STC15F100系列单片机器件手册

STC15F100	
STC15F101	STC15F101E
STC15F102	STC15F102E
STC15F103	STC15F103 E
STC15F104	STC15F104 E
STC15F105	STC15F105 E
IAP15F106	

Update date: 2010/9/12

目录

第1章 STC15F100系列单片机总体介绍

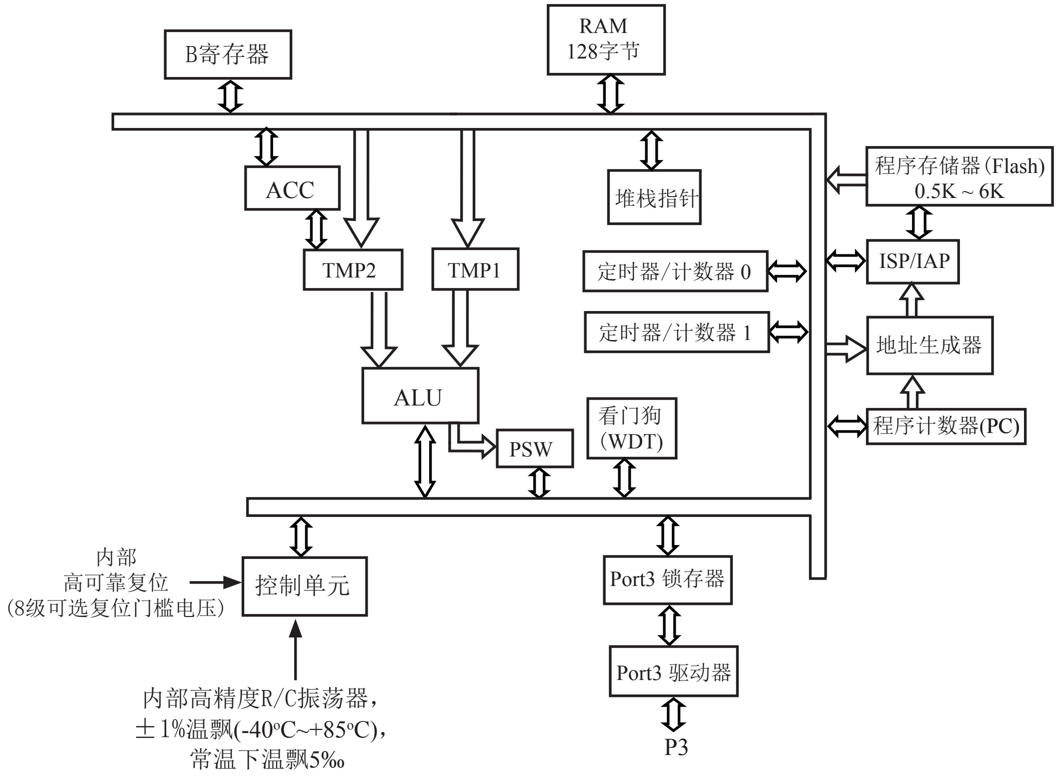
1.1 STC15F100系列单片机简介

STC15F100系列单片机是宏晶科技生产的单时钟/机器周期(1T)的单片机，是高速/高可靠/低功耗/超强抗干扰的新一代8051单片机，采用宏晶第八代加密技术，加密性超强，指令代码完全兼容传统8051，但速度快6-12倍。内部集成高精度R/C时钟，±1%温飘，常温下温飘5‰，5MHz~35MHz宽范围可设置，可彻底省掉外部昂贵的晶振。内部高可靠复位，8级可选复位门槛电压，可彻底省掉外部复位电路。

1. 增强型 8051 CPU，1T，单时钟/机器周期，速度比普通8051快6-12倍
2. 工作电压：
   STC15F100 系列工作电压：5.5V - 3.8V（5V 单片机）
   STC15L100 系列工作电压：3.6V - 2.4V（3V 单片机）
3. 内部高可靠复位，8级可选复位门槛电压，彻底省掉外部复位电路
4. 内部高精度R/C时钟，±1%温飘(-40°C~+85°C)，常温下温飘5‰，内部时钟从5MHz ~ 35MHz可选(5.5296MHz / 11.0592MHz / 22.1184MHz / 33.1776MHz)
5. 工作频率范围：5MHz ~ 35MHz，相当于普通8051的60MHz～420MHz
6. 低功耗设计：低速模式，空闲模式，掉电模式/停机模式(可由外部中断唤醒)
7. 支持掉电唤醒的管脚：INT0/P3.2, INT1/P3.3, $\overline{\text{INT2}}$, $\overline{\text{INT3}}$, $\overline{\text{INT4}}$
8. 0.5K/1K/2K/3K/4K/5K/6K字节片内Flash程序存储器， 擦写次数10万次以上
9. 片上集成 128 字节 RAM
10. 有EEPROM 功能，擦写次数10万次以上
11. ISP/IAP，在系统可编程/在应用可编程，无需编程器/仿真器
12. 2个16位可重装载定时器，兼容普通8051的定时器T0/T1，并可实现时钟输出和PWM功能。
13. 可编程时钟输出功能，T0在P3.5输出时钟，T1在P3.4输出时钟，在P3.4口还可输出内部高精度R/C时钟IRC_CLK(也可2分频输出IRC_CLK/2)。
14. 硬件看门狗(WDT)
15. 串口功能可由[P3.0/$\overline{\text{INT4}}$,P3.1]结合定时器实现
16. 先进的指令集结构，兼容普通8051指令集，有硬件乘法/除法指令
17. 6个通用I/O口，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70mA
18. 封装：SOP-8, DIP-8
19. 全部175°C 八小时高温烘烤，高品质制造保证

1.2 STC15F100系列单片机的内部结构

STC15F100系列单片机的内部结构框图如下图所示。STC15F100系列单片机中包含中央处理器(CPU)、程序存储器(Flash)、数据存储器(SRAM)、定时器、I/O口、看门狗、片内高精度R/C振荡时钟及高可靠复位等模块。



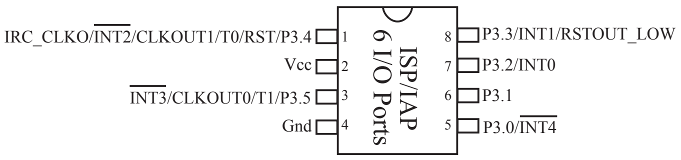
STC15F100系列内部结构框图

1.3 STC15系列单片机管脚图

1.3.1 STC15F100系列单片机管脚图

所有封装形式均满足欧盟RoHS要求，

强烈推荐选择SOP-8/28/20贴片封装，传统的插件SKDIP/DIP封装稳定供货

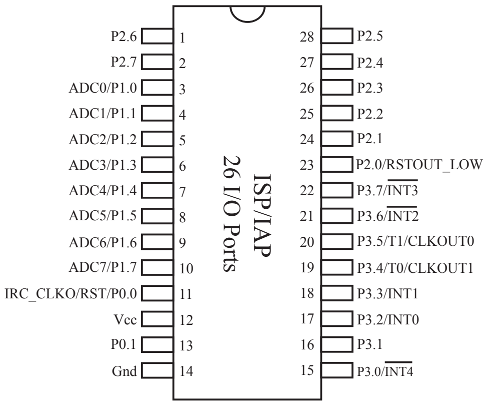


IRC_CLKO/$\overline{\text{INT2}}$/CLKOUT1/T0/RST/P3.4
Vcc
$\overline{\text{INT3}}$/CLKOUT0/T1/P3.5
Gnd
1
2
3
4
8
7
6
5
ISP/IAP
6 I/O Ports
P3.3/INT1/RSTOUT_LOW
P3.2/INT0
P3.1
P3.0/$\overline{\text{INT4}}$

SOP-8/DIP-8

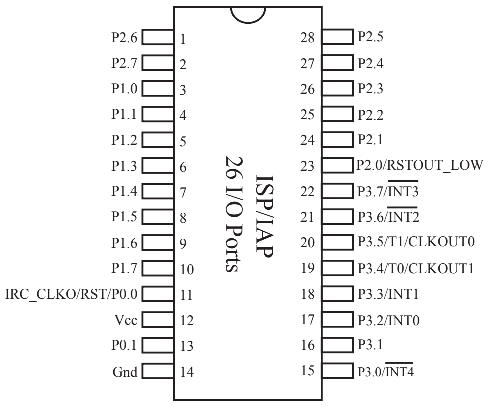
1.3.2 STC15F204EA系列单片机管脚图

STC15F204EA系列管脚图



SOP-28/SKDIP-28

STC15F204A系列管脚图

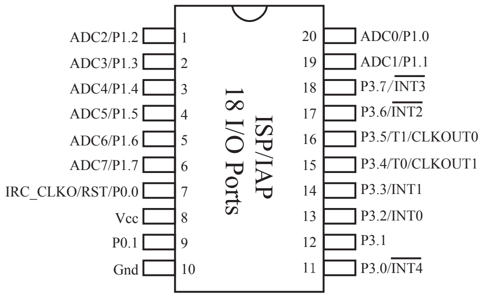


SOP-28/SKDIP-28

STC15F204EA系列(有A/D转换，有内部EEPROM)
STC15F204A系列 (有A/D转换，无内部EEPROM)

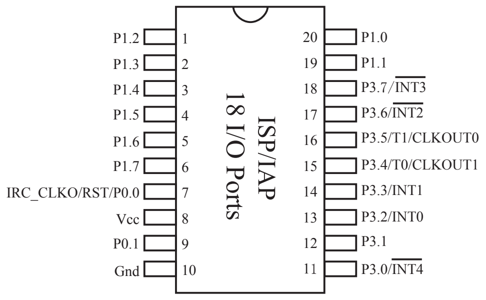
1.3.3 STC15S204EA系列单片机管脚图

STC15S204EA系列管脚图



SOP-20/DIP-20

STC15S204A系列管脚图



SOP-20/DIP-20

STC15S204EA系列(有A/D转换，有内部EEPROM)
STC15S204A系列 (有A/D转换，无内部EEPROM)

STC15S204EA系列是STC15F204EA系列的特殊版本，需延后至2010-10-25方有样品可提供。

1.4 STC15系列单片机选型一览表

1.4.1 STC15F100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byte)	SRAM字节	定时器	A/D 8路	看门狗(WDT)	内置复位	EEPROM	内部低压检测中断	内部可选复位门槛电压	支持掉电唤醒外部中断	掉电唤醒专用定时器	封装8-Pin（6个I/O口）价格(RMB ¥)	
													SOP-8	DIP-8
STC15F100系列单片机选型一览														
STC15F100	5.5-3.8	512	128	2	－	有	有	-	有	8级	5	－		
STC15F101	5.5-3.8	1K	128	2	－	有	有	-	有	8级	5	－		
STC15F101E	5.5-3.8	1K	128	2	－	有	有	2K	有	8级	5	－		
STC15F102	5.5-3.8	2K	128	2	－	有	有	-	有	8级	5	－		
STC15F102E	5.5-3.8	2K	128	2	－	有	有	2K	有	8级	5	－		
STC15F103	5.5-3.8	3K	128	2	－	有	有	-	有	8级	5	－		
STC15F103E	5.5-3.8	3K	128	2	－	有	有	2K	有	8级	5	－		
STC15F104	5.5-3.8	4K	128	2	－	有	有	-	有	8级	5	－		
STC15F104E	5.5-3.8	4K	128	2	－	有	有	1K	有	8级	5	－		
STC15F105	5.5-3.8	5K	128	2	－	有	有	-	有	8级	5	－		
STC15F105E	5.5-3.8	5K	128	2	－	有	有	1K	有	8级	5	－		
IAP15F106	5.5-3.8	6K	128	2	－	有	有	IAP	有	8级	5	－		
STC15L100系列单片机选型一览表														
STC15L100	3.6-2.4	512	128	2	－	有	有	-	有	8级	5	－		
STC15L101	3.6-2.4	1K	128	2	－	有	有	-	有	8级	5	－		
STC15L101E	3.6-2.4	1K	128	2	－	有	有	2K	有	8级	5	－		
STC15L102	3.6-2.4	2K	128	2	－	有	有	-	有	8级	5	－		
STC15L102E	3.6-2.4	2K	128	2	－	有	有	2K	有	8级	5	－		
STC15L103	3.6-2.4	3K	128	2	－	有	有	-	有	8级	5	－		
STC15L103E	3.6-2.4	3K	128	2	－	有	有	2K	有	8级	5	－		
STC15L104	3.6-2.4	4K	128	2	－	有	有	-	有	8级	5	－		
STC15L104E	3.6-2.4	4K	128	2	－	有	有	1K	有	8级	5	－		
STC15L105	3.6-2.4	5K	128	2	－	有	有	-	有	8级	5	－		
STC15L105E	3.6-2.4	5K	128	2	－	有	有	1K	有	8级	5	－		
IAP15L106	3.6-2.4	6K	128	2	－	有	有	IAP	有	8级	5	－		

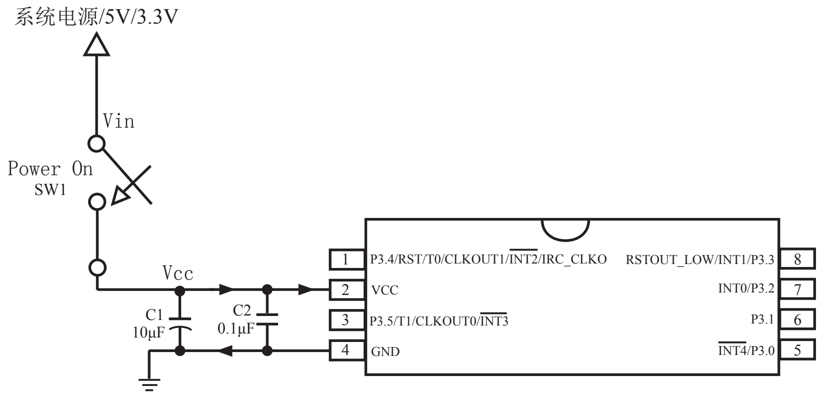
1.4.2 STC15F204EA系列单片机选型一览表

型号	工作电压（V）	Flash程序存储器(字节byte)	SRAM字节	定时器	A/D 8路	看门狗(WDT)	内置复位	EEPROM	内部低压检测中断	内部可选复位门槛电压	支持掉电唤醒外部中断	掉电唤醒专用定时器	封装28-Pin（26个I/O口）价格(RMB ¥) SOP-28	SKDIP-28
STC15F204EA系列单片机选型一览														
STC15F201A	5.5-3.8	1K	256	2	10位	有	有	-	有	8级	5	-		
STC15F201EA	5.5-3.8	1K	256	2	10位	有	有	2K	有	8级	5	-		
STC15F202A	5.5-3.8	2K	256	2	10位	有	有	-	有	8级	5	-		
STC15F202EA	5.5-3.8	2K	256	2	10位	有	有	2K	有	8级	5	-		
STC15F203A	5.5-3.8	3K	256	2	10位	有	有	-	有	8级	5	-		
STC15F203EA	5.5-3.8	3K	256	2	10位	有	有	2K	有	8级	5	-		¥2.65
STC15F204A	5.5-3.8	4K	256	2	10位	有	有	-	有	8级	5	-		
STC15F204EA	5.5-3.8	4K	256	2	10位	有	有	1K	有	8级	5	-		¥2.70
STC15F205A	5.5-3.8	5K	256	2	10位	有	有	-	有	8级	5	-		
STC15F205EA	5.5-3.8	5K	256	2	10位	有	有	1K	有	8级	5	-		¥2.75
IAP15F206A	5.5-3.8	6K	256	2	10位	有	有	IAP	有	8级	5	-		
STC15L204EA系列单片机选型一览表														
STC15L201A	3.6-2.4	1K	256	2	10位	有	有	-	有	8级	5	-		
STC15L201EA	3.6-2.4	1K	256	2	10位	有	有	2K	有	8级	5	-		¥2.55
STC15L202A	3.6-2.4	2K	256	2	10位	有	有	-	有	8级	5	-		
STC15L202EA	3.6-2.4	2K	256	2	10位	有	有	2K	有	8级	5	-		¥2.60
STC15L203A	3.6-2.4	3K	256	2	10位	有	有	-	有	8级	5	-		
STC15L203EA	3.6-2.4	3K	256	2	10位	有	有	2K	有	8级	5	-		¥2.65
STC15L204A	3.6-2.4	4K	256	2	10位	有	有	-	有	8级	5	-		
STC15L204EA	3.6-2.4	4K	256	2	10位	有	有	1K	有	8级	5	-		¥2.70
STC15L205A	3.6-2.4	5K	256	2	10位	有	有	-	有	8级	5	-		
STC15L205EA	3.6-2.4	5K	256	2	10位	有	有	1K	有	8级	5	-		¥2.75
IAP15L206A	3.6-2.4	6K	256	2	10位	有	有	IAP	有	8级	5	-		

1.4.3 STC15S204EA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byte)	SRAM字节	定时器	A/D 8路	看门狗(WDT)	内置复位	EEP ROM	内部低压检测中断	内部可选复位门槛电压	支持掉电唤醒外部中断	掉电唤醒专用定时器	封装20-Pin（18个I/O口）价格(RMB ¥)	
													SOP-20	DIP-20
STC15S204EA系列单片机选型一览														
STC15S201A	5.5-3.8	1K	256	2	10位	有	有	-	有	8级	5	−		
STC15S201EA	5.5-3.8	1K	256	2	10位	有	有	2K	有	8级	5	−		
STC15S202A	5.5-3.8	2K	256	2	10位	有	有	-	有	8级	5	−		
STC15S202EA	5.5-3.8	2K	256	2	10位	有	有	2K	有	8级	5	−		
STC15S203A	5.5-3.8	3K	256	2	10位	有	有	-	有	8级	5	−		
STC15S203EA	5.5-3.8	3K	256	2	10位	有	有	2K	有	8级	5	−		
STC15S204A	5.5-3.8	4K	256	2	10位	有	有	-	有	8级	5	−		
STC15S204EA	5.5-3.8	4K	256	2	10位	有	有	1K	有	8级	5	−		
STC15S205A	5.5-3.8	5K	256	2	10位	有	有	-	有	8级	5	−		
STC15S205EA	5.5-3.8	5K	256	2	10位	有	有	1K	有	8级	5	−		
IAP15S206A	5.5-3.8	6K	256	2	10位	有	有	IAP	有	8级	5	−		
STC15V204EA列单片机选型一览表														
STC15V201A	3.6-2.4	1K	256	2	10位	有	有	-	有	8级	5	−		
STC15V201EA	3.6-2.4	1K	256	2	10位	有	有	2K	有	8级	5	−		
STC15V202A	3.6-2.4	2K	256	2	10位	有	有	-	有	8级	5	−		
STC15V202EA	3.6-2.4	2K	256	2	10位	有	有	2K	有	8级	5	−		
STC15V203A	3.6-2.4	3K	256	2	10位	有	有	-	有	8级	5	−		
STC15V203EA	3.6-2.4	3K	256	2	10位	有	有	2K	有	8级	5	−		
STC15V204A	3.6-2.4	4K	256	2	10位	有	有	-	有	8级	5	−		
STC15V204EA	3.6-2.4	4K	256	2	10位	有	有	1K	有	8级	5	−		
STC15V205A	3.6-2.4	5K	256	2	10位	有	有	-	有	8级	5	−		
STC15V205EA	3.6-2.4	5K	256	2	10位	有	有	1K	有	8级	5	−		
IAP15V206A	3.6-2.4	6K	256	2	10位	有	有	IAP	有	8级	5	−		

1.5 STC15F100系列单片机最小应用系统

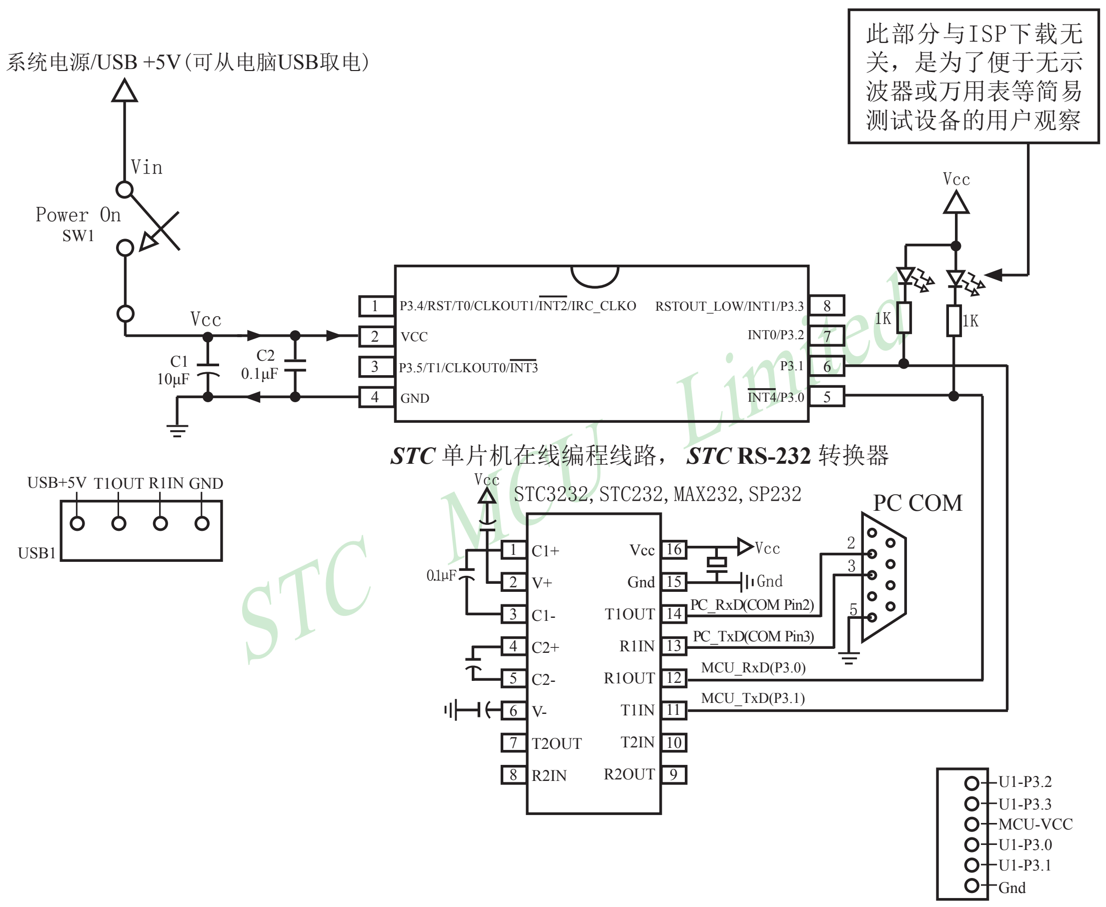

系统电源/5V/3.3V
Vin
Power On
SW1
Vcc
C1
10μF
C2
0.1μF
1
2
3
4
5
6
7
8
P3.4/RST/T0/CLKOUT1/$\overline{\text{INT2}}$/IRC_CLKO
VCC
P3.5/T1/CLKOUT0/$\overline{\text{INT3}}$
GND
RSTOUT_LOW/INT1/P3.3
INT0/P3.2
P3.1
$\overline{\text{INT4}}$/P3.0

内部高可靠复位，不需要外部复位电路
P3.4/RST/T0/CLKOUT1/$\overline{\text{INT2}}$/IRC_CLKO脚出厂时默认为I/O口，可以通过 STC-ISP 编程器将其设置为RST复位脚.

内部高精度R/C振荡器，温飘±1%(-40^{0}C~+85^{0}C), 常温下温飘5‰, 不需要昂贵的外部晶振

建议加上电容C1(10μF), C2(0.1μF), 可去除电源噪声，提高抗干扰能力

1.6 STC15F100系列在系统可编程(ISP)典型应用线路图



内部高可靠复位，不需要外部复位电路

P3.4/RST/T0/CLKOUT1/$\overline{\text{INT2}}$/IRC_CLKO脚出厂时默认为I/O口，可以通过 STC-ISP 编程器将其设置为RST复位脚.

内部高精度R/C振荡器，温飘±1%(-40^{0}C~+85^{0}C), 常温下温飘5‰, 不需要昂贵的外部晶振

建议加上电容C1(10μF), C2(0.1μF), 可去除电源噪声，提高抗干扰能力

1.7 STC15F100系列管脚说明

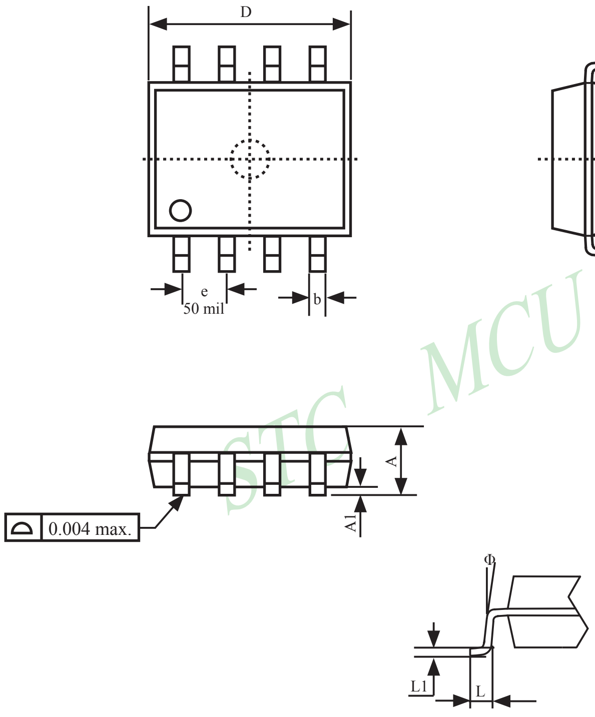
管脚	管脚编号	说明	
P3.0/$\overline{\text{INT4}}$	5	P3.0	标准I/O口 PORT3[0]
		$\overline{\text{INT4}}$	外部中断4, 只能下降沿中断 支持掉电唤醒
P3.1	6	标准I/O口 PORT3[1]	
P3.2/INT0	7	P3.2	标准I/O口 PORT3[2]
		INT0	外部中断0，既可上升沿中断也可下降沿中断. 如果IT0(TCON.0)被置为1, INT0管脚仅为下降沿中断。如果IT0(TCON.0)被清0，INT0管脚既支持上升沿中断也支持下降沿中断。 INT0支持掉电唤醒。
P3.3/INT1/ RSTOUT_LOW	8	P3.3	标准I/O口 PORT3[3]
		INT1	外部中断1，既可上升沿中断也可下降沿中断. 如果IT1(TCON.2)被置为1, INT1管脚仅为下降沿中断。如果IT1(TCON.2)被清0，INT1管脚既支持上升沿中断也支持下降沿中断。 INT1支持掉电唤醒。
		RSTOUT_LOW	复位后, 输出低电平，用户可用软件将其设置为高电平或低电平，如果要读外部状态，可将该口先置高后再读
P3.4/RST/T0/ CLKOUT1/$\overline{\text{INT2}}$ /IRC_CLKO	1	P3.4	标准I/O口 PORT3[4]
		RST	复位脚
		T0	定时器/计数器0的外部输入
		CLKOUT1	定时器/计数器1的时钟输出 可通过设置INT_CLKO[1]位/T1CLKO将该管脚配置为CLK-OUT1，也可对T1脚的外部时钟输入进行分频输出
		$\overline{\text{INT2}}$	外部中断2, 只能下降沿中断 支持掉电唤醒
		IRC_CLKO	内部R/C振荡时钟输出;输出的频率可为IRC_CLK/1或IRC_CLK/2
P3.5/T1/CLKOUT0/ $\overline{\text{INT3}}$	3	P3.5	标准I/O口 PORT3[5]
		T1	定时器/计数器1的外部输入
		CLKOUT0	定时器/计数器0的时钟输出 可通过设置INT_CLKO[0]位/T0CLKO将该管脚配置为CLK-OUT0, 也可对T0脚的外部时钟输入进行分频输出
		$\overline{\text{INT3}}$	外部中断3, 只能下降沿中断 支持掉电唤醒
Vcc	2	电源	
Gnd	4	接地	

1.7 STC15系列单片机封装尺寸图

1.7.1 STC15F100系列封装尺寸图

SOP-8 封装尺寸图

8-PIN SMALL OUTLINE PACKAGE (SOP-8)
Dimensions in Inches

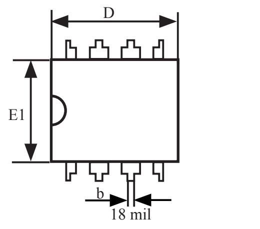


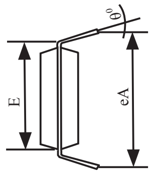
一般尺寸			
(测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	0.053	-	0.069
A1	0.004	-	0.010
b	-	0.016	-
D	0.189	-	0.196
E	0.228	-	0.244
E1	0.150	-	0.157
e	0.050		
L	0.016	-	0.050
L1	0.008		
Φ	0^0	-	8^0

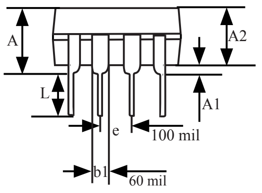
UNIT: INCH, 1 inch = 1000 mil

DIP-8 封装尺寸图

8-Pin Plastic Dual Inline Package (DIP-8)
Dimensions in Inches







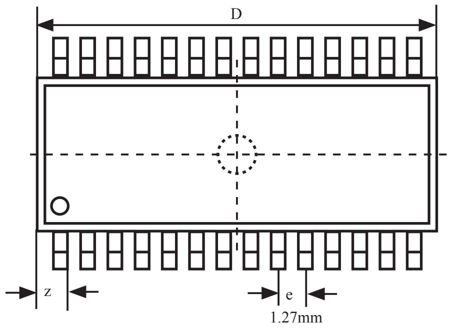
一般尺寸 (测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	-	-	0.210
A1	0.015	-	-
A2	0.125	0.130	0.135
b	-	0.018	-
b1	-	0.060	-
D	0.355	0.365	0.400
E	-	0.300	-
E1	0.245	0.250	0.255
e	-	0.100	-
L	0.115	0.130	0.150
θ^0	0	7	15
eA	0.335	0.355	0.375

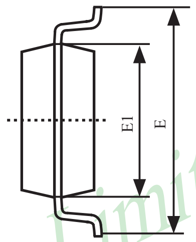
UNIT: INCH, 1 inch = 1000 mil

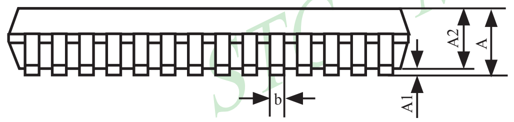
1.7.2 STC15F204EA系列封装尺寸图

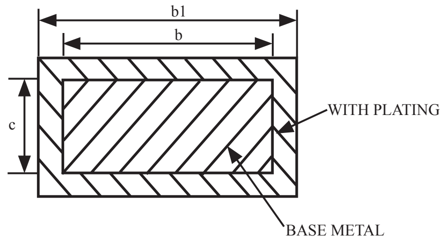
SOP-28 封装尺寸图

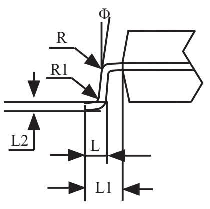
28-Pin Small Outline Package (SOP-28)
Dimensions in Millimeters







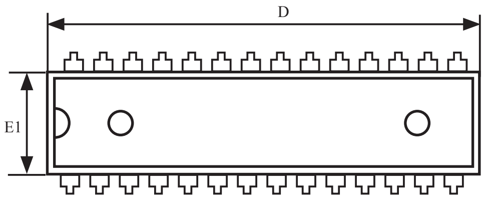


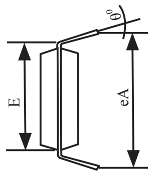


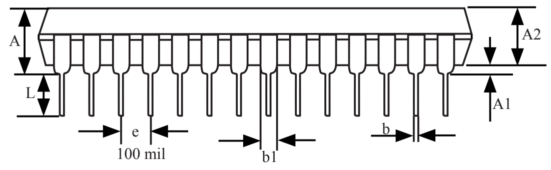
一般尺寸			
(测量单位 = MILLMETER / mm)			
符号	MIN.	NOM.	MAX.
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b	0.356	0.406	0.456
b1	0.366	0.426	0.486
c	-	0.254	-
D	17.750	17.950	18.150
E	10.100	10.300	10.500
E1	7.424	7.500	7.624
e		1.27	
L	0.764	0.864	0.964
L1	1.303	1.403	1.503
L2	-	0.274	-
R	-	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







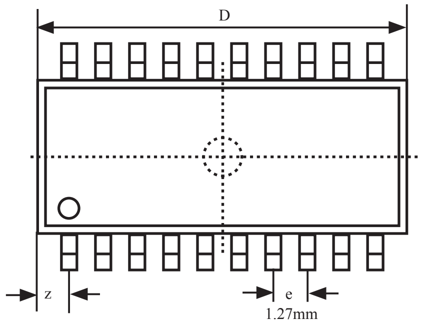
一般尺寸			
(测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	-	-	0.210
A1	0.015	-	-
A2	0.125	0.13	0.135
b	-	0.018	-
b1	-	0.060	-
D	1.385	1.390	1.40
E	-	0.310	-
E1	0.283	0.288	0.293
e	-	0.100	-
L	0.115	0.130	0.150
θ^0	0	7	15
eA	0.330	0.350	0.370

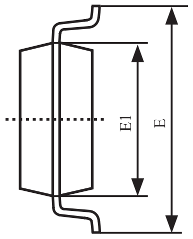
UNIT: INCH, 1 inch = 1000 mil

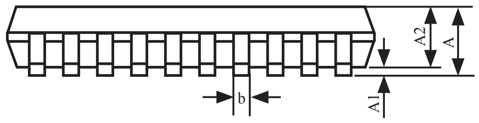
1.7.3 STC15S204EA系列封装尺寸图

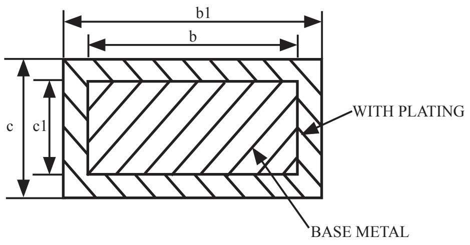
SOP-20 封装尺寸图

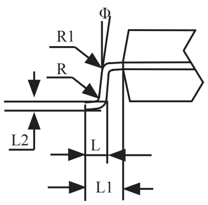
20-Pin Small Outline Package (SOP-20)
Dimensions in Inches and (Millimeters)







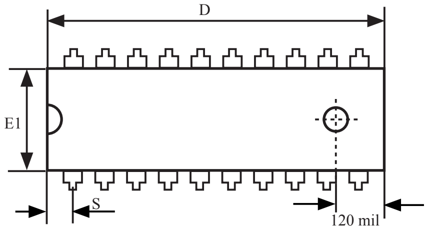


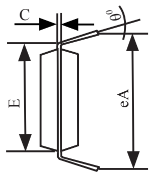


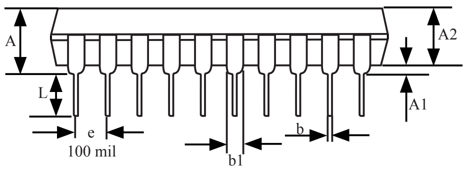
一般尺寸			
(测量单位 = MILLMETER/ mm)			
符号	MIN.	NOM.	MAX.
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b1	0.366	0.426	0.486
b	0.356	0.406	0.456
c	0.234	-	0.274
c1	-	0.254	-
D	12.500	12.700	12.900
E	10.206	10.306	10.406
E1	7.450	7.500	7.550
e		1.27	
L	0.800	0.864	0.900
L1	1.303	1.403	1.503
L2	-	0.274	-
R	-	0.300	-
R1	-	0.200	-
Φ	0^0	-	10^0
z	-	0.660	-

DIP-20 封装尺寸图

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





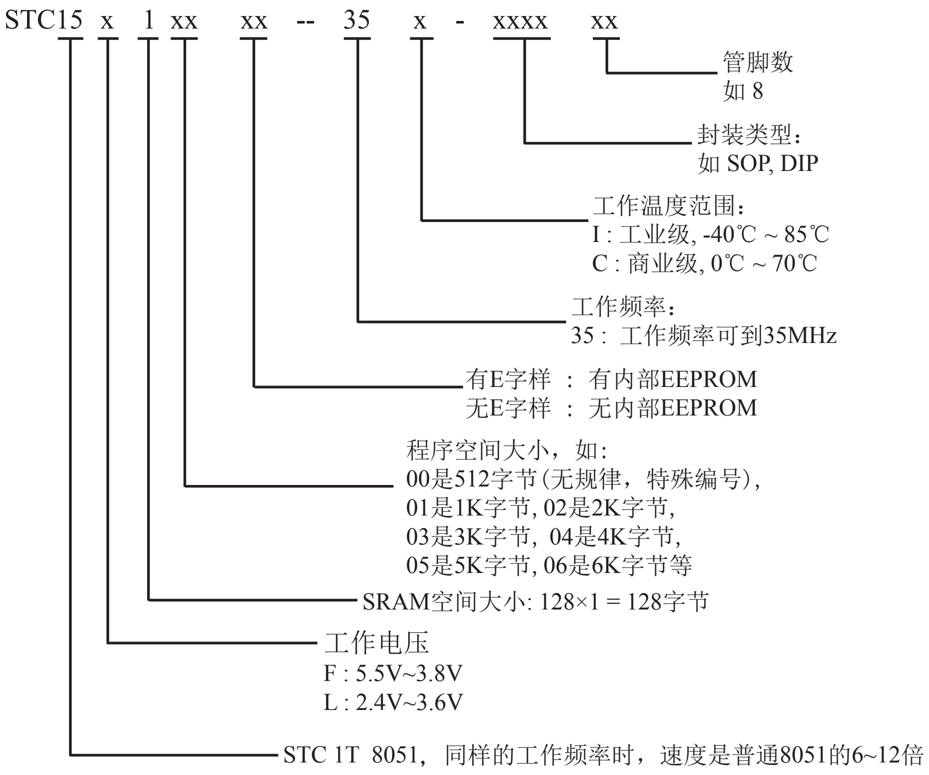


一般尺寸			
(测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	-	-	0.175
A1	0.015	-	-
A2	0.125	0.13	0.135
b	0.016	0.018	0.020
b1	0.058	0.060	0.064
C	0.008	0.010	0.11
D	1.012	1.026	1.040
E	0.290	0.300	0.310
E1	0.245	0.250	0.255
e	0.090	0.100	0.110
L	0.120	0.130	0.140
θ^0	0	-	15
eA	0.355	0.355	0.375
S	-	-	0.075

UNIT: INCH, 1 inch = 1000 mil

1.8 STC15系列单片机命名规则

1.8.1 STC15F100系列单片机命名规则



STC15 x 1 xx xx -- 35 x - xxxx xx

管脚数
如 8

封装类型:
如 SOP, DIP

工作温度范围:
I : 工业级, -40℃ ~ 85℃
C : 商业级, 0℃ ~ 70℃

工作频率:
35 : 工作频率可到35MHz

有E字样 : 有内部EEPROM
无E字样 : 无内部EEPROM

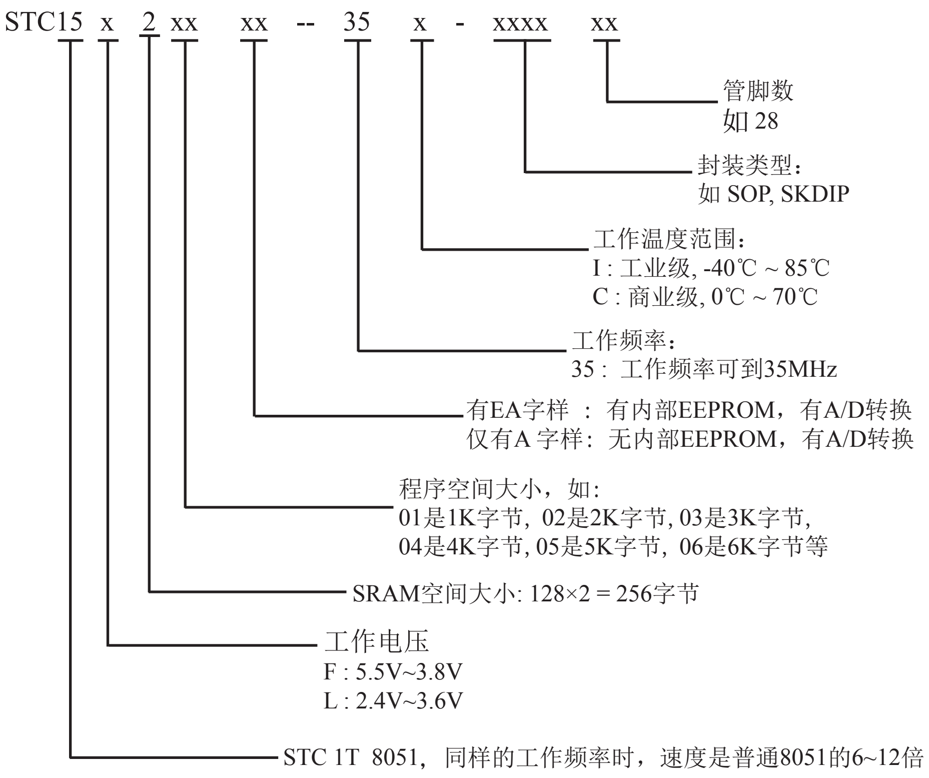
程序空间大小，如:
00是512字节(无规律，特殊编号),
01是1K字节, 02是2K字节,
03是3K字节, 04是4K字节,
05是5K字节, 06是6K字节等

SRAM空间大小: 128×1 = 128字节

工作电压
F : 5.5V~3.8V
L : 2.4V~3.6V

STC 1T 8051，同样的工作频率时，速度是普通8051的6~12倍

1.8.2 STC15F204EA系列单片机命名规则



STC15 x 2 xx xx -- 35 x - xxxx xx

管脚数
如 28

封装类型：
如 SOP, SKDIP

工作温度范围：
I : 工业级, -40℃ ~ 85℃
C : 商业级, 0℃ ~ 70℃

工作频率：
35 : 工作频率可到35MHz

有EA字样 ：有内部EEPROM，有A/D转换
仅有A 字样：无内部EEPROM，有A/D转换

程序空间大小，如：
01是1K字节，02是2K字节, 03是3K字节,
04是4K字节, 05是5K字节，06是6K字节等

SRAM空间大小: 128×2 = 256字节

工作电压
F : 5.5V~3.8V
L : 2.4V~3.6V

STC 1T 8051，同样的工作频率时，速度是普通8051的6~12倍

第2章 STC15F100系列的时钟，省电模式及复位

2.1 STC15F100系列单片机的内部时钟

STC15F100系列单片机只有一个时钟源 — 内部高精度R/C时钟，±1%温飘(-40^{0}C~+85^{0}C)，常温下温飘5‰

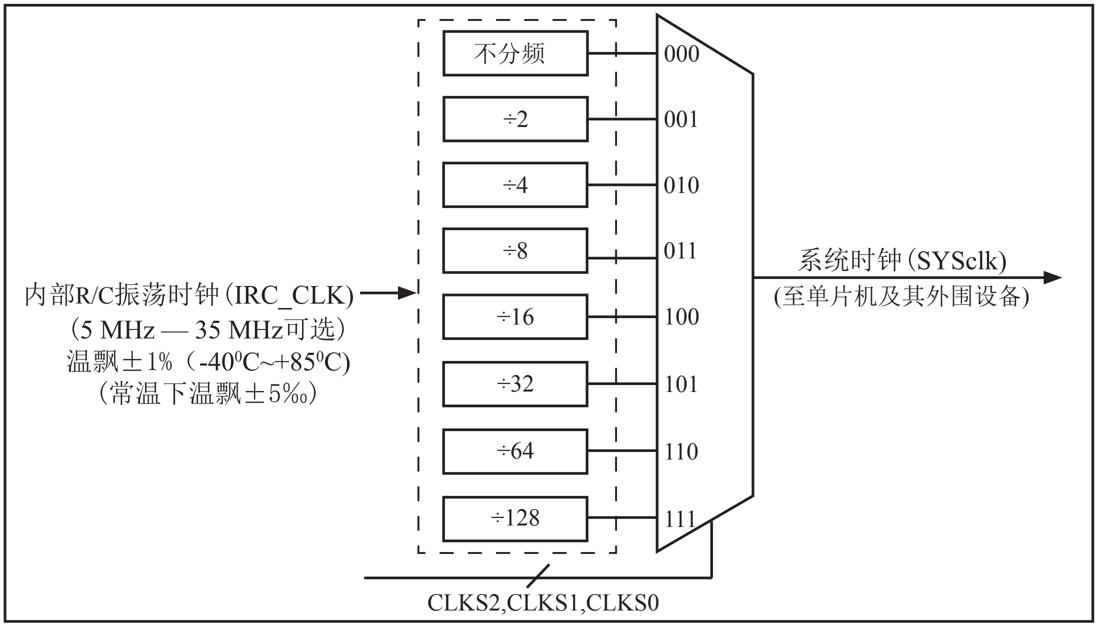
2.1.1 内部可选时钟及时钟分频和分频寄存器

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	97H	name	-	-	-	-	-	CLKS2	CLKS1	CLKS0

CLKS2	CLKS1	CLKS0	分频后CPU的实际工作时钟
0	0	0	内部R/C振荡时钟/1, 不分频
0	0	1	内部R/C振荡时钟/2
0	1	0	内部R/C振荡时钟/4
0	1	1	内部R/C振荡时钟/8
1	0	0	内部R/C振荡时钟/16
1	0	1	内部R/C振荡时钟/32
1	1	0	内部R/C振荡时钟/64
1	1	1	内部R/C振荡时钟/128



时钟结构

2.1.2 可编程时钟输出

STC15F100系列单片机有三种可编程时钟输出：IRC_CLKO/P3.4，CLKOUT0/P3.5，CLK-OUT1/P3.4. 只有内部R/C时钟频率为12MHz以下时，现版本的IRC_CLKO/P3.4才能正常输出。

IRC_CLKO : Internal R/C clock output register

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IRC_CLKO	BBH	name	EN_IRCO	-	-	-	DIVIRCO	-	-	-

如何利用IRC_CLKO/P3.4管脚输出时钟

IRC_CLKO/P0.0的时钟输出控制由IRC_CLKO寄存器的EN_IRCO位控制。设置EN_IRCO (IRC_CLKO.7)可将IRC_CLKO/P3.4管脚配置为内部R/C振荡时钟输出。通过设置DIVIRCO (IRC_CLKO.3)位可以设置内部R/C振荡时钟的输出频率是IRC_CLK/2还是IRC_CLK/1(不分频)

新增加的特殊功能寄存器：IRC_CLKO（地址：0xBB）

B7 - EN_IRCO ：

1，将IRC_CLKO/P3.4管脚配置为内部R/C振荡时钟输出

0，不允许IRC_CLKO/P3.4管脚配置为内部R/C振荡时钟输出

B3 – DIVIRCO ：

1，内部R/C振荡时钟的输出频率被2分频，输出时钟频率 ＝IRC_CLK/2

0，内部R/C振荡时钟的输出频率不被分频，输出时钟频率 ＝IRC_CLK/1

IRC_CLKO指内部R/C振荡时钟输出；IRC_CLK指内部R/C振荡时钟频率。

INT_CLKO : External Interrupt Enable and Clock Output register

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
INT_CLKO	8FH	name	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO

如何利用CLKOUT0/P3.5和CLKOUT1/P3.4管脚输出时钟

CLKOUT0/P3.5管脚是否输出时钟由INT_CLKO寄存器的T0CLKO位控制

B0 - T0CLKO : 1, 允许时钟输出

0, 禁止时钟输出

CLKOUT1/P3.4管脚是否输出时钟由INT_CLKO寄存器的T1CLKO位控制

B1 - T1CLKO : 1, 允许时钟输出

0, 禁止时钟输出

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0(16位自动重装模式)或模式2(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：INT_CLKO（地址：0x8F）

B6 - EX4 : 允许外部中断4($\overline{\text{INT4}}$)

B5 - EX3 : 允许外部中断3($\overline{\text{INT3}}$)

B4 - EX2 : 允许外部中断2($\overline{\text{INT2}}$)

B1 – T1CLKO ：

1，将P3.4/T0管脚配置为定时器1的时钟输出CLKOUT1，输出时钟频率= T1溢出率/2

若定时器/计数器T1工作在定时器模式0(16位自动重装模式)，

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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (65536-[RL_TH1, RL_TL1])/2

T1工作在12T模式(AUXR.6/T1x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH1, RL_TL1])/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) / (65536-[RL_TH1, RL_TL1])/2

若定时器/计数器T1工作在模式2(8位自动重装模式)，

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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (256-TH1)/2

T1工作在12T模式(AUXR.6/T1x12=0)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P3.4/T0管脚被配置为定时器1的时钟输出

B0 – T0CLKO ：

1，将P3.5/T1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

若定时器/计数器T0工作在定时器模式0(16位自动重装模式)时，

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(AUXR.7/T0x12=1)时的输出频率 = (SYSclk)/(65536-[RL_TH0, RL_TL0])/2

T0工作在12T模式(AUXR.7/T0x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH0, RL_TL0])/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) / (65536-[RL_TH0, RL_TL0])/2

若定时器/计数器T0工作在定时器模式2(8位自动重装模式)，如果C/$\overline{T}$=0，则：

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

T0工作在12T模式(AUXR.7/T0x12=0)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P3.5/T1管脚被配置为定时器0的时钟输出

AUXR : Auxiliary register

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

AUXR (地址：0x8E)

T0x12:

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:

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

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的C语言声明：

```
sfr     IRC_CLKO       =   0xBB;        //新增加的特殊功能寄存器IRC_CLKO的地址声明
sfr     INT_CLKO       =   0x8F;        //新增加的特殊功能寄存器INT_CLKO的地址声明
sfr     AUXR           =   0x8E;        //特殊功能寄存器AUXR的地址声明
```

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的汇编语言声明：

```
IRC_CLKO      EQU    0BBH          ;新增加的特殊功能寄存器IRC_CLKO的地址声明
INT_CLKO      EQU    8FH           ;新增加的特殊功能寄存器INT_CLKO的地址声明
AUXR          EQU    8EH           ;特殊功能寄存器AUXR的地址声明
```

2.2 STC15F100系列单片机的省电模式

STC15F100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：低速模式，空闲模式和掉电模式。正常工作模式下，STC15F100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)

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

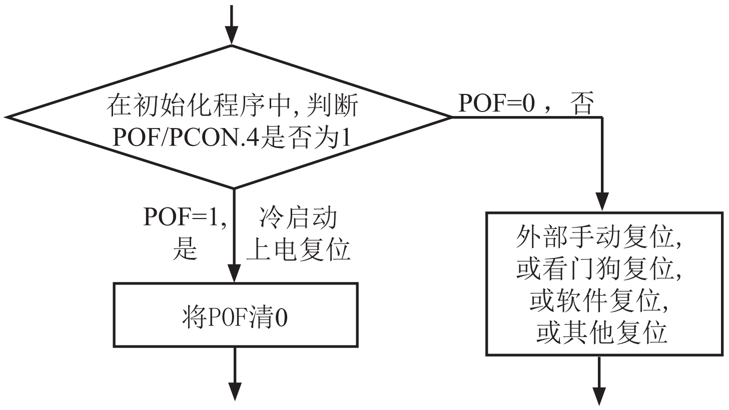
LVDF：低压检测标志位, 同时也是低压检测中断请求标志位。

在正常工作和空闲工作状态时，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。

在进入掉电工作状态前，如果低压检测电路未被允许可产生中断，则在进入掉电模式后，该低压检测电路不工作以降低功耗。如果被允许可产生低压检测中断，则在进入掉电模式后，该低压检测电路继续工作，在内部工作电压Vcc低于低压检测门槛电压后，产生低压检测中断，可将MCU从掉电状态唤醒。

POF：上电复位标志位，单片机停电后再上电，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），还是其他复位(外部复位脚输入复位信号产生的复位、内部看门狗复位、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，可由外部中断上升沿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，所以CPU、定时器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：INT0/P3.2, INT1/P3.3, $\overline{\text{INT2}}$/P3.4, $\overline{\text{INT3}}$/P3.5, $\overline{\text{INT4}}$/P3.0 。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。

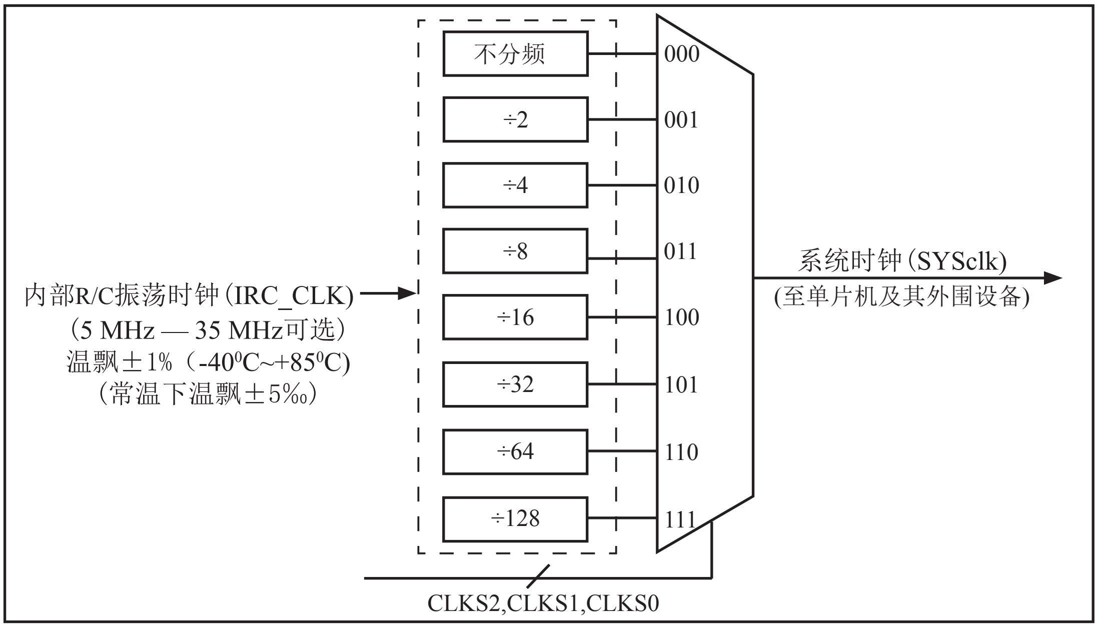
GF1,GF0：两个通用工作标志位, 用户可以任意使用。

2.2.1 低速模式

时钟分频器可以对内部时钟进行分频，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	97H	name	-	-	-	-	-	CLKS2	CLKS1	CLKS0

CLKS2	CLKS1	CLKS0	分频后CPU的实际工作时钟
0	0	0	内部R/C振荡时钟/1，不分频
0	0	1	内部R/C振荡时钟/2
0	1	0	内部R/C振荡时钟/4
0	1	1	内部R/C振荡时钟/8
1	0	0	内部R/C振荡时钟/16
1	0	1	内部R/C振荡时钟/32
1	1	0	内部R/C振荡时钟/64
1	1	1	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、内部低压检测电路、定时器等仍正常运行。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。

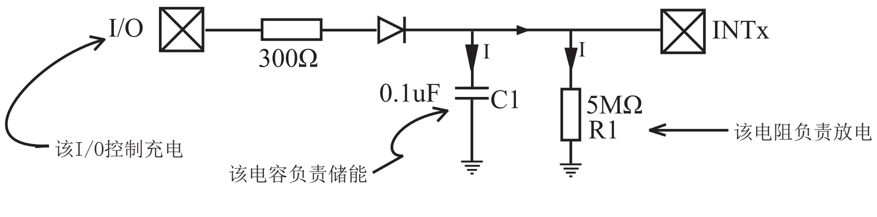
2.2.3 掉电模式/停机模式

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：INT0/P3.2，INT1/P3.3，$\overline{\text{INT2}}$/P3.4，$\overline{\text{INT3}}$/P3.5，$\overline{\text{INT4}}$/P3.0

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。

```
/*可由外部中断唤醒的掉电唤醒示例程序, ------------------------------------------- */
/*此程序只是一个示例程序，并不是针对STC15F100系列单片机来编写的-----*/
/*用户可以对此程序进行适当的修改，-----------------------------------------------*/
/*例如减少I/O口的使用，以至于本程序能适用STC15F100系列单片机 ---------*/
/*---------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- 演示STC 15 系列单片机由外部中断唤醒的掉电唤醒演示程序  --------------*/
/*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宏晶科技的资料及程序 ---------------------------*/
/*---------------------------------------------------------------------------------------------*/


#include <reg51.h>
#include <intrins.h>
sbit        Begin_LED = P3^1;                              //Begin-LED indicator indicates system start-up
unsigned  char        Is_Power_Down = 0;                   //Set this bit before go into Power-down mode
sbit        Is_Power_Down_LED_INT0             = P1^7;    //Power-Down wake-up LED indicator on INT0
sbit        Not_Power_Down_LED_INT0            = P1^6;    //Not  Power-Down wake-up LED indicator on INT0
sbit        Is_Power_Down_LED_INT1             = P1^5;    //Power-Down wake-up LED indicator on INT1
sbit        Not_Power_Down_LED_INT1            = P3^4;    //Not  Power-Down wake-up LED indicator on INT1
sbit        Power_Down_Wakeup_Pin_INT0         = P3^2;    //Power-Down wake-up pin on INT0
sbit        Power_Down_Wakeup_Pin_INT1         = P3^3;    //Power-Down wake-up pin on INT1
sbit        Normal_Work_Flashing_LED           = P1^3;    //Normal work LED indicator
void  Normal_Work_Flashing (void);
void  INT_System_init (void);
void  INT0_Routine (void);
void  INT1_Routine (void);

void main (void)
{
            unsigned  char        j = 0;
            unsigned  char        wakeup_counter = 0;
                                              //clear interrupt wakeup counter variable wakeup_counter
            Begin_LED = 0;                    //system start-up LED
            INT_System_init ( );              //Interrupt system initialization
            while(1)
```

```
        {
                P2 = wakeup_counter;
                wakeup_counter++;
                for(j=0; j<2; j++)
                {
                        Normal_Work_Flashing( );    //System normal work
                }
                Is_Power_Down = 1;                  //Set this bit before go into Power-down mode
                PCON    = 0x02;             //after this instruction, MCU will be in power-down mode
                                            //external clock stop
                _nop_( );
                _nop_( );
                _nop_( );
                _nop_( );
        }
}
void  INT_System_init (void)
{
        IT0     = 0;                        /* External interrupt 0, low electrical level triggered */
//      IT0     = 1;                        /* External interrupt 0, negative edge triggered */
        EX0     = 1;                        /* Enable external interrupt 0
        IT1     = 0;                        /* External interrupt 1, low electrical level triggered */
//      IT1     = 1;                        /* External interrupt 1, negative edge triggered */
        EX1     = 1;                        /* Enable external interrupt 1
        EA      = 1;                        /* Set Global Enable bit
}
void  INT0_Routine (void) interrupt 0
{
        if (Is_Power_Down)
        {
                //Is_Power_Down ==1;        /* Power-Down wakeup on INT0 */
                Is_Power_Down = 0;
                Is_Power_Down_LED_INT0 = 0;
                                /*open external interrupt 0 Power-Down wake-up LED indicator */
                while (Power_Down_Wakeup_Pin_INT0 == 0)
                {
                        /* wait higher */
                }
                Is_Power_Down_LED_INT0 = 1;
                                /* close external interrupt 0 Power-Down wake-up LED indicator */
        }
        else
        {
                Not_Power_Down_LED_INT0 = 0;     /* open external interrupt 0 normal work LED */
                while (Power_Down_Wakeup_Pin_INT0 ==0)
                {
                        /* wait higher */
                }
                Not_Power_Down_LED_INT0 = 1;     /* close external interrupt 0 normal work LED */
        }
}

void  INT1_Routine (void)   interrupt 2
{
        if (Is_Power_Down)
        {
                //Is_Power_Down ==1;          /* Power-Down wakeup on INT1 */
                Is_Power_Down = 0;
                Is_Power_Down_LED_INT1= 0;
                                /*open external interrupt 1 Power-Down wake-up LED indicator */
                while (Power_Down_Wakeup_Pin_INT1 == 0)
                {
                        /* wait higher */
                }
                Is_Power_Down_LED_INT1 = 1;
                                /* close external interrupt 1 Power-Down wake-up LED indicator */
        }
        else
        {
                Not_Power_Down_LED_INT1 = 0;     /* open external interrupt 1 normal work LED */
                while (Power_Down_Wakeup_Pin_INT1 ==0)
                {
                        /* wait higher */
                }
                Not_Power_Down_LED_INT1 = 1;     /* close external interrupt 1 normal work LED */
        }
}

void  delay (void)
{
        unsigned  int          j = 0x00;
        unsigned  int          k = 0x00;
        for (k=0; k<2; ++k)
        {
                for (j=0; j<=30000; ++j)
                {
                        _nop_( );
                        _nop_( );
                        _nop_( );
                        _nop_( );
                        _nop_( );
                        _nop_( );
                        _nop_( );
                        _nop_( );
                }
        }
}
void Normal_Work_Flashing (void)
{
        Normal_Work_Flashing_LED = 0;
        delay ( );
        Normal_Work_Flashing_LED = 1;
        delay ( );
}
```

The following program also demostrates that power-down mode or idle mode be woken-up by external interrupt, but is written in assembly language rather than C languge.

```
;*********************************************************
;Wake Up Idle and Wake Up Power Down
;*************************************************************
;/* --- STC MCU International Limited ---------------------------------------------------*/
;/* --- 演示STC 15 系列单片机由外部中断唤醒的掉电唤醒演示程序  --------------*/
;/*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宏晶科技的资料及程序 ---------------------------*/
;/*---------------------------------------------------------------------------------------------------*/



                ORG     0000H
                AJMP    MAIN
                ORG     0003H
int0_interrupt:
                CLR     P3.1                    ;open P3.1 LED indicator
                ACALL   delay                   ;delay in order to observe
                CLR     EA                      ;clear global enable bit, stop all interrupts
                RETI
                ORG     0013H
int1_interrupt:
                CLR     P3.5                    ;open P3.5 LED indicator
                ACALL   delay                   ;;delay in order to observe
                CLR     EA                      ;clear global enable bit, stop all interrupts
                RETI
                ORG     0100H
delay:
                CLR     A
                MOV     R0,     A
                MOV     R1,     A
                MOV     R2,     #02
```

```
delay_loop:
                DJNZ    R0,     delay_loop
                DJNZ    R1,     delay_loop
                DJNZ    R2,     delay_loop
                RET

main:
                MOV     R3,     #0              ;P3 LED increment mode changed
                                                ;start to run program
main_loop:
                MOV     A,      R3
                CPL     A
                MOV     P1,     A
                ACALL   delay
                INC     R3
                MOV     A,      R3
                SUBB    A,      #18H
                JC      main_loop
                MOV     P3,     #0FFH           ;close all LED, MCU go into power-down mode
                CLR     IT0                     ;low electrical level trigger external interrupt 0
        ;       SETB    IT0                     ;negative edge trigger external interrupt 0
                SETB    EX0                     ;enable external interrupt 0
                CLR     IT1                     ;low electrical level trigger external interrupt 1
        ;       SETB    IT1                     ;negative edge trigger external interrupt 1
                SETB    EX1                     ;enable external interrupt 1
                SETB    EA                      ;set the global enable
                                                ;if don't so, power-down mode cannot be wake up

;MCU will go into idle mode or power-down mode after the following instructions
                MOV     PCON,   #00000010B      ;Set PD bit, power-down mode (PD = PCON.1)
        ;       NOP
        ;       NOP
        ;       NOP
        ;       NOP
        ;       NOP
        ;       MOV     PCON,   #00000001B      ;Set IDL bit, idle mode (IDL = PCON.0)
        ;       NOP
        ;       NOP
        ;       NOP
                MOV     P1,     #0DFH           ;1101,1111
                NOP
                NOP
                NOP
                NOP
                NOP
WAIT1:
                SJMP    $                       ;dynamically stop
                END
```

2.3 复位

STC15F100系列单片机有6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。P0.0/RST管脚出厂时被配置为I/O口，要将其配置为复位管脚，可在ISP烧录程序时设置。如果P0.0/RST管脚已在ISP烧录程序时被设置为复位脚，那P0.0/RST就是芯片复位的输入脚。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 就可以实现系统复位了。

IAP_CONTR: ISP/IAP 控制寄存器

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CONTR	C7H	name	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0

IAPEN: ISP/IAP功能允许位。0：禁止IAP读/写/擦除Data Flash/EEPROM
1：允许IAP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IAP_TRIG送5Ah/A5h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 IAP_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 IAP_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 IAP_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 IAP_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

当电源电压VCC低于上电复位检测门槛电压时，所有的逻辑电路都会复位。当内部VCC上升至上电复位检测门槛电压以上后，延迟8192个时钟，上电复位结束。

2.3.4 MAX810专用复位电路复位

STC15F100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45mS复位延时，复位才能被解除。

2.3.5 内部低压检测复位

除了上电复位检测门槛电压外，STC15F100单片机还有一组更可靠的内部低压检测门槛电压。当电源电压VCC低于内部低压检测(LVD)门槛电压时，可产生复位(前提是在STC-ISP编程/烧录用户程序时,允许低压检测复位，即将低压检测门槛电压设置为复位门槛电压)。

STC15F100单片机内置了8级可选内部低压检测门槛电压。下表列出了不同温度下STC15F/L100系列单片机所有的低压检测门槛电压。

5V单片机的低压检测门槛电压：

-40 ^{0}C	25 ^{0}C	85 ^{0}C
4.74	4.64	4.60
4.41	4.32	4.27
4.14	4.05	4.00
3.90	3.82	3.77
3.69	3.61	3.56
3.51	3.43	3.38
3.36	3.28	3.23
3.21	3.14	3.09

如果用户所使用的是STC15F100系列5V单片机，那么用户可以根据单片机的实际工频率在STC-ISP编程器中选择上表中所列出的低压检测门槛电压作为复位门槛电压。如：常温下工作频率是20MHz以上时，可以选择4.32V电压作为复位门槛电压；常温下工作频率是12MHz以下时，可以选择3.82V电压作为复位门槛电压。

3.3V单片机的低压检测门槛电压：

-40 ^{0}C	25 ^{0}C	85 ^{0}C
3.11	3.08	3.09
2.85	2.82	2.83
2.63	2.61	2.61
2.44	2.42	2.43
2.29	2.26	2.26
2.14	2.12	2.12
2.01	2.00	2.00
1.90	1.89	1.89

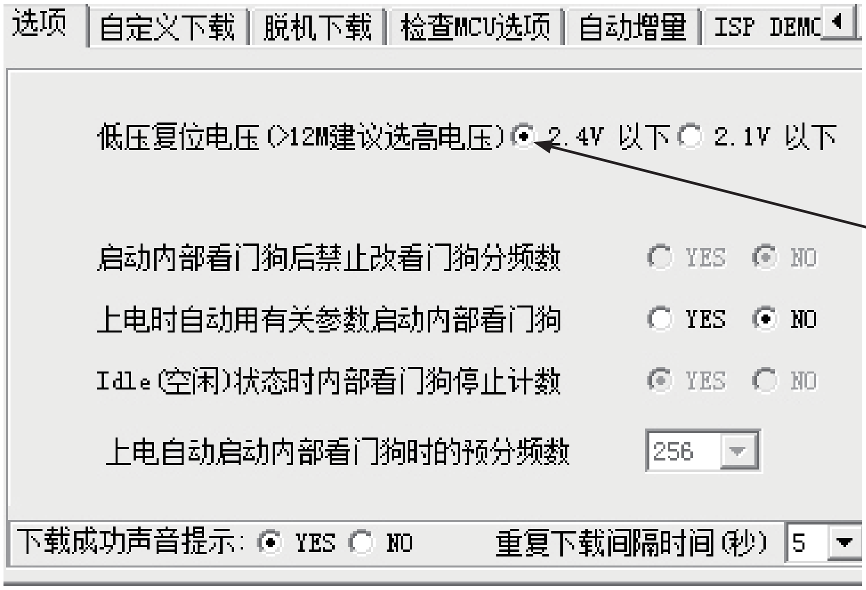
如果用户所使用的是STC15L100系列3.3V单片机，那么用户可以根据单片机的实际工作频率在STC-ISP编程器中选择上表中所列出的低压检测门槛电压作为复位门槛电压。如：常温下工作频率是20MHz以上时，可以选择2.82V电压作为内部低压检测复位门槛电压；常温下工作频率是12MHz以下时，可以选择2.42V电压作为复位门槛电压。

如果在STC-ISP编程/烧录用户应用程序时，不将低压检测设置为低压检测复位，则在用户程序中用户可将低压检测设置为低压检测中断。当电源电压VCC低于内部低压检测(LVD)门槛电压时，低压检测中断请求标志位(LVDF/PCON.5)就会被硬件置位。如果ELVD/IE.6(低压检测中断允许位)被设置为1，低压检测中断请求标志位就能产生一个低压检测中断。

在正常工作和空闲工作状态时，如果内部工作电压Vcc低于低压检测门槛电压，低压中断请求标志位(LVDF/PCON.5)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，LVDF/PCON.5都自动为1。该位要用软件清0，清0后，如内部工作电压Vcc低于低压检测门槛电压，该位又被自动设置为1。

在进入掉电工作状态前，如果低压检测电路未被允许可产生中断，则在进入掉电模式后，该低压检测电路不工作以降低功耗。如果被允许可产生低压检测中断(相应的中断允许位是ELVD/IE.6，中断请求标志位是LVDF/PCON.5)，则在进入掉电模式后，该低压检测电路继续工作，在内部工作电压Vcc低于低压检测门槛电压后，产生低压检测中断，可将MCU从掉电状态唤醒。

低压检测电路的一个重要应用：建议在电压偏低时，不要操作EEPROM/IAP



低压复位电压(>12M建议选高电压) [✓ 2.4V 以下] 2.1V 以下

启动内部看门狗后禁止改看门狗分频数 YES [✓ NO]

上电时自动用有关参数启动内部看门狗 YES [✓ NO]

Idle(空闲)状态时内部看门狗停止计数 [✓ YES] NO

上电自动启动内部看门狗时的预分频数 256

下载成功声音提示: [✓ YES] NO 重复下载间隔时间(秒) 5

STC15F100系列3V单片机复位门槛电压选择

此图与STC15系列单片机不符，需要更新，在此画出，只是为了让读者对低压检测门槛电压的使用有个形象直观地理解

与低压检测相关的一些寄存器：

PCON : 电源控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	-	-	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。

在正常工作和空闲工作状态时，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。

在进入掉电工作状态前，如果低压检测电路未被允许可产生中断，则在进入掉电模式后，该低压检测电路不工作以降低功耗。如果被允许可产生低压检测中断，则在进入掉电模式后，该低压检测电路继续工作，在内部工作电压Vcc低于低压检测门槛电压后，产生低压检测中断，可将MCU从掉电状态唤醒。

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

PD : 掉电模式控制位

IDL : 空闲模式控制位

GF1,GF0 : 两个通用工作标志位, 用户可以任意使用。

IE : 中断允许寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	ELVD	-	-	ET1	EX1	ET0	EX0

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

ELVD : 低压检测中断允许位
ELVD = 0, 禁止低压检测中断
ELVD = 1, 允许低压检测中断

IP : 中断优先级控制寄存器

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

PLVD : 低压检测中断优先级控制位
PLVD = 0, 低压检测中断位低优先级
PLVD = 1, 低压检测中断为高优先级

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。STC15F100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	0C1H	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，在宏晶的下载板上可以直接测试

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/


;本演示程序在STC-ISP Ver 3.0A.PCB的下载编程工具上测试通过,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    0C1H ;看门狗地址
WDT_TIME_LED      EQU    P3.5 ;用 P3.5 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    P3.4
                 ;用 P3.4 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

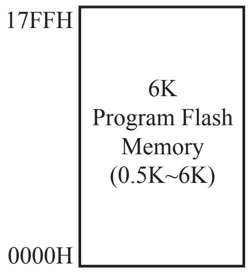
	复位源	现象
热启动复位	内部看门狗复位	会使单片机直接从用户程序区0000H处开始执行用户程序
	通过控制RESET脚产生的硬复位	会使系统 从用户程序区0000H处开始直接执行用户程序
	通过对IAP_CONTR寄存器送入20H产生的软复位	会使系统从用户程序区0000H处开始直接执行用户程序
	通过对IAP_CONTR寄存器送入60H产生的软复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用户程序区执行用户程序
冷启动复位	系统停电后再上电引起的硬复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用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

STC15F100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。STC15F100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，因为没有访问外部程序存储器的总线。STC15F100系列单片机内部有128字节的数据存储器。

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。STC15F100系列单片机内部集成了0.5K~6K字节的Flash程序存储器。STC15F100系列各种型号单片机的程序Flash存储器的地址如下表所示。



IAP15F106单片机程序存储器

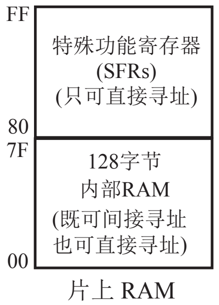
Type	Program Memory
STCF/L100	0000H~01FFH (512 Byte)
STC15F/L101	0000H~03FFH (1K)
STC15F/L101E	
STC15F/L102	0000H~07FFH (2K)
STC15F/L102E	
STC15F/L103	0000H~0BFFH (3K)
STC15F/L103E	
STC15F/L104	0000H~0FFFH (4K)
STC15F/L104E	
STC15F/L105	0000H~13FFH (5K)
STC15F/L105E	
IAP15F/L106	0000H~17FFH (6K)

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。

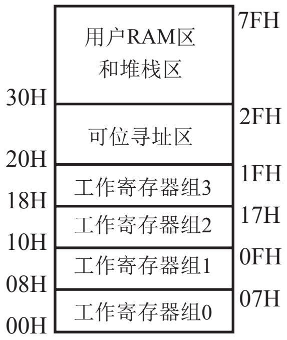
程序Flash存储器可在线反复编程擦写10万次以上，提高了使用的灵活性和方便性。

3.2 数据存储器(SRAM)

STC15F100系列单片机内部集成了128字节RAM，可用于存放程序执行的中间结果和过程数据。内部RAM的结构如下图所示，地址范围是00H~7FH。



片上 RAM



128字节的SRAM

内部数据存储器可分为2个部分：**128字节RAM(与传统8051兼容)**及特殊功能寄存器区。128字节的数据存储器既可直接寻址也可间接寻址。特殊功能寄存器区只可直接寻址。

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。见下节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60H以后的单元为宜。

PSW：程序状态字寄存器

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

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

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60H或更大的值为宜。STC15F100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。

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。STC15F100系列单片机内的特殊功能寄存器(SFR)的地址范围为80H~FFH, 特殊功能寄存器(SFR)必须用直接寻址指令访问。

STC15F100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									0FFH
0F0H	B 0000,0000								0F7H
0E8H									0EFH
0E0H	ACC 0000,0000								0E7H
0D8H									0DFH
0D0H	PSW 0000,00x0								0D7H
0C8H									0CFH
0C0H		WDT_CONR 0x00,0000	IAP_DATA 1111,1111	IAP_ADDRH 0000,0000	IAP_ADDRL 0000,0000	IAP_CMD xxxx,xx00	IAP_TRIG xxxx,xxxx	IAP_CONTR 0000,0000	0C7H
0B8H	IP x0xx,0000			IRC_CLKO 0xxx,0xxxx					0BFH
0B0H	P3 xx11,1111	P3M1 xx00,0000	P3M0 xx00,0000						0B7H
0A8H	IE 00xx,0000								0AFH
0A0H								Don't use	0A7H
098H							Don't use	Don't use	09FH
090H								CLK_DIV xxxx,x000	097H
088H	TCON 0000,0000	TMOD 0000,0000	TL0 0000,0000	TL1 0000,0000	TH0 0000,0000	TH1 0000,0000	AUXR 00xx,xxxx	INT_CLKO x000,xx00	08FH
080H		SP 0000,0111	DPL 0000,0000	DPH 0000,0000				PCON xx11,0000	087H
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

不可位寻址

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								复位值
			MSB							LSB	
SP	堆栈指针	81H									0000 0111B
DPTR DPL	数据指针(低)	82H									0000 0000B
DPH	数据指针(高)	83H									0000 0000B
PCON	电源控制寄存器	87H	-	-	LVDF	POF	GF1	GF0	PD	IDL	xx11 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	-	-	-	-	-	-	00xx xxxxB
INT_CLKO	外部中断允许和时钟输出寄存器	8FH	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO	x000 xx00B
CLK_DIV	时钟分频寄存器	97h	-	-	-	-	-	CLKS2	CLKS1	CLKS0	xxxx x000B
IE	中断允许寄存器	A8H	EA	ELVD	-	-	ET1	EX1	ET0	EX0	00xx 0000B
P3	Port 3	B0H	-	-	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	xx11 1111B
P3M1	P3口模式配置寄存器1	B1H									xx00 0000B
P3M0	P3口模式配置寄存器0	B2H									xx00 0000B
IP	中断优先级寄存器	B8H	-	PLVD	-	-	PT1	PX1	PT0	PX0	x0xx 0000B
IRC_CLKO	内部R/C时钟输出寄存器	BBH	EN_IRCO	-	-	-	DIVIRCO	-	-	-	0xxx,0xxxB
WDT_CONTR	看门狗控制寄存器	C1H	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	0x00 0000B
IAP_DATA	ISP/IAP 数据寄存器	C2H									1111 1111B
IAP_ADDRH	ISP/IAP 高8位地址寄存器	C3H									0000 0000B
IAP_ADDRL	ISP/IAP 低8位地址寄存器	C4H									0000 0000B
IAP_CMD	ISP/IAP 命令寄存器	C5H	-	-	-	-	-	-	MS1	MS0	xxxx xx00B
IAP_TRIG	ISP/IAP 命令触发寄存器	C6H									xxxx xxxxB
IAP_CONTR	ISP/IAP控制寄存器	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 x000B
PSW	程序状态字寄存器	D0H	CY	AC	F0	RS1	RS0	OV	-	P	0000 00x0B
ACC	累加器	E0H									0000 0000B
B	B寄存器	F0H									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	-	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。

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.

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60H或更大的值为宜。STC15F100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 STC15F100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

STC15F100系列单片机共有6个I/O口：P3.0~P3.5。其所有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。STC15F100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。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70mA。

I/O口工作类型设定

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

P3M1[5 : 0]	P3M0 [5 : 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270μA , 由于制造误差， 实际为270uA～ 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    P3M1, #xx101000B
        MOV    P3M0, #xx110000B
  ;P3.5为开漏,P3.4为强推挽输出,P3.3为高阻输入,P3.2/P3.1/P3.0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70mA。即从MCU-VCC流入的电流不超过70mA,从MCU-Gnd流出电流不超过70mA,整体流入/流出电流都不能超过70mA.

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	-	-	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0

P3M1 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M1	B1H	name	-	-	P3M1.5	P3M1.4	P3M1.3	P3M1.2	P3M1.1	P3M1.0

P3M0 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M0	B2H	name	-	-	P3M0.5	P3M0.4	P3M0.3	P3M0.2	P3M0.1	P3M0.0

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

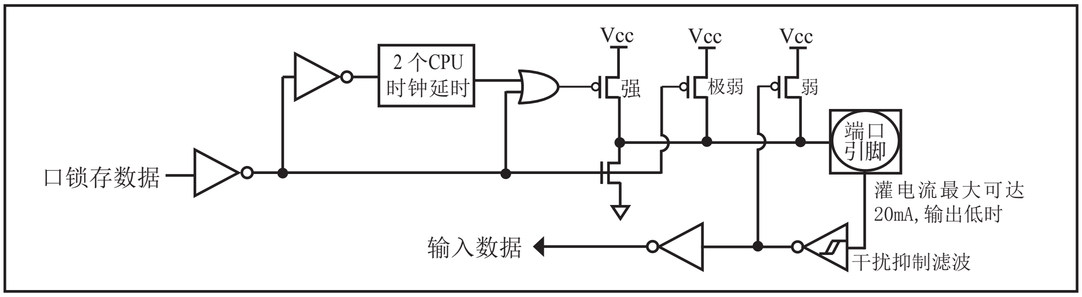
准双向口输出类型可用作输出和输入功能而不需重新配置口线输出状态。这是因为当口线输出为1时驱动能力很弱，允许外部装置将其拉低。当引脚输出为低时，它的驱动能力很强，可吸收相当大的电流。准双向口有3个上拉晶体管适应不同的需要。

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

STC15F100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。

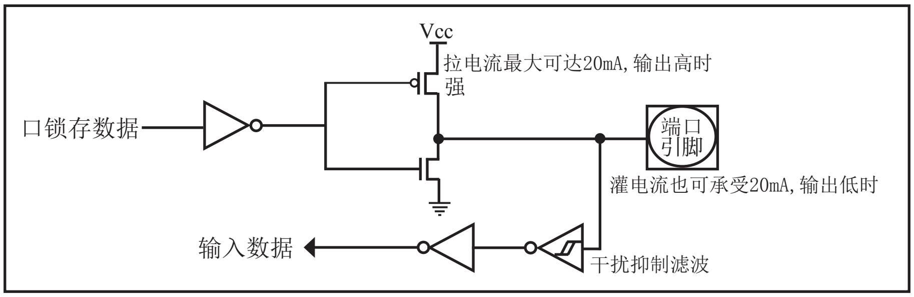
准双向口带有一个施密特触发输入以及一个干扰抑制电路。

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。

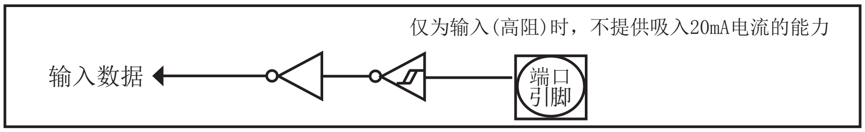
强推挽引脚配置如下图所示。



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。



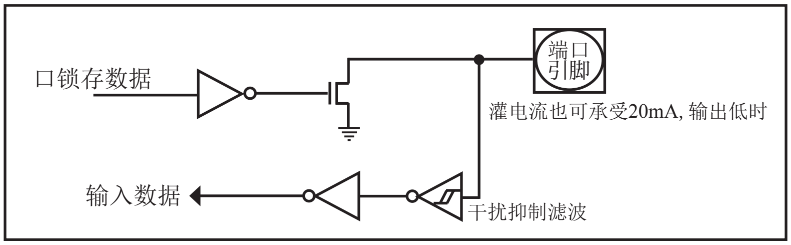
仅为输入(高阻)模式

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

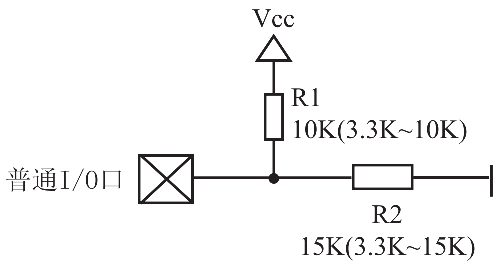
有些是外围接的是NPN三极管,没有加上拉电阻,其实基极串多大电阻,I/O口就应该上拉多大的电阻,或者将该I/O口设置为强推挽输出.

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

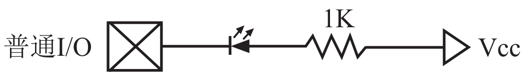
发现有些是做行列矩阵按键扫描电路时,实际工作时没有加限流电阻,实际工作时可能出现2个I/O口均输出为低,并且在按键按下时,短接在一起,我们知道一个CMOS电路的2个输出脚不应该直接短接在一起,按键扫描电路中,此时一个口为了读另外一个口的状态,必须先置高才能读另外一个口的状态,而8051单片机的弱上拉口在由0变为1时,会有2个时钟的强推挽高输出电流,输出到另外一个输出为低的I/O口,就有可能造成I/O口损坏.建议在其中的一侧加1K限流电阻,或者在软件处理上,不要出现按键两端的I/O口同时为低.

4.3 一种典型三极管控制电路



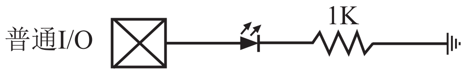
如果用弱上拉控制，建议加上拉电阻R1(3.3K～10K)，如果不加上拉电阻R1(3.3K～10K)，建议R2的值在15K以上，或用强推挽输出。

4.4 典型发光二极管控制电路



弱上拉/准双向口，用灌电流驱动发光二极管

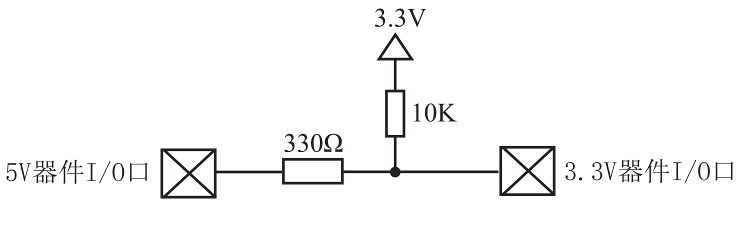
限流电阻尽量大于1K，最小不要小于470Ω



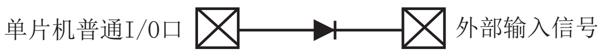
推挽/强上拉口，用拉电流驱动发光二极管

4.5 混合电压供电系统3V/5V器件I/O口互连

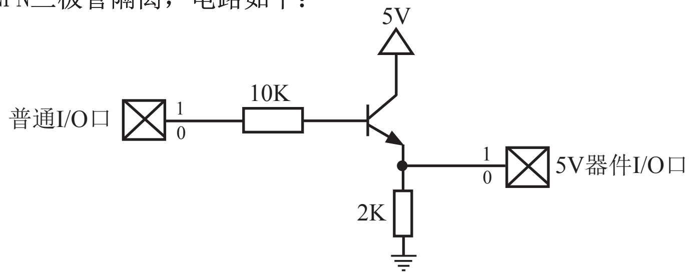
STC15F100系列5V单片机连接3.3V器件时，为防止3.3V器件承受不了5V，可将相应的5V单片机I/O口先串一个330Ω的限流电阻到3.3V器件I/O口，程序初始化时将5V器件的I/O口设置成开漏配置，断开内部上拉电阻，相应的3.3V器件I/O口外部加10K上拉电阻到3.3V器件的Vcc，这样高电平是3.3V，低电平是0V，输入输出一切正常。



STC15L100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。



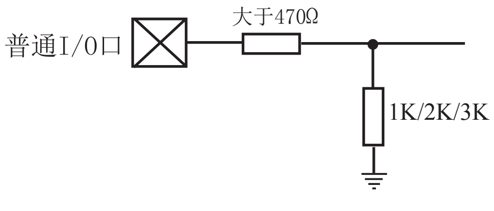
STC15L100系列3V单片机连接5V器件时，为防止3V器件承受不了5V，如果相应的I/O口是输出，可用一个NPN三极管隔离，电路如下：



4.6 如何让I/O口上电复位时为低电平

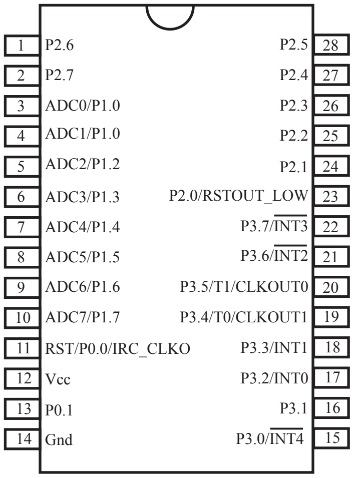
普通8051单片机上电复位时普通I/O口为弱上拉高电平输出,而很多实际应用要求上电时某些I/O口为低电平输出,否则所控制的系统(如马达)就会误动作,现STC15系列单片机由于既有弱上拉输出又有强推挽输出,就可以很轻松的解决此问题。

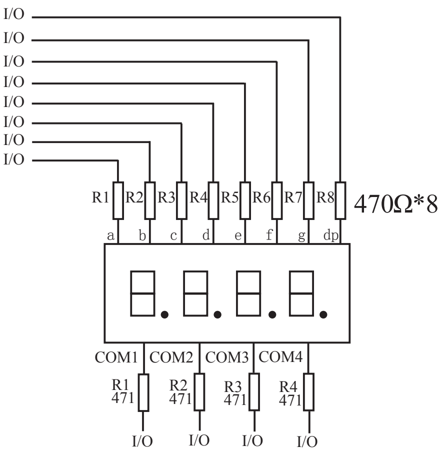
现可在STC15系列单片机I/O口上加一个下拉电阻(1K/2K/3K),这样上电复位时，虽然单片机内部I/O口是弱上拉/高电平输出，但由于内部上拉能力有限，而外部下拉电阻又较小,无法将其拉高,所以该I/O口上电复位时外部为低电平。如果要将此I/O口驱动为高电平，可将此I/O口设置为强推挽输出，而强推挽输出时，I/O口驱动电流可达20mA,故肯定可以将该口驱动为高电平输出。



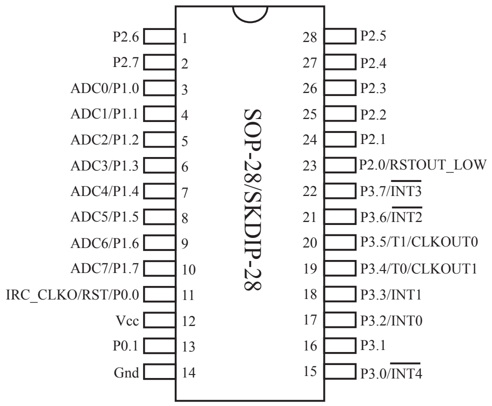
特别提示：STC15F100系列的P3.3/RSTOUT_LOW管脚上电复位后为低电平输出，其他管脚上电复位后均为高电平输出

4.7 I/O口直接驱动LED数码管应用线路图

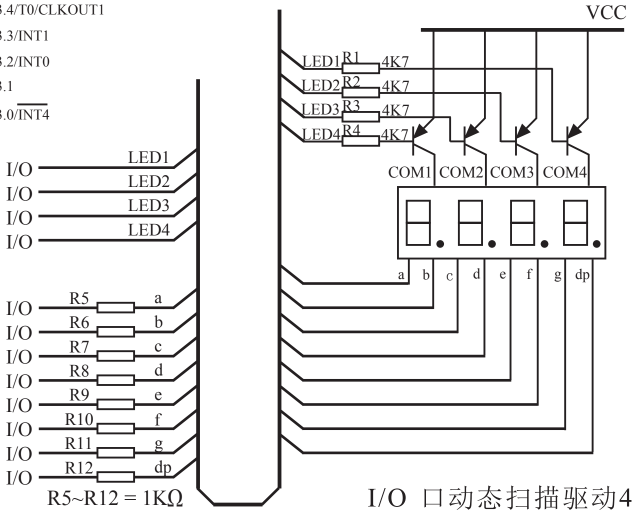




I/O 口动态扫描驱动4个
共阴极数码管参考电路图

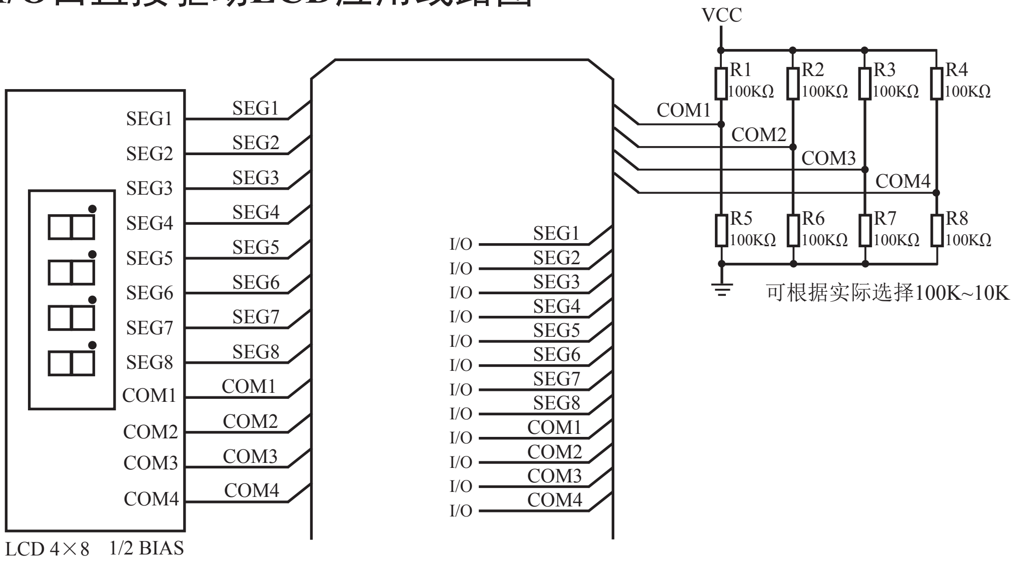


I/O 口动态扫描驱动数码时，可以一次点亮一个数码管中的8段，但为降低功耗，建议可以一次只点亮其中的4段或者2段



I/O 口动态扫描驱动4个
共阳极数码管参考电路图

4.8 I/O口直接驱动LCD应用线路图



如何点亮相应的LCD像素：

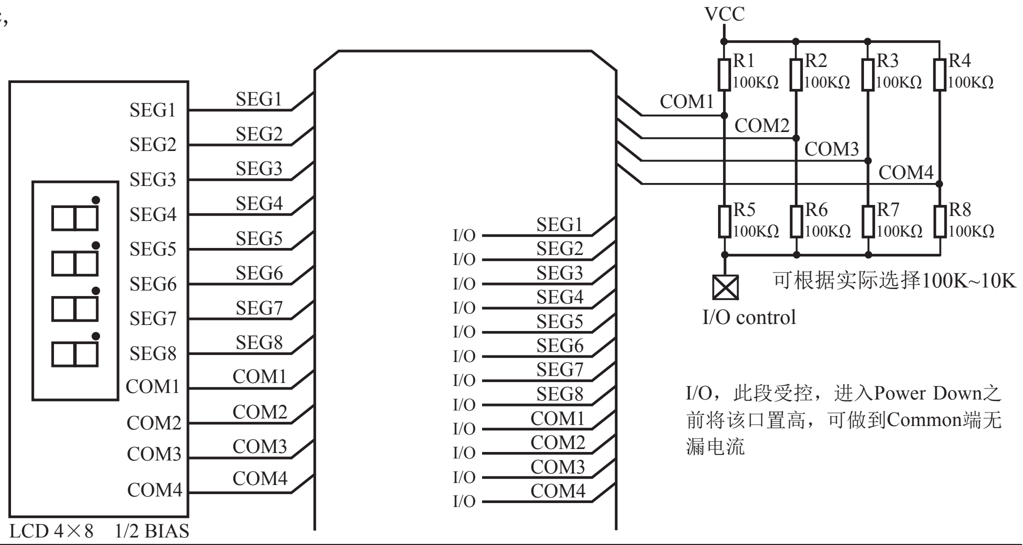
当相应的Common端和相应的Segment端压差大于1/2Vcc时，相应的像素就显示，当压差小于1/2Vcc时，相应的像素就不显示

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，



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

这条指令饿功能是将立即数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6~12倍

如果按功能分类，STC15F100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

STC15F100系列 指令执行所需时钟

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位立即数送入地址寄存器	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	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	@Ri, A	累加器送逻辑上在外部的片内扩展RAM（8位地址）	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	出栈送直接地址单元	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

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	寄存器内容送入累加器	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	立即加到累加器	2	12	2	6倍
ADDC　A，Rn	寄存器内容带进位加到累加器	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，d irect	累加器与直接地址单元相“ 或”	2	12	3	4倍
ORL A，# data	累加器与间接RAM单元相“ 或”	1	12	3	4倍
ORL direct， A	累加器与立即数相“ 或”	2	12	2	6倍
ORL direct， #data	直接地址单元与累加器相“ 或”	2	12	4	3倍
ORL A，#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， d irect	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	24	1	12倍
RLC A	累加器带进位位循环左移	1	48	1	12倍
RR A	累加器循环右移	1	48	1	12倍
RRC A	累加器带进位位循环右	1	12	1	12倍
SWAP A	累加器半字节交换	1	12	1	12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	3	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

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	3	8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

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，STC15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单片机的指令说明

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)←(A) + (C) + (Rn)

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 1	direct address

Operation: ADDC
(A)←(A) + (C) + (direct)

ADDC A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 1 1	0 1 1 i

Operation: ADDC
(A)←(A) + (C) + ((Ri))

ADDC A,#data

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 0	immediate data

Operation: ADDC
(A)←(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)← (PC)+ 2
(PC_{10-0})← 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: ADD

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

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    [[($A_{3-0}$) > 9] V [(AC) = 1]]
      THEN($A_{3-0}$) ← ($A_{3-0}$) + 6
              AND
IF    [[($A_{7-4}$) > 9] V [(C) = 1]]
      THEN ($A_{7-4}$) ← ($A_{7-4}$) + 6
```

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH.
No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.
Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7FH (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)←(A) −1

DEC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 1	1 r r r

Operation: DEC
(Rn)←(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

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

DJNZ <byte>, <rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to 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) ← (PC) + 2
(Rn) ← (Rn) – 1
IF  (Rn) > 0 or (Rn) < 0
    THEN
        (PC) ← (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
$(PC) \leftarrow (PC) + 2$
IF $(A) = 0$
    THEN    $(PC) \leftarrow (PC) + rel$
```

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
$(PC) \leftarrow (PC) + 3$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC_{7-0})$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC_{15-8})$
$(PC) \leftarrow addr_{15-0}$
```

LJMP addr16

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) $\leftarrow$ $addr_{15-0}$

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) $\leftarrow$ (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$_{15\text{-}0}$
DPH DPL ← #data$_{15\text{-}8}$ #data$_{7\text{-}0}$

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

MOVX @Ri, A

Bytes: 1

Cycles: 2

Encoding:

1 1 1 1	0 0 1 i

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

MOVX @DPTR, A

Bytes: 1

Cycles: 2

Encoding:

1 1 1 1	0 0 0 0

Operation: MOVX
$(DPTR) \leftarrow (A)$

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

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

Function: Logical-OR for byte variables

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

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

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

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

```
ORL   A, R0
```

will leave the Accumulator holding the value 0D7H (11010111B).
When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time.The instruction,

```
ORL     P1, #00110010B
```

will set bits 5,4, and 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) ← (A) - (C) - (Rn)

SUBB A, direct

Bytes: 2

Cycles: 1

Encoding:

1 0 0 1	0 1 0 1	direct address

Operation: SUBB
(A) ← (A) - (C) - (direct)

SUBB A, @Ri

Bytes: 1

Cycles: 1

Encoding:

1 0 0 1	0 1 1 i

Operation: SUBB
(A) ← (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 combinna-tion 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) ⊕ (Rn)

XRL A, direct

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 1	direct address

Operation: XRL
(A) ← (A) ⊕ (direct)

XRL A, @Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 1 0	0 1 1 i

Operation: XRL
(A) ← (A) ⊕ ((Ri))

XRL A, #data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 0	immediate data

Operation: XRL
(A) ← (A) ⊕ #data

XRL direct, A

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 0 1 0	direct address

Operation: XRL
(direct) ← (direct) ⊕ (A)

XRL direct, #dataw

Bytes: 3

Cycles: 2

Encoding:

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

Operation: XRL
(direct) ← (direct) ⊕ # 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。

STC15F100系列单片机提供了8个中断请求源，它们分别是：外部中断0(INT0)、定时器0中断、外部中断1(INT1)、定时器1中断、低压检测(LVD)中断、外部中断2($\overline{INT2}$)、外部中断3($\overline{INT3}$)以及外部中断4($\overline{INT4}$)。除外部中断2($\overline{INT2}$)、外部中断3($\overline{INT3}$)及外部中断4($\overline{INT4}$)固定是最低优先级中断外，其它的中断都具有两个中断优先级，可实现两级中断服务程序嵌套。用户可以用关总中断允许位(EA/IE.7)或相应中断的允许位屏蔽相应的中断请求，也可以用打开相应的中断允许位来使CPU响应相应的中断申请；每一个中断源可以用软件独立地控制为开中断或关中断状态；部分中断的优先级别均可用软件设置。高优先级的中断请求可以打断低优先级的中断，反之，低优先级的中断请求不可以打断高优先级的中断。当两个相同优先级的中断同时产生时，将由查询次序来决定系统先响应哪个中断。

STC15F100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	中断优先级	中断请求标志位	中断允许控制位
INT0 (外部中断 0)	0003H	0 (highest)	PX0	0/1	IE0	EX0/EA
Timer 0	000BH	1	PT0	0/1	TF0	ET0/EA
INT1 (外部中断1)	0013H	2	PX1	0/1	IE1	EX1/EA
Timer1	001BH	3	PT1	0/1	TF1	ET1/EA
No S1(UART1)	0023B	4				
ADC	002BH	5				
LVD	0033H	6	PLVD	0/1	LVDF	ELVD/EA
No PCA	003BH	7				
No S2(UART2)	0043H	8				
No SPI	004BH	9				
$\overline{\text{INT2}}$ （外部中断2）	0053H	10		0		EX2/EA
$\overline{\text{INT3}}$ (外部中断3)	005BH	11		0		EX3/EA
No BRT_INT	0063H	12				
-	006BH	13				
System Reserved	0073H	14				
System Reserved	007BH	15				
$\overline{\text{INT4}}$ (外部中断4)	0083H	16(lowest)		0		EX4/EA

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    LVD_Routine(void)         interrupt 6;
void    Int2_Routine(void)        interrupt 10;
void    Int3_Routine(void)        interrupt 11;
void    Int4_Routine(void)        interrupt 16;
```

6.1 中断结构

STC15F100系列单片机的中断系统结构示意图如图6-1所示

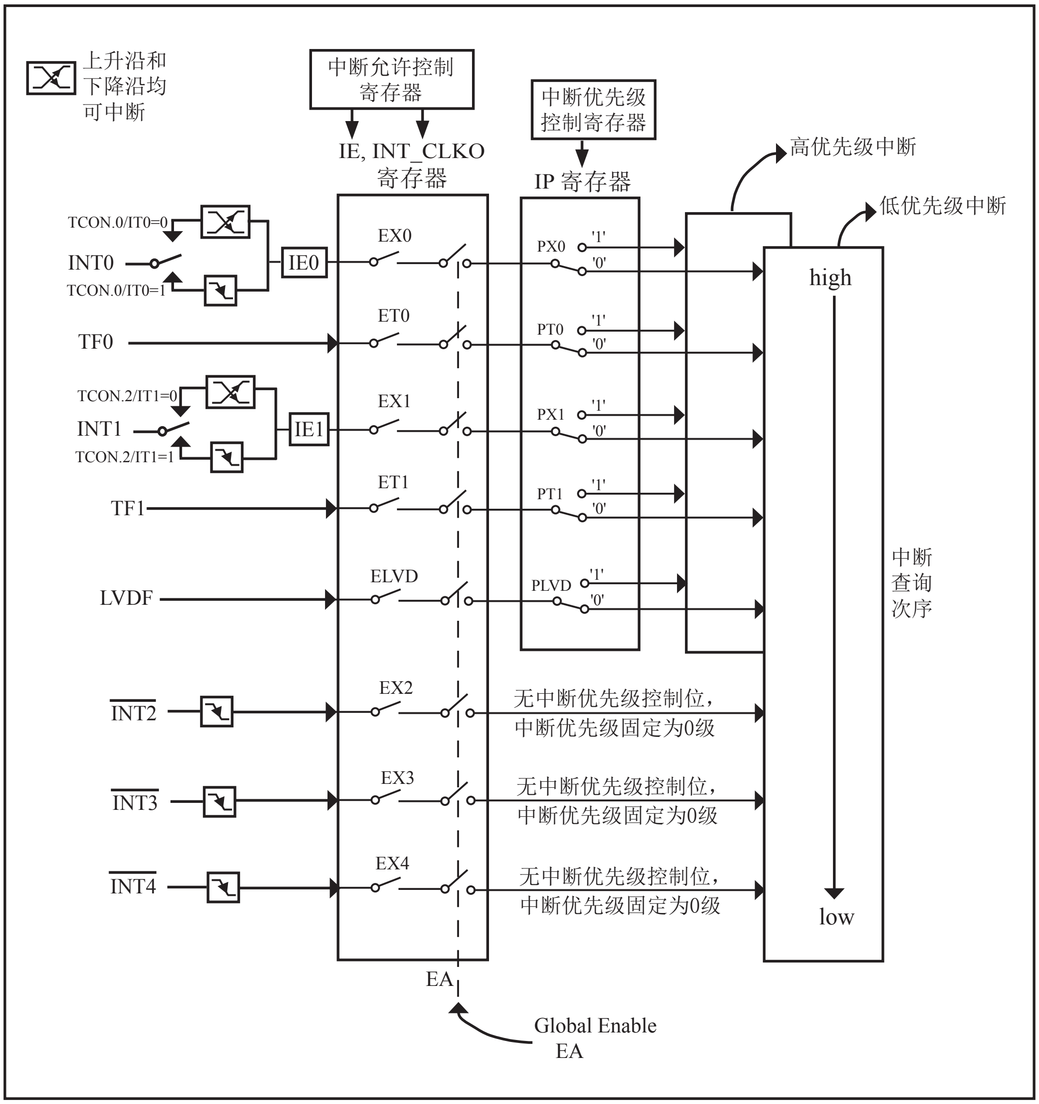


图6-1 STC15F100系列中断系统结构图

外部中断0(INT0)和外部中断1(INT1)既可上升沿触发，又可下降沿触发。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是上升沿触发还是下降沿触发。如果ITx = 0(x = 0,1)，那么系统在INTx(x = 0,1)脚探测到上升沿或下降沿后均可产生外部中断。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。

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。

低压检测(LVD)中断是由LVDF/PCON.5请求产生的。该位也需用软件清除。

外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)都只能下降沿触发。外部中断2~4的中断请求标志位被隐藏起来了，对用户不可见。当相应的中断服务程序执行后或EXn=0(n=2,3,4)，这些中断请求标志位会自动地被清0。外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)也可以用于将单片机从掉电模式唤醒。

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
INT0 (外部中断0)	(IT0 = 1): 下降沿；　(IT0 = 0): 上升沿和下降沿均可
Timer 0	定时器0溢出
INT1 (外部中断1)	(IT1 = 1): 下降沿；　(IT1 = 0): 上升沿和下降沿均可
Timer1	定时器1溢出
LVD	电源电压下降到低于LVD检测电压
$\overline{\text{INT2}}$ (外部中断2)	下降沿
$\overline{\text{INT3}}$ (外部中断3)	下降沿
$\overline{\text{INT4}}$ (外部中断4)	下降沿

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	ELVD	-	-	ET1	EX1	ET0	EX0	000x 0000B
IP	Interrupt Priority Low	B8H	-	PLVD	-	-	PT1	PX1	PT0	PX0	x00x 0000B
TCON	Timer Control register	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
PCON	Power Control register	87H	-	-	LVDF	POF	GF1	GF0	PD	IDL	xx11 0000B
INT_CLKO	External Interrupt enable and Clock output register	8FH	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO	x000 xx00B

上表中列出了与STC15F100系列单片机中断相关的所有寄存器，下面逐一地对这些寄存器进行介绍。

1. 中断允许寄存器IE和INT_CLKO

STC15F100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	ELVD	-	-	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。

ELVD：低压检测中断允许位，ELVD=1，允许低压检测中断，ELVD=0，禁止低压检测中断。

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。

INT_CLKO是STC15F100系列单片机新增寄存器，地址是8FH，INT_CLKO格式如下：

INT_CLKO : 外部中断允许和时钟输出寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
INT_CLKO	8FH	name	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO

EX4 : 外部中断4($\overline{\text{INT4}}$)中断允许位，EX4=1允许中断，EX4=0禁止中断。外部中断4($\overline{\text{INT4}}$)只能下降沿触发。

EX3 : 外部中断3($\overline{\text{INT3}}$)中断允许位，EX3=1允许中断，EX3=0禁止中断。外部中断3($\overline{\text{INT3}}$)也只能下降沿触发。

EX2 : 外部中断2($\overline{\text{INT2}}$)中断允许位，EX2=1允许中断，EX2=0禁止中断。外部中断2($\overline{\text{INT2}}$)同样只能下降沿触发。

T1CLKO,T0CLKO与中断无关，在此不作介绍。

STC15F100系列单片机复位以后，IE和INT_CLKO被清0，由用户程序置“1”或清“0”IE和INT_CLKO的相应位，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INT_CLKO(不可位寻址)的内容可用MOV　INT_CLKO, #DATA指令来解决。

2. 中断优先级控制寄存器IP

STC15F100系列单片机部分中断设有两个中断优先级，除外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)外，所有中断请求源可编程为高优先级中断或低优先级中断，可实现二级中断嵌套。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

STC15F100系列单片机的片内有一个优先级寄存器IP，其字节地址为B8H，只要用程序改变其内容，即可进行各中断源中断级别的设置，IP寄存器格式：

IP：中断优先级控制寄存器（可位寻址）

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

PLVD：低压检测中断优先级控制位。
PLVD=1，低压检测中断定义为高优先级中断；
PLVD=0，低压检测中断定义为低优先级中断。

PT1：定时器T1中断优先级控制位。
PT1=1，定时器T1中断定义为高优先级中断；
PT1=0，定时器T1中断定义为低优先级中断。

PX1：外部中断1中断优先级控制位。
PX1=1，外部中断1定义为高优先级中断；
PX1=0，外部中断1定义为低优先级中断。

PT0：定时器T0中断优先级控制位。
PT0=1，定时器T0定义为高优先级中断；
PT0=0，定时器T0定义为低优先级中断。

PX0：外部中断0中断优先级控制位。
PX0=1，外部中断0定义为高优先级中断；
PX0=0，外部中断0定义为低优先级中断。

中断优先级控制寄存器IP的各位都由用户程序置"1"和清"0"，可用位操作指令或字节操作指令更新IP的内容。以改变各中断源的中断优先级。STC15F100系列单片机复位后IP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（INT1/P3.3）中断请求标志。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，INT1/P3.3引脚上的上升沿或下降沿信号均可触发外部中断1。IT1=1，外部中断1为下降沿触发方式。

IE0：外部中断0（INT0/P3.2）中断请求标志。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。

IT0：外部中断0中断源类型选择位。IT0=0，INT0/P3.2引脚上的上升沿或下降沿均可触发外部中断0。IT0=1，外部中断0为下降沿触发方式。

4. 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	-	-	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。

在正常工作和空闲工作状态时，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。

在进入掉电工作状态前，如果低压检测电路未被允许可产生中断，则在进入掉电模式后，该低压检测电路不工作以降低功耗。如果被允许可产生低压检测中断，则在进入掉电模式后，该低压检测电路继续工作，在内部工作电压Vcc低于低压检测门槛电压后，产生低压检测中断，可将MCU从掉电状态唤醒。

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ELVD/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	ELVD	-	-	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。

ELVD： 低压检测中断允许位，ELVD=1，允许低压检测中断，ELVD=0，禁止低压检测中断。

6.3 中断优先级

除外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)外，STC15F100系列单片机的所有的中断都具有两个中断优先级，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，不能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，STC15F100系列单片机各中断优先查询次序如下：

	中断源	查询次序
0.	INT0	(highest)
1.	Timer 0	↓
2.	INT1	
3.	Timer 1	
4.		
5.		
6.	LVD	
7.		
8.		
9.		
10.	$\overline{\text{INT2}}$	
11.	$\overline{\text{INT3}}$	
12.		
13.		
14.		
15.		↓
16.	$\overline{\text{INT4}}$	(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    LVD_Routine(void)        interrupt 6;
void    Int2_Routine(void)       interrupt 10;
void    Int3_Routine(void)       interrupt 11;
void    Int4_Routine(void)       interrupt 16;
```

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。中断服务程序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
/	0023H
/	002BH
LVD	0033H
/	003BH
/	0043H
/	004BH
External Interrupt 2	0053H
External Interrupt 3	005BH
/	0063H
/	006BH
/	0073H
/	007BH
External Interrupt 4	0083H

当“转去执行中断”时，引起外部中断INT0/INT1/$\overline{\text{INT2}}$/$\overline{\text{INT3}}$/$\overline{\text{INT4}}$请求标志位和定时器/计数器0、定时器/计数器1的中断请求标志位将被硬件自动清零，其它中断的中断请求标志位需软件清"0"。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)。

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，上升沿或下降沿均可触发方式和仅下降沿触发方式。

TCON寄存器中的IT0/TCON.0和IT1/TCON.2决定了外部中断0和1是上升沿和下降沿均可触发还是仅下降沿触发。如果ITx = 0(x = 0,1)，那么系统在INTx(x = 0,1)脚探测到上升沿或下降沿后均可产生外部中断。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。

外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)都只能下降沿触发。外部中断2~4的中断请求标志位被隐藏起来了，对用户不可见，故也无需用户清"0"。当相应的中断服务程序执行后或EXn=0(n=2,3,4)，这些中断请求标志位会自动地被清0。外部中断2($\overline{\text{INT2}}$)、外部中断3($\overline{\text{INT3}}$)及外部中断4($\overline{\text{INT4}}$)也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个时钟，而且低电平也要持续至少一个时钟，才能确保该下降沿被CPU检测到。同样，如果外部中断是上升沿、下降沿均可触发，则要求必须在相应的引脚维持低电平或高电平至少1个时钟，而且高电平或低电平也要持续至少一个时钟，这样才能确保CPU能够检测到该上升沿或下降沿。

6.6 外部中断的测试程序(C程序和汇编程序)

6.6.1 外部中断0(INT0)的测试程序(可支持上升沿或下降沿中断)

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断0(上升沿/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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/


#include "reg51.h"

bit FLAG;                                   //1:上升沿中断   0:下降沿中断

//External interrupt0 service routine
void exint0() interrupt 0                   //中断0 (向量地址 0003H)
{
        FLAG = INT0;            //读 INT0(P3.2)管脚的状态, INT0=0(下降沿Falling ); INT0=1(上升沿Rising)
}

void main()
{
        IT0 = 0;                            //设置 INT0 的中断触发方式 (1:Falling only 0:Rising & Falling)
        EX0 = 1;                            //允许INT0 中断
        EA = 1;                             //开总中断
        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断0(上升沿/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宏晶科技的资料及程序 ----------*/
/*---------------------------------------------------------------------------------------*/



            FLAG    BIT    20H.0                       ;1:上升沿中断   0:下降沿中断

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

            ORG     0000H
            LJMP    MAIN

            ORG     0003H                              ;中断0 (向量地址 0003H)
            LJMP    EXINT0

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

            ORG     0100H
MAIN:
            MOV     SP,         #7FH                   ;初始化堆栈指针SP
            CLR     IT0                               ;设置 INT0 的中断触发方式 (1:Falling only 0:Rising & Falling)
            SETB    EX0                                ;允许INT0 中断
            SETB    EA                                 ;开总中断
            SJMP    $

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

EXINT0:
            PUSH    PSW
            MOV     C,          INT0                   ;读 INT0(P3.2)管脚的状态,
                                                       ;INT0=0(下降沿Falling ); INT0=1(上升沿Rising)

            MOV     FLAG,       C
            POP     PSW
            RETI

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

            END
```

6.6.2 外部中断1(INT1)的测试程序（可支持上升沿或下降沿中断）

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断1(上升沿/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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/



#include "reg51.h"

bit FLAG;                          //1:rising edge int 0:falling edge int

//External interrupt1 service routine
void exint1() interrupt 2          //interrupt 2 (location at 0013H)
{
        FLAG = INT1;               //read INT1(P3.3) port status, INT1=0(Falling); INT1=1(Rising)
}

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

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断1(上升沿/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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/



        FLAG    BIT    20H.0                    ;1:rising edge int 0:falling edge int

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

EXINT1:
        PUSH    PSW
        MOV     C,      INT1                    ;read INT1(P3.3) port status
        MOV     FLAG,   C                       ;INT1=0(Falling); INT1=1(Rising)
        POP     PSW
        RETI

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

        END
```

6.6.3 外部中断2($\overline{\text{INT2}}$)的测试程序（下降沿中断）

C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断2 (INT2) (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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




#include "reg51.h"

sfr INT_CLKO = 0x8f;                    //- EX4 EX3 EX2 - - T1CLKO T0CLKO

//External interrupt2 service routine
void exint2() interrupt 10              //interrupt 10 (location at 0053H)
{
}

void main()
{
        INT_CLKO |= 0x10;               //(EX2 = 1)enable INT2 interrupt
        EA = 1;                         //open global interrupt switch

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断2 (INT2) (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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/



INT_CLKO    DATA    08FH                        ;- EX4 EX3 EX2 - - T1CLKO T0CLKO

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

            ORG     0000H
            LJMP    MAIN

            ORG     0053H                       ;interrupt 10 (location at 0053H)
            LJMP    EXINT2

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

            ORG     0100H
MAIN:
            MOV     SP,         #7FH            ;initial SP
            ORL     INT_CLKO,           #10H    ;(EX2 = 1)enable INT2 interrupt
            SETB    EA                          ;open global interrupt switch
            SJMP    $

;-----------------------------------------
;External interrupt2 service routine

EXINT2:
            RETI

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

            END
```

6.6.4 外部中断3($\overline{\text{INT3}}$)的测试程序(下降沿中断)

C程序:

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断3 (INT3) (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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




#include "reg51.h"

sfr INT_CLKO = 0x8f;                    //- EX4 EX3 EX2 - - T1CLKO T0CLKO

//External interrupt3 service routine
void exint3() interrupt 11              //interrupt 11 (location at 005BH)
{
}

void main()
{
        INT_CLKO |= 0x20;               //(EX3 = 1)enable INT3 interrupt
        EA = 1;                         //open global interrupt switch

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited -------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断3 ($\overline{INT3}$) (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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/



INT_CLKO    DATA    08FH                ;- EX4 EX3 EX2 - - T1CLKO T0CLKO

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

            ORG     0000H
            LJMP    MAIN

            ORG     005BH               ;interrupt 11 (location at 005BH)
            LJMP    EXINT3

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

            ORG     0100H
MAIN:
            MOV     SP,         #7FH                ;initial SP
            ORL     INT_CLKO,           #20H        ;(EX3 = 1)enable $\overline{INT3}$ interrupt
            SETB    EA                              ;open global interrupt switch
            SJMP    $

;-----------------------------------------
;External interrupt 3 service routine

EXINT3:
            RETI

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

            END
```

6.6.5 外部中断4($\overline{\text{INT4}}$)的测试程序(下降沿中断)

C程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断4 (INT4) (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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




#include "reg51.h"

sfr INT_CLKO = 0x8f;                //- EX4 EX3 EX2 - - T1CLKO T0CLKO

//External interrupt4 service routine
void exint4() interrupt 16          //interrupt 16 (location at 0083H)
{
}

void main()
{
        INT_CLKO |= 0x40;           //(EX4 = 1)enable INT4 interrupt
        EA = 1;                     //open global interrupt switch

        while (1);
}
```

汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited -----------------------------------------*/
/* --- 演示STC 15 系列单片机外部中断4 (INT4) (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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/



INT_CLKO    DATA    08FH                                  ;- EX4 EX3 EX2 - - T1CLKO T0CLKO

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

            ORG     0000H
            LJMP    MAIN

            ORG     0083H                                 ;interrupt 16 (location at 0083H)
            LJMP    EXINT4

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

            ORG       0100H
MAIN:
            MOV       SP,         #7FH                    ;initial SP
            ORL       INT_CLKO,         #40H              ;(EX4 = 1)enable INT4 interrupt
            SETB      EA                                  ;open global interrupt switch
            SJMP      $

;-----------------------------------------
;External interrupt4 service routine

EXINT4:
            RETI

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

            END
```

第7章 定时器/计数器

STC15Fxx系列单片机内部设置了两个16位定时器/计数器T0和T1,它们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— C/$\overline{T}$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(16位自动重装模式),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	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	Timer Mode	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	Auxiliary register	8EH	T0x12	T1x12	-	-	-	-	-	-	00xx xxxxB
INT_CLKO	External interrupt enable and Clock Output register	8FH	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO	x000 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：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T1的运行控制位。该位由软件置位和清零。当GATE（TMOD.7）=0，TR1=1时就允许T1开始计数，TR1=0时禁止T1计数。当GATE（TMOD.7）=1，TR1=1且INT1输入高电平时，才允许T1计数。

TF0：T0溢出中断标志。T0被允许计数以后，从初值开始加1计数，当产生溢出时，由硬件置“1”TF0，向CPU请求中断，一直保持CPU响应该中断时，才由硬件清0（也可由查询软件清0）。

TR0：定时器T0的运行控制位。该位由软件置位和清零。当GATE（TMOD.3）=0，TR0=1时就允许T0开始计数，TR0=0时禁止T0计数。当GATE（TMOD.3）=1，TR1=0且INT0输入高电平时，才允许T0计数。

IE1：外部中断1请求源（INT1/P3.3）标志。IE1=1，外部中断向CPU请求中断，当CPU响应该中断时由硬件清“0”IE1。

IT1：外部中断源1触发控制位。IT1=0，上升沿或下降沿均可触发外部中断1。IT1=1，外部中断1程控为下降沿触发方式。

IE0：外部中断0请求源（INT0/P3.2）标志。IE0=1外部中断0向CPU请求中断，当CPU响应外部中断时，由硬件清“0”IE0（边沿触发方式）。

IT0：外部中断源0触发控制位。IT0=0，上升沿或下降沿均可触发外部中断0。IT0=1，外部中断0程控为下降沿触发方式。

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	16位自动重装定时器，当溢出时将RL_TH1和RL_TL1存放的值自动重装入TH1和TL1中。
	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	16位自动重装定时器，当溢出时将RL_TH0和RL_TL0存放的值自动重装入TH0和TL0中。
	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

STC15F100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条机器指令执行速度是固定的，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	-	-	-	-	-	-

T0x12：0，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
T1x12：0，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

4. T0和T1的时钟输出寄存器和外部中断允许INT_CLKO

CLKOUT0/P3.5和CLKOUT1/P3.4的时钟输出控制由INT_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0(16位自动重装模式)或模式2(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.

INT_CLKO格式如下：

INT_CLKO：外部中断允许和时钟输出寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
INT_CLKO	8FH	name	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO

T1CLKO ：

1，将P3.4/T0管脚配置为定时器1的时钟输出CLKOUT1，输出时钟频率= T1溢出率/2

若定时器/计数器T1工作在定时器模式0(16位自动重装模式)，

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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (65536-[RL_TH1, RL_TL1])/2

T1工作在12T模式(AUXR.6/T1x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH1, RL_TL1])/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) / (65536-[RL_TH1, RL_TL1])/2

若定时器/计数器T1工作在模式2(8位自动重装模式)，

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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (256-TH1)/2

T1工作在12T模式(AUXR.6/T1x12=0)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P3.4/T0管脚被配置为定时器1的时钟输出

T0CLKO ：

1，将P3.5/T1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

若定时器/计数器T0工作在定时器模式0(16位自动重装模式)时，

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(AUXR.7/T0x12=1)时的输出频率 = (SYSclk)/(65536-[RL_TH0, RL_TL0])/2

T0工作在12T模式(AUXR.7/T0x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH0, RL_TL0])/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) / (65536-[RL_TH0, RL_TL0])/2

若定时器/计数器T0工作在定时器模式2(8位自动重装模式)，如果C/$\overline{T}$=0且则：

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

T0工作在12T模式(AUXR.7/T0x12=0)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P3.5/T1管脚被配置为定时器0的时钟输出

EX4：外部中断4($\overline{\text{INT4}}$)中断允许位，EX4=1允许中断，EX4=0禁止中断。外部中断4($\overline{\text{INT4}}$)只能下降沿触发。

EX3：外部中断3($\overline{\text{INT3}}$)中断允许位，EX3=1允许中断，EX3=0禁止中断。外部中断3($\overline{\text{INT3}}$)也只能下降沿触发。

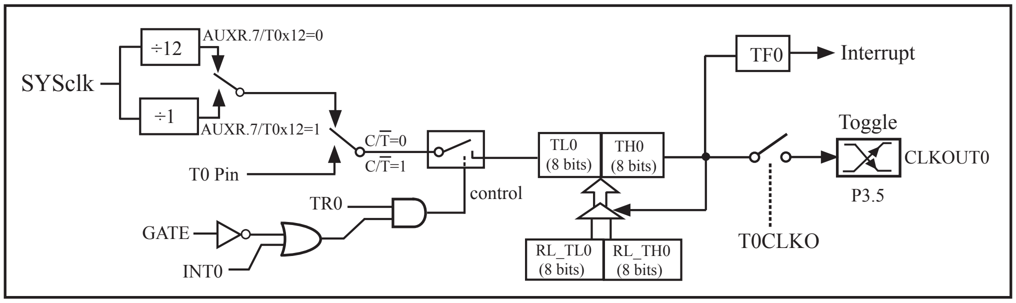
EX2：外部中断2($\overline{\text{INT2}}$)中断允许位，EX2=1允许中断，EX2=0禁止中断。外部中断2($\overline{\text{INT2}}$)同样只能下降沿触发。

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(16位自动重装)及测试程序，建议只学习此模式足矣

此模式下定时器/计数器0作为可自动重装载的16位计数器，如下图所示。



定时器/计数器0的模式 0: 16位自动重装

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对内部系统时钟计数，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。

STC15F100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有2个隐藏的寄存器RL_TH0和RL_TL0。RL_TH0与TH0共有同一个地址，RL_TL0与TL0共有同一个地址。当定时器0工作在模式0(TMOD[1:0]/[M1,M0]=00B)时，[TL0,TH0]的溢出不仅置位TF0，而且会自动将[RL_TL0,RL_TH0]的内容重新装入[TL0,TH0]。

当T0CLKO/INT_CLKO.0=1时，P3.5/T1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)/(65536-[RL_TH0, RL_TL0])/2

T0工作在12T模式(AUXR.7/T0x12=0)时的输出时钟频率=(SYSclk)/12/(65536-[RL_TH0, RL_TL0])/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) / (65536-[RL_TH0, RL_TL0])/2

定时器0的16位自动重装模式的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机定时器0的16位自动重装模式 ----------*/
/* 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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

/* define constants */
#define SYSclk 18432000L
#define MODE1T                    //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef MODE1T
#define T1MS (65536-SYSclk/1000)                    //1ms timer calculation method in 1T mode
#else
#define T1MS (65536-SYSclk/12/1000)          //1ms timer calculation method in 12T mode
#endif

/* define SFR */
sfr AUXR = 0x8e;                             //Auxiliary register
sbit TEST_LED = P3^1;                        //work LED, flash once per second

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

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

/* Timer0 interrupt routine */
void tm0_isr() interrupt 1 using 1
{
        if (count-- == 0)                           //1ms * 1000 -> 1s
        {
                count = 1000;                       //reset counter
                TEST_LED = ! TEST_LED;              //work LED flash
        }
}
```

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

/* main program */
void main()
{
#ifdef MODE1T
        AUXR = 0x80;                //timer0 work in 1T mode
#endif
        TMOD = 0x00;                //set timer0 as mode0 (16-bit auto-reload)
        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
}
```

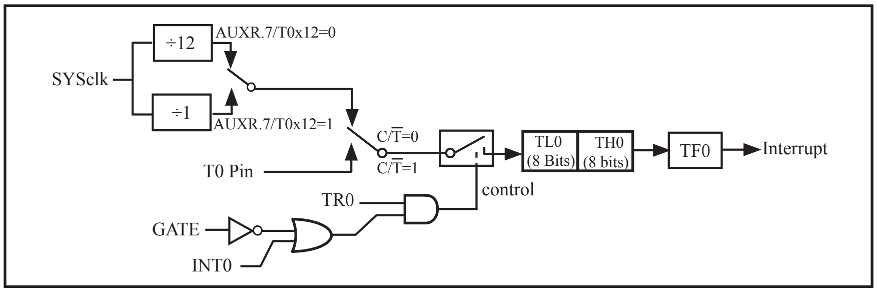
2. 汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机定时器0的16位自动重装模式 ----------*/
/* 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/




;/* 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  P3.1               ;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 MODE1T
            MOV     AUXR,   #80H                ;timer0 work in 1T mode
#endif
            MOV     TMOD,   #00H                ;set timer0 as mode0 (16-bit auto-reload)
            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     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
```

7.2.2 模式1(16位定时器)，不建议学习

此模式下定时器/计数器0作为16位定时器，如下图所示。



定时器/计数器0的模式 1: 16位定时器

此模式下，定时器0配置为16位的计数器，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。

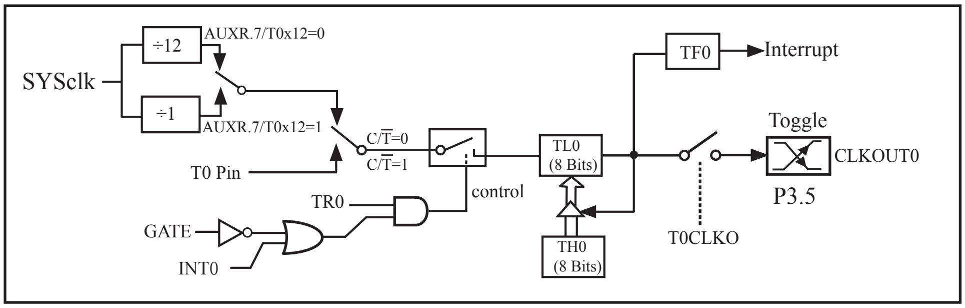
当GATE=0(TMOD.3)时，如TR0=1，则定时器计数。GATE=1时，允许由外部输入INT0控制定时器0，这样可实现脉宽测量。TR0为TCON 寄存器内的控制位，TCON寄存器各位的具体功能描述见上节TCON寄存器的介绍。

当C/$\overline{T}$=0时，多路开关连接到系统时钟的分频输出，T0对内部系统时钟计数，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。

STC15F100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。

7.2.3 模式2(8位自动重装模式)，不建议学习

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。

当T0CLKO/INT_CLKO.0=1时，P3.5/T1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(P3.4/T0)计数，则:

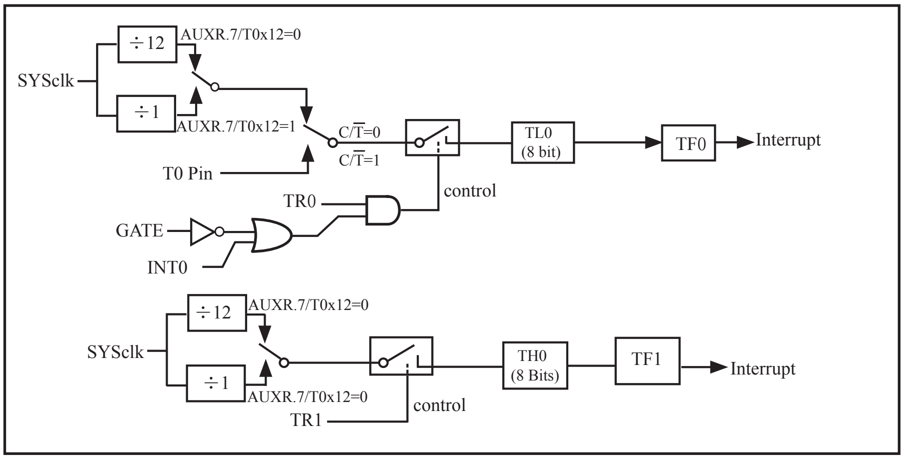
输出时钟频率 = (T0_Pin_CLK) / (256-TH0) / 2

7.2.4 模式3(两个8位计数器)，不建议学习

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/$\overline{\text{T}}$、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。



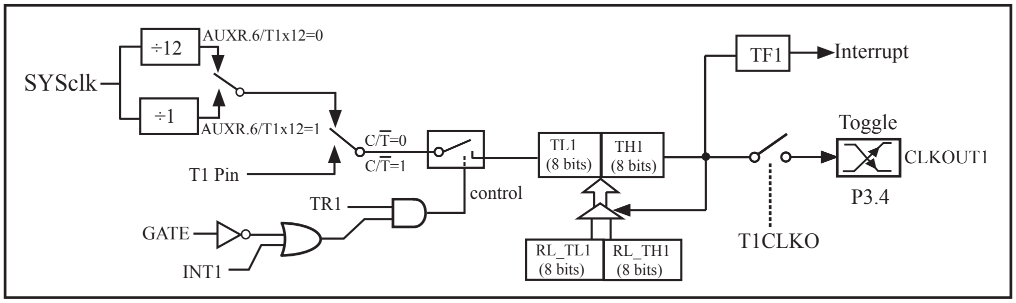
定时/计数器0 模式3: 两个8位计数器

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(16位自动重装)及测试程序，建议只学习此模式足矣

此模式下定时器/计数器1作为可自动重装载的16位计数器，如下图所示。



定时器/计数器1的模式 0: 16位自动重装

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对内部系统时钟计数，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。

STC15F100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有2个隐藏的寄存器RL_TH1和RL_TL1。RL_TH1与TH1共有同一个地址，RL_TL1与TL1共有同一个地址。当定时器1工作在模式0(TMOD[5:4]/[M1,M0]=00B)时，[TL1,TH1]的溢出不仅置位TF1，而且会自动将[RL_TL1,RL_TH1]的内容重新装入[TL1,TH1]。

当T1CLKO/INT_CLKO.1=1时，P3.4/T0管脚配置为定时器1的时钟输出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) / (65536-[RL_TH1, RL_TL1])/2

T1工作在12T模式(AUXR.6/T1x12=0)时的输出时钟频率=(SYSclk)/12/(65536-[RL_TH1, RL_TL1])/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) / (65536-[RL_TH1, RL_TL1])/2

定时器1的16位自动重装模式的测试程序

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机定时器1的16位自动重装模式 ----------*/
/* 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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

/* define constants */
#define SYSclk 18432000L
#define MODE1T                      //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef MODE1T
#define T1MS (65536-SYSclk/1000)            //1ms timer calculation method in 1T mode
#else
#define T1MS (65536-SYSclk/12/1000)         //1ms timer calculation method in 12T mode
#endif

/* define SFR */
sfr AUXR = 0x8e;                            //Auxiliary register
sbit TEST_LED = P3^1;                       //work LED, flash once per second

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

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

/* Timer1 interrupt routine */
void tm1_isr() interrupt 3 using 1
{
        if (count-- == 0)                           //1ms * 1000 -> 1s
        {
                count = 1000;                       //reset counter
                TEST_LED = ! TEST_LED;              //work LED flash
        }
}
```

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

/* main program */
void main()
{
#ifdef MODE1T
        AUXR = 0x40;                //timer1 work in 1T mode
#endif
        TMOD = 0x00;                //set timer1 as mode0 (16-bit auto-reload)
        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
}
```

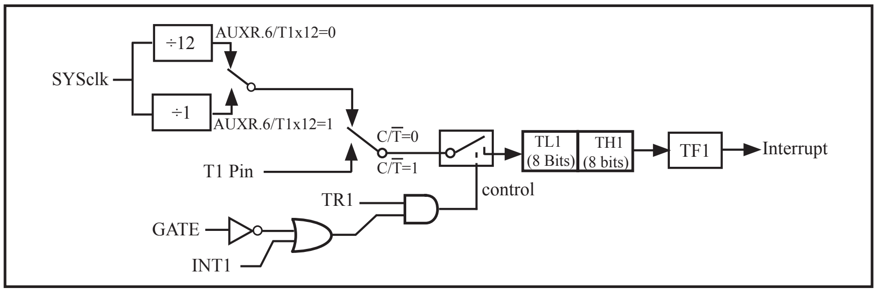
2. 汇编程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机定时器1的16位自动重装模式 ----------*/
/* 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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




;/* 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  P3.1           ;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,   #00H                    ;set timer1 as mode0 (16-bit auto-reload)
        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     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
```

7.3.2 模式1(16位定时器)，不建议学习

此模式下定时器/计数器1作为16位定时器，如下图所示。



定时器/计数器1的模式 1: 16位定时器

此模式下，定时器1配置为16位的计数器，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。

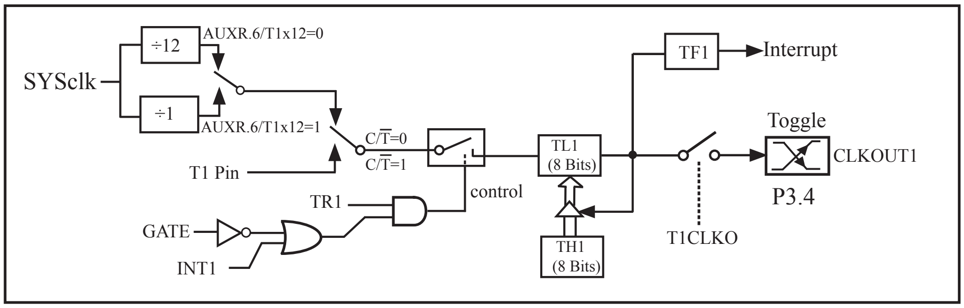
当GATE=0(TMOD.7)时，如TR1=1，则定时器计数。GATE=1时，允许由外部输入INT1控制定时器1，这样可实现脉宽测量。TR1为TCON寄存器内的控制位，TCON寄存器各位的具体功能描述见上节TCON寄存器的介绍。

当C/$\overline{T}$=0时，多路开关连接到系统时钟的分频输出，T1对内部系统时钟计数，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。

STC15F100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。

7.3.3 模式2(8位自动重装模式)，不建议学习

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/INT_CLKO.1=1时，P3.4/T0管脚配置为定时器1的时钟输出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(P3.5/T1)计数，则：

输出时钟频率 = (T1_Pin_CLK) / (256-TH1) / 2

7.4 可编程时钟输出

STC15F100系列有3种可编程时钟输出:IRC_CLKO/P0.0,CLKOUT0/P3.5,CLKOUT1/P3.4

IRC_CLKO : Internal R/C clock output register

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IRC_CLKO	BBH	name	EN_IRCO	-	-	-	DIVIRCO	-	-	-

如何利用IRC_CLKO/P0.0管脚输出时钟

IRC_CLKO/P0.0的时钟输出控制由IRC_CLKO寄存器的EN_IRCO位控制。设置EN_IRCO (IRC_CLKO.7)可将IRC_CLKO/P0.0管脚配置为内部R/C振荡时钟输出。通过设置DIVIRCO (IRC_CLKO.3)位可以设置内部R/C振荡时钟的输出频率是IRC_CLK/2还是IRC_CLK/1(不分频)

新增加的特殊功能寄存器：IRC_CLKO（地址：0xBB）

B7 - EN_IRCO ：

1，将IRC_CLKO/P0.0管脚配置为内部R/C振荡时钟输出

0，不允许IRC_CLKO/P0.0管脚配置为内部R/C振荡时钟输出

B3 － DIVIRCO ：

1，内部R/C振荡时钟的输出频率被2分频，输出时钟频率 ＝IRC_CLK/2

0，内部R/C振荡时钟的输出频率不被分频，输出时钟频率 ＝IRC_CLK/1

IRC_CLKO指内部R/C振荡时钟输出；IRC_CLK指内部R/C振荡时钟频率。

INT_CLKO : External Interrupt Enable and Clock Output register

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
INT_CLKO	8FH	name	-	EX4	EX3	EX2	-	-	T1CLKO	T0CLKO

如何利用CLKOUT0/P3.5和CLKOUT1/P3.4管脚输出时钟

如何利用CLKOUT0/P3.5和CLKOUT1/P3.4管脚输出时钟

CLKOUT0/P3.5管脚是否输出时钟由INT_CLKO寄存器的T0CLKO位控制

B0 - T0CLKO : 1, 允许时钟输出
0, 禁止时钟输出

CLKOUT1/P3.4管脚是否输出时钟由INT_CLKO寄存器的T1CLKO位控制

B1 - T1CLKO : 1, 允许时钟输出
0, 禁止时钟输出

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0(16位自动重装模式)或模式2(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：INT_CLKO（地址：0x8F）

B6 － EX4 ： 允许外部中断4($\overline{\text{INT4}}$)。

B5 － EX3 ： 允许外部中断3($\overline{\text{INT3}}$)。

B4 － EX2 ： 允许外部中断2($\overline{\text{INT2}}$)。

B1 – T1CLKO ：
1，将P3.4/T0管脚配置为定时器1的时钟输出CLKOUT1，输出时钟频率= T1溢出率/2
若定时器/计数器T1工作在定时器模式0(16位自动重装模式)，
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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (65536-[RL_TH1, RL_TL1])/2
T1工作在12T模式(AUXR.6/T1x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH1, RL_TL1])/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) / (65536-[RL_TH1, RL_TL1])/2
若定时器/计数器T1工作在模式2(8位自动重装模式)，
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(AUXR.6/T1x12=1)时的输出频率 = (SYSclk) / (256-TH1)/2
T1工作在12T模式(AUXR.6/T1x12=0)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P3.4/T0管脚被配置为定时器1的时钟输出

B0 – T0CLKO ：
1，将P3.5/T1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
若定时器/计数器T0工作在定时器模式0(16位自动重装模式)时，
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(AUXR.7/T0x12=1)时的输出频率 = (SYSclk)/(65536-[RL_TH0, RL_TL0])/2
T0工作在12T模式(AUXR.7/T0x12=0)时的输出频率 = (SYSclk) /12/ (65536-[RL_TH0, RL_TL0])/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) / (65536-[RL_TH0, RL_TL0])/2
若定时器/计数器T0工作在定时器模式2(8位自动重装模式)，如果$C/\overline{T}$=0, 则：
T0工作在1T模式(AUXR.7/T0x12=1)时的输出频率 = (SYSclk) / (256-TH0) / 2
T0工作在12T模式(AUXR.7/T0x12=0)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P3.5/T1管脚被配置为定时器0的时钟输出

AUXR : Auxiliary register

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

AUXR (地址：0x8E)
T0x12:
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:
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

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的C语言声明：

```
sfr     IRC_CLKO     =   0xBB;        //新增加的特殊功能寄存器IRC_CLKO的地址声明
sfr     INT_CLKO     =   0x8F;        //新增加的特殊功能寄存器INT_CLKO的地址声明
sfr     AUXR         =   0x8E;        //特殊功能寄存器AUXR的地址声明
```

特殊功能寄存器IRC_CLKO/INT_CLKO/AUXR的汇编语言声明：

```
IRC_CLKO      EQU    0BBH          ;新增加的特殊功能寄存器IRC_CLKO的地址声明
INT_CLKO      EQU    8FH           ;新增加的特殊功能寄存器INT_CLKO的地址声明
AUXR          EQU    8EH           ;特殊功能寄存器AUXR的地址声明
```

7.4.1 内部R/C时钟输出的测试程序(C程序和汇编程序)

1. C程序：

```
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机的内部R/C时钟输出 ----------------------*/
/* 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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/


sfr IRC_CLKO = 0xbb;            //EN_IRCO - - - DIVIRCO - - -

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

void main()
{
            IRC_CLKO = 0x80;            //1000,0000 P0.0 output clock signal which frequency is SYSclk
//          IRC_CLKO = 0x88;            //1000,1000 P0.0 output clock signal which frequency is SYSclk/2

            while (1);
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 系列单片机的内部R/C时钟输出 -----------------------*/
 /* 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宏晶科技的资料及程序 -----------*/
 /*---------------------------------------------------------------------------------------*/



IRC_CLKO    DATA    0BBH            ;EN_IRCO - - - DIVIRCO - - -

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

            ORG     0000H
            LJMP    MAIN

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

            ORG     0100H
MAIN:
            MOV     SP,#7FH                     ;initial SP
            MOV     IRC_CLKO,       #80H        ;1000,0000 P0.0 output clock signal which frequency is SYSclk
;           MOV     IRC_CLKO,#88H               ;1000,1000
                                                ;P0.0 output clock signal which frequency is SYSclk/2
            SJMP    $

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

            END
```

7.4.2 定时器0的可编程时钟输出的测试程序(C程序和汇编程序)

1. C程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
 /*---------------------------------------------------------------------------------------*/




#include "reg51.h"

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

/* define constants */
#define SYSclk 18432000L
//#define MODE1T                    //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef MODE1T
#define F38_4KHz (65536-SYSclk/2/38400)       //38.4KHz frequency calculation method of 1T mode
#else
#define F38_4KHz (65536-SYSclk/2/12/38400)    //38.4KHz frequency calculation method of 12T mode
#endif

/* define SFR */
sfr AUXR      = 0x8e;                         //Auxiliary register
sfr INT_CLKO = 0x8f;                          //External interrupt enable and clock output control register
sbit T0CLKO   = P3^5;                         //timer0 clock output pin

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

/* main program */
void main()
{
#ifdef MODE1T
        AUXR = 0x80;                          //timer0 work in 1T mode
#endif
        TMOD = 0x00;                          //set timer0 as mode0 (16-bit auto-reload)
        TL0 = F38_4KHz;                       //initial timer0 low byte
        TH0 = F38_4KHz >> 8;                  //initial timer0 high byte
        TR0 = 1;                              //timer0 start running
        INT_CLKO = 0x01;                      //enable timer0 clock output

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

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




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

#ifdef MODE1T
F38_4KHz    EQU 0FF10H      ;38.4KHz frequency calculation method of 1T mode is (65536-18432000/2/38400)
#else
F38_4KHz    EQU 0FFECH  ;38.4KHz frequency calculation method of 12T mode(65536-18432000/2/12/38400)
#endif

;/* define SFR */
            AUXR            DATA    08EH        ;Auxiliary register
            INT_CLKO        DATA    08FH        ;External interrupt enable and clock output control register
            T0CLKO          BIT       P3.5      ;timer0 clock output pin

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

;-----------------------------------------------
;/* main program */
MAIN:
#ifdef MODE1T
            MOV     AUXR,   #80H                        ;timer0 work in 1T mode
#endif
            MOV     TMOD,   #00H                        ;set timer0 as mode0 (16-bit auto-reload)
            MOV     TL0,    #LOW F38_4KHz               ;initial timer0 low byte
            MOV     TH0,    #HIGH F38_4KHz              ;initial timer0 high byte
            SETB    TR0
            MOV     INT_CLKO,       #01H                ;enable timer0 clock output

            SJMP    $

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

7.4.3 定时器1的可编程时钟输出的测试程序(C程序和汇编程序)

1. C程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*----------------------------------------------------------------------------------*/




#include "reg51.h"

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

/* define constants */
#define SYSclk 18432000L
//#define MODE1T                    //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode

#ifdef MODE1T
#define F38_4KHz (65536-SYSclk/2/38400)       //38.4KHz frequency calculation method of 1T mode
#else
#define F38_4KHz (65536-SYSclk/2/12/38400)  //38.4KHz frequency calculation method of 12T mode
#endif

/* define SFR */
sfr AUXR      = 0x8e;                         //Auxiliary register
sfr INT_CLKO = 0x8f;                          //External interrupt enable and clock output control register
sbit T1CLKO   = P3^4;                         //timer1 clock output pin

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

/* main program */
void main()
{
#ifdef MODE1T
   AUXR = 0x40;                               //timer1 work in 1T mode
#endif
   TMOD = 0x00;                               //set timer1 as mode0 (16-bit auto-reload)
   TL1 = F38_4KHz;                            //initial timer1 low byte
   TH1 = F38_4KHz >> 8;                       //initial timer1 high byte
   TR1 = 1;                                   //timer1 start running
   INT_CLKO = 0x02;                           //enable timer1 clock output

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

2. 汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
 /*---------------------------------------------------------------------------------------*/




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

#ifdef MODE1T
F38_4KHz    EQU  0FF10H     ;38.4KHz frequency calculation method of 1T mode is (65536-18432000/2/38400)
#else
F38_4KHz    EQU 0FFECH             ;38.4KHz frequency calculation method of 12T mode (65536-18432000/2/1
2/38400)
#endif

;/* define SFR */
AUXR                DATA      08EH                  ;Auxiliary register
INT_CLKO            DATA      08FH                  ;External interrupt enable and clock output control register
T1CLKO              BIT       P3.4                  ;timer1 clock output pin

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

            ORG       0000H
            LJMP      MAIN
;-----------------------------------------------
;/* main program */
MAIN:
#ifdef MODE1T
            MOV       AUXR,    #40H                             ;timer1 work in 1T mode
#endif
            MOV     TMOD,     #00H                              ;set timer1 as mode0 (16-bit auto-reload)
            MOV     TL1,      #LOW F38_4KHz                     ;initial timer1 low byte
            MOV     TH1,      #HIGH F38_4KHz                    ;initial timer1 high byte
            SETB    TR1
            MOV     INT_CLKO,           #02H                    ;enable timer1 clock output

            SJMP    $
;-----------------------------------------------

            END
```

7.5 古老的Intel 8051单片机定时器0/1应用举例

【例1】 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                          ；}
          SETB    ET1                         ；} 开中断
          ⋮                             ；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                      ；}
       MOV     TH1,#0D8H                      ；} 重新置初值
          ⋮
                                              ；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。STC15F100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2】 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 模拟串口的实现程序

8.1 利用定时器0实现模拟串口的测试程序(C程序和汇编程序)

----定时器0工作在16位自动重装模式

1. C程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机利用定时器0实现模拟串口功能--------*/
/* 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




#include "reg51.h"

//define baudrate const
//BAUD = 256 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
//NOTE: (SYSclk/3/BAUDRATE) must be greater than 98, (RECOMMEND GREATER THAN 110)

//#define BAUD     0xF400          // 1200bps @ 11.0592MHz
//#define BAUD     0xFA00          // 2400bps @ 11.0592MHz
//#define BAUD     0xFD00          // 4800bps @ 11.0592MHz
//#define BAUD     0xFE80          // 9600bps @ 11.0592MHz
//#define BAUD     0xFF40          //19200bps @ 11.0592MHz
//#define BAUD     0xFFA0          //38400bps @ 11.0592MHz

//#define BAUD     0xEC00          // 1200bps @ 18.432MHz
//#define BAUD     0xF600          // 2400bps @ 18.432MHz
//#define BAUD     0xFB00          // 4800bps @ 18.432MHz
//#define BAUD     0xFD80          // 9600bps @ 18.432MHz
//#define BAUD     0xFEC0          //19200bps @ 18.432MHz
#define BAUD       0xFF60          //38400bps @ 18.432MHz

//#define BAUD     0xE800          // 1200bps @ 22.1184MHz
//#define BAUD     0xF400          // 2400bps @ 22.1184MHz
//#define BAUD     0xFA00          // 4800bps @ 22.1184MHz
//#define BAUD     0xFD00          // 9600bps @ 22.1184MHz
//#define BAUD     0xFE80          //19200bps @ 22.1184MHz
//#define BAUD     0xFF40          //38400bps @ 22.1184MHz
//#define BAUD     0xFF80          //57600bps @ 22.1184MHz
```

```
sfr AUXR = 0x8E;
sbit RXB = P3^0;                                //define UART TX/RX port
sbit TXB = P3^1;

typedef bit BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

BYTE TBUF,RBUF;
BYTE TDAT,RDAT;
BYTE TCNT,RCNT;
BYTE TBIT,RBIT;
BOOL TING,RING;
BOOL TEND,REND;

void UART_INIT();

BYTE t, r;
BYTE buf[16];

void main()
{
        TMOD = 0x00;                            //timer0 in 16-bit auto reload mode
        AUXR = 0x80;                            //timer0 working at 1T mode
        TL0 = BAUD;
        TH0 = BAUD>>8;                          //initial timer0 and set reload value
        TR0 = 1;                                //tiemr0 start running
        ET0 = 1;                                //enable timer0 interrupt
        PT0 = 1;                                //improve timer0 interrupt priority
        EA = 1;                                 //open global interrupt switch

        UART_INIT();

        while (1)
        {                                       //user's function
                if (REND)
                {
                        REND = 0;
                        buf[r++ & 0x0f] = RBUF;
                }
                if (TEND)
                {
                        if (t != r)
                        {
                                TEND = 0;
                                TBUF = buf[t++ & 0x0f];
                                TING = 1;
                        }
                }
        }
}
```

```
//-----------------------------------------
//Timer interrupt routine for UART

void tm0() interrupt 1 using 1
{
        if (RING)
        {
                if (--RCNT == 0)
                {
                        RCNT = 3;           //reset send baudrate counter
                        if (--RBIT == 0)
                        {
                                RBUF = RDAT;              //save the data to RBUF
                                RING = 0;                 //stop receive
                                REND = 1;                 //set receive completed flag
                        }
                        else
                        {
                                RDAT >>= 1;
                                if (RXB) RDAT |= 0x80;    //shift RX data to RX buffer
                        }
                }
        }
        else if (!RXB)
        {
                RING = 1;                 //set start receive flag
                RCNT = 4;                 //initial receive baudrate counter
                RBIT = 9;                 //initial receive bit number (8 data bits + 1 stop bit)
        }

        if (--TCNT == 0)
        {
                TCNT = 3;                 //reset send baudrate counter
                if (TING)                 //judge whether sending
                {
                        if (TBIT == 0)
                        {
                                TXB = 0;          //send start bit
                                TDAT = TBUF;      //load data from TBUF to TDAT
                                TBIT = 9;         //initial send bit number (8 data bits + 1 stop bit)
                        }
                        else
                        {
                                TDAT >>= 1;             //shift data to CY
                                if (--TBIT == 0)
                                {
                                         TXB = 1;
                                        TING = 0;           //stop send
                                        TEND = 1;            //set send completed flag
                                }
                                else
                                {
                                        TXB = CY;            //write CY to TX port
                                }
                        }
                }
        }
}

//-----------------------------------------
//initial UART module variable

void UART_INIT()
{
        TING = 0;
        RING = 0;
        TEND = 1;
        REND = 0;
        TCNT = 0;
        RCNT = 0;
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机利用定时器0实现模拟串口功能--------*/
/* 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宏晶科技的资料及程序 -----------*/
/*----------------------------------------------------------------------------------*/




;-----------------------------------------
;define baudrate const
;BAUD = 65536 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
;NOTE: (SYSclk/3/BAUDRATE) must be greater than 75, (RECOMMEND GREATER THAN 100)

;BAUD    EQU     0F400H              ; 1200bps @ 11.0592MHz
;BAUD    EQU     0FA00H              ; 2400bps @ 11.0592MHz
;BAUD    EQU     0FD00H              ; 4800bps @ 11.0592MHz
;BAUD    EQU     0FE80H              ; 9600bps @ 11.0592MHz
;BAUD    EQU     0FF40H              ;19200bps @ 11.0592MHz
;BAUD    EQU     0FFA0H              ;38400bps @ 11.0592MHz
;BAUD    EQU     0FFC0H              ;57600bps @ 11.0592MHz

;BAUD    EQU     0EC00H              ; 1200bps @ 18.432MHz
;BAUD    EQU     0F600H              ; 2400bps @ 18.432MHz
;BAUD    EQU     0FB00H              ; 4800bps @ 18.432MHz
;BAUD    EQU     0FD80H              ; 9600bps @ 18.432MHz
;BAUD    EQU     0FEC0H              ;19200bps @ 18.432MHz
;BAUD    EQU     0FF60H              ;38400bps @ 18.432MHz
BAUD     EQU     0FF95H              ;57600bps @ 18.432MHz

;BAUD    EQU     0E800H              ; 1200bps @ 22.1184MHz
;BAUD    EQU     0F400H              ; 2400bps @ 22.1184MHz
;BAUD    EQU     0FA00H              ; 4800bps @ 22.1184MHz
;BAUD    EQU     0FD00H              ; 9600bps @ 22.1184MHz
;BAUD    EQU     0FE80H              ;19200bps @ 22.1184MHz
;BAUD    EQU     0FF40H              ;38400bps @ 22.1184MHz
;BAUD    EQU     0FF80H              ;57600bps @ 22.1184MHz
```

```
;-----------------------------------------
;define UART TX/RX port

RXB     BIT     P3.0
TXB     BIT     P3.1

;-----------------------------------------
;define SFR

AUXR    DATA    8EH

;-----------------------------------------
;define UART module variable

TBUF      DATA    08H               ;(R0) ready send data buffer   (USER WRITE ONLY)
RBUF      DATA    09H               ;(R1) received data buffer     (UAER READ ONLY)
TDAT      DATA    0AH               ;(R2) sending data buffer      (RESERVED FOR UART MODULE)
RDAT      DATA    0BH               ;(R3) receiving data buffer    (RESERVED FOR UART MODULE)
TCNT      DATA    0CH               ;(R4) send baudrate counter    (RESERVED FOR UART MODULE)
RCNT      DATA    0DH               ;(R5) receive baudrate counter (RESERVED FOR UART MODULE)
TBIT      DATA    0EH               ;(R6) send bit counter         (RESERVED FOR UART MODULE)
RBIT      DATA    0FH               ;(R7) receive bit counter      (RESERVED FOR UART MODULE)

TING      BIT     20H.0             ; sending flag   (USER WRITE "1" TO TRIGGER SEND DATA, CLEAR BY
MODULE)
RING      BIT     20H.1             ; receiving flag (RESERVED FOR UART MODULE)
TEND      BIT     20H.2             ; sent flag      (SET BY MODULE AND SHOULD USER CLEAR)
REND      BIT     20H.3             ; received flag  (SET BY MODULE AND SHOULD USER CLEAR)

RPTR      DATA    21H               ;circular queue read pointer
WPTR      DATA    22H               ;circular queue write pointer
BUFFER DATA    23H                  ;circular queue buffer (16 bytes)

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

      ORG     0000H
      LJMP    RESET

;-----------------------------------------
;Timer0 interrupt routine for UART

      ORG     000BH

      PUSH    ACC                   ;4 save ACC
      PUSH    PSW                   ;4 save PSW
      MOV     PSW, #08H             ;3 using register group 1
L_UARTSTART:
;-------------------
```

```
        JB      RING,   L_RING          ;4 judge whether receiving
        JB      RXB,    L_REND          ;  check start signal
L_RSTART:
        SETB    RING                    ;  set start receive flag
        MOV     R5,     #4              ;  initial receive baudrate counter
        MOV     R7,     #9              ;  initial receive bit number (8 data bits + 1 stop bit)
        SJMP    L_REND                  ;  end this time slice
L_RING:
        DJNZ    R5,     L_REND          ;4 judge whether sending
        MOV     R5,     #3              ;2 reset send baudrate counter
L_RBIT:
        MOV     C,      RXB             ;3 read RX port data
        MOV     A,      R3              ;1 and shift it to RX buffer
        RRC     A                       ;1
        MOV     R3,     A               ;2
        DJNZ    R7,     L_REND          ;4 judge whether the data have receive completed
L_RSTOP:
        RLC     A                       ;  shift out stop bit
        MOV     R1,     A               ;  save the data to RBUF
        CLR     RING                    ;  stop receive
        SETB    REND                    ;  set receive completed flag
L_REND:
;-------------------
L_TING:
        DJNZ    R4,     L_TEND          ;4 check send baudrate counter
        MOV     R4,     #3              ;2 reset it
        JNB     TING,   L_TEND          ;4 judge whether sending
        MOV     A,      R6              ;1 detect the sent bits
        JNZ     L_TBIT                  ;3 "0" means start bit not sent
L_TSTART:
        CLR     TXB                     ;  send start bit
        MOV     TDAT,   R0              ;  load data from TBUF to TDAT
        MOV     R6,     #9              ;  initial send bit number (8 data bits + 1 stop bit)
        JMP     L_TEND                  ;  end this time slice
L_TBIT:
        MOV     A,      R2              ;1 read data in TDAT
        SETB    C                       ;1 shift in stop bit
        RRC     A                       ;1 shift data to CY
        MOV     R2,     A               ;2 update TDAT
        MOV     TXB,    C               ;4 write CY to TX port
        DJNZ    R6,     L_TEND          ;4 judge whether the data have send completed
L_TSTOP:
        CLR     TING                    ;  stop send
        SETB    TEND                    ;  set send completed flag
L_TEND:
;-------------------
L_UARTEND:
        POP     PSW                     ;3 restore PSW
        POP     ACC                     ;3 restore ACC
        RETI                            ;4 (69)
```

```
;-----------------------------------------
;initial UART module variable

UART_INIT:
            CLR     TING
            CLR     RING
            SETB    TEND
            CLR     REND
            CLR     A
            MOV     TCNT,   A
            MOV     RCNT,   A
            RET

;-----------------------------------------
;main program entry

RESET:
            MOV     R0,     #7FH                ;clear RAM
            CLR     A
            MOV     @R0,    A
            DJNZ    R0,     $-1
            MOV     SP,     #7FH                ;initial SP
;-------------------
;system initial
            MOV     TMOD,   #00H                ;timer0 in 16-bit auto reload mode
            MOV     AUXR,   #80H                ;timer0 working at 1T mode
            MOV     TL0,    #LOW BAUD           ;initial timer0 and
            MOV     TH0,    #HIGH BAUD          ;set reload value
            SETB    TR0                         ;tiemr0 start running
            SETB    ET0                         ;enable timer0 interrupt
            SETB    PT0                         ;improve timer0 interrupt priority
            SETB    EA                          ;open global interrupt switch
            LCALL   UART_INIT
;-----------------------------------------
MAIN:
            JNB     REND,   CHECKREND           ;if (REND)
            CLR     REND                        ;{
            MOV     A,      RPTR                ;           REND = 0;
            INC     RPTR                        ;           BUFFER[RPTR++ & 0xf] = RBUF;
            ANL     A,      #0FH                ;}
            ADD     A,      #BUFFER             ;
            MOV     R0,     A                   ;
            MOV     @R0,    RBUF                ;
```

```
CHECKREND:
          JNB       TEND,     MAIN                ;if (TEND)
          MOV       A,        RPTR                ;{
          XRL       A,        WPTR                ;          if (WPTR != REND)
          JZ        MAIN                          ;          {
          CLR       TEND                          ;                    TEND = 0;
          MOV       A,        WPTR                ;                    TBUF = BUFFER[WPTR++ & 0xf];
          INC       WPTR                          ;                    TING = 1;
          ANL       A,        #0FH                ;          }
          ADD       A,        #BUFFER             ;}
          MOV       R0,       A                   ;
          MOV       TBUF,     @R0                 ;
          SETB      TING                          ;
          SJMP      MAIN

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

          END
```

8.2 利用定时器1实现模拟串口的测试程序(C程序和汇编程序)

----定时器1工作在16位自动重装模式

1. C程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 系列单片机利用定时器1实现模拟串口功能--------*/
/* 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/



#include "reg51.h"

//define baudrate const
//BAUD = 256 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
//NOTE: (SYSclk/3/BAUDRATE) must be greater than 98, (RECOMMEND GREATER THAN 110)

//#define BAUD     0xF400            // 1200bps @ 11.0592MHz
//#define BAUD     0xFA00            // 2400bps @ 11.0592MHz
//#define BAUD     0xFD00            // 4800bps @ 11.0592MHz
//#define BAUD     0xFE80            // 9600bps @ 11.0592MHz
//#define BAUD     0xFF40            //19200bps @ 11.0592MHz
//#define BAUD     0xFFA0            //38400bps @ 11.0592MHz

//#define BAUD     0xEC00            // 1200bps @ 18.432MHz
//#define BAUD     0xF600            // 2400bps @ 18.432MHz
//#define BAUD     0xFB00            // 4800bps @ 18.432MHz
//#define BAUD     0xFD80            // 9600bps @ 18.432MHz
//#define BAUD     0xFEC0            //19200bps @ 18.432MHz
#define BAUD       0xFF60            //38400bps @ 18.432MHz

//#define BAUD     0xE800            // 1200bps @ 22.1184MHz
//#define BAUD     0xF400            // 2400bps @ 22.1184MHz
//#define BAUD     0xFA00            // 4800bps @ 22.1184MHz
//#define BAUD     0xFD00            // 9600bps @ 22.1184MHz
//#define BAUD     0xFE80            //19200bps @ 22.1184MHz
//#define BAUD     0xFF40            //38400bps @ 22.1184MHz
//#define BAUD     0xFF80            //57600bps @ 22.1184MHz
```

```
sfr AUXR = 0x8E;
sbit RXB = P3^0;                                //define UART TX/RX port
sbit TXB = P3^1;

typedef bit BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

BYTE TBUF,RBUF;
BYTE TDAT,RDAT;
BYTE TCNT,RCNT;
BYTE TBIT,RBIT;
BOOL TING,RING;
BOOL TEND,REND;

void UART_INIT();

BYTE t, r;
BYTE buf[16];

void main()
{
        TMOD = 0x00;                            //timer1 in 16-bit auto reload mode
        AUXR = 0x40;                            //timer1 working at 1T mode
        TL1 = BAUD;
        TH1 = BAUD>>8;                          //initial timer1 and set reload value
        TR1 = 1;                                //tiemr1 start running
        ET1 = 1;                                //enable timer1 interrupt
        PT1 = 1;                                //improve timer1 interrupt priority
        EA = 1;                                 //open global interrupt switch

        UART_INIT();
        while (1)
        {                       //user's function
                if (REND)
                {
                        REND = 0;
                        buf[r++ & 0x0f] = RBUF;
                }
                if (TEND)
                {
                        if (t != r)
                        {
                                TEND = 0;
                                TBUF = buf[t++ & 0x0f];
                                TING = 1;
                        }
                }
        }
}
```

```
//-----------------------------------------
//Timer interrupt routine for UART

void tm1() interrupt 3 using 1
{
            if (RING)
            {
                        if (--RCNT == 0)
                        {
                                    RCNT = 3;                   //reset send baudrate counter
                                    if (--RBIT == 0)
                                    {
                                                RBUF = RDAT;      //save the data to RBUF
                                                RING = 0;               //stop receive
                                                REND = 1;              //set receive completed flag
                                    }
                                    else
                                    {
                                                RDAT >>= 1;
                                                if (RXB) RDAT |= 0x80;        //shift RX data to RX buffer
                                    }
                        }
            }
            else if (!RXB)
            {
                        RING = 1;                       //set start receive flag
                        RCNT = 4;                       //initial receive baudrate counter
                        RBIT = 9;                      //initial receive bit number (8 data bits + 1 stop bit)
            }

            if (--TCNT == 0)
            {
                        TCNT = 3;                       //reset send baudrate counter
                        if (TING)                      //judge whether sending
                        {
                                    if (TBIT == 0)
                                    {
                                                TXB = 0;               //send start bit
                                                TDAT = TBUF;      //load data from TBUF to TDAT
                                                TBIT = 9;              //initial send bit number (8 data bits + 1 stop bit)
                                    }
                                    else
                                    {
                                                TDAT >>= 1;             //shift data to CY
                                                if (--TBIT == 0)
                                                {
                                                        TXB = 1;
                                                        TING = 0;           //stop send
                                                        TEND = 1;           //set send completed flag
                                                }
                                                else
                                                {
                                                        TXB = CY;           //write CY to TX port
                                                }
                                    }
                        }
            }
}

//-----------------------------------------
//initial UART module variable

void UART_INIT()
{
            TING = 0;
            RING = 0;
            TEND = 1;
            REND = 0;
            TCNT = 0;
            RCNT = 0;
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited --------------------------------------*/
/* --- 演示STC 15 系列单片机利用定时器1实现模拟串口功能--------*/
/* 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/
```

```
;-----------------------------------------
;define baudrate const
;BAUD = 65536 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
;NOTE: (SYSclk/3/BAUDRATE) must be greater than 75, (RECOMMEND GREATER THEN 100)

;BAUD    EQU    0F400H          ; 1200bps @ 11.0592MHz
;BAUD    EQU    0FA00H          ; 2400bps @ 11.0592MHz
;BAUD    EQU    0FD00H          ; 4800bps @ 11.0592MHz
;BAUD    EQU    0FE80H          ; 9600bps @ 11.0592MHz
;BAUD    EQU    0FF40H          ;19200bps @ 11.0592MHz
;BAUD    EQU    0FFA0H          ;38400bps @ 11.0592MHz
;BAUD    EQU    0FFC0H          ;57600bps @ 11.0592MHz

;BAUD    EQU    0EC00H          ; 1200bps @ 18.432MHz
;BAUD    EQU    0F600H          ; 2400bps @ 18.432MHz
;BAUD    EQU    0FB00H          ; 4800bps @ 18.432MHz
;BAUD    EQU    0FD80H          ; 9600bps @ 18.432MHz
;BAUD    EQU    0FEC0H          ;19200bps @ 18.432MHz
;BAUD    EQU    0FF60H          ;38400bps @ 18.432MHz
BAUD    EQU    0FF95H           ;57600bps @ 18.432MHz

;BAUD    EQU    0E800H          ; 1200bps @ 22.1184MHz
;BAUD    EQU    0F400H          ; 2400bps @ 22.1184MHz
;BAUD    EQU    0FA00H          ; 4800bps @ 22.1184MHz
;BAUD    EQU    0FD00H          ; 9600bps @ 22.1184MHz
;BAUD    EQU    0FE80H          ;19200bps @ 22.1184MHz
;BAUD    EQU    0FF40H          ;38400bps @ 22.1184MHz
;BAUD    EQU    0FF80H          ;57600bps @ 22.1184MHz
```

```
;-----------------------------------------
;define UART TX/RX port

RXB     BIT     P3.0
TXB     BIT     P3.1

;-----------------------------------------
;define SFR

AUXR    DATA    8EH

;-----------------------------------------
;define UART module variable

TBUF    DATA    08H             ;(R0) ready send data buffer   (USER WRITE ONLY)
RBUF    DATA    09H             ;(R1) received data buffer     (UAER READ ONLY)
TDAT    DATA    0AH             ;(R2) sending data buffer      (RESERVED FOR UART MODULE)
RDAT    DATA    0BH             ;(R3) receiving data buffer    (RESERVED FOR UART MODULE)
TCNT    DATA    0CH             ;(R4) send baudrate counter    (RESERVED FOR UART MODULE)
RCNT    DATA    0DH             ;(R5) receive baudrate counter (RESERVED FOR UART MODULE)
TBIT    DATA    0EH             ;(R6) send bit counter         (RESERVED FOR UART MODULE)
RBIT    DATA    0FH             ;(R7) receive bit counter      (RESERVED FOR UART MODULE)

TING    BIT     20H.0           ;sending flag(USER WRITE"1"TO TRIGGER SEND DATA,CLEAR BY MOD-
ULE)
RING    BIT     20H.1           ; receiving flag (RESERVED FOR UART MODULE)
TEND    BIT     20H.2           ; sent flag      (SET BY MODULE AND SHOULD USER CLEAR)
REND    BIT     20H.3           ; received flag  (SET BY MODULE AND SHOULD USER CLEAR)

RPTR    DATA    21H             ;circular queue read pointer
WPTR    DATA    22H             ;circular queue write pointer
BUFFER DATA     23H             ;circular queue buffer (16 bytes)

;-----------------------------------------
        ORG     0000H
        LJMP    RESET

;-----------------------------------------
;Timer1 interrupt routine for UART

        ORG     001BH

        PUSH    ACC                     ;4 save ACC
        PUSH    PSW                     ;4 save PSW
        MOV     PSW,    #08H            ;3 using register group 1
L_UARTSTART:
;-------------------
        JB      RING,   L_RING          ;4 judge whether receiving
        JB      RXB,    L_REND          ;  check start signal
```

```
L_RSTART:
          SETB    RING                    ;  set start receive flag
          MOV     R5,       #4            ;  initial receive baudrate counter
          MOV     R7,       #9            ;  initial receive bit number (8 data bits + 1 stop bit)
          SJMP    L_REND                  ;  end this time slice
L_RING:
          DJNZ    R5,       L_REND        ;4 judge whether sending
          MOV     R5,       #3            ;2 reset send baudrate counter
L_RBIT:
          MOV     C,        RXB           ;3 read RX port data
          MOV     A,        R3            ;1 and shift it to RX buffer
          RRC     A                       ;1
          MOV     R3,       A             ;2
          DJNZ    R7,       L_REND        ;4 judge whether the data have receive completed
L_RSTOP:
          RLC     A                       ;  shift out stop bit
          MOV     R1,       A             ;  save the data to RBUF
          CLR     RING                    ;  stop receive
          SETB    REND                    ;  set receive completed flag
L_REND:
;------------------
L_TING:
          DJNZ    R4,       L_TEND        ;4 check send baudrate counter
          MOV     R4,       #3            ;2 reset it
          JNB     TING,     L_TEND        ;4 judge whether sending
          MOV     A,        R6            ;1 detect the sent bits
          JNZ     L_TBIT                  ;3 "0" means start bit not sent
L_TSTART:
          CLR     TXB                     ;  send start bit
          MOV     TDAT,     R0            ;  load data from TBUF to TDAT
          MOV     R6,       #9            ;  initial send bit number (8 data bits + 1 stop bit)
          JMP     L_TEND                  ;  end this time slice
L_TBIT:
          MOV     A,        R2            ;1 read data in TDAT
          SETB    C                       ;1 shift in stop bit
          RRC     A                       ;1 shift data to CY
          MOV     R2,       A             ;2 update TDAT
          MOV     TXB,      C             ;4 write CY to TX port
          DJNZ    R6,       L_TEND        ;4 judge whether the data have send completed
L_TSTOP:
          CLR     TING                    ;  stop send
          SETB    TEND                    ;  set send completed flag
L_TEND:
;------------------
L_UARTEND:
          POP     PSW                     ;3 restore PSW
          POP     ACC                     ;3 restore ACC
          RETI                            ;4 (69)
```

```
;-----------------------------------------
;initial UART module variable

UART_INIT:
            CLR     TING
            CLR     RING
            SETB    TEND
            CLR     REND
            CLR     A
            MOV     TCNT,A
            MOV     RCNT,A
            RET

;-----------------------------------------
;main program entry

RESET:
            MOV     R0,     #7FH            ;clear RAM
            CLR     A
            MOV     @R0,    A
            DJNZ    R0,     $-1
            MOV     SP,     #7FH            ;initial SP
;------------------
;system initial
            MOV     TMOD,   #00H                    ;timer1 in 16-bit auto reload mode
            MOV     AUXR,   #40H                    ;timer1 working at 1T mode
            MOV     TL1,    #LOW BAUD               ;initial timer1 and
            MOV     TH1,    #HIGH BAUD              ;set reload value
            SETB    TR1                             ;tiemr1 start running
            SETB    ET1                             ;enable timer1 interrupt
            SETB    PT1                             ;improve timer1 interrupt priority
            SETB    EA                              ;open global interrupt switch
            LCALL   UART_INIT
;-----------------------------------------
MAIN:
            JNB     REND,   CHECKREND               ;if (REND)
            CLR     REND                            ;{
            MOV     A,      RPTR                    ;           REND = 0;
            INC     RPTR                            ;           BUFFER[RPTR++ & 0xf] = RBUF;
            ANL     A,      #0FH                    ;}
            ADD     A,      #BUFFER                 ;
            MOV     R0,     A                       ;
            MOV     @R0,    RBUF                    ;
```

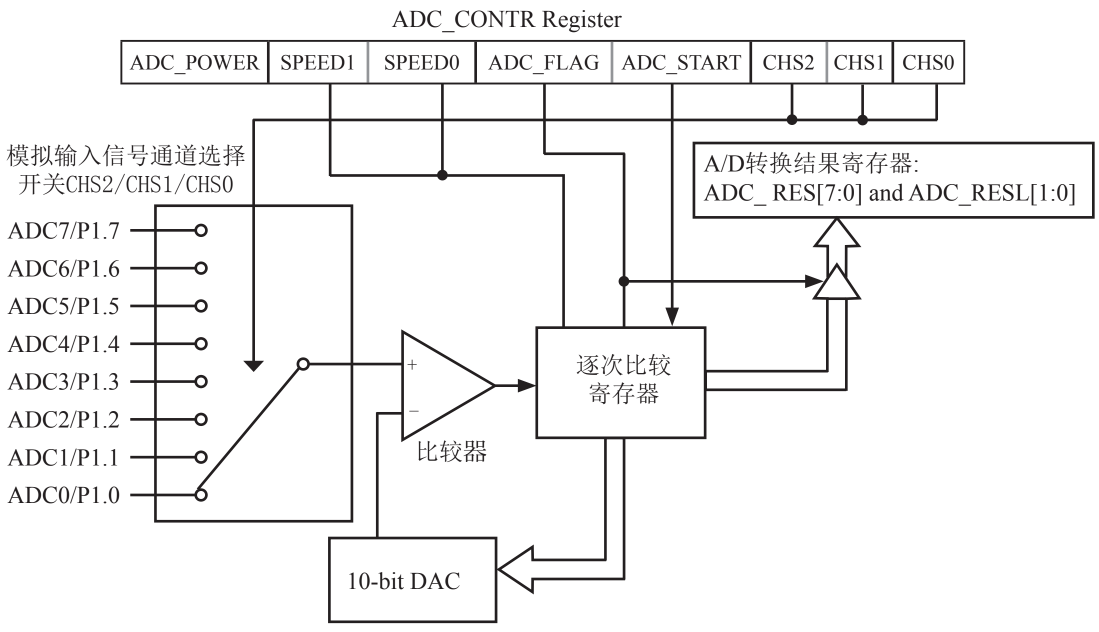
```
CHECKREND:
        JNB     TEND,   MAIN            ;if (TEND)
        MOV     A,      RPTR                    ;{
        XRL     A,      WPTR            ;       if (WPTR != REND)
        JZ      MAIN                    ;       {
        CLR     TEND                    ;               TEND = 0;
        MOV     A,      WPTR            ;               TBUF = BUFFER[WPTR++ & 0xf];
        INC     WPTR                    ;               TING = 1;
        ANL     A,      #0FH                    ;       }
        ADD     A,      #BUFFER                 ;}
         MOV    R0,     A                       ;
        MOV     TBUF,   @R0             ;
        SETB    TING                    ;
        SJMP    MAIN

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

        END
```

第9章 STC15F204EA系列单片机的A/D转换器

----STC15F100系列单片机无A/D转换功能，但STC15F204EA有A/D转换功能

9.1 A/D转换器的结构

STC15F204EA系列单片机ADC(A/D转换器)的结构如下图所示。



ADC_RES[7:0]

ADC_B9	ADC_B8	ADC_B7	ADC_B6	ADC_B5	ADC_B4	ADC_B3	ADC_B2

ADC_RESL[1:0]

-	-	-	-	-	-	ADC_B1	ADC_B0

STC15F204EA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_RES和ADC_RESL)以及ADC_CONTR构成。

STC15F204EA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寄存器，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_RES和ADC_RESL，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。

如果取完整的10位结果，按下面公式计算:

$$\boxed{\text{10-bit A/D Conversion Result:(ADC\_RES[7:0], ADC\_RESL[1:0])} = 1024 \text{ x } \frac{\text{Vin}}{\text{Vcc}}}$$

如果只取高8位结果，按下面公式计算:

$$\boxed{\text{8-bit A/D Conversion Result:(ADC\_RES[7:0])} = 256 \text{ x } \frac{\text{Vin}}{\text{Vcc}}}$$

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STC15F204EA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								复位值
			MSB							LSB	
P1ASF	P1 Analog Function Configure register	9DH	P17ASF	P16ASF	P15ASF	P14ASF	P13ASF	P12ASF	P11ASF	P10ASF	0000 0000B
ADC_CONTR	ADC Control Register	BCH	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0	0000 0000B
ADC_RES	ADC Result high	BDH									0000 0000B
ADC_RESL	ADC Result low	BEH									0000 0000B
IE	Interrupt Enable	A8H	EA	ELVD	EADC	-	ET1	EX1	ET0	EX0	000x 0000B
IP	Interrupt Priority Low	B8H	-	PLVD	PADC	-	PT1	PX1	PT0	PX0	x00x 0000B

1. P1口模拟功能控制寄存器P1ASF

STC15F204EA系列单片机的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300KHz(30万次/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P1口可继续作为I/O口使用(建议只作为输入)。需作为A/D使用的口需先将P1ASF特殊功能寄存器中的相应位置为‘1’，将相应的口设置为模拟功能。P1ASF寄存器的格式如下：

P1ASF : P1口模拟功能控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1ASF	9DF	name	P17ASF	P16ASF	P15ASF	P14ASF	P13ASF	P12ASF	P11ASF	P10ASF

P1ASF[7:0]	P1.x的功能	其中P1ASF寄存器地址为：[9DH](不能够进行位寻址)
P1ASF.0 = 1	P1.0口作为模拟功能A/D使用	
P1ASF.1 = 1	P1.1口作为模拟功能A/D使用	
P1ASF.2 = 1	P1.2口作为模拟功能A/D使用	
P1ASF.3 = 1	P1.3口作为模拟功能A/D使用	
P1ASF.4 = 1	P1.4口作为模拟功能A/D使用	
P1ASF.5 = 1	P1.5口作为模拟功能A/D使用	
P1ASF.6 = 1	P1.6口作为模拟功能A/D使用	
P1ASF.7 = 1	P1.7口作为模拟功能A/D使用	

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	BCH	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ADC 电源；

1：打开A/D转换器电源.

建议进入空闲模式和掉电模式前，将ADC电源关闭，即ADC_POWER =0，可降低功耗。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,如能将定时器/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	90个时钟周期转换一次，CPU工作频率21MHz时，A/D转换速度约300KHz
1	0	180个时钟周期转换一次
0	1	360个时钟周期转换一次
0	0	540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

3. A/DC转换结果寄存器ADC_RES、ADC_RESL

特殊功能寄存器ADC_RES和ADC_RESL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_RES	BDh	A/D转换结果寄存器高8位	ADC_RES9	ADC_RES8	ADC_RES7	ADC_RES6	ADC_RES5	ADC_RES4	ADC_RES3	ADC_RES2
ADC_RESL	BEh	A/D转换结果寄存器低2位	-	-	-	-	-	-	ADC_RES1	ADC_RES0

STC15F204EA系列单片机的10位A/D转换结果的高8位存放在ADC_RES中，低2位存放在ADC_RESL的低2位中。

如果用户需取完整10位结果，按下面公式计算：

$$\boxed{\text{10-bit A/D Conversion Result:(ADC\_RES[7:0], ADC\_RESL[1:0])} = 1024 \times \frac{Vin}{Vcc}}$$

如果用户只需取高8位结果，按下面公式计算：

$$\boxed{\text{8-bit A/D Conversion Result:(ADC\_RES[7:0])} = 256 \times \frac{Vin}{Vcc}}$$

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. 中断允许寄存器IE

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	ELVD	EADC	-	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：A/D转换中断允许位，EADC=1，允许A/D转换中断，EADC=0，禁止A/D转换中断。

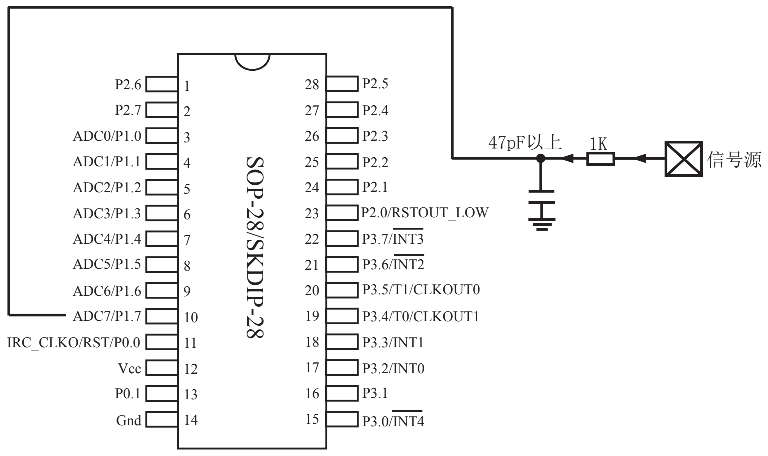
5. 中断优先级控制寄存器IP

IP：中断优先级控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IP	B8H	name	-	PLVD	PADC	-	PT1	PX1	PT0	PX0

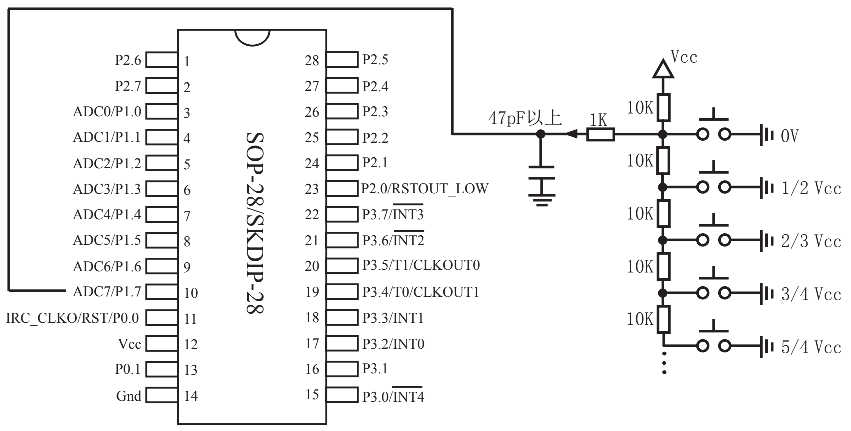
PADC ： A/D转换中断优先级控制位。PADC=1，A/D转换中断定义为高优先级中断；PADC=0，A/D转换中断定义为低优先级中断。

9.3 A/D转换典型应用线路

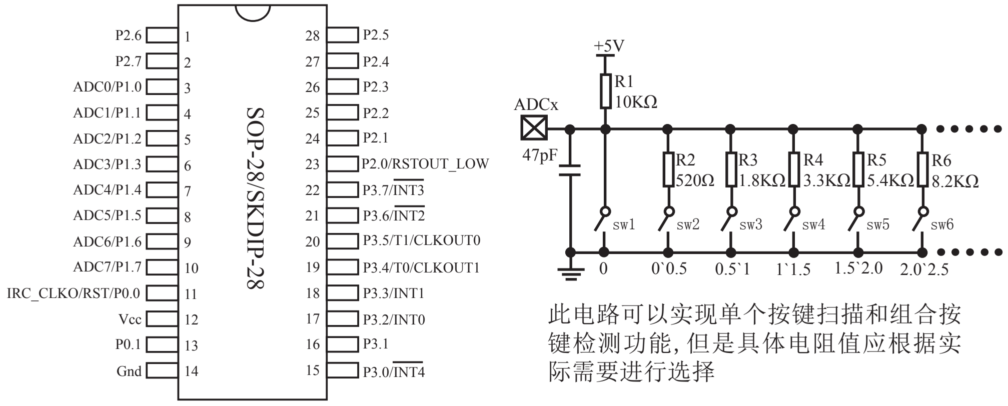


A/D转换在P1口，P1.0 - P1.7共8路

9.4 A/D做按键扫描应用线路图

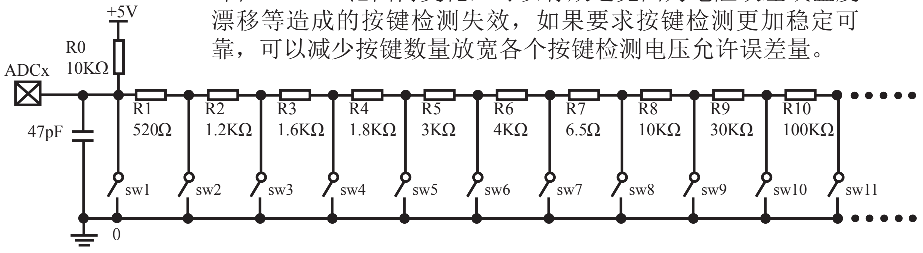


A/D转换在P1口，P1.0 - P1.7共8路



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

STC15F204EA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 International Limited --------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
 /*---------------------------------------------------------------------------------*/




/*使用了软件模拟串口输出*/

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

typedef bit BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

//define baudrate const
//BAUD = 256 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
//NOTE: (SYSclk/3/BAUDRATE) must be greater then 98, (RECOMMEND GREATER THEN 110)

//#define BAUD  0xF400          // 1200bps @ 11.0592MHz
//#define BAUD  0xFA00          // 2400bps @ 11.0592MHz
//#define BAUD  0xFD00          // 4800bps @ 11.0592MHz
//#define BAUD  0xFE80          // 9600bps @ 11.0592MHz
//#define BAUD  0xFF40          //19200bps @ 11.0592MHz
//#define BAUD  0xFFA0          //38400bps @ 11.0592MHz

//#define BAUD  0xEC00          // 1200bps @ 18.432MHz
//#define BAUD  0xF600          // 2400bps @ 18.432MHz
//#define BAUD  0xFB00          // 4800bps @ 18.432MHz
//#define BAUD  0xFD80          // 9600bps @ 18.432MHz
//#define BAUD  0xFEC0          //19200bps @ 18.432MHz
#define BAUD    0xFF60          //38400bps @ 18.432MHz
```

```
//#define BAUD  0xE800            // 1200bps @ 22.1184MHz
//#define BAUD  0xF400            // 2400bps @ 22.1184MHz
//#define BAUD  0xFA00            // 4800bps @ 22.1184MHz
//#define BAUD  0xFD00            // 9600bps @ 22.1184MHz
//#define BAUD  0xFE80            //19200bps @ 22.1184MHz
//#define BAUD  0xFF40            //38400bps @ 22.1184MHz
//#define BAUD  0xFF80            //57600bps @ 22.1184MHz

sfr AUXR = 0x8E;
sbit RXB  = P3^0;                 //define UART TX/RX port
sbit TXB  = P3^1;

/*Declare SFR associated with the ADC */
sfr ADC_CONTR   =   0xBC;         //ADC control register
sfr ADC_RES     =   0xBD;         //ADC hight 8-bit result register
sfr ADC_LOW2    =   0xBE;         //ADC low 2-bit result register
sfr P1ASF       =   0x9D;         //P1 secondary function control register

/*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            //540 clocks
#define ADC_SPEEDL      0x20            //360 clocks
#define ADC_SPEEDH      0x40            //180 clocks
#define ADC_SPEEDHH     0x60            //90 clocks

void InitUart();
void SendData(BYTE dat);
void Delay(WORD n);
void InitADC();

BYTE TBUF,RBUF;
BYTE TDAT,RDAT;
BYTE TCNT,RCNT;
BYTE TBIT,RBIT;
BOOL TING,RING;
BOOL TEND,REND;
BYTE ch = 0;                      //ADC channel NO.
```

```
void main()
{
            InitUart();                       //Init UART, use to show ADC result
            InitADC();                        //Init ADC sfr
            TMOD = 0x00;                      //timer0 in 16-bit auto reload mode
            AUXR = 0x80;                      //timer0 working at 1T mode
            TL0 = BAUD;
            TH0 = BAUD>>8;                    //initial timer0 and set reload value
            TR0 = 1;                          //tiemr0 start running
            IE = 0xa0;                        //Enable ADC interrupt and Open master interrupt switch
            ET0 = 1;                          //enable timer0 interrupt
            PT0 = 1;                          //improve timer0 interrupt priority
                                              //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_RES);                //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()
{
            P1ASF = 0xff;                     //Set all P1 as analog input port
            ADC_RES = 0;                      //Clear previous result
            ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ADC_START | ch;
            Delay(2);                         //ADC power-on delay and Start A/D conversion
}
```

```
/*----------------------------
Software delay function
----------------------------*/
void Delay(WORD n)
{
        WORD x;

        while (n--)
        {
                x = 5000;
                while (x--);
        }
}

//-----------------------------------------
//Timer interrupt routine for UART

void tm0() interrupt 1 using 1
{
        if (RING)
        {
                if (--RCNT == 0)
                {
                        RCNT = 3;               //reset send baudrate counter
                        if (--RBIT == 0)
                        {
                                RBUF = RDAT;            //save the data to RBUF
                                RING = 0;               //stop receive
                                REND = 1;               //set receive completed flag
                        }
                        else
                        {
                                RDAT >>= 1;
                                if (RXB) RDAT |= 0x80;  //shift RX data to RX buffer
                        }
                }
        }
        else if (!RXB)
        {
                RING = 1;               //set start receive flag
                RCNT = 4;               //initial receive baudrate counter
                RBIT = 9;               //initial receive bit number (8 data bits + 1 stop bit)
        }
        if (--TCNT == 0)
        {
                TCNT = 3;                       //reset send baudrate counter
                if (TING)                       //judge whether sending
                {
                        if (TBIT == 0)
                        {
                                TXB = 0;                //send start bit
                                TDAT = TBUF;            //load data from TBUF to TDAT
                                TBIT = 9;               //initial send bit number (8 data bits + 1 stop bit)
                        }
                        else
                        {
                                TDAT >>= 1;             //shift data to CY
                                if (--TBIT == 0)
                                {
                                        TXB = 1;
                                        TING = 0;               //stop send
                                        TEND = 1;               //set send completed flag
                                }
                                else
                                {
                                        TXB = CY;               //write CY to TX port
                                }
                        }
                }
        }
}

//-----------------------------------------
//initial UART module variable
void InitUart()
{
        TING = 0;
        RING = 0;
        TEND = 1;
        REND = 0;
        TCNT = 0;
        RCNT = 0;
}

//-----------------------------------------
//initial UART module variable
void SendData(BYTE dat)
{
        while (!TEND);
        TEND = 0;
        TBUF = dat;
        TING = 1;
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------------*/




;使用了软件模拟串口输出

;define baudrate const
;BAUD = 65536 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
;NOTE: (SYSclk/3/BAUDRATE) must be greater then 75, (RECOMMEND GREATER THEN 100)

;BAUD     EQU     0F400H      ; 1200bps @ 11.0592MHz
;BAUD     EQU     0FA00H      ; 2400bps @ 11.0592MHz
;BAUD     EQU     0FD00H      ; 4800bps @ 11.0592MHz
;BAUD     EQU     0FE80H      ; 9600bps @ 11.0592MHz
;BAUD     EQU     0FF40H      ;19200bps @ 11.0592MHz
;BAUD     EQU     0FFA0H      ;38400bps @ 11.0592MHz
;BAUD     EQU     0FFC0H      ;57600bps @ 11.0592MHz
;BAUD     EQU     0EC00H      ; 1200bps @ 18.432MHz
;BAUD     EQU     0F600H      ; 2400bps @ 18.432MHz
;BAUD     EQU     0FB00H      ; 4800bps @ 18.432MHz
;BAUD     EQU     0FD80H      ; 9600bps @ 18.432MHz
;BAUD     EQU     0FEC0H      ;19200bps @ 18.432MHz
;BAUD     EQU     0FF60H      ;38400bps @ 18.432MHz
BAUD      EQU     0FF95H      ;57600bps @ 18.432MHz
;BAUD     EQU     0E800H      ; 1200bps @ 22.1184MHz
;BAUD     EQU     0F400H      ; 2400bps @ 22.1184MHz
;BAUD     EQU     0FA00H      ; 4800bps @ 22.1184MHz
;BAUD     EQU     0FD00H      ; 9600bps @ 22.1184MHz
;BAUD     EQU     0FE80H      ;19200bps @ 22.1184MHz
;BAUD     EQU     0FF40H      ;38400bps @ 22.1184MHz
;BAUD     EQU     0FF80H      ;57600bps @ 22.1184MHz

;define UART TX/RX port
RXB       BIT     P3.0
TXB       BIT     P3.1

;define SFR
AUXR      DATA    8EH
```

```
;define UART module variable
TBUF     DATA    08H          ;(R0) ready send data buffer   (USER WRITE ONLY)
RBUF     DATA    09H          ;(R1) received data buffer     (UAER READ ONLY)
TDAT     DATA    0AH          ;(R2) sending data buffer      (RESERVED FOR UART MODULE)
RDAT     DATA    0BH          ;(R3) receiving data buffer    (RESERVED FOR UART MODULE)
TCNT     DATA    0CH          ;(R4) send baudrate counter    (RESERVED FOR UART MODULE)
RCNT     DATA    0DH          ;(R5) receive baudrate counter (RESERVED FOR UART MODULE)
TBIT     DATA    0EH          ;(R6) send bit counter         (RESERVED FOR UART MODULE)
RBIT     DATA    0FH          ;(R7) receive bit counter      (RESERVED FOR UART MODULE)

TING     BIT    20H.0      ;sending flag(USER WRITE"1"TO TRIGGER SEND DATA,CLEAR BY MODULE)
RING     BIT    20H.1        ; receiving flag (RESERVED FOR UART MODULE)
TEND     BIT    20H.2        ; sent flag      (SET BY MODULE AND SHOULD USER CLEAR)
REND     BIT    20H.3        ; received flag  (SET BY MODULE AND SHOULD USER CLEAR)

;/*Declare SFR associated with the ADC */
ADC_CONTR       EQU    0BCH         ;ADC control register
ADC_RES         EQU    0BDH         ;ADC high 8-bit result register
ADC_LOW2        EQU    0BEH         ;ADC low 2-bit result register
P1ASF           EQU    09DH         ;P1 secondary function control register

;/*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          ;540 clocks
ADC_SPEEDL      EQU    20H          ;360 clocks
ADC_SPEEDH      EQU    40H          ;180 clocks
ADC_SPEEDHH     EQU    60H          ;90 clocks

ADCCH      DATA    21H                ;ADC channel NO.

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

   ORG     000BH
   LJMP    TM0_ISR

   ORG     002BH
   LJMP    ADC_ISR
;-----------------------------------------
```

```
        ORG     0100H
MAIN:
        MOV     SP,         #7FH
        MOV     ADCCH, #0
        LCALL   INIT_UART                   ;Init UART, use to show ADC result
        LCALL   INIT_ADC                    ;Init ADC sfr
        MOV     TMOD,   #00H                ;timer0 in 16-bit auto reload mode
        MOV     AUXR,   #80H                ;timer0 working at 1T mode
        MOV     TL0,        #LOW BAUD       ;initial timer0 and
        MOV     TH0,        #HIGH BAUD      ;set reload value
        SETB    TR0                         ;tiemr0 start running
        MOV     IE,         #0A0H           ;Enable ADC interrupt and Open master interrupt switch
        SETB    ET0                         ;enable timer0 interrupt
        SETB    PT0                         ;improve timer0 interrupt priority
        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     P1ASF,   #0FFH              ;Set all P1 as analog input port
        MOV     ADC_RES, #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
;/*----------------------------
;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
;/*----------------------------
;Initial UART
;----------------------------*/
INIT_UART:
        CLR     TING
        CLR     RING
        SETB    TEND
        CLR     REND
        CLR     A
        MOV     TCNT,    A
        MOV     RCNT,    A
        RET
;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
        JNB     TEND,    $
        CLR     TEND
        MOV     TBUF,    A
        SETB    TING
        RET
```

```
;-----------------------------------------
;Timer0 interrupt routine for UART
TM0_ISR:
        PUSH    ACC                     ;4 save ACC
        PUSH    PSW                     ;4 save PSW
        MOV     PSW,    #08H            ;3 using register group 1
L_UARTSTART:
;-------------------
        JB      RING,   L_RING          ;4 judge whether receiving
        JB      RXB,    L_REND          ;  check start signal
L_RSTART:
        SETB    RING                    ;  set start receive flag
        MOV     R5,     #4              ;  initial receive baudrate counter
        MOV     R7,     #9              ;  initial receive bit number (8 data bits + 1 stop bit)
        SJMP    L_REND                  ;  end this time slice
L_RING:
        DJNZ    R5,     L_REND            ;4 judge whether sending
        MOV     R5,     #3              ;2 reset send baudrate counter
L_RBIT:
        MOV     C,      RXB             ;3 read RX port data
        MOV     A,      R3              ;1 and shift it to RX buffer
        RRC     A                       ;1
        MOV     R3,     A               ;2
        DJNZ    R7,     L_REND          ;4 judge whether the data have receive completed
L_RSTOP:
        RLC     A                       ;  shift out stop bit
        MOV     R1,     A               ;  save the data to RBUF
        CLR     RING                    ;  stop receive
        SETB    REND                    ;  set receive completed flag
L_REND:
;-----------------
L_TING:
        DJNZ    R4,     L_TEND          ;4 check send baudrate counter
        MOV     R4,     #3              ;2 reset it
        JNB     TING,   L_TEND          ;4 judge whether sending
        MOV     A,      R6              ;1 detect the sent bits
        JNZ     L_TBIT                  ;3 "0" means start bit not sent
L_TSTART:
        CLR     TXB                     ;  send start bit
        MOV     TDAT,   R0              ;  load data from TBUF to TDAT
        MOV     R6,     #9              ;  initial send bit number (8 data bits + 1 stop bit)
        JMP     L_TEND                  ;  end this time slice
L_TBIT:
        MOV     A,      R2              ;1 read data in TDAT
        SETB    C                       ;1 shift in stop bit
        RRC     A                       ;1 shift data to CY
        MOV     R2,     A               ;2 update TDAT
        MOV     TXB,    C               ;4 write CY to TX port
        DJNZ    R6,     L_TEND          ;4 judge whether the data have send completed
```

```
L_TSTOP:
          CLR       TING                        ;  stop send
          SETB      TEND                        ;  set send completed flag
L_TEND:
;-------------------
L_UARTEND:
          POP       PSW                         ;3 restore PSW
          POP       ACC                         ;3 restore ACC
          RETI                                  ;4 (69)

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

          END
```

9.6.2 A/D转换测试程序（ADC查询方式）

1. C程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited -------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*----------------------------------------------------------------------------------*/



/*使用了软件模拟串口输出*/

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

typedef bit BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

//define baudrate const
//BAUD = 256 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
//NOTE: (SYSclk/3/BAUDRATE) must be greater then 98, (RECOMMEND GREATER THEN 110)

//#define BAUD  0xF400          // 1200bps @ 11.0592MHz
//#define BAUD  0xFA00          // 2400bps @ 11.0592MHz
//#define BAUD  0xFD00          // 4800bps @ 11.0592MHz
//#define BAUD  0xFE80          // 9600bps @ 11.0592MHz
//#define BAUD  0xFF40          //19200bps @ 11.0592MHz
//#define BAUD  0xFFA0          //38400bps @ 11.0592MHz

//#define BAUD  0xEC00          // 1200bps @ 18.432MHz
//#define BAUD  0xF600          // 2400bps @ 18.432MHz
//#define BAUD  0xFB00          // 4800bps @ 18.432MHz
//#define BAUD  0xFD80          // 9600bps @ 18.432MHz
//#define BAUD  0xFEC0          //19200bps @ 18.432MHz
#define BAUD    0xFF60          //38400bps @ 18.432MHz

//#define BAUD  0xE800          // 1200bps @ 22.1184MHz
//#define BAUD  0xF400          // 2400bps @ 22.1184MHz
//#define BAUD  0xFA00          // 4800bps @ 22.1184MHz
//#define BAUD  0xFD00          // 9600bps @ 22.1184MHz
//#define BAUD  0xFE80          //19200bps @ 22.1184MHz
//#define BAUD  0xFF40          //38400bps @ 22.1184MHz
//#define BAUD  0xFF80          //57600bps @ 22.1184MHz
```

```
sfr AUXR = 0x8E;
sbit RXB = P3^0;                    //define UART TX/RX port
sbit TXB = P3^1;

/*Declare SFR associated with the ADC */
sfr ADC_CONTR   =   0xBC;                   //ADC control register
sfr ADC_RES     =   0xBD;                   //ADC high 8-bit result register
sfr ADC_LOW2    =   0xBE;                   //ADC low 2-bit result register
sfr P1ASF       =   0x9D;           //P1 secondary function control register

/*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                    //540 clocks
#define ADC_SPEEDL  0x20                    //360 clocks
#define ADC_SPEEDH  0x40                    //180 clocks
#define ADC_SPEEDHH 0x60                    //90 clocks

BYTE TBUF,RBUF;
BYTE TDAT,RDAT;
BYTE TCNT,RCNT;
BYTE TBIT,RBIT;
BOOL TING,RING;
BOOL TEND,REND;

void InitUart();
void InitADC();
void SendData(BYTE dat);
BYTE GetADCResult(BYTE ch);
void Delay(WORD n);
void ShowResult(BYTE ch);

void main()
{
        TMOD = 0x00;                        //timer0 in 16-bit auto reload mode
        AUXR = 0x80;                        //timer0 working at 1T mode
        TL0 = BAUD;
        TH0 = BAUD>>8;                      //initial timer0 and set reload value
        TR0 = 1;                            //tiemr0 start running
        ET0 = 1;                            //enable timer0 interrupt
        PT0 = 1;                            //improve timer0 interrupt priority
        EA = 1;                             //open global interrupt switch

        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_RES;                                 //Return ADC result
}

/*----------------------------
Initial ADC sfr
----------------------------*/
void InitADC()
{
        P1ASF = 0xff;                                   //Open 8 channels ADC function
        ADC_RES = 0;                                    //Clear previous result
        ADC_CONTR = ADC_POWER | ADC_SPEEDLL;
        Delay(2);                                       //ADC power-on and delay
}
```

```
/*----------------------------
Software delay function
----------------------------*/
void Delay(WORD n)
{
        WORD x;

        while (n--)
        {
                x = 5000;
                while (x--);
        }
}

//-----------------------------------------
//Timer interrupt routine for UART

void tm0() interrupt 1 using 1
{
        if (RING)
        {
                if (--RCNT == 0)
                {
                        RCNT = 3;                          //reset send baudrate counter
                        if (--RBIT == 0)
                        {
                                RBUF = RDAT;               //save the data to RBUF
                                RING = 0;                  //stop receive
                                REND = 1;                  //set receive completed flag
                        }
                        else
                        {
                                RDAT >>= 1;
                                if (RXB) RDAT |= 0x80;  //shift RX data to RX buffer
                        }
                }
        }
        else if (!RXB)
        {
                RING = 1;               //set start receive flag
                RCNT = 4;               //initial receive baudrate counter
                 RBIT = 9;              //initial receive bit number (8 data bits + 1 stop bit)
        }
        if (--TCNT == 0)
        {
                TCNT = 3;                       //reset send baudrate counter
                if (TING)                       //judge whether sending
                {
                        if (TBIT == 0)
                        {
                                TXB = 0;                //send start bit
                                TDAT = TBUF;      //load data from TBUF to TDAT
                                TBIT = 9;              //initial send bit number (8 data bits + 1 stop bit)
                        }
                        else
                        {
                                TDAT >>= 1;             //shift data to CY
                                if (--TBIT == 0)
                                {
                                        TXB = 1;
                                        TING = 0;          //stop send
                                        TEND = 1;           //set send completed flag
                                }
                                else
                                {
                                        TXB = CY;           //write CY to TX port
                                }
                        }
                }
        }
}
//-----------------------------------------
//initial UART module variable
void InitUart()
{
        TING = 0;
        RING = 0;
        TEND = 1;
        REND = 0;
        TCNT = 0;
        RCNT = 0;
}

//-----------------------------------------
//initial UART module variable
void SendData(BYTE dat)
{
         while (!TEND);
        TEND = 0;
        TBUF = dat;
        TING = 1;
}
```

2. 汇编程序：

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




;使用了软件模拟串口输出

;define baudrate const
;BAUD = 65536 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
;NOTE: (SYSclk/3/BAUDRATE) must be greater then 75, (RECOMMEND GREATER THEN 100)

;BAUD     EQU     0F400H          ; 1200bps @ 11.0592MHz
;BAUD     EQU     0FA00H          ; 2400bps @ 11.0592MHz
;BAUD     EQU     0FD00H          ; 4800bps @ 11.0592MHz
;BAUD     EQU     0FE80H          ; 9600bps @ 11.0592MHz
;BAUD     EQU     0FF40H          ;19200bps @ 11.0592MHz
;BAUD     EQU     0FFA0H          ;38400bps @ 11.0592MHz
;BAUD     EQU     0FFC0H          ;57600bps @ 11.0592MHz
;BAUD     EQU     0EC00H          ; 1200bps @ 18.432MHz
;BAUD     EQU     0F600H          ; 2400bps @ 18.432MHz
;BAUD     EQU     0FB00H          ; 4800bps @ 18.432MHz
;BAUD     EQU     0FD80H          ; 9600bps @ 18.432MHz
;BAUD     EQU     0FEC0H          ;19200bps @ 18.432MHz
;BAUD     EQU     0FF60H          ;38400bps @ 18.432MHz
BAUD      EQU     0FF95H          ;57600bps @ 18.432MHz
;BAUD     EQU     0E800H          ; 1200bps @ 22.1184MHz
;BAUD     EQU     0F400H          ; 2400bps @ 22.1184MHz
;BAUD     EQU     0FA00H          ; 4800bps @ 22.1184MHz
;BAUD     EQU     0FD00H          ; 9600bps @ 22.1184MHz
;BAUD     EQU     0FE80H          ;19200bps @ 22.1184MHz
;BAUD     EQU     0FF40H          ;38400bps @ 22.1184MHz
;BAUD     EQU     0FF80H          ;57600bps @ 22.1184MHz

;define UART TX/RX port
RXB       BIT     P3.0
TXB       BIT     P3.1

;define SFR
AUXR      DATA    8EH
```

```
;define UART module variable
TBUF     DATA   08H           ;(R0) ready send data buffer   (USER WRITE ONLY)
RBUF     DATA   09H           ;(R1) received data buffer     (UAER READ ONLY)
TDAT     DATA   0AH           ;(R2) sending data buffer      (RESERVED FOR UART MODULE)
RDAT     DATA   0BH           ;(R3) receiving data buffer    (RESERVED FOR UART MODULE)
TCNT     DATA   0CH           ;(R4) send baudrate counter    (RESERVED FOR UART MODULE)
RCNT     DATA   0DH           ;(R5) receive baudrate counter (RESERVED FOR UART MODULE)
TBIT     DATA   0EH           ;(R6) send bit counter         (RESERVED FOR UART MODULE)
RBIT     DATA   0FH           ;(R7) receive bit counter      (RESERVED FOR UART MODULE)

TING     BIT    20H.0     ;sending flag(USER WRITE"1"TO TRIGGER SEND DATA,CLEAR BY MODULE)
RING     BIT    20H.1         ; receiving flag (RESERVED FOR UART MODULE)
TEND     BIT    20H.2         ; sent flag      (SET BY MODULE AND SHOULD USER CLEAR)
REND     BIT    20H.3         ; received flag  (SET BY MODULE AND SHOULD USER CLEAR)

;/*Declare SFR associated with the ADC */
ADC_CONTR       EQU    0BCH               ;ADC control register
ADC_RES         EQU    0BDH               ;ADC high 8-bit result register
ADC_LOW2        EQU    0BEH               ;ADC low 2-bit result register
P1ASF           EQU    09DH               ;P1 secondary function control register

;/*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        ;540 clocks
ADC_SPEEDL      EQU    20H        ;360 clocks
ADC_SPEEDH      EQU    40H        ;180 clocks
ADC_SPEEDHH     EQU    60H        ;90 clocks
;-----------------------------------------
         ORG    0000H
         LJMP   MAIN
         ORG    000BH
         LJMP   TM0_ISR
;-----------------------------------------
MAIN:
         MOV    SP,      #7FH
         MOV    TMOD,    #00H                  ;timer0 in 16-bit auto reload mode
         MOV    AUXR,    #80H                  ;timer0 working at 1T mode
         MOV    TL0,     #LOW BAUD             ;initial timer0 and
         MOV    TH0,     #HIGH BAUD            ;set reload value
         SETB   TR0                            ;tiemr0 start running
         SETB   ET0                            ;enable timer0 interrupt
         SETB   PT0                            ;improve timer0 interrupt priority
         SETB   EA                             ;open global interrupt switch

         LCALL  INIT_UART                      ;Init UART, use to show ADC result
         LCALL  INIT_ADC                       ;Init ADC sfr
```

```
;-------------------------------
            MOV     A,          #55H
            LCALL   SEND_DATA               ;Show result
            MOV     A,          #66H
            LCALL   SEND_DATA               ;Show result
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_RES                         ;Return ADC result
        RET

;/*----------------------------
;Initial ADC sfr
;----------------------------*/
INIT_ADC:
        MOV     P1ASF,      #0FFH               ;Open 8 channels ADC function
        MOV     ADC_RES, #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:
        CLR     TING
        CLR     RING
        SETB    TEND
        CLR     REND
        CLR     A
        MOV     TCNT,   A
        MOV     RCNT,   A
        RET
```

```
;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
            JNB     TEND,   $
            CLR     TEND
            MOV     TBUF,   A
            SETB    TING
            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
;-----------------------------------------
;Timer0 interrupt routine for UART

TM0_ISR:
            PUSH    ACC                     ;4 save ACC
            PUSH    PSW                     ;4 save PSW
            MOV     PSW,    #08H            ;3 using register group 1
L_UARTSTART:
;-------------------
            JB      RING,   L_RING          ;4 judge whether receiving
            JB      RXB,    L_REND          ;check start signal
L_RSTART:
            SETB   RING                     ;  set start receive flag
            MOV    R5,      #4              ;  initial receive baudrate counter
            MOV    R7,      #9              ;  initial receive bit number (8 data bits + 1 stop bit)
            SJMP   L_REND                   ;  end this time slice
L_RING:
            DJNZ   R5,      L_REND          ;4 judge whether sending
            MOV    R5,      #3              ;2 reset send baudrate counter
```

```
L_RBIT:
          MOV     C,        RXB               ;3 read RX port data
          MOV     A,        R3                ;1 and shift it to RX buffer
          RRC     A                           ;1
          MOV     R3,       A                 ;2
          DJNZ    R7,       L_REND            ;4 judge whether the data have receive completed
L_RSTOP:
          RLC     A                           ;  shift out stop bit
          MOV     R1,       A                 ;  save the data to RBUF
          CLR     RING                        ;  stop receive
          SETB    REND                        ;  set receive completed flag
L_REND:
;-----------------
L_TING:
          DJNZ    R4,       L_TEND            ;4 check send baudrate counter
          MOV     R4,       #3                ;2 reset it
          JNB     TING,     L_TEND            ;4 judge whether sending
          MOV     A,        R6                ;1 detect the sent bits
          JNZ     L_TBIT                      ;3 "0" means start bit not sent
L_TSTART:
          CLR     TXB                         ;  send start bit
          MOV     TDAT,     R0                ;  load data from TBUF to TDAT
          MOV     R6,       #9                ;  initial send bit number (8 data bits + 1 stop bit)
          JMP     L_TEND                      ;  end this time slice
L_TBIT:
          MOV     A,        R2                ;1 read data in TDAT
          SETB    C                           ;1 shift in stop bit
          RRC     A                           ;1 shift data to CY
          MOV     R2,       A                 ;2 update TDAT
          MOV     TXB,      C                 ;4 write CY to TX port
          DJNZ    R6,       L_TEND            ;4 judge whether the data have send completed
L_TSTOP:
          CLR     TING                ;  stop send
          SETB    TEND                ;  set send completed flag
L_TEND:
;-------------------
L_UARTEND:
          POP     PSW                         ;3 restore PSW
          POP     ACC                         ;3 restore ACC
          RETI                                ;4 (69)

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

          END
```

第10章 STC15F100系列单片机EEPROM的应用

STC15F100系列单片机内部集成了1KB/2KB的EEPROM，其与程序空间是分开的。地址范围是0000H~03FFH(1KB)/0000H~07FFH(2KB)。1KB的EEPROM分为2个扇区，2KB的EEPROM分为4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的擦写次数在10万次以上，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。

10.1 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								复位值
			MSB							LSB	
IAP_DATA	ISP/IAP Flash Data Register	C2H									1111 1111B
IAP_ADDRH	ISP/IAP Flash Address High	C3H									0000 0000B
IAP_ADDRL	ISP/IAP Flash Address Low	C4H									0000 0000B
IAP_CMD	ISP/IAP Flash Command Register	C5H	-	-	-	-	-	-	MS1	MS0	xxxx x000B
IAP_TRIG	ISP/IAP Flash Command Trigger	C6H									xxxx xxxxB
IAP_CONTR	ISP/IAP Control Register	C7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 x000B
PCON	Power Control	87H	-	-	LVDF	POF	GF1	GF0	PD	IDL	xx11 0000B

1. ISP/IAP数据寄存器IAP_DATA

IAP_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IAP_ADDRH和IAP_ADDRL

IAP_ADDRH ：ISP/IAP 操作时的地址寄存器高八位。
IAP_ADDRL ：ISP/IAP 操作时的地址寄存器低八位。

3. ISP/IAP命令寄存器IAP_CMD

ISP/IAP命令寄存器IAP_CMD格式如下：

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CMD	C5H	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，IAP15F106/IAP15L106除外，IAP15F106/IAP15L106可在用户应用程序区修改用户应用程序区。

4. ISP/IAP命令触发寄存器IAP_TRIG

IAP_TRIG: ISP/IAP操作时的命令触发寄存器。
在IAPEN(IAP_CONTR.7) = 1 时, 对IAP_TRIG先写入5Ah, 再写入A5h, ISP/IAP命令才会生效。

ISP/IAP操作完成后，IAP地址高八位寄存器IAP_ADDRH、IAP地址低八位寄存器IAP_ADDRL和IAP命令寄存器IAP_CMD的内容不变。如果接下来要对下一个地址的数据进行ISP/IAP操作，需手动将该地址的高8位和低8位分别写入IAP_ADDRH和IAP_ADDRL寄存器。

每次IAP操作时，都要对IAP_TRIG先写入5AH，再写入A5H，ISP/IAP命令才会生效。

5. ISP/IAP命令寄存器IAP_CONTR

ISP/IAP控制寄存器IAP_CONTR格式如下：

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IAP_CONTR	C7H	name	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT2	WT0

IAPEN: ISP/IAP功能允许位。0：禁止IAP读/写/擦除Data Flash/EEPROM
1：允许IAP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IAP_TRIG送5Ah/A5h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 IAP_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 IAP_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 IAP_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 IAP_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 (=21us)	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：电源控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCON	87H	name	-	-	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。

5V单片机的低压检测门槛电压：

-40 ^{0}C	25 ^{0}C	85 ^{0}C
4.74	4.64	4.60
4.41	4.32	4.27
4.14	4.05	4.00
3.90	3.82	3.77
3.69	3.61	3.56
3.51	3.43	3.38
3.36	3.28	3.23
3.21	3.14	3.09

3.3V单片机的低压检测门槛电压：

-40 ^{0}C	25 ^{0}C	85 ^{0}C
3.11	3.08	3.09
2.85	2.82	2.83
2.63	2.61	2.61
2.44	2.42	2.43
2.29	2.26	2.26
2.14	2.12	2.12
2.01	2.00	2.00
1.90	1.89	1.89

10.2 STC15F100系列单片机EEPROM空间大小及地址

STC15F100系列单片机内部EEPROM选型一览表 STC15L100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
STC15F101E/ STC15L101E	2K	4	0000h	07FFh
STC15F102E/ STC15L102E	2K	4	0000h	07FFh
STC15F103E/ STC15L103E	2K	4	0000h	07FFh
STC15F104E/ STC15L104E	1K	2	0000h	03FFh
STC15F105E/ STC15L105E	1K	2	0000h	03FFh
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				
IAP15F106/ IAP15L106	–	12	0000h	17FFh

第一扇区		第二扇区		第三扇区		第四扇区		每个扇区512字节，共4个扇区。 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	01FFh	0200h	03FFh	0400h	05FFh	0600h	07FFh	
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	
0800h	09FFh	0A00h	0BFFh	0C00h	0DFFh	0E00h	0FFFh	
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	

10.3 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
        IAP_DATA        DATA    0C2h;   或      IAP_DATA        EQU     0C2h
        IAP_ADDRH       DATA    0C3h;   或      IAP_ADDRH       EQU     0C3h
        IAP_ADDRL       DATA    0C4h;   或      IAP_ADDRL       EQU     0C4h
        IAP_CMD         DATA    0C5h;   或      IAP_CMD         EQU     0C5h
        IAP_TRIG        DATA    0C6h;   或      IAP_TRIG        EQU     0C6h
        IAP_CONTR       DATA    0C7h;   或      IAP_CONTR       EQU     0C7h
;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     IAP_ADDRH,      #BYTE_ADDR_HIGH         ;送地址高字节 ┐ 地址需要改变时
        MOV     IAP_ADDRL,      #BYTE_ADDR_LOW          ;送地址低字节 ┘ 才需重新送地址
        MOV     IAP_CONTR,      #WAIT_TIME      ;设置等待时间   ┐ 此两句可以合成一句,
        ORL     IAP_CONTR,      #10000000B      ;允许ISP/IAP操作 ┘ 并且只送一次就够了
        MOV     IAP_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     IAP_TRIG,       #5Ah  ;先送5Ah,再送A5h到ISP/IAP触发寄存器,每次都需如此
        MOV     IAP_TRIG,       #0A5h       ;送完A5h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IAP_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     IAP_CONTR,      #00000000B      ;禁止ISP/IAP操作
        MOV     IAP_CMD,        #00000000B      ;去除ISP/IAP命令
        ;MOV    IAP_TRIG,       #00000000B      ;防止ISP/IAP命令误触发
        ;MOV    IAP_ADDRH,      #0FFh           ;送地址高字节单元为00,指向非EEPROM区
        ;MOV    IAP_ADDRL,      #0FFh           ;送地址低字节单元为00,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     IAP_DATA,       #ONE_DATA       ;送字节编程数据到IAP_DATA,
                                                ;只有数据改变时才需重新送
        MOV     IAP_ADDRH,      #BYTE_ADDR_HIGH         ;送地址高字节 ┐ 地址需要改变时
        MOV     IAP_ADDRL,      #BYTE_ADDR_LOW          ;送地址低字节 ┘ 才需重新送地址
        MOV     IAP_CONTR,      #WAIT_TIME              ;设置等待时间    ┐ 此两句可合成
        ORL     IAP_CONTR,      #10000000B              ;允许ISP/IAP操作 ┘ 一句,并且只
                                                                           送一次就够了
        MOV     IAP_CMD,        #ISP_IAP_BYTE_PROGRAM       ;送字节编程命令
        MOV     IAP_TRIG,       #5Ah    ;先送5Ah,再送A5h到ISP/IAP触发寄存器,每次都需如此
        MOV     IAP_TRIG,       #0A5h   ;送完A5h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     IAP_CONTR,      #00000000B      ;禁止ISP/IAP操作
        MOV     IAP_CMD,        #00000000B      ;去除ISP/IAP命令
        ;MOV    IAP_TRIG,       #00000000B      ;防止ISP/IAP命令误触发
        ;MOV    IAP_ADDRH,      #0FFh   ;送地址高字节单元为00,;指向非EEPROM区,防止误操作
        ;MOV    IAP_ADDRL,      #0FFh   ;送地址低字节单元为00,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，操作起来越灵活方便.
;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     IAP_ADDRH,      #SECTOR_FIRST_BYTE_ADDR_HIGH ;送扇区起始地址高字节
        MOV     IAP_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     IAP_CONTR,      #WAIT_TIME            ;设置等待时间 ┐ 此两句可以合
        ORL     IAP_CONTR,      #10000000B            ;允许ISP/IAP  ┘ 成一句，并且只
                                                                      送一次就够了
        MOV     IAP_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     IAP_TRIG,       #5Ah
                                ;先送5Ah，再送A5h到ISP/IAP触发寄存器，每次都需如此
        MOV     IAP_TRIG,       #0A5h         ;送完A5h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     IAP_CONTR,      #00000000B          ;禁止ISP/IAP操作
        MOV     IAP_CMD,        #00000000B          ;去除ISP/IAP命令
        ;MOV    IAP_TRIG,       #00000000B          ;防止ISP/IAP命令误触发
        ;MOV    IAP_ADDRH,      #0FFh           ;送地址高字节单元为00，指向非EEPROM区
        ;MOV    IAP_ADDRL,      #0FFh           ;送地址低字节单元为00，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5A和A5触发后，下一次IAP命令是否还需要送5A和A5触发？
答：是，一定要。

10.4 EEPROM测试程序(不使用模拟串口)

1. C程序:

```
;STC15F100系列单片机EEPROM/IAP 功能测试程序演示
/*------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*------------------------------------------------------------------------------------*/




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

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the IAP */
sfr  IAP_DATA      =   0xC2;           //Flash data register
sfr  IAP_ADDRH     =   0xC3;           //Flash address HIGH
sfr  IAP_ADDRL     =   0xC4;           //Flash address LOW
sfr  IAP_CMD       =   0xC5;           //Flash command register
sfr  IAP_TRIG      =   0xC6;           //Flash command trigger
sfr  IAP_CONTR     =   0xC7;           //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 STC15F100 series EEPROM
#define     IAP_ADDRESS     0x0000

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 = 0x5a;                //Send trigger command1 (0x5a)
        IAP_TRIG = 0xa5;                //Send trigger command2 (0xa5)
        _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 = 0x5a;                 //Send trigger command1 (0x5a)
        IAP_TRIG = 0xa5;                 //Send trigger command2 (0xa5)
        _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 = 0x5a;                 //Send trigger command1 (0x5a)
        IAP_TRIG = 0xa5;                 //Send trigger command2 (0xa5)
        _nop_();                         //MCU will hold here until ISP/IAP/EEPROM
                                         //operation complete
        IapIdle();
}
```

2. 汇编程序：

```
;STC15F100系列单片机EEPROM/IAP 功能测试程序演示
/*----------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*----------------------------------------------------------------------------------*/



;/*Declare SFRs associated with the IAP */
IAP_DATA        EQU     0C2H            ;Flash data register
IAP_ADDRH       EQU     0C3H            ;Flash address HIGH
IAP_ADDRL       EQU     0C4H            ;Flash address LOW
IAP_CMD         EQU     0C5H            ;Flash command register
IAP_TRIG        EQU     0C6H            ;Flash command trigger
IAP_CONTR       EQU     0C7H            ;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 STC15F100 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,       #5AH                ;Send trigger command1 (0x5a)
            MOV     IAP_TRIG,       #0A5H               ;Send trigger command2 (0xa5)
            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,       #5AH            ;Send trigger command1 (0x5a)
        MOV     IAP_TRIG,       #0A5H           ;Send trigger command2 (0xa5)
        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,       #5AH          ;Send trigger command1 (0x5a)
        MOV     IAP_TRIG,       #0A5H         ;Send trigger command2 (0xa5)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                      ;Close ISP/IAP/EEPROM function
        RET

        END
```

10.5 EEPROM测试程序(使用模拟串口送出显示)

1. C程序:

```
;STC15F100系列单片机EEPROM/IAP 功能测试程序演示
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




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

//define baudrate const
//BAUD = 256 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
//NOTE: (SYSclk/3/BAUDRATE) must be greater then 98, (RECOMMEND GREATER THEN 110)

//#define BAUD  0xF400          // 1200bps @ 11.0592MHz
//#define BAUD  0xFA00          // 2400bps @ 11.0592MHz
//#define BAUD  0xFD00          // 4800bps @ 11.0592MHz
//#define BAUD  0xFE80          // 9600bps @ 11.0592MHz
//#define BAUD  0xFF40          //19200bps @ 11.0592MHz
//#define BAUD  0xFFA0          //38400bps @ 11.0592MHz

//#define BAUD  0xEC00          // 1200bps @ 18.432MHz
//#define BAUD  0xF600          // 2400bps @ 18.432MHz
//#define BAUD  0xFB00          // 4800bps @ 18.432MHz
//#define BAUD  0xFD80          // 9600bps @ 18.432MHz
//#define BAUD  0xFEC0          //19200bps @ 18.432MHz
#define BAUD    0xFF60          //38400bps @ 18.432MHz

//#define BAUD  0xE800          // 1200bps @ 22.1184MHz
//#define BAUD  0xF400          // 2400bps @ 22.1184MHz
//#define BAUD  0xFA00          // 4800bps @ 22.1184MHz
//#define BAUD  0xFD00          // 9600bps @ 22.1184MHz
//#define BAUD  0xFE80          //19200bps @ 22.1184MHz
//#define BAUD  0xFF40          //38400bps @ 22.1184MHz
//#define BAUD  0xFF80          //57600bps @ 22.1184MHz
```

```
sfr AUXR = 0x8E;
sbit RXB = P3^0;                          //define UART TX/RX port
sbit TXB = P3^1;

typedef bit BOOL;
typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the IAP */
sfr IAP_DATA      =   0xC2;        //Flash data register
sfr IAP_ADDRH     =   0xC3;        //Flash address HIGH
sfr IAP_ADDRL     =   0xC4;        //Flash address LOW
sfr IAP_CMD       =   0xC5;        //Flash command register
sfr IAP_TRIG      =   0xC6;        //Flash command trigger
sfr IAP_CONTR     =   0xC7;        //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

//EEPROM Start address
#define IAP_ADDRESS 0x800

BYTE TBUF,RBUF;
BYTE TDAT,RDAT;
BYTE TCNT,RCNT;
BYTE TBIT,RBIT;
BOOL TING,RING;
BOOL TEND,REND;

void UART_INIT();
void UART_SEND(BYTE dat);
void Delay(BYTE n);
void IapIdle();
BYTE IapReadByte(WORD addr);
void IapProgramByte(WORD addr, BYTE dat);
void IapEraseSector(WORD addr);

void main()
{
        WORD i;
        BYTE j;

        TMOD = 0x00;                //timer0 in 16-bit auto reload mode
        AUXR = 0x80;                //timer0 working at 1T mode
        TL0 = BAUD;
        TH0 = BAUD>>8;              //initial timer0 and set reload value
        TR0 = 1;                    //tiemr0 start running
        ET0 = 1;                    //enable timer0 interrupt
        PT0 = 1;                    //improve timer0 interrupt priority
        EA = 1;                     //open global interrupt switch
        UART_INIT();

        P1 = 0xfe;                  //1111,1110 System Reset OK
        Delay(10);                  //Delay
        UART_SEND(0x5a);
        UART_SEND(0xa5);
        IapEraseSector(IAP_ADDRESS);        //Erase current sector
        for (i=0; i<512; i++)               //Check whether all sector data is FF
        {
                j = IapReadByte(IAP_ADDRESS+i);
                UART_SEND(j);
//              if (j != 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
        {
                j = IapReadByte(IAP_ADDRESS+i);
                UART_SEND(j);
                if (j != (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 = 0x5a;                   //Send trigger command1 (0x5a)
            IAP_TRIG = 0xa5;                   //Send trigger command2 (0xa5)
            _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 = 0x5a;                   //Send trigger command1 (0x5a)
            IAP_TRIG = 0xa5;                   //Send trigger command2 (0xa5)
            _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 = 0x5a;                   //Send trigger command1 (0x5a)
            IAP_TRIG = 0xa5;                   //Send trigger command2 (0xa5)
```

```
        _nop_();                //MCU will hold here until ISP/IAP/EEPROM operation complete
        IapIdle();
}
//-----------------------------------------
//Timer interrupt routine for UART

void tm0() interrupt 1 using 1
{
        if (RING)
        {
                if (--RCNT == 0)
                {
                        RCNT = 3;               //reset send baudrate counter
                        if (--RBIT == 0)
                        {
                                RBUF = RDAT;            //save the data to RBUF
                                RING = 0;               //stop receive
                                REND = 1;               //set receive completed flag
                        }
                        else
                        {
                                RDAT >>= 1;
                                if (RXB) RDAT |= 0x80;  //shift RX data to RX buffer
                        }
                }
        }
        else if (!RXB)
        {
                RING = 1;                       //set start receive flag
                RCNT = 4;                       //initial receive baudrate counter
                RBIT = 9;                       //initial receive bit number (8 data bits + 1 stop bit)
        }

        if (--TCNT == 0)
        {
                TCNT = 3;                       //reset send baudrate counter
                if (TING)                       //judge whether sending
                {
                        if (TBIT == 0)
                        {
                                TXB = 0;                //send start bit
                                TDAT = TBUF;            //load data from TBUF to TDAT
                                TBIT = 9;       //initial send bit number (8 data bits + 1 stop bit)
                        }
                        else
                        {
                                TDAT >>= 1;                     //shift data to CY
                                if (--TBIT == 0)
                                {
                                        TXB = 1;
                                        TING = 0;               //stop send
                                        TEND = 1;               //set send completed flag
                                }
                                else
                                {
                                        TXB = CY;           //write CY to TX port
                                }
                        }
                }
        }
}

//-----------------------------------------
//initial UART module variable
void UART_INIT()
{
        TING = 0;
        RING = 0;
        TEND = 1;
        REND = 0;
        TCNT = 0;
        RCNT = 0;
}

//-----------------------------------------
//initial UART module variable
void UART_SEND(BYTE dat)
{
        while (!TEND);
        TEND = 0;
        TBUF = dat;
        TING = 1;
}
```

2. 汇编程序：

```
;STC15F100系列单片机EEPROM/IAP 功能测试程序演示
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- 演示STC 15 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




;------------------------------------------
;define baudrate const
;BAUD = 65536 - SYSclk/3/BAUDRATE/M (1T:M=1; 12T:M=12)
;NOTE: (SYSclk/3/BAUDRATE) must be greater then 75, (RECOMMEND GREATER THAN 100)

;BAUD     EQU     0F400H          ; 1200bps @ 11.0592MHz
;BAUD     EQU     0FA00H          ; 2400bps @ 11.0592MHz
;BAUD     EQU     0FD00H          ; 4800bps @ 11.0592MHz
;BAUD     EQU     0FE80H          ; 9600bps @ 11.0592MHz
;BAUD     EQU     0FF40H          ;19200bps @ 11.0592MHz
;BAUD     EQU     0FFA0H          ;38400bps @ 11.0592MHz
;BAUD     EQU     0FFC0H          ;57600bps @ 11.0592MHz

;BAUD     EQU     0EC00H          ; 1200bps @ 18.432MHz
;BAUD     EQU     0F600H          ; 2400bps @ 18.432MHz
;BAUD     EQU     0FB00H          ; 4800bps @ 18.432MHz
;BAUD     EQU     0FD80H          ; 9600bps @ 18.432MHz
;BAUD     EQU     0FEC0H          ;19200bps @ 18.432MHz
;BAUD     EQU     0FF60H          ;38400bps @ 18.432MHz
BAUD      EQU     0FF95H          ;57600bps @ 18.432MHz

;BAUD     EQU     0E800H          ; 1200bps @ 22.1184MHz
;BAUD     EQU     0F400H          ; 2400bps @ 22.1184MHz
;BAUD     EQU     0FA00H          ; 4800bps @ 22.1184MHz
;BAUD     EQU     0FD00H          ; 9600bps @ 22.1184MHz
;BAUD     EQU     0FE80H          ;19200bps @ 22.1184MHz
;BAUD     EQU     0FF40H          ;38400bps @ 22.1184MHz
;BAUD     EQU     0FF80H          ;57600bps @ 22.1184MHz
```

```
;-----------------------------------------
;define UART TX/RX port

RXB        BIT     P3.0
TXB        BIT     P3.1

;-----------------------------------------
;define SFR

AUXR       DATA    8EH
;-----------------------------------------
;define UART module variable

TBUF       DATA    08H               ;(R0) ready send data buffer   (USER WRITE ONLY)
RBUF       DATA    09H               ;(R1) received data buffer     (UAER READ ONLY)
TDAT       DATA    0AH               ;(R2) sending data buffer      (RESERVED FOR UART MODULE)
RDAT       DATA    0BH               ;(R3) receiving data buffer    (RESERVED FOR UART MODULE)
TCNT       DATA    0CH               ;(R4) send baudrate counter    (RESERVED FOR UART MODULE)
RCNT       DATA    0DH               ;(R5) receive baudrate counter (RESERVED FOR UART MODULE)
TBIT       DATA    0EH               ;(R6) send bit counter         (RESERVED FOR UART MODULE)
RBIT       DATA    0FH               ;(R7) receive bit counter      (RESERVED FOR UART MODULE)

TING       BIT     20H.0             ;sending flag
                                     ;(USER WRITE"1"TO TRIGGER SEND DATA, CLEAR BY MODULE)
RING       BIT     20H.1             ; receiving flag (RESERVED FOR UART MODULE)
TEND       BIT     20H.2             ; sent flag      (SET BY MODULE AND SHOULD USER CLEAR)
REND       BIT     20H.3             ; received flag  (SET BY MODULE AND SHOULD USER CLEAR)

;/*Declare SFR associated with the IAP */
IAP_DATA        EQU    0C2H         ;Flash data register
IAP_ADDRH       EQU    0C3H         ;Flash address HIGH
IAP_ADDRL       EQU    0C4H         ;Flash address LOW
IAP_CMD         EQU    0C5H         ;Flash command register
IAP_TRIG        EQU    0C6H         ;Flash command trigger
IAP_CONTR       EQU    0C7H         ;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
```

```
;//EEPROM Start address
IAP_ADDRESS     EQU     0800H
;-----------------------------------------
            ORG     0000H
            LJMP    MAIN

;-----------------------------------------
;Timer0 interrupt routine for UART

            ORG     000BH

            PUSH    ACC                         ;4 save ACC
            PUSH    PSW                         ;4 save PSW
            MOV     PSW,    #08H                ;3 using register group 1
L_UARTSTART:
;-------------------
            JB      RING,   L_RING              ;4 judge whether receiving
            JB      RXB,    L_REND              ;  check start signal
L_RSTART:
            SETB    RING                        ;  set start receive flag
            MOV     R5,     #4                  ;  initial receive baudrate counter
            MOV     R7,     #9                  ;  initial receive bit number (8 data bits + 1 stop bit)
            SJMP    L_REND                      ;  end this time slice
L_RING:
            DJNZ    R5,     L_REND              ;4 judge whether sending
            MOV     R5,     #3                  ;2 reset send baudrate counter
L_RBIT:
            MOV     C,      RXB                 ;3 read RX port data
            MOV     A,      R3                  ;1 and shift it to RX buffer
            RRC     A                           ;1
            MOV     R3,     A                   ;2
            DJNZ    R7,     L_REND              ;4 judge whether the data have receive completed
L_RSTOP:
            RLC     A                           ;  shift out stop bit
            MOV     R1,     A                   ;  save the data to RBUF
            CLR     RING                        ;  stop receive
            SETB    REND                        ;  set receive completed flag
L_REND:
;-----------------
L_TING:
            DJNZ    R4,     L_TEND              ;4 check send baudrate counter
            MOV     R4,     #3                  ;2 reset it
            JNB     TING,   L_TEND              ;4 judge whether sending
            MOV     A,      R6                  ;1 detect the sent bits
            JNZ     L_TBIT                      ;3 "0" means start bit not sent
```

```
L_TSTART:
          CLR     TXB                          ;  send start bit
          MOV     TDAT,    R0                  ;  load data from TBUF to TDAT
          MOV     R6,      #9                  ;  initial send bit number (8 data bits + 1 stop bit)
          JMP     L_TEND                       ;  end this time slice
L_TBIT:
          MOV     A,       R2                  ;1 read data in TDAT
          SETB    C                            ;1 shift in stop bit
          RRC     A                            ;1 shift data to CY
          MOV     R2,      A                   ;2 update TDAT
          MOV     TXB,     C                   ;4 write CY to TX port
          DJNZ    R6,      L_TEND              ;4 judge whether the data have send completed
L_TSTOP:
          CLR     TING                         ;  stop send
          SETB    TEND                         ;  set send completed flag
L_TEND:
;-------------------
L_UARTEND:
          POP     PSW                          ;3 restore PSW
          POP     ACC                          ;3 restore ACC
          RETI                                 ;4 (69)

;-----------------------------------------
;initial UART module variable
UART_INIT:
          CLR     TING
          CLR     RING
          SETB    TEND
          CLR     REND
          CLR     A
          MOV     TCNT,    A
          MOV     RCNT,    A
          RET
;-----------------------------------------
;send UART data
UART_SEND:
          JNB     TEND,    $
          CLR     TEND
          MOV     TBUF,    A
          SETB    TING
          RET
```

```
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,     #7FH
        MOV     TMOD,   #00H                        ;timer0 in 16-bit auto reload mode
        MOV     AUXR,   #80H                        ;timer0 working at 1T mode
        MOV     TL0,    #LOW BAUD                   ;initial timer0 and
        MOV     TH0,    #HIGH BAUD                  ;set reload value
        SETB    TR0                                 ;tiemr0 start running
        SETB    ET0                                 ;enable timer0 interrupt
        SETB    PT0                                 ;improve timer0 interrupt priority
        SETB    EA                                  ;open global interrupt switch
        LCALL   UART_INIT

        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
        LCALL   UART_SEND
//      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
```

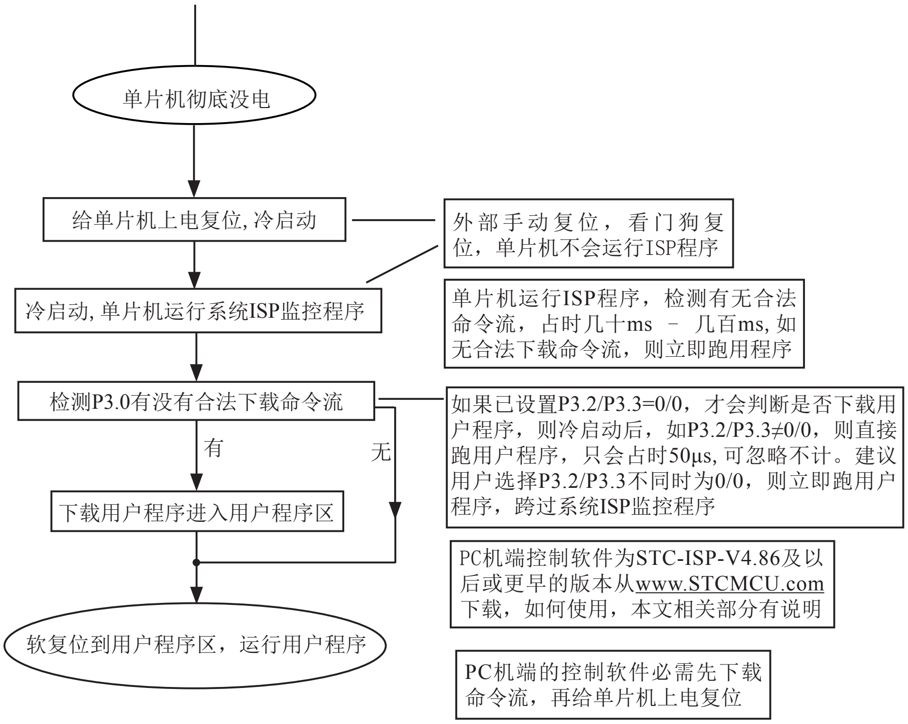
```
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS                    ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                              ;Set counter (512)
        MOV     R1,     #2
        MOV     R2,     #0
CHECK2:                                                 ;Verify 512 bytes data
        LCALL   IAP_READ                                ;Read Flash
        LCALL   UART_SEND
        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,       #5AH                    ;Send trigger command1 (0x5a)
        MOV     IAP_TRIG,       #0A5H                   ;Send trigger command2 (0xa5)
        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,       #5AH                    ;Send trigger command1 (0x5a)
        MOV     IAP_TRIG,       #0A5H                   ;Send trigger command2 (0xa5)
        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,       #5AH                    ;Send trigger command1 (0x5a)
        MOV     IAP_TRIG,       #0A5H                   ;Send trigger command2 (0xa5)
        NOP                             ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                                ;Close ISP/IAP/EEPROM function
        RET

        END
```

第11章 STC15系列单片机开发/编程工具说明

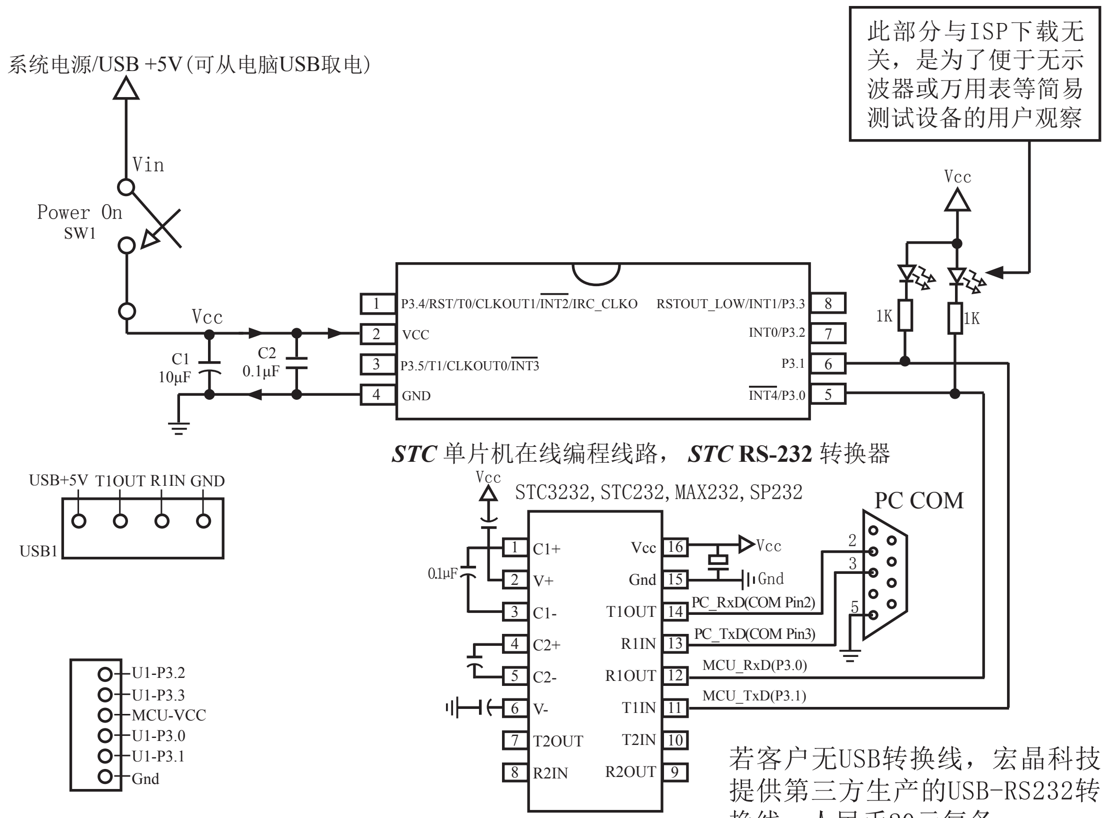
11.1 在系统可编程(ISP)原理，官方演示工具使用说明

11.1.1 在系统可编程(ISP)原理使用说明



单片机彻底没电
给单片机上电复位, 冷启动
外部手动复位，看门狗复位，单片机不会运行ISP程序
冷启动, 单片机运行系统ISP监控程序
单片机运行ISP程序，检测有无合法命令流，占时几十ms – 几百ms, 如无合法下载命令流，则立即跑用程序
检测P3.0有没有合法下载命令流
有
无
如果已设置P3.2/P3.3=0/0，才会判断是否下载用户程序，则冷启动后，如P3.2/P3.3≠0/0，则直接跑用户程序，只会占时50μs, 可忽略不计。建议用户选择P3.2/P3.3不同时为0/0，则立即跑用户程序，跨过系统ISP监控程序
下载用户程序进入用户程序区
PC机端控制软件为STC-ISP-V4.86及以后或更早的版本从www.STCMCU.com下载，如何使用，本文相关部分有说明
软复位到用户程序区，运行用户程序
PC机端的控制软件必需先下载命令流，再给单片机上电复位

11.1.2 STC15F100系列在系统可编程(ISP)典型应用线路图



若客户无USB转换线，宏晶科技提供第三方生产的USB-RS232转换线，人民币20元每条。

内部高可靠复位，不需要外部复位电路

P3.4/RST/T0/CLKOUT1/$\overline{\text{INT2}}$/IRC_CLKO脚出厂时默认为I/O口，可以通过 STC-ISP 编程器将其设置为RST复位脚.

内部高精度R/C振荡器，温飘±1%(-40^{0}C~+85^{0}C), 常温下温飘5‰, 不需要昂贵的外部晶振

建议加上电容C1(10μF), C2(0.1μF), 可去除电源噪声，提高抗干扰能力

如何产生虚拟串口:①安装Windows驱动程序；②插上USB-RS232转换线(若客户无USB转换线，宏晶科技提供第三方生产的USB-RS232转换线，人民币20元每条.)；③确定PC端口COM：右击我的电脑—>属性—>硬件—>设备管理器—>确定所扩展的串口是PC电脑虚拟的第几个COM.

STC15F100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。

STC15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）

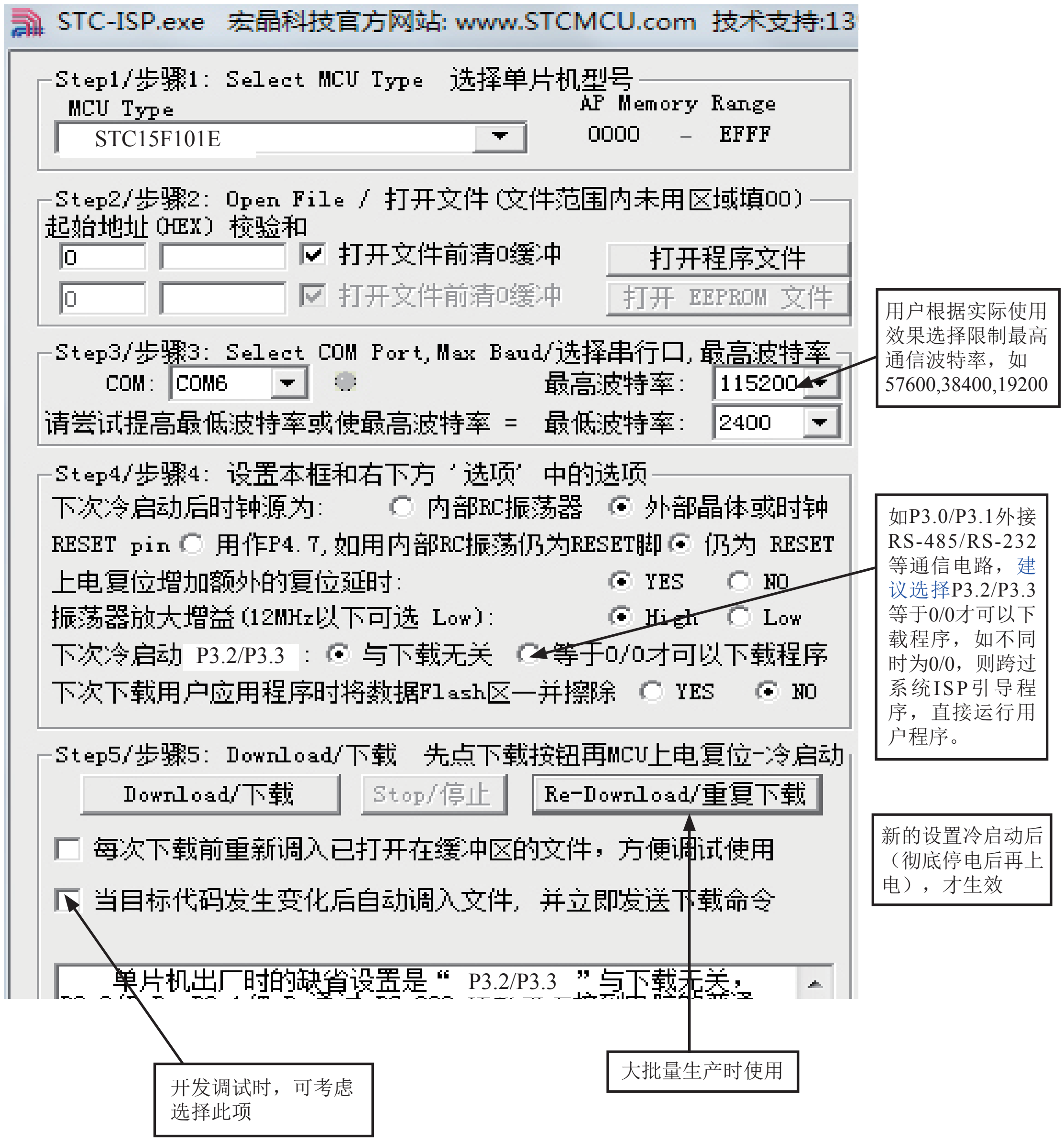
登陆 www.STCMCU.com 网站，从STC 半导体专栏下载PC（电脑）端的ISP 程序，然后 将其自解压，再安装即可（执行setup.exe），注意随时更新软件。

(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).STC15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P3.2/P3.3=0/0才可以下载程序”

11.1.3 电脑端的STC-ISP控制软件界面使用说明


STC-ISP.exe 宏晶科技官方网站: www.STCMCU.com 技术支持:13

Step1/步骤1: Select MCU Type 选择单片机型号
MCU Type: STC15F101E
AP Memory Range: 0000 - EFFF

Step2/步骤2: Open File / 打开文件 (文件范围内未用区域填00)
起始地址 (HEX) 0 校验和
[✓] 打开文件前清0缓冲 | 打开程序文件
0
[✓] 打开文件前清0缓冲 | 打开 EEPROM 文件

Step3/步骤3: Select COM Port, Max Baud/选择串行口, 最高波特率
COM: COM6 最高波特率: 115200
请尝试提高最低波特率或使最高波特率 = 最低波特率: 2400

Step4/步骤4: 设置本框和右下方 '选项' 中的选项
下次冷启动后时钟源为: 内部RC振荡器 [✓ 外部晶体或时钟]
RESET pin 用作P4.7,如用内部RC振荡仍为RESET脚 [✓ 仍为 RESET]
上电复位增加额外的复位延时: [✓ YES] NO
振荡器放大增益(12MHz以下可选 Low): [✓ High] Low
下次冷启动 P3.2/P3.3: [✓ 与下载无关] 等于0/0才可以下载程序
下次下载用户应用程序时将数据Flash区一并擦除 YES [✓ NO]

Step5/步骤5: Download/下载 先点下载按钮再MCU上电复位-冷启动
Download/下载 | Stop/停止 | Re-Download/重复下载
[ ] 每次下载前重新调入已打开在缓冲区的文件，方便调试使用
[ ] 当目标代码发生变化后自动调入文件，并立即发送下载命令

单片机出厂时的缺省设置是" P3.2/P3.3 "与下载无关，

用户根据实际使用效果选择限制最高通信波特率，如57600,38400,19200

如P3.0/P3.1外接RS-485/RS-232等通信电路，建议选择P3.2/P3.3等于0/0才可以下载程序，如不同时为0/0，则跨过系统ISP引导程序，直接运行用户程序。

新的设置冷启动后（彻底停电后再上电），才生效

开发调试时，可考虑选择此项

大批量生产时使用

上图并非STC15系列的实际界面，将会更新，在此画出，只是为了便于读者对STC-ISP编程工具的理解和认识。

Step1/步骤1：选择你所使用的单片机型号，如STC15F101E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，可让宏晶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”
(STC15F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P3.2/P3.3 = 0,0才可以下载用户程序”
这样冷启动后如 P3.2, P3.3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P3.2/P3.3 = 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).

11.1.4 宏晶科技的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STC15系列8Pin / 20Pin / 28Pin。我们在下载板的反面贴了一张标签纸，说明它是支持8Pin /20Pin / 28Pin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(宏晶提供)

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插头的一端插入宏晶的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，P3.2, P3.3 = 1, 1,不短接到地。

处于按下状态， P3.2, P3.3 = 0, 0, 短接到地。

如果单片机已被设成“下次冷启动P3.2/P3.3 = 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P3.2/P3.3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。

11.1.6 若无RS-232转换器，如何用宏晶的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P3.2,P3.3接入转换板CN2插座(如果需要的话)
4. 如须P3.2, P3.3 = 0, 0, 短接到地，可在用户系统上将其短接到地，或将P3.2/P3.3也从用户系统上引到STC-ISP Ver3.0A PCB 板上，将SW3开关按下，则P3.2/P3.3=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。

11.2 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。

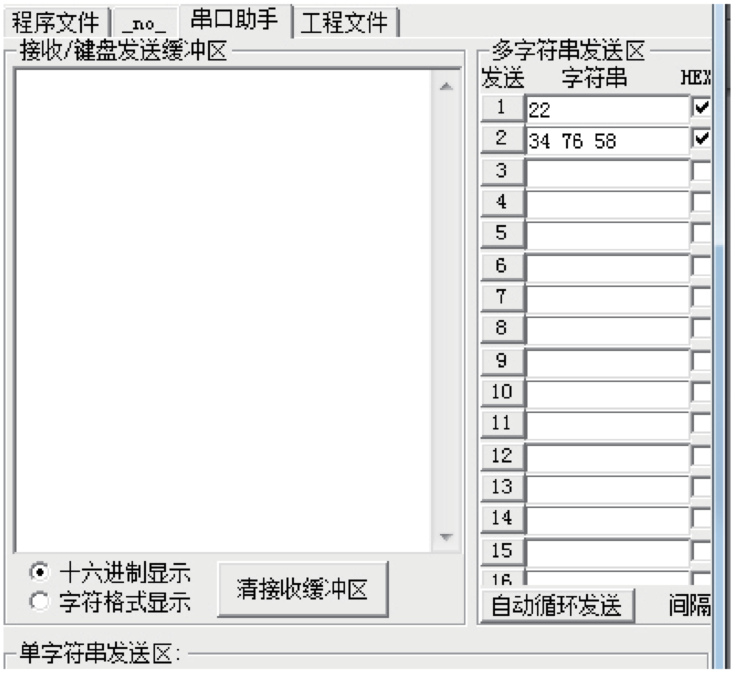
编程烧录器:

我们有: STC15F100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:有专门的STC15xx系列的下载板

仿真器:如您已有老的仿真器，可仿真普通8052的基本功能

STC15F100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.



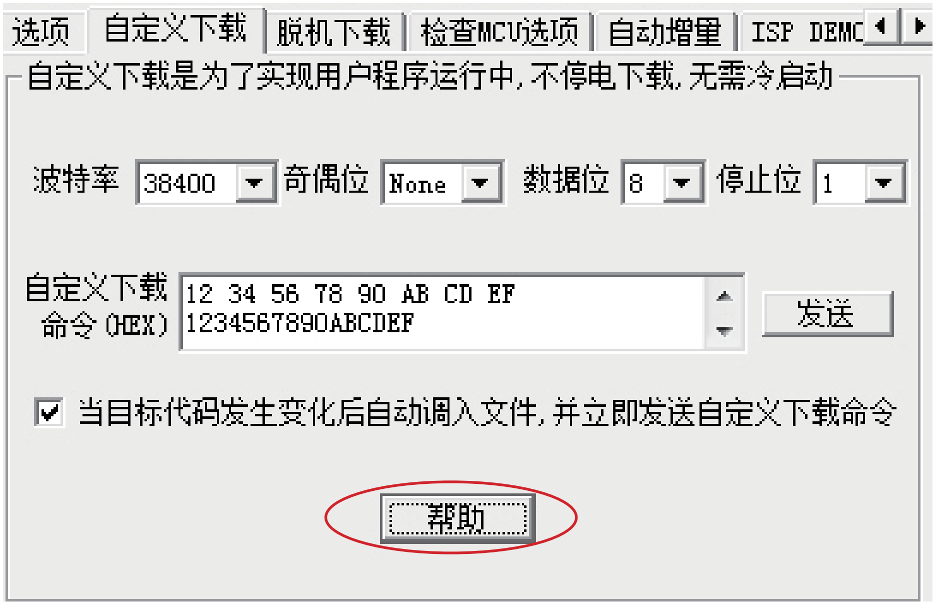
发送	字符串	HEX
1	22	✓
2	34 76 58	✓
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		

11.3 自定义下载演示程序(实现不停电下载)

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International 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宏晶科技的资料及程序 -----------*/
/*---------------------------------------------------------------------------------*/




#include <reg51.h>
#include <instrins.h>

sfr IAP_CONTR = 0xc7;
sbit MCU_Start_Led = P3^5;

#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：



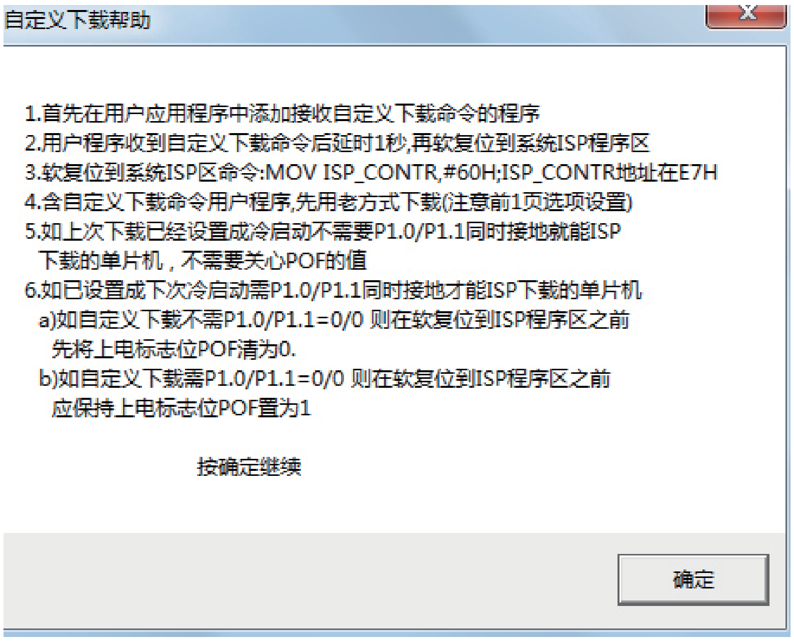
自定义下载是为了实现用户程序运行中，不停电下载，无需冷启动

波特率 38400　奇偶位 None　数据位 8　停止位 1

自定义下载命令（HEX）
12 34 56 78 90 AB CD EF
1234567890ABCDEF

[✓] 当目标代码发生变化后自动调入文件，并立即发送自定义下载命令

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



1.首先在用户应用程序中添加接收自定义下载命令的程序
2.用户程序收到自定义下载命令后延时1秒,再软复位到系统ISP程序区
3.软复位到系统ISP区命令:MOV ISP_CONTR,#60H;ISP_CONTR地址在E7H
4.含自定义下载命令用户程序,先用老方式下载(注意前1页选项设置)
5.如上次下载已经设置成冷启动不需要P1.0/P1.1同时接地就能ISP
  下载的单片机，不需要关心POF的值
6.如已设置成下次冷启动需P1.0/P1.1同时接地才能ISP下载的单片机
  a)如自定义下载不需P1.0/P1.1=0/0 则在软复位到ISP程序区之前
    先将上电标志位POF清为0.
  b)如自定义下载需P1.0/P1.1=0/0 则在软复位到ISP程序区之前
    应保持上电标志位POF置为1

按确定继续

注意左图中第3条目对STC15系列单片机而言是有误的。针对STC15系列应更正为:
3.软复位到系统ISP监控程序区命令:
MOV IAP_CONTR, #60H
;IAP_CONTR地址在C7H

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.

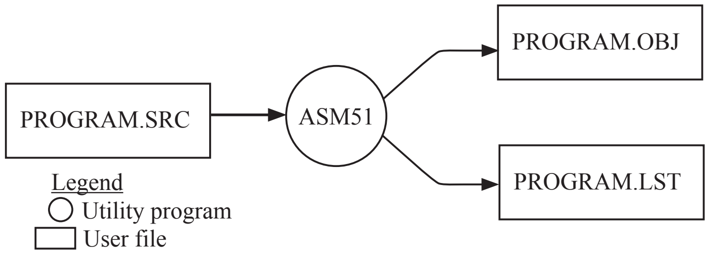


Figure 1 Assembling a source program

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 semi-colon (;). 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 contintued. 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

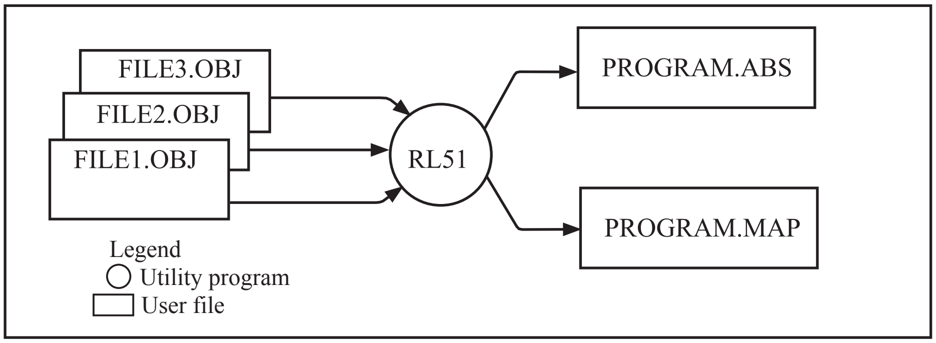
Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any affect on the program itself. They can be entered on the invocation line when a program is assembled, 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:

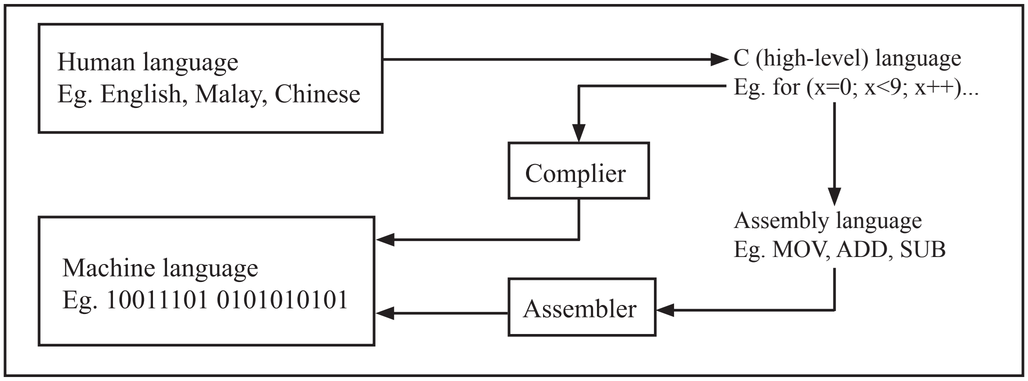
```
sbit portbit = P1^0;            /*Use variable portbit to refer to P1.0*/
main ( )
{
TMOD = 1;
while (1)
      {
         TH0 = 0xFE;
         TL0 = 0xC;
         TR0 = 1;
         while (TF0 !=1);
         TR0 = 0;
         TF0 = 0;
         portbit = !(P1.^0);
      }
}
```

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 COMPLIERS

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 = &num;
```

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 = &num;
```

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 STC15F100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	+5.5	V
DC power supply (3V)	VDD - VSS	-0.3	+3.6	V
Voltage on any pin	-	-0.3	VCC + 0.3	V

DC Specification (5V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
VDD	Operating Voltage	3.3	5.0	5.5	V	
IPD	Power Down Current	-	< 0.1	-	uA	5V
IIDL	Idle Current	-	3.0	-	mA	5V
ICC	Operating Current	-	4	20	mA	5V
VIL1	Input Low (P0,P1,P2,P3)	-	-	0.8	V	5V
VIH1	Input High (P0,P1,P2,P3)	2.0	-	-	V	5V
VIH2	Input High (RESET)	2.2	-	-	V	5V
IOL1	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	5V@Vpin=0.45V
IOH1	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	200	270	-	uA	5V
IOH2	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull, Strong-output)	-	20	-	mA	5V@Vpin=2.4V
IIL	Logic 0 input current (P0,P1,P2,P3)	-	-	50	uA	Vpin=0V
ITL	Logic 1 to 0 transition current (P0,P1,P2,P3)	100	270	600	uA	Vpin=2.0V

DC Specification (3V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
VDD	Operating Voltage	2.4	3.3	3.6	V	
IPD	Power Down Current	-	<0.1	-	uA	3.3V
IIDL	Idle Current	-	2.0	-	mA	3.3V
ICC	Operating Current	-	4	10	mA	3.3V
VIL1	Input Low (P0,P1,P2,P3)	-	-	0.8	V	3.3V
VIH1	Input High (P0,P1,P2,P3)	2.0	-	-	V	3.3V
VIH2	Input High (RESET)	2.2	-	-	V	3.3V
IOL1	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	3.3V@Vpin=0.45V
IOH1	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	140	170	-	uA	3.3V
IOH2	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull)	-	20	-	mA	3.3V
IIL	Logic 0 input current (P0,P1,P2,P3)	-	8	50	uA	Vpin=0V
ITL	Logic 1 to 0 transition current (P0,P1,P2,P3)	-	110	600	uA	Vpin=2.0V

附录D STC15xx系列单片机取代传统8051注意事项

STC15F100系列单片机的定时器0/定时器1与传统8051完全兼容，上电复位后，定时器部分缺省还是除12再计数的,所以定时器完全兼容。

STC15F100系列单片机对传统8051的111条指令执行速度全面提速,最快的指令快24倍,最慢的指令快3倍.靠软件延时实现精确延时的程序需要调整。

其它需注意的细节：

普通I/O口既作为输入又作为输出：

传统8051单片机执行I/O口操作,由高变低或由低变高,以及读外部状态都是12个时钟,而现在STC15F100系列单片机执行相应的操作是4个时钟.传统8051单片机如果对外输出为低,直接读外部状态是读不对的.必须先将I/O口置高才能够读对,而传统8051单片机由低变高的指令是12个时钟,该指令执行完成后,该I/O口也确实已变高.故可以紧跟着由低变高的指令后面,直接执行读该I/O口状态指令.而STC15F100系列单片机由于执行由低变高的指令是4个时钟,太快了,相应的指令执行完以后,I/O口还没有变高,要再过一个时钟之后,该I/O口才可以变高.故建议此状况下增加2个空操作延时指令再读外部口的状态。

I/O口驱动能力：

最新STC15F100系列单片机I/O口的灌电流是20mA,驱动能力超强,驱动大电流时,不容易烧坏.
传统STC89Cxx系列单片机I/O口的灌电流是6mA,驱动能力不够强,不能驱动大电流,建议使用STC15F100系列

看门狗：

最新STC15F100系列单片机的看门狗寄存器WDT_CONTR的地址在C1H,增加了看门狗复位标志位

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset value
WDT_CONTR	C1h	Watch-Dog-Timer Control register	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	xx00,0000

传统STC89系列增强型单片机看门狗寄存器WDT_CONTR的地址在E1H,没有看门狗复位标志位

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset value
WDT_CONTR	E1h	Watch-Dog-Timer Control register	-	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	xx00,0000

最新STC15F100系列单片机的看门狗在ISP烧录程序可设置上电复位后直接启动看门狗,而传统STC89系列单片机无此功能.故最新STC15F100系列单片机看门狗更可靠.

与EEPROM操作相关的寄存器

STC15Fxx单片机ISP/IAP控制寄存器地址和STC89xx系列单片机ISP/IAP控制寄存器地址不同如下：

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
STC15Fxx 系列 IAP_DATA STC89xx 系列 ISP_DATA	C2h E2h	ISP/IAP Flash Data Register									1111,1111
STC15Fxx 系列 IAP_ADDRH STC89xx 系列 ISP_ADDRH	C3h E3h	ISP/IAP Flash Address High									0000,0000
STC15Fxx 系列 IAP_ADDRL STC89xx 系列 ISP_ADDRL	C4h E4h	ISP/IAP Flash Address Low									0000,0000
STC15Fxx 系列 IAP_CMD STC89xx 系列 ISP_CMD	C5h E5h	ISP/IAP Flash Command Register	-	-	-	-	-	-	MS1	MS0	xxxx,xx00
STC15Fxx 系列 IAP_TRIG STC89xx 系列 ISP_TRIG	C6h E6h	ISP/IAP Flash Command Trigger									xxxx,xxxx
STC15Fxx系列 IAP_CONTR STC89xx 系列 ISP_CONTR	C7h E7h	ISP/IAP Control Register	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000,x000

ISP/IAP_TRIG寄存器有效启动IAP操作,需顺序送入的数据不一样:
STC15Fxx系列单片机的ISP/IAP命令要生效,要对IAP_TRIG寄存器按顺序先送5Ah,再送A5h方可
STC89xx 系列单片机的ISP/IAP命令要生效,要对IAP_TRIG寄存器按顺序先送46h,再送B9h方可

EEPROM起始地址不一样:
STC15Fxx系列单片机的EEPROM起始地址全部从0000h开始,每个扇区512字节
STC89xx系列单片机的EEPROM起始地址分别有从1000h/2000h/4000h/8000h开始的,程序兼容性不够好.

外部中断:

最新STC15Fxx系列单片机有5个外部中断。其中外部中断0(INT0)和外部中断1(INT1)可配置为2种中断触发方式:
第一种方式，仅下降沿触发中断，与传统8051的外部中断0和1的下降沿中断兼容。
第二种方式，上升沿中断和下降沿中断同时支持。
另外相对传统STC89系列单片机,最新的STC15Fxx系列单片机还增加了外部中断2、外部中断3和外部中断4，这三个新增的外部中断都只能下降沿触发中断。
而传统STC89系列单片机的外部中断0和外部中断1只可以配置为下降沿中断或低电平中断。

定时器:

最新STC15Fxx系列单片机的定时器/计数器0和定时器/计数器1与传统STC89系列单片机的定时器/计数器0和定时器/计数器1的最大不同在于定时器的工作模式0。最新STC15Fxx系列单片机的定时器/计数器0和定时器/计数器1的工作模式0是16位自动重装模式，而传统STC89系列单片机的定时器/计数器0和定时器/计数器1的模式0是13位定时/计数器模式。最新STC15Fxx系列单片机的定时器/计数器0和定时器/计数器1仍保留着其他3种工作模式，这3中工作模式与传统的STC89系列单片机的定时器/计数器0和定时器/计数器1的工作模式兼容。另外传统的STC89系列单片机还设有定时器2，而最新STC15Fxx系列单片机只有定时器0和1。

外部时钟和内部时钟:

最新STC15Fxx系列单片机内部集成了高精度R/C振荡器作为系统时钟，省掉了昂贵的外部晶体振荡时钟。而传统STC89系列单片机只能使用外部晶体或时钟作为系统时钟.

功耗:

功耗由2部分组成,晶体振荡器放大电路的功耗和单片机的数字电路功耗组成,
晶体振荡器放大电路的功耗: 最新STC15F100系列单片机比STC89xx系列低.
单片机的数字电路功耗:时钟频率越高,功耗越大,最新STC15F100系列单片机在相同工作频率下,指令执行速度比传统STC89系列单片机快3-24倍,故可用较低的时钟频率工作,这样功耗更低.而且STC15F100系列单片机可以利用内部的时钟分频器对时钟进行分频,以较低的频率工作,使得单片机的功耗更低。

掉电唤醒:

最新STC15Fxx系列单片机支持外部中断上升沿或下降沿均可唤醒，也可仅下降沿唤醒。传统STC89系列单片机是只支持外部中断低电平唤醒。

附录E STC15F100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byte)	SRAM字节	定时器	A/D 8路	看门狗(WDT)	内置复位	EEPROM	内部低压检测中断	内部可选复位门槛电压	支持掉电唤醒外部中断	掉电唤醒专用定时器	封装8-Pin（6个I/0口）价格(RMB ¥)	
													SOP-8	DIP-8
STC15F100系列单片机选型一览														
STC15F100	5.5-3.8	512	128	2	–	有	有	-	有	8级	5	–		¥1.19
STC15F101	5.5-3.8	1K	128	2	–	有	有	-	有	8级	5	–		¥1.40
STC15F101E	5.5-3.8	1K	128	2	–	有	有	2K	有	8级	5	–		¥1.45
STC15F102	5.5-3.8	2K	128	2	–	有	有	-	有	8级	5	–	¥1.30	¥1.50
STC15F102E	5.5-3.8	2K	128	2	–	有	有	2K	有	8级	5	–	¥1.35	¥1.55
STC15F103	5.5-3.8	3K	128	2	–	有	有	-	有	8级	5	–	¥1.40	¥1.60
STC15F103E	5.5-3.8	3K	128	2	–	有	有	2K	有	8级	5	–	¥1.45	¥1.65
STC15F104	5.5-3.8	4K	128	2	–	有	有	-	有	8级	5	–	¥1.50	¥1.70
STC15F104E	5.5-3.8	4K	128	2	–	有	有	1K	有	8级	5	–	¥1.55	¥1.75
STC15F105	5.5-3.8	5K	128	2	–	有	有	-	有	8级	5	–		
STC15F105E	5.5-3.8	5K	128	2	–	有	有	1K	有	8级	5	–		
IAP15F106	5.5-3.8	6K	128	2	–	有	有	IAP	有	8级	5	–		
STC15L100系列单片机选型一览表														
STC15L100	3.6-2.4	512	128	2	–	有	有	-	有	8级	5	–	¥0.99	¥1.19
STC15L101	3.6-2.4	1K	128	2	–	有	有	-	有	8级	5	–	¥1.20	¥1.40
STC15L101E	3.6-2.4	1K	128	2	–	有	有	2K	有	8级	5	–	¥1.25	¥1.45
STC15L102	3.6-2.4	2K	128	2	–	有	有	-	有	8级	5	–	¥1.30	¥1.50
STC15L102E	3.6-2.4	2K	128	2	–	有	有	2K	有	8级	5	–	¥1.35	¥1.55
STC15L103	3.6-2.4	3K	128	2	–	有	有	-	有	8级	5	–	¥1.40	¥1.60
STC15L103E	3.6-2.4	3K	128	2	–	有	有	2K	有	8级	5	–	¥1.45	¥1.65
STC15L104	3.6-2.4	4K	128	2	–	有	有	-	有	8级	5	–	¥1.50	¥1.70
STC15L104E	3.6-2.4	4K	128	2	–	有	有	1K	有	8级	5	–	¥1.55	¥1.75
STC15L105	3.6-2.4	5K	128	2	–	有	有	-	有	8级	5	–		
STC15L105E	3.6-2.4	5K	128	2	–	有	有	1K	有	8级	5	–		
IAP15L106	3.6-2.4	6K	128	2	–	有	有	IAP	有	8级	5	–		