:

0 1 1 1	0 0 1 1

Operation: JMP
(PC) ← (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: 2

Encoding:

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

Operation: JNB
(PC) ← (PC)+ 3
IF (bit) = 0
  THEN (PC) ← (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: 2

Encoding:

0 1 0 1	0 0 0 0		rel. address

Operation: JNC
(PC) ← (PC)+ 2
IF (C) = 0
THEN (PC) ← (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   LAEEL2
```

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

Bytes: 2

Cycles: 2

Encoding:

0 1 1 1	0 0 0 0		rel. address

Operation: JNZ
(PC) ← (PC)+ 2
IF (A) ≠ 0
THEN (PC) ← (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: 2

Encoding:

0 1 1 0	0 0 0 0	rel. address

Operation: JZ
(PC) ← (PC)+ 2
IF (A) = 0
THEN (PC) ← (PC) + rel

LCALL addr16

Function: Long call

Description: LCALL calls a subroutine loated 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: 2

Encoding:

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

Operation: LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC) ← $addr_{15-0}$

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: 2

Encoding:

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

Operation: LJMP
(PC) ← $addr_{15\text{-}0}$

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, @Rl      ;B < = 10H
MOV    @Rl, Pl     ;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.

MOV A,Rn

Bytes: 1

Cycles: 1

Encoding:

1 1 1 0	1 r r r

Operation: MOV
(A) ← (Rn)

***MOV A,direct**

Bytes: 2

Cycles: 1

Encoding:

1 1 1 0	0 1 0 1	direct address

Operation: MOV
(A)← (direct)

***MOV A, 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) ← ((Ri))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 0 0	immediate data

Operation: MOV
(A)← #data

MOV Rn, A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	1 r r r

Operation: MOV
(Rn)←(A)

MOV Rn,direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	1 r r r	direct addr.

Operation: MOV
(Rn)←(direct)

MOV Rn,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	1 r r r	immediate data

Operation: MOV
(Rn) ← #data

MOV direct, A

Bytes: 2

Cycles: 1

Encoding:

1 1 1 1	0 1 0 1	direct address

Operation: MOV
(direct) ← (A)

MOV direct, Rn

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	1 r r r	direct address

Operation: MOV
(direct) ← (Rn)

MOV direct, direct

Bytes: 3

Cycles: 2

Encoding:

1 0 0 0	0 1 0 1	dir.addr. (src)

Operation: MOV
(direct)← (direct)

MOV direct, @Ri

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 1 1 i	direct addr.

Operation: MOV
(direct)←((Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding:

0 1 1 1	0 1 0 1	direct address

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: 2

Encoding:

1 0 1 0	0 1 1 i	direct addr.

Operation: MOV

((Ri)) ← (direct)

MOV @Ri, #data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 1 i	immediate data

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 1	bit address

Operation: MOV

(C) ← (bit)

MOV bit,C

Bytes: 2

Cycles: 2

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: 2

Encoding:

1 0 0 1	0 0 0 0	immediate data 15-8

Operation: MOV
(DPTR) ← $\#data_{15-0}$
DPH DPL ← $\#data_{15-8}$ $\#data_{7-0}$

MOVC A , @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 defimed 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: 2

Encoding:

1 0 0 1	0 0 1 1

Operation: MOVC
(A) ← ((A)+(DPTR))

MOVC A,@A+PC

Bytes: 1

Cycles: 2

Encoding:

1 0 0 0	0 0 1 1

Operation: MOVC
(PC) ← (PC)+1
(A) ← ((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: 2

Encoding:

1 1 1 0	0 0 1 i

Operation: MOVX
(A) ← ((Ri))

MOVX A,@DPTR

Bytes: 1

Cycles: 2

Encoding:

1 1 1 0	0 0 0 0

Operation: MOVX
(A) ← ((DPTR))

MOVX @Ri, A

Bytes: 1

Cycles: 2

Encoding:

1 1 1 1	0 0 1 i

Operation: MOVX
((Ri))← (A)

MOVX @DPTR, A

Bytes: 1

Cycles: 2

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

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: 4

Encoding:

1 0 1 0	0 1 0 0

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

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) ← (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 1of output Port 1.

ORL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0 1 0 0	1 r r r

Operation: ORL

(A) ← (A)∨(Rn)

ORL A,direct

Bytes: 2

Cycles: 1

Encoding:

0 1 0 0	0 1 0 1	direct address

Operation: ORL

(A)← (A)∨(direct)

ORL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 0 0	0 1 1 i

Operation: ORL

(A)← (A)∨((Ri))

ORL A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 0 0	0 1 0 0	immediate data

Operation: ORL

(A)← (A)∨ #data

ORL direct, A

Bytes: 2

Cycles: 1

Encoding:

0 1 0 0	0 0 1 0	direct address

Operation: ORL

(direct)← (direct)∨(A)

ORL direct, #data

Bytes: 3

Cycles: 2

Encoding:

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

Operation: ORL

(direct) ← (direct)∨#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: 2

Encoding:

0 1 1 1	0 0 1 0	bit address

Operation: ORL

$(C) \leftarrow (C) \vee (bit)$

ORL C, /bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 0 0 0	bit address

Operation: ORL

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

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: 2

Encoding:

1 1 0 1	0 0 0 0	direct address

Operation: POP

(diect) $\leftarrow$ ((SP))

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

PUSH direct

Function: Push onto stack

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

Example: On entering 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: 2

Encoding:

1 1 0 0	0 0 0 0	direct address

Operation: PUSH
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (direct)$

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: 2

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: 2

Encoding:

0 0 1 1	0 0 1 0

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

RL A

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,

RL A

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
$(An+1) \leftarrow (An) \quad n = 0\text{-}6$
$(A0) \leftarrow (A7)$

RLC A

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) ← (An) n = 0-6
(A0) ← (C)
(C) ← (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) ← (An+1) n = 0 - 6
(A7) ← (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) ← (An) n = 0-6
(A7) ← (C)
(C) ← (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) ← 1

SETB bit

Bytes: 2

Cycles: 1

Encoding:

1 1 0 1	0 0 1 0	bit address

Operation: SETB
(bit) ← 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 0FEH would be an one-instruction infinite loop).

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 0 0 0	rel. address

Operation: SJMP
(PC) ← (PC)+2
(PC) ← (PC)+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

$(A_{3-0}) \rightleftarrows (A_{7-4})$

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 (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

XCH A, @R0

will leave RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A, Rn

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	1 r r r

Operation: XCH

$(A) \rightleftarrows (Rn)$

XCH A, direct

Bytes: 2

Cycles: 1

Encoding:

1 1 0 0	0 1 0 1	direct address

Operation: XCH

$(A) \rightleftarrows (direct)$

XCH A, @Ri

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	0 1 1 i

Operation: XCH

$(A) \rightleftarrows ((Ri))$

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

$(A_{3-0}) \rightleftarrows (Ri_{3-0})$

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 vatue 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinnation 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 outpue Port 1.

XRL A, Rn

Bytes: 1

Cycles: 1

Encoding:

0 1 1 0	1 r r r

Operation: XRL

(A) ← (A) $\oplus$ (Rn)

XRL A, direct

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 1	direct address

Operation: XRL

(A) ← (A) $\oplus$ (direct)

XRL A, @Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 1 0	0 1 1 i

Operation: XRL

(A) ← (A) $\oplus$ ((Ri))

XRL A, #data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 0	immediate data

Operation: XRL

(A) ← (A) $\oplus$ #data

XRL direct, A

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 0 1 0	direct address

Operation: XRL

(direct) ← (direct) $\oplus$ (A)

XRL direct, #data

Bytes: 3

Cycles: 2

Encoding:

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

Operation: XRL

(direct) ← (direct) $\oplus$ # data

第6章 中断系统

中断系统是为使CPU具有对外界紧急事件的处理能力而设置的。

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。

STC12C5410AD系列单片机提供了9个中断请求源，它们分别是：外部中断0($\overline{\text{INT0}}$)、定时器0中断、外部中断1($\overline{\text{INT1}}$)、定时器1中断、串口(UART)中断、A/D转换中断和SPI中断、PCA模块中断和低压检测(LVD)中断。所有的中断都具有4个中断优先级。用户可以用关总中断允许位(EA/IE.7)或相应中断的允许位来屏蔽所有的中断请求，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；每一个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及同优先级的中断。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。STC12C5410AD系列单片机的各个中断查询次序如下表6-1所示：

表6-1 中断查询次序

中断源	中断向量地址	相同优先级内的查询次序	中断优先级设置(IPH,IP)	优先级0（最低）	优先级1	优先级2	优先级3（最高）	中断请求标志位	中断允许控制位
$\overline{INT0}$ (外部中断 0)	0003H	0 (highest)	PX0H, PX0	0, 0	0, 1	1, 0	1, 1	IE0	EX0/EA
Timer 0	000BH	1	PT0H, PT0	0, 0	0, 1	1, 0	1, 1	TF0	ET0/EA
$\overline{INT1}$ (外部中断1)	0013H	2	PX1H, PX1	0, 0	0, 1	1, 0	1, 1	IE1	EX1/EA
Timer1	001BH	3	PT1H, PT1	0, 0	0, 1	1, 0	1, 1	TF1	ET1/EA
UART	0023H	4	PSH, PS	0, 0	0, 1	1, 0	1, 1	RI+TI	
ADC/SPI	002BH	5	PADC_SPIH, PADC_SPI	0, 0	0, 1	1, 0	1, 1	ADC_FLAG+SPIF	(EADCI+ESPI) / EADC_SPI / EA
PCA/LVD	0033H	6 (lowest)	PPCA_LVDH, PPCA_LVD	0, 0	0, 1	1, 0	1, 1	CF+CCF0+CCF1+CCF2+CCF3+LVDF	(ECF+ECCF0+ECCF1+ECCF2+ECCF3+ELVD) /EPCA_LVD/EA

通过设置新增加的特殊功能寄存器IPH中的相应位，可将中断优先级设为四级，如果只设置IP，那么中断优先级就只有两级，与传统8051单片机两级中断优先级完全兼容。

如果使用C语言编程，中断查询次序号就是中断号，例如：

```
void    Int0_Routine(void)          interrupt 0;
void    Timer0_Rountine(void)       interrupt 1;
void    Int1_Routine(void)          interrupt 2;
void    Timer1_Rountine(void)       interrupt 3;
void    UART_Routine(void)          interrupt 4;
void    ADC_SPI_Routine(void)       interrupt 5;
void    PCA_LVD_Routine(void)       interrupt 6;
```

6.1 中断结构

STC12C5410AD系列单片机的中断系统结构示意图如图6-1所示

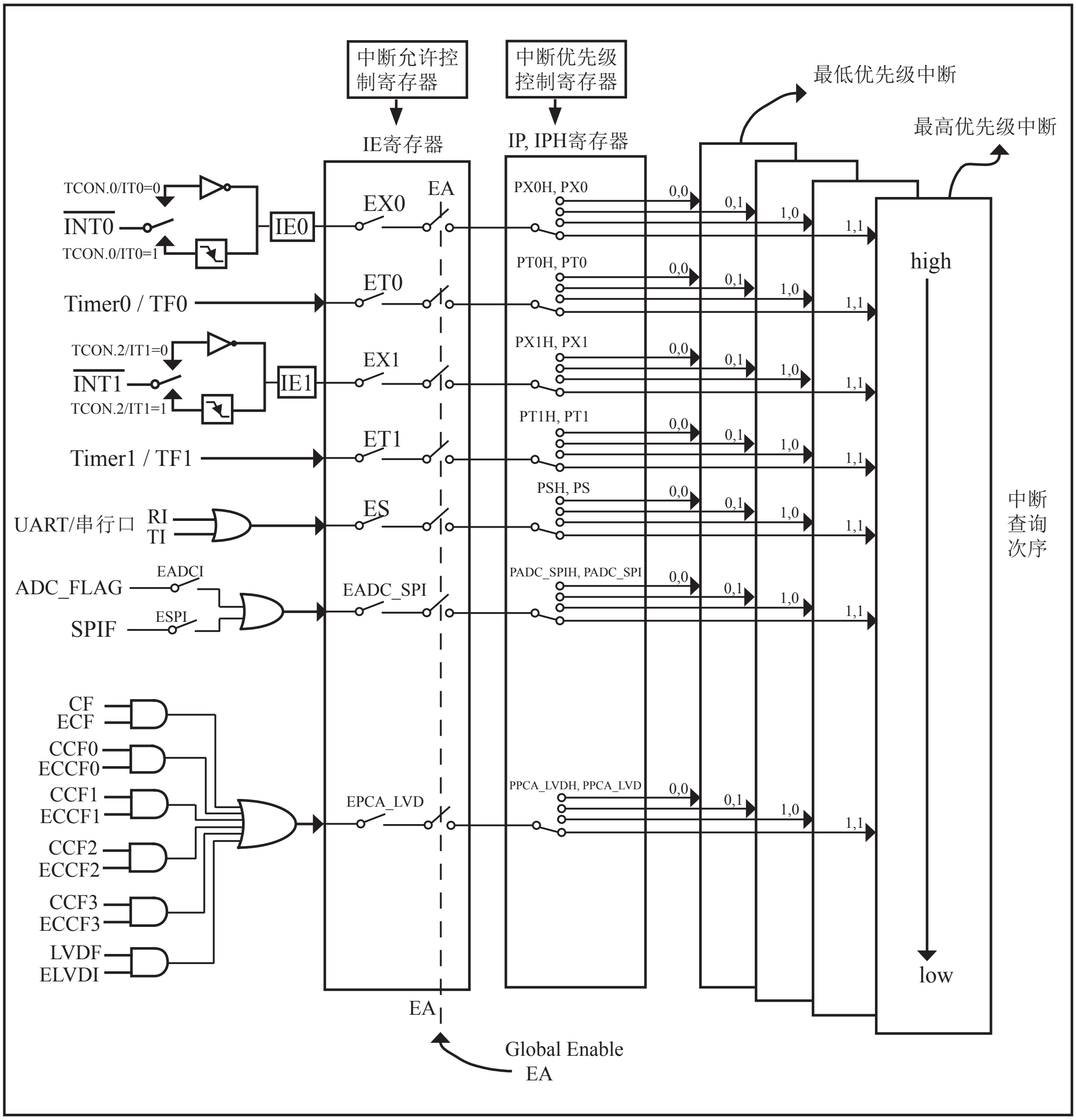


图6-1 STC12C5410AD系列中断系统结构图

外部中断0($\overline{\text{INT0}}$)和外部中断1($\overline{\text{INT1}}$)既可低电平触发，也可下降沿触发。请求两个外部中断的标志位是位于寄存器TCON中的IE0/TCON.1和IE1/TCON.3。当外部中断服务程序被响应后，中断请求标志位IE0和IE1会自动被清0。TCON寄存器中的IT0/TCON.0和IT1/TCON.2决定了外部中断0和1是低电平触发方式还是下降沿触发方式。如果ITx = 0(x = 0,1)，那么系统在INTx(x = 0,1)脚探测到低电平后可产生外部中断。如果ITx = 1(x = 0,1)，那么系统在INTx(x= 0,1)脚探测下降沿后可产生外部中断。外部中断0($\overline{\text{INT0}}$)和外部中断1($\overline{\text{INT1}}$)还可以用于将单片机从掉电模式唤醒。

定时器0和1的中断请求标志位是TF0和TF1。当定时器寄存器THx/TLx(x = 0,1)溢出时，溢出标志位TFx(x = 0,1)会被置位，定时器中断发生。当单片机转去执行该定时器中断时，定时器的溢出标志位TFx(x = 0,1)会被硬件清除。

当串行口接收中断请求标志位RI和串行口发送中断请求标志位TI中的任何一个被置为1后，串行口中断都会产生。

如果要允许A/D转换中断则需要将几个相应的控制位置1:

1、将EADCI置1，允许ADC中断，这是ADC中断的单独控制位。
2、将EADCI_SPI置1，允许ADC中断及SPI中断，这是ADC中断及SPI中断的总中断控制位，此位不打开，也是无法产生ADC中断的。
3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生ADC中断的A/D中断服务程序中要用软件清A/D中断请求标志位ADC_FLAG。

如果要允许SPI中断则需要将几个相应的控制位置1:

1、将ESPI置1，允许SPI中断，这是SPI中断的单独控制位。
2、将EADCI_SPI置1，允许ADC中断及SPI中断，这是ADC中断及SPI中断的总中断控制位，此位不打开，也是无法产生SPI中断的。
3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生SPI 中断的
SPI中断服务程序中要用软件清SPI中断请求标志位SPIF。

如果要允许低压中断则需要将几个相应的控制位置1:

1、将ELVDI置1，允许低压检测中断，这是低压中断的单独控制位。
2、将EPCA_LVD置1，允许PCA模块中断及低压检测中断，这是PCA模块中断及低压检测中断的总中断控制位，此位不打开，也是无法产生低压检测中断的。
3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生低压检测中断的低压检测中断服务程序中要用软件清低压中断请求标志位LVDF。

5V单片机，3.7V(±0.1)以下为低压，3V单片机，2.4V(±0.1)以下为低压，如ELVDI=1(允许低压中断)，则会产生低压中断

如果要允许PCA中断则需要将几个相应的控制位置1:

1、将ECF/ECCF0/ECCF1/ECCF2/ECCF3中断允许位需要置1的位置1，允许PCA模块中相应的模块产生中断，这些是PCA模块中相应模块的单独控制位。

2、将EPCA_LVD置1，允许PCA模块中断及低压检测中断，这是PCA模块中断及低压检测中断的总中断控制位，此位不打开，也是无法产生PCA 中断的。

3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生PCA中断的PCA中断服务程序中要用软件清相应的PCA中断请求标志位CF/CCF0/CCF1/CCF2/CCF3。

各个中断触发行为总结如下表6-2所示：

表6-2　中断触发

中断源	触发行为
$\overline{\text{INT0}}$ (外部中断0)	(IT0/TCON.0 = 1): 下降沿　　(IT0/TCON.0 = 0): 低电平
Timer 0	定时器0溢出
$\overline{\text{INT1}}$ (外部中断1)	(IT1/TCON.2 = 1): 下降沿　　(IT1/TCON.2 = 0): 低电平
Timer1	定时器1溢出
UART	发送或接受完成
ADC	A/D转换完成
LVD	电源电压下降到低于LVD检测电压
SPI	SPIF=1

6.2 中断寄存器

符号	描述	地址	位地址及符号 MSB							LSB	复位值
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
TCON	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
SCON	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000 0000B
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
PCON	Power Control	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
ADC_CONTR	ADC Control	C5H	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0	0000 0000B
CCON	PCA Control Register	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0	00xx 0000B
CMOD	PCA Mode Register	D9H	CIDL	-	-	-	CPS2	CPS1	CPS0	ECF	00xx 0000B
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000 0000B
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000 0000B
CCAPM2	PCA Module 2 Mode Register	DCH	-	ECOM2	CAPP2	CAPN2	MAT2	TOG2	PWM2	ECCF2	x000 0000B
CCAPM3	PCA Module 3 Mode Register	DDH	-	ECOM3	CAPP3	CAPN3	MAT3	TOG3	PWM3	ECCF3	x000 0000B
SPSTAT	SPI Status register	84H	SPIF	WCOL	-	-	-	-	-	-	

上表中列出了与STC12C5410AD系列单片机中断相关的所有寄存器，下面逐一地对部分寄存器进行介绍。

1. 中断允许寄存器IE、AUXR和WAKE_CLKO

STC12C5410AD系列单片机CPU对中断源的开放或屏蔽，每一个中断源是否被允许中断，是由内部的中断允许寄存器IE（IE为特殊功能寄存器，它的字节地址为A8H）控制的，其格式如下：

IE：中断允许寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA：CPU的总中断允许控制位，EA=1，CPU开放中断，EA=0，CPU屏蔽所有的中断申请。
EA的作用是使中断允许形成两级控制。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。

EPCA_LVD：PCA模块中断和低压检测中断允许位。这是PCA模块中断及低压检测LVD中断的总中断控制位，此位不打开，也是无法产生PCA中断和LVD中断的。
EPCA_LVD=1，允许PCA模块中断低压检测中断；
EPCA_LVD=0，禁止PCA模块中断低压检测中断。

EADC_SPI：A/D转换中断和SPI中断允许位，这是ADC中断及SPI中断的总中断控制位，此位不打开，也是无法产生ADC中断和SPI中断的。
EADC_SPI = 1，允许A/D转换中断和SPI中断；
EADC_SPI = 0，禁止A/D转换中断和SPI中断。

ES ：串行口1中断允许位。ES=1，允许串行口中断；ES=0，禁止串行口中断。

ET1：定时/计数器T1的溢出中断允许位。ET1=1，允许T1中断；ET1=0，禁止T1中断。

EX1：外部中断1中断允许位。EX1=1，允许外部中断1中断；EX1=0，禁止外部中断1中断。

ET0：T0的溢出中断允许位。ET0=1，允许T0中断；ET0=0禁止T0中断。

EX0：外部中断0中断允许位。EX0=1，允许中断；EX0=0禁止中断。

AUXR：辅助寄存器（不可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

EADCI: A/D转换中断单独允许位。
EADCI=1，允许A/D转换中断；
EADCI=0，禁止A/D转换中断。

ESPI: SPI中断单独允许位。
ESPI=1，允许SPI中断；
ESPI=0，禁止SPI中断。

ELVDI：低压检测LVD中断单独允许位。
ESPI=1，允许低压检测LVD中断；
ESPI=0，禁止低压检测LVD中断。

WAKE_CLKO :Clock output and Power-down Wakeup Control register(不可位寻址)

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO

B7 - PCAWAKEUP：在掉电模式下，是否允许PCA上升沿/下降沿中断唤醒powerdown。
0：禁止PCA上升沿/下降沿中断唤醒powerdown；
1：允许PCA上升沿/下降沿中断唤醒powerdown 。

B6 - RXD_PIN_IE：掉电模式下，允许P3.0(RXD)下降沿置RI，也能使RXD唤醒powerdown.
0：禁止P3.0(RXD)下降沿置RI，也禁止RXD唤醒powerdown；
1：允许P3.0(RXD)下降沿置RI，也允许RXD唤醒powerdown 。

B5 - T1_PIN_IE：掉电模式下, 允许T1/P3.5脚下降沿置T1中断标志, 也能使T1脚唤醒powerdown.
0：禁止T1/P3.5脚下降沿置T1中断标志, 也禁止T1脚唤醒powerdown；
1：允许T1/P3.5脚下降沿置T1中断标志, 也允许T1脚唤醒powerdown 。

B4 - T0_PIN_IE：掉电模式下, 允许T0/P3.4脚下降沿置T0中断标志, 也能使T0脚唤醒powerdown.
0：禁止T0/P3.4脚下降沿置T0中断标志, 也禁止T0脚唤醒powerdown；
1：允许T0/P3.4脚下降沿置T0中断标志, 也允许T0脚唤醒powerdown 。

T1CLKO和T0CLKO与定时器有关，在此不作介绍

STC12C5410AD系列单片机复位以后，IE、AUXR和WAKE_CLKO被清0，由用户程序置“1”或清“0”IE、AUXR和WAKE_CLKO相应的位，实现允许或禁止各中断源的中断申请，若使某一个中断源允许中断必须同时使CPU开放中断。更新IE的内容可由位操作指令来实现（SETB BIT；CLR BIT），也可用字节操作指令实现（即MOV IE，#DATA，ANL IE，#DATA；ORL IE，#DATA；MOV IE，A等）。更新AUXR和WAKE_CLKO(不可位寻址)的内容可用MOV AUXR, #DATA与MOV WAKE_CLKO #DATA指令来解决。

2. 中断优先级控制寄存器IP和IPH

传统8051单片机具有两个中断优先级，即高优先级和低优先级，可以实现两级中断嵌套。STC12C5410AD系列单片机通过设置新增加的特殊功能寄存器(IPH)中的相应位，可将中断优先级设置为4个中断优先级；如果只设置IP，那么中断优先级只有两级，与传统8051单片机两级中断优先级完全兼容。

一个正在执行的低优先级中断能被高优先级中断所中断，但不能被另一个低优先级中断所中断，一直执行到结束，遇到返回指令RETI，返回主程序后再执行一条指令才能响应新的中断申请。以上所述可归纳为下面两条基本规则：

1. 低优先级中断可被高优先级中断所中断，反之不能。
2. 任何一种中断(不管是高级还是低级)，一旦得到响应，不会再被它的同级中断所中断

STC12C5410AD系列单片机的片内各优先级控制寄存器的格式如下：

IPH: 中断优先级控制寄存器高(不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IPH	B7H	name	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H

IP : 中断优先级控制寄存器低 (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IP	B8H	name	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0

PPCA_LVDH, PPCA_LVD: PCA中断和低压检测LVD中断优先级控制位。

当PPCA_LVDH = 0且PPCA_LVD = 0时，PCA中断和低压检测LVD中断为最低优先级中断(优先级0)
当PPCA_LVDH = 0且PPCA_LVD = 1时，PCA中断和低压检测LVD中断为较低优先级中断(优先级1)
当PPCA_LVDH = 1且PPCA_LVD = 0时，PCA中断和低压检测LVD中断为较高优先级中断(优先级2)
当PPCA_LVDH = 1且PPCA_LVD = 1时，PCA中断和低压检测LVD中断为最高优先级中断(优先级3)

PADC_SPIH, PADC_SPI: A/D转换中断和SPI中断优先级控制位。

当PADC_SPIH,=0且PADC_SPI=0时，A/D转换中断和SPI中断为最低优先级中断(优先级0)
当PADC_SPIH=0且PADC_SPI=1时，A/D转换中断和SPI中断为较低优先级中断(优先级1)
当PADC_SPIH=1且PADC_SPI=0时，A/D转换中断和SPI中断为较高优先级中断(优先级2)
当PADC_SPIH=1且PADC_SPI=1时，A/D转换中断和SPI中断为最高优先级中断(优先级3)

PSH, PS: 串口1中断优先级控制位。
当PSH=0且PS=0时，串口1中断为最低优先级中断(优先级0)
当PSH=0且PS=1时，串口1中断为较低优先级中断(优先级1)
当PSH=1且PS=0时，串口1中断为较高优先级中断(优先级2)
当PSH=1且PS=1时，串口1中断为最高优先级中断(优先级3)

PT1H, PT1: 定时器1中断优先级控制位。
当PT1H=0且PT1=0时，定时器1中断为最低优先级中断(优先级0)
当PT1H=0且PT1=1时，定时器1中断为较低优先级中断(优先级1)
当PT1H=1且PT1=0时，定时器1中断为较高优先级中断(优先级2)
当PT1H=1且PT1=1时，定时器1中断为最高优先级中断(优先级3)

PX1H, PX1: 外部中断1优先级控制位。
当PX1H=0且PX1=0时，外部中断1为最低优先级中断(优先级0)
当PX1H=0且PX1=1时，外部中断1为较低优先级中断(优先级1)
当PX1H=1且PX1=0时，外部中断1为较高优先级中断(优先级2)
当PX1H=1且PX1=1时，外部中断1为最高优先级中断(优先级3)

PT0H, PT0: 定时器0中断优先级控制位。
当PT0H=0且PT0=0时，定时器0中断为最低优先级中断(优先级0)
当PT0H=0且PT0=1时，定时器0中断为较低优先级中断(优先级1)
当PT0H=1且PT0=0时，定时器0中断为较高优先级中断(优先级2)
当PT0H=1且PT0=1时，定时器0中断为最高优先级中断(优先级3)

PX0H, PX0: 外部中断0优先级控制位。
当PX0H=0且PX0=0时，外部中断0为最低优先级中断(优先级0)
当PX0H=0且PX0=1时，外部中断0为较低优先级中断(优先级1)
当PX0H=1且PX0=0时，外部中断0为较高优先级中断(优先级2)
当PX0H=1且PX0=1时，外部中断0为最高优先级中断(优先级3)

中断优先级控制寄存器IP和IPH的各位都由可用户程序置“1”和清“0”。但IP寄存器可位操作，所以可用位操作指令或字节操作指令更新IP的内容。而IPH寄存器的内容只能用字节操作指令来更新。STC12C5410AD系列单片机复位后IP和IPH均为00H，各个中断源均为低优先级中断。

3. 定时器/计数器控制寄存器TCON

TCON为定时器/计数器T0、T1的控制寄存器，同时也锁存T0、T1溢出中断源和外部请求中断源等，TCON格式如下：

TCON：定时器/计数器中断控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
TCON	88H	name	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置“1”TF1，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1的运行控制位。

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0的运行控制位。

IE1：外部中断1请求源（$\overline{\text{INT1}}$/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1中断源类型选择位。IT1=0，$\overline{\text{INT1}}$/P3.3引脚上的低电平信号可触发外部中断1。IT1=1，外部中断1为下降沿触发方式。

IE0：外部中断0请求源（$\overline{\text{INT0}}$/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0中断源类型选择位。IT0=0，$\overline{\text{INT0}}$/P3.2引脚上的低电平可触发外部中断0。IT0=1，外部中断0为下降沿触发方式。

4. 串行口控制寄存器SCON

SCON为串行口控制寄存器，SCON格式如下：

SCON：串行口控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SCON	98H	name	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI

RI： 串行口1接收中断标志。若串行口1允许接收且以方式0工作，则每当接收到第8位数据时置1；若以方式1、2、3工作且SM2=0时，则每当接收到停止位的中间时置1；当串行口以方式2或方式3工作且SM2=1时，则仅当接收到的第9位数据RB8为1后，同时还要接收到停止位的中间时置1。RI为1表示串行口1正向CPU申请中断(接收中断)，RI必须由用户的中断服务程序清零。

TI： 串行口1发送中断标志。串行口1以方式0发送时，每当发送完8位数据，由硬件置1；若以方式1、方式2或方式3发送时，在发送停止位的开始时置1。TI=1表示串行口1正在向CPU申请中断(发送中断)。值得注意的是，CPU响应发送中断请求，转向执行中断服务程序时并不将TI清零，TI必须由用户在中断服务程序中清零。

SCON寄存器的其他位与中断无关，在此不作介绍。

5. 低压检测中断相关寄存器：电源控制寄存器PCON

PCON为电源控制寄存器，PCON格式如下：

PCON : 电源控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

LVDF : 低压检测标志位,同时也是低压检测中断请求标志位。

如果内部工作电压Vcc低于低压检测门槛电压，该位自动置1，与低压检测中断是否被允许无关。即在内部工作电压Vcc低于低压检测门槛电压时，不管有没有允许低压检测中断，该位都自动为1。该位要用软件清0，清0后，如内部工作电压Vcc继续低于低压检测门槛电压，该位又被自动设置为1。

电源控制寄存器PCON中的其他位与低压检测中断无关，在此不作介绍。

在中断允许寄存器IE中，低压检测中断相应的允许位是EPCA_LVD/IE.6

IE : 中断允许寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA : 中断允许总控制位。

EA=0, 屏蔽了所有的中断请求；

EA=1，开放总中断，但每个中断源还有自己的独立允许控制位。

EA的作用是使中断允许形成两级控制。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。

EPCA_LVD : PCA模块中断和低压检测中断允许位。

EPCA_LVD = 0，禁止PCA模块和低压检测中断；

EPCA_LVD = 1，允许PCA模块和低压检测中断。

6. A/D转换控制寄存器ADC_CONTR

ADC_CONTR为A/D转换控制寄存器，ADC_CONTR格式如下：

ADC_CONTR：A/D转换控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
ADC_CONTR	C5H	name	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0

ADC_POWER：ADC电源控制位。当ADC_POWER=0时，关闭ADC电源；
当ADC_PWOER=1时，打开ADC电源。

ADC_FLAG：ADC转换结束标志位，可用于请求A/D转换的中断。当A/D转换完成后，ADC_FLAG=1,要用软件清0。不管是A/D转换完成后由该位申请产生中断，还是由软件查询该标志位A/D转换是否结束，当A/D转换完成后，ADC_FLAG=1，一定要软件清0。

ADC_START：ADC转换启动控制位，设置为“1”时,开始转换，转换结束后为0。

A/D转换控制寄存器ADC_CONTR中的其他位与中断无关，在此不作介绍。

在中断允许寄存器IE中，A/D转换器的中断允许位是EADC_SPI/IE.5

IE：中断允许寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA：　中断允许总控制位。
EA=0，屏蔽了所有的中断请求；
EA=1，开放总中断，但每个中断源还有自己的独立允许控制位。
EA的作用是使中断允许形成两级控制。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。

EADC_SPI：A/D转换中断和SPI中断允许位。
EADC_SPI = 0，禁止A/D转换中断和SPI中断；
EADC_SPI = 1，允许A/D转换中断和SPI中断。

6.3 中断优先级

STC12C5410AD系列单片机的所有的中断都具有4个中断优先级，对于这些中断请求源可编程为高优先级中断或低优先级中断，可实现两级中断服务程序嵌套。一个正在执行的低优先级中断能被高优先级中断所中断，但不能被另一个低优先级中断所中断，一直执行到结束，遇到返回指令RETI，返回主程序后再执行一条指令才能响应新的中断申请。以上所述可归纳为下面两条基本规则：

1. 低优先级中断可被高优先级中断所中断，反之不能。
2. 任何一种中断（不管是高级还是低级），一旦得到响应，不会再被它的同级中断所中断。

当同时收到几个同一优先级的中断要求时，哪一个要求得到服务，取决于内部的查询次序。这相当于在每个优先级内，还同时存在另一个辅助优先级结构，STC12C5410AD系列单片机各中断优先查询次序如下：

	中断源	查询次序
0.	$\overline{\text{INT0}}$	(highest)
1.	Timer 0	
2.	$\overline{\text{INT1}}$	
3.	Timer 1	
4.	UART1	
5.	ADC_SPI	
6.	PCA_LVD	(lowest)

如果使用C语言编程，中断查询次序号就是中断号，例如：

```
void    Int0_Routine(void)          interrupt 0;
void    Timer0_Rountine(void)       interrupt 1;
void    Int1_Routine(void)          interrupt 2;
void    Timer1_Rountine(void)       interrupt 3;
void    UART_Rountine(void)         interrupt 4;
void    ADC_SPI_Routine(void)       interrupt 5;
void    PCA_LVD_Routine(void)       interrupt 6;
```

6.4 中断处理

当某中断产生而且被CPU响应，主程序被中断，接下来将执行如下操作：

1. 当前正被执行的指令全部执行完毕；
2. PC值被压入栈；
3. 现场保护；
4. 阻止同级别其他中断；
5. 将中断向量地址装载到程序计数器PC；
6. 执行相应的中断服务程序。

中断服务程序ISR完成和该中断相应的一些操作。ISR以RETI(中断返回)指令结束，将PC值从栈中取回，并恢复原来的中断设置，之后从主程序的断点处继续执行。

当某中断被响应时，被装载到程序计数器PC中的数值称为中断向量，是同该中断源相对应的中断服务程序的起始地址。各中断源服务程序的入口地址（即中断向量）为：

中断源	中断向量
External Interrupt 0	0003H
Timer 0	000BH
External Interrupt 1	0013H
Timer 1	001BH
UART	0023H
ADC/SPI	002BH
PCA/LVD	0033H

当“转去执行中断”时，引起中断的标志位将被硬件自动清零。由于中断向量入口地址位于程序存储器的开始部分，所以主程序的第1条指令通常为跳转指令，越过中断向量区(LJMP MAIN)。

注意:不能用RET指令代替RETI指令

RET指令虽然也能控制PC返回到原来中断的地方，但RET指令没有清零中断优先级状态触发器的功能，中断控制系统会认为中断仍在进行，其后果是与此同级或低级的中断请求将不被响应。

若用户在中断服务程序中进行了入栈操作，则在RETI指令执行前应进行相应的出栈操作，即在中断服务程序中PUSH指令与POP指令必须成对使用，否则不能正确返回断点。

6.5 外部中断

外部中断0(INT0)和外部中断1(INT1)触发有两种触发方式，下降沿触发方式和低电平触发方式。

TCON寄存器中的IT0/TCON.0和IT1/TCON.2决定了外部中断0和1是下降沿触发还是低电平触发。如果ITx = 0(x = 0,1)，那么系统在INTx(x = 0,1)脚探测到下降沿后可产生外部中断。如果ITx = 1(x = 0,1)，那么系统在INTx(x= 0,1)脚探测低电平后才可产生外部中断。外部中断0(INT0)和外部中断1(INT1)还可以用于将单片机从掉电模式唤醒。

由于系统每个时钟对外部中断引脚采样1次，所以为了确保被检测到，输入信号应该至少维持2个系统时钟。如果外部中断是仅下降沿触发，要求必须在相应的引脚维持高电平至少1个系统时钟，而且低电平也要持续至少一个系统时钟，才能确保该下降沿被CPU检测到。同样，如果外部中断是低电平可触发，则要求必须在相应的引脚维持低电平至少2个系统时钟，这样才能确保CPU能够检测到该低电平信号。

6.6 中断测试程序

6.6.1 外部中断0($\overline{\text{INT0}}$)的测试程序

1. 程序1——演示外部中断0的下降沿中断

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断0(下降沿) ------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"

//External interrupt0 service routine
void exint0() interrupt 0                    //INT0, interrupt 0 (location at 0003H)
{
}

void main()
{
        IT0 = 1;                             //set INT0 interrupt type (1:Falling 0:Low level)
        EX0 = 1;                             //enable INT0 interrupt
        EA = 1;                              //open global interrupt switch

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T系列单片机外部中断0(下降沿) -----------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------*/
/*---------------------------------------------------------------------------------------*/

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0003H                   ;INT0, interrupt 0 (location at 0003H)
        LJMP    EXINT0

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,#7FH                 ;initial SP
        SETB    IT0                     ;set INT0 interrupt type (1:Falling 0:Low level)
        SETB    EX0                     ;enable INT0 interrupt
        SETB    EA                      ;open global interrupt switch
        SJMP    $

;-----------------------------------------
;External interrupt0 service routine

EXINT0:
        RETI

;-----------------------------------------

        END
```

2. 程序2——演示外部中断0的下降沿中断唤醒掉电模式

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断0(下降沿)唤醒掉电模式 -----*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

//External interrupt0 service routine
void exint0( )      interrupt 0                //INT0, interrupt 0 (location at 0003H)
{
}

void main()
{
        IT0 = 1;                               //set INT0 interrupt type (1:Falling 0:Low level)
        EX0 = 1;                               //enable INT0 interrupt
        EA = 1;                                //open global interrupt switch

        while (1)
        {
                INT0 = 1;                      //ready read INT0 port
                while (!INT0);                 //check INT0
                _nop_();
                _nop_();
                PCON = 0x02;                   //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T系列单片机外部中断0(下降沿)唤醒掉电模式 -----*/
/* 如果要在程序中使用或在文章中引用该程序， ----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------*/
/*---------------------------------------------------------------------------------*/

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0003H                   ;INT0, interrupt 0 (location at 0003H)
        LJMP    EXINT0

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,     #7FH            ;initial SP
        SETB    IT0                     ;set INT0 interrupt type (1:Falling 0:Low level)
        SETB    EX0                     ;enable INT0 interrupt
        SETB    EA                      ;open global interrupt switch
LOOP:
        SETB    INT0                    ;ready read INT0 port
        JNB     INT0,   $               ;check INT0
        NOP
        NOP
        MOV     PCON,   #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;External interrupt0 service routine

EXINT0:
        RETI

;-----------------------------------------

        END
```

6.6.2 外部中断1($\overline{\text{INT1}}$)的测试程序

1. 程序1——演示外部中断1的下降沿中断

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断1(下降沿) -----------------------*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"

//External interrupt1 service routine
void exint1() interrupt 2                    //INT1, interrupt 2 (location at 0013H)
{
}

void main()
{
        IT1 = 1;                             //set INT1 interrupt type (1:Falling only 0:Low level)
        EX1 = 1;                             //enable INT1 interrupt
        EA = 1;                              //open global interrupt switch

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断1(下降沿) -----------------------*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/

;----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0013H                   ;INT1, interrupt 2 (location at 0013H)
        LJMP    EXINT1

;----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,     #7FH            ;initial SP
        SETB    IT1                     ;set INT1 interrupt type (1:Falling 0:Low level)
        SETB    EX1                     ;enable INT1 interrupt
        SETB    EA                      ;open global interrupt switch
        SJMP    $

;----------------------------------------
;External interrupt1 service routine

EXINT1:
        RETI

;----------------------------------------

        END
```

2. 程序2——演示外部中断1的下降沿中断唤醒掉电模式

C程序：

```
/*-------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断1(下降沿)唤醒掉电模式 -----*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*-------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

//External interrupt0 service routine
void exint1( ) interrupt 2                    //INT1, interrupt 2 (location at 0013H)
{
}

void main()
{
        IT1 = 1;                              //set INT1 interrupt type (1:Falling 0:Low level)
        EX1 = 1;                              //enable INT1 interrupt
        EA = 1;                               //open global interrupt switch

        while (1)
        {
                INT1 = 1;                     //ready read INT1 port
                while (!INT1);                //check INT1
                _nop_();
                _nop_();
                PCON = 0x02;                  //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机外部中断1(下降沿)唤醒掉电模式 -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     0013H           ;INT1, interrupt 2 (location at 0013H)
        LJMP    EXINT1

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,#7FH         ;initial SP
        SETB    IT1             ;set INT1 interrupt type (1:Falling 0:Low level)
        SETB    EX1             ;enable INT1 interrupt
        SETB    EA              ;open global interrupt switch
LOOP:
        SETB    INT1            ;ready read INT1 port
        JNB     INT1,$          ;check INT1
        NOP
        NOP
        MOV     PCON,#02H       ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;External interrupt1 service routine

EXINT1:
        RETI

;-----------------------------------------
        END
```

6.6.3 定时器0中断(下降沿中断，可用于唤醒掉电模式)的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0中断(下降沿),可将单片机从掉电模式唤醒 --*/
/* 如果要在程序中使用或在文章中引用该程序， -------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -------------------------------*/
/*------------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

sfr WAKE_CLKO = 0x8f;

//External interrupt0 service routine
void t0int( ) interrupt 1                    //T0 interrupt, interrupt 1 (location at 000BH)
{
}

void main()
{
        WAKE_CLKO = 0x10;                    //enable T0 falling edge wakeup MCU from power-down mode
        ET0 = 1;                             //enable T0 interrupt
        EA = 1;                              //open global interrupt switch

        while (1)
        {
                T0 = 1;                      //ready read T0 port
                while (!T0);                 //check T0
                _nop_();
                _nop_();
                PCON = 0x02;                 //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0中断(下降沿),可将单片机从掉电模式唤醒  --*/
/* 如果要在程序中使用或在文章中引用该程序， -------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -------------------------------*/
/*------------------------------------------------------------------------------------------------------------*/
WAKE_CLKO   EQU 8FH

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     000BH                   ;T0 interrupt, interrupt 1 (location at 000BH)
        LJMP    T0INT

;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,#7FH                 ;initial SP
        MOV     WAKE_CLKO,    #10H      ;enable T0 falling edge wakeup MCU from power-down mode
        SETB    ET0                     ;enable T0 interrupt
        SETB    EA                      ;open global interrupt switch
LOOP:
        SETB    T0                      ;ready read T0 port
        JNB     T0      ,$              ;check T0
        NOP
        NOP
        MOV    PCON,    #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;T0 interrupt service routine

T0INT:
        RETI
;-----------------------------------------
        END
```

6.6.4 定时器1中断(下降沿中断，可用于唤醒掉电模式)的测试程序

1. C程序:

```
/*---------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1中断(下降沿),可将单片机从掉电模式唤醒  --*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------------------------*/
/*---------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

sfr     WAKE_CLKO = 0x8f;

//External interrupt0 service routine
void t1int() interrupt 3                //T1 interrupt, interrupt 3 (location at 001BH)
{
}

void main()
{
        WAKE_CLKO = 0x20;               //enable T1 falling edge wakeup MCU from power-down mode
        ET1 = 1;                        //enable T1 interrupt
        EA = 1;                         //open global interrupt switch

        while (1)
        {
                T1 = 1;                 //ready read T1 port
                while (!T1);            //check T1
                _nop_();
                _nop_();
                PCON = 0x02;            //MCU power down
                _nop_();
                _nop_();
                P1++;
        }
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1中断(下降沿),可将单片机从掉电模式唤醒 --*/
/* 如果要在程序中使用或在文章中引用该程序， -------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------------------------*/
/*---------------------------------------------------------------------------------------------------------*/
WAKE_CLKO   EQU 8FH

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     001BH               ;T1 interrupt, interrupt 3 (location at 001BH)
        LJMP    T1INT

;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,         #7FH            ;initial SP
        MOV     WAKE_CLKO,      #20H        ;enable T1 falling edge wakeup MCU from power-down mode
        SETB    ET1                         ;enable T1 interrupt
        SETB    EA                          ;open global interrupt switch
LOOP:
        SETB    T1                          ;ready read T1 port
        JNB     T1,         $               ;check T1
        NOP
        NOP
        MOV     PCON,       #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP

;-----------------------------------------
;T1 interrupt service routine

T1INT:
        RETI

;-----------------------------------------
        END
```

6.6.5 RxD中断(RxD/P3.0下降沿中断，可用于唤醒掉电模式)的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机串行口RxD中断(下降沿),可将单片机从掉电模式唤醒  --*/
/* 如果要在程序中使用或在文章中引用该程序， ----------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------------------------------*/
/*------------------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the PCA */
sfr WAKE_CLKO   =   0x8F;

void uart_isr() interrupt 4 using 1
{
        if (RI)
        {
                RI = 0;
        }
}

void main()
{
        WAKE_CLKO = 0x40;       //enable RXD falling edge wakeup MCU from power-down mode
        ES = 1;
        EA = 1;

        while (1)
        {
                RXD = 1;                        //ready read RXD port
                while (!RXD);                   //check RXD
                _nop_();
                _nop_();
                PCON = 0x02;                    //MCU power down
                _nop_();
                _nop_();
                P2++;
        }
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机串行口RxD中断(下降沿),可将单片机从掉电模式唤醒 --*/
/* 如果要在程序中使用或在文章中引用该程序， ---------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------------------------------*/
/*------------------------------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
WAKE_CLKO       EQU         8FH

;-----------------------------------------
            ORG         0000H
            LJMP        MAIN

            ORG         0023H
UART_ISR:
            JBC         RI,         EXIT                        ;clear RI flag
EXIT:
            RETI
;-----------------------------------------
            ORG         0100H
MAIN:
            MOV         WAKE_CLKO,      #40H                    ;enable RXD falling edge wakeup MCU
                                                                ;from power-down mode
            SETB        ES
            SETB        EA

LOOP:
            SETB        RXD                                     ;ready read RXD port
            JNB         RXD,        $                           ;check RXD
            NOP
            NOP
            MOV         PCON,       #02H                        ;MCU power down
            NOP
            NOP
            CPL         P1.0
            SJMP        LOOP
;-----------------------------------------
            END
```

6.6.7 PCA模块中断(可用于唤醒掉电模式)的测试程序

1. C程序:

```
/*------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机PCA模块中断,可将单片机从掉电模式唤醒  ----------------*/
/* 如果要在程序中使用或在文章中引用该程序,  ---------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------------------------------*/
/*------------------------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the PCA */
sbit    EPCAI           =   IE^6;
sfr     WAKE_CLKO       =   0x8F;

sfr     CCON    =   0xD8;           //PCA control register
sbit    CCF0    =   CCON^0;         //PCA module-0 interrupt flag
sbit    CCF1    =   CCON^1;         //PCA module-1 interrupt flag
sbit    CR      =   CCON^6;         //PCA timer run control bit
sbit    CF      =   CCON^7;         //PCA timer overflow flag
sfr     CMOD    =   0xD9;           //PCA mode register
sfr     CL      =   0xE9;           //PCA base timer LOW
sfr     CH      =   0xF9;           //PCA base timer HIGH
sfr     CCAPM0= 0xDA;               //PCA module-0 mode register
sfr     CCAP0L =    0xEA;           //PCA module-0 capture register LOW
sfr     CCAP0H =    0xFA;           //PCA module-0 capture register HIGH
sfr     CCAPM1= 0xDB;               //PCA module-1 mode register
sfr     CCAP1L =    0xEB;           //PCA module-1 capture register LOW
sfr     CCAP1H =    0xFB;           //PCA module-1 capture register HIGH
sfr     CCAPM2= 0xDC;               //PCA module-2 mode register
sfr     CCAP2L =    0xEC;           //PCA module-2 capture register LOW
sfr     CCAP2H =    0xFC;           //PCA module-2 capture register HIGH
sfr     CCAPM3= 0xDD;               //PCA module-3 mode register
sfr     CCAP3L =    0xED;           //PCA module-3 capture register LOW
sfr     CCAP3H  =   0xFD;           //PCA module-3 capture register HIGH
sfr     PCAPWM0     =   0xF2;
sfr     PCAPWM1     =   0xF3;
sfr     PCAPWM2     =   0xF4;
sfr     PCAPWM3     =   0xF5;
```

```
sbit    PCA_LED   =  P1^0;                //PCA test LED
sbit    CCP0      =  P3^7;

void PCA_isr() interrupt 7 using 1
{
        CCF0 = 0;                         //Clear interrupt flag
        PCA_LED = !PCA_LED;               //toggle the test pin while CCP0(P3.7) have a falling edge
}

void main()
{
        CCON = 0;                   //Initial PCA control register
                                    //PCA timer stop running
                                    //Clear CF flag
                                    //Clear all module interrupt flag
        CL = 0;                     //Reset PCA base timer
        CH = 0;
        CMOD = 0x00;                //Set PCA timer clock source as Fosc/12
                                    //Disable PCA timer overflow interrupt
        CCAPM0 = 0x11;              //PCA module-0 capture by a negative tigger on CCP0(P3.7)
                                    //and enable PCA interrupt
//      CCAPM0 = 0x21;              //PCA module-0 capture by a rising edge on CCP0(P3.7)
                                    //and enable PCA interrupt
//      CCAPM0 = 0x31;              //PCA module-0 capture by a transition (falling/rising edge)
                                    //on CCP0(P3.7) and enable PCA interrupt

        WAKE_CLKO = 0x80;           //enable PCA falling/raising edge wakeup MCU from power-down mode
        CR = 1;                     //PCA timer start run
        EPCAI = 1;
        EA = 1;

        while (1)
        {
                CCP0 = 1;                 //ready read CCP0 port
                while (!CCP0);            //check CCP0
                _nop_();
                _nop_();
                PCON = 0x02;              //MCU power down
                _nop_();
                _nop_();
                P2++;
        }
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- --------------------------------------------------------------*/
/* --- 演示STC 1T 系列单片机PCA模块中断,可将单片机从掉电模式唤醒 ----------------*/
/* 如果要在程序中使用或在文章中引用该程序， ---------------------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ----------------------------------*/
/*---------------------------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
EPCAI           BIT     IE.6
WAKE_CLKO       EQU     8FH

CCON    EQU     0D8H                    ;PCA control register
CCF0    BIT     CCON.0                  ;PCA module-0 interrupt flag
CCF1    BIT     CCON.1                  ;PCA module-1 interrupt flag
CR      BIT     CCON.6                  ;PCA timer run control bit
CF      BIT     CCON.7                  ;PCA timer overflow flag
CMOD    EQU     0D9H                    ;PCA mode register
CL      EQU     0E9H                    ;PCA base timer LOW
CH      EQU     0F9H                    ;PCA base timer HIGH
CCAPM0          EQU     0DAH            ;PCA module-0 mode register
CCAP0L          EQU     0EAH            ;PCA module-0 capture register LOW
CCAP0H          EQU     0FAH            ;PCA module-0 capture register HIGH
CCAPM1          EQU     0DBH            ;PCA module-1 mode register
CCAP1L          EQU     0EBH            ;PCA module-1 capture register LOW
CCAP1H          EQU     0FBH            ;PCA module-1 capture register HIGH
CCAPM2          EQU     0DCH            ;PCA module-2 mode register
CCAP2L          EQU     0ECH            ;PCA module-2 capture register LOW
CCAP2H          EQU     0FCH            ;PCA module-2 capture register HIGH
CCAPM3          EQU     0DDH            ;PCA module-3 mode register
CCAP3L          EQU     0EDH            ;PCA module-3 capture register LOW
CCAP3H          EQU     0FDH            ;PCA module-3 capture register HIGH

PCA_LED         BIT     P1.1            ;PCA test LED
CCP0            BIT     P3.7
```

```
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN

        ORG     0033H
PCA_ISR:
        CLR     CCF0                    ;Clear interrupt flag
        CPL     PCA_LED                 ;toggle the test pin while CCP0(P3.7) have a falling edge
        RETI
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
        CLR     A                       ;
        MOV     CL,     A               ;Reset PCA base timer
        MOV     CH,     A               ;
        MOV     CMOD,   #00H            ;Set PCA timer clock source as Fosc/12
                                        ;Disable PCA timer overflow interrupt
        MOV     CCAPM0,         #11H    ;PCA module-0 capture by a falling edge on CCP0(P3.7)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,         #21H    ;PCA module-0 capture by a rising edge on CCP0(P3.7)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,         #31H    ;PCA module-0 capture by a transition (falling/rising edge)
                                        ;on CCP0(P3.7) and enable PCA interrupt
;------------------------------
        MOV     WAKE_CLKO,      #80H    ;enable PCA falling/raising edge wakeup MCU from
                                        ;power-down mode
        SETB    CR                      ;PCA timer start run
        SETB    EPCAI
        SETB    EA

LOOP:
        SETB    CCP0                    ;ready read CCP0 port
        JNB     CCP0,   $               ;check CCP0
        NOP
        NOP
        MOV     PCON,   #02H            ;MCU power down
        NOP
        NOP
        CPL     P1.0
        SJMP    LOOP
;-----------------------------------------
        END
```

第7章 定时器/计数器

STC12C5410AD系列单片机有6个定时器，其中定时器0和定时器1两个16位定时器，与传统8051的定时器完全兼容，也可以设置为1T模式，当在定时器1做波特率发生器时，定时器0可以当两个8位定时器用(另外4路PCA/PWM可以再实现4个16位定时器)。

STC12C5410AD系列单片机内部设置的两个16位定时器/计数器T0和T1都具有计数方式和定时方式两种工作方式。对每个定时器/计数器(T0和T1)，在特殊功能寄存器TMOD中都有一控制位— 来选择T0或T1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(也有减法)的计数器，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(T0为P3.4,T1为P3.5),则为计数方式，每来一个脉冲加1

当定时器/计数器工作在定时模式时，特殊功能寄存器AUXR中的T0x12和T1x12分别决定是系统时钟/12还是系统时钟/1(不分频)后让T0和T1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(13位定时器/计数器),模式1(16位定时器/计数器模式),模式2(8位自动重装模式)，模式3(两个8位定时器/计数器)。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。

7.1 定时器/计数器的相关寄存器

符号	描述	地址	位地址及其符号 MSB							LSB	复位值
TCON	定时器控制寄存器	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	定时器模式寄存器	89H	GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0	0000 0000B
TL0	Timer Low 0	8AH									0000 0000B
TL1	Timer Low 1	8BH									0000 0000B
TH0	Timer High 0	8CH									0000 0000B
TH1	Timer High 1	8DH									0000 0000B
AUXR	辅助寄存器	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	时钟输出和掉电唤醒寄存器	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

1. 定时器/计数器控制寄存器TCON

TCON为定时器/计数器T0、T1的控制寄存器，同时也锁存T0、T1溢出中断源和外部请求中断源等，TCON格式如下：

TCON：定时器/计数器中断控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
TCON	88H	name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1：定时器/计数器T1溢出标志。T1被允许计数以后，从初值开始加1计数。当最高位产生溢出时由硬件置“1”TF1，向CPU请求中断，一直保持到CPU响应中断时，才由硬件清“0”TF1（TF1也可由程序查询清“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$\overline{\text{INT1}}$输入高电平时，才允许T1计数。

TF0：定时器/计数器T0溢出中断标志。T0被允许计数以后，从初值开始加1计数，当最高位产生溢出时，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”TF0（ TF0也可由程序查询清“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，TR1=0且$\overline{\text{INT0}}$输入高电平时，才允许T0计数。

IE1：外部中断1请求源（$\overline{\text{INT1}}$/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1触发方式控制位。IT1=0时，外部中断1为低电平触发方式，当$\overline{\text{INT1}}$（P3.3）输入低电平时，置位IE1 。采用低电平触发方式时，外部中断源（输入到$\overline{\text{INT1}}$）必须保持低电平有效，直到该中断被CPU 响应，同时在该中断服务程序执行完之前，外部中断源必须被清除(P3.3要变高)，否则将产生另一次中断。当IT1=1时，则外部中断1($\overline{\text{INT1}}$)端口由“1”→“0”下降沿跳变，激活中断请求标志位IE1 ，向主机请求中断处理。

IE0：外部中断0请求源（$\overline{\text{INT0}}$/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0触发方式控制位。IT0=0时，外部中断0为低电平触发方式，当$\overline{\text{INT0}}$（P3.2）输入低电平时，置位IE0。采用低电平触发方式时，外部中断源（输入到$\overline{\text{INT0}}$）必须保持低电平有效，直到该中断被CPU响应，同时在该中断服务程序执行完之前，外部中断源必须被清除（P3.2要变高），否则将产生另一次中断。当IT0=1时，则外部中断0($\overline{\text{INT0}}$)端口由“1”→“0”下降沿跳变，激活中断请求标志位IE1 ，向主机请求中断处理。

2. 定时器/计数器工作模式寄存器TMOD

定时和计数功能由特殊功能寄存器TMOD的控制位$C/\overline{T}$进行选择，TMOD寄存器的各位信息如下表所列。可以看出，2个定时/计数器有4种操作模式，通过TMOD的M1和M0选择。2个定时/计数器的模式0、1和2都相同，模式3不同，各模式下的功能如下所述。

寄存器TMOD各位的功能描述

TMOD　　地址：89H　　　　复位值：00H

不可位寻址

7	6	5	4	3	2	1	0
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0
定时器1				定时器0			

位	符号	功能
TMOD.7/	GATE	TMOD.7控制定时器1,置1时只有在$\overline{INT1}$脚为高及TR1控制位置1时才可打开定时器/计数器1。
TMOD.3/	GATE	TMOD.3控制定时器0,置1时只有在$\overline{INT0}$脚为高及TR0控制位置1时才可打开定时器/计数器0。
TMOD.6/	$C/\overline{T}$	TMOD.6控制定时器1用作定时器或计数器，清零则用作定时器(从内部系统时钟输入)，置1用作计数器(从T1/P3.5脚输入)
TMOD.2/	$C/\overline{T}$	TMOD.2控制定时器0用作定时器或计数器，清零则用作定时器(从内部系统时钟输入)，置1用作计数器(从T0/P3.4脚输入)
TMOD.5/TMOD.4	M1、M0	定时器定时器/计数器1模式选择
	0　0	13位定时器/计数器，兼容8048定时模式，TL1只用低5位参与分频，TH1整个8位全用。
	0　1	16位定时器/计数器，TL1、TH1全用
	1　0	8位自动重装载定时器,当溢出时将TH1存放的值自动重装入TL1.
	1　1	定时器/计数器1此时无效（停止计数）。
TMOD.1/TMOD.0	M1、M0	定时器/计数器0模式选择
	0　0	13位定时器/计数器，兼容8048定时模式，TL0只用低5位参与分频，TH0整个8位全用。
	0　1	16位定时器/计数器，TL0、TH0全用
	1　0	8位自动重装载定时器，当溢出时将TH0存放的值自动重装入TL0
	1　1	定时器0此时作为双8位定时器/计数器。TL0作为一个8位定时器/计数器，通过标准定时器0的控制位控制。TH0仅作为一个8位定时器,由定时器1的控制位控制。

3. 辅助寄存器AUXR

STC12C5410AD系列单片机是 1T 的8051单片机，为兼容传统8051，定时器0和定时器1复位后是传统8051的速度，即12分频，这是为了兼容传统8051。但也可不进行12分频，通过设置新增加的特殊功能寄存器AUXR,将T0,T1设置为1T。普通111条机器指令是固定的，快3到24倍，无法改变。

AUXR格式如下：

AUXR：辅助寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

STC12C5410AD系列是 1T 的8051单片机，为了兼容传统8051，定时器0和定时器1复位后是传统8051的速度，即12分频，这是为了兼容传统8051。但也可不进行12分频，实现真正的1T。

T0x12：定时器0速度控制位。

0：定时器0速度是8051单片机定时器的速度，即12分频；

1：定时器0速度是8051单片机定时器速度的12倍，即不分频。

T1x12：定时器1速度控制位。

0：定时器1速度是8051单片机定时器的速度，即12分频；

1：定时器1速度是8051单片机定时器速度的12倍，即不分频。

如果UART串口用T1作为波特率发生器，则由T1x12位决定UART串口是12T还是1T。

STC12C5410AD系列是1T的8051单片机,为了兼容传统8051,UART串口复位后是兼容传统8051的。

UART_M0x6：串口模式0的通信速度设置位。

0：UART串口模式0的速度是传统8051单片机串口的速度，即12分频；

1：UART串口模式0的速度是传统8051单片机串口速度的6倍，即2分频。

如果用定时器T1做波特率发生器时,UART串口的速度由T1的溢出率决定

AUXR寄存器的其他位是与中断有关的，在此不作介绍。

4. WAKE_CLKO：时钟输出和掉电唤醒寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO

PCAWAKEUP：在掉电模式下，是否允许PCA上升沿/下降沿中断唤醒powerdown。
0：禁止PCA上升沿/下降沿中断唤醒powerdown；
1：允许PCA上升沿/下降沿中断唤醒powerdown。

RXD_PIN_IE：掉电模式下，允许P3.0(RXD)下降沿置RI，也能使RXD唤醒powerdown.
0：禁止P3.0(RXD)下降沿置RI，也禁止RXD唤醒powerdown；
1：允许P3.0(RXD)下降沿置RI，也允许RXD唤醒powerdown。

T1_PIN_IE：掉电模式下, 允许T1/P3.5脚下降沿置T1中断标志, 也能使T1脚唤醒powerdown.
0：禁止T1/P3.5脚下降沿置T1中断标志, 也禁止T1脚唤醒powerdown；
1：允许T1/P3.5脚下降沿置T1中断标志, 也允许T1脚唤醒powerdown。

T0_PIN_IE：掉电模式下, 允许T0/P3.4脚下降沿置T0中断标志, 也能使T0脚唤醒powerdown.
0：禁止T0/P3.4脚下降沿置T0中断标志, 也禁止T0脚唤醒powerdown；
1：允许T0/P3.4脚下降沿置T0中断标志, 也允许T0脚唤醒powerdown。

T1CLKO：是否允许将P3.5/T1脚配置为定时器T1的时钟输出CLKOUT1
1：允许将P3.5/T1脚配置为定时器T1的时钟输出CLKOUT1，此时定时器T1只能工作在模式2(8位自动重装模式), CLKOUT1输出时钟频率= T1溢出率/2
T1工作在1T模式时的输出频率 = SYSclk / (256 - TH1) / 2
T1工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH1) / 2
0：不允许将P3.5/T1脚配置为定时器T1的时钟输出CLKOUT1

T0CLKO：是否允许将P3.4/T0脚配置为定时器T0的时钟输出CLKOUT0
1：允许将P3.4/T0脚配置为定时器T0的时钟输出CLKOUT0，此时定时器T0只能工作在模式2(8位自动重装模式), CLKOUT0输出时钟频率 = T0溢出率 / 2
T0工作在1T 模式时的输出频率 = SYSclk / (256 - TH0) / 2
T0工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH0) / 2
0：不允许将P3.4/T0脚配置为定时器T0的时钟输出CLKOUT0

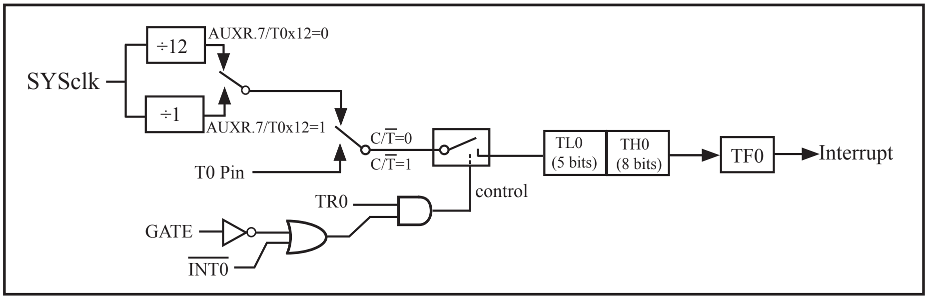
7.2 定时器/计数器0工作模式

通过对寄存器TMOD中的M1(TMOD.1)、M0(TMOD.0)的设置，定时器/计数器0有4种不同的工作模式

7.2.1 模式0(13位定时器/计数器)

将定时器设置成模式0时类似8048定时器，即8位计数器带32分频的预分频器。下图所示为定时器/计数器的模式0工作方式。此模式下，定时器0配置为13位的计数器，由TL0的低5位和TH0的8位所构成。TL0低5位溢出向TH0进位，TH0计数溢出置位TCON中的溢出标志位TF0。GATE(TMOD.3)=0 时，如TR0=1，则定时器计数。GATE=1 时，允许由外部输入$\overline{\text{INT1}}$控制定时器1，$\overline{\text{INT0}}$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TCON寄存器各位的具体功能描述表。

在模式0下定时器/计数器0作为13位定时器/计数器，如下图所示。



定时器/计数器0的模式 0: 13位定时器/计数器

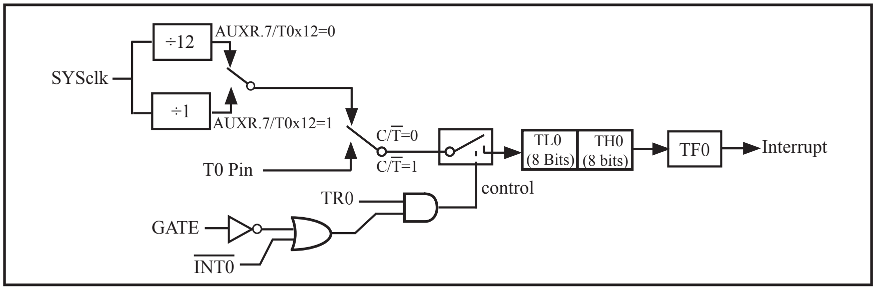
当C/$\overline{\text{T}}$=0时，多路开关连接到系统时钟的分频输出，T0对时钟周期计数，T0工作在定时方式。当C/$\overline{\text{T}}$=1时，多路开关连接到外部脉冲输入P3.4/T0，即T0工作在计数方式。

STC12C5410AD系列单片机的定时器有两种计数速率：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。

该模式下的13位寄存器包含TH0全部8个位及TL0的低5位。TL0的高3位不定，可将其忽略。置位运行标志（TR0）不能清零此寄存器。模式0的操作对于定时器0及定时器1都是相同的。2个不同的GATE位（TMOD.7和TMOD.3）分别分配给定时器1及定时器0。

7.2.2 模式1(16位定时器/计数器)

模式1除了使用了TH0及TL0全部16位外，其他与模式0完全相同。即此模式下定时器/计数器0作为16位定时器/计数器，如下图所示。



定时器/计数器0的模式 1: 16位定时器/计数器

此模式下，定时器配置为16位定时器/计数器，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(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/$\overline{T}$=0时，多路开关连接到系统时钟的分频输出，T0对时钟周期计数，T0工作在定时方式。当C/$\overline{T}$=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。

STC12C5410AD系列单片机的定时器有两种计数速率：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工作在16位定时器/计数器模式的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0的16位定时器/计数器模式 --*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/
#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

//-----------------------------------------------

/* define constants */
#define   FOSC    18432000L
#define   MODE    1T            //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE    1T
#define   T1MS    (65536-FOSC/1000)          //1ms timer calculation method in 1T mode
#else
#define   T1MS    (65536-FOSC/12/1000)       //1ms timer calculation method in 12T mode
#endif

/* define SFR */
sfr       AUXR       =    0x8e;              //Auxiliary register
sbit      TEST_LED   =    P0^0;              //work LED, flash once per second

/* define variables */
WORD      count;                             //1000 times counter

//-----------------------------------------------

/* Timer0 interrupt routine */
void tm0_isr() interrupt 1 using 1
{
          TL0 = T1MS;                        //reload timer0 low byte
          TH0 = T1MS >> 8;                   //reload timer0 high byte
          if (count-- == 0)                  //1ms * 1000 -> 1s
          {
                    count = 1000;            //reset counter
                    TEST_LED = ! TEST_LED;   //work LED flash
          }
}

//-----------------------------------------------
```

```
/* main program */
void main()
{
#ifdef   MODE   1T
         AUXR   = 0x80;                  //timer0 work in 1T mode
#endif
         TMOD   = 0x01;                  //set timer0 as mode1 (16-bit)
         TL0    = T1MS;                  //initial timer0 low byte
         TH0    = T1MS >> 8;             //initial timer0 high byte
         TR0    = 1;                     //timer0 start running
         ET0    = 1;                     //enable timer0 interrupt
         EA     = 1;                     //open global interrupt switch
         count  = 0;                     //initial counter

         while (1);                      //loop
}
```

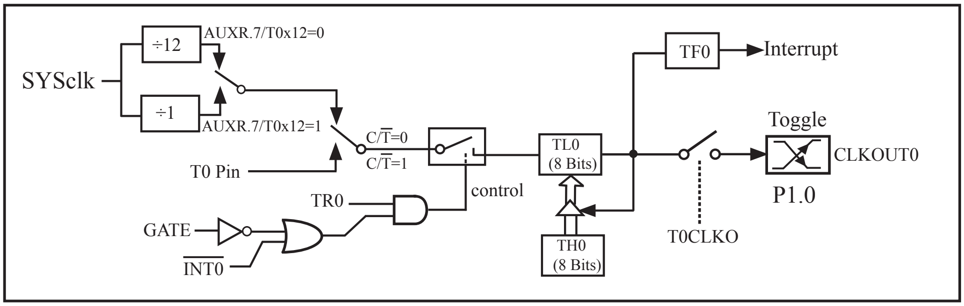
2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0的16位定时器/计数器模式 ---*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*-------------------------------------------------------------------------------------*/

;/* define constants */
#define  MODE1T                ;Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef   MODE1T
T1MS     EQU     0B800H        ;1ms timer calculation method in 1T mode is (65536-18432000/1000)
#else
T1MS     EQU     0FA00H        ;1ms timer calculation method in 12T mode is (65536-18432000/12/1000)
#endif

;/* define SFR */
AUXR             DATA    8EH           ;Auxiliary register
TEST_LED         BIT     P1.0          ;work LED, flash once per second

;/* define variables */
COUNT            DATA    20H           ;1000 times counter (2 bytes)

;-----------------------------------------------
```

```
        ORG     0000H
        LJMP    MAIN
        ORG     000BH
        LJMP    TM0_ISR

;-----------------------------------------------

;/* main program */
MAIN:
#ifdef  MODE    1T
        MOV     AUXR,   #80H                    ;timer0 work in 1T mode
#endif
        MOV     TMOD,   #01H                    ;set timer0 as mode1 (16-bit)
        MOV     TL0,    #LOW    T1MS            ;initial timer0 low byte
        MOV     TH0,    #HIGH   T1MS            ;initial timer0 high byte
        SETB    TR0                             ;timer0 start running
        SETB    ET0                             ;enable timer0 interrupt
        SETB    EA                              ;open global interrupt switch
        CLR     A
        MOV     COUNT,  A
        MOV     COUNT+1, A                      ;initial counter
        SJMP    $

;-----------------------------------------------

;/* Timer0 interrupt routine */
TM0_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     TL0,    #LOW    T1MS            ;reload timer0 low byte
        MOV     TH0,    #HIGH   T1MS            ;reload timer0 high byte
        MOV     A,      COUNT
        ORL     A,      COUNT+1                 ;check whether count(2byte) is equal to 0
        JNZ     SKIP
        MOV     COUNT, #LOW 1000                ;1ms * 1000 -> 1s
        MOV     COUNT+1,  #HIGH 1000
        CPL     TEST_LED                        ;work LED flash
SKIP:
        CLR     C
        MOV     A,      COUNT                   ;count--
        SUBB    A,      #1
        MOV     COUNT,  A
        MOV     A,      COUNT+1
        SUBB    A,      #0
        MOV     COUNT+1,  A
        POP     PSW
        POP     ACC
        RETI

;-----------------------------------------------

        END
```

7.2.3 模式2(8位自动重装模式)

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

TL0的溢出不仅置位TF0，而且将TH0内容重新装入TL0，TH0内容由软件预置，重装时TH0内容不变。

在此模式下，当T0CLKO/WAKE_CLKO.0=1时，P1.0/ADC0管脚配置为定时器0的时钟输出CLKOUT0。输出时钟频率 = T0 溢出率/2

如果C/$\overline{T}$=0，定时器/计数器T0对内部系统时钟计数，则：

T0工作在1T模式(AUXR.7/T0x12=1)时的输出时钟频率=(SYSclk) / (256-TH0)/2

T0工作在12T模式(AUXR.7/T0x12=0)时的输出时钟频率=(SYSclk)/12/(256-TH0)/2

如果C/$\overline{T}$=1，定时器/计数器T0是对外部脉冲输入(P1.0/ADC0)计数，则：

输出时钟频率 = (T0_Pin_CLK) / (256-TH0) / 2

;定时器0中断(下降沿中断)的测试程序，定时器0工作在8位自动重装模式
;下面程序中的定时器中断不能将单片机从掉电模式唤醒

1. C程序：

```
/*------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------*/
/* --- STC 1T Series MCU T0(Falling edge) Demo -------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*------------------------------------------------------------------------------*/

#include "reg51.h"

sfr         AUXR = 0x8e;                        //Auxiliary register

//T0 interrupt service routine
void t0int( ) interrupt 1                       //T0 interrupt (location at 000BH)
{
}

void main()
{
            AUXR = 0x80;                        //timer0 work in 1T mode
            TMOD = 0x06;                        //set timer0 as counter mode2 (8-bit auto-reload)
            TL0 = TH0 = 0xff;                   //fill with 0xff to count one time
            TR0 = 1;                            //timer0 start run
            ET0 = 1;                            //enable T0 interrupt
            EA = 1;                             //open global interrupt switch

            while (1);
}
```

2. 汇编程序：

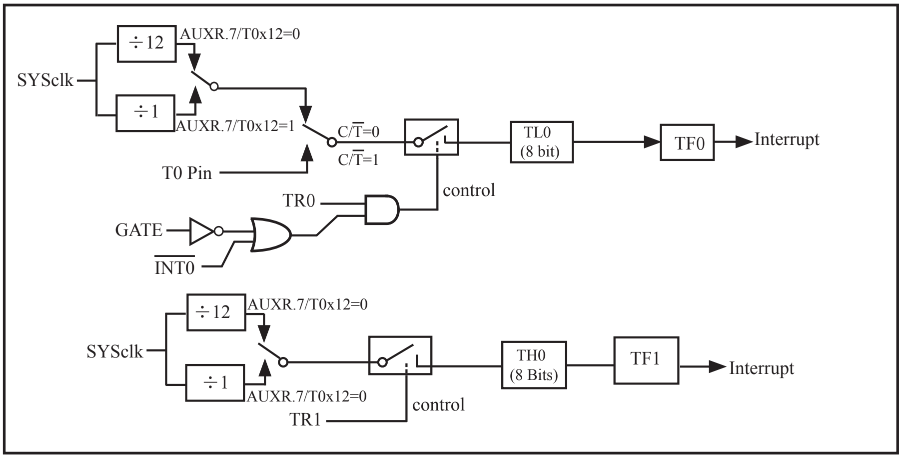
```
/*------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------*/
/* --- STC 1T Series MCU T0(Falling edge) Demo -------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*------------------------------------------------------------------------------*/

AUXR    DATA    08EH                            ;Auxiliary register

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     000BH                           ;T0 interrupt (location at 000BH)
        LJMP    T0INT

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,     #7FH                    ;initial SP
        MOV     AUXR,   #80H                    ;timer0 work in 1T mode
        MOV     TMOD,   #06H                    ;set timer0 as counter mode2 (8-bit auto-reload)
        MOV     A,      #0FFH
        MOV     TL0,    A                       ;fill with 0xff to count one time
        MOV     TH0,    A
        SETB    TR0                             ;timer0 start run
        SETB    ET0                             ;enable T0 interrupt
        SETB    EA                              ;open global interrupt switch
        SJMP    $

;-----------------------------------------
;T0 interrupt service routine

T0INT:
        RETI

;-----------------------------------------

        END
```

7.2.4 模式3(两个8位计数器)

对定时器1，在模式3时，定时器1停止计数，效果与将TR1设置为0相同。

对定时器0，此模式下定时器0的TL0及TH0作为2个独立的8位计数器。下图为模式3时的定时器0逻辑图。TL0占用定时器0的控制位：C/$\overline{T}$、GATE、TR0、INT0及TF0。TH0限定为定时器功能（计数器周期），占用定时器1的TR1及TF1。此时，TH0控制定时器1中断。

模式3是为了增加一个附加的8位定时器/计数器而提供的，使单片机具有三个定时器/计数器。模式3只适用于定时器/计数器0，定时器T1处于模式3时相当于TR1=0，停止计数，而T0可作为两个定时器用。



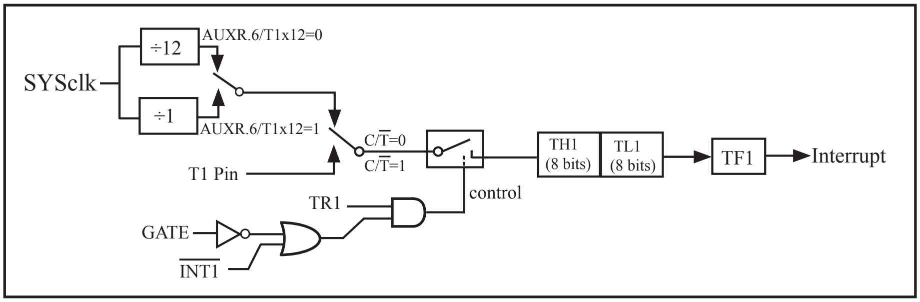
定时/计数器0 模式3: 两个8位计数器

7.3 定时器/计数器1工作模式

通过对寄存器TMOD中的M1(TMOD.5)、M0(TMOD.4)的设置，定时器/计数器1有3种不同的工作模式。

7.3.1 模式0(13位定时器/计数器)

此模式下定时器/计数器1作为13位定时器/计数器，有TL1的低5位和TH1的8位所构成，如下图所示。模式0的操作对于定时器1和定时器0是相同的。



定时器/计数器1的模式 0: 13位定时器/计数器

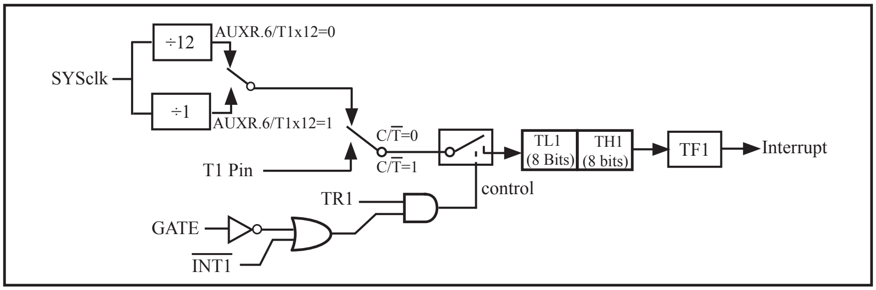
当GATE=0(TMOD.7)时，如TR1=1，则定时器计数。GATE=1时，允许由外部输入$\overline{\text{INT1}}$控制定时器1，这样可实现脉宽测量。TR1为TCON寄存器内的控制位，TCON寄存器各位的具体功能描述见上节TCON寄存器的介绍。

当C/$\overline{\text{T}}$=0时，多路开关连接到系统时钟的分频输出，T1对时钟周期计数，T1工作在定时方式。当C/$\overline{\text{T}}$=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。

STC12C5410AD系列单片机的定时器有两种计数速率：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。

7.3.2 模式1(16位定时器/计数器)

此模式下定时器/计数器1作为16位定时器/计数器，如下图所示。



定时器/计数器1的模式 1: 16位定时器/计数器

此模式下，定时器1配置为16位定时器/计数器，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(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/$\overline{T}$=0时，多路开关连接到系统时钟的分频输出，T1对时钟周期计数，T1工作在定时方式。当C/$\overline{T}$=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。

STC12C5410AD系列单片机的定时器有两种计数速率：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工作在16位定时器/计数器模式的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1的16位定时器/计数器模式 --*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/
#include "reg51.h"

typedef   unsigned  char        BYTE;
typedef   unsigned  int         WORD;

//-----------------------------------------------

/* define constants */
#define   FOSC      18432000L
#define   MODE      1T          //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE      1T
#define   T1MS      (65536-FOSC/1000)           //1ms timer calculation method in 1T mode
#else
#define   T1MS      (65536-FOSC/12/1000)        //1ms timer calculation method in 12T mode
#endif

/* define SFR */
sfr       AUXR       =     0x8e;                //Auxiliary register
sbit      TEST_LED   =     P0^0;                //work LED, flash once per second

/* define variables */
WORD      count;                                //1000 times counter

//-----------------------------------------------

/* Timer0 interrupt routine */
void tm1_isr() interrupt 3 using 1
{
          TL1 = T1MS;                           //reload timer1 low byte
          TH1 = T1MS >> 8;                      //reload timer1 high byte
          if (count-- == 0)                     //1ms * 1000 -> 1s
          {
                    count = 1000;               //reset counter
                    TEST_LED = ! TEST_LED;      //work LED flash
          }
}

//-----------------------------------------------
```

```
/* main program */
void main( )
{
#ifdef    MODE    1T
          AUXR    = 0x40;                 //timer1 work in 1T mode
#endif
          TMOD    = 0x10;                 //set timer1 as mode1 (16-bit)
          TL1     = T1MS;                 //initial timer1 low byte
          TH1     = T1MS >> 8;            //initial timer1 high byte
          TR1     = 1;                    //timer1 start running
          ET1      = 1;                   //enable timer1 interrupt
          EA      = 1;                    //open global interrupt switch
          count   = 0;                    //initial counter

          while (1);                      //loop
}
```

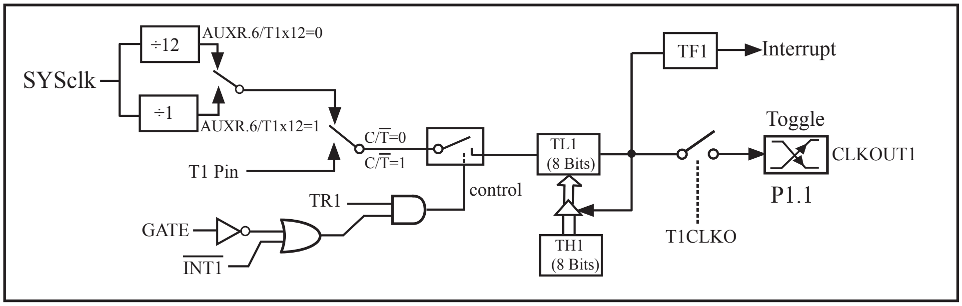
2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1的16位定时器/计数器模式 ---*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

;/* define constants */
#define   MODE    1T              ;Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE    1T
T1MS      EQU     0B800H           ;1ms timer calculation method in 1T mode is (65536-18432000/1000)
#else
T1MS      EQU     0FA00H           ;1ms timer calculation method in 12T mode is (65536-18432000/12/1000)
#endif

;/* define SFR */
AUXR        DATA        8EH             ;Auxiliary register
TEST_LED    BIT         P1.0            ;work LED, flash once per second

;/* define variables */
COUNT   DATA    20H                     ;1000 times counter (2 bytes)

;----------------------------------------------
```

```
        ORG     0000H
        LJMP    MAIN
        ORG     001BH
        LJMP    TM1_ISR

;-----------------------------------------------

;/* main program */
MAIN:
#ifdef MODE1T
        MOV      AUXR,  #40H                 ;timer1 work in 1T mode
#endif
        MOV     TMOD,   #10H                 ;set timer1 as mode1 (16-bit)
        MOV     TL1,    #LOW T1MS            ;initial timer1 low byte
        MOV     TH1,    #HIGH T1MS           ;initial timer1 high byte
        SETB    TR1                          ;timer1 start running
        SETB    ET1                          ;enable timer1 interrupt
        SETB    EA                           ;open global interrupt switch
        CLR     A
        MOV     COUNT,  A
        MOV     COUNT+1,A                    ;initial counter
        SJMP    $

;-----------------------------------------------

;/* Timer1 interrupt routine */
TM1_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     TL1,    #LOW T1MS            ;reload timer1 low byte
        MOV     TH1,    #HIGH T1MS           ;reload timer1 high byte
        MOV     A,      COUNT
        ORL     A,      COUNT+1              ;check whether count(2byte) is equal to 0
        JNZ     SKIP
        MOV     COUNT,  #LOW 1000            ;1ms * 1000 -> 1s
        MOV     COUNT+1,        #HIGH 1000
        CPL     TEST_LED                     ;work LED flash
SKIP:
        CLR     C
        MOV     A,      COUNT                ;count--
        SUBB    A,      #1
        MOV     COUNT,  A
        MOV     A,      COUNT+1
        SUBB    A,      #0
        MOV     COUNT+1,A
        POP     PSW
        POP     ACC
        RETI

;-----------------------------------------------

        END
```

7.3.3 模式2(8位自动重装模式)

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

TL1的溢出不仅置位TF1，而且将TH1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/WAKE_CLKO.1=1时，P1.1/ADC1管脚配置为定时1的时钟输出。

输出时钟频率 = T1 溢出率/2

如果C/$\overline{T}$=0，定时器/计数器T1对内部系统时钟计数，则

T1工作在1T模式(AUXR.6/T1x12=1)时的输出时钟频率=(SYSclk) / (256-TH1)/2

T1工作在12T模式(AUXR.6/T1x12=0)时的输出时钟频率=(SYSclk)/12/(256-TH1)/2

如果C/$\overline{T}$=1，定时器/计数器T1是对外部脉冲输入(P1.1/ADC1)计数，则：

输出时钟频率 = (T1_Pin_CLK) / (256-TH1) / 2

;定时器1中断(下降沿中断)的测试程序，定时器1工作在8位自动重装模式
;下面程序中的定时器中断不能将单片机从掉电模式唤醒

1. C程序：

```
/*------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------*/
/* --- STC 1T Series MCU T1(Falling edge) Demo -------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*------------------------------------------------------------------------------*/

#include "reg51.h"

sfr AUXR = 0x8e;                              //Auxiliary register

//T1 interrupt service routine
void t1int( ) interrupt 3                     //T1 interrupt (location at 001BH)
{
}

void main()
{
        AUXR = 0x40;                  //timer1 work in 1T mode
        TMOD = 0x60;                  //set timer1 as counter mode2 (8-bit auto-reload)
        TL1 = TH1 = 0xff;             //fill with 0xff to count one time
        TR1 = 1;                      //timer1 start run
        ET1 = 1;                      //enable T1 interrupt
        EA = 1;                       //open global interrupt switch

        while (1);
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ----------------------------------*/
/* --- STC 1T Series MCU T1(Falling edge) Demo -------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*----------------------------------------------------------------------------*/

AUXR    DATA    08EH                        ;Auxiliary register

;-----------------------------------------
;interrupt vector table

        ORG     0000H
        LJMP    MAIN

        ORG     001BH                       ;T1 interrupt (location at 001BH)
        LJMP    T1INT

;-----------------------------------------

        ORG     0100H
MAIN:
        MOV     SP,     #7FH                ;initial SP
        MOV     AUXR,   #40H                ;timer1 work in 1T mode
        MOV     TMOD,   #60H                ;set timer1 as counter mode2 (8-bit auto-reload)
        MOV     A,      #0FFH
        MOV     TL1,    A                   ;fill with 0xff to count one time
        MOV     TH1,    A
        SETB    TR1                         ;timer1 start run
        SETB    ET1                         ;enable T1 interrupt
        SETB    EA                          ;open global interrupt switch
        SJMP    $

;-----------------------------------------
;T1 interrupt service routine

T1INT:
        RETI

;-----------------------------------------

        END
```

7.4 可编程时钟输出及测试程序(C程序和汇编程序)

STC12C5410AD系列单片机有2路可编程时钟输出:CLKOUT0/ADC0/P1.0, CLKOUT1/ADC1/P1.1

与可编程时钟输出有关的特殊功能寄存器:
AUXR : Auxiliary register

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

WAKE_CLKO :Clock output and Power-down Wakeup Control register

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	LVD_WAKE	-	T1CLKO	T0CLKO

特殊功能寄存器AUXR/WAKE_CLKO/BRT的C语言声明:

```
sfr    AUXR        =   0x8E;      //特殊功能寄存器AUXR的地址声明
sfr    WAKE_CLKO   =   0x8F;      //新增加特殊功能寄存器WAKE_CLKO的地址声明
```

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的汇编语言声明:

```
AUXR         EQU    8EH        ;特殊功能寄存器AUXR的地址声明
WAKE_CLKO    EQU    8FH        ;新增加的特殊功能寄存器WAKE_CLKO的地址声明
```

如何利用CLKOUT0/P1.0和CLKOUT1/P1.1管脚输出时钟:

CLKOUT0/P1.0和CLKOUT1/P1.1的时钟输出控制由WAKE_CLKO寄存器的T0CLKO位和T1CLKO位控制。CLKOUT0的输出时钟频率由定时器0控制, CLKOUT1的输出时钟频率由定时器1控制， 相应的定时器需要工作在定时器的模式2方式(8位自动重装载模式), 不要允许相应的定时器中断, 免得CPU反复进中断.

新增加的特殊功能寄存器： WAKE_CLKO(地址：0x8F)

WAKE_CLKO :Clock output and Power-down Wakeup Control register(不可位寻址)

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO

B7 - PCAWAKEUP：在掉电模式下，是否允许PCA上升沿/下降沿中断唤醒powerdown。
0：禁止PCA上升沿/下降沿中断唤醒powerdown；
1：允许PCA上升沿/下降沿中断唤醒powerdown 。

B6 - RXD_PIN_IE：掉电模式下，允许P3.0(RXD)下降沿置RI，也能使RXD唤醒powerdown.
0：禁止P3.0(RXD)下降沿置RI，也禁止RXD唤醒powerdown；
1：允许P3.0(RXD)下降沿置RI，也允许RXD唤醒powerdown 。

B5 - T1_PIN_IE：掉电模式下, 允许T1/P3.5脚下降沿置T1中断标志, 也能使T1脚唤醒powerdown.
0：禁止T1/P3.5脚下降沿置T1中断标志, 也禁止T1脚唤醒powerdown；
1：允许T1/P3.5脚下降沿置T1中断标志, 也允许T1脚唤醒powerdown 。

B4 - T0_PIN_IE：掉电模式下,允许T0/P3.4脚下降沿置T0中断标志,也能使T0脚唤醒powerdown.
0：禁止T0/P3.4脚下降沿置T0中断标志,也禁止T0脚唤醒powerdown；
1：允许T0/P3.4脚下降沿置T0中断标志,也允许T0脚唤醒powerdown 。

B1 - T1CLKO： 是否允许将P1.1/ADC1脚配置为定时器T1的时钟输出CLKOUT1
1： 允许将P1.1脚配置为定时器T1的时钟输出CLKOUT1，此时定时器T1只能工作在模式2(8位自动重装模式),CLKOUT1输出时钟频率= T1溢出率/2
T1工作在1T模式时的输出频率 = SYSclk / (256 - TH1) / 2
T1工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH1) / 2
0：不允许将P1.1脚配置为定时器T1的时钟输出CLKOUT1

B0 - T0CLKO：是否允许将P1.0/ADC0脚配置为定时器T0的时钟输出CLKOUT0
1：允许将P1.0脚配置为定时器T0的时钟输出CLKOUT0，此时定时器T0只能工作在模式2(8位自动重装模式),CLKOUT0输出时钟频率 = T0溢出率 / 2
T0工作在1T 模式时的输出频率 = SYSclk / (256 - TH0) / 2
T0工作在12T模式时的输出频率 = SYSclk / 12 / (256 - TH0) / 2
0：不允许将P1.0脚配置为定时器T0的时钟输出CLKOUT0

特殊功能寄存器：AUXR(地址：0x8E)

AUXR : Auxiliary register(不可位寻址)

SFR Name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

B7 - T0x12：定时器0速度控制位。
0：定时器0速度是8051单片机定时器的速度，即12分频；
1：定时器0速度是8051单片机定时器速度的12倍，即不分频。

B6 - T1x12：定时器1速度控制位。
0：定时器1速度是8051单片机定时器的速度，即12分频；
1：定时器1速度是8051单片机定时器速度的12倍，即不分频。

如果UART串口用T1作为波特率发生器，则由T1x12位决定UART串口是12T还是1T。

B5 - UART_M0x6：串口模式0的通信速度设置位。
0：UART串口模式0的速度是传统8051单片机串口的速度，即12分频；
1：UART串口模式0的速度是传统8051单片机串口速度的6倍，即2分频。

7.4.1 定时器0的可编程时钟输出的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0的可编程时钟输-----------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"

//-----------------------------------------------

/* define constants */
#define   FOSC      18432000L
//#define MODE      1T              //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE      1T
#define   F38_4KHz          (256-FOSC/2/38400)          //38.4KHz frequency calculation method of 1T mode
#else
#define   F38_4KHz          (256-FOSC/2/12/38400)       //38.4KHz frequency calculation method of 12T mode
#endif

/* define SFR */
sfr       AUXR              = 0x8e;                     //Auxiliary register
sfr       WAKE_CLKO         = 0x8f;                     //wakeup and clock output control register
sbit      T0CLKO            = P1^0;                     //timer0 clock output pin

//-----------------------------------------------

/* main program */
void main()
{
#ifdef    MODE      1T
          AUXR      =       0x80;                       //timer0 work in 1T mode
#endif
          TMOD      =       0x02;                       //set timer0 as mode2 (8-bit auto-reload)
          TL0       =       F38_4KHz;                   //initial timer0
          TH0       =       F38_4KHz;                   //initial timer0
          TR0       =       1;                          //timer0 start running
          WAKE_CLKO =       0x01;                       //enable timer0 clock output

          while (1);                                    //loop
}
```

2. 汇编程序：

```
/*-------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器0的可编程时钟输-----------------*/
/* 如果要在程序中使用或在文章中引用该程序，  -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*-------------------------------------------------------------------------------------*/

;/* define constants */
#define   MODE   1T              ;Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE   1T
F38_4KHz  EQU    010H            ;38.4KHz frequency calculation method of 1T mode is (256-18432000/2/38400)
#else
F38_4KHz  EQU    0ECH            ;38.4KHz frequency calculation method of 12T mode (256-18432000/2/12/38400)
#endif

;/* define SFR */
AUXR            DATA    08EH                ;Auxiliary register
WAKE_CLKO       DATA    08FH                ;wakeup and clock output control register
T0CLKO          BIT     P1.0                ;timer0 clock output pin

;---------------------------------------------

        ORG     0000H
        LJMP    MAIN

;---------------------------------------------

;/* main program */
MAIN:
#ifdef MODE1T
        MOV     AUXR,   #80H                ;timer0 work in 1T mode
#endif
        MOV     TMOD,   #02H                ;set timer0 as mode2 (8-bit auto-reload)
        MOV     TL0,    #F38_4KHz           ;initial timer0
        MOV     TH0,    #F38_4KHz           ;initial timer0
        SETB    TR0
        MOV     WAKE_CLKO, #01H             ;enable timer0 clock output

        SJMP    $

;---------------------------------------------

        END
```

7.4.2 定时器1的可编程时钟输出的测试程序

1. C程序：

```
/*----------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1的可编程时钟输------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*----------------------------------------------------------------------------------*/

#include "reg51.h"

//-----------------------------------------------

/* define constants */
#define    FOSC      18432000L
//#define  MODE    1T                //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef     MODE    1T
#define    F38_4KHz          (256-FOSC/2/38400)          //38.4KHz frequency calculation method of 1T mode
#else
#define    F38_4KHz          (256-FOSC/2/12/38400)       //38.4KHz frequency calculation method of 12T mode
#endif

/* define SFR */
sfr        AUXR              = 0x8e;                     //Auxiliary register
sfr        WAKE_CLKO         = 0x8f;                     //wakeup and clock output control register
sbit       T1CLKO            = P1^1;                     //timer1 clock output pin

//-----------------------------------------------

/* main program */
void main()
{
#ifdef     MODE    1T
           AUXR    =  0x40;                              //timer1 work in 1T mode
#endif
           TMOD    =  0x20;                              //set timer1 as mode2 (8-bit auto-reload)
           TL1     =  F38_4KHz;                          //initial timer1
           TH1     =  F38_4KHz;                          //initial timer1
           TR1     =  1;                                 //timer1 start running
           WAKE_CLKO = 0x02;                             //enable timer1 clock output

           while (1);                                    //loop
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机定时器1的可编程时钟输-----------------*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

;/* define constants */
#define   MODE   1T          ;Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef    MODE   1T
F38_4KHz  EQU    010H        ;38.4KHz frequency calculation method of 1T mode is (256-18432000/2/38400)
#else
F38_4KHz  EQU    0ECH        ;38.4KHz frequency calculation method of 12T mode (256-18432000/2/12/38400)
#endif

;/* define SFR */
AUXR          DATA    08EH          ;Auxiliary register
WAKE_CLKO     DATA    08FH          ;wakeup and clock output control register
T1CLKO        BIT     P1.1          ;timer1 clock output pin

;-----------------------------------------------

          ORG     0000H
          LJMP    MAIN

;-----------------------------------------------

;/* main program */
MAIN:
#ifdef    MODE    1T
          MOV     AUXR,    #40H             ;timer1 work in 1T mode
#endif
          MOV     TMOD,    #20H             ;set timer1 as mode2 (8-bit auto-reload)
          MOV     TL1,     #F38_4KHz        ;initial timer1
          MOV     TH1,     #F38_4KHz        ;initial timer1
          SETB    TR1
          MOV     WAKE_CLKO, #02H           ;enable timer1 clock output

          SJMP    $

;-----------------------------------------------

          END
```

7.5 古老Intel 8051单片机定时器0/1的应用举例

【例1】 定时/计数器编程，定时/计数器的应用编程主要需考虑：根据应用要求，通过程序初始化，正确设置控制字，正确计算和计算计数初值，编写中断服务程序，适时设置控制位等。通常情况下，设置顺序大致如下：

1)工作方式控制字（TMOD、T2CON）的设置；
2)计数初值的计算并装入THx、TLx、RCAP2H、RCAP2L；
3)中断允许位ETx、EA的设置，使主机开放中断；
4) 启/停位TRx的设置等。

现以定时/计数器0或1为例作一简要介绍。

8051系列单片机的定时器/计数器0或1是以不断加1进行计数的，即属加1计数器，因此，就不能直接将实际的计数值作为计数初值送入计数寄存器THx、TLx中去，而必须将实际计数值以2^8、2^{13}、2^{16}为模求补，以其补码作为计数初值设置THx和TLx。

设：实际计数值为X，计数器长度为n（n=8、13、16），则应装入计数器THx、TLx中的计数初值为2^n-x，式中2^n为取模值。例如，工作方式0的计数长度为13位，则n=13，以2^{13}为模，工作方式1的计数长度为16，则n=16，以2^{16}为模等等。所以，计数初值为（x）= 2^n-x。

对于定时模式，是对机器周期计数，而机器周期与选定的主频密切相关。因此，需根据应用系统所选定的主频计算出机器周期值。现以主频6MHz为例，则机器周期为：

$$一个机器周期=\frac{12}{主振频率}=\frac{12}{6\times10^6}\mu s = 2\mu s$$

实际定时时间Tc = x·Tp

式中Tp为机器周期，Tc为所需定时时间，x为所需计数次数。Tp和Tp一般为已知值，在求出Tp后即可求得所需计数值x，再将x求补码，即求得定时计数初值。即

（x）补= 2^n - x

例如，设定时时间Tc = 5ms，机器周期TP = 2μs，可求得定时计数次数

$$x = \frac{5ms}{2\mu s} = 2500次$$

设选用工作方式1，则n=16，则应设置的定时时间计数初值为：

（x）补= 2^{16} - x=65536-2500=63036，还需将它分解成两个8位十六进制数，分别求得低8位为3CH装入TLx，高8位为F6H装入THx中。

工作方式0、1、2的最大计数次数分别为8192、65536和256。

对外部事件计数模式，只需根据实际计数次数求补后变换成两个十六进制码即可。

【例2】 定时/计数器应用编程，设某应用系统，选择定时/计数器1定时模式，定时时间Tc = 10ms，主频频率为12MHz，每10ms向主机请求处理。选定工作方式1。计算得计数初值：低8位初值为F0H，高8位初值为D8H。

（1）初始化程序

所谓初始化，一般在主程序中根据应用要求对定时/计数器进行功能选择及参数设定等预置程序，本例初始化程序如下：

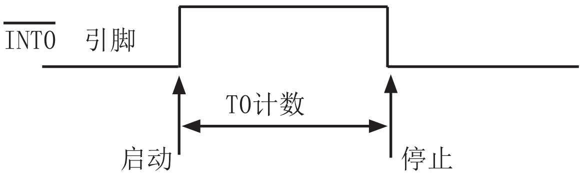
```
START:
          ⋮                         ；主程序段
        MOV    SP，#60H             ；设置堆栈区域
        MOV    TMOD，#10H           ；选择T1、定时模式，工作方式1
        MOV    TH1，#0D8H           ；设置高字节计数初值
        MOV    TL1，#0F0H           ；设置低字节计数初值
        SETB   EA                   ；} 开中断
        SETB   ET1                  ；}
          ⋮                         ；其他初始化程序
        SETB   TR1                  ；启动T1开始计时
          ⋮                         ；    继续主程序
```

（2）中断服务程序

```
INTT1： PUSH   A                    ；}
        PUSH DPL                    ；} 现场保护
        PUSH DPH                    ；}
          ⋮
        MOV    TL1,#0F0H            ；} 重新置初值
        MOV    TH1,#0D8H            ；}
          ⋮                         ；中断处理主体程序
        POP    DPH                  ；}
        POP    DPL                  ；} 现场恢复
        POP    A                    ；}
        RETI                        ；返回
```

这里展示了中断服务子程序的基本格式。STC12C5410AD系列单片机的中断属于矢量中断，每一个矢量中断源只留有8个字节单元，一般是不够用的，常需用转移指令转到真正的中断服务子程序区去执行。

【例3】 对外部正脉冲测宽。选择定时/计数器2进行脉宽测试较方便，但也可选用定时/计数器0或定时/计数器1进行测宽操作。本例选用定时/计数器0（T0）以定时模式，工作方式1对INT0引脚上的正脉冲进行脉宽测试。



设置GATE为1，机器周期TP为1μs。本例程序段编制如下：

```
INTT0:    MOV    TMOD，#09H      ；设T0为定时方式1，GATE为1
          MOV    TL0，#00H       ；
          MOV    TH0，#00H       ；TH0,TL0清0
          CLR    EX0             ；关INT0中断
LOP1:     JB     P3.2，LOP1      ；等待INT0引低电平
LOP2:     JNB    P3.2，LOP2      ；等待INT0引脚高电平
          SETB   TR0             ；启动T0开始计数
LOP3:     JB     P3.2，LOP3      ；等待INT0低电平
          CLR    TR0             ；停止T0计数
          MOV    A，TL0          ；低字节计数值送A
          MOV    B，TH0          ；高字节计数值送B
           ⋮                     ；计算脉宽和处理
```

【例4】 利用定时/计数器0或定时/计数器1的Tx端口改造成外部中断源输入端口的应用设计。

在某些应用系统中常会出现原有的两个外部中断源INT0和INT1不够用，而定时/计数器有多余，则可将Tx用于增加的外部中断源。现选择定时/计数器1为对外部事件计数模式工作方式2（自动再装入），设置计数初值为FFH，则T1端口输入一个负跳变脉冲，计数器即回0溢出，置位对应的中断请求标志位TF1为1，向主机请求中断处理，从而达到了增加一个外部中断源的目的。应用定时/计数器1（T1）的中断矢量转入中断服务程序处理。其程序示例如下：

（1）主程序段：

```
        ORG     0000H
        AJMP    MAIN                    ；转主程序
        ORG     001BH
        LJMP    INTER                   ；转T1中断服务程序
         ⋮
        ORG     0100                    ；主程序入口
MAIN：  …
         ⋮
        MOV     SP，#60H                ；设置堆栈区
        MOV     TMOD，#60H      ；设置定时/计数器1，计数方式2
        MOV     TL1，#0FFH      ；设置计数常数
        MOV     TH1，#0FFH
        SETB    EA                      ；开中断
        SETB    ET1                     ；开定时/计数器1中断
        SETB    TR1                     ；启动定时/计数器1计数
         ⋮
```

（2）中断服务程序（具体处理程序略）

```
            ORG     1000H
INTER:      PUSH    A           ;
            PUSH    DPL         ;   现场入栈保护
            PUSH    DPH         ;
            ⋮
            ⋮                   ;
            ⋮                   ;   中断处理主体程序
            POP     DPH         ;
            POP     DPL         ;   现场出栈复原
            POP     A           ;
            RETI                ; 返回
```

这是中断服务程序的基本格式。

【例5】 某应用系统需通过P1.0和P1.1分别输出周期为200μs和400μs的方波。为此，系统选用定时器/计数器0（T0），定时方式3，主频为6MHz，TP=2μs，经计算得定时常数为9CH和38H。

本例程序段编制如下：

（1）初始化程序段

```
            ⋮
PLT0:       MOV     TMOD,#03H   ; 设置T0定时方式3
            MOV     TL0，#9CH   ; 设置TL0初值
            MOV     TH0，#38H   ; 设置TH0初值
            SETB    EA          ;
            SETB    ET0         ;   开中断
            SETB    ET1         ;
            SETB    TR0         ; 启动
            SETB    TR1         ; 启动
            ⋮
```

(2)中断服务程序段

1)

```
INT0P：  ⋮
         ⋮
        MOV    TL0，#9CH              ；重新设置初值
        CPL    P1.0                   ；对P1.0输出信号取反
         ⋮
        RETI                          ；返回
```

2)

```
INT1P   ⋮
        ⋮
        MOV    TH0，#38H              ；重新设置初值
        CPL    P1.1                   ；对P1.1输出信号取反
        ⋮
        RETI                          ；返回
```

在实际应用中应注意的问题如下。

（1）定时/计数器的实时性

定时/计数器启动计数后，当计满回0溢出向主机请求中断处理，由内部硬件自动进行。但从回0溢出请求中断到主机响应中断并作出处理存在时间延迟，且这种延时随中断请求时的现场环境的不同而不同，一般需延时3个机器周期以上，这就给实时处理带来误差。大多数应用场合可忽略不计，但对某些要求实时性苛刻的场合，应采用补偿措施。

这种由中断响应引起的时间延时，对定时/计数器工作于方式0或1而言有两种含义：一是由于中断响应延时而引起的实时处理的误差；二是如需多次且连续不间断地定时/计数，由于中断响应延时，则在中断服务程序中再置计数初值时已延误了若干个计数值而引起误差，特别是用于定时就更明显。

例如选用定时方式1设置系统时钟，由于上述原因就会产生实时误差。这种场合应采用动态补偿办法以减少系统始终误差。所谓动态补偿，即在中断服务程序中对THx、TLx重新置计数初值时，应将THx、TLx从回0溢出又重新从0开始继续计数的值读出，并补偿到原计数初值中去进行重新设置。可考虑如下补偿方法：

```
      ⋮
CLR     EA                  ；禁止中断
MOV     A，TLx              ；读TLx中已计数值
ADD     A，#LOW             ；LOW为原低字节计数初值
MOV     TLx，A              ；设置低字节计数初值
MOV     A，#HIGH            ；原高字节计数初值送A
ADDC    A，THx              ；高字节计数初值补偿
MOV     THx，A              ；置高字节计数初值
SETB    EA                  ；开中断
      ⋮
```

（2）动态读取运行中的计数值

在动态读取运行中的定时/计数器的计数值时，如果不加注意，就可能出错。这是因为不可能在同一时刻同时读取THx和TLx中的计数值。比如，先读TLx后读THx，因为定时/计数器处于运行状态，在读TLx时尚未产生向THx进位，而在读THx前已产生进位，这时读得的THx就不对了；同样，先读THx后读TLx也可能出错。

一种可避免读错的方法是：先读THx，后读TLx，将两次读得的THx进行比较；若两次读得的值相等，则可确定读的值是正确的，否则重复上述过程，重复读得的值一般不会再错。此法的软件编程如下：

```
RDTM：  MOV  A，THx                 ；读取THx存A中
        MOV  R0，TLx                ；读取TLx存R0中
        CJNE A，THx，RDTM           ；比较两次THx值,若相等,则读得的
                                    ；值正确,程序往下执行,否则重读
        MOV     R1，A               ；将THx存于R1中
      ⋮
```

第8章 串行口通信

STC12C5410AD系列单片机具有1个采用UART(Universal Asychronous Receiver/Transmitter)工作方式的全双工串行通信接口。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。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。发送缓冲器只能写入而不能读出，接收缓冲器只能读出而不能写入，因而两个缓冲器可以共用一个地址码。串行口的两个缓冲器统称为串行通信特殊功能寄存器SBUF，其共用的地址码是99H。

STC12C5410AD系列单片机的两个串行口都有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。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。STC-12C5410AD系列单片机串行口对应的硬件部分是TxD/P3.1和RxD/P3.0引脚。

STC12C5410AD系列单片机的串行通信口，除用于数据通信外，还可方便地构成一个或多个并行I/O口，或作串—并转换，或用于扩展串行外设等。

8.1 串行口的相关寄存器

符号	描述	地址	位地址及符号 MSB							LSB	复位值
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
SCON	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000 0000B
SBUF	Serial Buffer	99H									xxxx xxxxB
PCON	Power Control	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_LVD	PS	PT1	PX1	PT0	PX0	x000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_LVDH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
SADEN	Slave Address Mask	B9H									0000 0000B
SADDR	Slave Address	A9H									0000 0000B
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RxD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

1. 串行口的控制寄存器SCON和PCON

STC12C5410AD系列单片机的串行口设有两个控制寄存器：串行控制寄存器SCON和波特率选择特殊功能寄存器PCON。

串行控制寄存器SCON用于选择串行通信的工作方式和某些控制功能。其格式如下：

SCON：串行控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SCON	98H	name	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI

SM0/FE：当PCON寄存器中的SMOD0/PCON.6位为1时，该位用于帧错误检测。当检测到一个无效停止位时，通过UART接收器设置该位。它必须由软件清零。

当PCON寄存器中的SMOD0/PCON.6位为0时，该位和SM1一起指定串行通信的工作方式，如下表所示。

其中SM0、SM1按下列组合确定串行口1的工作方式：

SM0	SM1	工作方式	功能说明	波特率
0	0	方式0	同步移位串行方式：移位寄存器	当UART_M0x6 = 0时，波特率是SYSclk/12, 当UART_M0x6 = 1时，波特率是SYSclk / 2
0	1	方式1	8位UART，波特率可变	$(2^{SMOD}/32)\times$(定时器1的溢出率)
1	0	方式2	9位UART	$(2^{SMOD}/64)$ x SYSclk系统工作时钟频率
1	1	方式3	9位UART，波特率可变	$(2^{SMOD}/32)$x(定时器1的溢出率)
当T1x12 = 0时，定时器1的溢出率 = SYSclk/12/(256 - T H1)； 当T1x12 = 1时, 定时器1的溢出率 = SYSclk / (256 - T H1)				

SM2：允许方式2或方式3多机通信控制位。

在方式2或方式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。在方式2或方式3中，如果SM2位为0且REN位为1，接收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.

方式1和方式0是非多机通信方式，在这两种方式时，要设置SM2 应为0。

REN：允许/禁止串行接收控制位。由软件置位REN，即REN=1为允许串行接收状态，可启动串行接收器RxD，开始接收信息。软件复位REN，即REN=0，则禁止接收。

TB8：在方式2或方式3，它为要发送的第9位数据，按需要由软件置位或清0。例如，可用作数据的校验位或多机通信中表示地址帧/数据帧的标志位。在方式0和方式1中，该位不用.

RB8：在方式2或方式3，是接收到的第9位数据，作为奇偶校验位或地址帧/数据帧的标志位。方式0中不用RB8(置SM2=0). 方式1中也不用RB8(置SM2=0, RB8是接收到的停止位)。

TI: 发送中断请求中断标志位。在方式0，当串行发送数据第8位结束时，由内部硬件自动置位，即TI=1，向主机请求中断，响应中断后TI必须用软件清零，即TI=0。在其他方式中，则在停止位开始发送时由内部硬件置位，即TI=1，响应中断后TI必须用软件清零。

RI: 接收中断请求标志位。在方式0，当串行接收到第8位结束时由内部硬件自动置位RI=1，向主机请求中断，响应中断后RI必须用软件清零，即RI=0。在其他方式中，串行接收到停止位的中间时刻由内部硬件置位，即RI=1，向CPU发中断申请，响应中断后RI必须由软件清零。

SCON的所有位可通过整机复位信号复位为全“0”。SCON的字节地址为98H，可位寻址，各位地址为98H~~9FH，可用软件实现位设置。

串行通信的中断请求：当一帧发送完成，内部硬件自动置位TI，即TI=1，请求中断处理；当接收完一帧信息时，内部硬件自动置位RI，即RI=1，请求中断处理。由于TI和RI以“或逻辑”关系向主机请求中断，所以主机响应中断时事先并不知道是TI还是RI请求的中断，必须在中断服务程序中查询TI和RI进行判别，然后分别处理。因此，两个中断请求标志位均不能由硬件自动置位，必须通过软件清0，否则将出现一次请求多次响应的错误。

电源控制寄存器PCON中的SMOD/PCON. 7用于设置方式1、方式2、方式3的波特率是否加倍。

电源控制寄存器PCON格式如下：

PCON：电源控制寄存器（不可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

SMOD：波特率选择位。当用软件置位SMOD，即SMOD=1，则使串行通信方式1、2、3的波特率加倍；SMOD=0，则各工作方式的波特率加倍。复位时SMOD=0。

SMOD0：帧错误检测有效控制位。当SMOD0=1，SCON寄存器中的SM0/FE位用于FE(帧错误检测)功能；当SMOD0=0，SCON寄存器中的SM0/FE位用于SM0功能，和SM1一起指定串行口的工作方式。复位时SMOD0=0

2. 串行口数据缓冲寄存器SBUF

STC12C5410AD系列单片机的串行口缓冲寄存器(SBUF)的地址是99H，实际是2个缓冲器，写SBUF的操作完成待发送数据的加载，读SBUF的操作可获得已接收到的数据。两个操作分别对应两个不同的寄存器，1个是只写寄存器，1个是只读寄存器。

串行通道内设有数据寄存器。在所有的串行通信方式中，在写入SBUF信号(MOV SBUF,A)的控制下，把数据装入相同的9位移位寄存器，前面8位为数据字节，其最低位为移位寄存器的输出位。根据不同的工作方式会自动将“1”或TB8的值装入移位寄存器的第9位,并进行发送.

串行通道的接收寄存器是一个输入移位寄存器。在方式0时它的字长为8位，其他方式时为9位。当一帧接收完毕，移位寄存器中的数据字节装入串行数据缓冲器SBUF中,其第9位则装入SCON寄存器中的RB8位。如果由于SM2使得已接收到的数据无效时,RB8和SBUF中内容不变.

由于接收通道内设有输入移位寄存器和SBUF缓冲器，从而能使一帧接收完将数据由移位寄存器装入SBUF后，可立即开始接收下一帧信息，主机应在该帧接收结束前从SBUF缓冲器中将数据取走，否则前一帧数据将丢失。SBUF以并行方式送往内部数据总线。

3. 辅助寄存器AUXR

辅助寄存器AUXR的格式及各位含义如下：

AUXR：辅助寄存器 （不可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-

T0x12：定时器0速度设置位

0, 定时器0是传统8051速度，12分频;

1, 定时器0的速度是传统8051的12倍，不分频

T1x12: 定时器1速度设置位

0, 定时器1是传统8051速度，12分频;

1, 定时器1的速度是传统8051的12倍，不分频

如果UART串口用定时器1做波特率发生器，T1x12位就可以控制UART串口是12T还是1T了。

UART_M0x6：串行口模式0的通信速度设置位

0，UART串口的模式0是传统12T的8051速度，12分频;

1，UART串口的模式0的速度是传统12T的8051的6倍，2分频

寄存器AUXR中的其他位是与中断有关的，在此不作介绍。

5. 从机地址控制寄存器SADEN和SADDR

为了方便多机通信，STC12C5410AD系列单片机设置了从机地址控制寄存器SADEN和SADDR。其中SADEN是从机地址掩模寄存器(地址为B9H，复位值为00H)，SADDR是从机地址寄存器(地址为A9H，复位值为00H)。

6. 与串行口中断相关的寄存器IE、IP和IPH

串行口中断允许位ES位于中断允许寄存器IE中，中断允许寄存器的格式如下：

IE：中断允许寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA：CPU的总中断允许控制位，EA=1，CPU开放中断，EA=0，CPU屏蔽所有的中断申请。
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。

ES：串行口中断允许位，ES=1，允许串行口中断，ES=0，禁止串行口中断。

IPH：中断优先级控制寄存器高（不可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IPH	B7H	name	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H

IP：中断优先级控制寄存器低（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IP	B8H	name	-	PPCA_LVD	PADC	PS	PT1	PX1	PT0	PX0

PSH, PS:　串行口中断优先级控制位。

当PSH=0且PS=0时，串行口中断为最低优先级中断(优先级0)
当PSH=0且PS=1时，串行口中断为较低优先级中断(优先级1)
当PSH=1且PS=0时，串行口中断为较高优先级中断(优先级2)
当PSH=1且PS=1时，串行口中断为最高优先级中断(优先级3)

8.2 串行口工作模式

STC12C5410AD系列单片机的串行通信接口有4种工作模式，可通过软件编程对SCON中的SM0、SM1的设置进行选择。其中模式1、模式2和模式3为异步通信，每个发送和接收的字符都带有1个启动位和1个停止位。在模式0中，串行口被作为1个简单的移位寄存器使用。

8.2.1 串行口工作模式0：同步移位寄存器

在模式0状态，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/AUXR.5 = 0时，其波特率固定为SYSclk/12。当串行口模式0的通信速度设置位UART_M0x6/AUXR.5 = 1时，其波特率固定为SYSclk/2。串行口数据由RxD/P3.0端输入，同步移位脉冲（SHIFTCLOCK）由TxD/P3.1输出，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/AUXR.5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（SHIFTCLOCK）。波形如图8-1中“发送”所示。

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接收过程：模式0接收时，复位接收中断请求标志RI，即RI＝0，置位允许接收控制位REN=1时启动串行模式0接收过程。启动接收过程后，RxD为串行输入端，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/AUXR.5确定是12分频还是2分频)。其时序图如图8-1中“接收”所示。

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。

串行口工作模式0的示意图如图8-1所示

由示意图中可见，由TX和RX控制单元分别产生中断请求信号并置位TI=1或RI =1，经“或门“送主机请求中断，所以主机响应中断后必须软件判别是TI还是RI请求中断，必须软件清0中断请求标志位TI或RI。

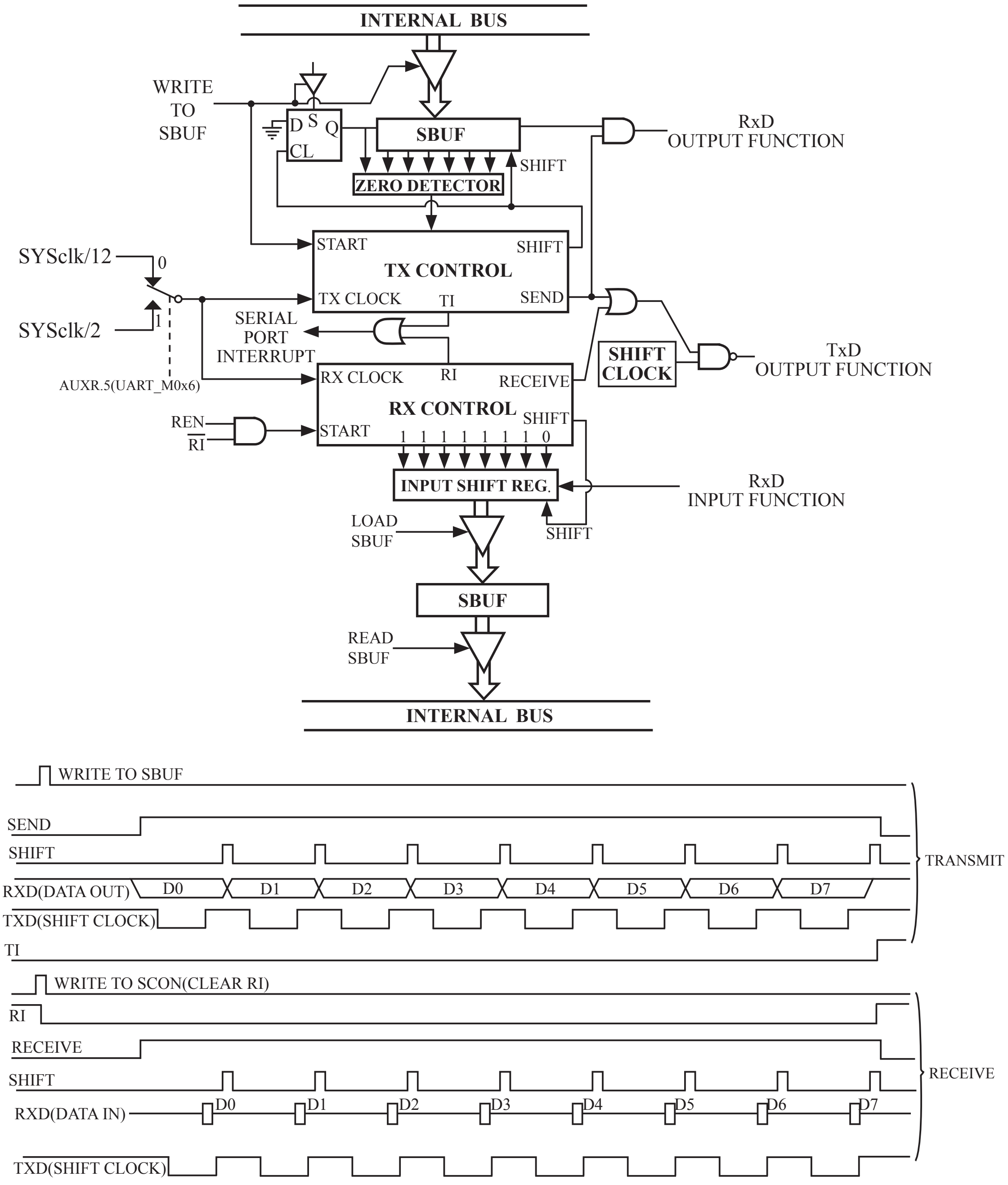


图8-1 串行口1模式0功能结构及时序示意图

8.2.2 串行口工作模式1：8位UART，波特率可变

当软件设置SCON的SM0、SM1为“01”时，串行口则以模式1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。TxD/P3.1为发送信息，RxD/P3.0为接收端接收信息，串行口为全双工接受/发送串行口。

图8-2为串行模式1的功能结构示意图及接收/发送时序图

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。发送各位的定时是由16分频计数器同步。

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，接收器便以选定波特率的16分频的速率采样串行接收端口RxD，当检测到RxD端口从“1”→“0”的负跳变时就启动接收器准备接收数据，并立即复位16分频计数器，将1FFH植装入移位寄存器。复位16分频计数器是使它与输入位时间同步。

16分频计数器的16个状态是将1波特率（每位接收时间）均为16等份，在每位时间的7、8、9状态由检测器对RxD端口进行采样，所接收的值是这次采样直经“三中取二”的值，即3次采样至少2次相同的值，以此消除干扰影响，提高可靠性。在起始位，如果接收到的值不为“0”（低电平），则起始位无效，复位接收电路，并重新检测″1″→″0″的跳变。如果接收到的起始位有效，则将它输入移位寄存器，并接收本帧的其余信息。

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，置位RI，即RI=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，必须由软件清0，即RI=0。通常情况下，串行通信工作于模式1时，SM2设置为″0″。

串行通信模式1的波特率是可变的，可变的波特由定时器/计数器1产生。

串行通信模式1的波特率=2^{SMOD}/32×(定时器/计数器1溢出率)

当T1x12 = 0时，定时器1的溢出率 = SYSclk/12/(256 - TH1)；

当T1x12 = 1时, 定时器1的溢出率 = SYSclk / (256 - TH1)

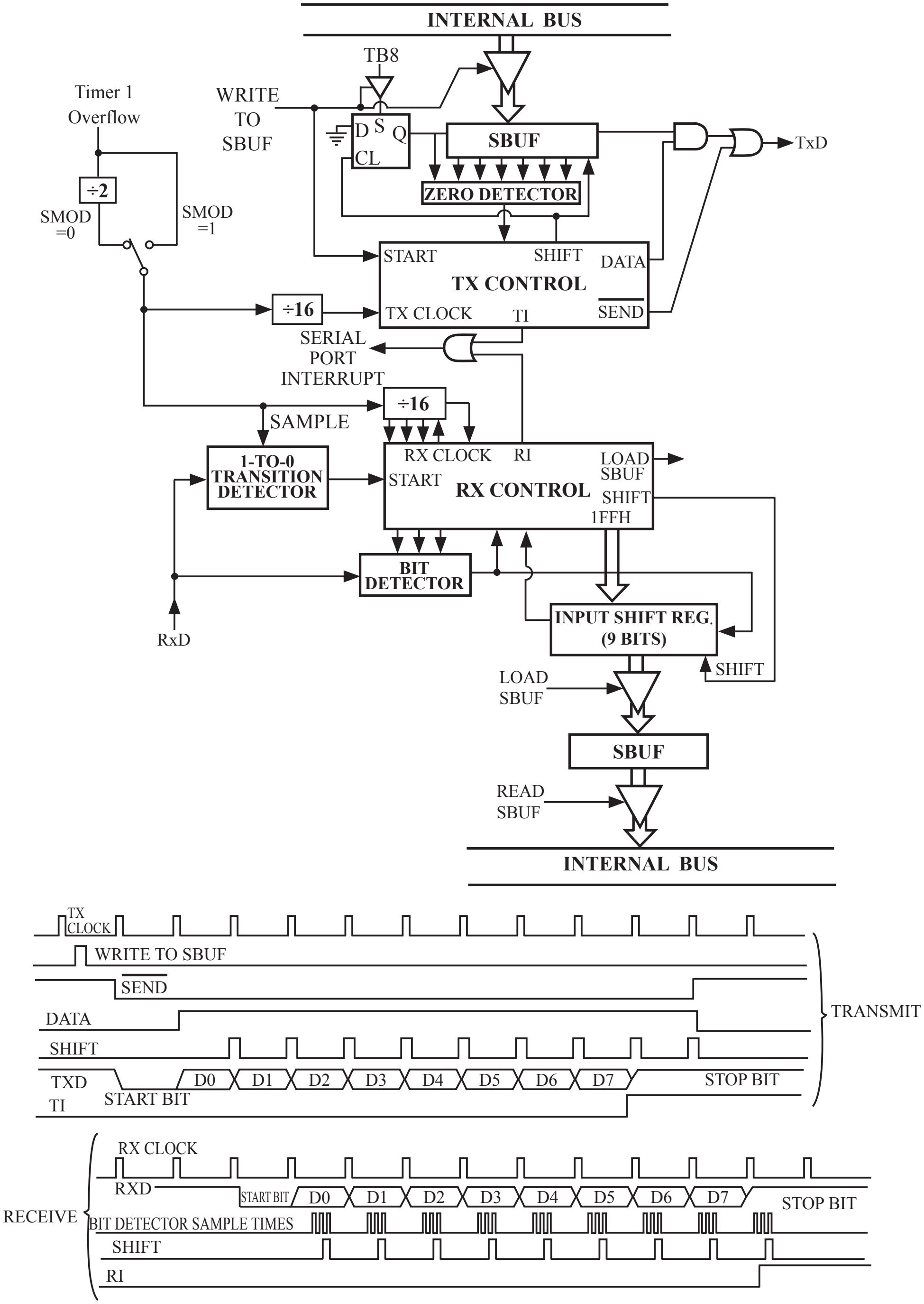


图8-2　串行口模式1功能结构示意图及接收/发送时序图

8.2.3 串行口工作模式2：9位UART，波特率固定

当SM0、SM1两位为10时，串行口工作在模式2。串行口工作模式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/P3.1为发送端口，RxD/P3.0为接收端口，以全双工模式进行接收/发送。

模式2的波特率为：

串行通信模式2波特率=2^{SMOD}/64×(SYSclk系统工作时钟频率)

上述波特率可通过软件对PCON中的SMOD位进行设置，当SMOD=1时，选择1/32(SYSclk)；当SMOD=0时，选择1/64(SYSclk)，故而称SMOD为波特率加倍位。可见，模式2的波特率基本上是固定的。

图8-3为串行通信模式2的功能结构示意图及其接收/发送时序图。

由图8-3可知，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，并且接收到的第9数据位RB8=1。

当上述两条件同时满足时，才将接收到的移位寄存器的数据装入SBUF和RB8中，并置位RI=1，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。

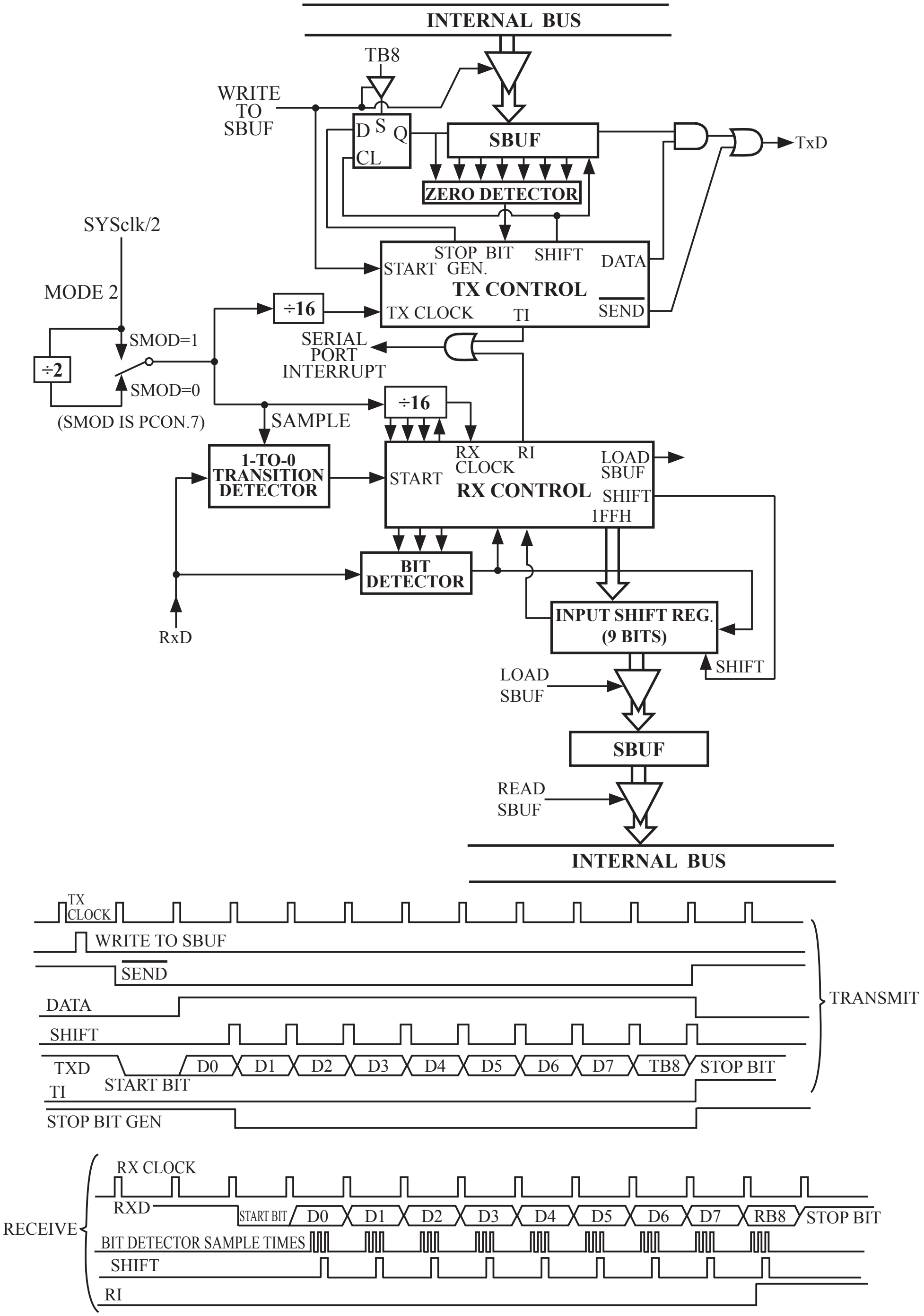


图8-3　串行口模式2功能结构示意图及接收/发送时序图

8.2.4 串行口工作模式3：9位UART，波特率可变

当SM0、SM1两位为11时，串行口工作在模式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/P3.1为发送端口，RxD/P3.0为接收端口，以全双工模式进行接收/发送。

模式3的波特率为：

串行通信模式3波特率=2^{SMOD}/32×(定时器/计数器1的溢出率)

当T1x12 = 0时，定时器1的溢出率 = SYSclk/12/(256 - TH1);

当T1x12 = 1时, 定时器1的溢出率 = SYSclk / (256 - TH1)

可见，模式3和模式1一样，其波特率可通过软件对定时器/计数器1或独立波特率发生器的设置进行波特率的选择，是可变的。

图8-4为串行口工作模式3的功能结构示意图及其接收/发送时序图。

由图8-4可知，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，并且接收到的第9数据位RB8=1。

当上述两条件同时满足时，才将接收到的移位寄存器的数据装入SBUF和RB8中，并置位RI=1，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。

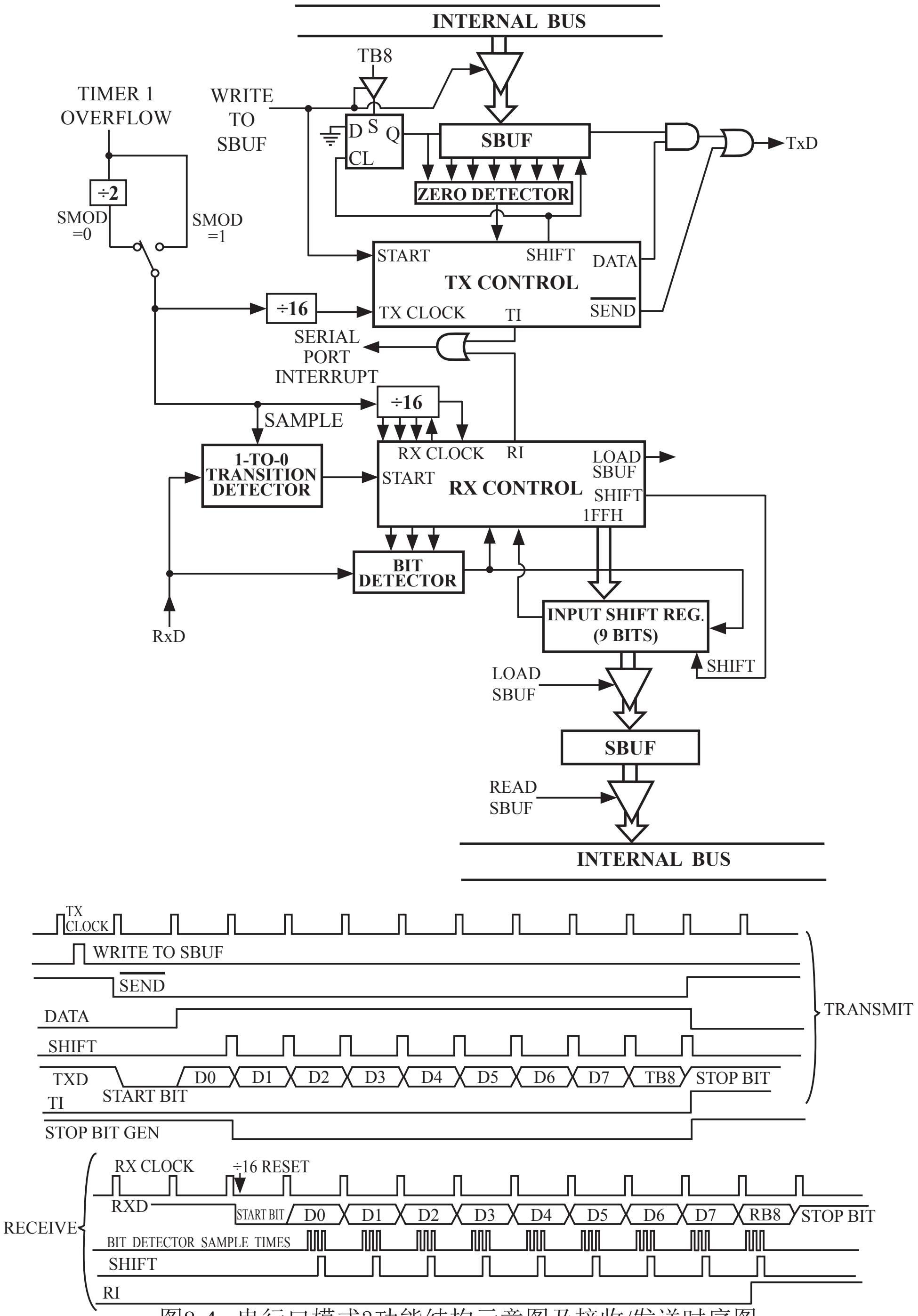


图8-4 串行口模式3功能结构示意图及接收/发送时序图

8.3 串行通信中波特率的设置

STC12C5410AD系列单片机串行通信的波特率随所选工作模式的不同而异，对于工作模式0和模式2，其波特率与系统时钟频率SYSclk和PCON中的波特率选择位SMOD有关，而模式1和模式3的波特率除与SYSclk和PCON位有关外，还与定时器/计数器1或BRT独立波特率发生器设置有关。通过对定时器/计数器1或BRT独立波特率发生器的设置，可选择不同的波特率，所以这种波特率是可变的。

串行通信模式0，其波特率与系统时钟频率SYSclk有关 。

当模式0的通信速度设置位UART_M0x6/AUXR.5 = 0时，其波特率 = SYSclk/12。

当模式0的通信速度设置位UART_M0x6/AUXR.5 = 1时，其波特率 = SYSclk/2。

一旦SYSclk选定且UART_M0x6/AUXR.5设置好，则串行通信工作模式0的波特率固定不变。

串行通信工作模式2，其波特率除与SYSclk有关外，还与SMOD位有关。

其基本表达式为：串行通信模式2波特率=$2^{SMOD}/64\times$(SYSclk系统工作时钟频率)

当SMOD=1时，波特率=2/64(SYSclk)=1/32(SYSclk)；

当SMOD=0时，波特率=1/64(SYSclk)。

当SYSclk选定后，通过软件设置PCON中的SMOD位，可选择两种波特率。所以，这种模式的波特率基本固定。

串行通信模式1和3，其波特率是可变的：

模式1、3波特率=$2^{SMOD}/32\times$(定时器/计数器1的溢出率)

当T1x12 = 0时，定时器1的溢出率 = SYSclk/12/(256 - TH1)；

当T1x12 = 1时, 定时器1的溢出率 = SYSclk / (256 - TH1)

通过对定时器/计数器1的设置，可灵活地选择不同的波特率。在实际应用中多半选用串行模式1或串行模式3。显然，为选择波特率，关键在于定时器/计数器1的溢出率的计算。SMOD的选择，只需根据需要执行下列指令就可实现SMOD=0或1；

```
MOV    PCON，#00H          ；使SMOD=0
MOV    PCON，#80H          ；使SMOD=1
```

SMOD只占用电源控制寄存器PCON的最高一位，其他各位的具体设置应根据实际情况而定。

为选择波特率，关键在于定时器/计数器1的溢出率。下面介绍如何计算定时器/计数器1的溢出率。

定时器/计数器1的溢出率定义为：单位时间（秒）内定时器/计数器1回0溢出的次数，即定时器/计数器1的溢出率=定时器/计数器1的溢出次数/秒。

STC12C5201AD系列单片机设有两个定时器/计数器，因定时器/计数器1具有4种工作方式，而常选用定时器/计数器1的工作方式2（8位自动重装）作为波特率的溢出率。

设置定时器/计数器1工作于定时模式的工作方式2（8位自动重装），TL1的计数输入来自于SYSclk经12分频或不分频(由T1x12/AUXR.6确定是12分频还是不分频)的脉冲。当T1x12/AUXR.6=0时，单片机工作在12T模式，TL1的计数输入来自于SYSclk经12分频的脉冲；当T1x12/AUXR.6=1时，单片机工作在1T模式，TL1的计数输入来自于SYSclk不经过分频的脉冲。可见，定时器/计数器1的溢出率与SYSclk和自动重装值N有关，SYSclk越大，特别是N越大，溢出率也就越高。例如：当N=FFN，则每隔一个时钟即溢出一次（极限情况）；若N=00H，则需每隔256个时钟才溢出一次；当SYSclk=6MHz且T1x12/AUXR.6=0时，一个时钟为2μs，当SYSclk=6MHz且T1x12/AUXR.6=1时，一个时钟约为0.167μs(快12倍)。SYSclk=12MHz且T1x12/AUXR.6=0时，则一个时钟为1μs，当SYSclk=6MHz且T1x12/AUXR.6=1时，一个时钟约为0.083μs(快12倍)。对于一般情况下，

当T1x12/AUXR.6=0时，定时器/计数器1溢出一次所需的时间为：$(2^8—N)\times12$时钟$=(2^8—N)\times12\times\frac{1}{SYSclk}$

当T1x12/AUXR.6=1时，定时器/计数器1溢出一次所需的时间为：$(2^8—N)\times1$时钟$=(2^8—N)\times\frac{1}{SYSclk}$

于是得定时器/计数器每秒溢出的次数，即

当T1x12/AUXR.6=0时，定时器/计数器1的溢出率$=SYSclk/12\times(2^8-N)$ (次/秒)

当T1x12/AUXR.6=1时，定时器/计数器1的溢出率$=SYSclk\times(2^8-N)$ (次/秒)

式中SYSclk为系统时钟频率，N为再装入时间常数。

下面现以定时器/计数器1工作于方式2为例，

设： T1x12/AUXR.6=0, SYSclk=6MHz, N=FFH,

定时器/计数器1工作于方式2的溢出率为$6\times10^6/\{12\times(256-255)\}=0.5\times10^6$(次/秒)；

设： T1x12/AUXR.6=0, SYSclk=12MHz, N=FFH,

定时器/计数器1工作于方式2的溢出率 = 1×10^6（次/秒）；

设： T1x12/AUXR.6=0, SYSclk=12MHz, N=00H,

定时器/计数器1工作于方式2的溢出率 = $12\times10^6/12\times256\approx3906$（次/秒）

设： T1x12/AUXR.6=1, SYSclk=6MHz, N=FFH,

定时器/计数器1工作于方式2的溢出率为$6\times10^6/(256-255)=6\times10^6$(次/秒)；

设： T1x12/AUXR.6=1, SYSclk=12MHz, N=00H,

定时器/计数器1工作于方式2的溢出率 = $12\times10^6/256=46875$（次/秒）

下表给出各种常用波特率与定时器/计数器1各参数之间的关系。

常用波特率与定时器/计数器1各参数关系（T1x12/AUXR.6=0）

常用波特率		系统时钟频率 (MHz)	SMOD	定时器1		
				$C/\overline{T}$	方式	重新装入值
方式0 MAX：1M		12	×	×	×	×
方式2 MAX：375K		12	1	×	×	×
方式1和3	62.5K	12	1	0	2	FFH
	19.2K	11.059	1	0	2	FDH
	9.6K	11.059	0	0	2	FDH
	4.8K	11.059	0	0	2	FAH
	2.4K	11.059	0	0	2	F4H
	1.2K	11.059	0	0	2	F8H
	137.5	11.986	0	0	2	1DH
	110	6	0	0	2	72H
	110	12	0	0	1	FFFBH

设置波特率的初始化程序段如下：

```
        ⋮
MOV   TMOD，  #20H        ；设置定时器/计数器1定时、工作方式2
MOV   TH1，   #××H        ； 设置定时常数N
MOV   TL1，   #××H        ；
SETB  TR1                 ；启动定时器/计数器1
MOV   PCON，  #80H        ；设置SMOD=1
MOV   SCON，  #50H        ；设置串行通信方式1
        ⋮
```

执行上述程序段后，即可完成对定时器/计数器1的操作方式及串行通信的工作方式和波特率的设置。

由于用其他工作方式设置波特率计算方法较复杂，一般应用较少，故不一一论述。

8.4 串行口的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机串行口功能 (8-bit/9-bit) -----------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef   unsigned  char      BYTE;
typedef   unsigned  int       WORD;

#define   FOSC     18432000L                    //System frequency
#define   BAUD     9600                         //UART baudrate

/*Define UART parity mode*/
#define   NONE_PARITY    0                      //None parity
#define   ODD_PARITY     1                      //Odd parity
#define   EVEN_PARITY    2                      //Even parity
#define   MARK_PARITY    3                      //Mark parity
#define   SPACE_PARITY   4                      //Space parity

#define   PARITYBIT      EVEN_PARITY            //Testing even parity

sbit      bit9 = P2^2;                          //P2.2 show UART data bit9
bit       busy;

void SendData(BYTE dat);
void SendString(char *s);

void main()
{
#if (PARITYBIT == NONE_PARITY)
          SCON = 0x50;                          //8-bit variable UART
#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
          SCON = 0xda;                          //9-bit variable UART, parity bit initial to 1
#elif (PARITYBIT == SPACE_PARITY)
          SCON = 0xd2;                          //9-bit variable UART, parity bit initial to 0
#endif
```

```
            TMOD    = 0x20;                          //Set Timer1 as 8-bit auto reload mode
            TH1     = TL1 = -(FOSC/12/32/BAUD);      //Set auto-reload vaule
            TR1     = 1;                             //Timer1 start run
            ES      = 1;                             //Enable UART interrupt
            EA      = 1;                             //Open master interrupt switch

            SendString("STC12C5410AD\r\nUart Test !\r\n");
            while(1);
}

/*----------------------------
UART interrupt service routine
----------------------------*/
void Uart_Isr() interrupt 4 using 1
{
            if (RI)
            {
                        RI = 0;                      //Clear receive interrupt flag
                        P0 = SBUF;                   //P0 show UART data
                        bit9 = RB8;                  //P2.2 show parity bit
            }
            if (TI)
            {
                        TI = 0;                      //Clear transmit interrupt flag
                        busy = 0;                    //Clear transmit busy flag
            }
}
/*----------------------------
Send a byte data to UART
Input: dat (data to be sent)
Output:None
----------------------------*/
void SendData(BYTE dat)
{
            while (busy);                            //Wait for the completion of the previous data is sent
            ACC = dat;                               //Calculate the even parity bit P (PSW.0)
            if (P)                                   //Set the parity bit according to P
            {
            #if (PARITYBIT == ODD_PARITY)
                        TB8 = 0;                     //Set parity bit to 0
            #elif (PARITYBIT == EVEN_PARITY)
                        TB8 = 1;                     //Set parity bit to 1
            #endif
            }
```

```
        else
        {
        #if (PARITYBIT == ODD_PARITY)
                TB8 = 1;                        //Set parity bit to 1
        #elif (PARITYBIT == EVEN_PARITY)
                TB8 = 0;                        //Set parity bit to 0
        #endif
        }
                busy = 1;
                SBUF = ACC;                     //Send data to UART buffer
}

/*----------------------------
Send a string to UART
Input: s (address of string)
Output:None
----------------------------*/
void SendString(char *s)
{
        while (*s)                              //Check the end of the string
        {
                SendData(*s++);                 //Send current char and increment string ptr
        }
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机串行口功能 (8-bit/9-bit) -----------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/

;/*Define UART parity mode*/
#define   NONE_PARITY    0                      //None parity
#define   ODD_PARITY     1                      //Odd parity
#define   EVEN_PARITY    2                      //Even parity
#define   MARK_PARITY    3                      //Mark parity
#define   SPACE_PARITY   4                      //Space parity

#define   PARITYBIT      EVEN_PARITY            //Testing even parity
;-----------------------------------------
BUSY      BIT       20H.0                       ;transmit busy flag
;-----------------------------------------
          ORG       0000H
          LJMP      MAIN

          ORG       0023H
          LJMP      UART_ISR
;-----------------------------------------
          ORG       0100H
MAIN:
          CLR       BUSY
          CLR       EA
          MOV       SP,       #3FH
#if (PARITYBIT == NONE_PARITY)
          MOV       SCON,     #50H              ;8-bit variable UART
#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
          MOV       SCON,     #0DAH             ;9-bit variable UART, parity bit initial to 1
#elif (PARITYBIT == SPACE_PARITY)
          MOV       SCON,     #0D2H             ;9-bit variable UART, parity bit initial to 0
#endif
;-------------------------------
```

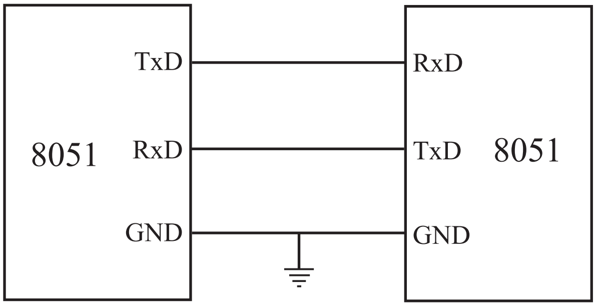
```
        MOV     TMOD,   #20H                    ;Set Timer1 as 8-bit auto reload mode
        MOV     A,      #0FBH                   ;256-18432000/12/32/9600
        MOV     TH1,    A                       ;Set auto-reload vaule
        MOV     TL1,    A
        SETB    TR1                             ;Timer1 start run
        SETB    ES                              ;Enable UART interrupt
        SETB    EA                              ;Open master interrupt switch
;-------------------------------
        MOV     DPTR,   #TESTSTR                ;Load string address to DPTR
        LCALL   SENDSTRING                      ;Send string
;-------------------------------
        SJMP    $
;-----------------------------------------
TESTSTR:                ;Test string
        DB      "STC12C5410AD Uart Test !",     0DH,0AH,0

;/*----------------------------
;UART2 interrupt service routine
;----------------------------*/
UART_ISR:
        PUSH    ACC
        PUSH    PSW
        JNB     RI,     CHECKTI                 ;Check RI bit
        CLR     RI                              ;Clear RI bit
        MOV     P0,     SBUF                    ;P0 show UART data
        MOV     C,      RB8
        MOV     P2.2,   C                       ;P2.2 show parity bit
CHECKTI:
        JNB     TI,     ISR_EXIT                ;Check S2TI bit
        CLR     TI                              ;Clear S2TI bit
        CLR     BUSY                            ;Clear transmit busy flag
ISR_EXIT:
        POP     PSW
        POP     ACC
        RETI

;/*----------------------------
;Send a byte data to UART
;Input: ACC (data to be sent)
;Output:None
;----------------------------*/
SENDDATA:
        JB      BUSY,   $                       ;Wait for the completion of the previous data is sent
        MOV     ACC,    A                       ;Calculate the even parity bit P (PSW.0)
        JNB     P,      EVEN1INACC              ;Set the parity bit according to P
```

```
ODD1INACC:
#if (PARITYBIT == ODD_PARITY)
        CLR     TB8                     ;Set parity bit to 0
#elif (PARITYBIT == EVEN_PARITY)
        SETB    TB8                     ;Set parity bit to 1
#endif
        SJMP    PARITYBITOK
EVEN1INACC:
#if (PARITYBIT == ODD_PARITY)
        SETB    TB8                     ;Set parity bit to 1
#elif (PARITYBIT == EVEN_PARITY)
        CLR     TB8                     ;Set parity bit to 0
#endif
PARITYBITOK:                            ;Parity bit set completed
        SETB    BUSY
        MOV     SBUF,   A               ;Send data to UART buffer
        RET

;/*----------------------------
;Send a string to UART
;Input: DPTR (address of string)
;Output:None
;----------------------------*/
SENDSTRING:
        CLR     A
        MOVC    A,      @A+DPTR         ;Get current char
        JZ      STRINGEND               ;Check the end of the string
        INC     DPTR                    ;increment string ptr
        LCALL   SENDDATA                ;Send current char
        SJMP    SENDSTRING              ;Check next
STRINGEND:
        RET
;----------------------------------------
        END
```

8.5 双机通信

STC12C5410AD系列单片机的串行通信根据其应用可分为双机通信和多机通信两种。下面先介绍双机通信。

如果两个8051应用系统相距很近，可将它们的串行端口直接相连（TXD—RXD，RXD—TXD，GND—GND—地），即可实现双机通信。为了增加通信距离，减少通道及电源干扰，可采用RS—232C或RS—422、RS—485标准进行双机通信，两通信系统之间采用光—电隔离技术，以减少通道及电源的干扰，提高通信可靠性。



为确保通信成功，通信双方必须在软件上有系列的约定通常称为软件通信“协议”。现举例简介双机异步通信软件“协议”如下：

通信双方均选用2400波特的传输速率，设系统的主频SYSclk=6MHz,甲机发送数据，乙机接收数据。在双机开始通信时，先由甲机发送一个呼叫信号（例如“06H”），以询问乙机是否可以接收数据；乙机接收到呼叫信号后，若同意接收数据，则发回“00H”作为应答信号，否则发“05H”表示暂不能接收数据，；甲机只有在接收到乙机的应答信号“00H”后才可将存储在外部数据存储器中的内容逐一发送给乙机，否则继续向乙机发呼叫信号，直到乙机同意接收。其发送数据格式如下：

字节数n	数据1	数据2	数据3	…	数据n	累加校验和

字节数n：甲机向乙机发送的数据个数；

数据1~数据n：甲机将向乙机发送的n帧数据；

累加校验和：为字节数n、数据1、…、数据n,这(n+1)个字节内容的算术累加和.

乙机根据接收到的“校验和”判断已接收到的n个数据是否正确。若接收正确,向甲机回发“0FH”信号,否则回发“F0H”信号。甲机只有在接收到乙机发回的“0FH”信号才算完成发送任务，返回被调用的程序，否则继续呼叫，重发数据。

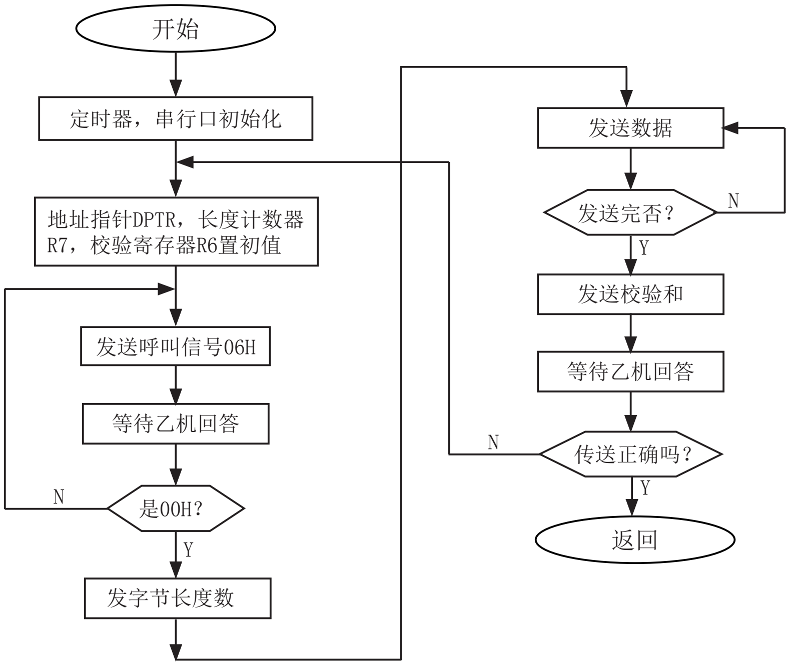
不同的通信要求，软件“协议”内容也不一样，有关需甲、乙双方共同遵守的约定应尽量完善，以防止通信不能正确判别而失败。

STC12C5410AD系列单片机的串行通信，可直接采用查询法，也可采用自动中断法。

（1）查询方式双机通信软件举例

①甲机发送子程序段

下图为甲机发送子程序流程图。



甲机发送程序设置：

(a) 波特率设置：选用定时器/计数器1定时模式、工作方式2，计数常数F3H，SMOD=1。波特率为2400（位/秒）；

(b) 串行通信设置：异步通信方式1，允许接收；

(c) 内部RAM和工作寄存器设置：31H和30H单元存放发送的数据块首地址；2FH单元存放发送的数据块个数；R6为累加和寄存器。

甲机发送子程序清单：

```
START:
        MOV     TMOD，#20H            ；设置定时器/计数器1定时、工作方式2
        MOV     TH1，   #0F3H         ；设置定时计数常数
        MOV     TL1，   #0F3H         ；
        MOV     SCON，  #50H          ；串口初始化
        MOV     PCON，  #80H          ；设置SMOD=1
        SETB    TR1                   ；启动定时
ST-RAM:
        MOV     DPH，   31H           ；设置外部RAM数据指针
        MOV     DPL，   30H           ；DPTR初值
        MOV     R7，    2FH           ；发送数据块数送R7
        MOV     R6，    #00H          ；累加和寄存器R6清0
TX-ACK:
        MOV     A，     #06H          ；
        MOV     SBUF，  A             ；发送呼叫信号“06H”
WAIT1:
        JBC     T1,     RX – YES      ；等待发送完呼叫信号
        SJMP    WAIT1                 ；未发送完转WATI1
RX-YES:
        JBC     RI，    NEXT1         ；判断乙机回答信号
        SJMP    RX-YES                ；未收到回答信号，则等待
NEXT1:
        MOV     A，     SBUF          ；接收回答信号送A
        CJNE    A，     #00H,   TX-ACK；判断是否“00H”，否则重发呼叫信号
TX-BYT:
        MOV     A,      R7            ；
        MOV     SBUF,   A             ；发送数据块数n
        ADD     A,      R6
        MOV     R6,     A
WAIT2:
        JBC     TI,     TX-NES        ；
        JMP     WAIT2                 ；等待发送完
TX-NES:
        MOVX    A,      @DPTR         ；从外部RAM取发送数据
        MOV     SBUF,   A             ；发送数据块
        ADD     A,      R6
        MOV     R6,     A
        INC     DPTR                  ；DPTR指针加1
```

```
WAIT3：
        JBC      TI,      NEXT2              ；判断一数据块发送完否
        SJMP     WAIT3                       ；等待发送完
NEXT2：
        DJNZ     R7,      TX-NES             ；判断发送全部结束否
TX-SUM：
        MOV      A,       R6                 ；发送累加和给乙机
        MOV      SBUF，   A
WAIT4：
        JBC TI,  RX-0FH                      ；
        SJMP     WAIT4                       ；等待发送完
RX-0FH：
        JBC      RI,      IF-0FH             ；
        SJMP     RX-0FH                      ；等待接收乙机回答信号
IF-0FH：
        MOV      A,       SBUF；             ；
        CJNE     A,       #0FH,    ST-RAM    ；判断传输是否正确，否则重新发送
        RET                                  ；返回
```

乙机接收子程序段

接收程序段的设置：

（a） 波特率设置初始化：同发送程序；

（b） 串行通信初始化：同发送程序；

（c） 寄存器设置：

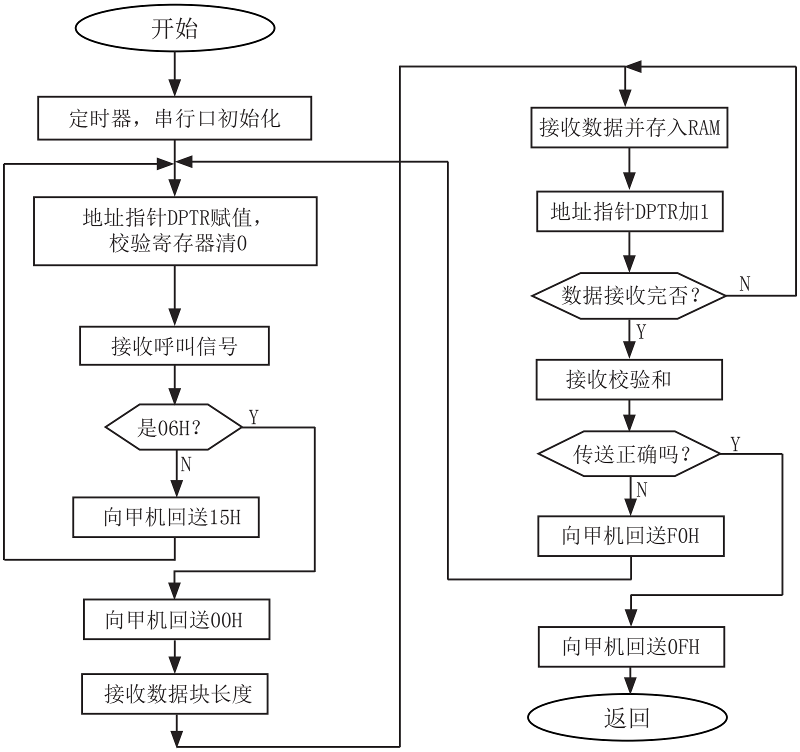
内部RAM 31H、30H单元存放接收数据缓冲区首地址。

R7——数据块个数寄存器。

R6——累加和寄存器。

（d） 向甲机回答信号：“0FH”为接收正确，“F0H”为传送出错，“00H”为同意接收数据，“05H”为暂不接收。

下图为双机通信查询方式乙机接收子程序流程图。



接收子程序清单：

```
TART:
        MOV     TMOD，#20H
        MOV     TH1，   #0F3H          ；
        MOV     TL1，   #0F3H          ；定时器/计数器1设置
        SETB    TR1                    ；启动定时器/计数器1
        MOV     SCON，  #50H           ；置串行通信方式1，允许接收
        MOV     PCON，  #80H           ；SMOD置位
ST-RAM:
        MOV     DPH，   31H            ；
        MOV     DPL，   30H            ；设置DPTR首地址
        MOV     R6，    #00H           ；校验和寄存器清0
```

```
RX-ACK：
        JBC     RI，     IF-06H          ；判断接收呼叫信号
        SJMP    RX-ACK                   ；等待接收呼叫信号
IF-06H：
        MOV     A，      SBUF            ；呼叫信号送A
        CJNEA   #06H，   TX-05H          ；判断呼叫信号正确否？
TX-00H：
        MOV     A，      #00H            ；
        MOV     SBUF，   A               ；  向甲机发送“00H”，同意接收
WAIT1：
        JBC     TI，     RX-BYS          ；等待应答信号发送完
        SJMP    WAIT1
TX-05H：
        MOV     A，      #05H            ；向甲机发送“05H”呼叫
        MOV     SBUF，   A               ；不正确信号
WAIT2：
        JBC     TI，     HAVE1           ；等待发送完
        SJMP    WAIT2
HAVE1：
        LJMP    RX-ACK                   ；因呼叫错，返回重新接收呼叫
RX-BYS：
        JBC     RI，     HAVE2           ；等待接收数据块个数
        SJMP    RX-BYS                   ；
HAVE2：
        MOV     A，      SBUF            ；
        MOV     R7，     A               ；数据块个数帧送R7,R6
        MOV     R6，     A               ；
RX-NES：
        JBC     RI，     HAVE3           ；
        SJMP    RX-NES                   ；  接收数据帧
HAVE3：
        MOV     A，      SBUF            ；
        MOVX    @DPTR，A                 ；接收到的数据存入外部RAM
        INC     DPTR                     ；
        ADD     A，      R6              ；
        MOV     R6，     A               ；  形成累加和
        DJNZ    R7，     RX-NES          ；判断数据是否接收完
```

```
RX-SUM：
        JBC     RI，     HAVE4               ；
        SJMP    RX-SUM                       ；  等待接收校验和
HAVE4：
        MOV     A，      SBUF                ；
        CJNE    A，      R6，     TX-ERR      ；  判断传输是否正确
TX-RIT：
        MOV     A，      #0FH                ；
        MOV     SBUF，   A                   ；  向甲机发送接收正确信息
WAIT3：
        JBC     TI，     GOOD                ；
        SJMP    WAIT3                        ；  等待发送结束
TX-ERR：
        MOV     A，      #0F0H               ；向甲机发送传输有误信号
        MOV     SBUF，   A
WAIT4：
        JBC     TI，     AGAIN               ；等待发送完
        SJMP    WAIT4
AGAIN：
        LJMP    ST-RAM                       ；返回重新开始接收
GOOD：
        RET                                  ；传输正确返回
```

（2）中断方式双机通信软件举例

在很多应用场合，双机通信的双方或一方采用中断方式以提高通信效率。由于STC-12C5410AD系列单片机的串行通信是双工的，且中断系统只提供一个中断矢量入口地址，所以实际上是中断和查询必须相结合，即接收/发送均可各自请求中断，响应中断时主机并不知道是谁请求中断，统一转入同一个中断矢量入口，必须由中断服务程序查询确定并转入对应的服务程序进行处理。

这里，任以上述协议为例，甲方（发送方）任以查询方式通信（从略），乙方（接收方）则改用中断—查询方式进行通信。

在中断接收服务程序中，需设置三个标志位来判断所接收的信息是呼叫信号还是数据块个数，是数据还是校验和。增设寄存器：内部RAM32H单元为数据块个数寄存器，33H单元为校验和寄存器，位地址7FH、7EH、7DH为标志位。

乙机接收中断服务程序清单

采用中断方式时，应在主程序中安排定时器/计数器、串行通信等初始化程序。通信接收的数据存放在外部RAM的首地址也需在主程序中确定。

主程序：

```
        ORG     0000H
        AJMP    START               ；转至主程序起始处
        ORG     0023H
        LIMP    SERVE               ；转中断服务程序处
        •
        •
        •
START：
        MOV     TMOD，#20H          ；定义定时器/计数器1定时、工作方式2
        MOV     TH1，   #0F3H       ；
        MOV     TL1，   #0F3H       ；  设置波特率为2400位/秒
        MOV     SCON，  #50H        ；设置串行通信方式1，允许接收
        MOV     PCON，  #80H        ；设置SMOD=1
        SETB    TR1                 ；启动定时器
        SETB    7FH                 ；
        SETB    7EH                 ；设置标志位为1
        SETB    7DH                 ；
        MOV     31H，#10H           ；  规定接收的数据存储于外部RAM的
        MOV     30H，#00H           ；  起始地址1000H
        MOV     33H，#00H           ；累加和单元清0
        SETB    EA                  ；
        SETB    ES                  ；  开中断
        •
        •
        •
```

中断服务程序：

```
SERVE：
        CLR     EA                      ；关中断
        CLR     RI                      ；清除接收中断请求标志
        PUSH    DPH                     ；
        PUSH    DPL                     ； 现场保护
        PUSH    A                       ；
        JB      7FH，   RXACK           ；判断是否是呼叫信号
        JB      7EH，   RXBYS           ；判断是否是数据块数据
        JB      7DH，   RXDATA          ；判断是否是接收数据帧
RXSUM：
        MOV     A，     SBUF            ；接收到的校验和
        CJNE    A， 33H，  TXERR        ；判断传输是否正确
TXRI：
        MOV     A，     #0FH            ；
        MOV     SBUF，  A               ；向甲机发送接收正确信号“0FH”
WAIT1：
        JNB     TI，    WAITI           ；等待发送完毕
        CLR     TI                      ；清除发送中断请求标志位
        SJMP    AGAIN                   ；转结束处理
TXERR：
        MOV     A，     #0F0H           ；
        MOV     SBUF，  A               ；向甲机发送接收出错信号“F0H”
WAIT2：
        JNB     TI，    WAIT2           ；等待发送完毕
        CLR     TI                      ；清除发送中断请求标志
        SJMP    AGAIN                   ；转结束处理
RXACK：
        MOV     A，     SBUF            ；判断是否是呼叫信号“06H”
        XRL     A，     #06H            ；异或逻辑处理
        JZ      TXREE                   ；是呼叫，则转TXREE
TXNACK：
        MOV     A，     #05H            ；接收到的不是呼叫信号，则向甲机发送
        MOV     SBUF，  A               ；“05H”，要求重发呼叫
```

```
WAIT3：
        JNB     TI，    WAIT3       ； 等待发送结束
        CLR     TI
        SJMP    RETURN              ；转恢复现场处理
TXREE：
        MOV     A，     #00H        ；接收到的是呼叫信号，发送“00H”
        MOV     SBUF，  A           ；接收到的是呼叫信号，发送“00H”
WAIT4：
        JNB     TI，    WAIT4       ；等待发送完毕
        CLR     TI                  ；清除TI标志
        CLR     7FH                 ；清除呼叫标志
        SJMP    RETURN              ；转恢复现场处理
RXBYS：
        MOV     A，     SBUF        ；接收到数据块数
        MOV     32H，   A           ；存入32H单元
        ADD     A，     33H         ；
        MOV     33H，   A           ；  形成累加和
        CLR     7EH                 ；清除数据块数标志
        SJMP    RETURN              ；转恢复现场处理
RXDATA：
        MOV     DPH，   31H         ；
        MOV     DPL，   30H         ；  设置存储数据地址指针
        MOV     A，     SBUF        ；读取数据帧
        MOVX    @DPTR, A            ；将数据存外部RAM
        INC     DPTR                ；地址指针加1
        MOV     31H，   DPH         ；
        MOV     30H，   DPL         ；  保存地址指针值
        ADD     A，     33H         ；
        MOV     33H，   A           ；  形成累加和
        DJNZ    32H，   RETURN      ；判断数据接收完否
        CLR     7DH                 ；清数据接收完标志
        SJMP    RETURN              ；转恢复现场处理
```

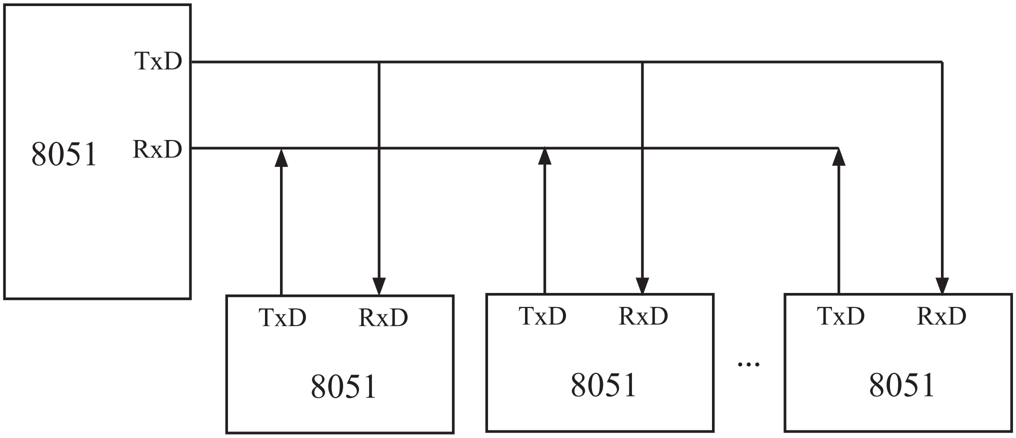
```
AGAIN：
        SETB    7FH                 ；
        SETB    7EH                 ；恢复标志位
        SETB    7DH                 ；
        MOV     33H，    #00H        ；累加和单元清0
        MOV     31H，    #10H        ；
        MOV     30H，    #00H        ；恢复接收数据缓冲区首地址
RETURN：
        POP     A                   ；
        POP     DPL                 ；恢复现场
        POP     DPH                 ；
        SETB    EA                  ；开中断
        RET1                        ；返回
```

上述程序清单中，ORG为程序段说明伪指令，在程序汇编时，它向汇编程序说明该程序段的起始地址。

在实际应用中情况多种多样，而且是两台独立的计算机之间进行信息传输。因此，应周密考虑通信协议，以保证通信的正确性和成功率

8.6 多机通信

在很多实际应用系统中，需要多台微计算机协调工作。STC12C5410AD系列单片机的串行通信方式2和方式3具有多机通信功能，可构成各种分布式通信系统。下图为全双工主从式多机通信系统的连接框图。



上图为一台主机和几台从机组成的全双工多机通信系统。主机可与任一台从机通信，而从机之间的通信必须通过知己转发。

（1）多机通信的基本原理

在多机通信系统中，为保证主机（发送）与多台从机（接收）之间能可靠通信，串行通信必须具备识别能力。MCS-51系列单片机的串行通信控制寄存器SCON中设有多机通信选择位SM2。当程序设置SM2=1，串行通信工作于方式2或方式8，发送端通过对TB8的设置以区别于发送的是地址帧（TB8=1）还是数据帧（TB8=0），接收端通过对接收到RB8进行识别：当SM2=1，若接收到RB8=1，则被确认为呼叫地址帧，将该帧内容装入SBUF中，并置位RI=1，向CPU请求中断，进行地址呼叫处理；若RB8=0为数据帧，将不予理睬，接收的信息被丢弃。若SM2=0，则无论是地址帧还是数据帧均接收，并置位RI=1，向CPU请求中断，将该帧内容装入SBUF。据此原理，可实现多机通信。

对于上图的从机式多机通信系统，从机的地址为0，1，2，…，n。实现多机通信的过程如下：

① 置全部从机的SM2=1，处于只接收地址帧状态。

② 主机首先发送呼叫地址帧信息，将TB8设置为1，以表示发送的是 呼叫地址帧。

③ 所有从机接收到呼叫地址帧后，各自将接收到的主机呼叫的地址与本机的地址相比较：若比较结果相等，则为被寻址从机，清除SM2=0，准备接收从主机发送的数据帧，直至全部数据传输完；若比较不相等，则为非寻址从机，任维持SM2=1不变，对其后发来的数据帧不予理睬，即接收到的数据帧内容不装入SBUF，不置位，RI=0，不会产生中断请求，直至被寻址为止。

④ 主机在发送完呼叫地址帧后，接着发送一连串的数据帧，其中的

TB8=0，以表示为数据帧。

⑤ 当主机改变从机通信时间则再发呼叫地址帧，寻呼其他从机，原先

被寻址的从机经分析得知主机在寻呼其他从机时，恢复其SM2=1，对其后主机发送的数据帧不予理睬。

上述过程均在软件控制下实现。

（2）多机通信协议简述

由于串行通信是在二台或多台各自完全独立的系统之间进行信息传

输这就需要根据时间通信要求制定某些约定，作为通信规范遵照执行，协议要求严格、完善，不同的通信要求，协议的内容也不相同。在多机通信系统中要考虑的问题较多，协议内容比较复杂。这里仅例举几条作一说明。

上图的主从式多机通信系统，允许配置255台从机，各从机的地址分别为00H~FEH。

① 约定地址FFH为全部从机的控制命令，命令各从机恢复SM2=1状态，准备接收主机的地址呼叫。

② 主机和从机的联络过程约定：主机首先发送地址呼叫帧，被寻址的从机回送本机地址给主机，经验证地址相符后主机再向被寻址的从机发送命令字，被寻址的从机根据命令字要求回送本机的状态，若主机判断状态正常，主机即开始发送或接收数据帧，发送或接收的第一帧为传输数据块长度。

③ 约定主机发送的命令字为：

00H：要求从机接收数据块；

01H：要求从机发送数据块；

•

•

•

其他：非法命令。

④ 从机的状态字格式约定为：

B7	B6	B5	B4	B3	B2	B1	B0
ERR	0	0	0	0	0	TRDY	RRDY

定义： 若ERR=1，从机接收到非法命令；

若TRDY=1，从机发送准备就绪；

若RRDY=1，从机接收准备就绪；

⑤ 其他：如传输出错措施等。

（3）程序举例

在实际应用中如传输波特率不太高，系统实时性有一定要求以及希望提高通信效率，则多半采用中断控制方式，但程序调试较困难，这就要求提高程序编制的正确性。采用查询方式，则程序调试较方便。这里仅以中断控制方式为例简单介绍主—从机之间一对一通信软件。

① 主机发送程序

该主机要发送的数据存放在内部RAM中，数据块的首地址为51H，数据块长度存放做50H单元中，有关发送前的初始化、参数设置等采用子程序格式，所有信息发送均由中断服务程序完成。当主机需要发送时，在完成发送子程序的调用之后，随即返回主程序继续执行。以后只需查询PSW·5的F0标志位的状态即可知道数据是否发送完毕。

要求主机向#5从机发送数据，中断服务程序选用工作寄存存器区1的R0~R7。

主机发送程序清单：

```
        ORG     0000H
        AJMP    MAIN                        ；转主程序
        ORG     0023H                       ；发送中断服务程序入口
        LJMP    SERVE                       ；转中断服务程序
          :
MAIN：   . . .   . . .                      ；主程序
          :
        ORG     1000H                       ；发送子程序入口
TXCALL：
        MOV     TMOD，#20H                  ；设置定时器/计数器1定时、方式2
        MOV     TH1，   #0F3H               ；设置波特率为2400位/秒
        MOV     TL1，   #0F3H               ；置位SMOD
        MOV     PCON，#80H                  ；
        SETB    TR1                         ；启动定时器/计数器1
        MOV     SCON，#0D8H                 ；串行方式8，允许接收，TB8=1
        SETB    EA                          ；开中断总控制位
        CLR     ES                          ；禁止串行通信中断
TXADDR：
        MOV     SBUF，  #05H                ；发送呼叫从机地址
WAIT1：
        JNB     TI，    WAIT1               ；等待发送完毕
        CLR     TI                          ；复位发送中断请求标志
```

```
RXADDR：
        JNB     RI，     RXADDR                 ；等待从机回答本机地址
        CLR     TI                              ；复位接收中断请求标志
        MOV     A，      SBUF                   ；读取从机回答的本机地址
        CJNE    A，      #05H，  TXADDR         ；判断呼叫地址符否，否则重发
        CLR     TB8                             ；地址相符，复位TB8=0，准备发数据
        CLR     PSW. 5                          ；复位F0=0标志位
        MOV     08H，    #50H                   ；发送数据地址指针送R0
        MOV     0CH，    50H                    ；数据块长度送R4
        INC     0CH                             ；数据块长度加1
        SETB    ES                              ；允许串行通信中断
        RET                                     ；返回主程序
         ⋮
SERVE：
        CLR     TI                              ；中断服务程序段，清中断请求标志TI
        PUSH    PSW                             ；
        PUSH    A                               ；  现场入栈保护
        CLR     RS1                             ；
        SETB    RS0                             ；  选择工作寄存器区1
TXDATA：
        MOV     SBUF，   @R0                    ；发送数据块长度及数据
WAIT2：
        JNB     TI，     WAIT2                  ；等待发送完毕
        CLR     TI                              ；复位TI=0
        INC     R0                              ；地址指针加1
        DJNZ    R4，     RETURN                 ；数据块未发送完，转返回
        SETB    PSW. 5                          ；已发送完毕置位F0=1
        CLR     ES                              ；关闭串行中断
RETURN：
        POP     A                               ；
        POP     PSW                             ；  恢复现场
        RETI                                    ；返回
```

②从机接收程序

主机发送的地址呼叫帧，所有的从机均接收，若不是呼叫本机地址即从中断返回；若是本机地址，则回送本机地址给主机作为应答，并开始接收主机发送来的数据块长度帧，并存放于内部RAM的60H单元中，紧接着接收的数据帧存放于61H为首地址的内部RAM单元中，程序中还选用20·0H、20·1H位作标志位，用来判断接收的是地址、数据块长度还是数据，选用了2FH、2EH两个字节单元用于存放数据字节数和存储数据指针。#5从机的接收程序如下，供参考。

#5从机接收程序清单：

```
        ORG     0000H
        AJMP    START                   ；转主程序段
        ORG     0023H
        LJMP    SERVE                   ；从中断入口转中断服务程序
        ORG     0100H
START：
        MOV     TMOD，#20H              ；主程序段：初始化程序，设置定时
        MOV     TH1，   #0F3H           ；器/计数器1定时、工作方式2，设
        MOV     TL1，   #0F3H           ；置波特率为2400位/秒的有关初值
        MOV     PCON，  #80H            ；置位SMOD
        MOV     SCON，  #0F0H           ；设置串行方式3，允许接收，SM2=1
        SETB    TR1                     ；启动定时器/计数器1
        SETB    20•0                    ；
        SETB    20•1                    ；  置标志位为1
        SETB    EA                      ；
        SETB    ES                      ；  开中断
        ·
        ·
        ·
        ORG     1000H
SERVE：
        CLR     RI                      ；清接收请求中断标志RI=0
        PUSH    A                       ；
        PUSH    PSW                     ；  现场保护
        CLR     RS1                     ；
        SETB    RS0                     ；  选择工作寄存器区1
        JB      20•0H，    ISADDR       ；判断是否是地址帧
        JB      20•1H，    ISBYTE       ；判断是否是数据块长度帧
```

```
ISDATA：
        MOV     R0，     2EH                          ；数据指针送R0
        MOV     A，      SBUF                         ；接收数据
        MOV     @R0，    A
        INC     2EH                                   ；数据指针加1
        DJNZ    2FH，    RETURN                       ；判断数据接收完否？
        SETB    20・0H                                ；
        SETB    20・1H                                ；恢复标志位
        SETB    SM2                                   ；
        SJMP    RETURN                                ；转入恢复现场，返回
ISADDR：
        MOV     A，      SBUF                         ； 是地址呼叫，判断与本机地址
        CJNE    A，      #05H，    RETURN             ；相符否，不符则转返回
        MOV     SBUF，   #01H                         ；相符，发回答信号“01H”
WAIT：
        JNB     TI，     WAIT                         ；等待发送结束
        CLR     TI                                    ；清0TI，20・0，SM2
        CLR     20・0H                                ；清0TI，20・0，SM2
        CLR     SM2                                   ；清0TI，20・0，SM2
        SJMP    RETURN                                ；转返回
ISBYTES：
        MOV     A，      SBUF                         ；接收数据块长度帧
        MOV     R0，     #60H                         ；
        MOV     @R0，    A                            ；将数据块长度存入内部RAM
        MOV     2FH，    A                            ；60H单元及2FH单元
        MOV     2EH，    #61H                         ；置首地址61H于2EH单元
        CLR     20・1H                                ；清20・1H标志，表示以后接收的为数据
RETURN：
        POP     PSW                                   ；
        POP     A                                     ；恢复现场
        RETI                                          ；返回
```

多机通信方式可多种多样，上例仅以最简单的住一从式作了简单介绍，仅供参考。

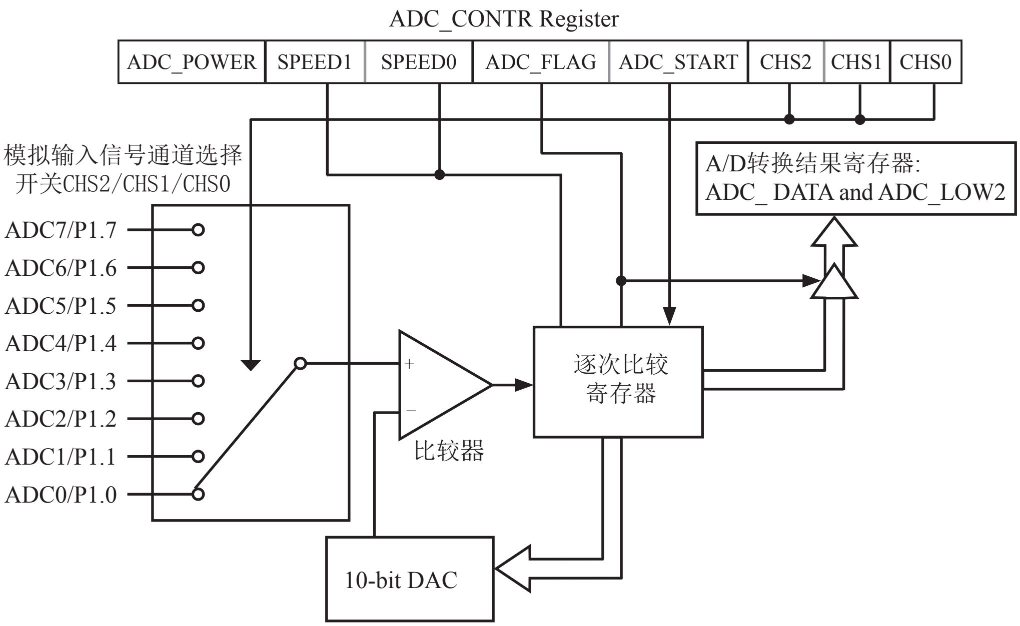
对于串行通信工作方式0的同步方式，常用于通过移位寄存器进行扩展并行I/O口，或配置某些串行通信接口的外部设备。例如，串行打印机、显示器等。这里就不一一举例了。

第9章 STC12C5410AD系列单片机的A/D转换器

9.1 A/D转换器的结构

STC12C5410AD系列带A/D转换的单片机的A/D转换口在P1口(P1.7-P1.0)，有8路10位高速A/D转换器,速度可达到100KHz(10万次/秒)。8路电压输入型A/D，可做温度检测、电池电压检测、按键扫描、频谱检测等。上电复位后P1口为弱上拉型I/O口，用户可以通过软件设置将8路中的任何一路设置为A/D转换，不需作为A/D使用的口可继续作为I/O口使用。

STC12C5410AD系列单片机ADC(A/D转换器)的结构如下图所示。



A/D转换结果寄存器格式如下：

ADC_DATA[7:0]

ADC_B9	ADC_B8	ADC_B7	ADC_B6	ADC_B5	ADC_B4	ADC_B3	ADC_B2

ADC_LOW2[1:0]

-	-	-	-	-	-	ADC_B1	ADC_B0

STC12C5410AD系列单片机ADC由多路选择开关、比较器、逐次比较寄存器、10位DAC、转换结果寄存器(ADC_DATA和ADC_LOW2)以及ADC_CONTR构成。

STC12C5410AD系列单片机的ADC是逐次比较型ADC。逐次比较型ADC由一个比较器和D/A转换器构成，通过逐次比较逻辑，从最高位(MSB)开始，顺序地对每一输入电压与内置D/A转换器输出进行比较，经过多次比较，使转换所得的数字量逐次逼近输入模拟量对应值。逐次比较型A/D转换器具有速度高，功耗低等优点。

从上图可以看出，通过模拟多路开关，将通过ADC0~7的模拟量输入送给比较器。用数/模转换器(DAC)转换的模拟量与本次输入的模拟量通过比较器进行比较，将比较结果保存到逐次比较器，并通过逐次比较寄存器输出转换结果。A/D转换结束后，最终的转换结果保存到ADC转换结果寄存器ADC_DATA和ADC_LOW2，同时，置位ADC控制寄存器ADC_CONTR中的A/D转换结束标志位ADC_FLAG,以供程序查询或发出中断申请。模拟通道的选择控制由ADC控制寄存器ADC_CONTR中的CHS2 ~ CHS0确定。ADC的转换速度由ADC控制寄存器中的SPEED1和SPEED0确定。在使用ADC之前，应先给ADC上电，也就是置位ADC控制寄存器中的ADC_POWER位。

如果用户取完整的10位结果，则按下面公式计算:

$$\text{10-bit A/D Conversion Result:(ADC\_DATA[7:0], ADC\_LOW2[1:0])} = 1024 \times \frac{Vin}{Vcc}$$

如果舍弃ADC_LOW2的低2位,只用ADC_DATA寄存器的8位结果，按下面公式计算:

$$\text{8-bit A/D Conversion Result:(ADC\_DATA[7:0])} = 256 \times \frac{Vin}{Vcc}$$

式中，Vin为模拟输入通道输入电压，Vcc为单片机实际工作电压，用单片机工作电压作为模拟参考电压。

9.2 与A/D转换相关的寄存器

与STC12C5410AD系列单片机A/D转换相关的寄存器列于下表所示。

符号	描述	地址	位地址及其符号 MSB							LSB	复位值
P1M0	P1口模式配置寄存器0	91H									0000 0000B
P1M1	P1口模式配置寄存器1	92H									0000 0000B
ADC_CONTR	ADC Control Register	C5H	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0	0000 0000B
ADC_DATA	A/D转换结果寄存器，高8位	C6H									0000 0000B
ADC_LOW2	A/D转换结果寄存器，低2位	BEH									0000 0000B
AUXR	Auxiliary register	A2H	T0x12	T1x12	UART_M0x12	EADCI	ESPI	ELVDI	-	-	0000 00xxB
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B

1. P1口模拟配置寄存器P1M0和P1M1

需作为A/D使用的P1口需先将其设置为开漏模式或高阻输入，在P1M0(地址91H)、P1M1(地址92H)寄存器中对相应的位进行设置。

P1口设定 <P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0口>(P1口地址：90H)

P1M0 [7 : 0]	P1M1 [7 : 0]	I/O 口模式（P1.x 如做A/D使用，需先将其设置成开漏或高阻输入）
0	0	准双向口(传统8051 I/O 口模式）， 灌电流可达20mA，拉电流为230μA，
0	1	推挽输出（ 强上拉输出， 可达20mA，要加限流电阻，尽量少用）
1	0	仅为输入（ 高阻），如果该I/O口需作为A/D使用，可选此模式
1	1	开漏(Open Drain)，如果该I/O口需作为A/D使用，可选此模式

2. ADC控制寄存器ADC_CONTR

ADC_CONTR寄存器的格式如下：

ADC_CONTR：ADC控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
ADC_CONTR	C5H	name	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0

对ADC_CONTR寄存器进行操作，建议直接用MOV赋值语句，不要用‘与’和‘或’语句。

ADC_POWER：ADC电源控制位。

0：关闭A/D转换器电源；

1：打开A/D转换器电源.

建议进入空闲模式前，将ADC电源关闭，即ADC_POWER =0。启动A/D转换前一定要确认A/D电源已打开，A/D转换结束后关闭A/D电源可降低功耗，也可不关闭。初次打开内部A/D转换模拟电源，需适当延时，等内部模拟电源稳定后，再启动A/D转换。

建议启动A/D转换后，在A/D转换结束之前，不改变任何I/O口的状态，有利于高精度A/D转换，若能将定时器/串行口/中断系统关闭更好。

SPEED1，SPEED0：模数转换器转换速度控制位

SPEED1	SPEED0	A/D转换所需时间
1	1	270个时钟周期转换一次，CPU工作频率27MHz时，A/D转换速度约100KHz
1	0	540个时钟周期转换一次
0	1	810个时钟周期转换一次
0	0	1080个时钟周期转换一次

ADC_FLAG：模数转换器转换结束标志位，当A/D转换完成后，ADC_FLAG=1，要由软件清0。不管是A/D转换完成后由该位申请产生中断，还是由软件查询该标志位A/D转换是否结束，当A/D转换完成后，ADC_FLAG = 1，一定要软件清0。

ADC_START：模数转换器(ADC)转换启动控制位，设置为“1”时，开始转换,转换结束后为0

CHS2/CHS1/CHS0：模拟输入通道选择，CHS2/CHS1/CHS0

CHS2	CHS1	CHS0	Analog Channel Select（模拟输入通道选择）
0	0	0	选择　P1.0 作为A/D输入来用
0	0	1	选择　P1.1 作为A/D输入来用
0	1	0	选择　P1.2 作为A/D输入来用
0	1	1	选择　P1.3 作为A/D输入来用
1	0	0	选择　P1.4 作为A/D输入来用
1	0	1	选择　P1.5 作为A/D输入来用
1	1	0	选择　P1.6 作为A/D输入来用
1	1	1	选择　P1.7 作为A/D输入来用

程序中需要注意的事项:

由于是2套时钟,所以,设置ADC_CONTR控制寄存器后,要加4个空操作延时才可以正确读到ADC_CONTR寄存器的值,原因是设置ADC_CONTR控制寄存器的语句执行后,要经过4个CPU时钟的延时,其值才能够保证被设置进ADC_CONTR控制寄存器.

```
MOV     ADC_CONTR,  #DATA
NOP
NOP
NOP
NOP
MOV     A,          ADC_CONTR
                    ;经过4个时钟延时后,才能够正确读到ADC_CONTR控制寄存器的值
```

3. A/D转换结果寄存器ADC_DATA、ADC_LOW2

特殊功能寄存器ADC_DATA和ADC_LOW2寄存器用于保存A/D转换结果，其格式如下：

Mnemonic	Add	Name	B7	B6	B5	B4	B3	B2	B1	B0
ADC_DATA	C6h	A/D转换结果寄存器,全部8位有效，为10位A/D转换结果的高8位	-	-	-	-	-	-	-	-
ADC_LOW2	BEh	A/D转换结果寄存器，只有低2位有效，为10位A/D转换结果的低2位	x	x	x	x	x	x	-	-

此时，如果用户需取完整10位结果，按下面公式计算:

$$\text{10-bit A/D Conversion Result:(ADC\_DATA[7:0], ADC\_LOW2[1:0])} = 1024 \times \frac{Vin}{Vcc}$$

如果用户只需取8位结果，按下面公式计算:

$$\text{8-bit A/D Conversion Result:(ADC\_DATA[7:0])} = 256 \times \frac{Vin}{Vcc}$$

式中，Vin为模拟输入通道输入电压，Vcc为单片机实际工作电压，用单片机工作电压作为模拟参考电压。

4. 与A/D中断有关的寄存器

IE：中断允许寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0

EA：CPU的中断开放标志，EA=1，CPU开放中断，EA=0，CPU屏蔽所有的中断申请。
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。

EADC_SPI：A/D转换中断和SPI中断允许位。
EADC_SPI =1，允许A/D转换中断和SPI中断；
EADC_SPI =0，禁止A/D转换中断和SPI中断。

AUXR：辅助寄存器（不可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	T0x12	T1x12	UART_M0x12	EADCI	ESPI	ELVDI	-	-

EADCI：A/D转换中断单独允许位。
EADCI =1，允许A/D转换中断；
EADCI =0，禁止A/D转换中断。

如果要允许A/D转换中断则需要将几个相应的控制位置1:
1、将EADCI置1，允许ADC中断，这是ADC中断的单独控制位。
2、将EADCI_SPI置1，允许ADC中断及SPI中断，这是ADC中断及SPI中断的总中断控制位，此位不打开，也是无法产生ADC中断的。
3、将EA置1，打开单片机总中断控制位，此位不打开，也是无法产生ADC中断的A/D中断服务程序中要用软件清A/D中断请求标志位ADC_FLAG。

IPH：中断优先级控制寄存器高（不可位寻址）

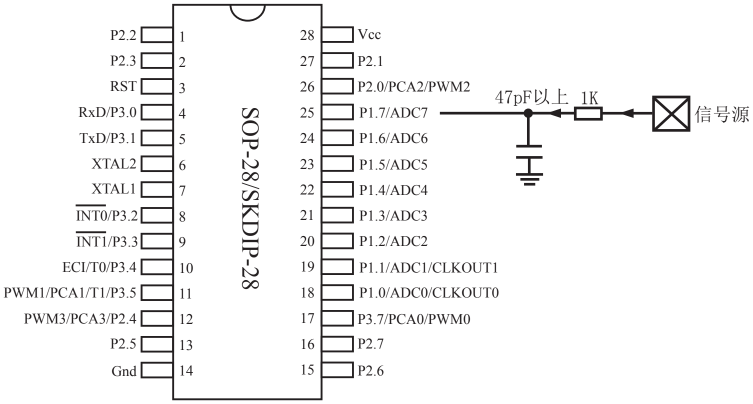
SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IPH	B7H	name	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H

IP：中断优先级控制寄存器低（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IP	B8H	name	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0

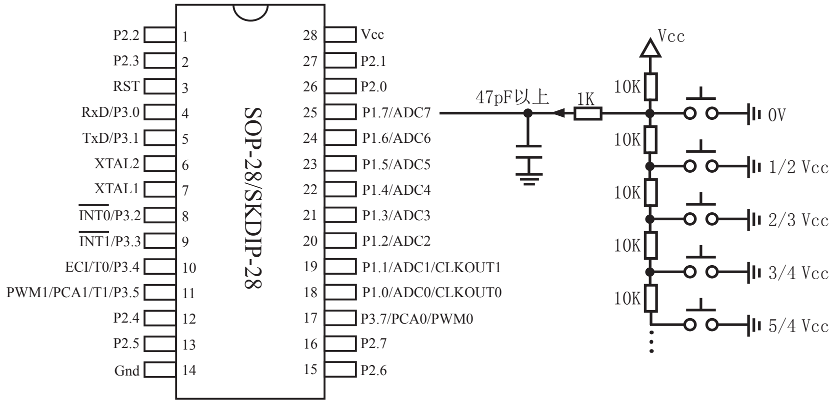
PADC_SPIH, PADC_SPI: A/D转换中断优先级控制位。
当PADC_SPIH=0且PADC_SPI=0时，A/D转换中断和SPI中断为最低优先级中断(优先级0)
当PADC_SPIH=0且PADC_SPI=1时，A/D转换中断和SPI中断为较低优先级中断(优先级1)
当PADC_SPIH=1且PADC_SPI=0时，A/D转换中断和SPI中断为较高优先级中断(优先级2)
当PADC_SPIH=1且PADC_SPI=1时，A/D转换中断和SPI中断为最高优先级中断(优先级3)
‘

9.3 A/D转换典型应用线路

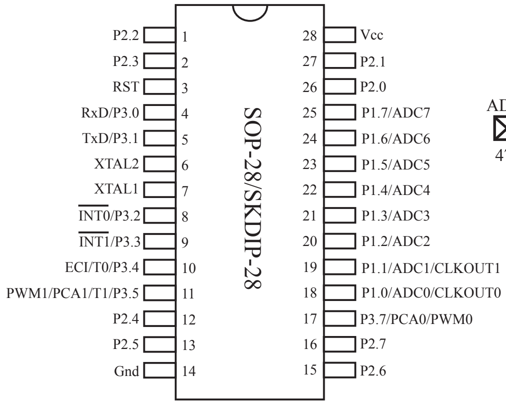


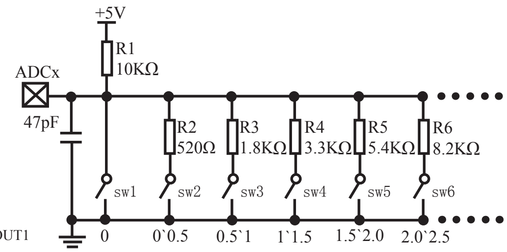
A/D转换在P1口，P1.0 - P1.7共8路

9.4 A/D做按键扫描应用线路图



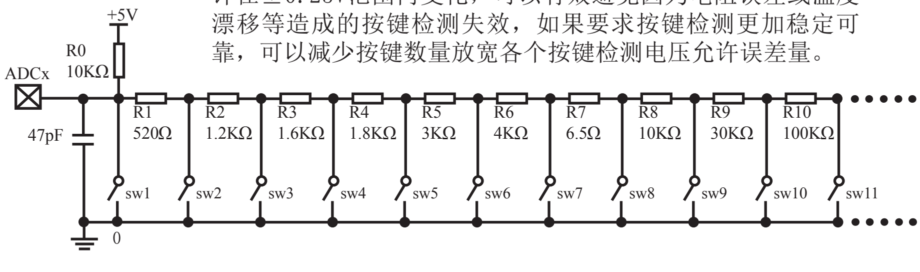
A/D转换在P1口，P1.0 - P1.7共8路





此电路可以实现单个按键扫描和组合按键检测功能，但是具体电阻值应根据实际需要进行选择

本电路图采用10个按键等间隔分压，每个按键正负误差余量允许在±0.25V范围内变化，可以有效避免因为电阻误差或温度漂移等造成的按键检测失效，如果要求按键检测更加稳定可靠，可以减少按键数量放宽各个按键检测电压允许误差量。



9.5 A/D转换模块的参考电压源

STC12C5410AD系列单片机的参考电压源是输入工作电压Vcc，所以一般不用外接参考电压源。如7805的输出电压是5V，但实际电压可能是4.88V 到4.96V，用户需要精度比较高的话，可在出厂时将实际测出的工作电压值记录在单片机内部的EEPROM里面，以供计算。

如果有些用户的Vcc不固定，如电池供电，电池电压在5.3V-4.2V之间漂移，则Vcc不固定，就需要在8路A/D转换的一个通道外接一个稳定的参考电压源，来计算出此时的工作电压Vcc，再计算出其他几路A/D转换通道的电压。如可在ADC转换通道的第七通道外接一个1.25V（或1V，或．．．）的基准参考电压源，由此求出此时的工作电压Vcc，再计算出其它几路A/D转换通道的电压(理论依据是短时间之内，Vcc不变)。

9.6 A/D转换测试程序(C程序和汇编程序)

9.6.1 A/D转换测试程序(ADC中断方式)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机  A/D转换功能-----------------------------*/
/* 如果要在程序中使用或在文章中引用该程序，  -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

#define   FOSC    18432000L
#define   BAUD    9600

typedef   unsigned char    BYTE;
typedef   unsigned int     WORD;

/*Declare SFR associated with the ADC */
sfr       AUXR          =  0x8e;
sfr       ADC_CONTR     =  0xC5;          //ADC control register
sfr       ADC_DATA      =  0xC6;          //ADC high 8-bit result register
sfr       ADC_LOW2      =  0xBE;          //ADC low 2-bit result register
sfr       P1M0          =  0x91;          //P1 mode control register0
sfr       P1M1          =  0x92;          //P1 mode control register1

/*Define ADC operation const for ADC_CONTR*/
#define   ADC_POWER     0x80                      //ADC power control bit
#define   ADC_FLAG      0x10                      //ADC complete flag
#define   ADC_START     0x08                      //ADC start control bit
#define   ADC_SPEEDLL   0x00                      //1080 clocks
#define   ADC_SPEEDL    0x20                      //810 clocks
#define   ADC_SPEEDH    0x40                      //540 clocks
#define   ADC_SPEEDHH   0x60                      //270 clocks

void InitUart();
void SendData(BYTE dat);
void Delay(WORD n);
void InitADC();
```

```
BYTE    ch = 0;                                  //ADC channel NO.

void main()
{
        InitUart();                              //Init UART, use to show ADC result
        InitADC();                               //Init ADC sfr
        AUXR |= 0x10;                            //set EADCI
        IE = 0xa0;                               //Enable ADC interrupt and Open master interrupt switch
                                                 //Start A/D conversion
        while (1);
}
/*----------------------------
ADC interrupt service routine
----------------------------*/
void adc_isr() interrupt 5 using 1
{
        ADC_CONTR &= !ADC_FLAG;                  //Clear ADC interrupt flag

        SendData(ch);                            //Show Channel NO.
        SendData(ADC_DATA);                      //Get ADC high 8-bit result and Send to UART

        //if you want show 10-bit result, uncomment next line
        //    SendData(ADC_LOW2);                //Show ADC low 2-bit result

        if (++ch > 7) ch = 0;                    //switch to next channel
        ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ADC_START | ch;
}
/*----------------------------
Initial ADC sfr
----------------------------*/
void InitADC( )
{
        P1 = P1M0 = P1M1 = 0xff;                         //Set all P1 as Open-Drain mode
        ADC_DATA = 0;                                    //Clear previous result
        ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ADC_START | ch;
        Delay(2);                                        //ADC power-on delay and Start A/D conversion
}
/*----------------------------
Initial UART
----------------------------*/
void InitUart()
{
        SCON = 0x5a;                                     //8 bit data ,no parity bit
        TMOD = 0x20;                                     //T1 as 8-bit auto reload
        TH1 = TL1 = -(FOSC/12/32/BAUD);                  //Set Uart baudrate
        TR1 = 1;                                         //T1 start running
}
```

```
/*----------------------------
Send one byte data to PC
Input: dat (UART data)
Output:-
----------------------------*/
void SendData(BYTE dat)
{
        while (!TI);                    //Wait for the previous data is sent
        TI = 0;                         //Clear TI flag
        SBUF = dat;                     //Send current data
}

/*----------------------------
Software delay function
----------------------------*/
void Delay(WORD n)
{
        WORD x;

        while (n--)
        {
                x = 5000;
                while (x--);
        }
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机 A/D转换功能------------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the ADC */
AUXR            EQU     8EH
ADC_CONTR       EQU     0C5H                ;ADC control register
ADC_DATA        EQU     0C6H                ;ADC high 8-bit result register
ADC_LOW2        EQU     0BEH                ;ADC low 2-bit result register
P1M0            EQU     091H                ;P1 mode control register0
P1M1            EQU     092H                ;P1 mode control register1

;/*Define ADC operation const for ADC_CONTR*/
ADC_POWER       EQU     80H                 ;ADC power control bit
ADC_FLAG        EQU     10H                 ;ADC complete flag
ADC_START       EQU     08H                 ;ADC start control bit
ADC_SPEEDLL     EQU     00H                 ;1080 clocks
ADC_SPEEDL      EQU     20H                 ;810 clocks
ADC_SPEEDH      EQU     40H                 ;540 clocks
ADC_SPEEDHH     EQU     60H                 ;270 clocks

ADCCH           DATA    20H                 ;ADC channel NO.

;-----------------------------------------
        ORG     0000H
        LJMP    MAIN

        ORG     002BH
        LJMP    ADC_ISR
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,     #3FH
        MOV     ADCCH, #0
        LCALL   INIT_UART                   ;Init UART, use to show ADC result
        LCALL   INIT_ADC                    ;Init ADC sfr
        ORL     AUXR,   #10H                ;set EADCI
        MOV     IE,     #0A0H       ;Enable ADC interrupt and Open master interrupt switch
        SJMP    $
```

```
;/*----------------------------
;ADC interrupt service routine
;----------------------------*/
ADC_ISR:
          PUSH    ACC
          PUSH    PSW

          ANL     ADC_CONTR,     #NOT ADC_FLAG          ;Clear ADC interrupt flag
          MOV     A,        ADCCH
          LCALL   SEND_DATA                             ;Send channel NO.
          MOV     A,        ADC_RES                     ;Get ADC high 8-bit result
          LCALL   SEND_DATA                             ;Send to UART

;//if you want show 10-bit result, uncomment next 2 lines
;         MOV     A,        ADC_LOW2                    ;Get ADC low 2-bit result
;         LCALL   SEND_DATA                             ;Send to UART

          INC     ADCCH
          MOV     A,        ADCCH
          ANL     A,        #07H
          MOV     ADCCH, A
          ORL     A,        #ADC_POWER | ADC_SPEEDLL | ADC_START
          MOV     ADC_CONTR,     A                ;ADC power-on delay and re-start A/D conversion
          POP     PSW
          POP     ACC
          RETI

;/*----------------------------
;Initial ADC sfr
;----------------------------*/
INIT_ADC:
          MOV     A,        #0FFH
          MOV     P1,       A
          MOV     P1M0,     A
          MOV     P1M1,     A                     ;Set all P1 as Open-Drain mode
          MOV     ADC_DATA,  #0                   ;Clear previous result
          MOV     A,        ADCCH
          ORL     A,        #ADC_POWER | ADC_SPEEDLL | ADC_START
          MOV     ADC_CONTR,  A                   ;ADC power-on delay and Start A/D conversion
          MOV     A,        #2
          LCALL   DELAY
          RET
```

```
;/*----------------------------
;Initial UART
;----------------------------*/
INIT_UART:
        MOV     SCON,   #5AH                    ;8 bit data ,no parity bit
        MOV     TMOD,   #20H                    ;T1 as 8-bit auto reload
        MOV     A,      #-5                     ;Set Uart baudrate -(18432000/12/32/9600)
        MOV     TH1,    A                       ;Set T1 reload value
        MOV     TL1,    A
        SETB    TR1                             ;T1 start running
        RET

;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
        JNB     TI,     $                       ;Wait for the previous data is sent
        CLR     TI                              ;Clear TI flag
        MOV     SBUF,   A                       ;Send current data
        RET

;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        MOV     R2,     A
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

        END
```

9.6.2 A/D转换测试程序(ADC查询方式)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- --------------------------------------*/
/* --- 演示STC 1T 系列单片机 A/D转换功能------------------------------*/
/* 如果要在程序中使用或在文章中引用该程序, ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

#define   FOSC     18432000L
#define   BAUD     9600

typedef   unsigned  char      BYTE;
typedef   unsigned  int       WORD;

/*Declare SFR associated with the ADC */
sfr       ADC_CONTR      =  0xC5;                //ADC control register
sfr       ADC_DATA       =  0xC6;                //ADC high 8-bit result register
sfr       ADC_LOW2       =  0xBE;                //ADC low 2-bit result register
sfr       P1M0           =  0x91;                //P1 mode control register0
sfr       P1M1           =  0x92;                //P1 mode control register1

/*Define ADC operation const for ADC_CONTR*/
#define   ADC_POWER      0x80                    //ADC power control bit
#define   ADC_FLAG       0x10                    //ADC complete flag
#define   ADC_START      0x08                    //ADC start control bit
#define   ADC_SPEEDLL    0x00                    //1080 clocks
#define   ADC_SPEEDL     0x20                    //810 clocks
#define   ADC_SPEEDH     0x40                    //540 clocks
#define   ADC_SPEEDHH    0x60                    //270 clocks

void InitUart();
void InitADC();
void SendData(BYTE dat);
BYTE GetADCResult(BYTE ch);
void Delay(WORD n);
void ShowResult(BYTE ch);
```

```
void main()
{
            InitUart();                                         //Init UART, use to show ADC result
            InitADC();                                          //Init ADC sfr
            while (1)
            {
                        ShowResult(0);                          //Show Channel0
                        ShowResult(1);                          //Show Channel1
                        ShowResult(2);                          //Show Channel2
                        ShowResult(3);                          //Show Channel3
                        ShowResult(4);                          //Show Channel4
                        ShowResult(5);                          //Show Channel5
                        ShowResult(6);                          //Show Channel6
                        ShowResult(7);                          //Show Channel7
            }
}

/*----------------------------
Send ADC result to UART
----------------------------*/
void ShowResult(BYTE ch)
{
            SendData(ch);                                       //Show Channel NO.
            SendData(GetADCResult(ch));                         //Show ADC high 8-bit result

//if you want show 10-bit result, uncomment next line
//          SendData(ADC_LOW2);                                 //Show ADC low 2-bit result
}

/*----------------------------
Get ADC result
----------------------------*/
BYTE GetADCResult(BYTE ch)
{
            ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ch | ADC_START;
            _nop_();                                            //Must wait before inquiry
            _nop_();
            _nop_();
            _nop_();
            while (!(ADC_CONTR & ADC_FLAG));                    //Wait complete flag
            ADC_CONTR &= ~ADC_FLAG;                             //Close ADC

            return ADC_DATA;                                    //Return ADC result
}
```

```
/*----------------------------
Initial UART
----------------------------*/
void InitUart()
{
        SCON = 0x5a;                        //8 bit data ,no parity bit
        TMOD = 0x20;                        //T1 as 8-bit auto reload
        TH1 = TL1 = -(FOSC/12/32/BAUD);     //Set Uart baudrate
        TR1 = 1;                            //T1 start running
}

/*----------------------------
Initial ADC sfr
----------------------------*/
void InitADC()
{
         P1 = P1M0 = P1M1 = 0xff;           //Set all P1 as Open-Drain mode
        ADC_DATA = 0;                       //Clear previous result
        ADC_CONTR = ADC_POWER | ADC_SPEEDLL;
        Delay(2);                           //ADC power-on and delay
}

/*----------------------------
Send one byte data to PC
Input: dat (UART data)
Output:-
----------------------------*/
void SendData(BYTE dat)
{
        while (!TI);                        //Wait for the previous data is sent
        TI = 0;                             //Clear TI flag
        SBUF = dat;                         //Send current data
}

/*----------------------------
Software delay function
----------------------------*/
void Delay(WORD n)
{
        WORD x;

        while (n--)
        {
                x = 5000;
                while (x--);
        }
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机 A/D转换功能------------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the ADC */
ADC_CONTR       EQU     0C5H                    ;ADC control register
ADC_DATA        EQU     0C6H                    ;ADC high 8-bit result register
ADC_LOW2        EQU     0BEH                    ;ADC low 2-bit result register
P1M0            EQU     091H                    ;P1 mode control register0
P1M1            EQU     092H                    ;P1 mode control register1

;/*Define ADC operation const for ADC_CONTR*/
ADC_POWER       EQU     80H                     ;ADC power control bit
ADC_FLAG        EQU     10H                     ;ADC complete flag
ADC_START       EQU     08H                     ;ADC start control bit
ADC_SPEEDLL     EQU     00H                     ;1080 clocks
ADC_SPEEDL      EQU     20H                     ;810 clocks
ADC_SPEEDH      EQU     40H                     ;540 clocks
ADC_SPEEDHH     EQU     60H                     ;270 clocks

;-----------------------------------------
        ORG     0000H
        LJMP    MAIN
;-----------------------------------------
        ORG     0100H
MAIN:
        LCALL   INIT_UART                       ;Init UART, use to show ADC result
        LCALL   INIT_ADC                        ;Init ADC sfr
;-------------------------------
NEXT:
        MOV     A,      #0
        LCALL   SHOW_RESULT                     ;Show channel0 result
        MOV     A,#1
        LCALL   SHOW_RESULT                     ;Show channel1 result
        MOV     A,#2
        LCALL   SHOW_RESULT                     ;Show channel2 result
```

```
        MOV     A,      #3
        LCALL   SHOW_RESULT                 ;Show channel3 result
        MOV     A,      #4
        LCALL   SHOW_RESULT                 ;Show channel4 result
        MOV     A,      #5
        LCALL   SHOW_RESULT                 ;Show channel5 result
        MOV     A,      #6
        LCALL   SHOW_RESULT                 ;Show channel6 result
        MOV     A,      #7
        LCALL   SHOW_RESULT                 ;Show channel7 result

        SJMP    NEXT

;/*----------------------------
;Send ADC result to UART
;Input: ACC (ADC channel NO.)
;Output:-
;----------------------------*/
SHOW_RESULT:
        LCALL   SEND_DATA                   ;Show Channel NO.
        LCALL   GET_ADC_RESULT              ;Get high 8-bit ADC result
        LCALL   SEND_DATA                   ;Show result

;//if you want show 10-bit result, uncomment next 2 lines
;       MOV     A,      ADC_LOW2            ;Get low 2-bit ADC result
;       LCALL   SEND_DATA                   ;Show result
        RET

;/*----------------------------
;Read ADC conversion result
;Input: ACC (ADC channel NO.)
;Output:ACC (ADC result)
;----------------------------*/
GET_ADC_RESULT:
        ORL     A,      #ADC_POWER | ADC_SPEEDLL | ADC_START
        MOV     ADC_CONTR,    A             ;Start A/D conversion
        NOP                                 ;Must wait before inquiry
        NOP
        NOP
        NOP
WAIT:
        MOV     A,      ADC_CONTR                           ;Wait complete flag
        JNB     ACC.4,  WAIT        ;ADC_FLAG(ADC_CONTR.4)
        ANL     ADC_CONTR     ,#NOT ADC_FLAG                ;Clear ADC_FLAG
        MOV     A,      ADC_DATA                            ;Return ADC result
        RET
```

```
;/*----------------------------
;Initial ADC sfr
;----------------------------*/
INIT_ADC:
        MOV     A,      #0FFH
        MOV     P1,     A
        MOV     P1M0,   A
        MOV     P1M1,   A                               ;Set all P1 as Open-Drain mode
        MOV     ADC_DATA,       #0                      ;Clear previous result
        MOV     ADC_CONTR,      #ADC_POWER | ADC_SPEEDLL
        MOV     A,      #2                              ;ADC power-on and delay
        LCALL   DELAY
        RET
;/*----------------------------
;Initial UART
;----------------------------*/
INIT_UART:
        MOV     SCON,   #5AH                            ;8 bit data ,no parity bit
        MOV     TMOD,   #20H                            ;T1 as 8-bit auto reload
        MOV     A,      #-5                             ;Set Uart baudrate -(18432000/12/32/9600)
        MOV     TH1,    A                               ;Set T1 reload value
        MOV     TL1,    A
        SETB    TR1                                     ;T1 start running
        RET
;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
        JNB     TI,$                                    ;Wait for the previous data is sent
        CLR     TI                                      ;Clear TI flag
        MOV     SBUF,   A                               ;Send current data
        RET
;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        MOV     R2,     A
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

        END
```

第10章 STC12C5410AD系列单片机PCA/PWM应用

STC12C5410AD系列单片机集成了4路可编程计数器阵列(PCA)模块，可用于软件定时器、外部脉冲的捕捉、高速输出以及脉宽调制(PWM)输出。

10.1 与PCA/PWM应用有关的特殊功能寄存器

STC12C5410AD系列 1T 8051单片机 PCA/PWM特殊功能寄存器表 PCA/PWM SFRs

符号	描述	地址	位地址及其符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
CCON	PCA Control Register	D8H	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0	00xx,xx00
CMOD	PCA Mode Register	D9H	CIDL	-	-	-	-	CPS1	CPS0	ECF	0xxx,x000
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000,0000
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000,0000
CCAPM2	PCA Module 2 Mode Register	DCH	-	ECOM2	CAPP2	CAPN2	MAT2	TOG2	PWM2	ECCF2	x000,0000
CCAPM3	PCA Module 3 Mode Register	DDH	-	ECOM3	CAPP3	CAPN3	MAT3	TOG3	PWM3	ECCF3	x000,0000
CL	PCA Base Timer Low	E9H									0000,0000
CH	PCA Base Timer High	F9H									0000,0000
CCAP0L	PCA Module-0 Capture Register Low	EAH									0000,0000
CCAP0H	PCA Module-0 Capture Register High	FAH									0000,0000
CCAP1L	PCA Module-1 Capture Register Low	EBH									0000,0000
CCAP1H	PCA Module-1 Capture Register High	FBH									0000,0000
CCAP2L	PCA Module-2 Capture Register Low	ECH									0000,0000
CCAP2H	PCA Module-2 Capture Register High	FCH									0000,0000
CCAP3L	PCA Module-3 Capture Register Low	EDH									0000,0000
CCAP3H	PCA Module-3 Capture Register High	FDH									0000,0000
PCA_PWM0	PCA PWM Mode Auxiliary Register 0	F2H	-	-	-	-	-	-	EPC0H	EPC0L	xxxx,xx00
PCA_PWM1	PCA PWM Mode Auxiliary Register 1	F3H	-	-	-	-	-	-	EPC1H	EPC1L	xxxx,xx00
PCA_PWM2	PCA PWM Mode Auxiliary Register 2	F4H	-	-	-	-	-	-	EPC2H	EPC2L	xxxx,xx00
PCA_PWM3	PCA PWM Mode Auxiliary Register 3	F5H	-	-	-	-	-	-	EPC3H	EPC3L	xxxx,xx00

1. PCA工作模式寄存器CMOD

PCA工作模式寄存器的格式如下：

CMOD：PCA工作模式寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CMOD	D9H	name	CIDL	-	-	-	-	CPS1	CPS0	ECF

CIDL：空闲模式下是否停止PCA计数的控制位。

当CIDL=0时，空闲模式下PCA计数器继续工作；

当CIDL=1时，空闲模式下PCA计数器停止工作。

CPS1、CPS0：PCA计数脉冲源选择控制位。PCA计数脉冲选择如下表所示。

CPS1	CPS0	选择PCA/PWM时钟源输入
0	0	0，系统时钟，SYSclk/12
0	1	1，系统时钟，SYSclk/2
1	0	2，定时器0的溢出脉冲。由于定时器0可以工作在1T模式，所以可以达到计一个时钟就溢出，从而达到最高频率CPU工作时钟SYSclk。通过改变定时器0的溢出率，可以实现可调频率的PWM输出
1	1	3，ECI/P3.4脚输入的外部时钟（最大速率＝SYSclk/2）

ECF：PCA计数溢出中断使能位。

当ECF = 0时，禁止寄存器CCON中CF位的中断；

当ECF = 1时，允许寄存器CCON中CF位的中断。

2. PCA控制寄存器CCON

PCA控制寄存器的格式如下：

CCON：PCA控制控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCON	D8H	name	CF	CR	-	-	CCF3	CCF2	CCF1	CCF0

CF：PCA计数器阵列溢出标志位。当PCA计数器溢出时，CF由硬件置位。如果CMOD寄存器的ECF位置位， 则CF标志可用来产生中断。CF位可通过硬件或软件置位， 但只可通过软件清零。

CR：PCA计数器阵列运行控制位。该位通过软件置位， 用来起动PCA计数器阵列计数。该位通过软件清零， 用来关闭PCA计数器。

CCF3：PCA模块3中断标志。当出现匹配或捕获时该位由硬件置位。该位必须通过软件清零。

CCF2：PCA模块2中断标志。当出现匹配或捕获时该位由硬件置位。该位必须通过软件清零。

CCF1：PCA模块1中断标志。当出现匹配或捕获时该位由硬件置位。该位必须通过软件清零。

CCF0：PCA模块0中断标志。当出现匹配或捕获时该位由硬件置位。该位必须通过软件清零。

3. PCA比较/捕获寄存器CCAPMn (n=0,1,2,3)

PCA模块的比较/捕获寄存器的格式如下：

CCAPMn：PCA模块n(n=0, 1, 2, 3)的比较/捕获寄存器

SFR name	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCAPMn	name	-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn

B7：保留为将来之用。

ECOMn：允许比较器功能控制位。
当ECOMn＝1时，允许比较器功能。

CAPPn：正捕获控制位。
当CAPPn＝1时，允许上升沿捕获。

CAPNn：负捕获控制位。
当CAPNn＝1时，允许下降沿捕获。

MATn：匹配控制位。
当MATn＝1时，PCA计数值与模块的比较/捕获寄存器的值的匹配将置位CCON寄存器的中断标志位CCFn。

TOGn：翻转控制位。
当TOGn＝1时，工作在PCA高速输出模式，PCA计数器的值与模块的比较/捕获寄存器的值的匹配将使CCPn脚翻转。
(CCP0/P3.7, CCP1/P3.5, CCP2/P2.0, CCP3/P2.4)

PWMn：脉宽调节模式。
当PWMn＝1时，允许CCPn脚用作脉宽调节输出。
(CCP0/P3.7, CCP1/P3.5, CCP2/P2.0, CCP3/P2.4)

ECCFn：使能CCFn中断。使能寄存器CCON的比较/捕获标志CCFn， 用来产生中断。

PCA模块的工作模式设定表如下表所列：

PCA模块工作模式设定（CCAPMn寄存器，n = 0,1,2,3）

-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	模块功能
	0	0	0	0	0	0	0	无此操作
	X	1	0	0	0	0	X	16位捕获模式，由CCPn的上升沿触发
	X	0	1	0	0	0	X	16位捕获模式，由CCPn的下降沿触发
	X	1	1	0	0	0	X	16位捕获模式，由CCPn的跳变触发
	1	0	0	1	0	0	X	16位软件定时器
	1	0	0	1	1	0	X	16位高速输出
	1	0	0	0	0	1	0	8位PWM

4. PCA的16位计数器 — 低8位CL和高8位CH

CL和CH地址分别为E9H和F9H，复位值均为00H，用于保存PCA的装载值。

5. PCA捕捉/比较寄存器 — CCAPnL(低位字节)和CCAPnH(高位字节)

当PCA模块用于捕获或比较时，它们用于保存各个模块的16位捕捉计数值；当PCA模块用于PWM模式时，它们用来控制输出的占空比。其中，n=0、1、2、3，分别对应模块0、模块1、模块2和模块3。复位值均为00H。它们对应的地址分别为：

CCAP0L — EAH、CCAP0H — FAH：模块0的捕捉/比较寄存器。

CCAP1L — EBH、CCAP1H — FBH：模块1的捕捉/比较寄存器。

CCAP2L — ECH、CCAP2H — FCH：模块2的捕捉/比较寄存器。

CCAP3L — EDH、CCAP3H — FDH：模块3的捕捉/比较寄存器。

6. PCA模块PWM寄存器PCA_PWM0, PCA_PWM1, PCA_PWM2和PCA_PWM3

PCA模块0的PWM寄存器的格式如下：

PCA_PWM0 : PCA模块0的PWM寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM0	F2H	name	-	-	-	-	-	-	EPC0H	EPC0L

EPC0H：在PWM模式下，与CCAP0H组成9位数。

EPC0L：在PWM模式下，与CCAP0L组成9位数。

PCA模块1的PWM寄存器的格式如下：

PCA_PWM1 : PCA模块1的PWM寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM1	F3H	name	-	-	-	-	-	-	EPC1H	EPC1L

EPC1H：在PWM模式下，与CCAP1H组成9位数。

EPC1L：在PWM模式下，与CCAP1L组成9位数。

PCA模块2的PWM寄存器的格式如下：

PCA_PWM2 : PCA模块2的PWM寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM2	F4H	name	-	-	-	-	-	-	EPC2H	EPC2L

EPC2H：在PWM模式下，与CCAP2H组成9位数。

EPC2L：在PWM模式下，与CCAP2L组成9位数。

PCA模块3的PWM寄存器的格式如下：

PCA_PWM3 : PCA模块3的PWM寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM3	F5H	name	-	-	-	-	-	-	EPC3H	EPC3L

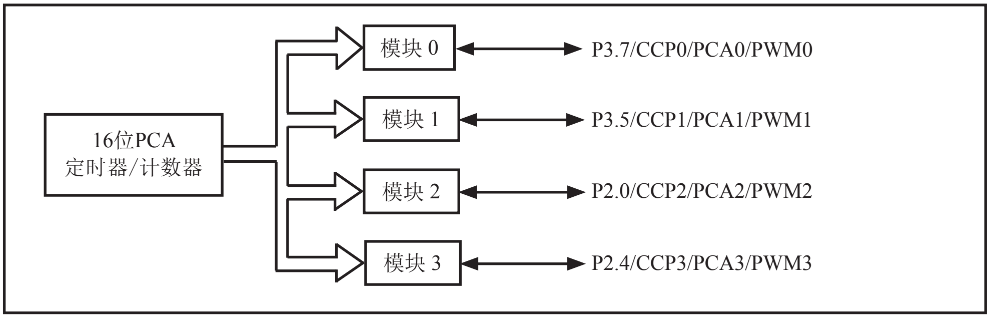
EPC3H：在PWM模式下，与CCAP3H组成9位数。

EPC3L：在PWM模式下，与CCAP3L组成9位数。

10.2 PCA/PWM模块的结构

STC12C5410AD系列单片机有4路可编程计数器阵列PCA/PWM

PCA含有一个特殊的16位定时器，有4个16位的捕获/比较模块与之相连，如下图所示。

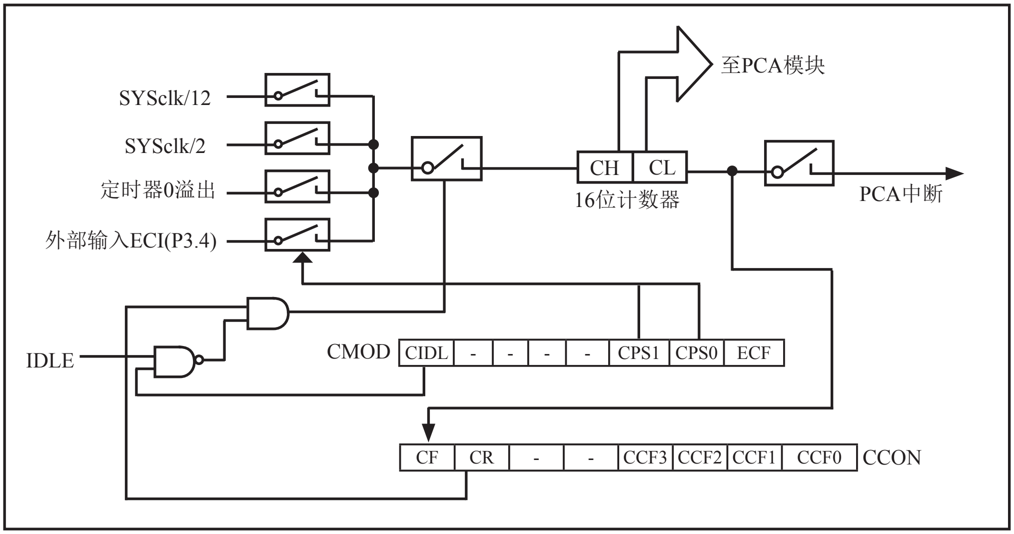


PCA模块结构

每个模块可编程工作在4种模式下：上升/下降沿捕获、软件定时器、高速输出或可调制脉冲输出。

STC12C5410AD系列：模块0连接到P3.7/CCP0/PCA0/PWM0
模块1连接到P3.5/CCP1/PCA1/PWM1
模块2连接到P2.0/CCP2/PCA2/PWM2
模块3连接到P2.4/CCP3/PCA3/PWM3

16位PCA定时器/计数器是2个模块的公共时间基准，其结构如下图所示。



PCA 定时器/计数器结构

寄存器CH和CL的内容是正在自由递增计数的16位PCA定时器的值。PCA定时器是4个模块的公共时间基准，可通过编程工作在：1/12系统时钟、1/2系统时钟、定时器0溢出或ECI脚的输入（P3.4）。定时器的计数源由CMOD特殊功能寄存器中的CPS1和CPS0位来确定（见CMOD特殊功能寄存器说明）。

CMOD特殊功能寄存器还有2个位与PCA相关。它们分别是：CIDL，空闲模式下允许停止PCA；ECF，置位时，使能PCA中断，当PCA定时器溢出将PCA计数溢出标志CF（CCON.7）置位。

CCON特殊功能寄存器包含PCA的运行控制位（CR）和PCA定时器标志（CF）以及各个模块的标志（CCF3/CCF2/CCF1/CCF0）。通过软件置位CR位（CCON.6）来运行PCA。CR位被清零时PCA关闭。当PCA计数器溢出时，CF位（CCON.7）置位,如果CMOD寄存器的ECF位置位,就产生中断。CF位只可通过软件清除。CCON寄存器的位0～3是PCA各个模块的标志（位0对应模块0，位1对应模块1，位2对应模块2，位3对应模块3），当发生匹配或比较时由硬件置位。这些标志也只能通过软件清除。所有模块共用一个中断向量。

PCA的每个模块都对应一个特殊功能寄存器。它们分别是：模块0对应CCAPM0，模块1对应CCAPM1，模块2对应CCAPM2，模块3对应CCAPM3。特殊功能寄存器包含了相应模块的工作模式控制位。

当模块发生匹配或比较时，ECCFn位（CCAPMn.0，n＝0，1，2,3由工作的模块决定）使能CCON特殊功能寄存器的CCFn标志来产生中断。

PWM（CCAPMn.1）用来使能脉宽调制模式。

当PCA计数值与模块的捕获/比较寄存器的值相匹配时，如果TOG位（CCAPMn.2）置位，模块的CEXn输出将发生翻转。

当PCA计数值与模块的捕获/比较寄存器的值相匹配时，如果匹配位MATn（CCAPMn.3）置位，CCON寄存器的CCFn位将被置位。

CAPNn（CCAPMn.4）和CAPPn（CCAPMn.5）用来设置捕获输入的有效沿。CAPNn位使能下降沿有效，CAPPn位使能上升沿有效。如果两位都置位，则两种跳变沿都被使能，捕获可在两种跳变沿产生。

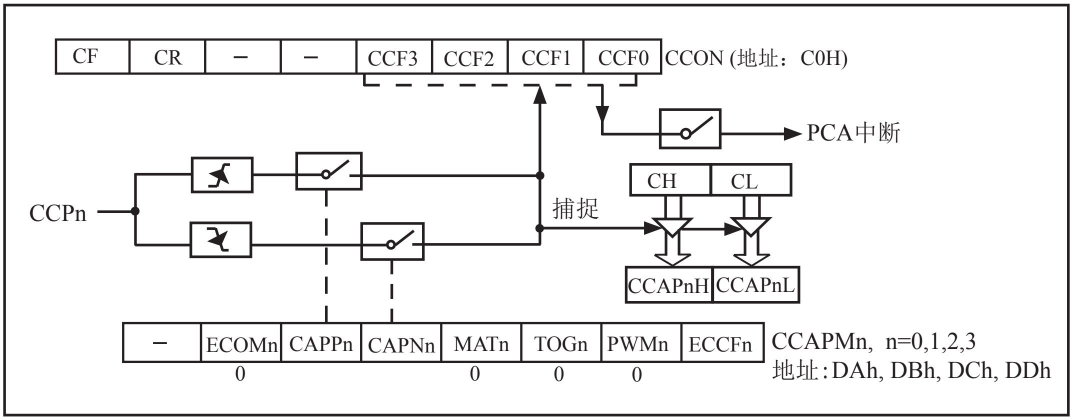
通过置位CCAPMn寄存器的ECOMn位（CCAPMn.6）来使能比较器功能。

每个PCA模块还对应另外两个寄存器，CCAPnH和CCAPnL。当出现捕获或比较时，它们用来保存16位的计数值。当PCA模块用在PWM模式中时，它们用来控制输出的占空比。

10.3 PCA模块的工作模式

10.3.1 捕获模式

PCA模块工作于捕获模式的结构图如下图所示。要使一个PCA模块工作在捕获模式，寄存器CCAPMn的两位（CAPNn和CAPPn）或其中任何一位必须置1。PCA模块工作于捕获模式时，对模块的外部CCPn输入（CCP0/P3.7,CCP1/P3.5,CCP2/P2.0,CCP3/P2.4）的跳变进行采样。当采样到有效跳变时，PCA硬件就将PCA计数器阵列寄存器（CH和CL）的值装载到模块的捕获寄存器中（CCAPnL和CCAPnH）。

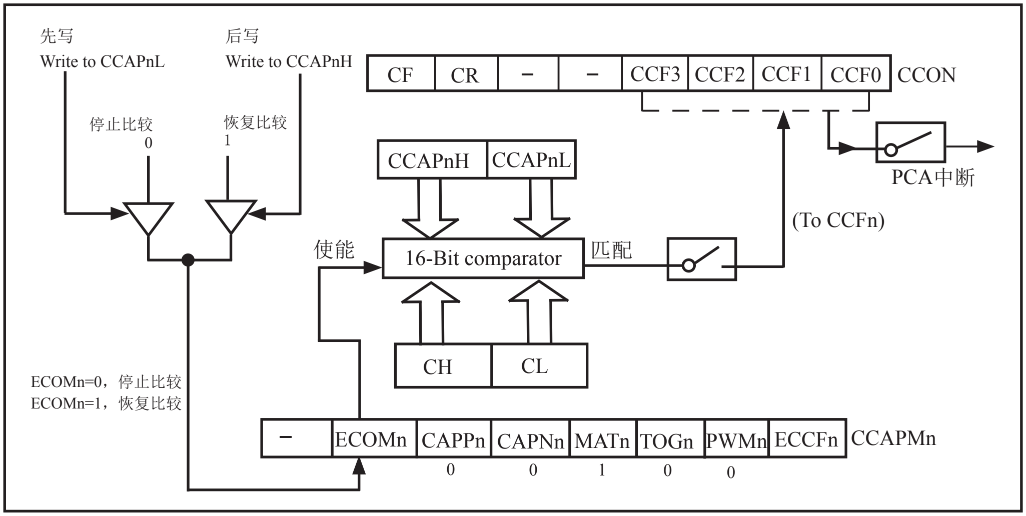


PCA Capture Mode (PCA捕获模式图)

如果CCON特殊功能寄存器中的位CCFn和CCAPMn特殊功能寄存器中的位ECCFn位被置位，将产生中断。可在中断服务程序中判断哪一个模块产生了中断，并注意中断标志位的软件清零问题。

10.3.2 16位软件定时器模式

16位软件定时器模式结构图如下图所示。



PCA Software Timer Mode / PCA模块的16位软件定时器模式/PCA比较模式

通过置位CCAPMn寄存器的ECOMn和MATn位，可使PCA模块用作软件定时器（上图）。PCA定时器的值与模块捕获寄存器的值相比较，当两者相等时，如果位CCFn(在CCON特殊功能寄存器中)和位ECCFn（在CCAPMn特殊功能寄存器中）都置位，将产生中断。

[CH,CL]每隔一定的时间自动加1，时间间隔取决于选择的时钟源。例如，当选择的时钟源为SYSclk/12，每12个时钟周期[CH,CL]加1。当[CH,CL]增加到等于[CCAPnH, CCAPnL]时，CCFn=1，产生中断请求。如果每次PCA模块中断后，在中断服务程序中断给[CCAPnH, CCAPnL]增加一个相同的数值，那么下次中断来临的间隔时间T也是相同的，从而实现了定时功能。定时时间的长短取决于时钟源的选择以及PCA计数器计数值的设置。下面举例说明PCA计数器计数值的计算方法。

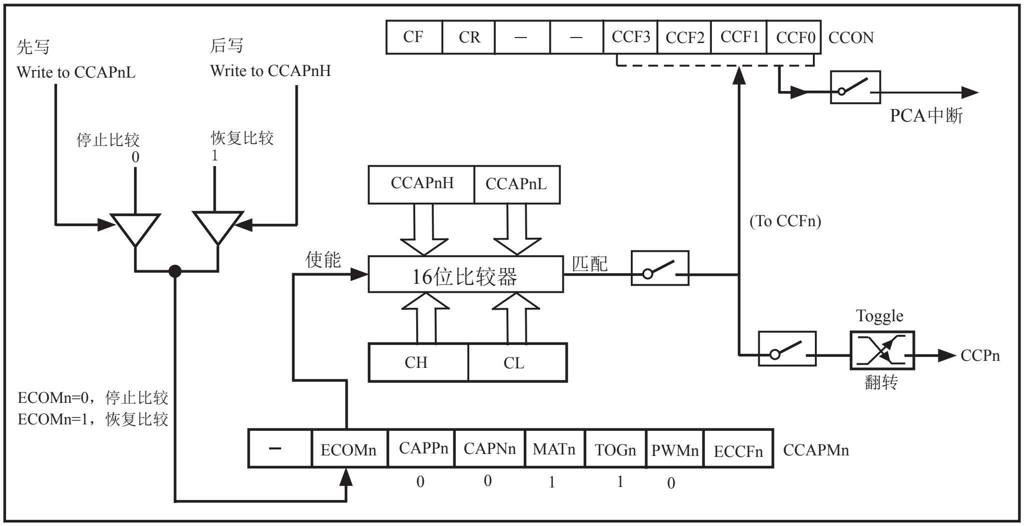
假设，系统时钟频率SYSclk = 18.432MHz, 选择的时钟源为SYSclk/12, 定时时间T为5ms, 则PCA计数器计数值为：

PCA计数器的计数值 = T / ((1 / SYSclk)×12) = 0.005 / ((1 / 18432000)×12) = 7680 (10进制数)
= 1E00H (16进制数)

也就是说，PCA计时器计数1E00H次，定时时间才是5ms，这也就是每次给[CCAPnH, CCAPnL]增加的数值(步长)。

10.3.3 高速输出模式

该模式中(下图)，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CEXn输出将发生翻转。要激活高速输出模式，CCAPMn寄存器的TOGn,MATn和ECOMn位必须都置位。



PCA High-Speed Output Mode / PCA 高速输出模式

CCAPnL的值决定了PCA模块n的输出脉冲频率。当PCA时钟源是SYSclk/2时，输出脉冲的频率F为：

$$f = SYSclk / (4 \times CCAPnL)$$

其中，SYSclk为系统时钟频率。由此，可以得到CCAPnL的值CCAPnL = SYSclk / (4×f).

如果计算出的结果不是整数，则进行四舍五入取整，即

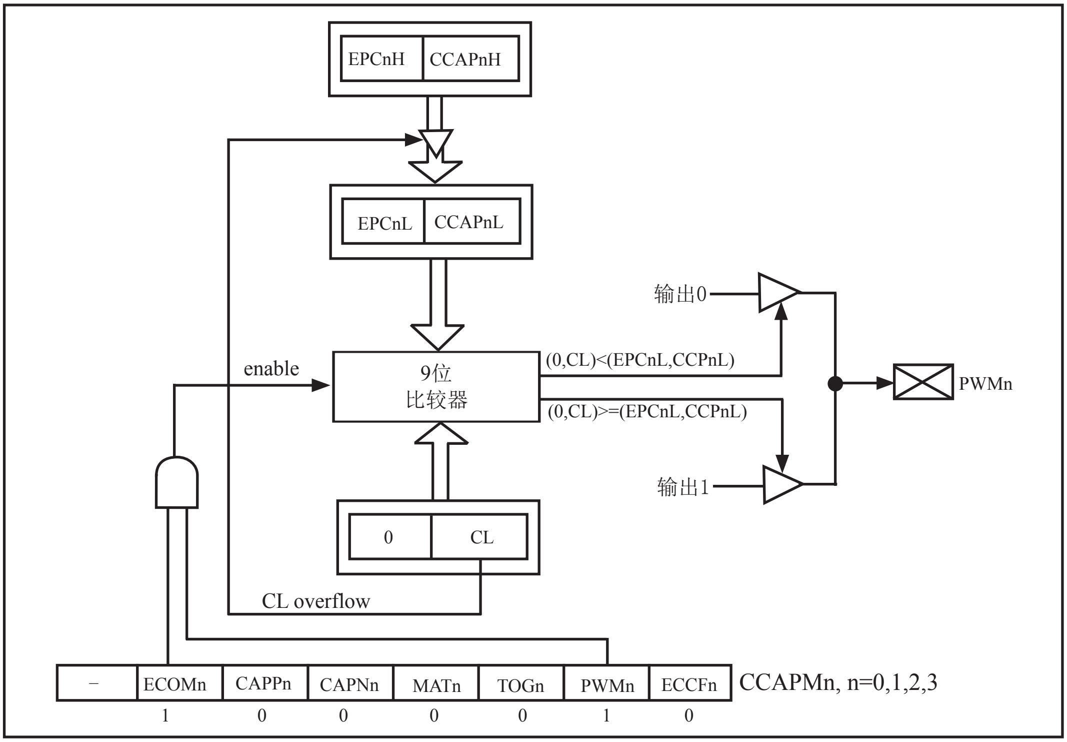
$$CCAPnL = INT (SYSclk / (4 \times f) + 0.5)$$

其中，INT( )为取整运算，直接去掉小数。例如，假设SYSclk = 20MHz，要求PCA高速脉冲输出125kHz的方波，则CCAPnL中的值应为：

$$CCAPnL = INT (20000000 / (4 \times 125000) + 0.5) = INT (40 + 0.5) = 40 = 28H$$

10.3.4 脉宽调节模式(PWM)

脉宽调制(PWM，Pulse Width Modulation)是一种使用程序来控制波形占空比、周期、相位波形的技术，在三相电机驱动、D/A转换等场合有广泛的应用。STC12C5410AD系列单片机的PCA模块可以通过程序设定，使其工作于8位PWM模式。PWM模式的结构如下图所示。



PCA PWM mode / 可调制脉冲宽度输出模式

所有PCA模块都可用作PWM输出（上图）。输出频率取决于PCA定时器的时钟源。

由于所有模块共用仅有的PCA定时器，所有它们的输出频率相同。各个模块的输出占空比是独立变化的，与使用的捕获寄存器[EPCnL, CCAPnL]有关。当寄存器CL的值小于[EPCnL, CCAPnL]时，输出为低；当寄存器CL的值等于或大于[EPCnL, CCAPnL]时，输出为高。当CL的值由FF变为00溢出时，[EPCnH, CCAPnH]的内容装载到[EPCnL, CCAPnL]中。这样就可实现无干扰地更新PWM。要使能PWM模式，模块CCAPMn寄存器的PWMn和ECOMn位必须置位。

由于PWM是8位的，所以： $\text{PWM的频率} = \dfrac{\text{PCA时钟输入源频率}}{256}$

PCA时钟输入源可以从以下4种中选择一种：SYSclk/12，SYSclk/2, 定时器0的溢出，ECI/P3.4输入。

举例：要求PWM输出频率为38KHz，选SYSclk/2为PCA/PWM时钟输入源，求出SYSclk的值

由计算公式38000=SYSclk/2/256，得到外部时钟频率SYSclk=38000 x 256 x 2=19,456,000

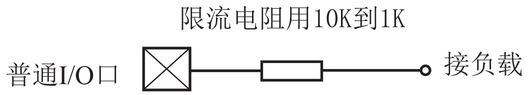
如果要实现可调频率的PWM输出, 可选择定时器0的溢出率或者ECI脚的输入作为PCA/PWM的时钟输入源

当EPCnL = 0及ECCAPnL = 00H时, PWM固定输出高

当EPCnL = 1及CCAPnL = 0FFH时, PWM固定输出低

当某个I/O口作为PWM使用时, 该口的状态:

PWM之前口的状态	PWM输出时口的状态
弱上拉/准双向	强推挽输出/强上拉输出，要加输出限流电阻1K-10K
强推挽输出/强上拉输出	强推挽输出/强上拉输出，要加输出限流电阻1K-10K
仅为输入/高阻	PWM无效
开漏	开漏



10.4 用PCA功能扩展外部中断的示例程序(C程序和汇编程序)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 用PCA功能扩展外部中断 -----------*/
/* 如果要在程序中使用或在文章中引用该程序, -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the PCA */
sbit    EPCAI       = IE^6;
sfr     CCON        = 0xD8;         //PCA control register
sbit    CCF0        = CCON^0;       //PCA module-0 interrupt flag
sbit    CCF1        = CCON^1;       //PCA module-1 interrupt flag
sbit    CR          = CCON^6;       //PCA timer run control bit
sbit    CF          = CCON^7;       //PCA timer overflow flag
sfr     CMOD        = 0xD9;         //PCA mode register
sfr     CL          = 0xE9;         //PCA base timer LOW
sfr     CH          = 0xF9;         //PCA base timer HIGH
sfr     CCAPM0      = 0xDA;         //PCA module-0 mode register
sfr     CCAP0L      = 0xEA;         //PCA module-0 capture register LOW
sfr     CCAP0H      = 0xFA;         //PCA module-0 capture register HIGH
sfr     CCAPM1      = 0xDB;         //PCA module-1 mode register
sfr     CCAP1L      = 0xEB;         //PCA module-1 capture register LOW
sfr     CCAP1H      = 0xFB;         //PCA module-1 capture register HIGH
sfr     CCAPM2      = 0xDC;         //PCA module-2 mode register
sfr     CCAP2L      = 0xEC;         //PCA module-2 capture register LOW
sfr     CCAP2H      = 0xFC;         //PCA module-2 capture register HIGH
sfr     CCAPM3      = 0xDD;         //PCA module-3 mode register
sfr     CCAP3L      = 0xED;         //PCA module-3 capture register LOW
sfr     CCAP3H      = 0xFD;         //PCA module-3 capture register HIGH
sfr     PCAPWM0     = 0xF2;
sfr     PCAPWM1     = 0xF3;
sfr     PCAPWM2     = 0xF4;
sfr     PCAPWM3     = 0xF5;
```

```
sbit      PCA_LED        =  P1^0;                        //PCA test LED

void PCA_isr() interrupt 7 using 1
{
          CCF0 = 0;                        //Clear interrupt flag
          PCA_LED = !PCA_LED;              //toggle the test pin while CEX0(P1.3) have a falling edge
}

void main()
{
          CCON = 0;                 //Initial PCA control register
                                    //PCA timer stop running
                                    //Clear CF flag
                                    //Clear all module interrupt flag
          CL = 0;                   //Reset PCA base timer
          CH = 0;
          CMOD = 0x00;              //Set PCA timer clock source as Fosc/12
                                    //Disable PCA timer overflow interrupt
          CCAPM0 = 0x11;            //PCA module-0 capture by a negative tigger on CEX0(P1.3)
                                    //and enable PCA interrupt
//        CCAPM0 = 0x21;            //PCA module-0 capture by a rising edge on CEX0(P1.3)
                                    //and enable PCA interrupt
//        CCAPM0 = 0x31;            //PCA module-0 capture by a transition (falling/rising edge)
                                    //on CEX0(P1.3) and enable PCA interrupt

          CR = 1;                   //PCA timer start run
          EPCAI = 1;
          EA = 1;

          while (1);
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示  STC 1T 系列单片机  用PCA功能扩展外部中断 -----------*/
/* 如果要在程序中使用或在文章中引用该程序，  ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/
;/*Declare SFR associated with the PCA */
EPCAI       BIT     IE.6

CCON        EQU     0D8H            ;PCA control register
CCF0        BIT     CCON.0          ;PCA module-0 interrupt flag
CCF1        BIT     CCON.1          ;PCA module-1 interrupt flag
CR          BIT     CCON.6          ;PCA timer run control bit
CF          BIT     CCON.7          ;PCA timer overflow flag
CMOD        EQU     0D9H            ;PCA mode register
CL          EQU     0E9H            ;PCA base timer LOW
CH          EQU     0F9H            ;PCA base timer HIGH
CCAPM0      EQU     0DAH            ;PCA module-0 mode register
CCAP0L      EQU     0EAH            ;PCA module-0 capture register LOW
CCAP0H      EQU     0FAH            ;PCA module-0 capture register HIGH
CCAPM1      EQU     0DBH            ;PCA module-1 mode register
CCAP1L      EQU     0EBH            ;PCA module-1 capture register LOW
CCAP1H      EQU     0FBH            ;PCA module-1 capture register HIGH
CCAPM2      EQU     0DCH            ;PCA module-2 mode register
CCAP2L      EQU     0ECH            ;PCA module-2 capture register LOW
CCAP2H      EQU     0FCH            ;PCA module-2 capture register HIGH
CCAPM3      EQU     0DDH            ;PCA module-3 mode register
CCAP3L      EQU     0EDH            ;PCA module-3 capture register LOW
CCAP3H      EQU     0FDH            ;PCA module-3 capture register HIGH

PCA_LED     BIT     P1.0            ;PCA test LED

;----------------------------------------
            ORG     0000H
            LJMP    MAIN
```

```
        ORG     0033H
PCA_ISR:
        CLR     CCF0                        ;Clear interrupt flag
        CPL     PCA_LED                     ;toggle the test pin while CEX0(P1.3) have a falling edge
        RETI

;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0                  ;Initial PCA control register
                                            ;PCA timer stop running
                                            ;Clear CF flag
                                            ;Clear all module interrupt flag
        CLR     A                           ;
        MOV     CL,     A                   ;Reset PCA base timer
        MOV     CH,     A                   ;
        MOV     CMOD,   #00H                ;Set PCA timer clock source as Fosc/12
                                            ;Disable PCA timer overflow interrupt
        MOV     CCAPM0,#11H                 ;PCA module-0 capture by a falling edge on CEX0(P1.3)
                                            ;and enable PCA interrupt
;       MOV     CCAPM0,#21H                 ;PCA module-0 capture by a rising edge on CEX0(P1.3)
                                            ;and enable PCA interrupt
;       MOV     CCAPM0,#31H                 ;PCA module-0 capture by a transition (falling/rising edge)
                                            ;on CEX0(P1.3) and enable PCA interrupt
;-------------------------------
        SETB    CR                          ;PCA timer start run
        SETB    EPCAI
        SETB    EA

        SJMP    $
;-----------------------------------------
        END
```

10.5 用PCA功能实现定时器的示例程序(C程序和汇编程序)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 用PCA功能实现16位定时器 --------*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

#define   FOSC    18432000L
#define   T100Hz  (FOSC / 12 / 100)

typedef   unsigned char    BYTE;
typedef   unsigned int     WORD;

/*Declare SFR associated with the PCA */
sbit      EPCAI        =   IE^6;
sfr       CCON         =   0xD8;          //PCA control register
sbit      CCF0         =   CCON^0;        //PCA module-0 interrupt flag
sbit      CCF1         =   CCON^1;        //PCA module-1 interrupt flag
sbit      CR           =   CCON^6;        //PCA timer run control bit
sbit      CF           =   CCON^7;        //PCA timer overflow flag
sfr       CMOD         =   0xD9;          //PCA mode register
sfr       CL           =   0xE9;          //PCA base timer LOW
sfr       CH           =   0xF9;          //PCA base timer HIGH
sfr       CCAPM0       =   0xDA;          //PCA module-0 mode register
sfr       CCAP0L       =   0xEA;          //PCA module-0 capture register LOW
sfr       CCAP0H       =   0xFA;          //PCA module-0 capture register HIGH
sfr       CCAPM1       =   0xDB;          //PCA module-1 mode register
sfr       CCAP1L       =   0xEB;          //PCA module-1 capture register LOW
sfr       CCAP1H       =   0xFB;          //PCA module-1 capture register HIGH
sfr       CCAPM2       =   0xDC;          //PCA module-2 mode register
sfr       CCAP2L       =   0xEC;          //PCA module-2 capture register LOW
sfr       CCAP2H       =   0xFC;          //PCA module-2 capture register HIGH
sfr       CCAPM3       =   0xDD;          //PCA module-3 mode register
sfr       CCAP3L       =   0xED;          //PCA module-3 capture register LOW
sfr       CCAP3H       =   0xFD;          //PCA module-3 capture register HIGH
```

```
sbit    PCA_LED     =   P1^0;                   //PCA test LED

BYTE    cnt;
WORD    value;

void PCA_isr() interrupt 7 using 1
{
        CCF0 = 0;                           //Clear interrupt flag
        CCAP0L = value;
        CCAP0H = value >> 8;                //Update compare value
        value += T100Hz;
        if (cnt-- == 0)
        {
                cnt = 100;                  //Count 100 times
                PCA_LED = !PCA_LED;         //Flash once per second
        }
}

void main()
{
        CCON = 0;                           //Initial PCA control register
                                            //PCA timer stop running
                                            //Clear CF flag
                                            //Clear all module interrupt flag
        CL = 0;                             //Reset PCA base timer
        CH = 0;
        CMOD = 0x00;                        //Set PCA timer clock source as Fosc/12
                                            //Disable PCA timer overflow interrupt
        value = T100Hz;
        CCAP0L = value;
        CCAP0H = value >> 8;                //Initial PCA module-0
        value += T100Hz;
        CCAPM0 = 0x49;                      //PCA module-0 work in 16-bit timer mode
                                            //and enable PCA interrupt

        CR = 1;                             //PCA timer start run
        EPCAI = 1;
        EA = 1;
        cnt = 0;

        while (1);
}
```

2. 汇编程序：

```
/*-----------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 用PCA功能实现16位定时器 --------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*-----------------------------------------------------------------------------------*/

T100Hz          EQU     3C00H                   ;(18432000 / 12 / 100)

;/*Declare SFR associated with the PCA */
EPCAI           BIT     IE.6

CCON            EQU     0D8H                    ;PCA control register
CCF0            BIT     CCON.0                  ;PCA module-0 interrupt flag
CCF1            BIT     CCON.1                  ;PCA module-1 interrupt flag
CR              BIT     CCON.6                  ;PCA timer run control bit
CF              BIT     CCON.7                  ;PCA timer overflow flag
CMOD            EQU     0D9H                    ;PCA mode register
CL              EQU     0E9H                    ;PCA base timer LOW
CH              EQU     0F9H                    ;PCA base timer HIGH
CCAPM0          EQU     0DAH                    ;PCA module-0 mode register
CCAP0L          EQU     0EAH                    ;PCA module-0 capture register LOW
CCAP0H          EQU     0FAH                    ;PCA module-0 capture register HIGH
CCAPM1          EQU     0DBH                    ;PCA module-1 mode register
CCAP1L          EQU     0EBH                    ;PCA module-1 capture register LOW
CCAP1H          EQU     0FBH                    ;PCA module-1 capture register HIGH
CCAPM2          EQU     0DCH                    ;PCA module-2 mode register
CCAP2L          EQU     0ECH                    ;PCA module-2 capture register LOW
CCAP2H          EQU     0FCH                    ;PCA module-2 capture register HIGH
CCAPM3          EQU     0DDH                    ;PCA module-3 mode register
CCAP3L          EQU     0EDH                    ;PCA module-3 capture register LOW
CCAP3H          EQU     0FDH                    ;PCA module-3 capture register HIGH

PCA_LED         BIT     P1.0                    ;PCA test LED

CNT             EQU     20H
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN
```

```
        ORG     0033H
        LJMP    PCA_ISR
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,     #3FH                    ;Initial stack point
        MOV     CCON,   #0                      ;Initial PCA control register
                                                ;PCA timer stop running
                                                ;Clear CF flag
                                                ;Clear all module interrupt flag
        CLR     A                               ;
        MOV     CL,     A                       ;Reset PCA base timer
        MOV     CH,     A                       ;
        MOV     CMOD,   #00H                    ;Set PCA timer clock source as Fosc/12
                                                ;Disable PCA timer overflow interrupt
;-------------------------------
        MOV     CCAP0L,#LOW T100Hz              ;
        MOV     CCAP0H,#HIGH T100Hz             ;Initial PCA module-0
        MOV     CCAPM0,#49H         ;PCA module-0 work in 16-bit timer mode and enable PCA interrupt
;-------------------------------
        SETB    CR                              ;PCA timer start run
        SETB    EPCAI
        SETB    EA
        MOV     CNT,    #100

        SJMP    $
;-----------------------------------------
PCA_ISR:
        PUSH    PSW
        PUSH    ACC
        CLR     CCF0                            ;Clear interrupt flag
        MOV     A,      CCAP0L
        ADD     A,      #LOW T100Hz             ;Update compare value
        MOV     CCAP0L,A
        MOV     A,      CCAP0H
        ADDC    A,      #HIGH T100Hz
        MOV     CCAP0H,A
        DJNZ    CNT,    PCA_ISR_EXIT            ;count 100 times
        MOV     CNT,    #100
        CPL     PCA_LED                         ;Flash once per second
PCA_ISR_EXIT:
        POP     ACC
        POP     PSW
        RETI
;-----------------------------------------
        END
```

10.6 PCA输出高速脉冲的示例程序(C程序和汇编程序)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 PCA输出高速脉冲 --------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

#define   FOSC      18432000L
#define   T100KHz (FOSC / 4 / 100000)

typedef   unsigned char     BYTE;
typedef   unsigned int      WORD;

/*Declare SFR associated with the PCA */
sbit      EPCAI          =  IE^6;

sfr       CCON           =  0xD8;                    //PCA control register
sbit      CCF0           =  CCON^0;                  //PCA module-0 interrupt flag
sbit      CCF1           =  CCON^1;                  //PCA module-1 interrupt flag
sbit      CR             =  CCON^6;                  //PCA timer run control bit
sbit      CF             =  CCON^7;                  //PCA timer overflow flag
sfr       CMOD           =  0xD9;                    //PCA mode register
sfr       CL             =  0xE9;                    //PCA base timer LOW
sfr       CH             =  0xF9;                    //PCA base timer HIGH
sfr       CCAPM0         =  0xDA;                    //PCA module-0 mode register
sfr       CCAP0L         =  0xEA;                    //PCA module-0 capture register LOW
sfr       CCAP0H         =  0xFA;                    //PCA module-0 capture register HIGH
sfr       CCAPM1         =  0xDB;                    //PCA module-1 mode register
sfr       CCAP1L         =  0xEB;                    //PCA module-1 capture register LOW
sfr       CCAP1H         =  0xFB;                    //PCA module-1 capture register HIGH
sfr       CCAPM2         =  0xDC;                    //PCA module-2 mode register
sfr       CCAP2L         =  0xEC;                    //PCA module-2 capture register LOW
sfr       CCAP2H         =  0xFC;                    //PCA module-2 capture register HIGH
sfr       CCAPM3         =  0xDD;                    //PCA module-3 mode register
sfr       CCAP3L         =  0xED;                    //PCA module-3 capture register LOW
sfr       CCAP3H         =  0xFD;                    //PCA module-3 capture register HIGH
```

```
sbit    PCA_LED     =  P1^0;                     //PCA test LED

BYTE    cnt;
WORD    value;

void PCA_isr( ) interrupt 7 using 1
{
        CCF0 = 0;                       //Clear interrupt flag
        CCAP0L = value;
        CCAP0H = value >> 8;            //Update compare value
        value += T100KHz;
}

void main()
{
        CCON = 0;                       //Initial PCA control register
                                        //PCA timer stop running
                                        //Clear CF flag
                                        //Clear all module interrupt flag
        CL = 0;                         //Reset PCA base timer
        CH = 0;
        CMOD = 0x02;                    //Set PCA timer clock source as Fosc/2
                                        //Disable PCA timer overflow interrupt
        value = T100KHz;
        CCAP0L = value;                 //P1.3 output 100KHz square wave
        CCAP0H = value >> 8;            //Initial PCA module-0
        value += T100KHz;
        CCAPM0 = 0x4d;                  //PCA module-0 work in 16-bit timer mode
                                        //and enable PCA interrupt, toggle the output pin CEX0(P1.3)

        CR = 1;                         //PCA timer start run
        EPCAI = 1;
        EA = 1;
        cnt = 0;

        while (1);
}
```

2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 PCA输出高速脉冲 --------------------*/
/* 如果要在程序中使用或在文章中引用该程序， -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

T100KHz         EQU     2EH                 ;(18432000 / 4 / 100000)

;/*Declare SFR associated with the PCA */
CCON            EQU     0D8H                ;PCA control register
CCF0            BIT     CCON.0              ;PCA module-0 interrupt flag
CCF1            BIT     CCON.1              ;PCA module-1 interrupt flag
CR              BIT     CCON.6              ;PCA timer run control bit
CF              BIT     CCON.7              ;PCA timer overflow flag
CMOD            EQU     0D9H                ;PCA mode register
CL              EQU     0E9H                ;PCA base timer LOW
CH              EQU     0F9H                ;PCA base timer HIGH
CCAPM0          EQU     0DAH                ;PCA module-0 mode register
CCAP0L          EQU     0EAH                ;PCA module-0 capture register LOW
CCAP0H          EQU     0FAH                ;PCA module-0 capture register HIGH
CCAPM1          EQU     0DBH                ;PCA module-1 mode register
CCAP1L          EQU     0EBH                ;PCA module-1 capture register LOW
CCAP1H          EQU     0FBH                ;PCA module-1 capture register HIGH
CCAPM2          EQU     0DCH                ;PCA module-2 mode register
CCAP2L          EQU     0ECH                ;PCA module-2 capture register LOW
CCAP2H          EQU     0FCH                ;PCA module-2 capture register HIGH
CCAPM3          EQU     0DDH                ;PCA module-3 mode register
CCAP3L          EQU     0EDH                ;PCA module-3 capture register LOW
CCAP3H          EQU     0FDH                ;PCA module-3 capture register HIGH
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN

        ORG     0033H
PCA_ISR:
        PUSH    PSW
        PUSH    ACC
        CLR     CCF0                        ;Clear interrupt flag
```

```
        MOV     A,      CCAP0L
        ADD     A,      #T100KHz                ;Update compare value
        MOV     CCAP0L,A
        CLR     A
        ADDC    A,      CCAP0H
        MOV     CCAP0H,A

PCA_ISR_EXIT:
        POP     ACC
        POP     PSW
        RETI
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
        CLR     A                       ;
        MOV     CL,     A               ;Reset PCA base timer
        MOV     CH,     A               ;
        MOV     CMOD,   #02H            ;Set PCA timer clock source as Fosc/2
                                        ;Disable PCA timer overflow interrupt
;-------------------------------
        MOV     CCAP0L,#T100KHz         ;P1.3 output 100KHz square wave
        MOV     CCAP0H,#0               ;Initial PCA module-0
        MOV     CCAPM0,#4dH             ;PCA module-0 work in 16-bit timer mode
                                        ;and enable PCA interrupt, toggle the output pin CEX0(P1.3)
;-------------------------------
        SETB    CR                      ;PCA timer start run
        SETB    EA

        SJMP    $
;-----------------------------------------
        END
```

10.7 PCA输出PWM的示例程序(C程序和汇编程序)

1. C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示 STC 1T 系列单片机 PCA输出PWM --------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

#define   FOSC    18432000L

typedef   unsigned char    BYTE;
typedef   unsigned int     WORD;

/*Declare SFR associated with the PCA */
sfr     CCON        =  0xD8;           //PCA control register
sbit    CCF0        =  CCON^0;         //PCA module-0 interrupt flag
sbit    CCF1        =  CCON^1;         //PCA module-1 interrupt flag
sbit    CR          =  CCON^6;         //PCA timer run control bit
sbit    CF          =  CCON^7;         //PCA timer overflow flag
sfr     CMOD        =  0xD9;           //PCA mode register
sfr     CL          =  0xE9;           //PCA base timer LOW
sfr     CH          =  0xF9;           //PCA base timer HIGH
sfr     CCAPM0      =  0xDA;           //PCA module-0 mode register
sfr     CCAP0L      =  0xEA;           //PCA module-0 capture register LOW
sfr     CCAP0H      =  0xFA;           //PCA module-0 capture register HIGH
sfr     CCAPM1      =  0xDB;           //PCA module-1 mode register
sfr     CCAP1L      =  0xEB;           //PCA module-1 capture register LOW
sfr     CCAP1H      =  0xFB;           //PCA module-1 capture register HIGH
sfr     CCAPM2      =  0xDC;           //PCA module-2 mode register
sfr     CCAP2L      =  0xEC;           //PCA module-2 capture register LOW
sfr     CCAP2H      =  0xFC;           //PCA module-2 capture register HIGH
sfr     CCAPM3      =  0xDD;           //PCA module-3 mode register
sfr     CCAP3L      =  0xED;           //PCA module-3 capture register LOW
sfr     CCAP3H      =  0xFD;           //PCA module-3 capture register HIGH
sfr     PCAPWM0     =  0xF2;
sfr     PCAPWM1     =  0xF3;
sfr     PCAPWM2     =  0xF4;
sfr     PCAPWM3     =  0xF5;
```

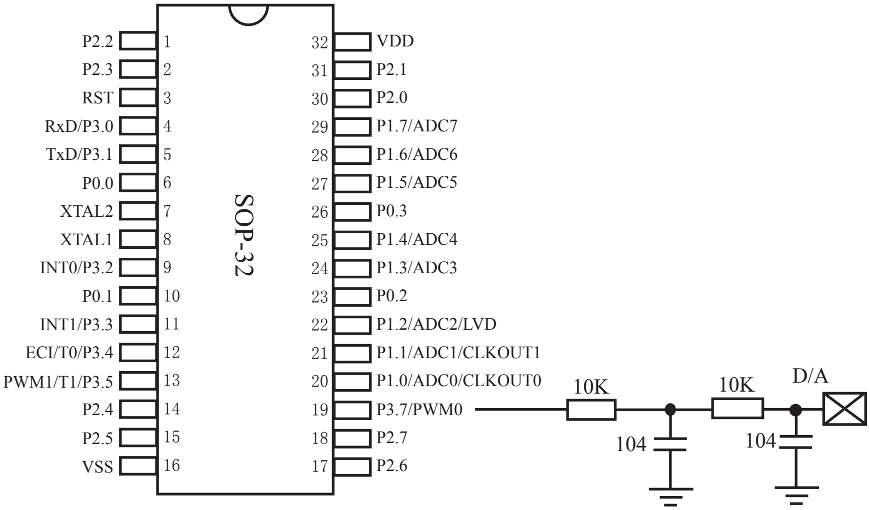
```
void main()
{
        CCON = 0;                       //Initial PCA control register
                                        //PCA timer stop running
                                        //Clear CF flag
                                        //Clear all module interrupt flag
        CL = 0;                         //Reset PCA base timer
        CH = 0;
        CMOD = 0x02;                    //Set PCA timer clock source as Fosc/2
                                        //Disable PCA timer overflow interrupt
        CCAP0H = CCAP0L = 0x80;         //PWM0 port output 50% duty cycle square wave
        CCAPM0 = 0x42;                  //PCA module-0 work in 8-bit PWM mode
                                        //and no PCA interrupt

        CCAP1H = CCAP1L = 0xff;         //PWM1 port output 0% duty cycle square wave
        PCAPWM1 = 0x03;
        CCAPM1 = 0x42;                  //PCA module-1 work in 8-bit PWM mode
                                        //and no PCA interrupt

        CR = 1;                         //PCA timer start run

        while (1);
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示  STC 1T 系列单片机  PCA输出PWM -------------------------*/
/* 如果要在程序中使用或在文章中引用该程序，  ----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
CCON        EQU     0D8H            ;PCA control register
CCF0        BIT     CCON.0          ;PCA module-0 interrupt flag
CCF1        BIT     CCON.1          ;PCA module-1 interrupt flag
CR          BIT     CCON.6          ;PCA timer run control bit
CF          BIT     CCON.7          ;PCA timer overflow flag
CMOD        EQU     0D9H            ;PCA mode register
CL          EQU     0E9H            ;PCA base timer LOW
CH          EQU     0F9H            ;PCA base timer HIGH
CCAPM0      EQU     0DAH            ;PCA module-0 mode register
CCAP0L      EQU     0EAH            ;PCA module-0 capture register LOW
CCAP0H      EQU     0FAH            ;PCA module-0 capture register HIGH
CCAPM1      EQU     0DBH            ;PCA module-1 mode register
CCAP1L      EQU     0EBH            ;PCA module-1 capture register LOW
CCAP1H      EQU     0FBH            ;PCA module-1 capture register HIGH
CCAPM2      EQU     0DCH            ;PCA module-2 mode register
CCAP2L      EQU     0ECH            ;PCA module-2 capture register LOW
CCAP2H      EQU     0FCH            ;PCA module-2 capture register HIGH
CCAPM3      EQU     0DDH            ;PCA module-3 mode register
CCAP3L      EQU     0EDH            ;PCA module-3 capture register LOW
CCAP3H      EQU     0FDH            ;PCA module-3 capture register HIGH

;-----------------------------------------
            ORG     0000H
            LJMP    MAIN
;-----------------------------------------
            ORG     0100H
MAIN:
            MOV     CCON,   #0      ;Initial PCA control register
                                    ;PCA timer stop running
                                    ;Clear CF flag
                                    ;Clear all module interrupt flag
```

```
        CLR     A                               ;
        MOV     CL,     A                       ;Reset PCA base timer
        MOV     CH,     A                       ;
        MOV     CMOD,   #02H                    ;Set PCA timer clock source as Fosc/2
                                                ;Disable PCA timer overflow interrupt
;-------------------------------
        MOV     A,      #080H           ;
        MOV     CCAP0H,A                ;PWM0 port output 50% duty cycle square wave
        MOV     CCAP0L,A                ;
        MOV     CCAPM0,#42H             ;PCA module-0 work in 8-bit PWM mode and no PCA interrupt
;-------------------------------
        MOV     A,      #0C0H           ;
        MOV     CCAP1H,A                ;PWM1 port output 25% duty cycle square wave
        MOV     CCAP1L,A                ;
        MOV     CCAPM1,#42H             ;PCA module-1 work in 8-bit PWM mode and no PCA interrupt
;-------------------------------
        SETB    CR                      ;PCA timer start run

        SJMP    $
;-----------------------------------------
        END
```

10.8 利用PWM实现D/A功能的典型应用线路图



第11章 同步串行外围接口(SPI接口)

STC12C5410AD系列单片机还提供另一种高速串行通信接口 —— SPI接口。SPI是一种全双工、高速、同步的通信总线，有两种操作模式：主模式和从模式。在主模式中支持高达3 Mbps的速率(工作频率为12MHz时，如果CPU主频采用20MHz到36MHz，则可更高，从模式时速度无法太快，SYSclk/8以内较好)，还具有传输完成标志和写冲突标志保护。

11.1 与SPI功能模块相关的特殊功能寄存器

STC12C5410AD系列 1T 8051单片机SPI功能模块特殊功能寄存器 SPI Management SFRs

符号	描述	地址	位地址及其符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
SPCTL	SPI Control Register	85H	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	0000,0100
SPSTAT	SPI Status Register	84H	SPIF	WCOL	-	-	-	-	-	-	00xx,xxxx
SPDAT	SPI Data Register	86H									0000,0000

1. SPI控制寄存器SPCTL

SPI控制寄存器的格式如下：

SPCTL：SPI控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPCTL	85H	name	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SSIG：$\overline{SS}$引脚忽略控制位。

SSIG=1，MSTR（位4）确定器件为主机还是从机；

SSIG=0，$\overline{SS}$脚用于确定器件为主机还是从机. $\overline{SS}$脚可作为I/O口使用(见SPI主从选择表)

SPEN：SPI使能位。

SPEN=1，SPI使能；

SPEN=0，SPI被禁止，所有SPI引脚都作为I/O口使用。

DORD：设定SPI数据发送和接收的位顺序。

DORD=1，数据字的LSB(最低位)最先发送；

DORD=0，数据字的MSB(最高位)最先发送。

MSTR：主/从模式选择位（见SPI主从选择表）。

CPOL：SPI时钟极性。

CPOL=1，SPICLK空闲时为高电平。SPICLK的前时钟沿为下降沿而后沿为上升沿。

CPOL=0，SPICLK空闲时为低电平。SPICLK的前时钟沿为上升沿而后沿为下降沿。

CPHA：SPI时钟相位选择。

CPHA=1，数据在SPICLK的前时钟沿驱动，并在后时钟沿采样。

CPHA=0，数据在$\overline{SS}$为低（SSIG＝00）时被驱动，在SPICLK的后时钟沿被改变，并在前时钟沿被采样。（注：SSIG = 1时的操作未定义）

SPR1、SPR0：SPI时钟速率选择控制位。SPI时钟选择如下表所列。

SPI时钟频率的选择

SPR1	SPR0	时钟(SCLK)
0	0	CPU_CLK/4
0	1	CPU_CLK/16
1	0	CPU_CLK/64
1	1	CPU_CLK/128

其中，CPU_CLK是CPU时钟。

2. SPI状态寄存器SPSTAT

SPI状态寄存器的格式如下：

SPSTAT: SPI状态寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPCTL	CDH	name	SPIF	WCOL	-	-	-	-	-	-

SPIF：SPI传输完成标志。

当一次串行传输完成时，SPIF置位。此时，如果SPI中断被打开(即ESPI (IE2.1) 和EA(IE.7) 都置位)，则产生中断。当SPI处于主模式且SSIG=0时，如果$\overline{SS}$为输入并被驱动为低电平，SPIF也将置位, 表示“模式改变”。SPIF标志通过软件向其写入"1"清零。

WCOL：SPI写冲突标志。

在数据传输的过程中如果对SPI 数据寄存器SPDAT执行写操作，WCOL将置位。WCOL标志通过软件向其写入"1"清零。

3. SPI数据寄存器SPDAT

SPI数据寄存器的格式如下：

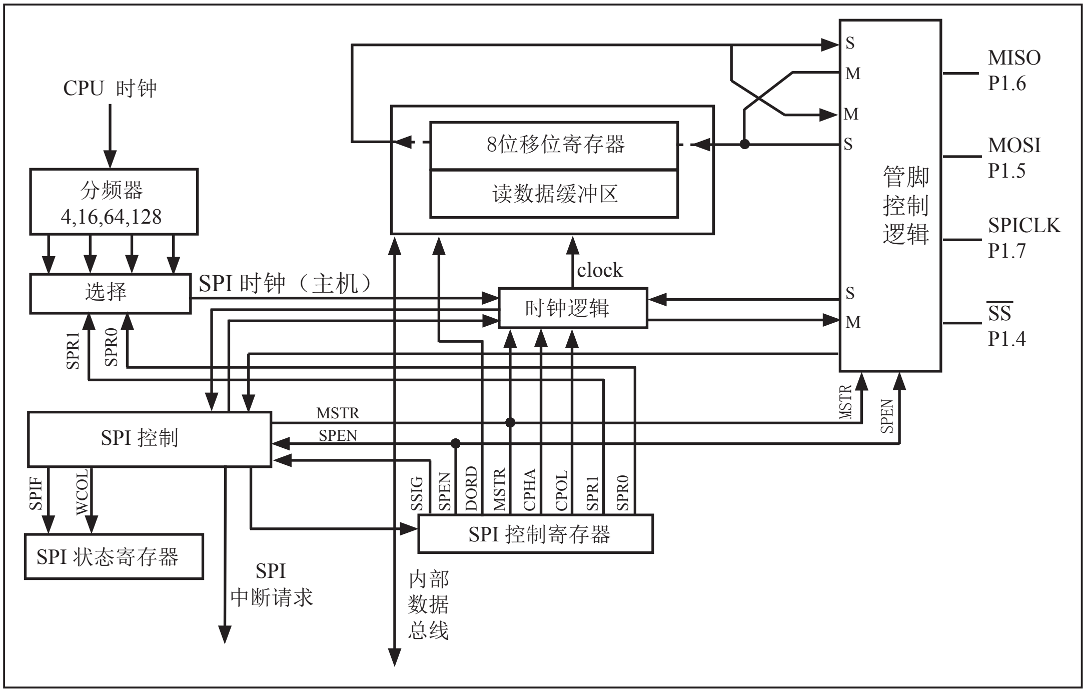
SPDAT: SPI数据寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPDAT	CFH	name								

SPDAT.7 - SPDAT.0：传输的数据位Bit7～Bit0

11.2 SPI接口的结构

STC12C5410AD系列单片机的SPI功能方框图如下图所示。



SPI 功能方框图

SPI的核心是一个8位移位寄存器和数据缓冲器，数据可以同时发送和接收。在SPI数据的传输过程中，发送和接收的数据都存储在数据缓冲器中。

对于主模式，若要发送一字节数据，只需将这个数据写到SPDAT寄存器中。主模式下$\overline{SS}$信号不是必需的；但是在从模式下，必须在$\overline{SS}$信号变为有效并接收到合适的时钟信号后，方可进行数据传输。在从模式下，如果一个字节传输完成后，$\overline{SS}$信号变为高电平，这个字节立即被硬件逻辑标志为接收完成，SPI接口准备接收下一个数据。

11.3 SPI接口的数据通信

SPI接口有4个管脚：SCLK/P1.7, MOSI/P1.5, MISO/P1.6和$\overline{SS}$/P1.4 。

MOSI (Master Out Slave In，主出从入)：主器件的输出和从器件的输入，用于主器件到从器件的串行数据传输。根据SPI规范，多个从机共享一根MOSI信号线。在时钟边界的前半周期，主机将数据放在MOSI信号线上，从机在该边界处获取该数据。

MISO (Master In Slave Out，主入从出)：从器件的输出和主器件的输入，用于实现从器件到主器件的数据传输。SPI规范中，一个主机可连接多个从机，因此，主机的MISO信号线会连接到多个从机上，或者说，多个从机共享一根MISO信号线。当主机与一个从机通信时，其他从机应将其MISO引脚驱动置为高阻状态。

SCLK (SPI Clock，串行时钟信号)：串行时钟信号是主器件的输出和从器件的输入，用于同步主器件和从器件之间在MOSI和MISO线上的串行数据传输。当主器件启动一次数据传输时，自动产生8个SCLK时钟周期信号给从机。在SCLK的每个跳变处(上升沿或下降沿)移出一位数据。所以，一次数据传输可以传输一个字节的数据。

SCLK、MOSI和MISO通常和两个或更多SPI器件连接在一起。数据通过MOSI由主机传送到从机，通过MISO由从机传送到主机。SCLK信号在主模式时为输出，在从模式时为输入。如果SPI系统被禁止，即SPEN(SPCTL.6)=0(复位值)，这些管脚都可作为I/O口使用。

$\overline{SS}$(Slave Select，从机选择信号)：这是一个输入信号，主器件用它来选择处于从模式的SPI模块。主模式和从模式下，$\overline{SS}$的使用方法不同。在主模式下，SPI接口只能有一个主机，不存在主机选择问题。该模式下$\overline{SS}$不是必需的。主模式下通常将主机的$\overline{SS}$管脚通过10KΩ的电阻上拉高电平。每一个从机的$\overline{SS}$接主机的I/O口，由主机控制电平高低，以便主机选择从机。在从模式下，不了发送还是接收，$\overline{SS}$信号必须有效。因此在一次数据传输开始之前必须将$\overline{SS}$为低电平。SPI主机可以使用I/O口选择一个SPI器件作为当前的从机。在典型的配置中，SPI主机使用I/O口选择一个SPI器件作为当前的从机。

SPI从器件通过其$\overline{SS}$脚确定是否被选择。如果满足下面的条件之一，$\overline{SS}$就被忽略：

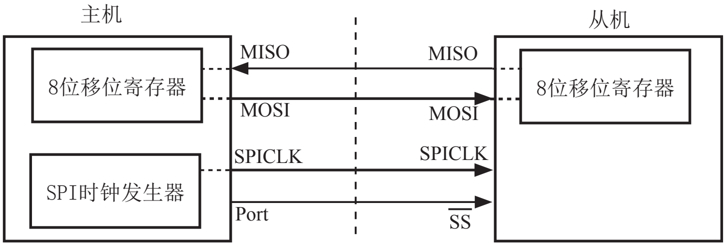
- 如果SPI系统被禁止，即SPEN(SPCTL.6)= 0(复位值)
- 如果SPI配置为主机，即MSTR(SPCTL.4)=1, 并且P1.4配置为输出(通过P1M0.4和P1M1.4)
- 如果$\overline{SS}$脚被忽略，即SSIG(SPCTL.7)＝ 1，该脚配置用于I/O口功能。

注：即使SPI被配置为主机（MSTR ＝ 1），它仍然可以通过拉低$\overline{SS}$脚配置为从机（如果P1.4配置为输入且SSIG=0）。要使能该特性，应当置位SPIF(SPSTAT.7)。

11.3.1 SPI接口的数据通信方式

STC12C5410AD系列单片机的SPI接口的数据通信方式有3种：单主机—从机方式、双器件方式(器件可互为主机和从机)和单主机—多从机方式。

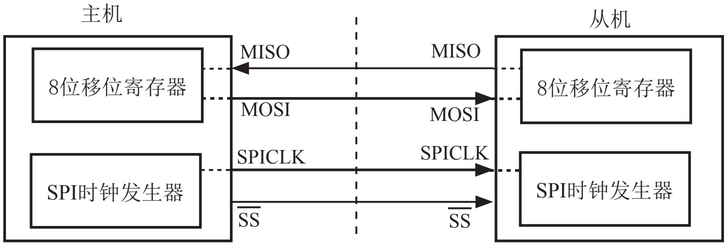
单主机—单从机方式的连接图如下SPI图1所示。



SPI图1　SPI单主机—单从机 配置

在上图SPI图1中，从机的SSIG(SPCTL.7)为0，$\overline{SS}$用于选择从机。SPI主机可使用任何端口（包括P1.4/$\overline{SS}$）来驱动$\overline{SS}$脚。主机SPI与从机SPI的8位移位寄存器连接成一个循环的16位移位寄存器。当主机程序向SPDAT寄存器写入一个字节时，立即启动一个连续的8位移位通信过程：主机的SCLK引脚向从机的SCLK引脚发出一串脉冲，在这串脉冲的驱动下，主机SPI的8位移位寄存器中的数据移动到了从机SPI的8移位寄存器中。与此同时，从机SPI的8位移位寄存器中的数据移动到了主机SPI的8位移位寄存器中。由此，主机既可向从机发送数据，又可读从机中的数据。

双器件方式(器件可互为主机和从机)的连接图如下SPI图2所示。



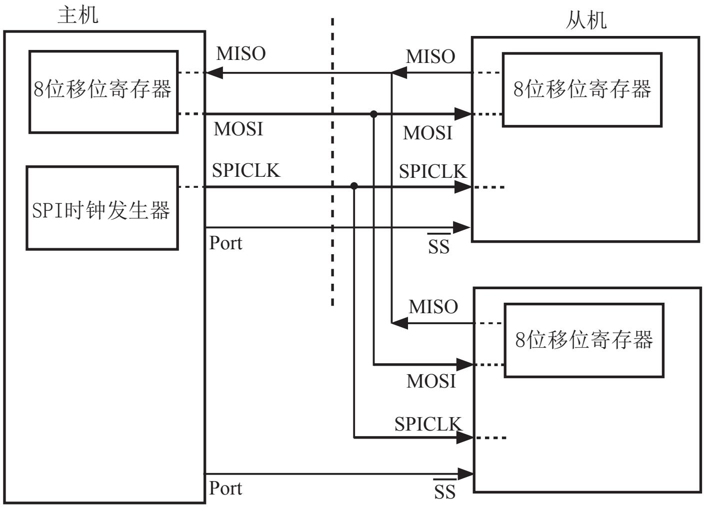
SPI图2　SPI双器件配置(器件可互为主从)

上图SPI图2所示为两个器件互为主从的情况。当没有发生SPI操作时，两个器件都可配置为主机(MSTR=1)，将SSIG清零并将P1.4($\overline{SS}$)配置为准双向模式。当其中一个器件启动传输时，它可将P1.4配置为输出并驱动为低电平，这样就强制另一个器件变为从机。

双方初始化时将自己设置成忽略$\overline{SS}$脚的SPI从模式。当一方要主动发送数据时，先检测$\overline{SS}$脚的电平，如果$\overline{SS}$脚是高电平，就将自己设置成忽略$\overline{SS}$脚的主模式。通信双方平时将SPI设置成没有被选中的从模式。在该模式下，MISO、MOSI、SCLK均为输入，当多个MCU的SPI接口以此模式并联时不会发生总线冲突。这种特性在互为主/从、一主多从等应用中很有用。

注意：互为主/从模式时，双方的SPI速率必须相同。如果使用外部晶体振荡器，双方的晶体频率也要相同。

双器件方式(器件可互为主机和从机)的连接图如下SPI图3所示。



SPI图3　SPI 单主机-多从机 配置

在上图SPI图3 中，从机的SSIG(SPCTL.7)为0，从机通过对应的$\overline{SS}$信号被选中。SPI主机可使用任何端口(包括P1.4/$\overline{SS}$)来驱动$\overline{SS}$脚。

11.3.2 对SPI进行配置

STC12C5410AD系列单片机进行SPI通信时，主机和从机的选择由SPEN、SSIG、$\overline{SS}$引脚(P1.4)和MSTR联合控制。下表所示为主/从模式的配置以及模式的使用和传输方向。

SPI 主从模式选择

SPEN	SSIG	$\overline{SS}$脚 P1.4	MSTR	主或从模式	MISO P1.6	MOSI P1.5	SPICLK P1.7	备注
0	X	P1.4	X	SPI功能禁止	P1.6	P1.5	P1.7	SPI禁止。P1.4/P1.5/P1.6/P1.7作为普通I/O口使用
1	0	0	0	从机模式	输出	输入	输入	选择作为从机
1	0	1	0	从机模式未被选中	高阻	输入	输入	未被选中。MISO为高阻状态，以避免总线冲突
1	0	0	1—>0	从机模式	输出	输入	输入	P1.4/$\overline{SS}$配置为输入或准双向口。SSIG为0。如果择$\overline{SS}$被驱动为低电平，则被选择作为从机。当$\overline{SS}$变为低电平时，MSTR将清零。注：当$\overline{SS}$处于输入模式时，如被驱动为低电平且SSIG=0时，MSTR位自动清零。
1	0	1	1	主(空闲)	输入	高阻	高阻	当主机空闲时MOSI和SCLK为高阻态以避免总线冲突。用户必须将SCLK上拉或下拉（根据CPOL/SPCTL.3 的取值）以避免SCLK出现悬浮状态。
				主(激活)		输出	输出	作为主机激活时，MOSI和SCLK为推挽输出
1	1	P1.4	0	从	输出	输入	输入	
1	1	P1.4	1	主	输入	输出	输出	

11.3.3 作为主机/从机时的额外注意事项

作为从机时的额外注意事项

当CPHA＝0时，SSIG必须为0，$\overline{SS}$脚必须取反并且在每个连续的串行字节之间重新设置为高电平。如果SPDAT寄存器在$\overline{SS}$有效（低电平）时执行写操作，那么将导致一个写冲突错误。CPHA=0且SSIG=0时的操作未定义。

当CPHA＝1时，SSIG可以置位。如果SSIG＝0，$\overline{SS}$脚可在连续传输之间保持低有效（即一直固定为低电平）。这种方式有时适用于具有单固定主机和单从机驱动MISO数据线的系统。

作为主机时的额外注意事项

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，主机对SPI数据寄存器的写操作将启动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。

需要注意的是，主机可以通过将对应器件的$\overline{SS}$脚驱动为低电平实现与之通信。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）置位并产生一个中断（如果SPI中断使能）。主机和从机CPU的两个移位寄存器可以看作是一个16 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。

11.3.4 通过$\overline{SS}$改变模式

如果SPEN=1， SSIG=0且MSTR=1，SPI使能为主机模式。$\overline{SS}$脚可配置为输入或准双向模式。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系统执行以下动作:

1）MSTR清零并且CPU变成从机。这样SPI就变成从机。MOSI和SCLK强制变为输入模式，而MISO则变为输出模式。

2）SPSTAT的SPIF标志位置位。如果SPI中断已被使能，则产生SPI中断。

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所清零而用户想继续将SPI作为主机，这时就必须重新置位MSTR，否则就进入从机模式。

11.3.5 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进行写操作时，WCOL位(SPSTAT.6)将置位以指示数据冲突。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。

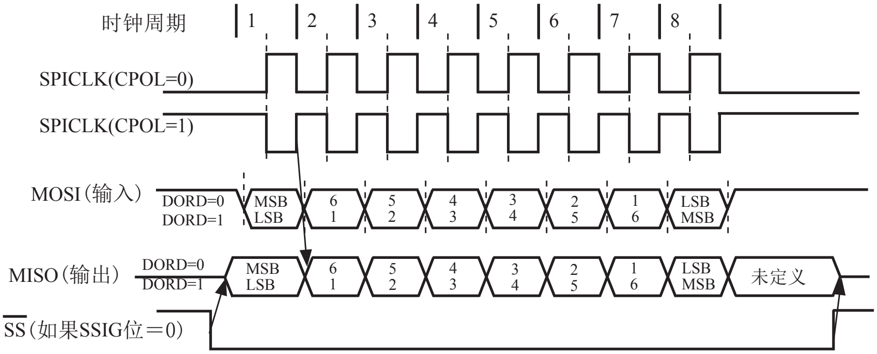
接收数据时，接收到的数据传送到一个并行读数据缓冲区，这样将释放移位寄存器以进行下一个数据的接收。但必须在下个字符完全移入之前从数据寄存器中读出接收到的数据，否则，前一个接收数据将丢失。

WCOL可通过软件向其写入“1”清零。

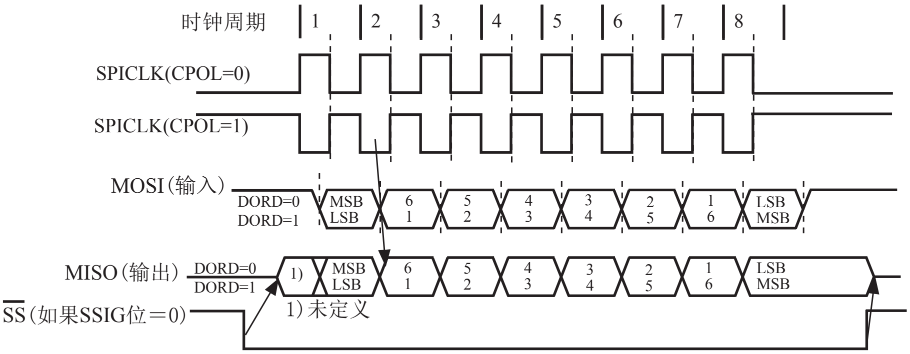
11.3.6 数据模式

时钟相位位(CPHA)允许用户设置采样和改变数据的时钟边沿。时钟极性位CPOL允许用户设置时钟极性。

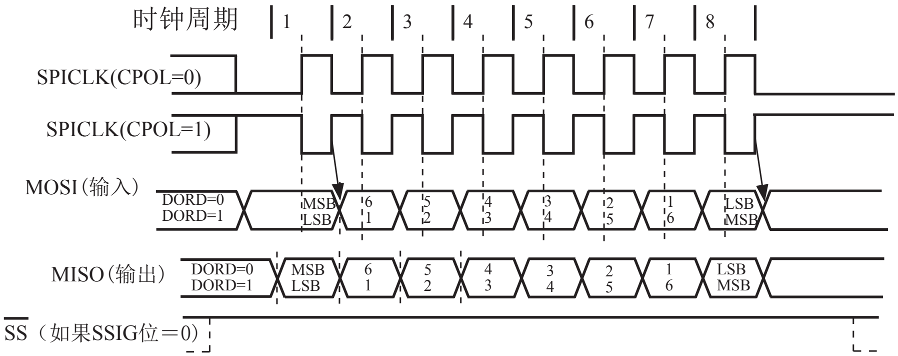
SPI图4～图7 所示为时钟相位位CPHA的不同设定。



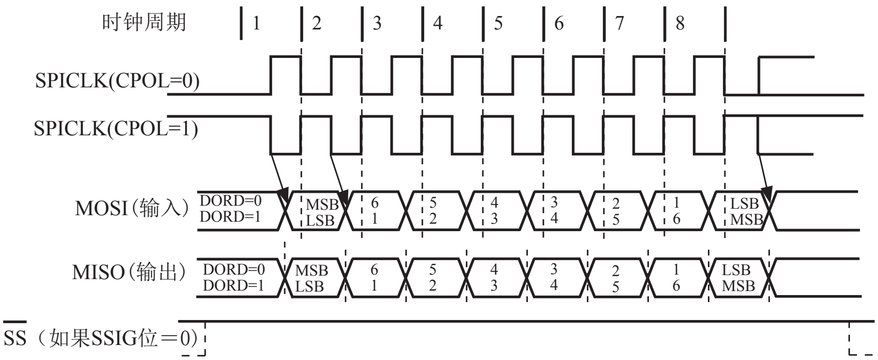
SPI图4　SPI 从机传输格式（CPHA=0）



SPI图5　SPI从机传输格式（CPHA=1）



SPI图6　SPI主机传输格式（CPHA=0）



SPI图7　SPI主机传输格式（CPHA=1）

SPI接口的时钟信号线SCLK有Idle和Active两种状态：Idle状态时指在不进行数据传输的时候(或数据传输完成后)SCLK所处的状态；Active是与Idle相对的一种状态。

时钟相位位(CPHA)允许用户设置采样和改变数据的时钟边沿。时钟极性CPOL允许用户设置时钟极性。

如果CPOL=0,Idle状态=低电平，Active状态=高电平；

如果CPOL=1,Idle状态=高电平，Active状态=低电平。

主机总是在SCLK=Idle状态时，将下一位要发送的数据置于数据线MOSI上。

从Idle状态到Active状态的转变，称为SCLK前沿；从Active状态到Idle状态的转变，称为SCLK后沿。一个SCLK前沿和后沿构成一个SCLK时钟周期，一个SCLK时钟周期传输一位数据。

SPI时钟预分频器选择

SPI时钟预分频器选择是通过SPCTL寄存器中的SPR1-SPR0位实现的

SPI时钟频率的选择

SPR1	SPR0	时钟(SCLK)
0	0	CPU_CLK/4
0	1	CPU_CLK/16
1	0	CPU_CLK/64
1	1	CPU_CLK/128

其中，CPU_CLK是CPU时钟。

11.4 适用单主单从系统的SPI功能测试程序

11.4.1 中断方式

1. C程序

```
/*------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------*/
/* --- 演示STC 1T 系列单片机  SPI功能(适用单主单从，中断方式)----*/
/* 如果要在程序中使用或在文章中引用该程序，  --------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------*/
/*------------------------------------------------------------------------------------------*/

#include "reg51.h"

#define   MASTER                                        //define:master undefine:slave
#define   FOSC      18432000L
#define   BAUD      (256 - FOSC / 32 / 115200)

typedef   unsigned char    BYTE;
typedef   unsigned int     WORD;
typedef   unsigned long    DWORD;

sfr       AUXR    =  0x8e;                              //Auxiliary register

sfr       SPSTAT  =  0x84;                              //SPI status register
#define   SPIF      0x80                                //SPSTAT.7
#define   WCOL      0x40                                //SPSTAT.6
sfr       SPCTL   =  0x85;                              //SPI control register
#define   SSIG      0x80                                //SPCTL.7
#define   SPEN      0x40                                //SPCTL.6
#define   DORD      0x20                                //SPCTL.5
#define   MSTR      0x10                                //SPCTL.4
#define   CPOL      0x08                                //SPCTL.3
#define   CPHA      0x04                                //SPCTL.2
#define   SPDHH     0x00                                //CPU_CLK/4
#define   SPDH      0x01                                //CPU_CLK/16
#define   SPDL      0x02                                //CPU_CLK/64
#define   SPDLL     0x03                                //CPU_CLK/128
sfr       SPDAT   =  0x86;                              //SPI data register
sbit      SPISS   =  P1^3;                              //SPI slave select, connect to slave' SS(P1.4) pin

sbit      EADC_SPI  =  IE^5;
#define   ESPI      0x08;                               //AUXR.3
```

```
void InitUart();
void InitSPI();
void SendUart(BYTE dat);                //send data to PC
BYTE RecvUart();                        //receive data from PC

//////////////////////////////////////////////////////////

void main()
{
        InitUart();                     //initial UART
        InitSPI();                      //initial SPI
        AUXR |= ESPI;
        EADC_SPI = 1;
        EA = 1;

        while (1)
        {
        #ifdef    MASTER                //for master (receive UART data from PC and send it to slave,
                                        //in the meantime receive SPI data from slave and send it to PC)
                ACC = RecvUart();
                SPISS = 0;              //pull low slave SS
                SPDAT = ACC;            //trigger SPI send
        #endif
        }
}

//////////////////////////////////////////////////////////

void spi_isr()      interrupt 5   using 1       //SPI interrupt routine 5 (002BH)
{
        SPSTAT = SPIF | WCOL;           //clear SPI status
#ifdef MASTER
        SPISS = 1;                      //push high slave SS
        SendUart(SPDAT);                //return received SPI data
#else                                   //for salve (receive SPI data from master and
        SPDAT = SPDAT;                  //        send previous SPI data to master)
#endif
}

//////////////////////////////////////////////////////////
```

```
void InitUart()
{
        SCON = 0x5a;                    //set UART mode as 8-bit variable baudrate
        TMOD = 0x20;                    //timer1 as 8-bit auto reload mode
        AUXR = 0x40;                    //timer1 work at 1T mode
        TH1 = TL1 = BAUD;               //115200 bps
        TR1 = 1;
}

//////////////////////////////////////////////////////////

void InitSPI()
{
        SPDAT = 0;                      //initial SPI data
        SPSTAT = SPIF | WCOL;           //clear SPI status
#ifdef MASTER
        SPCTL = SPEN | MSTR;            //master mode
#else
        SPCTL = SPEN;                   //slave mode
#endif
}

//////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
        while (!TI);                    //wait pre-data sent
        TI = 0;                         //clear TI flag
        SBUF = dat;                     //send current data
}

//////////////////////////////////////////////////////////

BYTE RecvUart()
{
        while (!RI);                    //wait receive complete
        RI = 0;                         //clear RI flag
        return SBUF;                    //return receive data
}
```

2. 汇编程序

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------*/
/* --- 演示STC 1T 系列单片机  SPI功能(适用单主单从，中断方式)----*/
/* 如果要在程序中使用或在文章中引用该程序，  --------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------*/
/*------------------------------------------------------------------------------------*/

//#define  MASTER                       //define:master undefine:slave

AUXR    DATA    08EH                    ;Auxiliary register
SPSTAT  DATA    084H                    ;SPI status register
SPIF    EQU     080H                    ;SPSTAT.7
WCOL    EQU     040H                    ;SPSTAT.6
SPCTL   DATA    085H                    ;SPI control register
SSIG    EQU     080H                    ;SPCTL.7
SPEN    EQU     040H                    ;SPCTL.6
DORD    EQU     020H                    ;SPCTL.5
MSTR    EQU     010H                    ;SPCTL.4
CPOL    EQU     008H                    ;SPCTL.3
CPHA    EQU     004H                    ;SPCTL.2
SPDHH   EQU     000H                    ;CPU_CLK/4
SPDH    EQU     001H                    ;CPU_CLK/16
SPDL    EQU     002H                    ;CPU_CLK/64
SPDLL   EQU     003H                    ;CPU_CLK/128
SPDAT   DATA    086H                    ;SPI data register
SPISS   BIT     P1.3                    ;SPI slave select, connect to slave' SS(P1.4) pin

EADC_SPI        BIT     IE.5
ESPI            EQU     08H             ;AUXR.3

;//////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET

        ORG     002BH                   ;SPI interrupt routine 5(2BH)
SPI_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     SPSTAT, #SPIF | WCOL    ;clear SPI status
```

```
#ifdef    MASTER
          SETB    SPISS                       ;push high slave SS
          MOV     A,        SPDAT             ;return received SPI data
          LCALL   SEND_UART
#else                                         //for salve (receive SPI data from master and
          MOV     SPDAT,  SPDAT               ;        send previous SPI data to master)
#endif
          POP     PSW
          POP     ACC
          RETI

;///////////////////////////////////////////////////////

          ORG     0100H
RESET:
          LCALL   INIT_UART                   ;initial UART
          LCALL   INIT_SPI                    ;initial SPI
          ORL     AUXR,   #ESPI               ;enable SPI interrupt
          SETB    EADC_SPI
          SETB    EA
MAIN:
#ifdef MASTER                   //for master (receive UART data from PC and send it to slave,
          LCALL   RECV_UART     ; in the meantimereceive SPI data from slave and send it to PC)
          CLR     SPISS         ;pull low slave SS
          MOV     SPDAT,  A     ;trigger SPI send
#endif
          SJMP    MAIN

;///////////////////////////////////////////////////////

INIT_UART:
          MOV     SCON,   #5AH  ;set UART mode as 8-bit variable baudrate
          MOV     TMOD,   #20H  ;timer1 as 8-bit auto reload mode
          MOV     AUXR,   #40H  ;timer1 work at 1T mode
          MOV     TL1,    #0FBH ;115200 bps(256 - 18432000 / 32 / 115200)
          MOV     TH1,    #0FBH
          SETB    TR1
          RET

;///////////////////////////////////////////////////////
```

```
INIT_SPI:
        MOV     SPDAT,   #0                      ;initial SPI data
        MOV     SPSTAT,  #SPIF | WCOL            ;clear SPI status
#ifdef MASTER
        MOV     SPCTL,   #SPEN | MSTR            ;master mode
#else
        MOV     SPCTL,   #SPEN                   ;slave mode
#endif
        RET

;/////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,      $                       ;wait pre-data sent
        CLR     TI                               ;clear TI flag
        MOV     SBUF,    A                       ;send current data
        RET

;/////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,$                             ;wait receive complete
        CLR     RI                               ;clear RI flag
        MOV     A,       SBUF                    ;return receive data
        RET
        RET

;/////////////////////////////////////////////////////////

        END
```

11.4.2 查询方式

1. C程序

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用单主单从，查询方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------*/
/*------------------------------------------------------------------------------------*/

#include "reg51.h"

//#define MASTER                                  //define:master undefine:slave
#define FOSC        18432000L
#define BAUD        (256 - FOSC / 32 / 115200)

typedef unsigned char   BYTE;
typedef unsigned int    WORD;
typedef unsigned long   DWORD;

sfr     AUXR    =   0x8e;                       //Auxiliary register

sfr     SPSTAT  =   0x84;                       //SPI status register
#define SPIF    0x80                            //SPSTAT.7
#define WCOL    0x40                            //SPSTAT.6
sfr     SPCTL   =   0x85;                       //SPI control register
#define SSIG    0x80                            //SPCTL.7
#define SPEN    0x40                            //SPCTL.6
#define DORD    0x20                            //SPCTL.5
#define MSTR    0x10                            //SPCTL.4
#define CPOL    0x08                            //SPCTL.3
#define CPHA    0x04                            //SPCTL.2
#define SPDHH   0x00                            //CPU_CLK/4
#define SPDH    0x01                            //CPU_CLK/16
#define SPDL    0x02                            //CPU_CLK/64
#define SPDLL   0x03                            //CPU_CLK/128
sfr     SPDAT   =   0x86;                       //SPI data register
sbit    SPISS   =   P1^3;                       //SPI slave select, connect to slave' SS(P1.4) pin

void    InitUart();
void    InitSPI();
void    SendUart(BYTE dat);                     //send data to PC
BYTE    RecvUart();                             //receive data from PC
BYTE    SPISwap(BYTE dat);                      //swap SPI data between master and slave
```

```
/////////////////////////////////////////////////////////////

void main()
{
        InitUart();                     //initial UART
        InitSPI();                      //initial SPI

        while (1)
        {
        #ifdef MASTER           //for master (receive UART data from PC and send it to slave,
                                //  in the meantime receive SPI data from slave and send it to PC)
                SendUart(SPISwap(RecvUart()));
        #else                           //for salve (receive SPI data from master and
                ACC = SPISwap(ACC);     //      send previous SPI data to master)
        #endif
        }
}

/////////////////////////////////////////////////////////////

void InitUart()
{
        SCON = 0x5a;                            //set UART mode as 8-bit variable baudrate
        TMOD = 0x20;                            //timer1 as 8-bit auto reload mode
        AUXR = 0x40;                            //timer1 work at 1T mode
        TH1 = TL1 = BAUD;                       //115200 bps
        TR1 = 1;
}

/////////////////////////////////////////////////////////////

void InitSPI()
{
        SPDAT = 0;                              //initial SPI data
        SPSTAT = SPIF | WCOL;                   //clear SPI status
#ifdef MASTER
        SPCTL = SPEN | MSTR;                    //master mode
#else
        SPCTL = SPEN;                           //slave mode
#endif
}
```

```
/////////////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
        while (!TI);                    //wait pre-data sent
        TI = 0;                         //clear TI flag
        SBUF = dat;                     //send current data
}

/////////////////////////////////////////////////////////////////

BYTE RecvUart()
{
        while (!RI);                    //wait receive complete
        RI = 0;                         //clear RI flag
        return SBUF;                    //return receive data
}

/////////////////////////////////////////////////////////////////

BYTE SPISwap(BYTE dat)
{
#ifdef MASTER
        SPISS = 0;                      //pull low slave SS
#endif
        SPDAT = dat;                    //trigger SPI send
        while (!(SPSTAT & SPIF));       //wait send complete
        SPSTAT = SPIF | WCOL;           //clear SPI status
#ifdef MASTER
        SPISS = 1;                      //push high slave SS
#endif
        return SPDAT;                   //return received SPI data
}
```

2. 汇编程序

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用单主单从，查询方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------*/
/*---------------------------------------------------------------------------------------*/

//#define  MASTER                          //define:master undefine:slave


AUXR    DATA    08EH                       ;Auxiliary register
SPSTAT  DATA    084H                       ;SPI status register
SPIF    EQU     080H                       ;SPSTAT.7
WCOL    EQU     040H                       ;SPSTAT.6
SPCTL   DATA    085H                       ;SPI control register
SSIG    EQU     080H                       ;SPCTL.7
SPEN    EQU     040H                       ;SPCTL.6
DORD    EQU     020H                       ;SPCTL.5
MSTR    EQU     010H                       ;SPCTL.4
CPOL    EQU     008H                       ;SPCTL.3
CPHA    EQU     004H                       ;SPCTL.2
SPDHH   EQU     000H                       ;CPU_CLK/4
SPDH    EQU     001H                       ;CPU_CLK/16
SPDL    EQU     002H                       ;CPU_CLK/64
SPDLL   EQU     003H                       ;CPU_CLK/128
SPDAT   DATA    086H                       ;SPI data register
SPISS   BIT     P1.3                       ;SPI slave select, connect to slave' SS(P1.4) pin

;////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET
        ORG     0100H
RESET:
        LCALL   INIT_UART                  ;initial UART
        LCALL   INIT_SPI                   ;initial SPI
```

```
MAIN:
#ifdef  MASTE                   //for master (receive UART data from PC and send it to slave, in the meantime
        LCALL   RECV_UART                           ;        receive SPI data from slave and send it to PC)
        LCALL   SPI_SWAP
        LCALL   SEND_UART
#else                                               //for salve (receive SPI data from master and
        LCALL   SPI_SWAP                            ;        send previous SPI data to master)
#endif
        SJMP    MAIN
;////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH                        ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H                        ;timer1 as 8-bit auto reload mode
        MOV     AUXR,   #40H                        ;timer1 work at 1T mode
        MOV     TL1,    #0FBH                       ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET

;////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,  #0                          ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL                ;clear SPI status
#ifdef  MASTER
        MOV     SPCTL,  #SPEN | MSTR                ;master mode
#else
        MOV     SPCTL,  #SPEN                       ;slave mode
#endif
        RET

;////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $                           ;wait pre-data sent
        CLR     TI                                  ;clear TI flag
        MOV     SBUF,   A                           ;send current data
        RET

;////////////////////////////////////////////////////////
```

```
RECV_UART:
        JNB     RI,     $                       ;wait receive complete
        CLR     RI                              ;clear RI flag
        MOV     A,      SBUF                    ;return receive data
        RET
        RET

;///////////////////////////////////////////////////////////

SPI_SWAP:
#ifdef MASTER
        CLR     SPISS                           ;pull low slave SS
#endif
        MOV     SPDAT,  A                       ;trigger SPI send
WAIT:
        MOV     A,      SPSTAT
        JNB     ACC.7,  WAIT                    ;wait send complete
        MOV     SPSTAT, #SPIF | WCOL            ;clear SPI status
#ifdef MASTER
        SETB    SPISS                           ;push high slave SS
#endif
        MOV      A,     SPDAT                   ;return received SPI data
        RET

;///////////////////////////////////////////////////////////

        END
```

11.5 适用互为主从系统的SPI功能测试程序

11.5.1 中断方式

1. C程序

```
/*------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用互为主从系统，中断方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------------*/
/*--------------------------------------------------------------------------------------------*/

#include "reg51.h"

#define FOSC       18432000L
#define BAUD       (256 - FOSC / 32 / 115200)

typedef   unsigned  char      BYTE;
typedef   unsigned  int       WORD;
typedef   unsigned  long      DWORD;

sfr       AUXR   =  0x8e;                     //Auxiliary register

sfr       SPSTAT =  0x84;                     //SPI status register
#define   SPIF      0x80                      //SPSTAT.7
#define   WCOL      0x40                      //SPSTAT.6
sfr       SPCTL  =  0x85;                     //SPI control register
#define   SSIG    0x80                        //SPCTL.7
#define   SPEN    0x40                        //SPCTL.6
#define   DORD    0x20                        //SPCTL.5
#define   MSTR    0x10                        //SPCTL.4
#define   CPOL    0x08                        //SPCTL.3
#define   CPHA    0x04                        //SPCTL.2
#define   SPDHH   0x00                        //CPU_CLK/4
#define   SPDH    0x01                        //CPU_CLK/16
#define   SPDL    0x02                        //CPU_CLK/64
#define   SPDLL   0x03                        //CPU_CLK/128
sfr       SPDAT   =      0x86;                        //SPI data register
sbit      SPISS   =      P1^3;        //SPI slave select, connect to other MCU's SS(P1.4) pin

sbit      EADC_SPI  =  IE^5;
#define   ESPI    0x08;               //AUXR.3
```

```
void InitUart();
void InitSPI();
void SendUart(BYTE dat);                        //send data to PC
BYTE RecvUart();                                //receive data from PC

bit MSSEL;                                      //1: master 0:slave

//////////////////////////////////////////////////////////

void main()
{
        InitUart();             //initial UART
        InitSPI();              //initial SPI
        AUXR |= ESPI;
        EADC_SPI = 1;
        EA = 1;

        while (1)
        {
                if (RI)
                {
                        SPCTL = SPEN | MSTR;            //set as master
                        MSSEL = 1;
                        ACC = RecvUart();
                        SPISS = 0;                      //pull low slave SS
                        SPDAT = ACC;                    //trigger SPI send
                }
        }
}

//////////////////////////////////////////////////////////

void spi_isr( ) interrupt 5 using 1                     //SPI interrupt routine 5 (002BH)
{
        SPSTAT = SPIF | WCOL;                           //clear SPI status
        if (MSSEL)
        {
                SPCTL = SPEN;                           //reset as slave
                MSSEL = 0;
                SPISS = 1;                              //push high slave SS
                SendUart(SPDAT);                        //return received SPI data
        }
        else
        {                                       //for salve (receive SPI data from master and
                SPDAT = SPDAT;                  //      send previous SPI data to master)
  }
}
```

```
void InitUart()
{
            SCON = 0x5a;                        //set UART mode as 8-bit variable baudrate
            TMOD = 0x20;                        //timer1 as 8-bit auto reload mode
            AUXR = 0x40;                        //timer1 work at 1T mode
            TH1 = TL1 = BAUD;                   //115200 bps
            TR1 = 1;
}

//////////////////////////////////////////////////////////////

void InitSPI()
{
            SPDAT = 0;                          //initial SPI data
            SPSTAT = SPIF | WCOL;               //clear SPI status
            SPCTL = SPEN;                       //slave mode
}

//////////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
            while (!TI);                        //wait pre-data sent
            TI = 0;                             //clear TI flag
            SBUF = dat;                         //send current data
}

//////////////////////////////////////////////////////////////

BYTE RecvUart()
{
            while (!RI);                        //wait receive complete
            RI = 0;                             //clear RI flag
            return SBUF;                        //return receive data
}
```

2. 汇编程序

```
/*-------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- -----------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用互为主从系统，中断方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 --------------------*/
/*-------------------------------------------------------------------------------------------*/

AUXR    DATA    08EH                    ;Auxiliary register
SPSTAT  DATA    084H                    ;SPI status register
SPIF    EQU     080H                    ;SPSTAT.7
WCOL    EQU     040H                    ;SPSTAT.6
SPCTL   DATA    085H                    ;SPI control register
SSIG    EQU     080H                    ;SPCTL.7
SPEN    EQU     040H                    ;SPCTL.6
DORD    EQU     020H                    ;SPCTL.5
MSTR    EQU     010H                    ;SPCTL.4
CPOL    EQU     008H                    ;SPCTL.3
CPHA    EQU     004H                    ;SPCTL.2
SPDHH   EQU     000H                    ;CPU_CLK/4
SPDH    EQU     001H                    ;CPU_CLK/16
SPDL    EQU     002H                    ;CPU_CLK/64
SPDLL   EQU     003H                    ;CPU_CLK/128
SPDAT   DATA    086H                    ;SPI data register
SPISS   BIT     P1.3                    ;SPI slave select, connect to other MCU's SS(P1.4) pin

EADC_SPI        BIT     IE.5
ESPI            EQU     08H             ;AUXR.3

MSSEL   BIT   20H.0                     ;1: master 0:slave

;//////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET

        ORG     002BH                   ;SPI interrupt routine
SPI_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     SPSTAT, #SPIF | WCOL    ;clear SPI status
        JBC     MSSEL, MASTER_SEND
```

```
SLAVE_RECV:
                                                    //for salve (receive SPI data from master and
        MOV     SPDAT,  SPDAT                       ;       send previous SPI data to master)
        JMP     SPI_EXIT
MASTER_SEND:
        SETB    SPISS                               ;push high slave SS
        MOV     SPCTL,  #SPEN                       ;           ;reset as slave
        MOV     A,      SPDAT                       ;return received SPI data
        LCALL   SEND_UART
SPI_EXIT:
        POP     PSW
        POP     ACC
        RETI

;////////////////////////////////////////////////////////

        ORG     0100H
RESET:
        MOV     SP,#3FH
        LCALL   INIT_UART                           ;initial UART
        LCALL   INIT_SPI                            ;initial SPI
        ORL     AUXR,   #ESPI
        SETB    EADC_SPI
        SETB    EA
MAIN:
        JNB     RI,     $                           ;wait UART data
        MOV     SPCTL,  #SPEN | MSTR                ; ;set as master
        SETB    MSSEL
        LCALL   RECV_UART                           ;receive UART data from PC
        CLR     SPISS                               ;pull low slave SS
        MOV     SPDAT, A                                        ;trigger SPI send
        SJMP    MAIN

;////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH                        ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H                        ;timer1 as 8-bit auto reload mode
        MOV     AUXR    ,#40H                       ;timer1 work at 1T mode
        MOV     TL1,    #0FBH                       ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET
```

```
;//////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,  #0                      ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL            ;clear SPI status
        MOV     SPCTL,  #SPEN                   ;slave mode
        RET

;//////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $                       ;wait pre-data sent
        CLR     TI                              ;clear TI flag
        MOV     SBUF,   A                       ;send current data
        RET

;//////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,     $                       ;wait receive complete
        CLR     RI                              ;clear RI flag
        MOV     A,      SBUF                    ;return receive data
        RET
        RET

;//////////////////////////////////////////////////////////

        END
```

11.5.2 查询方式

1. C程序

```
/*---------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用互为主从系统，查询方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 ---------------------*/
/*---------------------------------------------------------------------------------------------*/

#include "reg51.h"

#define   FOSC           18432000L
#define   BAUD           (256 - FOSC / 32 / 115200)

typedef   unsigned char  BYTE;
typedef   unsigned int   WORD;
typedef   unsigned long  DWORD;

sfr       AUXR   =       0x8e;                  //Auxiliary register

sfr       SPSTAT =       0x84;                       //SPI status register
#define   SPIF           0x80                   //SPSTAT.7
#define   WCOL           0x40                   //SPSTAT.6
sfr       SPCTL  =       0x85;                       //SPI control register
#define   SSIG           0x80                   //SPCTL.7
#define   SPEN           0x40                   //SPCTL.6
#define   DORD           0x20                   //SPCTL.5
#define   MSTR           0x10                   //SPCTL.4
#define   CPOL           0x08                   //SPCTL.3
#define   CPHA           0x04                   //SPCTL.2
#define   SPDHH          0x00                   //CPU_CLK/4
#define   SPDH           0x01                   //CPU_CLK/16
#define   SPDL           0x02                   //CPU_CLK/64
#define   SPDLL          0x03                   //CPU_CLK/128
sfr       SPDAT  =       0x86;                       //SPI data register
sbit      SPISS  =       P1^3;                  //SPI slave select, connect to slave' SS(P1.4) pin

void      InitUart();
void      InitSPI();
void      SendUart(BYTE dat);                   //send data to PC
BYTE      RecvUart();                           //receive data from PC
BYTE      SPISwap(BYTE dat);                    //swap SPI data between master
```

```
////////////////////////////////////////////////////////////

void main()
{
	InitUart();				//initial UART
	InitSPI();				//initial SPI

	while (1)
	{
		if (RI)
		{
			SPCTL = SPEN | MSTR;		//set as master
			SendUart(SPISwap(RecvUart()));
			SPCTL = SPEN;			//reset as slave
		}
		if (SPSTAT & SPIF)
		{
			SPSTAT = SPIF | WCOL; //clear SPI status
			SPDAT = SPDAT;		//mov data from receive buffer to send buffer
		}
	}
}

////////////////////////////////////////////////////////////

void InitUart()
{
	SCON = 0x5a;				//set UART mode as 8-bit variable baudrate
	TMOD = 0x20;				 //timer1 as 8-bit auto reload mode
	AUXR = 0x40;				//timer1 work at 1T mode
	TH1 = TL1 = BAUD;			//115200 bps
	TR1 = 1;
}

////////////////////////////////////////////////////////////

void InitSPI()
{
	SPDAT = 0;				//initial SPI data
	SPSTAT = SPIF | WCOL;			//clear SPI status
	SPCTL = SPEN;				//slave mode
}
```

```
////////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
        while (!TI);                    //wait pre-data sent
        TI = 0;                         //clear TI flag
        SBUF = dat;                     //send current data
}

////////////////////////////////////////////////////////////

BYTE RecvUart()
{
        while (!RI);                    //wait receive complete
        RI = 0;                         //clear RI flag
        return SBUF;                    //return receive data
}

////////////////////////////////////////////////////////////

BYTE SPISwap(BYTE dat)
{
        SPISS = 0;                      //pull low slave SS
        SPDAT = dat;                    //trigger SPI send
        while (!(SPSTAT & SPIF));       //wait send complete
        SPSTAT = SPIF | WCOL;           //clear SPI status
        SPISS = 1;                      //push high slave SS
        return SPDAT;                   //return received SPI data
}
```

2. 汇编程序

```
/*------------------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ------------------------------------------------*/
/* --- 演示STC 1T 系列单片机 SPI功能(适用互为主从系统，查询方式)---*/
/* 如果要在程序中使用或在文章中引用该程序， --------------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -------------------*/
/*------------------------------------------------------------------------------------------------*/

AUXR    DATA    08EH                ;Auxiliary register
SPSTAT  DATA    084H                ;SPI status register
SPIF    EQU     080H                ;SPSTAT.7
WCOL    EQU     040H                ;SPSTAT.6
SPCTL   DATA    085H                ;SPI control register
SSIG    EQU     080H                ;SPCTL.7
SPEN    EQU     040H                ;SPCTL.6
DORD    EQU     020H                ;SPCTL.5
MSTR    EQU     010H                ;SPCTL.4
CPOL    EQU     008H                ;SPCTL.3
CPHA    EQU     004H                ;SPCTL.2
SPDHH   EQU     000H                ;CPU_CLK/4
SPDH    EQU     001H                ;CPU_CLK/16
SPDL    EQU     002H                ;CPU_CLK/64
SPDLL   EQU     003H                ;CPU_CLK/128
SPDAT   DATA    086H                ;SPI data register
SPISS   BIT     P1.3                ;SPI slave select, connect to slave' SS(P1.4) pin

;////////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET
        ORG     0100H
RESET:
        LCALL   INIT_UART           ;initial UART
        LCALL   INIT_SPI            ;initial SPI
MAIN:
        JB      RI,     MASTER_MODE
```

```
SLAVE_MODE:
        MOV     A,      SPSTAT
        JNB     ACC.7,  MAIN
        MOV     SPSTAT, #SPIF | WCOL                    ;clear SPI status
        MOV     SPDAT,  SPDAT                           ;return received SPI data
        SJMP    MAIN
MASTER_MODE:
        MOV     SPCTL,  #SPEN | MSTR                    ;set as master
        LCALL   RECV_UART                               ;receive UART data from PC
        LCALL   SPI_SWAP                ;send it to slave, in the meantime, receive SPI data from slave
        LCALL   SEND_UART               ;send SPI data to PC
        MOV     SPCTL,  #SPEN           ;       ;reset as slave
        SJMP    MAIN

;/////////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH            ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H            ;timer1 as 8-bit auto reload mode
        MOV     AUXR,   #40H            ;timer1 work at 1T mode
        MOV     TL1,    #0FBH           ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET

;/////////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,  #0                              ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL                    ;clear SPI status
        MOV     SPCTL,  #SPEN                           ;slave mode
        RET

;/////////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $                               ;wait pre-data sent
        CLR     TI                                      ;clear TI flag
        MOV     SBUF,   A                               ;send current data
        RET
```

```
;///////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,     $                       ;wait receive complete
        CLR     RI                              ;clear RI flag
        MOV     A,      SBUF                    ;return receive data
        RET
        RET

;///////////////////////////////////////////////////////////

SPI_SWAP:
        CLR     SPISS                           ;pull low slave SS
        MOV     SPDAT,  A                       ;trigger SPI send
WAIT:
        MOV     A,      SPSTAT
        JNB     ACC.7,  WAIT                    ;wait send complete
        MOV     SPSTAT, #SPIF | WCOL            ;clear SPI status
        SETB    SPISS                           ;push high slave SS
        MOV     A,      SPDAT                   ;return received SPI data
        RET

;///////////////////////////////////////////////////////////

        END
```

第12章 STC12C5410AD系列EEPROM的应用

STC12C5410AD系列单片机内部集成了的EEPROM是与程序空间是分开的，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。使用时，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。

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Vcc偏低时，建议不要进行EEPROM/IAP操作。

需要注意的是：5V单片机在3.7V以上对EEPROM进行操作才有效,3.7V以下对EEPROM进行操作,MCU不执行此功能,但会继续往下执行程序。3.3V单片机在2.4V以上对EEPROM进行操作才有效,2.4V以下对EEPROM进行操作,MCU不执行此功能,但会继续往下执行程序.所以建议上电复位后在初始化程序时加200mS延时。可通过判断LVDF标志位判断Vcc的电压是否正常。

12.1 IAP及EEPROM新增特殊功能寄存器介绍

符号	描述	地址	位地址及符号 MSB							LSB	复位值
ISP_DATA	ISP/IAP Flash Data Register	E2H									1111 1111B
ISP_ADDRH	ISP/IAP Flash Address High	E3H									0000 0000B
ISP_ADDRL	ISP/IAP Flash Address Low	E4H									0000 0000B
ISP_CMD	ISP/IAP Flash Command Register	E5H	-	-	-	-	-	-	MS1	MS0	xxxx xx00B
ISP_TRIG	ISP/IAP Flash Command Trigger	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP Control Register	E7H	ISPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 1000B

1. ISP/IAP数据寄存器ISP_DATA

ISP_DATA：ISP/IAP操作时的数据寄存器。
ISP/IAP 从Flash读出的数据放在此处，向Flash写的数据也需放在此处

2. ISP/IAP地址寄存器ISP_ADDRH和ISP_ADDRL

ISP_ADDRH : ISP/IAP 操作时的地址寄存器高八位。该寄存器地址为E3H, 复位后值为00H.
ISP_ADDRL ：ISP/IAP 操作时的地址寄存器低八位。该寄存器地址为E4H, 复位后值为00H.

3. ISP/IAP命令寄存器ISP_CMD

ISP/IAP命令寄存器ISP_CMD格式如下：

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CMD	E5H	name	-	-	-	-	-	-	MS1	MS0

MS1	MS0	命令 / 操作　模式选择
0	0	Standby　待机模式，无ISP操作
0	1	从用户的应用程序区对"Data Flash/EEPROM区"进行字节读
1	0	从用户的应用程序区对"Data Flash/EEPROM区"进行字节编程
1	1	从用户的应用程序区对"Data Flash/EEPROM区"进行扇区擦除

程序在用户应用程序区时，仅可以对数据Flash区(EEPROM)进行字节读/字节编程/扇区擦除, STC12C5612AD/STC12LE5612AD/STC12C5628AD/STC12LE5628AD/STC12C5630AD/STC12LE5630AD等除外，这几个型号可在应用程序区修改应用程序区。

4. ISP/IA命令触发寄存器ISP_TRIG

ISP_TRIG: ISP/IAP 操作时的命令触发寄存器。
在ISPEN(ISP_CONTR.7) = 1 时, 对ISP_TRIG先写入46h, 再写入B9h, ISP/IAP命令才会生效。

ISP/IAP操作完成后，ISP地址高八位寄存器ISP_ADDRH、ISP地址低八位寄存器ISP_ADDRL和ISP命令寄存器ISP_CMD的内容不变。如果接下来要对下一个地址的数据进行ISP/IAP操作，需手动将该地址的高8位和低8位分别写入ISP_ADDRH和ISP_ADDRL寄存器。

每次ISP/IAP操作时，都要对ISP_TRIG先写入46H，再写入B9H，ISP/IAP命令才会生效。

5. ISP/IAP命令寄存器ISP_CONTR

ISP/IAP控制寄存器ISP_CONTR格式如下：

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
ISP_CONTR	E7H	name	ISPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT2	WT0

ISPEN: ISP/IAP功能允许位。0：禁止IAP/ISP读/写/擦除Data Flash/EEPROM
1：允许IAP/ISP读/写/擦除Data Flash/EEPROM

SWBS：软件选择从用户应用程序区启动(送0)，还是从系统ISP监控程序区启动(送1)。
要与SWRST直接配合才可以实现

SWRST：0：不操作；　1：产生软件系统复位，硬件自动复位。

CMD_FAIL：如果送了ISP/IAP命令，并对ISP_TRIG送46h/B9h触发失败，则为1，需由软件清零.

```
;在用户应用程序区(AP区)软件复位并从用户应用程序区(AP区)开始执行程序
MOV ISP_CONTR, #00100000B ;SWBS = 0(选择AP区), SWRST = 1(软复位)
;在用户应用程序区(AP区)软件复位并从系统ISP监控程序区开始执行程序
MOV ISP_CONTR, #01100000B ;SWBS = 1(选择ISP区), SWRST = 1(软复位)
;在系统ISP监控程序区软件复位并从用户应用程序区(AP 区)开始执行程序
MOV ISP_CONTR, #00100000B ;SWBS = 0(选择AP 区), SWRST = 1(软复位)
;在系统ISP监控程序区软件复位并从系统ISP监控程序区开始执行程序
MOV ISP_CONTR, #01100000B ;SWBS = 1(选择ISP区), SWRST = 1(软复位)
```

设置等待时间			CPU等待时间(多少个CPU工作时钟)			
WT2	WT1	WT0	Read/读 (2个时钟)	Program/编程 (=55us)	Sector Erase 扇区擦除 (=21ms)	Recommended System Clock 跟等待参数对应的推荐系统时钟
1	1	1	2个时钟	55个时钟	21012个时钟	≤ 1MHz
1	1	0	2个时钟	110个时钟	42024个时钟	≤ 2MHz
1	0	1	2个时钟	165个时钟	63036个时钟	≤ 3MHz
1	0	0	2个时钟	330个时钟	126072个时钟	≤ 6MHz
0	1	1	2个时钟	660个时钟	252144个时钟	≤ 12MHz
0	1	0	2个时钟	1100个时钟	420240个时钟	≤ 20MHz
0	0	1	2个时钟	1320个时钟	504288个时钟	≤ 24MHz
0	0	0	2个时钟	1760个时钟	672384个时钟	≤ 30MHz

6. 工作电压过低判断，此时不要进行EEPROM/IAP操作

PCON寄存器定义如下：

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

LVDF: 低压检测标志位，当工作电压Vcc低于低压检测门槛电压时，该位置1。该位要由软件清0
当低压检测电路发现工作电压Vcc偏低时，不要进行EEPROM/IAP操作。

12.2 STC12C5410AD系列单片机EEPROM空间大小及地址

STC12C5410AD系列单片机内部可用Data Flash(EEPROM)的地址(与程序空间是分开的)：如果对应用程序区进行IAP写数据/擦除扇区的动作，则该语句会被单片机忽略，继续执行下一句。程序在用户应用程序区(AP区)时，仅可以对Data Flash(EEPROM)进行IAP/ISP操作。

STC12C5412AD/STC12LE5412AD等型号除外，这几个型号可在应用程序区修改应用程序

STC12C5410AD系列单片机内部EEPROM选型一览表 STC12LE5410AD系列单片机内部EEPROM选型一览表				
型号	EEPROM字节数	扇区数	起始扇区 首地址	结束扇区 末尾地址
STC12C5401AD/STC12LE5401AD	12K	24	0000h	2FFFh
STC12C5402AD/STC12LE5402AD	12K	24	0000h	2FFFh
STC12C5404AD/STC12LE5404AD	12K	24	0000h	2FFFh
STC12C5406AD/STC12LE5406AD	12K	24	0000h	2FFFh
STC12C5408AD/STC12LE5408AD	12K	24	0000h	2FFFh
STC12C5410AD/STC12LE5410AD	12K	24	0000h	2FFFh
以下系列特殊，可在用户程序区直接修改程序，所有Flash空间均可作EEPROM修改				
STC12C5412AD/STC12LE5412AD	-	24	0000h	2FFFh

STC12C5410AD单片机的内部EEPROM地址表 STC12LE5410AD单片机的内部EEPROM地址表								
第一扇区		第二扇区		第三扇区		第四扇区		每个扇区 512字节 建议同一次修改的数据放在同一扇区，不是同一次修改的数据放在不同的扇区，不必用满，当然可全用
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
0000h	1FFh	200h	3FFh	400h	5FFh	600h	7FFh	
第五扇区		第六扇区		第七扇区		第八扇区		
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
800h	9FFh	A00h	BFFh	C00h	DFFh	E00h	FFFh	
第九扇区		第十扇区		第十一扇区		第十二扇区		
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
1000h	11FFh	1200h	13FFh	1400h	15FFh	1600h	17FFh	
第十三扇区		第十四扇区		第十五扇区		第十六扇区		
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
1800h	19FFh	1A00h	1BFFh	1C00h	1DFFh	1E00h	1FFFh	
第十七扇区		第十八扇区		第十九扇区		第二十扇区		
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
2000h	21FFh	2200h	23FFh	2400h	25FFh	2600h	27FFh	
第二十一扇区		第二十二扇区		第二十三扇区		第二十四扇区		
起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	起始地址	结束地址	
2800h	29FFh	2A00h	2BFFh	2C00h	2DFFh	2E00h	2FFFH	

12.3 IAP及EEPROM汇编简介

```
;用DATA还是EQU声明新增特殊功能寄存器地址要看你用的汇编器/编译器
        ISP_DATA            DATA    0E2h;   或      ISP_DATA        EQU     0E2h
        ISP_ADDRH           DATA    0E3h;   或      ISP_ADDRH       EQU     0E3h
        ISP_ADDRL           DATA    0E4h;   或      ISP_ADDRL       EQU     0E4h
        ISP_CMD             DATA    0E5h;   或      ISP_CMD         EQU     0E5h
        ISP_TRIG            DATA    0E6h;   或      ISP_TRIG        EQU     0E6h
        ISP_CONTR           DATA    0E7h;   或      ISP_CONTR       EQU     0E7h
;定义ISP/IAP命令及等待时间
        ISP_IAP_BYTE_READ       EQU     1       ;字节读
        ISP_IAP_BYTE_PROGRAM    EQU     2       ;字节编程,前提是该字节是空，0FFh
        ISP_IAP_SECTOR_ERASE    EQU     3       ;扇区擦除,要某字节为空，要擦一扇区
        WAIT_TIME               EQU     0       ;设置等待时间,30MHz以下0,24M以下1,
                ;20MHz以下2,12M以下3,6M以下4,3M以下5,2M以下6,1M以下7,
;字节读
        MOV     ISP_ADDRH,      #BYTE_ADDR_HIGH         ;送地址高字节   地址需要改变时
        MOV     ISP_ADDRL,      #BYTE_ADDR_LOW          ;送地址低字节   才需重新送地址
        MOV     ISP_CONTR,      #WAIT_TIME      ;设置等待时间          此两句可以合成一句,
        ORL     ISP_CONTR,      #10000000B      ;允许ISP/IAP操作       并且只送一次就够了
        MOV     ISP_CMD,        #ISP_IAP_BYTE_READ
                                        ;送字节读命令,命令不需改变时,不需重新送命令
        MOV     ISP_TRIG,       #46h ;先送46h,再送B9h到ISP/IAP触发寄存器,每次都需如此
        MOV     ISP_TRIG,       #0B9h           ;送完B9h后，ISP/IAP命令立即被触发起动
;CPU等待IAP动作完成后，才会继续执行程序。
        NOP                             ;数据读出到ISP_DATA寄存器后，CPU继续执行程序
        MOV     A,      ISP_DATA                ;将读出的数据送往Acc
```

```
;以下语句可不用,只是出于安全考虑而已
        MOV     ISP_CONTR,     #00000000B     ;禁止ISP/IAP操作
        MOV     ISP_CMD,       #00000000B     ;去除ISP/IAP命令
        ;MOV    ISP_TRIG,      #00000000B     ;防止ISP/IAP命令误触发
        ;MOV    ISP_ADDRH,     #0FFh          ;送地址高字节单元为FFH,指向非EEPROM区
        ;MOV    ISP_ADDRL,     #0FFh          ;送地址低字节单元为FFH,防止误操作

;字节编程，该字节为FFh/空时，可对其编程，否则不行,要先执行扇区擦除
        MOV     ISP_DATA,      #ONE_DATA      ;送字节编程数据到ISP_DATA,
                                              ;只有数据改变时才需重新送
        MOV     ISP_ADDRH,     #BYTE_ADDR_HIGH       ;送地址高字节   ┐ 地址需要改变时
        MOV     ISP_ADDRL,     #BYTE_ADDR_LOW        ;送地址低字节   ┘ 才需重新送地址
        MOV     ISP_CONTR,     #WAIT_TIME            ;设置等待时间   ┐ 此两句可合成
        ORL     ISP_CONTR,     #10000000B            ;允许ISP/IAP操作 ┘ 一句，并且只
                                                                        送一次就够了
        MOV     ISP_CMD,       #ISP_IAP_BYTE_PROGRAM     ;送字节编程命令
        MOV     ISP_TRIG,      #46h   ;先送46h,再送B9h到ISP/IAP触发寄存器,每次都需如此
        MOV     ISP_TRIG,      #0B9h  ;送完B9h后，ISP/IAP命令立即被触发起动
;CPU等待IAP动作完成后，才会继续执行程序.
        NOP                                     ;字节编程成功后，CPU继续执行程序
;以下语句可不用,只是出于安全考虑而已
        MOV     ISP_CONTR,     #00000000B     ;禁止ISP/IAP操作
        MOV     ISP_CMD,       #00000000B     ;去除ISP/IAP命令
        ;MOV    ISP_TRIG,      #00000000B     ;防止ISP/IAP命令误触发
        ;MOV    ISP_ADDRH,     #0FFh  ;送地址高字节单元为FFH,;指向非EEPROM区,防止误操作
        ;MOV    ISP_ADDRL,     #0FFh  ;送地址低字节单元为FFH,指向非EEPROM区,防止误操作
```

```
;扇区擦除，没有字节擦除，只有扇区擦除，512字节/扇区,每个扇区用得越少越方便
;如果要对某个扇区进行擦除，而其中有些字节的内容需要保留，则需将其先读到单片机
;内部的RAM中保存，再将该扇区擦除，然后将须保留的数据写回该扇区，所以每个扇区
;中用的字节数越少越好，操作起来越灵活越快.
;扇区中任意一个字节的地址都是该扇区的地址,无需求出首地址.

        MOV     ISP_ADDRH,      #SECTOR_FIRST_BYTE_ADDR_HIGH ;送扇区起始地址高字节
        MOV     ISP_ADDRL,      #SECTOR_FIRST_BYTE_ADDR_LOW  ;送扇区起始地址低字节
                                                    ;地址需要改变时才需重新送地址
        MOV     ISP_CONTR,      #WAIT_TIME          ;设置等待时间   此两句可以合
        ORL     ISP_CONTR,      #10000000B          ;允许ISP/IAP    成一句,并且只
                                                                    送一次就够了
        MOV     ISP_CMD,        #ISP_IAP_SECTOR_ERASE
                                    ;送扇区擦除命令,命令不需改变时,不需重新送命令
        MOV     ISP_TRIG,       #46h
                                ;先送46h,再送B9h到ISP/IAP触发寄存器,每次都需如此
        MOV     ISP_TRIG,       #0B9h       ;送完B9h后，ISP/IAP命令立即被触发起动
;CPU等待IAP动作完成后，才会继续执行程序.
        NOP                                 ;扇区擦除成功后，CPU继续执行程序
;以下语句可不用,只是出于安全考虑而已
        MOV     ISP_CONTR,      #00000000B          ;禁止ISP/IAP操作
        MOV     ISP_CMD,        #00000000B          ;去除ISP/IAP命令
        ;MOV    ISP_TRIG,       #00000000B          ;防止ISP/IAP命令误触发
        ;MOV    ISP_ADDRH,      #0FFh           ;送地址高字节单元为FFH,指向非EEPROM区
        ;MOV    ISP_ADDRL,      #0FFh           ;送地址低字节单元为FFH,防止误操作
```

小常识：（STC单片机的Data Flash 当EEPROM功能使用）

3个基本命令----字节读，字节编程，扇区擦除

字节编程：将“1”写成“1”或“0”，将“0”写成“0”。如果某字节是FFH,才可对其进行字节编程。如果该字节不是FFH，则须先将整个扇区擦除，因为只有“扇区擦除”才可以将“0”变为“1”。

扇区擦除：只有“扇区擦除”才可能将“0”擦除为“1”。

大建议：

1.同一次修改的数据放在同一扇区中，不是同一次修改的数据放在另外的扇区,就不须读出保护。

2.如果一个扇区只用一个字节，那就是真正的EEPROM,STC单片机的Data Flash比外部EEPROM要快很多，读一个字节/编程一个字节大概是2个时钟/55uS。

3.如果在一个扇区中存放了大量的数据，某次只需要修改其中的一个字节或一部分字节时，则另外的不需要修改的数据须先读出放在STC单片机的RAM中，然后擦除整个扇区，再将需要保留的数据和需修改的数据按字节逐字节写回该扇区中(只有字节写命令，无连续字节写命令)。这时每个扇区使用的字节数是使用的越少越方便(不需读出一大堆需保留数据)。

常问的问题：

1：IAP指令完成后，地址是否会自动“加1”或“减1”？

答：不会

2：送46和B9触发后，下一次IAP命令是否还需要送46和B9触发？

答：是，一定要。

12.4 EEPROM测试程序

1. C程序：

```
;STC12C5410AD系列单片机EEPROM/IAP 功能测试程序演示
/*------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机 EEPROM/IAP功能--------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ---------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*------------------------------------------------------------------------------*/

#include "reg51.h"
#include "intrins.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the IAP */
sfr      IAP_DATA       =  0xE2;       //Flash data register
sfr      IAP_ADDRH      =  0xE3;       //Flash address HIGH
sfr      IAP_ADDRL      =  0xE4;       //Flash address LOW
sfr      IAP_CMD        =  0xE5;       //Flash command register
sfr      IAP_TRIG       =  0xE6;       //Flash command trigger
sfr      IAP_CONTR      =  0xE7;       //Flash control register

/*Define ISP/IAP/EEPROM command*/
#define  CMD_IDLE       0              //Stand-By
#define  CMD_READ       1              //Byte-Read
#define  CMD_PROGRAM 2                 //Byte-Program
#define  CMD_ERASE      3              //Sector-Erase

/*Define ISP/IAP/EEPROM operation const for IAP_CONTR*/
//#define ENABLE_IAP    0x80           //if SYSCLK<30MHz
//#define ENABLE_IAP    0x81           //if SYSCLK<24MHz
#define  ENABLE_IAP     0x82           //if SYSCLK<20MHz
//#define ENABLE_IAP    0x83           //if SYSCLK<12MHz
//#define ENABLE_IAP    0x84           //if SYSCLK<6MHz
//#define ENABLE_IAP    0x85           //if SYSCLK<3MHz
//#define ENABLE_IAP    0x86           //if SYSCLK<2MHz
//#define ENABLE_IAP    0x87           //if SYSCLK<1MHz

//Start address for STC12C5410AD series EEPROM
#define   IAP_ADDRESS   0x0400

void Delay(BYTE n);
void IapIdle();
BYTE IapReadByte(WORD addr);
```

```
void IapProgramByte(WORD addr, BYTE dat);
void IapEraseSector(WORD addr);

void main()
{
        WORD i;

        P1 = 0xfe;                              //1111,1110 System Reset OK
        Delay(10);                              //Delay
        IapEraseSector(IAP_ADDRESS);            //Erase current sector
        for (i=0; i<512; i++)                   //Check whether all sector data is FF
        {
                if (IapReadByte(IAP_ADDRESS+i) != 0xff)
                goto Error;                     //If error, break
        }
        P1 = 0xfc;                              //1111,1100 Erase successful
        Delay(10);                              //Delay
        for (i=0; i<512; i++)                   //Program 512 bytes data into data flash
        {
                IapProgramByte(IAP_ADDRESS+i, (BYTE)i);
        }
        P1 = 0xf8;                              //1111,1000 Program successful
        Delay(10);                              //Delay
        for (i=0; i<512; i++)                   //Verify 512 bytes data
        {
                if (IapReadByte(IAP_ADDRESS+i) != (BYTE)i)
                goto Error;                     //If error, break
        }
        P1 = 0xf0;                              //1111,0000 Verify successful
        while (1);
Error:
        P1 &= 0x7f;                             //0xxx,xxxx IAP operation fail
        while (1);
}

/*----------------------------
Software delay function
----------------------------*/
void Delay(BYTE n)
{
        WORD x;

        while (n--)
        {
                x = 0;
                while (++x);
        }
}
```

```
/*----------------------------
Disable ISP/IAP/EEPROM function
Make MCU in a safe state
----------------------------*/
void IapIdle()
{
        IAP_CONTR = 0;                  //Close IAP function
        IAP_CMD = 0;                    //Clear command to standby
        IAP_TRIG = 0;                   //Clear trigger register
        IAP_ADDRH = 0x80;               //Data ptr point to non-EEPROM area
        IAP_ADDRL = 0;                  //Clear IAP address to prevent misuse
}

/*----------------------------
Read one byte from ISP/IAP/EEPROM area
Input: addr (ISP/IAP/EEPROM address)
Output:Flash data
----------------------------*/
BYTE IapReadByte(WORD addr)
{
        BYTE dat;                       //Data buffer

        IAP_CONTR = ENABLE_IAP;         //Open IAP function, and set wait time
        IAP_CMD = CMD_READ;             //Set ISP/IAP/EEPROM READ command
        IAP_ADDRL = addr;               //Set ISP/IAP/EEPROM address low
        IAP_ADDRH = addr >> 8;          //Set ISP/IAP/EEPROM address high
        IAP_TRIG = 0x46;                //Send trigger command1 (0x46)
        IAP_TRIG = 0xb9;                //Send trigger command2 (0xb9)
        _nop_();                        //MCU will hold here until ISP/IAP/EEPROM
                                        //operation complete
        dat = IAP_DATA;                 //Read ISP/IAP/EEPROM data
        IapIdle();                      //Close ISP/IAP/EEPROM function

        return dat;                     //Return Flash data
}

/*----------------------------
Program one byte to ISP/IAP/EEPROM area
Input: addr (ISP/IAP/EEPROM address)
    dat (ISP/IAP/EEPROM data)
Output:-
----------------------------*/
```

```
void IapProgramByte(WORD addr, BYTE dat)
{
        IAP_CONTR = ENABLE_IAP;         //Open IAP function, and set wait time
        IAP_CMD = CMD_PROGRAM;          //Set ISP/IAP/EEPROM PROGRAM command
        IAP_ADDRL = addr;               //Set ISP/IAP/EEPROM address low
        IAP_ADDRH = addr >> 8;          //Set ISP/IAP/EEPROM address high
        IAP_DATA = dat;                 //Write ISP/IAP/EEPROM data
        IAP_TRIG = 0x46;                //Send trigger command1 (0x46)
        IAP_TRIG = 0xb9;                //Send trigger command2 (0xb9)
        _nop_();                        //MCU will hold here until ISP/IAP/EEPROM
                                        //operation complete
        IapIdle();
}

/*----------------------------
Erase one sector area
Input: addr (ISP/IAP/EEPROM address)
Output:-
----------------------------*/
void IapEraseSector(WORD addr)
{
        IAP_CONTR = ENABLE_IAP;         //Open IAP function, and set wait time
        IAP_CMD = CMD_ERASE;            //Set ISP/IAP/EEPROM ERASE command
        IAP_ADDRL = addr;               //Set ISP/IAP/EEPROM address low
        IAP_ADDRH = addr >> 8;          //Set ISP/IAP/EEPROM address high
        IAP_TRIG = 0x46;                //Send trigger command1 (0x46)
        IAP_TRIG = 0xb9;                //Send trigger command2 (0xb9)
        _nop_();                        //MCU will hold here until ISP/IAP/EEPROM
                                        //operation complete
        IapIdle();
}
```

2. 汇编程序：

```
;STC12C5410AD系列单片机EEPROM/IAP 功能测试程序演示
/*---------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* --- 演示STC 1T 系列单片机  EEPROM/IAP功能--------------------------*/
/* 如果要在程序中使用或在文章中引用该程序， ------------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/

;/*Declare SFRs associated with the IAP */
IAP_DATA        EQU     0E2H            ;Flash data register
IAP_ADDRH       EQU     0E3H            ;Flash address HIGH
IAP_ADDRL       EQU     0E4H            ;Flash address LOW
IAP_CMD         EQU     0E5H            ;Flash command register
IAP_TRIG        EQU     0E6H            ;Flash command trigger
IAP_CONTR       EQU     0E7H            ;Flash control register

;/*Define ISP/IAP/EEPROM command*/
CMD_IDLE        EQU     0               ;Stand-By
CMD_READ        EQU     1               ;Byte-Read
CMD_PROGRAM     EQU     2               ;Byte-Program
CMD_ERASE       EQU     3               ;Sector-Erase

;/*Define ISP/IAP/EEPROM operation const for IAP_CONTR*/
;ENABLE_IAP     EQU     80H             ;if SYSCLK<30MHz
;ENABLE_IAP     EQU     81H             ;if SYSCLK<24MHz
ENABLE_IAP      EQU     82H             ;if SYSCLK<20MHz
;ENABLE_IAP     EQU     83H             ;if SYSCLK<12MHz
;ENABLE_IAP     EQU     84H             ;if SYSCLK<6MHz
;ENABLE_IAP     EQU     85H             ;if SYSCLK<3MHz
;ENABLE_IAP     EQU     86H             ;if SYSCLK<2MHz
;ENABLE_IAP     EQU     87H             ;if SYSCLK<1MHz

;//Start address for STC12C5410AD series EEPROM
IAP_ADDRESS EQU 0000H
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     P1,     #0FEH           ;1111,1110 System Reset OK
        LCALL   DELAY                   ;Delay
```

```
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS            ;Set ISP/IAP/EEPROM address
        LCALL    IAP_ERASE                      ;Erase current sector
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS            ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                      ;Set counter (512)
        MOV     R1,     #2
CHECK1:                                         ;Check whether all sector data is FF
        LCALL   IAP_READ                        ;Read Flash
        CJNE    A,      #0FFH,  ERROR           ;If error, break
        INC     DPTR                            ;Inc Flash address
        DJNZ    R0,     CHECK1                  ;Check next
        DJNZ    R1,     CHECK1                  ;Check next
;-------------------------------
        MOV     P1,     #0FCH                   ;1111,1100 Erase successful
        LCALL   DELAY                           ;Delay
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS            ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                      ;Set counter (512)
        MOV     R1,     #2
        MOV     R2,     #0                      ;Initial test data
NEXT:                                           ;Program 512 bytes data into data flash
        MOV     A,      R2                      ;Ready IAP data
        LCALL   IAP_PROGRAM                     ;Program flash
        INC     DPTR                            ;Inc Flash address
        INC     R2                              ;Modify test data
        DJNZ    R0,     NEXT                    ;Program next
        DJNZ    R1,     NEXT                    ;Program next
;-------------------------------
        MOV     P1,     #0F8H                   ;1111,1000 Program successful
        LCALL   DELAY                           ;Delay
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS            ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                      ;Set counter (512)
        MOV     R1,     #2
        MOV     R2,     #0
CHECK2:                                         ;Verify 512 bytes data
        LCALL   IAP_READ                        ;Read Flash
        CJNE    A,   2,   ERROR                 ;If error, break
        INC     DPTR                            ;Inc Flash address
        INC     R2                              ;Modify verify data
        DJNZ    R0,     CHECK2                  ;Check next
        DJNZ    R1,     CHECK2                  ;Check next
;-------------------------------
        MOV     P1,     #0F0H                   ;1111,0000 Verify successful
        SJMP    $
;-------------------------------
```

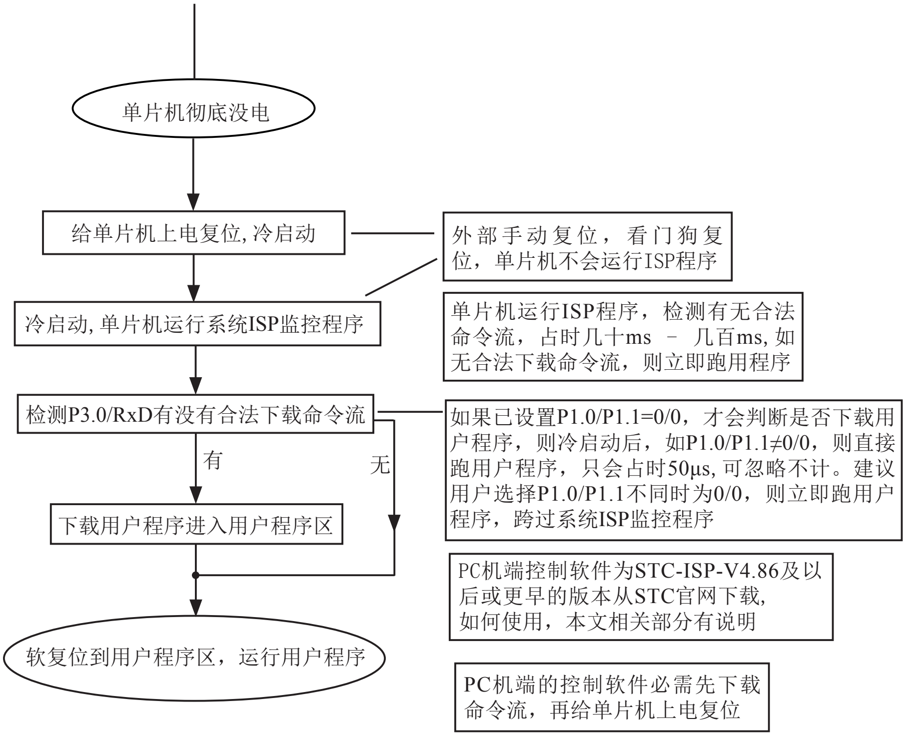
```
ERROR:
        MOV     P0,     R0
        MOV     P2,     R1
        MOV     P3,     R2
        CLR     P1.7                    ;0xxx,xxxx IAP operation fail
        SJMP    $

;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
        MOV     R2,     #20H
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

;/*----------------------------
;Disable ISP/IAP/EEPROM function
;Make MCU in a safe state
;----------------------------*/
IAP_IDLE:
        MOV     IAP_CONTR,      #0              ;Close IAP function
        MOV     IAP_CMD,        #0              ;Clear command to standby
        MOV     IAP_TRIG,       #0              ;Clear trigger register
        MOV     IAP_ADDRH,      #80H            ;Data ptr point to non-EEPROM area
        MOV     IAP_ADDRL,      #0              ;Clear IAP address to prevent misuse
        RET

;/*----------------------------
;Read one byte from ISP/IAP/EEPROM area
;Input: DPTR(ISP/IAP/EEPROM address)
;Output:ACC (Flash data)
;----------------------------*/
IAP_READ:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_READ       ;Set ISP/IAP/EEPROM READ command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV     IAP_TRIG,       #46H            ;Send trigger command1 (0x46)
        MOV     IAP_TRIG,       #0B9H           ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        MOV     A,      IAP_DATA                ;Read ISP/IAP/EEPROM data
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET
```

```
;/*----------------------------
;Program one byte to ISP/IAP/EEPROM area
;Input: DPAT(ISP/IAP/EEPROM address)
;ACC (ISP/IAP/EEPROM data)
;Output:-
;----------------------------*/
IAP_PROGRAM:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_PROGRAM    ;Set ISP/IAP/EEPROM PROGRAM command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV     IAP_DATA,       A               ;Write ISP/IAP/EEPROM data
        MOV     IAP_TRIG,       #46H            ;Send trigger command1 (0x46)
        MOV     IAP_TRIG,       #0B9H           ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET

;/*----------------------------
;Erase one sector area
;Input: DPTR(ISP/IAP/EEPROM address)
;Output:-
;----------------------------*/
IAP_ERASE:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_ERASE      ;Set ISP/IAP/EEPROM ERASE command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV   IAP_TRIG,#46H                     ;Send trigger command1 (0x46)
        MOV   IAP_TRIG,#0B9H                    ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET

        END
```

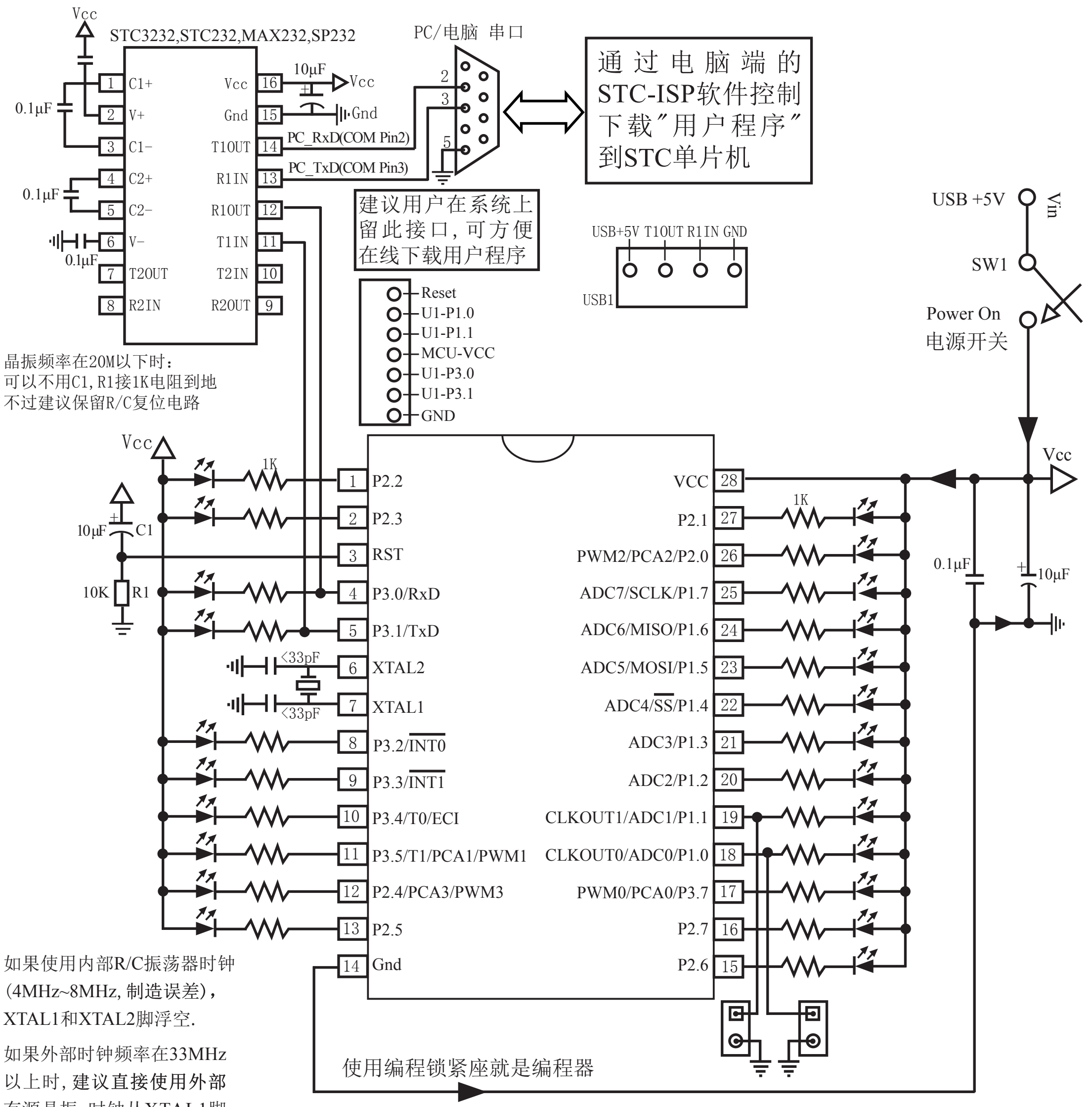
第13章 STC12系列单片机开发/编程工具说明

13.1 在系统可编程(ISP)原理，官方演示工具使用说明

13.1.1 在系统可编程(ISP)原理使用说明



13.1.2 STC12C5410AD系列在系统可编程(ISP)典型应用线路图



此线路已做成一个STC12C5410AD系列单片机ISP下载编程工具

若客户无USB转换线，STC提供第三方生产的USB-RS232转换线，人民币20元每条。

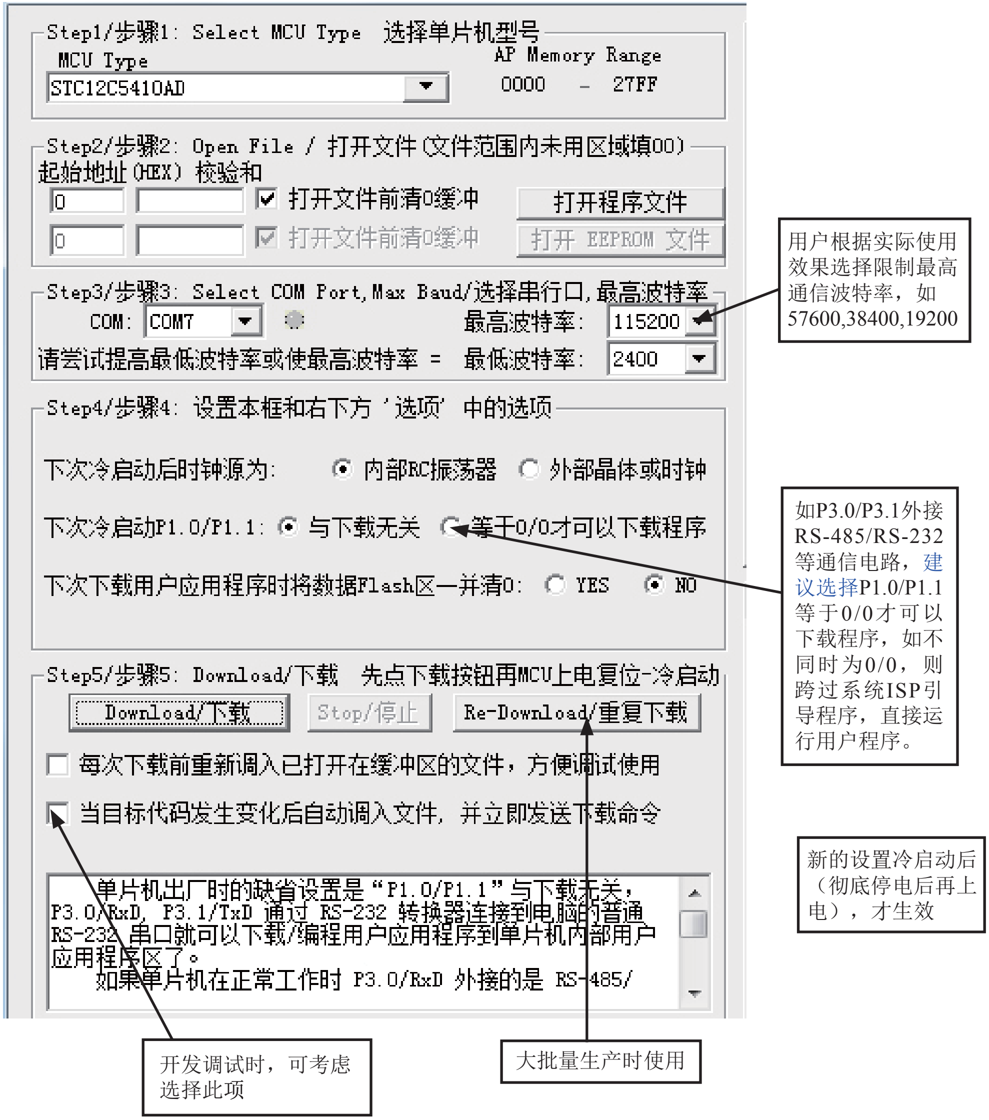
STC12C5410AD系列单片机具有在系统可编程(ISP)特性，ISP的好处是：省去购买通用编程器，单片机在用户系统上即可下载/烧录用户程序，而无须将单片机从已生产好的产品上拆下，再用通用编程器将程序代码烧录进单片机内部。有些程序尚未定型的产品可以一边生产，一边完善，加快了产品进入市场的速度，减小了新产品由于软件缺陷带来的风险。由于可以在用户的目标系统上将程序直接下载进单片机看运行结果对错，故无须仿真器。

STC12系列单片机内部固化有ISP系统引导固件，配合PC端的控制程序即可将用户的程序代码下载进单片机内部，故无须编程器(速度比通用编程器快，几秒一片) 。

如何获得及使用STC 提供的ISP 下载工具（STC-ISP.exe 软件）：

(1). 获得STC提供的ISP下载工具（软件）
登陆STC官方网站，从STC半导体专栏下载PC（电脑）端的ISP程序，然后将其自解压，再安装即可（执行setup.exe）， 注意随时更新软件。
(2). 使用STC-ISP下载工具（软件）,请随时更新，目前已到Ver4.86版本以上,
支持*.bin,*.hex(Intel 16 进制格式)文件，少数*.hex 文件不支持的话，请转换成*.bin 文件，请随时注意升级PC（电脑）端的STC-ISP.EXE 程序。
(3).STC12系列单片机出厂时就已完全加密。需要单片机内部的电放光后上电复位(冷起动)才运行系统ISP程序，如从 P3.0检测到合法的下载命令流就下载用户程序，如检测不到就复位到用户程序区，运行用户程序。
(4).如果用户板上P3.0，P3.1接了RS-485等电路，下载时需要将其断开。用户系统接了RS-485等通信电路，推荐在选项中选择“下次冷启动时需P1.0/P1.1=0/0才可以下载程序”

13.1.3 电脑端的ISP控制软件界面使用说明



Step1/步骤1：选择你所使用的单片机型号，如STC12C5A60X等

Step2/步骤2：打开文件，要烧录用户程序，必须调入用户的程序代码（*.bin, *.hex）

Step3/步骤3：选择串行口，你所使用的电脑串口，如串行口1--COM1,串行口2--COM2,...

有些新式笔记本电脑没有RS-232串行口,可买一条USB-RS232转接器，人民币50元左右。

有些USB-RS232转接器，不能兼容，可让STC帮你购买经过测试的转换器。

Step4/步骤4：选择下次冷启动后，时钟源为“内部R/C振荡器”还是“外部晶体或时钟”

(STC12系列单片机只有内部R/C振荡时钟)

Step5/步骤5：选择“Download/下载”按钮下载用户的程序进单片机内部，可重复执行

Step5/步骤5， 也可选择“Re-Download/重复下载”按钮

下载时注意看提示，主要看是否要给单片机上电或复位，下载速度比一般通用编程器快。一定要先选择“Download/下载”按钮，然后再给单片机上电复位(先彻底断电)，而不要先上电，先上电，检测不到合法的下载命令流，单片机就直接跑用户程序了。

关于硬件连接：

(1). MCU/单片机 RXD(P3.0) --- RS-232转换器 --- PC/电脑 TXD(COM Port Pin3)

(2). MCU/单片机 TXD(P3.1) --- RS-232转换器 --- PC/电脑 RXD(COM Port Pin2)

(3). MCU/单片机 GND ---------------------- PC/电脑 GND(COM Port Pin5)

(4). 如果您的系统P3.0/P3.1连接到RS-485电路，推荐

在选项里选择“下次冷启动需要P1.0/P1.1 = 0,0才可以下载用户程序”

这样冷启动后如 P1.0, P1.1不同时为0,单片机直接运行用户程序，免得由于RS-485总线上的乱码造成单片机反复判断乱码是否为合法，浪费几百mS的时间，其实如果你的系统本身P3.0,P3.1就是做串口使用，也建议选择P1.0/P1.1 = 0/0才可下载用户程序，以便下次冷启动直接运行用户程序。

(5). RS-232转换器可选用MAX232/SP232(4.5-5.5V),MAX3232/SP3232(3V-5.5V).

13.1.4 STC-ISP下载编程工具硬件使用说明

如用户系统没有RS-232接口，

可使用STC-ISP Ver 3.0A.PCB演示板作为编程工具

STC-ISP Ver 3.0A PCB板可以焊接3种电路，分别支持STC12系列16Pin /20Pin / 28Pin / 32Pin。我们在下载板的反面贴了一张标签纸，说明它是支持16Pin /20Pin / 28Pin / 32Pin中的哪一种，用户要特别注意。在正面焊的编程烧录用锁紧座都是40Pin的，锁紧座第20-Pin接的是地线，请将单片机的地线对着锁紧座的地线插。

在STC-ISP Ver 3.0A PCB 板完成下载编程用户程序的工作：

关于硬件连接：

(1). 根据单片机的工作电压选择单片机电源电压

A. 5V单片机, 短接JP1的MCU-VCC, +5V电源管脚

B. 3V单片机, 短接JP1的MCU-VCC, 3.3V电源管脚

(2). 连接线(STC提供)

A. 将一端有9芯连接座的插头插入PC/电脑RS-232串行接口插座用于通信

B. 将同一端的USB插头插入PC/电脑USB接口用于取电

C. 将只有一个USB插头的一端插入STC-ISP Ver 3.0A PCB板USB1插座用于RS-232通信和供电, 此时USB +5V Power灯亮(D43, USB接口有电)

(3). 其他插座不需连接

(4). SW1开关处于非按下状态，此时MCU-VCC Power灯不亮(D41)，没有给单片机通电

(5). SW3开关

处于非按下状态，P1.0, P1.1 = 1, 1,不短接到地。

处于按下状态， P1.0, P1.1 = 0, 0, 短接到地。

如果单片机已被设成“下次冷启动P1.0/P1.1 = 0,0才判P3.0有无合法下载命令流”就必须将SW3开关处于按下状态，让单片机的P1.0/P1.1短接到地

(6). 将单片机插进U1-Socket锁紧座，锁紧单片机, 注意单片机是8-Pin/20-Pin/28-Pin, 而U1-Socket锁紧座是40-Pin, 我们的设计是靠下插，靠近晶体的那一端插。

(7). 关于软件：选择“Download/下载”(必须在给单片机上电之前让PC先发一串合法下载命令)

(8). 按下SW1开关，给单片机上电复位，此时MCU-VCC Power灯亮(D41)

此时STC单片机进入ISP模式(STC12系列冷启动进入ISP)

(9). 下载成功后，再按SW1开关，此时SW1开关处于非按下状态，MCU-VCC Power灯不亮(D41)，给单片机断电，取下单片机，换上新的单片机。

13.1.5 若无RS-232转换器，如何用ISP下载板做RS-232通信转换

利用STC-ISP Ver 3.0A PCB 板进行RS-232转换
单片机在用户自己的板上完成下载/烧录：

1. U1-Socket锁紧座不得插入单片机
2. 将用户系统上的电源(MCU-VCC,GND)及单片机的P3.0,P3.1接入转换板CN2插座
   这样用户系统上的单片机就具备了与PC/电脑进行通信的能力
3. 将用户系统的单片机的P1.0,P1.1接入转换板CN2插座(如果需要的话)
4. 如须P1.0, P1.1 = 0, 0, 短接到地，可在用户系统上将其短接到地，或将P1.0/P1.1也从用户系统上引到STC-ISP Ver3.0A PCB 板上，将SW3开关按下，则P1.0/P1.1=0,0。
5. 关于软件：选择“Download/下载”
6. 给单片机系统上电复位(注意是从用户系统自供电，不要从电脑USB取电,电脑USB座不插)
7. 下载程序时，如用户板有外部看门狗电路，不得启动，单片机必须有正确的复位,但不能在ISP下载程序时被外部看门狗复位,如有，可将外部看门狗电路WDI端/或WDO端浮空。
8. 如有RS-485晶片连到P3.0,P3.1,或其他线路，在下载时应将其断开。

13.2 编译器/汇编器，编程器，仿真器

STC 单片机应使用何种编译器/汇编器:

1.任何老的编译器/汇编器都可以支持，流行用Keil C51

2.把STC单片机，当成Intel的8052/87C52/87C54/87C58,Philips的P87C52/P87C54/P87C58就可以了.

3.如果要用到扩展的专用特殊功能寄存器，直接对该地址单元设置就行了，当然先声明特殊功能寄存器的地址较好。

编程烧录器:

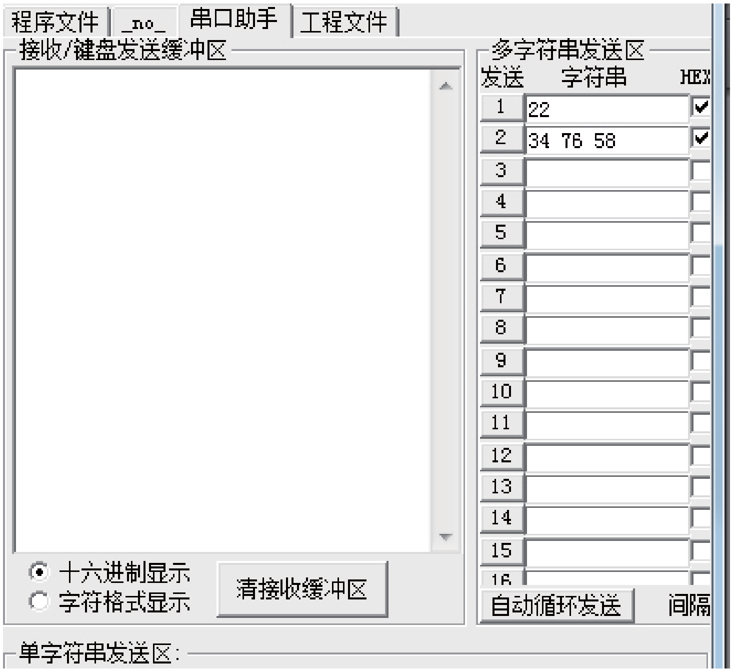
我们有: STC12C5410AD系列ISP经济型下载编程工具(人民币50元，可申请免费样品)

注意:有专门下载28PIN/20PIN的不同演示板,

28PIN是28PIN的演示板,20PIN是20PIN的演示板

仿真器:如您已有老的仿真器，可仿真普通8052的基本功能

STC12C5410AD系列单片机扩展功能如它仿不了,可以用 STC-ISP.EXE 直接下载用户程序看运行结果就可以了,如需观察变量,可自己写一小段测试程序通过串口输出到电脑端的STC-ISP.EXE的“串口助手”来显示,也很方便。无须添加新的设备.



无仿真器如何调试/开发用户程序

1 ． 首先参照本手册当中的“用定时器1做波特率发生器”， 调通串口程序，这样，要观察变量就可以自己写一小段测试程序将变量通过串口输出到电脑端的STC-ISP.EXE的“串口调试助手”来显示,也很方便。

2.调通按键扫描程序(到处都有大量的参考程序)

3.调通用户系统的显示电路程序，此时变量/寄存器也可以通过用户系统的显示电路显示了

4.调通A/D检测电路(我们用户手册里面有完整的参考程序)

5.调通PWM 等电路(我们用户手册里面有完整的参考程序)

这样分步骤模块化调试用户程序，有些系统，熟练的8051用户，三天就可以调通了，难度不大的系统，一般一到二周就可以调通。

用户的串口输出显示程序可以在输出变量/寄存器的值之后，继续全速运行用户程序，也可以等待串口送来的“继续运行命令”，方可继续运行用户程序，这就相当于断点。这种断点每设置一个地方，就必须调用一次该显示寄存器/变量的程序，有点麻烦，但却很实用。

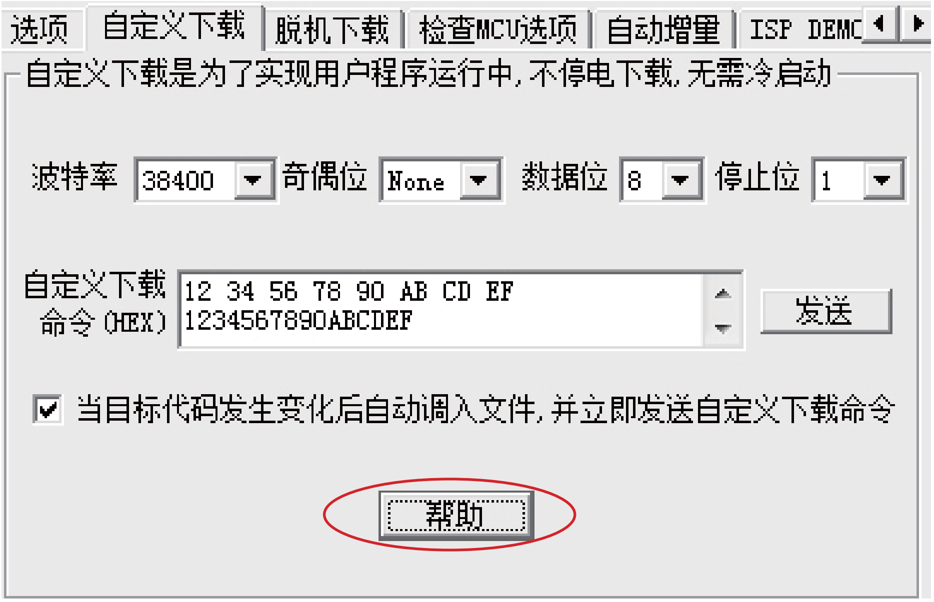
13.3 自定义下载演示程序(实现不停电下载)

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU Limited ---------------- ---------------------------------------*/
/* ---  演示STC 1T 系列单片机  利用软件实现自定义下载--------------*/
/* 如果要在程序中使用或在文章中引用该程序，  -----------------------*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/

#include <reg51.h>
#include <instrins.h>

sfr IAP_CONTR = 0xc7;
sbit MCU_Start_Led = P1^7;

#define Self_Define_ISP_Download_Command 0x22
#define RELOAD_COUNT 0xfb          //18.432MHz,12T,SMOD=0,9600bps
//#define RELOAD_COUNT 0xf6        //18.432MHz,12T,SMOD=0,4800bps
//#define RELOAD_COUNT 0xec        //18.432MHz,12T,SMOD=0,2400bps
//#define RELOAD_COUNT 0xd8        //18.432MHz,12T,SMOD=0,1200bps

void serial_port_initial(void);
void send_UART(unsigned char);
void UART_Interrupt_Receive(void);
void soft_reset_to_ISP_Monitor(void);
void delay(void);
void display_MCU_Start_Led(void);

void main(void)
{
        unsigned char i = 0;

        serial_port_initial();                  //Initial UART
        display_MCU_Start_Led();                //Turn on the work LED
        send_UART(0x34);                        //Send UART test data
        send_UART(0xa7);                        // Send UART test data
        while (1);
}
```

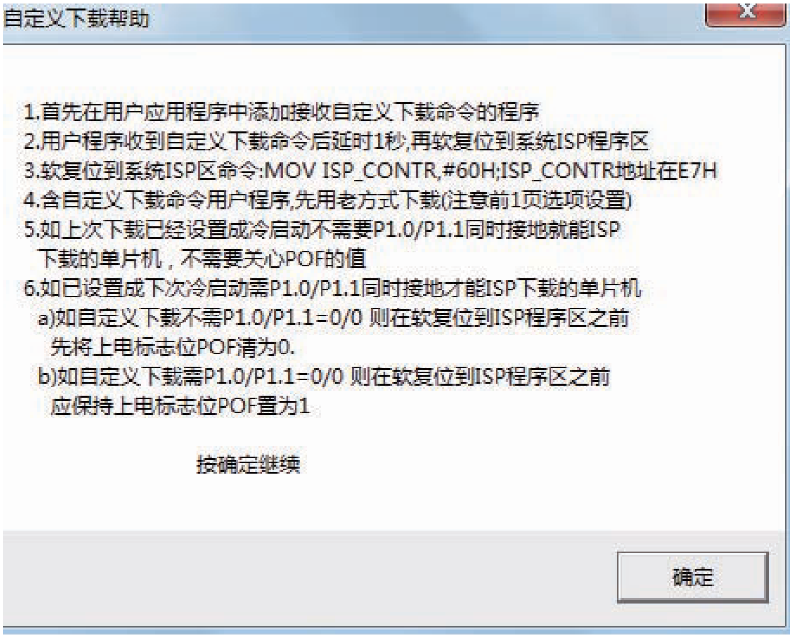
```
void send_UART(unsigned char i)
{
        ES = 0;                         //Disable serial interrupt
        TI = 0;                         //Clear TI flag
        SBUF = i;                       //send this data
        while (!TI);                    //wait for the data is sent
        TI = 0;                         //clear TI flag
        ES = 1;                         //enable serial interrupt
}

void UART_Interrupt)Receive(void) interrupt 4 using 1
{
        unsigned char k = 0;
        if (RI)
        {
                RI = 0;
                k = SBUF;
                if (k == Self_Define_ISP_Command)       //check the serial data
                {
                        delay();                        //delay 1s
                        delay();                        //delay 1s
                        soft_reset_to_ISP_Monitor();
                }
        }
        if (TI)
        {
                TI = 0;
        }
}

void soft_reset_to_ISP_Monitor(void)
{
        IAP_CONTR = 0x60;               //0110,0000 soft reset system to run ISP monitor
}

void delay(void)
{
        unsigned int j = 0;
        unsigned int g = 0;
```

```
	for (j=0; j<5; j++)
	{
		for (g=0; g<60000; g++)
		{
			_nop_();
			_nop_();
			_nop_();
			_nop_();
			_nop_();
		}
	}
}

void display_MCU_Start_Led(void)
{
	unsigned char i = 0;
	for (i=0; i<3; i++)
	{
		MCU_Start_Led = 0;		//Turn on work LED
		dejay();
		MCU_Start_Led = 1;		//Turn off work LED
		dejay();
		MCU_Start_Led = 0;		//Turn on work LED
	}
}
```

自定义下载在STC的电脑端ISP软件STC-ISP.EXE 中，还应做相应设置，具体参考设置见下图：



点击软件STC-ISP.EXE中的帮助按钮（如上图所示），可见详细的帮助说明，如下图所示



附录A：汇编语言编程

INTRODUCTION

Assembly language is a computer language lying between the extremes of machine language and high-level language like Pascal or C use words and statements that are easily understood by humans, although still a long way from "natural" language.Machine language is the binary language of computers.A machine language program is a series of binary bytes representing instructions the computer can execute.

Assembly language replaces the binary codes of machine language with easy to remember "mnemonics"that facilitate programming.For example, an addition instruction in machine language might be represented by the code "10110011".It might be represented in assembly language by the mnemonic "ADD".Programming with mnemonics is obviously preferable to programming with binary codes.

Of course, this is not the whole story. Instructions operate on data, and the location of the data is specified by various "addressing modes" emmbeded in the binary code of the machine language instruction. So, there may be several variations of the ADD instruction, depending on what is added. The rules for specifying these variations are central to the theme of assembly language programming.

An assembly language program is not executable by a computer. Once written, the program must undergo translation to machine language. In the example above, the mnemonic "ADD" must be translated to the binary code "10110011". Depending on the complexity of the programming environment, this translation may involve one or more steps before an executable machine language program results. As a minimum, a program called an "assembler" is required to translate the instruction mnemonics to machine language binary codes. Afurther step may require a "linker" to combine portions of program from separate files and to set the address in memory at which th program may execute. We begin with a few definitions.

An assembly language program i a program written using labels, mnemonics, and so on, in which each statement corresponds to a machine instruction. Assembly language programs, often called source code or symbolic code, cannot be executed by a computer.

A machine language program is a program containing binary codes that represent instructions to a computer. Machine language programs, often called object code, are executable by a computer.

A assembler is a program that translate an assembly language program into a machine language program. The machine language program (object code) may be in "absolute" form or in "relocatable" form. In the latter case, "linking" is required to set the absolute address for execution.

A linker is a program that combines relocatable object programs (modules) and produces an absolute object program that is executable by a computer. A linker is sometimes called a "linker/locator" to reflect its separate functions of combining relocatable modules (linking) and setting the address for execution (locating).

A segment is a unit of code or data memory. A segment may be relocatable or absolute. A relocatable segment has a name, type, and other attributes that allow the linker to combine it with other paritial segments, if required, and to correctly locate the segment. An absolute segment has no name and cannot be combined with other segments.

A module contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. A module may be thought of as a "file" in many instances.

A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules. A program contains only the binary codes for instructions (with address and data constants) that are understood by a computer.

ASSEMBLER OPERATION

There are many assembler programs and other support programs available to facilitate the development of applications for the 8051 microcontroller. Intel's original MCS-51 family assembler, ASM51, is no longer available commercially. However, it set the standard to which the others are compared.

ASM51 is a powerful assembler with all the bells and whistles. It is available on Intel development systems and on the IBM PC family of microcomputers. Since these "host" computers contain a CPU chip other than the 8051, ASM51 is called a cross assembler. An 8051 source program may be written on the host computer (using any text editor) and may be assembled to an object file and listing file (using ASM51), but the program may not be executed. Since the host system's CPU chip is not an 8051, it does not understand the binary instruction in the object file. Execution on the host computer requires either hardware emulation or software simulation of the target CPU. A third possibility is to download the object program to an 8051-based target system for execution.

ASM51 is invoked from the system prompt by

ASM51 source_file [assembler_controls]

The source file is assembled and any assembler controls specified take effect. The assembler receives a source file as input (e.g., PROGRAM.SRC) and generates an object file (PROGRAM.OBJ) and listing file (PROGRAM.LST) as output. This is illustrated in Figure 1.

Since most assemblers scan the source program twice in performing the translation to machine language, they are described as two-pass assemblers. The assembler uses a location counter as the address of instructions and the values for labels. The action of each pass is described below.

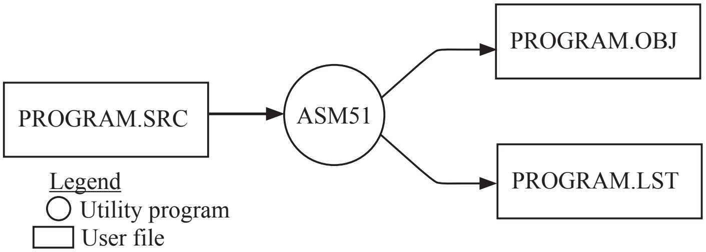


Figure 1 Assembling a source program

Pass one

During the first pass, the source file is scanned line-by-line and a symbol table is built. The location counter defaults to 0 or is set by the ORG (set origin) directive. As the file is scanned, the location counter is incremented by the length of each instruction. Define data directives (DBs or DWs) increment the location counter by the number of bytes defined. Reserve memory directives (DSs) increment the location counter by the number of bytes reserved.

Each time a label is found at the beginning of a line, it is placed in the symbol table along with the current value of the location counter. Symbols that are defined using equate directives (EQUs) are placed in the symbol table along with the "equated" value. The symbol table is saved and then used during pass two.

Pass two

During pass two, the object and listing files are created. Mnemonics are converted to opcodes and placed in the output files. Operands are evaluated and placed after the instruction opcodes. Where symbols appear in the operand field, their values are retrieved from the symbol table (created during pass one) and used in calculating the correct data or addresses for the instructions.

Since two passes are performed, the source program may use "forward references", that is, use a symbol before it is defined. This would occur, for example, in branching ahead in a program.

The object file, if it is absolute, contains only the binary bytes (00H-0FH) of the machine language program. A relocatable object file will also contain a sysmbol table and other information required for linking and locating. The listing file contains ASCII text codes (02H-7EH) for both the source program and the hexadecimal bytes in the machine language program.

A good demonstration of the distinction between an object file and a listing file is to display each on the host computer's CRT display (using, for example, the TYPE command on MS-DOS systems). The listing file clearly displays, with each line of output containing an address, opcode, and perhaps data, followed by the program statement from the source file. The listing file displays properly because it contains only ASCII text codes. Displaying the object file is a problem, however. The output will appear as "garbage", since the object file contains binary codes of an 8051 machine language program, rather than ASCII text codes.

ASSEMBLY LANGUAGE PROGRAM FORMAT

Assembly language programs contain the following:

Machine instructions
Assembler directives
Assembler controls
Comments

Machine instructions are the familiar mnemonics of executable instructions (e.g., ANL). Assembler directives are instructions to the assembler program that define program structure, symbols, data, constants, and so on (e.g., ORG). Assembler controls set assembler modes and direct assembly flow (e.g., $TITLE). Comments enhance the readability of programs by explaining the purpose and operation of instruction sequences.

Those lines containing machine instructions or assembler directives must be written following specific rules understood by the assembler. Each line is divided into "fields" separated by space or tab characters. The general format for each line is as follows:

```
[label:]    mnemonic   [operand]    [, operand]     […]   [;commernt]
```

Only the mnemonic field is mandatory. Many assemblers require the label field, if present, to begin on the left in column 1, and subsequent fields to be separated by space or tab charecters. With ASM51, the label field needn't begin in column 1 and the mnemonic field needn't be on the same line as the label field. The operand field must, however, begin on the same line as the mnemonic field. The fields are described below.

Label Field

A label represents the address of the instruction (or data) that follows. When branching to this instruction, this label is usded in the operand field of the branch or jump instruction (e.g., SJMP SKIP).

Whereas the term "label" always represents an address, the term "symbol" is more general. Labels are one type of symbol and are identified by the requirement that they must terminate with a colon(:). Symbols are assigned values or attributes, using directives such as EQU, SEGMENT, BIT, DATA, etc. Symbols may be addresses, data constants, names of segments, or other constructs conceived by the programmer. Symbols do not terminate with a colon. In the example below, PAR is a symbol and START is a label (which is a type of symbol).

```
PAR      EQU    500                        ;"PAR" IS A SYMBOL WHICH
                                           ;REPRESENTS THE VALUE 500
START:   MOV    A,      #0FFH              ;"START" IS A LABEL WHICH
                                           ;REPRESENTS THE ADDRESS OF
                                           ;THE MOV INSTRUCTION
```

A symbol (or label) must begin with a letter, question mark, or underscore (_); must be followed by letters, digit, "?", or "_"; and can contain up to 31 characters. Symbols may use upper- or lowercase characters, but they are treated the same. Reserved words (mnemonics, operators, predefined symbols, and directives) may not be used.

Mnemonic Field

Intruction mnemonics or assembler directives go into mnemonic field, which follows the label field. Examples of instruction mnemonics are ADD, MOV, DIV, or INC. Examples of assembler directives are ORG, EQU, or DB.

Operand Field

The operand field follows the mnemonic field. This field contains the address or data used by the instruction. A label may be used to represent the address of the data, or a symbol may be used to represent a data constant. The possibilities for the operand field are largely dependent on the operation. Some operations have no operand (e.g., the RET instruction), while others allow for multiple operands separated by commas. Indeed, the possibilties for the operand field are numberous, and we shall elaborate on these at length. But first, the comment field.

Comment Field

Remarks to clarify the program go into comment field at the end of each line. Comments must begin with a semicolon (;). Each lines may be comment lines by beginning them with a semicolon. Subroutines and large sections of a program generally begin with a comment block—serveral lines of comments that explain the general properties of the section of software that follows.

Special Assembler Symbols

Special assembler symbols are used for the register-specific addressing modes. These include A, R0 through R7, DPTR, PC, C and AB. In addition, a dollar sign ($) can be used to refer to the current value of the location counter. Some examples follow.

```
        SETB    C
        INC     DPTR
        JNB     TI , $
```

The last instruction above makes effective use of ASM51's location counter to avoid using a label. It could also be written as

```
HERE:   JNB     TI , HERE
```

Indirect Address

For certain instructions, the operand field may specify a register that contains the address of the data. The commercial "at" sign (@) indicates address indirection and may only be used with R0, R1, the DPTR, or the PC, depending on the instruction. For example,

```
        ADD     A , @R0
        MOVC    A , @A+PC
```

The first instruction above retrieves a byte of data from internal RAM at the address specified in R0. The second instruction retrieves a byte of data from external code memory at the address formed by adding the contents of the accumulator to the program counter. Note that the value of the program counter, when the add takes place, is the address of the instruction following MOVC. For both instruction above, the value retrieved is placed into the accumulator.

Immediate Data

Instructions using immediate addressing provide data in the operand field that become part of the instruction. Immediate data are preceded with a pound sign (#). For example,

```
CONSTANT    EQU    100
            MOV    A ,     #0FEH
            ORL    40H ,   #CONSTANT
```

All immediate data operations (except MOV DPTR,#data) require eight bits of data. The immediate data are evaluated as a 16-bit constant, and then the low-byte is used. All bits in the high-byte must be the same (00H or FFH) or the error message "value will not fit in a byte" is generated. For example, the following instructions are syntactically correct:

```
MOV    A ,    #0FF00H
MOV    A ,    #00FFH
```

But the following two instructions generate error messages:

```
MOV    A ,    #0FE00H
MOV    A ,    #01FFH
```

If signed decimal notation is used, constants from -256 to +255 may also be used. For example, the following two instructions are equivalent (and syntactically correct):

```
MOV    A ,    #-256
MOV    A ,    #0FF00H
```

Both instructions above put 00H into accumulator A.

Data Address

Many instructions access memory locations using direct addressing and require an on-chip data memory address (00H to 7FH) or an SFR address (80H to 0FFH) in the operand field. Predefined symbols may be used for the SFR addresses. For example,

```
MOV    A ,    45H
MOV    A ,    SBUF            ;SAME AS MOV A, 99H
```

Bit Address

One of the most powerful features of the 8051 is the ability to access individual bits without the need for masking operations on bytes. Instructions accessing bit-addressable locations must provide a bit address in internal data memory (00h to 7FH) or a bit address in the SFRs (80H to 0FFH).

There are three ways to specify a bit address in an instruction: (a) explicitly by giving the address, (b) using the dot operator between the byte address and the bit position, and (c) using a predefined assembler symbol. Some examples follow.

```
SETB   0E7H                   ;EXPLICIT BIT ADDRESS
SETB   ACC.7                  ;DOT OPERATOR (SAME AS ABOVE)
JNB    TI ,    $              ;"TI" IS A PRE-DEFINED SYMBOL
JNB    99H,    $              ;(SAME AS ABOVE)
```

Code Address

A code address is used in the operand field for jump instructions, including relative jumps (SJMP and conditional jumps), absolute jumps and calls (ACALL, AJMP), and long jumps and calls (LJMP, LCALL).

The code address is usually given in the form of a label.

ASM51 will determine the correct code address and insert into the instruction the correct 8-bit signed offset, 11-bit page address, or 16-bit long address, as appropriate.

Generic Jumps and Calls

ASM51 allows programmers to use a generic JMP or CALL mnemonic. "JMP" can be used instead of SJMP, AJMP or LJMP; and "CALL" can be used instead of ACALL or LCALL. The assembler converts the generic mnemonic to a "real" instruction following a few simple rules. The generic mnemonic converts to the short form (for JMP only) if no forward references are used and the jump destination is within -128 locations, or to the absolute form if no forward references are used and the instruction following the JMP or CALL instruction is in the same 2K block as the destination instruction. If short or absolute forms cannot be used, the conversion is to the long form.

The conversion is not necessarily the best programming choice. For example, if branching ahead a few instrucions, the generic JMP will always convert to LJMP even though an SJMP is probably better. Consider the following assembled instructions sequence using three generic jumps.

LOC	OBJ	LINE	SOURCE			
1234		1		ORG	1234H	
1234	04	2	START:	INC	A	
1235	80FD	3		JMP	START	;ASSEMBLES AS SJMP
12FC		4		ORG	START + 200	
12FC	4134	5		JMP	START	;ASSEMBLES AS AJMP
12FE	021301	6		JMP	FINISH	;ASSEMBLES AS LJMP
1301	04	7	FINISH:	INC	A	
		8		END		

The first jump (line 3) assembles as SJMP because the destination is before the jump (i.e., no forward reference) and the offset is less than -128. The ORG directive in line 4 creates a gap of 200 locations between the label START and the second jump, so the conversion on line 5 is to AJMP because the offset is too great for SJMP. Note also that the address following the second jump (12FEH) and the address of START (1234H) are within the same 2K page, which, for this instruction sequence, is bounded by 1000H and 17FFH. This criterion must be met for absolute addressing. The third jump assembles as LJMP because the destination (FINISH) is not yet defined when the jump is assembled (i.e., a forward reference is used). The reader can verify that the conversion is as stated by examining the object field for each jump instruction.

ASSEMBLE-TIME EXPRESSION EVALUATION

Values and constants in the operand field may be expressed three ways: (a) explicitly (e.g.,0EFH), (b) with a predefined symbol (e.g., ACC), or (c) with an expression (e.g.,2 + 3). The use of expressions provides a powerful technique for making assembly language programs more readable and more flexible. When an expression is used, the assembler calculates a value and inserts it into the instruction.

All expression calculations are performed using 16-bit arithmetic; however, either 8 or 16 bits are inserted into the instruction as needed. For example, the following two instructions are the same:

```
MOV     DPTR,   #04FFH + 3
MOV     DPTR,   #0502H              ;ENTIRE 16-BIT RESULT USED
```

If the same expression is used in a "MOV A,#data" instruction, however, the error message "value will not fit in a byte" is generated by ASM51. An overview of the rules for evaluateing expressions follows.

Number Bases

The base for numeric constants is indicated in the usual way for Intel microprocessors. Constants must be followed with "B" for binary, "O" or "Q" for octal, "D" or nothing for decimal, or "H" for hexadecimal. For example, the following instructions are the same:

```
MOV     A , #15H
MOV     A , #1111B
MOV     A , #0FH
MOV     A , #17Q
MOV     A , #15D
```

Note that a digit must be the first character for hexadecimal constants in order to differentiate them from labels (i.e., "0A5H" not "A5H").

Charater Strings

Strings using one or two characters may be used as operands in expressions. The ASCII codes are converted to the binary equivalent by the assembler. Character constants are enclosed in single quotes ('). Some examples follow.

```
CJNE    A ,  # 'Q', AGAIN
SUBB    A ,  # '0'              ;CONVERT ASCII DIGIT TO BINARY DIGIT
MOV     DPTR, # 'AB'
MOV     DPTR, #4142H            ;SAME AS ABOVE
```

Arithmetic Operators

The arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
MOD	modulo (remainder after division)

For example, the following two instructions are same:

```
MOV     A,    10 +10H
MOV     A,    #1AH
```

The following two instructions are also the same:

```
MOV     A,    #25 MOD 7
MOV     A,    #4
```

Since the MOD operator could be confused with a symbol, it must be seperated from its operands by at least one space or tab character, or the operands must be enclosed in parentheses. The same applies for the other operators composed of letters.

Logical Operators

The logical operators are

OR	logical OR
AND	logical AND
XOR	logical Exclusive OR
NOT	logical NOT (complement)

The operation is applied on the corresponding bits in each operand. The operator must be separated from the operands by space or tab characters. For example, the following two instructions are the same:

```
        MOV     A, # '9'   AND   0FH
        MOV     A, #9
```

The NOT operator only takes one operand. The following three MOV instructions are the same:

```
THREE           EQU     3
MINUS_THREE     EQU     -3
                MOV     A,      # (NOT THREE) + 1
                MOV     A,      #MINUS_THREE
                MOV     A,      #11111101B
```

Special Operators

The sepcial operators are

```
        SHR     shift right
        SHL     shift left
        HIGH    high-byte
        LOW     low-byte
        ()      evaluate first
```

For example, the following two instructions are the same:

```
        MOV     A, #8  SHL  1
        MOV     A, #10H
```

The following two instructions are also the same:

```
        MOV     A, #HIGH  1234H
        MOV     A, #12H
```

Relational Operators

When a relational operator is used between two operands, the result is alwalys false (0000H) or true (FFFFH). The operators are

```
        EQ      =       equals
        NE      < >     not equals
        LT      <       less than
        LE      <=      less than or equal to
        GT      >       greater than
        GE      >=      greater than or equal to
```

Note that for each operator, two forms are acceptable (e.g., "EQ" or "="). In the following examples, all relational tests are "true":

```
        MOV     A, #5 = 5
        MOV     A,#5 NE 4
        MOV     A,# 'X'  LT  'Z'
        MOV     A,# 'X'  >=  'X'
        MOV     A,#$ > 0
        MOV     A,#100 GE 50
```

So, the assembled instructions are equal to

```
MOV     A, #0FFH
```

Even though expressions evaluate to 16-bit results (i.e., 0FFFFH), in the examples above only the low-order eight bits are used, since the instruction is a move byte operation. The result is not considered too big in this case, because as signed numbers the 16-bit value FFFFH and the 8-bit value FFH are the same (-1).

Expression Examples

The following are examples of expressions and the values that result:

Expression	**Result**
'B' - 'A'	0001H
8/3	0002H
155 MOD 2	0001H
4 * 4	0010H
8 AND 7	0000H
NOT 1	FFFEH
'A' SHL 8	4100H
LOW 65535	00FFH
(8 + 1) * 2	0012H
5 EQ 4	0000H
'A' LT 'B'	FFFFH
3 <= 3	FFFFHss

A practical example that illustrates a common operation for timer initialization follows: Put -500 into Timer 1 registers TH1 and TL1. In using the HIGH and LOW operators, a good approach is

```
VALUE       EQU     -500
            MOV     TH1, #HIGH  VALUE
            MOV     TL1, #LOW VALUE
```

The assembler converts -500 to the corresponding 16-bit value (FE0CH); then the HIGH and LOW operators extract the high (FEH) and low (0CH) bytes. as appropriate for each MOV instruction.

Operator Precedence

The precedence of expression operators from highest to lowest is

```
( )
HIGH LOW
*  /  MOD  SHL  SHR
+ -
EQ  NE  LT  LE  GT  GE  =  <>  <  <=  >  >=
NOT
AND
OR  XOR
```

When operators of the same precedence are used, they are evaluated left to right.
Examples:

Expression	**Value**
HIGH ('A' SHL 8)	0041H
HIGH 'A' SHL 8	0000H
NOT 'A' - 1	FFBFH
'A' OR 'A' SHL 8	4141H

ASSEMBLER DIRECTIVES

Assembler directives are instructions to the assembler program. They are not assembly language instructions executable by the target microprocessor. However, they are placed in the mnemonic field of the program. With the exception of DB and DW, they have no direct effect on the contents of memory.

ASM51 provides several catagories of directives:

Assembler state control (ORG, END, USING)
Symbol definition (SEGMENT, EQU, SET, DATA, IDATA, XDATA, BIT, CODE)
Storage initialization/reservation (DS, DBIT, DB, DW)
Program linkage (PUBLIC, EXTRN,NAME)
Segment selection (RSEG, CSEG, DSEG, ISEG, ESEG, XSEG)

Each assembler directive is presented below, ordered by catagory.

Assembler State Control

ORG (Set Origin) The format for the ORG (set origin) directive is

```
ORG     expression
```

The ORG directive alters the location counter to set a new program origin for statements that follow. A label is not permitted. Two examples follow.

```
ORG     100H                    ;SET LOCATION COUNTER TO 100H
ORG     ($ + 1000H)  AND  0F00H ;SET TO NEXT 4K BOUNDARY
```

The ORG directive can be used in any segment type. If the current segment is absolute, the value will be an absolute address in the current segment. If a relocatable segment is active, the value of the ORG expression is treated as an offset from the base address of the current instance of the segment.

End The format of the END directive is

```
END
```

END should be the last statement in the source file. No label is permitted and nothing beyond the END statement is processed by the assembler.

Using The format of the END directive is

```
USING   expression
```

This directive informs ASM51 of the currently active register bank. Subsequent uses of the predefined symbolic register addresses AR0 to AR7 will convert to the appropriate direct address for the active register bank. Consider the following sequence:

```
USING   3
PUSH    AR7
USING   1
PUSH    AR7
```

The first push above assembles to PUSH 1FH (R7 in bank 3), whereas the second push assembles to PUSH 0FH (R7 in bank 1).

Note that USING does not actually switch register banks; it only informs ASM51 of the active bank. Executing 8051 instructions is the only way to switch register banks. This is illustrated by modifying the example above as follows:

```
MOV     PSW, #00011000B     ;SELECT REGISTER BANK 3
USING   3
PUSH    AR7                 ;ASSEMBLE TO PUSH 1FH
MOV     PSW, #00001000B     ;SELECT REGISTER BANK 1
USING   1
PUSH    AR7                 ;ASSEMBLE TO PUSH 0FH
```

Symbol Definition

The symbol definition directives create symbols that represent segment, registers, numbers, and addresses. None of these directives may be preceded by a label. Symbols defined by these directives may not have been previously defined and may not be redefined by any means. The SET directive is the only exception. Symbol definiton directives are described below.

Segment The format for the SEGMENT directive is shown below.

```
symbol          SEGMENT         segment_type
```

The symbol is the name of a relocatable segment. In the use of segments, ASM51 is more complex than conventional assemblers, which generally support only "code" and "data" segment types. However, ASM51 defines additional segment types to accommodate the diverse memory spaces in the 8051. The following are the defined 8051 segment types (memory spaces):

CODE (the code segment)
XDATA (the external data space)
DATA (the internal data space accessible by direct addressing, 00H–07H)
IDATA (the entire internal data space accessible by indirect addressing, 00H–07H)
BIT (the bit space; overlapping byte locations 20H–2FH of the internal data space)

For example, the statement

```
EPROM           SEGMENT         CODE
```

declares the symbol EPROM to be a SEGMENT of type CODE. Note that this statement simply declares what EPROM is. To actually begin using this segment, the RSEG directive is used (see below).

EQU (Equate) The format for the EQU directive is

```
Symbol          EQU     expression
```

The EQU directive assigns a numeric value to a specified symbol name. The symbol must be a valid symbol name, and the expression must conform to the rules described earlier.

The following are examples of the EQU directive:

```
N27             EQU     27              ;SET N27 TO THE VALUE 27
HERE            EQU     $               ;SET "HERE" TO THE VALUE OF
                                        ;THE LOCATION COUNTER
CR              EQU     0DH             ;SET CR (CARRIAGE RETURN) TO 0DH
MESSAGE:        DB 'This is a message'
LENGTH          EQU     $ - MESSAGE     ;"LENGTH" EQUALS LENGTH OF "MESSAGE"
```

Other Symbol Definition Directives The SET directive is similar to the EQU directive except the symbol may be redefined later, using another SET directive.

The DATA, IDATA, XDATA, BIT, and CODE directives assign addresses of the corresponding segment type to a symbol. These directives are not essential. A similar effect can be achieved using the EQU directive; if used, however, they evoke powerful type-checking by ASM51. Consider the following two directives and four instructions:

```
FLAG1       EQU     05H
FLAG2       BIT     05H
            SETB    FLAG1
            SETB    FLAG2
            MOV     FLAG1, #0
            MOV     FLAG2, #0
```

The use of FLAG2 in the last instruction in this sequence will generate a "data segment address expected" error message from ASM51. Since FLAG2 is defined as a bit address (using the BIT directive), it can be used in a set bit instruction, but it cannot be used in a move byte instruction. Hence, the error. Even though FLAG1 represents the same value (05H), it was defined using EQU and does not have an associated address space. This is not an advantage of EQU, but rather, a disadvantage. By properly defining address symbols for use in a specific memory space (using the directives BIT, DATA, XDATA,ect.), the programmer takes advantage of ASM51's powerful type-checking and avoids bugs from the misuse of symbols.

Storage Initialization/Reservation

The storage initialization and reservation directives initialize and reserve space in either word, byte, or bit units. The space reserved starts at the location indicated by the current value of the location counter in the currently active segment. These directives may be preceded by a label. The storage initialization/reservation directives are described below.

DS (Define Storage) The format for the DS (define storage) directive is

```
[label:]    DS      expression
```

The DS directive reserves space in byte units. It can be used in any segment type except BIT. The expression must be a valid assemble-time expression with no forward references and no relocatable or external references. When a DS statement is encountered in a program, the location counter of the current segment is incremented by the value of the expression. The sum of the location counter and the specified expression should not exceed the limitations of the current address space.

The following statement create a 40-byte buffer in the internal data segment:

```
            DSEG    AT      30H     ;PUT IN DATA SEGMENT (ABSOLUTE, INTERNAL)
LENGTH      EQU     40
BUFFER:     DS      LENGRH          ;40 BYTES RESERVED
```

The label BUFFER represents the address of the first location of reserved memory. For this example, the buffer begins at address 30H because "AT 30H" is specified with DSEG. The buffer could be cleared using the following instruction sequence:

```
        MOV     R7,     #LENGTH
        MOV     R0,     #BUFFER
LOOP:   MOV     @R0,    #0
        DJNZ    R7,     LOOP
        (continue)
```

To create a 1000-byte buffer in external RAM starting at 4000H, the following directives could be used:

```
XSTART      EQU     4000H
XLENGTH     EQU     1000
            XSEG        AT    XSTART
XBUFFER:    DS   XLENGTH
```

This buffer could be cleared with the following instruction sequence:

```
        MOV     DPTR,  #XBUFFER
LOOP:   CLR     A
        MOVX    @DPTR,  A
        INC     DPTR
        MOV     A,      DPL
        CJNE    A,      #LOW (XBUFFER + XLENGTH + 1), LOOP
        MOV     A,      DPH
        CJNE    A,      #HIGH (XBUFFER + XLENGTH + 1), LOOP
        (continue)
```

This is an excellent example of a powerful use of ASM51's operators and assemble-time expressions. Since an instruction does not exist to compare the data pointer with an immediate value, the operation must be fabricated from available instructions. Two compares are required, one each for the high- and low-bytes of the DPTR. Furthermore, the compare-and-jump-if-not-equal instruction works only with the accumulator or a register, so the data pointer bytes must be moved into the accumulator before the CJNE instruction. The loop terminates only when the data pointer has reached XBUFFER + LENGTH + 1. (The "+1" is needed because the data pointer is incremented after the last MOVX instruction.)

DBIT The format for the DBIT (define bit) directive is,

```
[label:]        DBIT    expression
```

The DBIT directive reserves space in bit units. It can be used only in a BIT segment. The expression must be a valid assemble-time expression with no forward references. When the DBIT statement is encountered in a program, the location counter of the current (BIT) segment is incremented by the value of the expression. Note that in a BIT segment, the basic unit of the location counter is bits rather than bytes. The following directives creat three flags in a absolute bit segment:

```
            BSEG            ;BIT SEGMENT (ABSOLUTE)
KEFLAG:     DBIT    1       ;KEYBOARD STATUS
PRFLAG:     DBIT    1       ;PRINTER STATUS
DKFLAG:     DBIT    1       ;DISK STATUS
```

Since an address is not specified with BSEG in the example above, the address of the flags defined by DBIT could be determined (if one wishes to to so) by examining the symbol table in the .LST or .M51 files. If the definitions above were the first use of BSEG, then KBFLAG would be at bit address 00H (bit 0 of byte address 20H). If other bits were defined previously using BSEG, then the definitions above would follow the last bit defined.

DB (Define Byte) The format for the DB (define byte) directive is,

```
[label:]        DB      expression  [, expression]  […]
```

The DB directive initializes code memory with byte values. Since it is used to actually place data constants in code memory, a CODE segment must be active. The expression list is a series of one or more byte values (each of which may be an expression) separated by commas.

The DB directive permits character strings (enclosed in single quotes) longer than two characters as long as they are not part of an expression. Each character in the string is converted to the corresponding ASCII code. If a label is used, it is assigned the address of th first byte. For example, the following statements

```
          CSEG   AT       0100H
SQUARES:  DB     0, 1, 4, 9, 16, 25       ;SQUARES OF NUMBERS 0-5
MESSAGE:  DB     'Login:', 0              ;NULL-TERMINATED CHARACTER STRING
```

When assembled, result in the following hexadecimal memory assignments for external code memory:

Address	**Contents**
0100	00
0101	01
0102	04
0103	09
0104	10
0105	19
0106	4C
0107	6F
0108	67
0109	69
010A	6E
010B	3A
010C	00

DW (Define Word) The format for the DW (define word) directive is

```
[label:]          DW       expression          [, expression]  […]
```

The DW directive is the same as the DB directive except two memory locations (16 bits) are assigned for each data item. For example, the statements

```
CSEG    AT       200H
DW      $, 'A', 1234H, 2, 'BC'
```

result in the following hexadecimal memory assignments:

Address	**Contents**
0200	02
0201	00
0202	00
0203	41
0204	12
0205	34
0206	00
0207	02
0208	42
0209	43

Program Linkage

Program linkage directives allow the separately assembled modules (files) to communicate by permitting inter-module references and the naming of modules. In the following discussion, a "module" can be considered a "file." (In fact, a module may encompass more than one file.)

Public The format for the PUBLIC (public symbol) directive is

PUBLIC symbol [, symbol] […]

The PUBLIC directive allows the list of specified symbols to known and used outside the currently assembled module. A symbol declared PUBLIC must be defined in the current module. Declaring it PUBLIC allows it to be referenced in another module. For example,

PUBLIC INCHAR, OUTCHR, INLINE, OUTSTR

Extrn The format for the EXTRN (external symbol) directive is

EXTRN segment_type (symbol [, symbol] […], …)

The EXTRN directive lists symbols to be referenced in the current module that are defined in other modules. The list of external symbols must have a segment type associated with each symbol in the list. (The segment types are CODE, XDATA, DATA, IDATA, BIT, and NUMBER. NUMBER is a type-less symbol defined by EQU.) The segment type indicates the way a symbol may be used. The information is important at link-time to ensure symbols are used properly in different modules.

The PUBLIC and EXTRN directives work together. Consider the two files, MAIN.SRC and MESSAGES.SRC. The subroutines HELLO and GOOD_BYE are defined in the module MESSAGES but are made available to other modules using the PUBLIC directive. The subroutines are called in the module MAIN even though they are not defined there. The EXTRN directive declares that these symbols are defined in another module.

MAIN.SRC:

```
                EXTRN       CODE (HELLO, GOOD_BYE)
                …
                CALL        HELLO
                …
                CALL        GOOD_BYE
                …
                END
```

MESSAGES.SRC:

```
                PUBLIC      HELLO, GOOD_BYE
                …
HELLO:          (begin subroutine)
                …
                RET
GOOD_BYE:       (begin subroutine)
                …
                RET
                …
                END
```

Neither MAIN.SRC nor MESSAGES.SRC is a complete program; they must be assembled separately and linked together to form an executable program. During linking, the external references are resolved with correct addresses inserted as the destination for the CALL instructions.

Name The format for the NAME directive is

NAME module_name

All the usual rules for symbol names apply to module names. If a name is not provided, the module takes on the file name (without a drive or subdirectory specifier and without an extension). In the absence of any use of the NAME directive, a program will contain one module for each file. The concept of "modules," therefore, is somewhat cumbersome, at least for relatively small programming problems. Even programs of moderate size (encompassing, for example, several files complete with relocatable segments) needn't use the NAME directive and needn't pay any special attention to the concept of "modules." For this reason, it was mentioned in the definition that a module may be considered a "file," to simplify learning ASM51. However, for very large programs (several thousand lines of code, or more), it makes sense to partition the problem into modules, where, for example, each module may encompass several files containing routines having a common purpose.

Segment Selection Directives

When the assembler encounters a segment selection directive, it diverts the following code or data into the selected segment until another segment is selected by a segment selection directive. The directive may select may select a previously defined relocatable segment or optionally create and select absolute segments.

RSEG (Relocatable Segment) The format for the RSEG (relocatable segment) directive is

```
RSEG          segment_name
```

Where "segment_name" is the name of a relocatable segment previously defined with the SEGMENT directive. RSEG is a "segment selection" directive that diverts subsequent code or data into the named segment until another segment selection directive is encountered.

Selecting Absolute Segments RSEG selects a relocatable segment. An "absolute" segment, on the other hand, is selected using one of the directives:

```
CSEG    (AT  address)
DSEG    (AT  address)
ISEG    (AT  address)
BSEG    (AT  address)
XSEG    (AT  address)
```

These directives select an absolute segment within the code, internal data, indirect internal data, bit, or external data address spaces, respectively. If an absolute address is provided (by indicating "AT address"), the assembler terminates the last absolute address segment, if any, of the specified segment type and creates a new absolute segment starting at that address. If an absolute address is not specified, the last absolute segment of the specified type is continuted. If no absolute segment of this type was previously selected and the absolute address is omitted, a new segment is created starting at location 0. Forward references are not allowed and start addresses must be absolute.

Each segment has its own location counter, which is always set to 0 initially. The default segment is an absolute code segment; therefore, the initial state of the assembler is location 0000H in the absolute code segment. When another segment is chosen for the first time, the location counter of the former segment retains the last active value. When that former segment is reselected, the location counter picks up at the last active value. The ORG directive may be used to change the location counter within the currently selected segment.

ASSEMBLER CONTROLS

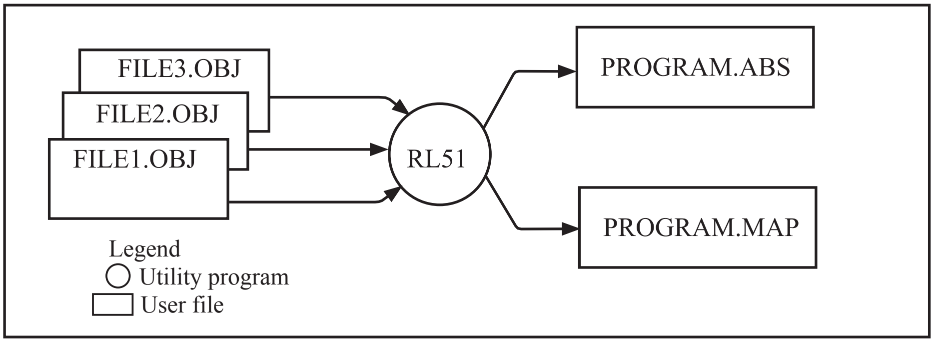
Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any affect on the program itself. They can be entered on the invocation line when a program is assembled, or they can be placed in the source file. Assembler controls appearing in the source file must be preceded with a dollor sign and must begin in column 1.

There are two categories of assembler controls: primary and general. Primary controls can be placed in the invocation line or at the beginnig of the source program. Only other primary controls may precede a primary control. General controls may be placed anywhere in the source program.

LINKER OPERATION

In developing large application programs, it is common to divide tasks into subprograms or modules containing sections of code (usually subroutines) that can be written separately from the overall program. The term "modular programming" refers to this programming strategy. Generally, modules are relocatable, meaning they are not intended for a specific address in the code or data space. A linking and locating program is needed to combine the modules into one absolute object module that can be executed.

Intel's RL51 is a typical linker/locator. It processes a series of relocatable object modules as input and creates an executable machine language program (PROGRAM, perhaps) and a listing file containing a memory map and symbol table (PROGRAM.M51). This is illustrated in following figure.



Linker operation

As relocatable modules are combined, all values for external symbols are resolved with values inserted into the output file. The linker is invoked from the system prompt by

RL51 input_list [T0 output_file] [location_controls]

The input_list is a list of relocatable object modules (files) separated by commas. The output_list is the name of the output absolute object module. If none is supplied, it defaults to the name of the first input file without any suffix. The location_controls set start addresses for the named segments.

For example, suppose three modules or files (MAIN.OBJ, MESSAGES.OBJ, and SUBROUTINES.OBJ) are to be combined into an executable program (EXAMPLE), and that these modules each contain two relocatable segments, one called EPROM of type CODE, and the other called ONCHIP of type DATA. Suppose further that the code segment is to be executable at address 4000H and the data segment is to reside starting at address 30H (in internal RAM). The following linker invocation could be used:

RS51 MAIN.OBJ, MESSAGES.OBJ, SUBROUTINES.OBJ TO EXAMPLE & CODE
(EPROM (4000H) DATA (ONCHIP (30H))

Note that the ampersand character "&" is used as the line continuaton character.

If the program begins at the label START, and this is the first instruction in the MAIN module, then execution begins at address 4000H. If the MAIN module was not linked first, or if the label START is not at the beginning of MAIN, then the program's entry point can be determined by examining the symbol table in the listing file EXAMPLE.M51 created by RL51. By default, EXAMPLE.M51 will contain only the link map. If a symbol table is desired, then each source program must have used the SDEBUG control. The following table shows the assembler controls supported by ASM51.

Assembler controls supported by ASM51				
NAME	PRIMARY/ GENERAL	DEFAULT	ABBREV.	MEANING
DATE (date)	P	DATE( )	DA	Place string in header (9 char. max.)
DEBUG	P	NODEBUG	DB	Outputs debug symbol information to object file
EJECT	G	not applicable	EJ	Continue listing on next page
ERRORPRINT (file)	P	NOERRORPRINT	EP	Designates a file to receive error messages in addition to the listing file (defauts to console)
NOERRORPRINT	P	NOERRORPRINT	NOEP	Designates that error messages will be printed in listing file only
GEN	G	GENONLY	GO	List only the fully expanded source as if all lines generated by a macro call were already in the source file
GENONLY	G	GENONLY	NOGE	List only the original source text in the listing file
INCLUED(file)	G	not applicable	IC	Designates a file to be included as part of the program
LIST	G	LIST	LI	Print subsequent lines of source code in listing file
NOLIST	G	LIST	NOLI	Do not print subsequent lines of source code in lisitng file
MACRO (men_precent)	P	MACRO(50)	MR	Evaluate and expand all macro calls. Allocate percentage of free memory for macro processing
NOMACRO	P	MACRO(50)	NOMR	Do not evalutate macro calls
MOD51	P	MOD51	MO	Recognize the 8051-specific predefined special function registers
NOMOD51	P	MOD51	NOMO	Do not recognize the 8051-specific predefined special function registers
OBJECT(file)	P	OBJECT(source.OBJ)	OJ	Designates file to receive object code
NOOBJECT	P	OBJECT(source.OBJ)	NOOJ	Designates that no object file will be created
PAGING	P	PAGING	PI	Designates that listing file be broken into pages and each will have a header
NOPAGING	P	PAGING	NOPI	Designates that listing file will contain no page breaks
PAGELENGTH (N)	P	PAGELENGT(60)	PL	Sets maximun number of lines in each page of listing file (range=10 to 65536)
PAGE WIDTH (N)	P	PAGEWIDTH(120)	PW	Set maximum number of characters in each line of listing file (range = 72 to 132)
PRINT(file)	P	PRINT(source.LST)	PR	Designates file to receive source listing
NOPRINT	P	PRINT(source.LST)	NOPR	Designates that no listing file will be created
SAVE	G	not applicable	SA	Stores current control settings from SAVE stack
RESTORE	G	not applicable	RS	Restores control settings from SAVE stack
REGISTERBANK (rb,...)	P	REGISTERBANK(0)	RB	Indicates one or more banks used in program module
NOREGISTER-BANK	P	REGISTERBANK(0)	NORB	Indicates that no register banks are used
SYMBOLS	P	SYMBOLS	SB	Creates a formatted table of all symbols used in program
NOSYMBOLS	P	SYMBOLS	NOSB	Designates that no symbol table is created
TITLE(string)	G	TITLE( )	TT	Places a string in all subsequent page headers (max.60 characters)
WORKFILES (path)	P	same as source	WF	Designates alternate path for temporay workfiles
XREF	P	NOXREF	XR	Creates a cross reference listing of all symbols used in program
NOXREF	P	NOXREF	NOXR	Designates that no cross reference list is created

MACROS

The macro processing facility (MPL) of ASM51 is a "string replacement" facility. Macros allow frequently used sections of code be defined once using a simple mnemonic and used anywhere in the program by inserting the mnemonic. Programming using macros is a powerful extension of the techniques described thus far. Macros can be defined anywhere in a source program and subsequently used like any other instruction. The syntax for macro definition is

```
%*DEFINE     (call_pattern)     (macro_body)
```

Once defined, the call pattern is like a mnemonic; it may be used like any assembly language instruction by placing it in the mnemonic field of a program. Macros are made distinct from "real" instructions by preceding them with a percent sign, "%". When the source program is assembled, everything within the macro-body, on a character-by-character basis, is substituted for the call-pattern. The mystique of macros is largely unfounded. They provide a simple means for replacing cumbersome instruction patterns with primitive, easy-to-remember mnemonics. The substitution, we reiterate, is on a character-by-character basis—nothing more, nothing less.

For example, if the following macro definition appears at the beginning of a source file,

```
%*DEFINE     (PUSH_DPTR)
                  (PUSH    DPH
                   PUSH    DPL
                   )
```

then the statement

```
%PUSH_DPTR
```

will appear in the .LST file as

```
PUSH    DPH
PUSH    DPL
```

The example above is a typical macro. Since the 8051 stack instructions operate only on direct addresses, pushing the data pointer requires two PUSH instructions. A similar macro can be created to POP the data pointer.

There are several distinct advantages in using macros:

A source program using macros is more readable, since the macro mnemonic is generally more indicative of the intended operation than the equivalent assembler instructions.
The source program is shorter and requires less typing.
Using macros reduces bugs
Using macros frees the programmer from dealing with low-level details.

The last two points above are related. Once a macro is written and debugged, it is used freely without the worry of bugs. In the PUSH_DPTR example above, if PUSH and POP instructions are used rather than push and pop macros, the programmer may inadvertently reverse the order of the pushes or pops. (Was it the high-byte or low-byte that was pushed first?) This would create a bug. Using macros, however, the details are worked out once—when the macro is written—and the macro is used freely thereafter, without the worry of bugs.

Since the replacement is on a character-by-character basis, the macro definition should be carefully constructed with carriage returns, tabs, ect., to ensure proper alignment of the macro statements with the rest of the assembly language program. Some trial and error is required.

There are advanced features of ASM51's macro-processing facility that allow for parameter passing, local labels, repeat operations, assembly flow control, and so on. These are discussed below.

Parameter Passing

A macro with parameters passed from the main program has the following modified format:

```
%*DEFINE        (macro_name  (parameter_list))   (macro_body)
```

For example, if the following macro is defined,

```
%*DEFINE        (CMPA#  (VALUE))
          (CJNE    A, #%VALUE, $ + 3
           )
```

then the macro call

```
%CMPA#  (20H)
```

will expand to the following instruction in the .LST file:

```
CJNE    A, #20H, $ + 3
```

Although the 8051 does not have a "compare accumulator" instruction, one is easily created using the CJNE instruction with "$+3" (the next instruction) as the destination for the conditional jump. The CMPA# mnemonic may be easier to remember for many programmers. Besides, use of the macro unburdens the programmer from remembering notational details, such as "$+3."

Let's develop another example. It would be nice if the 8051 had instructions such as

```
JUMP    IF  ACCUMULATOR  GREATER  THAN  X
JUMP    IF  ACCUMULATOR  GREATER  THAN  OR  EQUAL  TO  X
JUMP    IF  ACCUMULATOR  LESS  THAN  X
JUMP    IF  ACCUMULATOR  LESS  THAN  OR  EQUAL  TO  X
```

but it does not. These operations can be created using CJNE followed by JC or JNC, but the details are tricky. Suppose, for example, it is desired to jump to the label GREATER_THAN if the accumulator contains an ASCII code greater than "Z" (5AH). The following instruction sequence would work:

```
CJNE    A, #5BH, $÷3
JNC     GREATER_THAN
```

The CJNE instruction subtracts 5BH (i.e., "Z" + 1) from the content of A and sets or clears the carry flag accordingly. CJNE leaves C=1 for accumulator values 00H up to and including 5AH. (Note: 5AH-5BH<0, therefore C=1; but 5BH-5BH=0, therefore C=0.) Jumping to GREATER_THAN on the condition "not carry" correctly jumps for accumulator values 5BH, 5CH, 5DH, and so on, up to FFH. Once details such as these are worked out, they can be simplified by inventing an appropriate mnemonic, defining a macro, and using the macro instead of the corresponding instruction sequence. Here's the definition for a "jump if greater than" macro:

```
%*DEFINE        (JGT (VALUE,  LABEL))
                    (CJNE    A, #%VALUE+1, $+3      ;JGT
                     JNC      %LABEL
                    )
```

To test if the accumulator contains an ASCII code greater than "Z," as just discussed,the macro would be called as

```
%JGT    ('Z', GREATER_THAN)
```

ASM51 would expand this into

```
CJNE    A, #5BH, $+3        ;JGT
JNC     GREATER_THAN
```

The JGT macro is an excellent example of a relevant and powerful use of macros. By using macros, the programmer benefits by using a meaningful mnemonic and avoiding messy and potentially bug-ridden details.

Local Labels

Local labels may be used within a macro using the following format:

```
%*DEFINE        (macro_name  [(parameter_list)])
                          [LOCAL   list_of_local_labels]    (macro_body)
```

For example, the following macro definition

```
%*DEFINE    (DEC_DPTR)    LOCAL  SKIP
                (DEC      DPL                          ;DECREMENT DATA POINTER
                 MOV      A,        DPL
                 CJNE     A,        #0FFH,    %SKIP
                 DEC      DPL
%SKIP:           )
```

would be called as

```
%DEC_DPTR
```

and would be expanded by ASM51 into

```
          DEC      DPL                          ;DECREMENT  DATA  POINTER
          MOV      A,        DPL
          CJNE     A,        #0FFH,    SKIP00
          DEC      DPH
SKIP00:
```

Note that a local label generally will not conflict with the same label used elsewhere in the source program, since ASM51 appends a numeric code to the local label when the macro is expanded. Furthermore, the next use of the same local label receives the next numeric code, and so on.

The macro above has a potential "side effect." The accumulator is used as a temporary holding place for DPL. If the macro is used within a section of code that uses A for another purpose, the value in A would be lost. This side effect probably represents a bug in the program. The macro definition could guard against this by saving A on the stack. Here's an alternate definition for the DEC_DPTR macro:

```
%*DEFINE        (DEC_DPTR)      LOCAL SKIP
                  (PUSHACC
                 DEC      DPL                          ;DECREMENT DATA POINTER
                 MOV      A,        DPL
                 CJNE     A,        #0FFH, %SKIP
                 DEC      DPH
%SKIP:           POP      ACC
                 )
```

Repeat Operations

This is one of several built-in (predefined) macros. The format is

```
%REPEAT         (expression)        (text)
```

For example, to fill a block of memory with 100 NOP instructions,

```
%REPEAT    (100)
 (NOP
 )
```

Control Flow Operations

The conditional assembly of section of code is provided by ASM51's control flow macro definition. The format is

```
%IF  (expression)   THEN  (balanced_text)
[ELSE     (balanced_text)]  FI
```

For example,

```
INTRENAL        EQU     1           ;1 = 8051 SERIAL I/O DRIVERS
                                    ;0 = 8251 SERIAL I/O DRIVERS
                .
                .
                %IF (INTERNAL) THEN
(INCHAR:        .                   ;8051 DRIVERS
                .
OUTCHR:         .
                .
                ) ELSE
(INCHAR:        .                   ;8251 DRIVERS
                .
OUTCHR:         .
                .
                )
```

In this example, the symbol INTERNAL is given the value 1 to select I/O subroutines for the 8051's serial port, or the value 0 to select I/O subroutines for an external UART, in this case the 8251. The IF macro causes ASM51 to assemble one set of drivers and skip over the other. Elsewhere in the program, the INCHAR and OUTCHR subroutines are used without consideration for the particular hardware configuration. As long as the program as assembled with the correct value for INTERNAL, the correct subroutine is executed.

附录B：C语言编程

ADVANTAGES AND DISADVANTAGES OF 8051 C

The advantages of programming the 8051 in C as compared to assembly are:

- Offers all the benefits of high-level, structured programming languages such as C, including the ease of writing subroutines
- Often relieves the programmer of the hardware details that the complier handles on behalf of the programmer
- Easier to write, especially for large and complex programs
- Produces more readable program source codes

Nevertheless, 8051 C, being very similar to the conventional C language, also suffers from the following disadvantages:

- Processes the disadvantages of high-level, structured programming languages.
- Generally generates larger machine codes
- Programmer has less control and less ability to directly interact with hardware

To compare between 8051 C and assembly language, consider the solutions to the Example—Write a program using Timer 0 to create a 1KHz square wave on P1.0.

A solution written below in 8051 C language:

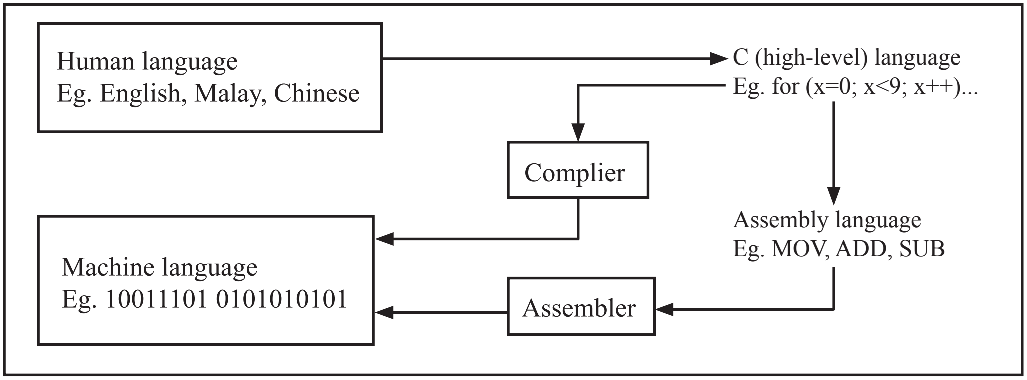
```
sbit portbit = P1^0;            /*Use variable portbit to refer to P1.0*/
main ( )
{
TMOD = 1;
while (1)
      {
         TH0 = 0xFE;
         TL0 = 0xC;
         TR0 = 1;
         while (TF0 !=1);
         TR0 = 0;
         TF0 = 0;
         portbit = !(P1.^0);
      }
}
```

A solution written below in assembly language:

```
              ORG     8100H
              MOV     TMOD,   #01H          ;16-bit timer mode
      LOOP:   MOV     TH0,    #0FEH         ;-500 (high byte)
              MOV     TL0,    #0CH          ;-500 (low byte)
              SETB    TR0                   ;start timer
      WAIT:   JNB     TF0,    WAIT          ;wait for overflow
              CLR     TR0                   ;stop timer
              CLR     TF0                   ;clear timer overflow flag
              CPL     P1.0                  ;toggle port bit
              SJMP    LOOP                  ;repeat
              END
```

Notice that both the assembly and C language solutions for the above example require almost the same number of lines. However, the difference lies in the readability of these programs. The C version seems more human than assembly, and is hence more readable. This often helps facilitate the human programmer's efforts to write even very complex programs. The assembly language version is more closely related to the machine code, and though less readable, often results in more compact machine code. As with this example, the resultant machine code from the assembly version takes 83 bytes while that of the C version requires 149 bytes, an increase of 79.5%!

The human programmer's choice of either high-level C language or assembly language for talking to the 8051, whose language is machine language, presents an interesting picture, as shown in following figure.



Conversion between human, high-level, assembly, and machine language

8051 C COMPILERS

We saw in the above figure that a complier is needed to convert programs written in 8051 C language into machine language, just as an assembler is needed in the case of programs written in assembly language. A complier basically acts just like an assembler, except that it is more complex since the difference between C and machine language is far greater than that between assembly and machine language. Hence the complier faces a greater task to bridge that difference.

Currently, there exist various 8051 C complier, which offer almost similar functions. All our examples and programs have been compiled and tested with Keil's μ Vision 2 IDE by Keil Software, an integrated 8051 program development envrionment that includes its C51 cross compiler for C. A cross compiler is a compiler that normally runs on a platform such as IBM compatible PCs but is meant to compile programs into codes to be run on other platforms such as the 8051.

DATA TYPES

8051 C is very much like the conventional C language, except that several extensions and adaptations have been made to make it suitable for the 8051 programming environment. The first concern for the 8051 C programmer is the data types. Recall that a data type is something we use to store data. Readers will be familiar with the basic C data types such as int, char, and float, which are used to create variables to store integers, characters, or floating-points. In 8051 C, all the basic C data types are supported, plus a few additional data types meant to be used specifically with the 8051.

The following table gives a list of the common data types used in 8051 C. The ones in bold are the specific 8051 extensions. The data type **bit** can be used to declare variables that reside in the 8051's bit-addressable locations (namely byte locations 20H to 2FH or bit locations 00H to 7FH). Obviously, these bit variables can only store bit values of either 0 or 1. As an example, the following C statement:

```
bit    flag = 0;
```

declares a bit variable called flag and initializes it to 0.

Data types used in 8051 C language

Data Type	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2,147,483,648 to +2,147,483,647
unsigned long	32	4	0 to 4,294,967,295
float	32	4	±1.175494E-38 to ±3.402823E+38
sbit	1		0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

The data type **sbit** is somewhat similar to the bit data type, except that it is normally used to declare 1-bit variables that reside in special function registes (SFRs). For example:

```
sbit     P = 0xD0;
```

declares the **sbit** variable P and specifies that it refers to bit address D0H, which is really the LSB of the PSW SFR. Notice the difference here in the usage of the assignment ("=") operator. In the context of **sbit** declarations, it indicatess what address the **sbit** variable resides in, while in **bit** declarations, it is used to specify the initial value of the **bit** variable.

Besides directly assigning a bit address to an **sbit** variable, we could also use a previously defined **sfr** variable as the base address and assign our **sbit** variable to refer to a certain bit within that **sfr**. For example:

```
sfr      PSW = 0xD0;
sbit     P = PSW^0;
```

This declares an **sfr** variable called PSW that refers to the byte address D0H and then uses it as the base address to refer to its LSB (bit 0). This is then assigned to an **sbit** variable, P. For this purpose, the carat symbol (^) is used to specify bit position 0 of the PSW.

A third alternative uses a constant byte address as the base address within which a certain bit is referred. As an illustration, the previous two statements can be replaced with the following:

```
sbit     P = 0xD0 ^ 0;
```

Meanwhile, the **sfr** data type is used to declare byte (8-bit) variables that are associated with SFRs. The statement:

```
sfr      IE = 0xA8;
```

declares an **sfr** variable IE that resides at byte address A8H. Recall that this address is where the Interrupt Enable (IE) SFR is located; therefore, the sfr data type is just a means to enable us to assign names for SFRs so that it is easier to remember.

The **sfr16** data type is very similar to **sfr** but, while the **sfr** data type is used for 8-bit SFRs, **sfr16** is used for 16-bit SFRs. For example, the following statement:

```
sfr16    DPTR = 0x82;
```

declares a 16-bit variable DPTR whose lower-byte address is at 82H. Checking through the 8051 architecture, we find that this is the address of the DPL SFR, so again, the **sfr16** data type makes it easier for us to refer to the SFRs by name rather than address. There's just one thing left to mention. When declaring **sbit**, **sfr**, or **sfr16** variables, remember to do so outside main, otherwise you will get an error.

In actual fact though, all the SFRs in the 8051, including the individual flag, status, and control bits in the bit-addressable SFRs have already been declared in an include file, called reg51.h, which comes packaged with most 8051 C compilers. By using reg51.h, we can refer for instance to the interrupt enable register as simply IE rather than having to specify the address A8H, and to the data pointer as DPTR rather than 82H. All this makes 8051 C programs more human-readable and manageable. The contents of reg51.h are listed below.

```
/*--------------------------------------------------------------------------------------------------------------------
REG51.H
Header file for generic 8051 microcontroller.
----------------------------------------------------------------------------------------------------------------------*/
/* BYTE Register */
sfr     P0      = 0x80;
sfr     P1      = 0x90;
sfr     P2      = 0xA0;
sfr     P3      = 0xB0;
sfr     PSW     = 0xD0;
sfr     ACC     = 0xE0;
sfr     B       = 0xF0;
sfr     SP      = 0x81;
sfr     DPL     = 0x82;
sfr     DPH     = 0x83;
sfr     PCON    = 0x87;
sfr     TCON    = 0x88;
sfr     TMOD    = 0x89;
sfr     TL0     = 0x8A;
sfr     TL1     = 0x8B;
sfr     TH0     = 0x8C;
sfr     TH1     = 0x8D;
sfr     IE      = 0xA8;
sfr     IP      = 0xB8;
sfr     SCON    = 0x98;
sfr     SBUF    = 0x99;
/* BIT Register */
/* PSW */
sbit    CY      = 0xD7;
sbit    AC      = 0xD6;
sbit    F0      = 0xD5;
sbit    RS1     = 0xD4;
sbit    RS0     = 0xD3;
sbit    OV      = 0xD2;
sbit    P       = 0xD0;
/* TCON */
sbit    TF1     = 0x8F;
sbit    TR1     = 0x8E;
sbit    TF0     = 0x8D;
sbit    TR0     = 0x8C;
sbit    IE1     = 0x8B;
sbit    IT1     = 0x8A;
sbit    IE0     = 0x89;
sbit    IT0     = 0x88;
/* IE */
sbit    EA      = 0xAF;
sbit    ES      = 0xAC;
sbit    ET1     = 0xAB;
sbit    EX1     = 0xAA;
sbit    ET0     = 0xA9;
sbit    EX0     = 0xA8;
/* IP */
sbit    PS      = 0xBC;
sbit    PT1     = 0xBB;
sbit    PX1     = 0xBA;
sbit    PT0     = 0xB9;
sbit    PX0     = 0xB8;
/* P3 */
sbit    RD      = 0xB7;
sbit    WR      = 0xB6;
sbit    T1      = 0xB5;
sbit    T0      = 0xB4;
sbit    INT1    = 0xB3;
sbit    INT0    = 0xB2;
sbit    TXD     = 0xB1;
sbit    RXD     = 0xB0;
/* SCON */
sbit    SM0     = 0x9F;
sbit    SM1     = 0x9E;
sbit    SM2     = 0x9D;
sbit    REN     = 0x9C;
sbit    TB8     = 0x9B;
sbit    RB8     = 0x9A;
sbit    TI      = 0x99;
sbit    RI      = 0x98;
```

MEMORY TYPES AND MODELS

The 8051 has various types of memory space, including internal and external code and data memory. When declaring variables, it is hence reasonable to wonder in which type of memory those variables would reside. For this purpose, several memory type specifiers are available for use, as shown in following table.

Memory types used in 8051 C language	
Memory Type	Description (Size)
code	Code memory (64 Kbytes)
data	Directly addressable internal data memory (128 bytes)
idata	Indirectly addressable internal data memory (256 bytes)
bdata	Bit-addressable internal data memory (16 bytes)
xdata	External data memory (64 Kbytes)
pdata	Paged external data memory (256 bytes)

The first memory type specifier given in above table is **code**. This is used to specify that a variable is to reside in code memory, which has a range of up to 64 Kbytes. For example:

```
char    code    errormsg[ ] = "An error occurred" ;
```

declares a char array called errormsg that resides in code memory.

If you want to put a variable into data memory, then use either of the remaining five data memory specifiers in above table. Though the choice rests on you, bear in mind that each type of data memory affect the speed of access and the size of available data memory. For instance, consider the following declarations:

```
signed int  data  num1;
bit bdata  numbit;
unsigned int  xdata  num2;
```

The first statement creates a signed int variable num1 that resides in inernal **data** memory (00H to 7FH). The next line declares a bit variable numbit that is to reside in the bit-addressable memory locations (byte addresses 20H to 2FH), also known as **bdata**. Finally, the last line declares an unsigned int variable called num2 that resides in external data memory, **xdata**. Having a variable located in the directly addressable internal data memory speeds up access considerably; hence, for programs that are time-critical, the variables should be of type **data**. For other variants such as 8052 with internal data memory up to 256 bytes, the **idata** specifier may be used. Note however that this is slower than data since it must use indirect addressing. Meanwhile, if you would rather have your variables reside in external memory, you have the choice of declaring them as **pdata** or **xdata**. A variable declared to be in **pdata** resides in the first 256 bytes (a page) of external memory, while if more storage is required, **xdata** should be used, which allows for accessing up to 64 Kbytes of external data memory.

What if when declaring a variable you forget to explicitly specify what type of memory it should reside in, or you wish that all variables are assigned a default memory type without having to specify them one by one? In this case, we make use of **memory models**. The following table lists the various memory models that you can use.

Memory models used in 8051 C language	
Memory Model	Description
Small	Variables default to the internal data memory (data)
Compact	Variables default to the first 256 bytes of external data memory (pdata)
Large	Variables default to external data memory (xdata)

A program is explicitly selected to be in a certain memory model by using the C directive, #pragma. Otherwise, the default memory model is **small**. It is recommended that programs use the small memory model as it allows for the fastest possible access by defaulting all variables to reside in internal data memory.

The **compact** memory model causes all variables to default to the first page of external data memory while the **large** memory model causes all variables to default to the full external data memory range of up to 64 Kbytes.

ARRAYS

Often, a group of variables used to store data of the same type need to be grouped together for better readability. For example, the ASCII table for decimal digits would be as shown below.

ASCII table for decimal digits	
Decimal Digit	ASCII Code In Hex
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H

To store such a table in an 8051 C program, an array could be used. An array is a group of variables of the same data type, all of which could be accessed by using the name of the arrary along with an appropriate index.

The array to store the decimal ASCII table is:

```
int       table [10] =
{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39};
```

Notice that all the elements of an array are separated by commas. To access an individul element, an index starting from 0 is used. For instance, table[0] refers to the first element while table[9] refers to the last element in this ASCII table.

STRUCTURES

Sometime it is also desired that variables of different data types but which are related to each other in some way be grouped together. For example, the name, age, and date of birth of a person would be stored in different types of variables, but all refer to the person's personal details. In such a case, a structure can be declared. A structure is a group of related variables that could be of different data types. Such a structure is declared by:

```
struct    person  {
                  char name;
                  int age;
                  long DOB;
                  };
```

Once such a structure has been declared, it can be used like a data type specifier to create structure variables that have the member's name, age, and DOB. For example:

```
struct    person    grace = {"Grace", 22, 01311980};
```

would create a structure variable grace to store the name, age, and data of birth of a person called Grace. Then in order to access the specific members within the person structure variable, use the variable name followed by the dot operator (.) and the member name. Therefore, grace.name, grace.age, grace.DOB would refer to Grace's name, age, and data of birth, respectively.

POINTERS

When programming the 8051 in assembly, sometimes register such as R0, R1, and DPTR are used to store the addresses of some data in a certain memory location. When data is accessed via these registers, indirect addressing is used. In this case, we say that R0, R1, or DPTR are used to point to the data, so they are essentially pointers.

Correspondingly in C, indirect access of data can be done through specially defined pointer variables. Pointers are simply just special types of variables, but whereas normal variables are used to directly store data, pointer variables are used to store the addresses of the data. Just bear in mind that whether you use normal variables or pointer variables, you still get to access the data in the end. It is just whether you go directly to where it is stored and get the data, as in the case of normal variables, or first consult a directory to check the location of that data before going there to get it, as in the case of pointer variables.

Declaring a pointer follows the format:

```
data_type     *pointer_name;
```

where

data_type	refers to which type of data that the pointer is pointing to
*	denotes that this is a pointer variable
pointer_name	is the name of the pointer

As an example, the following declarations:

```
int   * numPtr
int   num;
numPtr = &num;
```

first declares a pointer variable called numPtr that will be used to point to data of type int. The second declaration declares a normal variable and is put there for comparison. The third line assigns the address of the num variable to the numPtr pointer. The address of any variable can be obtained by using the address operator, &, as is used in this example. Bear in mind that once assigned, the numPtr pointer contains the address of the num variable, not the value of its data.

The above example could also be rewritten such that the pointer is straightaway initialized with an address when it is first declared:

```
int   num;
int   * numPtr = &num;
```

In order to further illustrate the difference between normal variables and pointer variables, consider the following, which is not a full C program but simply a fragment to illustrate our point:

```
int   num = 7;
int   * numPtr = &num;
printf ("%d\n", num);
printf ("%d\n", numPtr);
printf ("%d\n", &num);
printf ("%d\n", *numPtr);
```

The first line declare a normal variable, num, which is initialized to contain the data 7. Next, a pointer variable, numPtr, is declared, which is initialized to point to the address of num. The next four lines use the printf( ) function, which causes some data to be printed to some display terminal connected to the serial port. The first such line displays the contents of the num variable, which is in this case the value 7. The next displays the contents of the numPtr pointer, which is really some weird-looking number that is the address of the num variable.The third such line also displays the addresss of the num variable because the address operator is used to obtain num's address. The last line displays the actual data to which the numPtr pointer is pointing, which is 7. The * symbol is called the indirection operator, and when used with a pointer, indirectly obtains the data whose address is pointed to by the pointer. Therefore, the output display on the terminal would show:

7
13452 (or some other weird-looking number)
13452 (or some other weird-looking number)
7

A Pointer's Memory Type

Recall that pointers are also variables, so the question arises where they should be stored. When declaring pointers, we can specify different types of memory areas that these pointers should be in, for example:

int * xdata numPtr = & num;

This is the same as our previous pointer examples. We declare a pointer numPtr, which points to data of type int stored in the num variable. The difference here is the use of the memory type specifier **xdata** after the *. This is specifies that pointer numPtr should reside in external data memory (**xdata**), and we say that the pointer's memory type is **xdata**.

Typed Pointers

We can go even further when declaring pointers. Consider the example:

int data * xdata numPtr = #

The above statement declares the same pointer numPtr to reside in external data memory (**xdata**), and this pointer points to data of type int that is itself stored in the variable num in internal data memory (**data**). The memory type specifier, **data**, before the * specifies the *data memory type* while the memory type specifier, **xdata**, after the * specifies the pointer memory type.

Pointer declarations where the data memory types are explicitly specified are called typed pointers. Typed pointers have the property that you specify in your code where the data pointed by pointers should reside. The size of typed pointers depends on the data memory type and could be one or two bytes.

Untyped Pointers

When we do not explicitly state the data memory type when declaring pointers, we get untyped pointers, which are generic pointers that can point to data residing in any type of memory. Untyped pointers have the advantage that they can be used to point to any data independent of the type of memory in which the data is stored. All untyped pointers consist of 3 bytes, and are hence larger than typed pointers. Untyped pointers are also generally slower because the data memory type is not determined or known until the complied program is run at runtime. The first byte of untyped pointers refers to the data memory type, which is simply a number according to the following table. The second and third bytes are,respectively,the higher-order and lower-order bytes of the address being pointed to.

An untyped pointer is declared just like normal C, where:

int * xdata numPtr = #

does not explicitly specify the memory type of the data pointed to by the pointer. In this case, we are using untyped pointers.

Data memory type values stored in first byte of untyped pointers	
Value	Data Memory Type
1	idata
2	xdata
3	pdata
4	data/bdata
5	code

FUNCTIONS

In programming the 8051 in assembly, we learnt the advantages of using subroutines to group together common and frequently used instructions. The same concept appears in 8051 C, but instead of calling them subroutines, we call them **functions**. As in conventional C, a function must be declared and defined. A function definition includes a list of the number and types of inputs, and the type of the output (return type), puls a description of the internal contents, or what is to be done within that function.

The format of a typical function definition is as follows:

```
return_type    function_name (arguments)    [memory]  [reentrant]  [interrupt]  [using]
{
        …
}
```

where

return_type	refers to the data type of the return (output) value
function_name	is any name that you wish to call the function as
arguments	is the list of the type and number of input (argument) values
memory	refers to an explicit memory model (small, compact or large)
reentrant	refers to whether the function is reentrant (recursive)
interrupt	indicates that the function is acctually an ISR
using	explicitly specifies which register bank to use

Consider a typical example, a function to calculate the sum of two numbers:

```
int  sum (int a, int b)
{
        return  a + b;
}
```

This function is called sum and takes in two arguments, both of type int. The return type is also int, meaning that the output (return value) would be an int. Within the body of the function, delimited by braces, we see that the return value is basically the sum of the two agruments. In our example above, we omitted explicitly specifying the options: memory, reentrant, interrupt, and using. This means that the arguments passed to the function would be using the default small memory model, meaning that they would be stored in internal data memory. This function is also by default non-recursive and a normal function, not an ISR. Meanwhile, the default register bank is bank 0.

Parameter Passing

In 8051 C, parameters are passed to and from functions and used as function arguments (inputs). Nevertheless, the technical details of where and how these parameters are stored are transparent to the programmer, who does not need to worry about these techinalities. In 8051 C, parameters are passed through the register or through memory. Passing parameters through registers is faster and is the default way in which things are done. The registers used and their purpose are described in more detail below.

Registers used in parameter passing				
Number of Argument	Char / 1-Byte Pointer	INT / 2-Byte Pointer	Long/Float	Generic Pointer
1	R7	R6 & R7	R4–R7	R1–R3
2	R5	R4 &R5	R4–R7	
3	R3	R2 & R3		

Since there are only eight registers in the 8051, there may be situations where we do not have enough registers for parameter passing. When this happens, the remaining parameters can be passed through fixed memory locations. To specify that all parameters will be passed via memory, the NOREGPARMs control directive is used. To specify the reverse, use the REGPARMs control directive.

Return Values

Unlike parameters, which can be passed by using either registers or memory locations, output values must be returned from functions via registers. The following table shows the registers used in returning different types of values from functions.

Registers used in returning values from functions		
Return Type	Register	Description
bit	Carry Flag (C)	
char/unsigned char/1-byte pointer	R7	
int/unsigned int/2-byte pointer	R6 & R7	MSB in R6, LSB in R7
long/unsigned long	R4–R7	MSB in R4, LSB in R7
float	R4–R7	32-bit IEEE format
generic pointer	R1–R3	Memory type in R3, MSB in R2, LSB in R1

附录C：STC12C5410AD系列单片机电气特性

Absolute Maximum Ratings

Parameter	Symbol	Min	Max	Unit
Srotage temperature	TST	-55	+125	℃
Operating temperature (I)	TA	-40	+85	℃
Operating temperature (C)	TA	0	+70	℃
DC power supply (5V)	VDD - VSS	-0.3	+6.0	V
DC power supply (3V)	VDD - VSS	-0.3	+4.0	V
Voltage on any pin	-	-0.5	5.5	V

DC Specification (5V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
V_{DD}	Operating Voltage	3.5	5.0	5.5	V	
I_{PD}	Power Down Current	-	< 0.1	-	uA	5V
I_{IDL}	Idle Current	-	3.0	-	mA	5V
I_{CC}	Operating Current	-	4	20	mA	5V
V_{IL1}	Input Low Voltage(P0,P1,P2,P3)	-	-	0.8	V	5V
V_{IL2}	Input Low Voltage (RESET, XTAL1)			1.5	V	5V
V_{IH1}	Input High Voltage (P0,P1,P2,P3)	2.0	-	-	V	5V
V_{IH2}	Input High Voltage (RESET)	3.0	-	-	V	5V
I_{OL1}	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	5V
I_{OH1}	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	150	230	-	uA	5V
I_{OH2}	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull, Strong-output)	-	20	-	mA	5V
I_{IL}	Logic 0 input current (P0,P1,P2,P3)	-	18	50	uA	Vpin=0V
I_{TL}	Logic 1 to 0 transition current (P0,P1,P2,P3)	-	270	600	uA	Vpin=2.0V

DC Specification (3V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
V_{DD}	Operating Voltage	2.2	3.3	3.8	V	
I_{PD}	Power Down Current	-	<0.1	-	uA	3.3V
I_{IDL}	Idle Current	-	2.0	-	mA	3.3V
I_{CC}	Operating Current	-	4	10	mA	3.3V
V_{IL1}	Input Low (P0,P1,P2,P3)	-	-	0.8	V	3.3V
V_{IL2}	Input low voltage (RESET,XTAL1)	-	-	1.5	V	3.3V
V_{IH1}	Input High (P0,P1,P2,P3)	2.0	-	-	V	3.3V
V_{IH2}	Input High (RESET)	3.0	-	-	V	3.3V
I_{OL1}	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	3.3V@Vpin=0.45V
I_{OH1}	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	40	70	-	uA	3.3V
I_{OH2}	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull)	-	20	-	mA	3.3V
I_{IL}	Logic 0 input current (P0,P1,P2,P3)	-	8	50	uA	Vpin=0V
I_{TL}	Logic 1 to 0 transition current (P0,P1,P2,P3)	-	110	600	uA	Vpin=2.0V

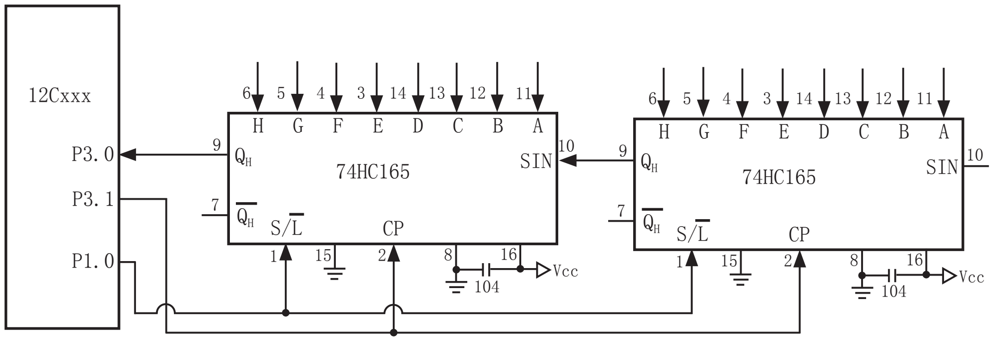
附录D：用串口扩展I/O接口

STC12C5410AD系列单片机串行口的方式0可用于I/O扩展。如果在应用系统中，串行口未被占用，那么将它用来扩展并行I/O口是一种经济、实用的方法。

在操作方式0时，串行口作同步移位寄存器，其波特率是固定的，为SYSclk/12（SYSclk为系统时钟频率）。数据由RXD端（P3.0）出入，同步移位时钟由TXD端（P3.1）输出。发送、接收的是8位数据，低位在先。

一、用74HC165扩展并行输入口

下图是利用两片74HC165扩展二个8位并行输入口的接口电路图。



74HC165是8位并行置入移位寄存器。当移位/置入端(S/L)由高到低跳变时，并行输入端的数据置入寄存器；当S/L=1，且时钟禁止端（第15脚）为低电平时，允许时钟输入，这时在时钟脉冲的作用下，数据将由Q_A到Q_H方向移位。

上图中，TXD(P3.1)作为移位脉冲输出端与所有74HC165的移位脉冲输入端CP相连；RXD（P3.0）作为串行输入端与74HC165的串行输出端Q_H相连；P1.0用来控制74HC165的移位与置入而同S/L相连；74HC165的时钟禁止端（15脚）接地，表示允许时钟输入。当扩展多个8位输入口时，两芯片的首尾（Q_H与S_{IN}）相连。

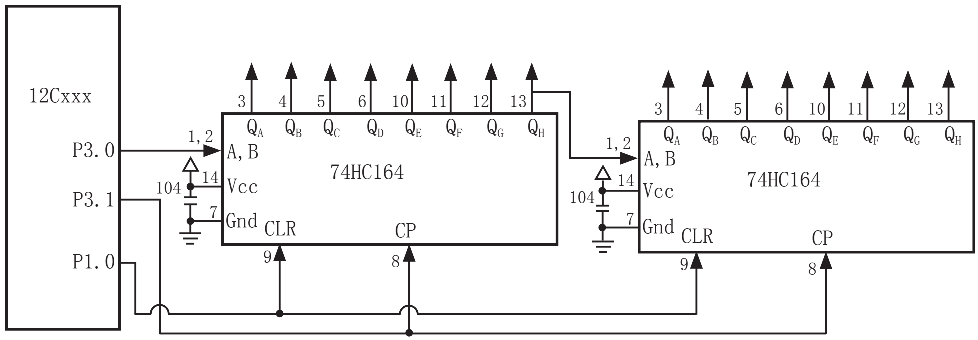
下面的程序是从16位扩展口读入5组数据（每组二个字节），并把它们转存到内部RAM 20H开始的单元中。

```
            MOV     R7，#05H                 ；设置读入组数
            MOV     R0，#20H                 ；设置内部RAM数据区首址
START：     CLR     P1.0                     ；并行置入数据，S/L=0
            SETB P1.0                        ；允许串行移位S/L=1
            MOV     R1，#02H               ；设置每组字节数，即外扩74LS165的个数
RXDATA：    MOV     SCON，#00010000B       ；设串行方式0，允许接收，启动接收过程
WAIT：      JNB     RI，WAIT                 ；未接收完一帧，循环等待
            CLR     RI                       ；清RI标志，准备下次接收
            MOV     A，SBUF                  ；读入数据
            MOV     @R0，A                   ；送至RAM缓冲区
            INC     R0                       ；指向下一个地址
            DJNZ    R1，RXDATA               ；为读完一组数据，继续
            DJNZ    R7，START                ；5组数据未读完重新并行置入
            ……                               ；对数据进行处理
```

上面的程序对串行接收过程采用的是查询等待的控制方式，如有必要，也可改用中断方式。从理论上讲，按上图方法扩展的输入口几乎是无限的，但扩展的越多，口的操作速度也就越慢。

二、用74HC164扩展并行输出口

74HC164是8位串入并出移位寄存器。下图是利用74HC164扩展二个8位输出口的接口电路。

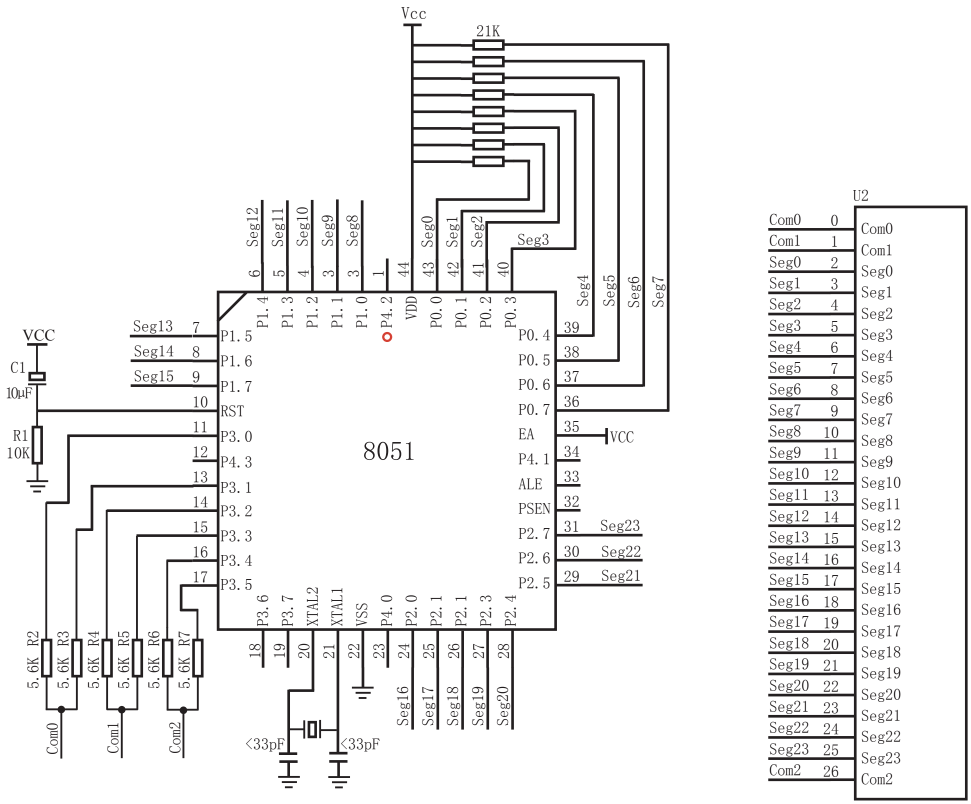


当单片机串行口工作在方式0的发送状态时，串行数据由P3.0（RXD）送出，移位时钟由P3.1（TXD）送出。在移位时钟的作用下，串行口发送缓冲器的数据一位一位地移入74HC164中。需要指出的是，由于74HC164无并行输出控制端，因而在串行输入过程中，其输出端的状态会不断变化，故在某些应用场合，在74HC164的输出端应加接输出三态门控制，以便保证串行输入结束后再输出数据。

下面是将RAM缓冲区30H、31H的内容串行口由74HC164并行输出的子程序。

```
START： MOV      R7，#02H       ；设置要发送的字节个数
        MOV      R0，#30H       ；设置地址指针
        MOV      SCON，#00H     ；设置串行口方式0
SEND：  MOV      A，@R0
        MOV      SBUF，A        ；启动串行口发送过程
WAIT：  JNB      TI，WAIT       ；一帧数据未发送完，循等待
        CLR      TI
        INC      R0             ；取下一个数
        DJNZ     R7，SEND
        RET
```

附录E：利用STC单片机普通I/O驱动LCD显示



本资料不提供技术支持，请自行消化吸收

```
NAME LcdDriver
#include<reg52.h>

;*******************************************************************************
;the LCD is 1/3 duty and 1/3 bias; 3Com*24Seg; 9 display RAM;
;
;                          Bit7    Bit6    Bit5    Bit4    Bit3    Bit2    Bit1    Bit0
;Com0:  Com0Data0:         Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com0Data1:         Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com0Data2:         Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;Com1:  Com1Data0:         Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com1Data1:         Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com1Data2:         Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;Com2:  Com2Data0:         Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com2Data1:         Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com2Data2:         Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;*******************************************************************************
;Com0:  P3^0,P3^1  when P3^0 = P3^1 = 1     then Com0=VCC(=5V);
;                       P3^0 = P3^1 = 0     then Com0=GND(=0V);
;                       P3^0 = 1, P3^1=0    then Com0=1/2 VCC;
;Com1:  P3^2,P3^3  the same as the Com0
;Com2:  P3^4,P3^5  the same as the Com0
;

sbit    SEG0  =P0^0
sbit    SEG1  =P0^1
sbit    SEG2  =P0^2
sbit    SEG3  =P0^3
sbit    SEG4  =P0^4
sbit    SEG5  =P0^5
sbit    SEG6  =P0^6
sbit    SEG7  =P0^7
sbit    SEG8  =P1^0
sbit    SEG9  =P1^1
sbit    SEG10 =P1^2
```

```
sbit    SEG11 =P1^3
sbit    SEG12 =P1^4
sbit    SEG13 =P1^5
sbit    SEG14 =P1^6
sbit    SEG15 =P1^7
sbit    SEG16 =P2^0
sbit    SEG17 =P2^1
sbit    SEG18 =P2^2
sbit    SEG19 =P2^3
sbit    SEG20 =P2^4
sbit    SEG21 =P2^5
sbit    SEG22 =P2^6
sbit    SEG23 =P2^7
;********************************************************************************

;======Interrupt=================================
    CSEG        AT      0000H
    LJMP        start

    CSEG        AT      000BH
    LJMP        int_t0

;======register=================================
lcdd_bit SEGMENT BIT
    RSEG lcdd_bit
    OutFlag:    DBIT 1      ;the output display reverse flag
lcdd_data SEGMENT DATA
    RSEG lcdd_data
    Com0Data0:   DS  1
    Com0Data1:   DS  1
    Com0Data2:   DS  1
    Com1Data0:   DS  1
    Com1Data1:   DS  1
    Com1Data2:   DS  1
    Com2Data0:   DS  1
    Com2Data1:   DS  1
    Com2Data2:   DS  1
    TimeS:       DS  1
```

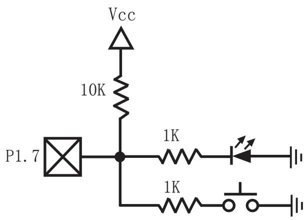
```
;======Interrupt Code===========================
t0_int    SEGMENT       CODE
          RSEG    t0_int
          USING  1
;******************************************************************
;Time0 interrupt
;ths system crystalloid is 22.1184MHz
;the time to get the Time0 interrupr is 2.5mS
;the whole duty is 2.5mS*6=15mS, including reverse
;******************************************************************
int_t0:
          ORL      TL0,#00H
          MOV      TH0,#0EEH
          PUSH     ACC
          PUSH     PSW
          MOV      PSW,#08H
          ACALL OutData
          POP      PSW
          POP      ACC
          RETI

;======SUB CODE==================================
uart_sub SEGMENT CODE
          RSEG     uart_sub
          USING  0
;*******************************************************************
;initial the display RAM data
;if want to display other,then you may add other data to this RAM
;Com0:   Com0Data0,Com0Data1,Com0Data2
;Com1:   Com1Data0,Com1Data1,Com1Data2
;Com2:   Com2Data0,Com0Data1,Com0Data2
;********************************************************************
InitComData:                ;it will display "11111111"
          MOV      Com0Data0,      #24H
          MOV      Com0Data1,      #49H
          MOV      Com0Data2,      #92H
```

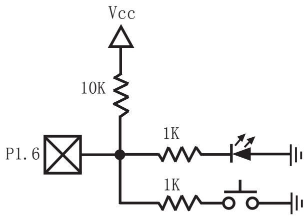
```
        MOV     Com1Data0,      #92H
        MOV     Com1Data1,      #24H
        MOV     Com1Data2,      #49H
        MOV     Com2Data0,      #00H
        MOV     Com2Data1,      #00H
        MOV     Com2Data2,      #00H
        RET

;*******************************************************************
;reverse the display data
;*******************************************************************
RetComData:
        MOV     R0,     #Com0Data0          ;get the first data address
        MOV     R7,     #9
RetCom_0:
        MOV     A,      @R0
        CPL     A
        MOV     @R0,    A
        INC     R0
        DJNZ    R7,     RetCom_0
        RET
;*******************************************************************
;get the display Data and send to Output register
;*******************************************************************
OutData:
        INC     TimeS
        MOV     A,      TimeS
        MOV     P3,     #11010101B              ;clear display,all Com are 1/2VCC and invalidate
        CJNE    A,      #01H,   OutData_1       ;judge the duty
        MOV     P0,     Com0Data0
        MOV     P1,     Com0Data1
        MOV     P2,     Com0Data2
        JNB     OutFlag,OutData_00
        MOV     P3,     #11010111B              ;Com0 is work and is VCC
        RET
```

```
OutData_00:
        MOV     P3,     #11010100B          ;Com0 is work and is GND
        RET
OutData_1:
        CJNE    A,      #02H,OutData_2
        MOV     P0,     Com1Data0
        MOV     P1,     Com1Data1
        MOV     P2,     Com1Data2
        JNB     OutFlag,OutData_10
        MOV     P3,     #11011101B          ;Com1 is work and is VCC
        RET
OutData_10:
        MOV     P3,     #11010001B          ;Com1 is work and is GND
        RET
OutData_2:
        MOV     P0,     Com2Data0
        MOV     P1,     Com2Data1
        MOV     P2,     Com2Data2
        JNB     OutFlag,OutData_20
        MOV     P3,     #11110101B          ;Com2 is work and is VCC
        SJMP    OutData_21
OutData_20:
        MOV  P3,#11000101B          ;Com2 is work and is GND
OutData_21:
        MOV     TimeS,  #00H
        ACALL RetComData
        CPL     OutFlag
        RET

;======Main Code================================
uart_main SEGMENT CODE
        RSEG    uart_main
        USING   0
```

```
start:
        MOV     SP,#40H
        CLR     OutFlag
        MOV     TimeS,#00H
        MOV     TL0,#00H
        MOV     TH0,#0EEH
        MOV     TMOD,#01H
        MOV     IE,#82H
        ACALL InitComData
        SETB    TR0
Main:
        NOP
        SJMP    Main

        END
```

附录F：一个I/O口驱动发光二极管并扫描按键





利用STC12C5410AD系列单片机的I/O口可设置成弱上拉,强上拉(推挽)输出,仅为输入(高阻),开漏四种模式的特性,可以利用STC12C5410AD系列单片机的I/O口同时作为发光二极管驱动及按键检测用，可以大幅节省I/O口。

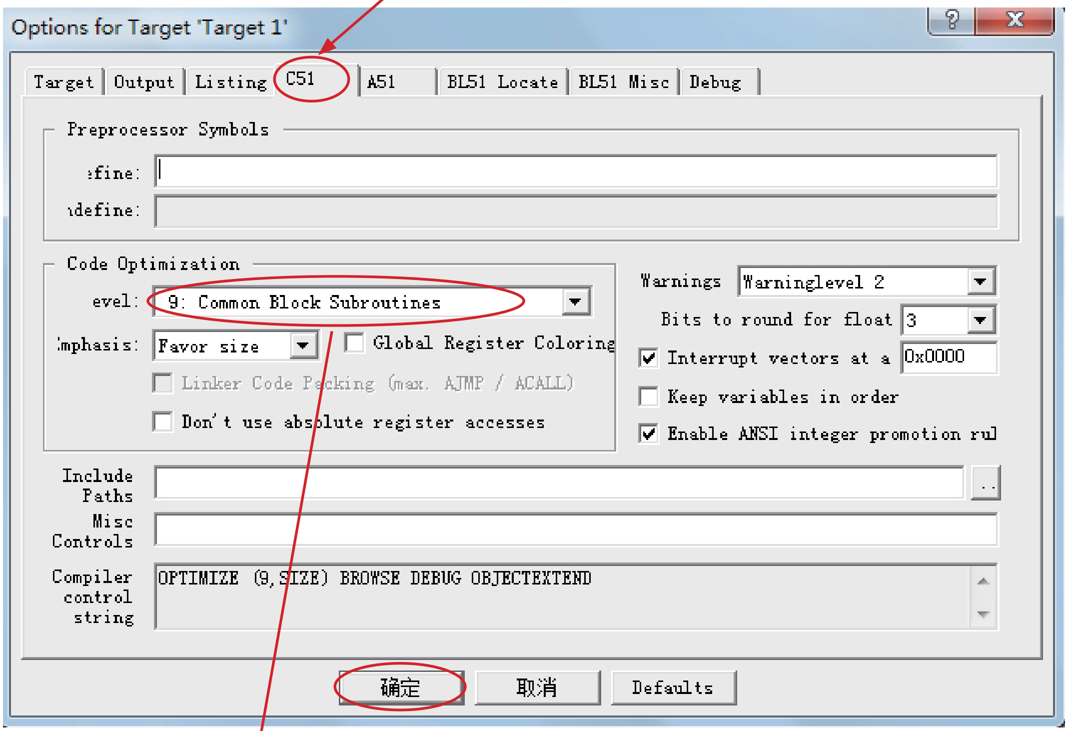
当驱动发光二极管时，将该I/O口设置成强推挽输出，输出高即可点亮发光二极管。

当检测按键时，将该I/O口设置成弱上拉输入，再读外部口的状态，即可检测按键。

附录G：如何利用Keil C软件减少代码长度

在Keil C软件中选择作如下设置，能将原代码长度最大减少10K。

1. 在“Project”菜单中选择“Options for Target”
2. 在“Options for Target”中选择“C51”



3. 选择按空间大小，9级优化程序
4. 点击“确定”后，重新编译程序即可。

附录H：STC12C5410AD系列单片机应用注意事项

关于复位电路：

晶振频率在20M 以下时:可以不用外部复位电路，原复位电路可以保留，也可以不用，不用时复位脚可经过1K 电阻短接到地,或者直接短接到地。不过建议设计时PCB 板上保留R/C 复位电路，实际使用时再决定用或不用。

关于时钟：

如果使用内部R/C 振荡器时钟(4MHz～8MHz,制造误差加温漂),XTAL1 和XTAL2 脚浮空.

如果外部时钟频率在27MHz以上时,建议采用实际基本频率就是标称频率的晶体,不要采用三泛音的晶体(基本频率是标称频率的1/3),因为外围参数搭配不当,时钟往往振荡在标称频率的1/3,即基频.或直接使用外部有源石英晶体振荡器,时钟从XTAL1 脚输入,XTAL2 脚必须浮空.

关于I/O口：

少数用户反映I/O口有损坏现象,后发现是

有些是I/O口由低变高读外部状态时,读不对,实际没有损坏,软件处理一下即可

是因为1T的8051单片机速度太快了,软件执行由低变高指令后立即读外部状态,此时由于实际输出还没有变高,就有可能读不对,正确的方法是在软件设置由低变高后加1到2个空操作指令延时,再读就对了.

有些实际没有损坏,加上拉电阻就OK了

是因为外围接的是SPI/I2C等漏极开漏的电路,要加10K上拉电阻.

有些是外围接的是NPN三极管,没有加上拉电阻,其实基极串多大电阻,I/O口就应该上拉多大的电阻,或者将该I/O 口设置为强推挽输出.

有些确实是损坏了,原因:

发现有些是驱动LED发光二极管没有加限流电阻,建议加1K以上的限流电阻,至少也要加470欧姆以上

发现有些是做行列矩阵按键扫描电路时,实际工作时没有加限流电阻,实际工作时可能出现2个I/O口均输出为低,并且在按键按下时,短接在一起,我们知道一个CMOS电路的2个输出脚不应该直接短接在一起,按键扫描电路中,此时一个口为了读另外一个口的状态,必须先置高才能读另外一个口的状态,而8051单片机的弱上拉口在由0变为1时,会有2个时钟的强推挽高输出电流,输出到另外一个输出为低的I/O口,就有可能造成I/O口损坏.建议在其中的一侧加1K限流电阻,或者在软件处理上,不要出现按键两端的I/O口同时为低.

关于电源：

在电源两端应该加一个10uF以上的电解电容和一个0.1uF的小电容,进行电源去藕滤波.

附录I：每日更新内容的备忘录

2011-9-9更新内容：

增加了5.3.1节“（中文的）传统8051单片机指令定义详解”，并附有5.3.2节的英文文本参考。