

STC15F104ESW系列单片机器件手册

- 超强抗干扰，超级加密
- 采用STC第八代加密技术
- 彻底省掉外部昂贵的晶振和复位电路
- 可省掉外部EEPROM，利用IAP技术
- ISP/IAP，在线编程，无需编程器/仿真器
- 128字节SRAM
- 1个时钟/机器周期8051
- 高速，高可靠
- 超低功耗，超低价
- 超强抗静电，超强抗干扰

2012年5月开始供货

STC15F101ESW
STC15F102ESW
STC15F103ESW
STC15F104ESW
STC15F105ESW
IAP15F106SW

STC15L101ESW
STC15L102ESW
STC15L103ESW
STC15L104ESW
STC15L105ESW
IAP15L106SW

采用**STC第八代加密技术**，现悬赏**10万元人民币**请专家帮忙查找加密有无漏洞

STC-ISP：最方便的在线升级软件，无需编程器，无需仿真器

STC——8051单片机全球第一品牌，全球最大的8051单片机设计公司

请同行不要再抄袭我们的设计、规格和管脚排列，再抄袭就很无耻了

全部中国本土独立自主知识产权，请全体中国人民支持，您的支持是中国本土力量前进的有力保证。

封装后，全部**175°C八小时高温烘烤**，高品质制造保证

技术支持网站：**www.STCMCU.com**

Update date: 2011/12/7

目录

第1章 STC15F104ESW系列单片机总体介绍	8
1.1 STC15F104ESW系列单片机简介(2012年5月开始供货)	8
1.2 STC15F104ESW系列单片机的内部结构	10
1.3 STC15系列单片机管脚图及选型一览表	11
1.3.1 STC15F104ESW系列单片机管脚图(2012年5月开始供货)	11
1.3.2 STC15F104ESW系列单片机选型一览表	12
1.3.3 STC15F2K60S2系列单片机管脚图(2012年3月开始供货)	13
1.3.4 STC15F2K60S2系列单片机选型一览表	16
1.3.3 STC15F4K60S4系列单片机管脚图	17
1.3.4 STC15F4K60S4系列单片机选型一览表	21
1.3.5 STC15F1K20AD系列单片机管脚图(2012年5月开始供货)	22
1.3.6 STC15F1K20AD系列单片机选型一览表	23
1.3.7 STC15F412EACS系列单片机管脚图.....	24
1.3.8 STC15F412EACS系列单片机选型一览表.....	25
1.3.9 STC15F204EA系列单片机管脚图	26
——A版本现已供货, B版本2012年4月~6月开始供货.....	26
1.3.10 STC15F204EA系列单片机选型一览表	27
1.3.11 STC15F104E系列单片机管脚图.....	28
——A版本现已供货, D版本2012年3月开始供货.....	28
1.3.12 STC15F104E系列单片机选型一览表	29
1.4 STC15F104ESW系列单片机最小应用系统.....	30
1.5 STC15F104ESW系列在系统可编程(ISP)典型应用线路图.....	31
1.5.1 利用RS-232转换器的典型应用线路图	31
1.5.2 利用USB转串口的典型应用线路图.....	32
1.6 STC15F104ESW系列管脚说明.....	33
1.7 STC15系列单片机封装尺寸图.....	34
1.8 STC15系列单片机命名规则	48
1.8.1 STC15F104ESW系列单片机命名规则	48
1.8.2 STC15F2K60S2系列单片机命名规则	49
1.8.3 STC15F4K60S4系列单片机命名规则	50
1.8.4 STC15F1K20AD系列单片机命名规则	51
1.8.5 STC15F412EACS系列单片机命名规则.....	52
1.8.6 STC15F204EA系列单片机命名规则	53

1.8.7 STC15F104E系列单片机命名规则	54
1.9 串行口在多个管脚之间切换及其测试程序(C和汇编)	55
1.10 每个单片机具有全球唯一身份证号码(ID号)及其测试程序	58
第2章 STC15系列的时钟, 省电模式及复位	64
2.1 STC15F104ESW系列单片机的内部可配置时钟.....	64
2.1.1 在STC-ISP下载/编程工具中设置内部高精度R/C时钟的振荡频率	64
2.1.2 内部时钟分频和分频寄存器	65
2.1.3 可编程时钟输出(也可作分频器使用)	66
2.1.3.1 与可编程时钟输出有关的特殊功能寄存器	66
2.1.3.2 内部R/C时钟输出及测试程序(C和汇编).....	68
2.1.3.2 定时器2对系统时钟或外部引脚T2的时钟输入进行可编程分频输出及测试程序...	70
2.2 STC15F104ESW系列单片机的省电模式.....	74
2.2.1 低速模式及其测试程序(C和汇编)	76
2.2.2 空闲模式(功耗<1mA)及其测试程序(C和汇编)	79
2.2.3 掉电模式/停机模式及其测试程序(C和汇编)	81
2.2.3.1 掉电模式/停机模式被唤醒后程序执行流程说明及测试程序(C和汇编).....	82
2.2.3.2 用掉电唤醒专用定时器唤醒掉电模式/停机模式的测试程序(C和汇编).....	85
2.2.3.3 用外部中断INT0(上升沿+下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编).....	87
2.2.3.4 用外部中断INT1(上升沿+下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编).....	89
2.2.3.5 用外部中断 $\overline{\text{INT2}}$ (下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编).....	91
2.2.3.6 用外部中断 $\overline{\text{INT3}}$ (下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编).....	93
2.2.3.7 用外部中断 $\overline{\text{INT4}}$ (下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编).....	95
2.2.3.8 用RxD管脚由高到低的变化唤醒掉电模式/停机模式的测试程序(C和汇编).....	97
2.3 复位.....	101
2.3.1 外部RST引脚复位.....	101
2.3.2 软件复位及其测试程序(C和汇编)	101
2.3.3 掉电复位/上电复位	104
2.3.4 MAX810专用复位电路复位.....	104
2.3.5 内部低压检测复位	104
2.3.6 看门狗(WDT)复位.....	108
2.3.7 冷启动复位和热启动复位	113
第3章 片内存储器和特殊功能寄存器(SFRs)	114
3.1 程序存储器.....	114
3.2 数据存储器(SRAM)	115
3.3 特殊功能寄存器(SFRs)	117
第4章 STC15F104ESW系列单片机的I/O口结构	123
4.1 I/O口各种不同的工作模式及配置介绍.....	123

4.2 管脚P5.4/RST的特别说明.....	124
4.3 与I/O口有关的特殊功能寄存器及其在程序中的地址声明	125
4.4 STC15F104ESW系列单片机P1/P3/P5口的测试程序.....	127
4.5 I/O口各种不同的工作模式结构框图.....	130
4.5.1 准双向口输出配置	130
4.5.2 强推挽输出配置	131
4.5.3 仅为输入（高阻）配置	131
4.5.4 开漏输出配置(若外加上拉电阻，也可读)	131
4.6 一种典型三极管控制电路	133
4.7 典型发光二极管控制电路	133
4.8 混合电压供电系统3V/5V器件I/O口互连.....	133
4.9 如何让I/O口上电复位时为低电平	134
4.10 I/O口直接驱动LED数码管应用线路图.....	135
4.11 I/O口直接驱动LCD应用线路图.....	136
4.12 STC15系列单片机I/O口软件模拟I ² C接口的测试程序.....	137
4.12.1 STC15系列单片机I/O口软件模式I2C接口的主机模式	137
4.12.2 STC15系列单片机I/O口软件模式I2C接口的从机模式	141
第5章 指令系统	144
5.1 寻址方式	144
5.1.1 立即寻址	144
5.1.2 直接寻址	144
5.1.3 间接寻址	144
5.1.4 寄存器寻址	145
5.1.5 相对寻址	145
5.1.6 变址寻址	145
5.1.7 位寻址	145
5.2 完整指令集对照表(与传统8051对照)	146
——共111条指令，每条指令的详细执行时间	146
5.3 传统8051单片机指令定义详解(中文&English)	152
5.3.1 传统8051单片机指令定义详解	152
5.3.2 Instruction Definitions of Traditional 8051 MCU	192
第6章 中断系统	229
6.1 中断结构图.....	230
6.2 中断向量入口地址/查询次序/优先级/请求标志/允许位表.....	232
6.3 在Keil C中如何声明中断函数	233

6.4 中断寄存器.....	234
6.5 中断优先级.....	238
6.6 中断处理	240
6.7 外部中断	241
6.8 中断的测试程序(C和汇编)	242
6.8.1 外部中断0(INT0)的测试程序	242
6.8.1.1 外部中断INT0(上升沿+下降沿)的测试程序(C和汇编).....	242
6.8.1.2 外部中断INT0(下降沿)的测试程序(C和汇编).....	244
6.8.2 外部中断1(INT1)的测试程序	246
6.8.2.1 外部中断INT1(上升沿+下降沿)的测试程序(C和汇编).....	246
6.8.2.2 外部中断INT1(下降沿)的测试程序(C和汇编).....	248
6.8.3 外部中断2($\overline{\text{INT2}}$)(下降沿中断)的测试程序(C和汇编)	250
6.8.4 外部中断3($\overline{\text{INT3}}$)(下降沿中断)的测试程序(C和汇编)	252
6.8.5 外部中断4($\overline{\text{INT4}}$)(下降沿中断)的测试程序(C和汇编)	254
6.8.6 T2扩展为外部下降沿中断的测试程序(C和汇编).....	255
——利用T2的外部计数方式	255
第7章 定时器/计数器T2及其应用	258
7.1 定时器/计数器T2的相关寄存器.....	258
7.2 定时器/计数器2作定时器及其测试程序(C和汇编)	261
7.2.1 定时器2的16位自动重装载模式的测试程序(C和汇编).....	262
7.2.2 定时器2扩展为外部下降沿中断的测试程序(C和汇编).....	265
7.3 定时器/计数器2对内部系统时钟或外部引脚T2的时钟输入进行可编程 时钟分频输出及其测试程序(C和汇编)	268
7.4 定时器/计数器2作串行口波特率发生器及其测试程序	272
7.5 如何将定时器T2的速度提高12倍	278
7.6 可编程时钟输出(也可作分频器使用)	279
7.6.1 与可编程时钟输出有关的特殊功能寄存器	279
7.6.2 内部R/C时钟输出及其测试程序(C和汇编).....	281
7.6.3 定时器2对系统时钟或外部引脚T2的时钟输入进行可编程分频输出	283
——及测试程序(C和汇编).....	283
7.7 掉电唤醒专用定时器, 进入掉电模式后可将单片机唤醒	287
——及测试程序(C和汇编).....	287
第8章 串行口通信	291
8.1 串行口的相关寄存器	291
8.2 串行口工作模式	296
8.2.1 串行口工作模式0: 同步移位寄存器(建议初学者不学)	296

8.2.2 串行口工作模式1: 8位UART, 波特率可变	298
8.2.3 串行口工作模式2: 9位UART, 波特率固定(建议不学习)	301
8.2.4 串行口工作模式3: 9位UART, 波特率可变	303
8.3 串行口的波特率的设置	305
8.4 串行口的测试程序(C和汇编)	308
——定时器2作串口波特率发生器	308
8.5 双机通信	314
8.6 多机通信	325
第9章 STC15F104ESW系列单片机EEPROM的应用	331
9.1 IAP及EEPROM新增特殊功能寄存器介绍	331
9.2 STC15F104ESW系列单片机EEPROM空间大小及地址	335
9.3 IAP及EEPROM汇编简介	336
9.4 EEPROM测试程序(C和汇编)	340
9.4.1 EEPROM测试程序(不用串口送出数据)(C和汇编)	340
9.4.2 EEPROM测试程序(使用串口送出数据)(C和汇编)	348
第10章 STC15系列单片机开发/编程工具说明	357
10.1 在系统可编程(ISP)原理, 官方演示工具使用说明	357
10.1.1 在系统可编程(ISP)原理使用说明	357
10.1.2 STC15F104ESW系列在系统可编程(ISP)典型应用线路图	358
10.1.2.1 利用RS-232转换器的典型应用线路图	358
10.1.2.2 利用USB转串口的典型应用线路图	360
10.1.3 电脑端的STC-ISP下载控制软件界面使用说明	361
10.1.3.1 STC-ISP下载控制软件Ver4.88的界面使用说明	361
10.1.3.2 最新STC15系列单片机的ISP下载控制软件Ver6.01的界面使用说明	363
10.1.4 STC-ISP(最方便的在线升级软件)下载编程工具硬件使用说明	368
10.1.5 若无RS-232转换器, 如何用STC的ISP下载板做RS-232通信转换	369
10.1.6 如何解决VB版ISP工具在XP或WIN7下控件过期或不能注册的问题	370
10.2 编译器/汇编器, 编程器, 仿真器	373
10.3 自定义下载演示程序(实现不停电下载)	374
附录A 汇编语言编程	377
附录B C语言编程	399
附录C STC15F104ESW系列单片机电气特性	409
附录D: 用串口扩展I/O接口	410
附录E: 一个I/O口驱动发光二极管并扫描按键	413

附录F: STC15系列单片机取代传统8051注意事项	414
附录G: STC15F104ESW系列对指令系统的提升.....	417
附录H: 如何利用Keil C软件减少代码长度.....	423
附录I: 逻辑代数的基础	424
——无微机原理基础的用户请从本章开始学习	424
I.1 数制与编码	424
I.1.1 数制转换.....	424
I.1.2 原码、反码及补码.....	427
I.1.3 常用编码.....	427
I.2 几种常用的逻辑运算及其图形符号	429

第1章 STC15F104ESW系列单片机总体介绍

1.1 STC15F104ESW系列单片机简介(2012年5月开始供货)

STC15F104ESW系列单片机是STC生产的单时钟/机器周期(1T)的单片机，是高速/高可靠/低功耗/超强抗干扰的新一代8051单片机，采用STC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8-12倍。内部集成高精度R/C时钟， $\pm 1\%$ 温飘，常温下温飘5%，5MHz~35MHz宽范围可设置，彻底省掉外部昂贵的晶振和复位电路(内部已集成复位电路)。内置128字节大容量SRAM，1个独立的高速异步串行通信口(UART)，可在多个管脚之间进行切换，分时复用可作两个串口使用，针对多串行口通信/电机控制/强干扰场合。

在 Keil C 开发环境中，选择 Intel 8052 编译，头文件包含<reg51.h>即可

1. 增强型 8051 CPU，1T，单时钟/机器周期，速度比普通8051快8-12倍
2. 工作电压：

STC15F104ESW 系列工作电压：5.5V - 3.8V (5V 单片机)

STC15L104ESW 系列工作电压：3.6V - 2.4V (3V 单片机)

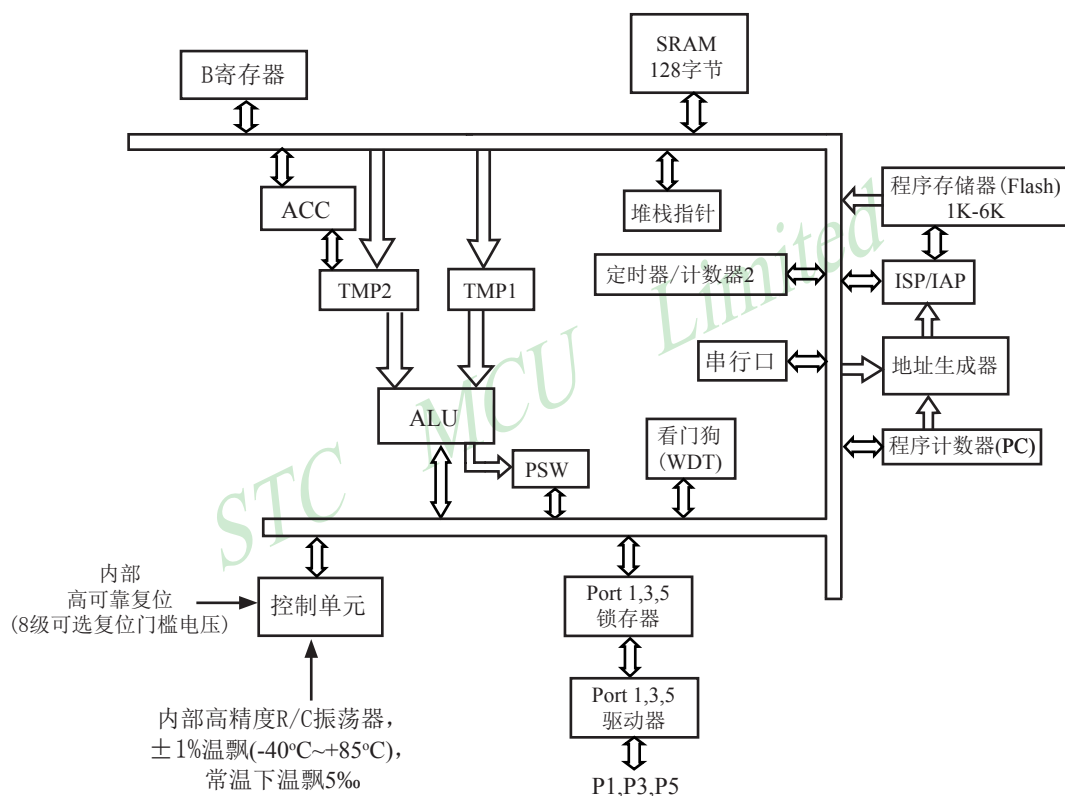
3. 1K/2K/3K/4K/5K/6K字节片内Flash程序存储器，擦写次数10万次以上
4. 片内128字节的SRAM
5. 大容量片内EEPROM，擦写次数10万次以上
6. ISP/IAP，在系统可编程/在应用可编程，无需编程器，无需仿真器
7. 内部高可靠复位，8级可选复位门槛电压，彻底省掉外部复位电路
8. 内部高精度R/C时钟， $\pm 1\%$ 温飘($-40^{\circ}\text{C}\sim+85^{\circ}\text{C}$)，常温下温飘5%，内部时钟从5MHz~35MHz可选(5.5296MHz / 11.0592MHz / 22.1184MHz / 33.1776MHz)
9. 工作频率范围：5MHz~35MHz，相当于普通8051的60MHz~420MHz
10. 一个完全独立的串口，可在多个管脚之间进行切换，分时复用可当2个串口使用：
串口(RxD/P3.0, TxD/P3.1)可以切换到(RxD_3/P3.6, TxD_3/P3.7)。
11. 低功耗设计：低速模式，空闲模式，掉电模式/停机模式。
12. 可将掉电模式/停机模式唤醒的资源有：INT0/P3.2, INT1/P3.3 (INT0/INT1上升沿下降沿中断均可), $\overline{\text{INT2}}$ /P3.6, $\overline{\text{INT3}}$ /P3.7, $\overline{\text{INT4}}$ /P3.0 ($\overline{\text{INT2}}$ / $\overline{\text{INT3}}$ / $\overline{\text{INT4}}$ 仅可下降沿中断)；RxD(可在RxD/P3.0和RxD_3/P3.6之间切换)；内部低功耗掉电唤醒专用定时器。
13. 增加了内部低功耗掉电唤醒专用定时器，也可将MCU从掉电模式/停机模式唤醒。
14. 共1个定时器，1个16位可重载的定时器T2，并可实现可编程时钟输出T2CLK0

15. 可编程时钟输出功能:T2在P3. 0输出时钟, 在P5. 4口输出内部高精度R/C时钟IRC_CLK0(可分频IRC_CLK/1, IRC_CLK/2, IRC_CLK/4), 。
16. 硬件看门狗(WDT)
17. 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
18. 通用I/O口(14个), 复位后为: 准双向口/弱上拉(普通8051传统I/O口) 可设置成四种模式: 准双向口/弱上拉, 强推挽/强上拉, 仅为输入/高阻, 开漏
每个I/O口驱动能力均可达到20mA, 但整个芯片最大不要超过90mA.
如果I/O口不够用, 可外接74HC595(参考价0.21元)来扩展I/O口.
19. 封装: SOP-16, DIP-16.
20. 全部175°C 八小时高温烘烤, 高品质制造保证
21. 开发环境: 在 Keil C 开发环境中, 选择 Intel 8052 编译即可

STC MCU Limited

1.2 STC15F104ESW系列单片机的内部结构

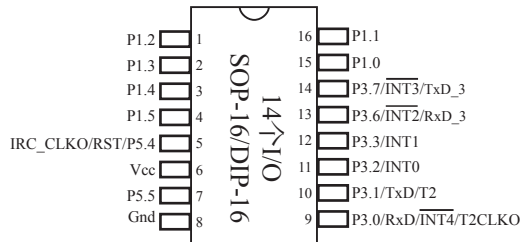
STC15F104ESW系列单片机的内部结构框图如下图所示。STC15F104ESW系列单片机中包含中央处理器(CPU)、程序存储器(Flash)、数据存储器(SRAM)、定时器、I/O口、看门狗、UART串口,片内高精度R/C振荡时钟及高可靠复位等模块。STC15F104ESW系列单片机几乎包含了数据采集和控制中所需的所有单元模块,可称得上是一个片上系统。



STC15F104ESW系列内部结构框图

1.3 STC15系列单片机管脚图及选型一览表

1.3.1 STC15F104ESW系列单片机管脚图 (2012年5月开始供货)



T2CLKO是指定时器T2的时钟输出
(与CLKOUT2同，有时也写作CLKOUT2)。

T2CLKO除可以做可编程时钟输出外，还可以作分频器使用。

中国大陆本土STC姚永平独立创新设计：
请不要再抄袭我们的设计、规格和管脚排列，
再抄袭就很无耻了。

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
P_SW2	BAH	Peripheral function switch	S1_S1	-	-						0xxx,xxxx

串口/S1可在2个地方切换，由 S1_S1 控制位来选择	
S1_S1	串口/S1可在P1/P3之间来回切换
0	串口/S1在 [P3. 0/RxD, P3. 1/TxD]
1	串口/S1在 [P3. 6/RxD_3, P3. 7/TxD_3]

1.3.2 STC15F104ESW系列单片机选型一览表

型号	工作电压(V)	Flash程序存储器(byte)	大容量SRAM字节	串行口	SPI	普通定时器	CCP PCA PWM 定时器	掉电唤醒专用定时器	A/D 8路	看门狗	内置复位	EEPROM	内部低压检测中断	内部可选复位门电压	支持掉电唤醒外部中断	所有封装	
																SOP-16 / DIP-16	
																部分封装	
																价格(RMB ¥)	
																SOP-16	DIP-16
STC15F104ESW系列单片机选型一览																	
STC15F101ESW	5.5-3.8	1K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15F102ESW	5.5-3.8	2K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15F103ESW	5.5-3.8	3K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15F104ESW	5.5-3.8	4K	128	1	-	1	-	有	-	有	有	1K	有	8级	5		
STC15F105ESW	5.5-3.8	5K	128	1	-	1	-	有	-	有	有	1K	有	8级	5		
IAP15F106SW	5.5-3.8	6K	128	1	-	1	-	有	-	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	
STC15L104ESW系列单片机选型一览表																	
STC15L101ESW	2.4-3.6	1K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15L102ESW	2.4-3.6	2K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15L103ESW	2.4-3.6	3K	128	1	-	1	-	有	-	有	有	2K	有	8级	5		
STC15L104ESW	2.4-3.6	4K	128	1	-	1	-	有	-	有	有	1K	有	8级	5		
STC15L105ESW	2.4-3.6	5K	128	1	-	1	-	有	-	有	有	1K	有	8级	5		
IAP15L106SW	2.4-3.6	6K	128	1	-	1	-	有	-	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	

提供客制化IC服务

1.3.3 STC15F2K60S2系列单片机管脚图 (2012年3月开始供货)

所有封装形式均满足欧盟RoHS要求,

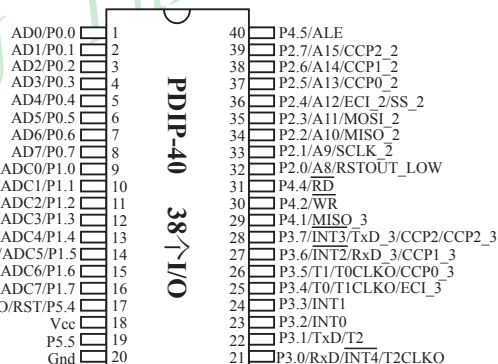
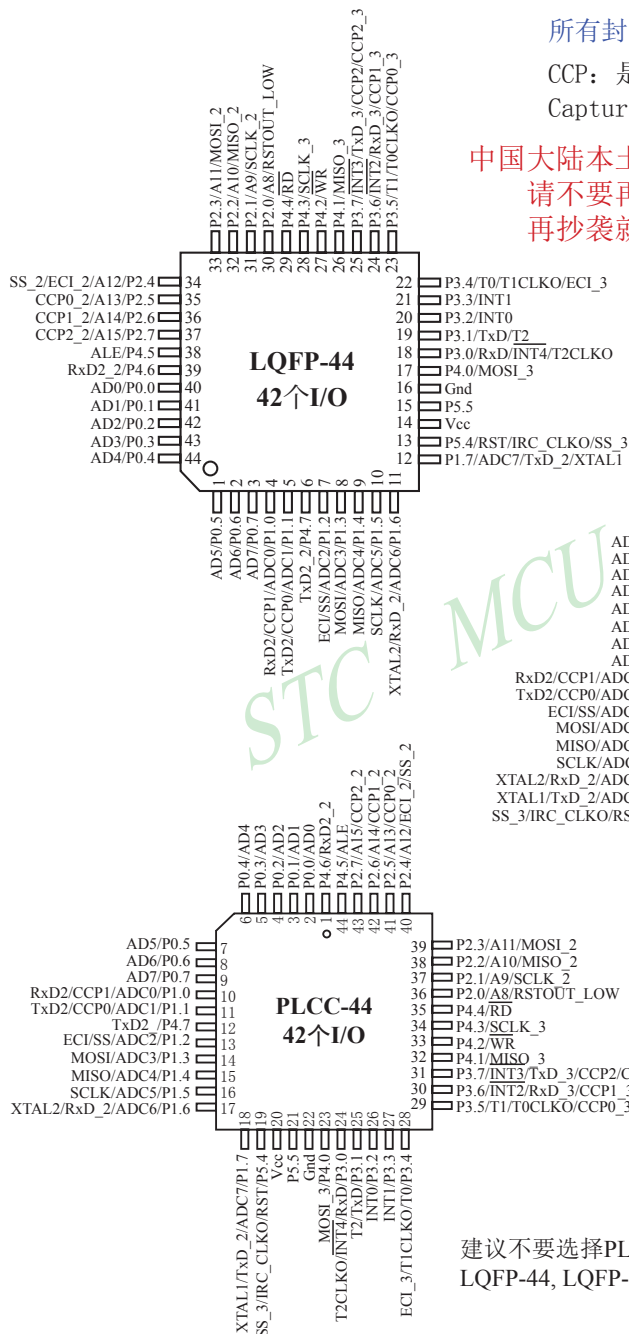
CCP: 是英文单词的缩写

Capture (捕获), Compare (比较), PWM (脉宽调制)

中国大陆本土STC姚永平独立创新设计:
请不要再抄袭我们的设计、规格和管脚排列,
再抄袭就很无耻了。

特别注意: P0口可复用为地址(Address)/数据(Data)总线使用, 不是作A/D转换使用。A/D转换通道在P1口。

因此: 管脚图中P0.x/ADx是指P0.x管脚可作为地址(Address)/数据(Data)总线使用, 而P1.x/ADCx才是指P1.x管脚可作为A/D转换通道使用。



T0CLKO是指定时器T0的时钟输出
(与CLKOUT0同, 有时也写作CLKOUT0);

T1CLKO是指定时器T1的时钟输出
(与CLKOUT1同, 有时也写作CLKOUT1);

T2CLKO是指定时器T2的时钟输出
(与CLKOUT2同, 有时也写作CLKOUT2)。

T0CLKO/T1CLKO/T2CLKO除可以做可编程时钟输出外, 还可以作分频器使用。

建议不要选择PLCC-44封装, 推荐尽量选择
LQFP-44, LQFP-32, SOP-32, SOP-28封装。

Capture(捕获), Compare(比较), PWM(脉宽调制)

The image displays two pin configurations for the AT89C51F1 microcontroller. The left diagram shows the LQFP-32 package, a square package with pins on all four sides. The right diagram shows the SOP-32 package, a smaller package with pins on two opposite sides. Both diagrams include pin numbers and corresponding functions.

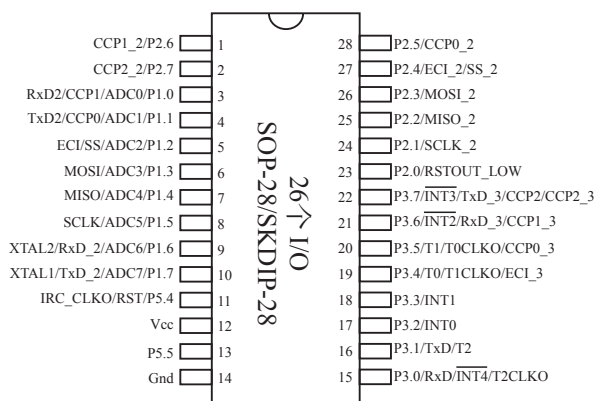
LQFP-32 30个I/O

Pin	Function
1	RxD2/CCP1/ADC0/P1.0
2	TxD2/CCP0/ADC1/P1.1
3	ECI/SS/ADC2/P1.2
4	MOSI/ADC3/P1.3
5	MISO/ADC4/P1.4
6	SCLK/ADC5/P1.5
7	XTAL1/TxD_2/ADC6/P1.6
8	XTAL1/TxD_2/ADC7/P1.7
9	P5.4/RST/IRC_CLKO
10	Vcc
11	P5.5
12	Gnd
13	P3.0/RxD/TxT4/T2CLKO
14	P3.1/TxD/T2
15	P3.2/INT0
16	P3.3/INT1
17	P3.4/T0T1CLKO/ECI_3
18	P3.5/T1/T0CLKO/CCP0_3
19	P3.6/INT2/RxD_3/CCP1_3
20	P3.7/INT3/TxD_3/CCP2/CCP2_3
21	P2.0/RSTOUT_LOW
22	P2.1/SCLK_2
23	P2.2/MISO_2
24	P2.3/MOSI_2
25	SS_2/ECI_2/P2.4
26	CCP0_2/P2.5
27	CCP1_2/P2.6
28	CCP2_2/P2.7
29	P0.0
30	P0.1
31	P0.2
32	P0.3

SOP-32 30个I/O

Pin	Function
1	P0.0
2	P0.1
3	P0.2
4	P0.3
5	RxD2/CCP1/ADC0/P1.0
6	TxD2/CCP0/ADC1/P1.1
7	ECI/SS/ADC2/P1.2
8	MOSI/ADC3/P1.3
9	MISO/ADC4/P1.4
10	SCLK/ADC5/P1.5
11	XTAL2/RxD_2/ADC6/P1.6
12	XTAL1/TxD_2/ADC7/P1.7
13	IRC_CLKO/RST/P5.4
14	Vcc
15	P5.5
16	Gnd
17	P3.0/RxD/TxT4/T2CLKO
18	P3.1/TxD/T2
19	P3.2/INT0
20	P3.3/INT1
21	P3.4/T0T1CLKO/ECI_3
22	P3.5/T1/T0CLKO/CCP0_3
23	P3.6/INT2/RxD_3/CCP1_3
24	P3.7/INT3/TxD_3/CCP2/CCP2_3
25	P2.0/RSTOUT_LOW
26	P2.1/SCLK_2
27	P2.2/MISO_2
28	P2.3/MOSI_2
29	P2.4/ECI_2/SS_2
30	P2.5/CCP0_2
31	P2.6/CCP1_2
32	P2.7/CCP2_2

T0CLKO/T1CLKO/T2CLKO除可以做可编程时钟输出外,还可以作分频器使用。



请不要再抄袭我们的设计、规格和管脚排列，再抄袭就很无耻了。

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
AUXR1 P_SW1	A2H	Auxiliary register 1	S1_S0	CCP_S0	SPI_S0	S2_S0	GF2	ADJ	0	DPS	0000,0000
P_SW2	BAH	Peripheral function switch	S1_S1	CCP_S1	SPI_S1						000x,xxxx
IRC_CLK0	BBH	Internal R/C clock output register	-	-	ALE_P4.5	-	-	-	IRCS1	IRCS0	xx0x,xx00

CCP可在3个地方切换，由 CCP_S1 / CCP_S0 两个控制位来选择

CCP_S1	CCP_S0	CCP可在P1/P2/P3之间来回切换
0	0	CCP在[P1. 2/ECI, P1. 1/CCP0, P1. 0/CCP1, P3. 7/CCP2]
0	1	CCP在[P2. 4/ECI_2, P2. 5/CCP0_2, P2. 6/CCP1_2, P2. 7/CCP2_2]
1	0	CCP在[P3. 4/ECI_3, P3. 5/CCP0_3, P3. 6/CCP1_3, P3. 7/CCP2_3]
1	1	无效

SPI可在3个地方切换，由 SPI_S1 / SPI_S0 两个控制位来选择

SPI_S1	SPI_S0	SPI可在P1/P2/P4之间来回切换
0	0	SPI在[P1. 2/SS, P1. 3/MOSI, P1. 4/MISO, P1. 5/SCLK]
0	1	SPI在[P2. 4/SS_2, P2. 3/MOSI_2, P2. 2/MISO_2, P2. 1/SCLK_2]
1	0	SPI在[P5. 4/SS_3, P4. 0/MOSI_3, P4. 1/MISO_3, P4. 3/SCLK_3]
1	1	无效

串口/S1可在3个地方切换，由 S1_S0 及 S1_S1 控制位来选择

S1_S1	S1_S0	串口/S1可在P1/P3之间来回切换
0	0	串口/S1在[P3. 0/RxD, P3. 1/TxD]
0	1	串口/S1在[P1. 6/RxD_2/XTAL2, P1. 7/TxD_2/XTAL1] 串口在P1口时要使用内部时钟
1	0	串口/S1在[P3. 6/RxD_3, P3. 7/TxD_3]
1	1	无效

串口2/S2可在2个地方切换，由 S2_S0 控制位来选择

S2_S0	S2可在P1/P4之间来回切换
0	串口2/S2在[P1. 0/RxD2, P1. 1/TxD2]
1	串口2/S2在[P4. 6/RxD2_2, P4. 7/TxD2_2]

ALE/P4. 5:

- 0, 复位后IRC_CLK0. 5=0, ALE/P4. 5脚是ALE信号, 只有在用MOVX指令访问片外扩展器件时才有信号输出
1, 通过设置IRC_CLK0. 5= 1, 将ALE/P4. 5脚设置成I/O口 (P4. 5)

1.3.4 STC15F2K60S2系列单片机选型一览表

型号	工作电压(V)	Flash程序存储器(byte)	大容量SRAM字节	串行口	SPI	普通定时器	CCP/PCA/PWM定时器	掉电唤醒专用定时器	A/D 8路	看门狗	内置复位	EEPROM	内部低压检测中断	内部可选复位门限电压	支持掉电唤醒外部中断	所有封装 LQFP44/PDIP40/ PLCC44/SOP32/ LQFP32/SOP28/ SKDIP28 (请尽量不要选择PLCC封装)	
																部分封装 价格(RMB ¥)	
																LQFP44	SOP28
STC15F104ESW系列单片机选型一览表																	
STC15F2K08S2	5.5-3.8	8K	2K	2	有	3	3-ch	有	10位	有	有	2K	有	8级	5	¥4.5	
STC15F2K16S2	5.5-3.8	16K	2K	2	有	3	3-ch	有	10位	有	有	45K	有	8级	5	¥4.7	
STC15F2K20S2	5.5-3.8	20K	2K	2	有	3	3-ch	有	10位	有	有	41K	有	8级	5	¥4.8	
STC15F2K32S2	5.5-3.8	32K	2K	2	有	3	3-ch	有	10位	有	有	29K	有	8级	5	¥4.9	
STC15F2K40S2	5.5-3.8	40K	2K	2	有	3	3-ch	有	10位	有	有	21K	有	8级	5	¥5.5	
STC15F2K48S2	5.5-3.8	48K	2K	2	有	3	3-ch	有	10位	有	有	13K	有	8级	5	¥5.5	
STC15F2K52S2	5.5-3.8	52K	2K	2	有	3	3-ch	有	10位	有	有	9K	有	8级	5	¥5.5	
STC15F2K56S2	5.5-3.8	56K	2K	2	有	3	3-ch	有	10位	有	有	5K	有	8级	5	¥5.5	
STC15F2K60S2	5.5-3.8	60K	2K	2	有	3	3-ch	有	10位	有	有	1K	有	8级	5	¥5.5	
IAP15F2K62S2	5.5-3.8	62K	2K	2	有	3	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区修改用户程序	
STC15L104ESW系列单片机选型一览表																	
STC15L2K08S2	2.4-3.6	8K	2K	2	有	3	3-ch	有	10位	有	有	2K	有	8级	5	¥4.5	
STC15L2K16S2	2.4-3.6	16K	2K	2	有	3	3-ch	有	10位	有	有	45K	有	8级	5	¥4.7	
STC15L2K20S2	2.4-3.6	20K	2K	2	有	3	3-ch	有	10位	有	有	41K	有	8级	5	¥4.8	
STC15L2K32S2	2.4-3.6	32K	2K	2	有	3	3-ch	有	10位	有	有	29K	有	8级	5	¥4.9	
STC15L2K40S2	2.4-3.6	40K	2K	2	有	3	3-ch	有	10位	有	有	21K	有	8级	5	¥5.5	
STC15L2K48S2	2.4-3.6	48K	2K	2	有	3	3-ch	有	10位	有	有	13K	有	8级	5	¥5.5	
STC15L2K52S2	2.4-3.6	52K	2K	2	有	3	3-ch	有	10位	有	有	9K	有	8级	5	¥5.5	
STC15L2K56S2	2.4-3.6	56K	2K	2	有	3	3-ch	有	10位	有	有	5K	有	8级	5	¥5.5	
STC15L2K60S2	2.4-3.6	60K	2K	2	有	3	3-ch	有	10位	有	有	1K	有	8级	5	¥5.5	
IAP15L2K62S2	2.4-3.6	62K	2K	2	有	3	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区修改用户程序	

提供客制化IC服务

以上单价为200K起订
量小每片需加0.3元-1元
以上价格运费由客户承担, 零售1片起
如对价格不满, 可来电要求降价

1.3.3 STC15F4K60S4系列单片机管脚图

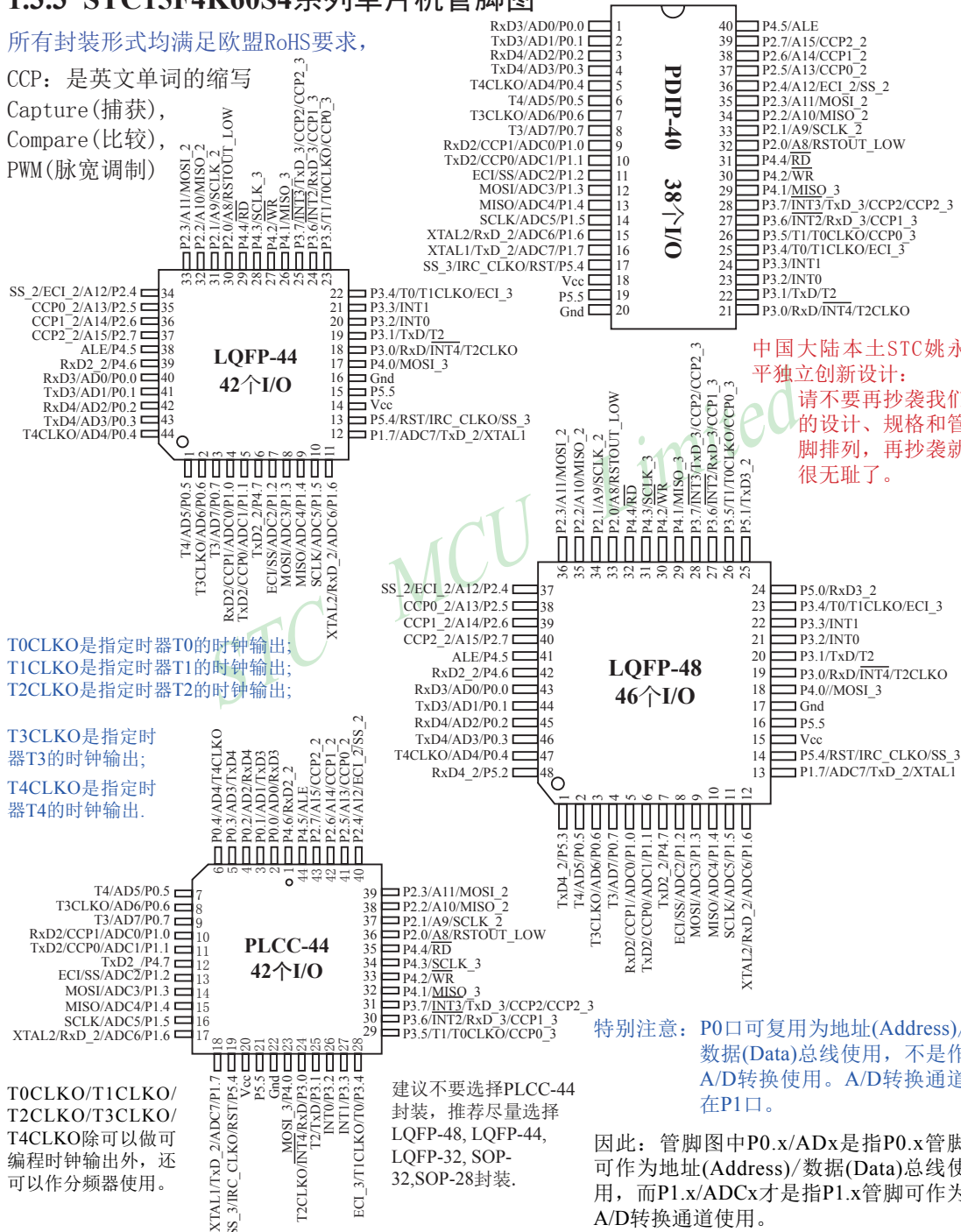
所有封装形式均满足欧盟RoHS要求,

CCP: 是英文单词的缩写

Capture(捕获),

Compare(比较),

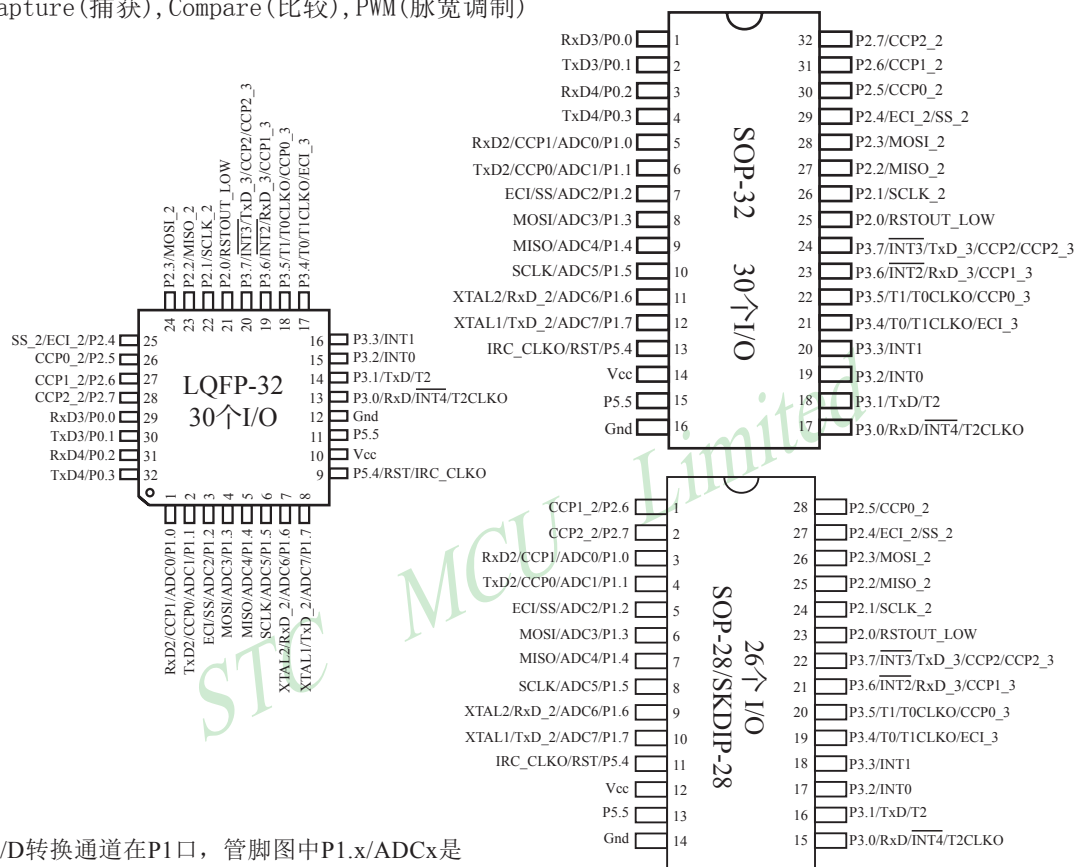
PWM(脉宽调制)



所有封装形式均满足欧盟RoHS要求，

CCP：是英文单词的缩写

Capture (捕获), Compare (比较), PWM (脉宽调制)



A/D转换通道在P1口，管脚图中P1.x/ADCx是指P1.x管脚可作为A/D转换通道使用。

中国大陆本土STC姚永平独立创新设计：
请不要再抄袭我们的设计、规格和管脚排列，
再抄袭就很无耻了。

T0CLKO是指定时器T0的时钟输出
(与CLKOUT0同，有时也写作CLKOUT0)；

T1CLKO是指定时器T1的时钟输出
(与CLKOUT1同，有时也写作CLKOUT1)；

T2CLKO是指定时器T2的时钟输出
(与CLKOUT2同，有时也写作CLKOUT2)；

T3CLKO是指定时器T3的时钟输出
(与CLKOUT3同，有时也写作CLKOUT3)；

T4CLKO是指定时器T4的时钟输出
(与CLKOUT4同，有时也写作CLKOUT4)。

T0CLKO/T1CLKO/T2CLKO/T3CLKO/T4CLKO
除可以做可编程时钟输出外，还可以作分频器使用。

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
AUXR1 P_SW1	A2H	Auxiliary register 1	S1_S0	CCP_S0	SPI_S0	S2_S0	GF2	ADJ	0	DPS	0000,0000
P_SW2	BAH	Peripheral function switch	S1_S1	CCP_S1	SPI_S1				S4_S0	S3_S0	000x,xx00
IRC_CLKO	BBH	Internal R/C clock output register	-	-	ALE_P4.5	-	T4CLKO	T3CLKO	IRCS1	IRCS0	000x,0000

CCP可在3个地方切换，由 CCP_S1 / CCP_S0 两个控制位来选择

CCP_S1	CCP_S0	CCP可在P1/P2/P3之间来回切换
0	0	CCP在[P1. 2/ECI, P1. 1/CCP0, P1. 0/CCP1, P3. 7/CCP2]
0	1	CCP在[P2. 4/ECI_2, P2. 5/CCP0_2, P2. 6/CCP1_2, P2. 7/CCP2_2]
1	0	CCP在[P3. 4/ECI_3, P3. 5/CCP0_3, P3. 6/CCP1_3, P3. 7/CCP2_3]
1	1	无效

SPI可在3个地方切换，由 SPI_S1 / SPI_S0 两个控制位来选择

SPI_S1	SPI_S0	SPI可在P1/P2/P4之间来回切换
0	0	SPI在[P1. 2/SS, P1. 3/MOSI, P1. 4/MISO, P1. 5/SCLK]
0	1	SPI在[P2. 4/SS_2, P2. 3/MOSI_2, P2. 2/MISO_2, P2. 1/SCLK_2]
1	0	SPI在[P5. 4/SS_3, P4. 0/MOSI_3, P4. 1/MISO_3, P4. 3/SCLK_3]
1	1	无效

串口/S1可在3个地方切换，由 S1_S0 及 S1_S1 控制位来选择

S1_S1	S1_S0	串口/S1可在P1/P3之间来回切换
0	0	串口/S1在[P3. 0/RxD, P3. 1/TxD]
0	1	串口/S1在[P1. 6/RxD_2/XTAL2, P1. 7/TxD_2/XTAL1] 串口在P1口时要使用内部时钟
1	0	串口/S1在[P3. 6/RxD_3, P3. 7/TxD_3]
1	1	无效

串口2/S2可在2个地方切换，由 S2_S0 控制位来选择

S2_S0	S2可在P1/P4之间来回切换
0	串口2/S2在[P1. 0/RxD2, P1. 1/TxD2]
1	串口2/S2在[P4. 6/RxD2_2, P4. 7/TxD2_2]

串口3/S3可在2个地方切换，由 S3_S0 控制位来选择	
S3_S0	S3可在P0/P5之间来回切换
0	串口3/S3在[P0. 0/RxD3, P0. 1/TxD3]
1	串口3/S3在[P5. 0/RxD3_2, P5. 1/TxD3_2]

串口4/S4可在2个地方切换，由 S4_S0 控制位来选择	
S4_S0	S4可在P0/P5之间来回切换
0	串口4/S4在[P0. 2/RxD4, P0. 3/TxD4]
1	串口4/S4在[P5. 2/RxD4_2, P5. 3/TxD4_2]

ALE/P4. 5:

0, 复位后IRC_CLK0. 5=0, ALE/P4. 5脚是ALE信号, 只有在用MOVX指令访问片外扩展器件时才有信号输出

1, 通过设置IRC_CLK0. 5= 1, 将ALE/P4. 5脚设置成I/O口 (P4. 5)

STC MCU Limited

1.3.4 STC15F4K60S4系列单片机选型一览表

型号	工作电压(V)	Flash程序存储器(byte)	大容量SRAM字节	串行口	SPI	普通定时器	CCP/PCA/PWM定时器	掉电唤醒专用定时器	A/D 8路	看门狗	内置复位	EEPROM	内部低压检测中断	内部可选复位门电压	支持掉电唤醒外部中断	所有封装 LQFP48/LQFP44/ PDIP40/PLCC44/ SOP32/LQFP32/ SOP28/SKDIP28 (请尽量不要选择PLCC封装)	
																部分封装 价格(RMB ¥)	
																LQFP48	LQFP44
STC15F4K60S4系列单片机选型一览表																	
STC15F4K08S4	5.5-3.8	8K	4K	4	有	5	3-ch	有	10位	有	有	2K	有	8级	5	¥8	
STC15F4K16S4	5.5-3.8	16K	4K	4	有	5	3-ch	有	10位	有	有	45K	有	8级	5	¥9	
STC15F4K20S4	5.5-3.8	20K	4K	4	有	5	3-ch	有	10位	有	有	41K	有	8级	5	¥11	
STC15F4K32S4	5.5-3.8	34K	4K	4	有	5	3-ch	有	10位	有	有	29K	有	8级	5	¥12	
STC15F4K40S4	5.5-3.8	40K	4K	4	有	5	3-ch	有	10位	有	有	21K	有	8级	5	¥12	
STC15F4K48S4	5.5-3.8	48K	4K	4	有	5	3-ch	有	10位	有	有	13K	有	8级	5	¥12	
STC15F4K52S4	5.5-3.8	52K	4K	4	有	5	3-ch	有	10位	有	有	9K	有	8级	5	¥12	
STC15F4K56S4	5.5-3.8	56K	4K	4	有	5	3-ch	有	10位	有	有	5K	有	8级	5	¥12	
STC15F4K60S4	5.5-3.8	60K	4K	4	有	5	3-ch	有	10位	有	有	1K	有	8级	5	¥12	
IAP15F4K62S4	5.5-3.8	62K	4K	4	有	5	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区修改用户程序	
STC15L4K60S4系列单片机选型一览表																	
STC15L4K08S4	2.4-3.6	8K	4K	4	有	5	3-ch	有	10位	有	有	2K	有	8级	5	¥8	
STC15L4K16S4	2.4-3.6	16K	4K	4	有	5	3-ch	有	10位	有	有	45K	有	8级	5	¥9	
STC15L4K20S4	2.4-3.6	20K	4K	4	有	5	3-ch	有	10位	有	有	41K	有	8级	5	¥11	
STC15L4K32S4	2.4-3.6	34K	4K	4	有	5	3-ch	有	10位	有	有	29K	有	8级	5	¥12	
STC15L4K40S4	2.4-3.6	40K	4K	4	有	5	3-ch	有	10位	有	有	21K	有	8级	5	¥12	
STC15L4K48S4	2.4-3.6	48K	4K	4	有	5	3-ch	有	10位	有	有	13K	有	8级	5	¥12	
STC15L4K52S4	2.4-3.6	52K	4K	4	有	5	3-ch	有	10位	有	有	9K	有	8级	5	¥12	
STC15L4K56S4	2.4-3.6	56K	4K	4	有	5	3-ch	有	10位	有	有	5K	有	8级	5	¥12	
STC15L4K60S4	2.4-3.6	60K	4K	4	有	5	3-ch	有	10位	有	有	1K	有	8级	5	¥12	
IAP15L4K62S4	2.4-3.6	62K	4K	4	有	5	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区修改用户程序	

提供客制化IC服务

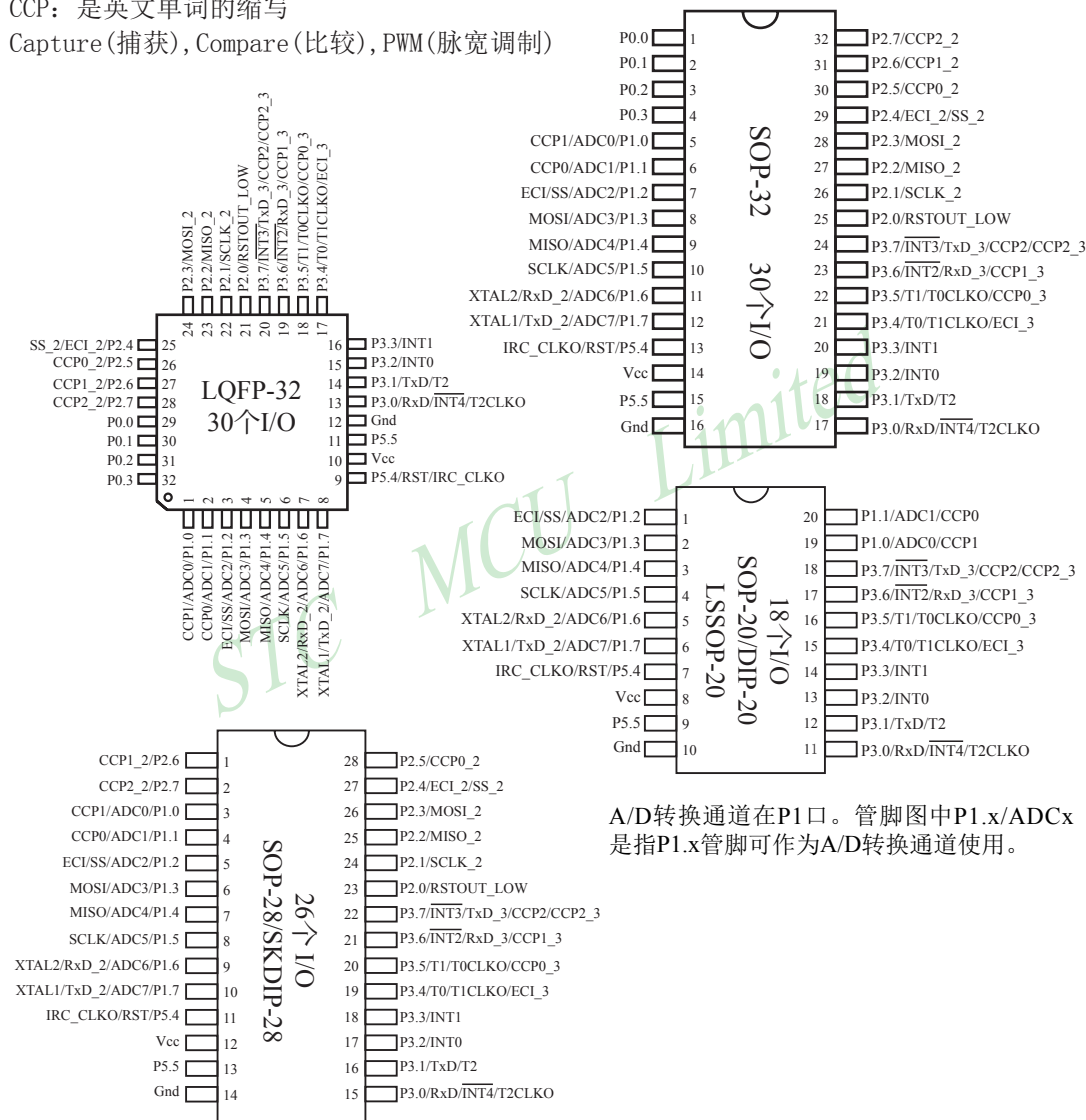
以上单价为200K起订
量小每片需加0.3元-1元
以上价格运费由客户承担, 零售1片起
如对价格不满, 可来电要求降价

1.3.5 STC15F1K20AD系列单片机管脚图 (2012年5月开始供货)

所有封装形式均满足欧盟RoHS要求，

CCP: 是英文单词的缩写

Capture (捕获), Compare (比较), PWM (脉宽调制)



A/D转换通道在P1口。管脚图中P1.x/ADCx是指P1.x管脚可作为A/D转换通道使用。

T0CLKO是指定时器T0的时钟输出 (与CLKOUT0同, 有时也写作CLKOUT0);

T1CLKO是指定时器T1的时钟输出 (与CLKOUT1同, 有时也写作CLKOUT1);

T2CLKO是指定时器T2的时钟输出 (与CLKOUT2同, 有时也写作CLKOUT2)。

中国大陆本土STC姚永平独立创新设计;

请不要再抄袭我们的设计、规格和管脚排列, 再抄袭就很无耻了。

T0CLKO/T1CLKO/T2CLKO除可以做可编程时钟输出外, 还可以作分频器使用。

1.3.6 STC15F1K20AD系列单片机选型一览表

型号	工作电压(V)	Flash程序存储器(byte)	大容量SRAM字节	串行口	SPI	普通定时器	CCP/PCA/PWM定时器	掉电唤醒专用定时器	A/D 8路	看门狗	内置复位	EEPROM	内部低压检测中断	内部可选复位阈值电压	支持掉电唤醒外部中断	所有封装 SOP-32/LQFP-32/ SOP-28/SKDIP-28/ SOP-20/DIP-20/ LSSOP-20	
																部分封装 价格(RMB ¥)	
																SOP-32	SOP-28
STC15F1K20AD系列单片机选型一览表																	
STC15F1K04AD	5.5-3.8	4K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F1K08AD	5.5-3.8	8K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F1K12AD	5.5-3.8	12K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F1K16AD	5.5-3.8	16K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F1K20AD	5.5-3.8	20K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F1K24AD	5.5-3.8	24K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
IAP15F1K28AD	5.5-3.8	28K	1K	1	有	3	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	
STC15L1K20AD系列单片机选型一览表																	
STC15L1K04AD	2.4-3.6	4K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L1K08AD	2.4-3.6	8K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L1K12AD	2.4-3.6	12K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L1K16AD	2.4-3.6	16K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L1K20AD	2.4-3.6	20K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L1K24AD	2.4-3.6	24K	1K	1	有	3	3-ch	有	10位	有	有	4K	有	8级	5		
IAP15L1K28AD	2.4-3.6	28K	1K	1	有	3	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	

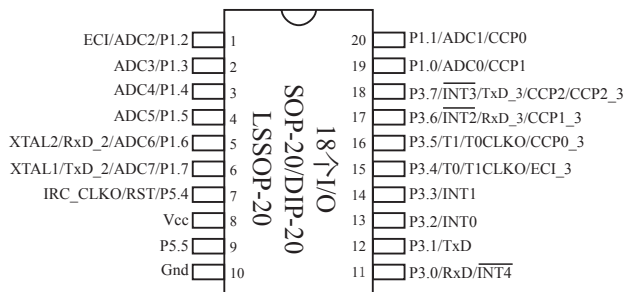
提供客制化IC服务

1.3.7 STC15F412EACS系列单片机管脚图

所有封装形式均满足欧盟RoHS要求，

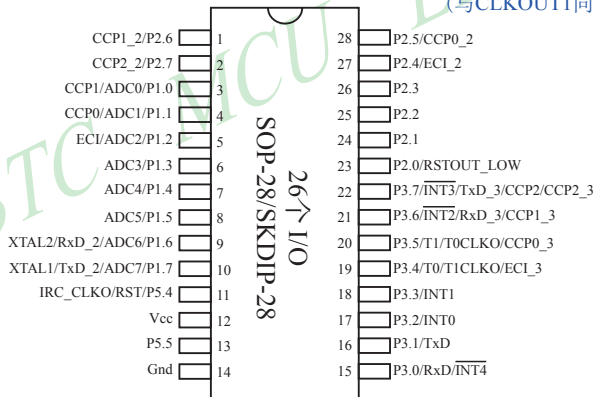
CCP：是英文单词的缩写

Capture (捕获), Compare (比较), PWM (脉宽调制)



A/D转换通道在P1口。管脚图中P1.x/ADCx是指P1.x管脚可作为A/D转换通道使用。

T0CLKO是指定时器T0的时钟输出(与CLKOUT0同，有时也写作CLKOUT0)；
T1CLKO是指定时器T1的时钟输出(与CLKOUT1同，有时也写作CLKOUT1)。



T0CLKO/T1CLKO除可以做可编程时钟输出外，还可以作分频器使用。

中国大陆本土STC姚永平独立创新设计：

请不要再抄袭我们的设计、规格和管脚排列，
再抄袭就很无耻了。

1.3.8 STC15F412EACS系列单片机选型一览表

型号	工作电压 (V)	Flash 程序 存储器 (byte)	大容量 SRAM 字节	串 行 口	普 通 定 时 器	CCP PCA PWM 定 时 器	掉电 唤醒 专用 定 时 器	A/D 8路	看 门 狗	内 置 复 位	EEP ROM	内 部 低 压 检 测 中 断	内 部 可 选 复 位 门 电 压	支持 掉电 唤醒 外部 中 断	所有封装 SOP-28/SKDIP-28/ SOP-20/DIP-20/ LSSOP-20	
															部分封装 价格(RMB ¥)	
															SOP20	SOP28
STC15F412EACS系列单片机选型一览																
STC15F402EACS	5.5-3.8	2K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F404EACS	5.5-3.8	4K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F408EACS	5.5-3.8	8K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15F410EACS	5.5-3.8	10K	512	1	2	3-ch	有	10位	有	有	3K	有	8级	5		
STC15F412EACS	5.5-3.8	12K	512	1	2	3-ch	有	10位	有	有	1K	有	8级	5		
IAP15F413EACS	5.5-3.8	13K	512	1	2	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	
STC15L412EACS系列单片机选型一览表																
STC15L402EACS	2.4-3.6	2K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L404EACS	2.4-3.6	4K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L408EACS	2.4-3.6	8K	512	1	2	3-ch	有	10位	有	有	4K	有	8级	5		
STC15L410EACS	2.4-3.6	10K	512	1	2	3-ch	有	10位	有	有	3K	有	8级	5		
STC15L412EACS	2.4-3.6	12K	512	1	2	3-ch	有	10位	有	有	1K	有	8级	5		
IAP15L413EACS	2.4-3.6	13K	512	1	2	3-ch	有	10位	有	有	IAP	有	8级	5	用户可在程序区 修改用户程序	

提供客制化IC服务

1.3.9 STC15F204EA系列单片机管脚图

——A版本现已供货，B版本2012年4月~6月开始供货

中国大陆本土STC姚永平独立创新设计：

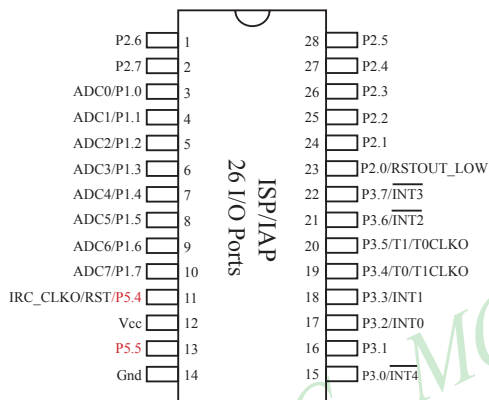
请不要再抄袭我们的设计、规格和管脚排列，
再抄袭就很无耻了。

所有封装形式均满足欧盟RoHS要求，

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

现STC15F204EA系列单片机A版本已开始供货，B版本将在2012年4月—6月开始供货

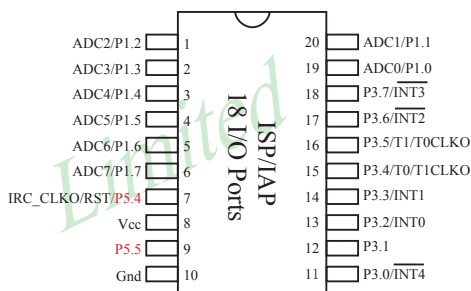
STC15F204EA系列管脚图



SOP-28/SKDIP-28

CCP：是英文单词的缩写

Capture (捕获), Compare (比较), PWM (脉宽调制)



SOP-20/DIP-20/LSSOP-20

以上为STC15F204EA系列B版本管脚图

特别声明：A版本和B版本管脚图中有两个管脚有差别

B版本中为	P5.4/RST/IRC_CLKO	P5.5
A版本中为	P0.0/RST/IRC_CLKO	P0.1

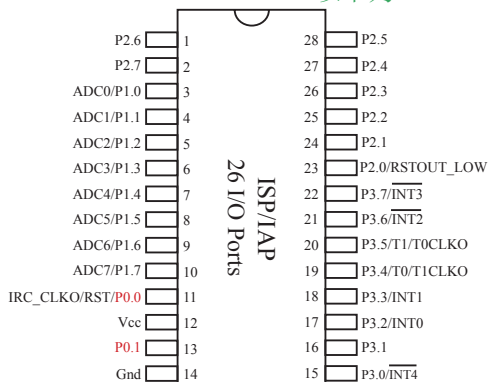
T0CLKO是指定时器T0的时钟输出

(与CLKOUT0同，有时也写作CLKOUT0)；

T1CLKO是指定时器T1的时钟输出

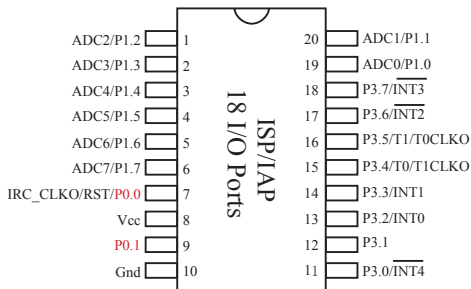
(与CLKOUT1同，有时也写作CLKOUT1)。

以下为STC15F204EA系列A版本管脚图



SOP-28/SKDIP-28

T0CLKO/T1CLKO除可以做可编程时钟输出外，还可以作分频器使用。



SOP-20/DIP-20

STC15F204EA系列单片机的A版本无LSSOP-20封装

1.3.10 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	内部可选复位门电压	支持掉电唤醒外部中断	掉电唤醒专用定时器	封装28-Pin SOP-28/SKDIP-28 (26个I/O口) 价格(RMB ¥)		封装20-Pin SOP-20/DIP-20/ LSSOP-20 (18个I/O口) 价格(RMB ¥)	
													SOP-28	SKDIP-28	SOP-20	DIP-20
STC15F204EA系列单片机选型一览表																
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	-	¥2.35	¥2.55	¥2.35	¥2.55
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	-	¥2.40	¥2.60	¥2.40	¥2.60
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	-	¥2.45	¥2.65	¥2.45	¥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	-	¥2.50	¥2.70	¥2.50	¥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	-	¥2.55	¥2.75	¥2.55	¥2.75
IAP15F206A	5.5-3.8	6K	256	2	10位	有	有	IAP	有	8级	5	-	用户可在程序区直接修改程序			
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	-	¥2.35	¥2.55	¥2.35	¥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	-	¥2.40	¥2.60	¥2.40	¥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	-	¥2.45	¥2.65	¥2.45	¥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	-	¥2.50	¥2.70	¥2.50	¥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	-	¥2.55	¥2.75	¥2.55	¥2.75
IAP15L206A	3.6-2.4	6K	256	2	10位	有	有	IAP	有	8级	5	-	用户可在程序区直接修改程序			

提供客制化IC服务

STC15F204EA系列A版本单片机无LSSOP-20封装;
但STC15F204EA系列A版本单片机有LSSOP-20封装。
现STC15F204EA系列单片机A版本已供货,

B版本2012年4-6月开始供货。

以上单价为200K起订

量小每片需加0.3元-1元

以上价格运费由客户承担, 零售1片起

如对价格不满, 可来电要求降价

计划在STC15F204EA系列B版本的部分单片机中增加内部低功耗的掉电唤醒专用定时器

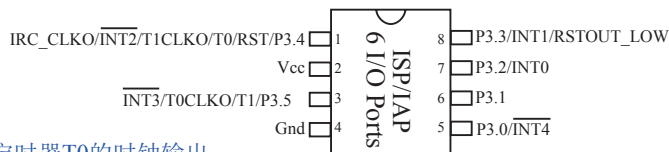
因为程序区的最后7个字节单元被强制性的放入全球唯一ID号的内容, 所以用户实际可以使用的
程序空间大小要比选型表中的大小少7个字节。

1.3.11 STC15F104E系列单片机管脚图

——A版本现已供货，D版本2012年3月开始供货

所有封装形式均满足欧盟RoHS要求，强烈推荐选择SOP-8贴片封装，传统的插件DIP-8封装稳定供货

现STC15F104E系列单片机A版本已开始供货，D版本将在2012年3月开始供货



T0CLKO是指定时器T0的时钟输出

(与CLKOUT0同，有时也写作CLKOUT0); SOP-8/DIP-8

T1CLKO是指定时器T1的时钟输出

(与CLKOUT1同，有时也写作CLKOUT1).

T0CLKO/T1CLKO除可以做可编程时钟输出外，还可以作分频器使用。

中国大陆本土STC姚永平独立创新设计：

请不要再抄袭我们的设计、规格和管脚排列，再抄袭就很无耻了。

1.3.12 STC15F104E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
STC15F104E系列单片机选型一览表														
STC15F100	5.5-3.8	512	128	2	-	有	有	-	有	8级	5	-	¥0.99	¥1.19
STC15F101	5.5-3.8	1K	128	2	-	有	有	-	有	8级	5	-	¥1.20	¥1.40
STC15F101E	5.5-3.8	1K	128	2	-	有	有	2K	有	8级	5	-	¥1.25	¥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	-		
STC15L104E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	-		

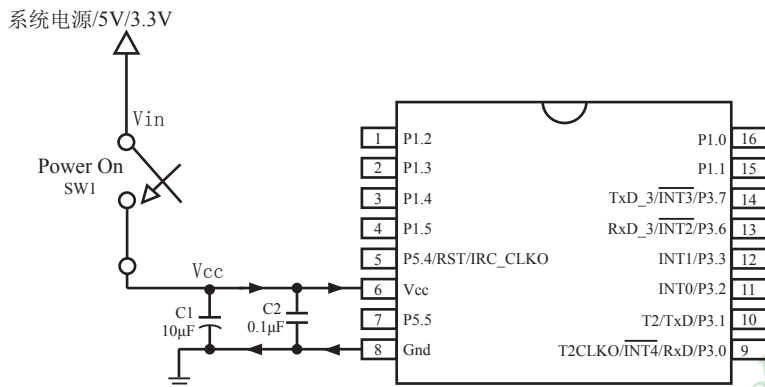
提供客制化IC服务

以上单价为200K起订
量小每片需加0.3元-1元
以上价格运费由客户承担, 零售1片起
如对价格不满, 可来电要求降价

计划在STC15F104E系列D版本的部分单片机中增加内部低功耗的掉电唤醒专用定时器

因为程序区的最后7个字节单元被强制性的放入全球唯一ID号的内容, 所以用户实际可以使用的程序空间大小要比选型表中的大小少7个字节。

1.4 STC15F104ESW系列单片机最小应用系统



内部高可靠复位，不需要外部复位电路

P5.4/RST/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 $\pm 1\%$ ($-40^{\circ}\text{C} \sim +85^{\circ}\text{C}$)，常温下温飘 5% ，不需要昂贵的外部晶振

建议加上电容C1($10\mu\text{F}$)，C2($0.1\mu\text{F}$)，可去除电源噪声，提高抗干扰能力

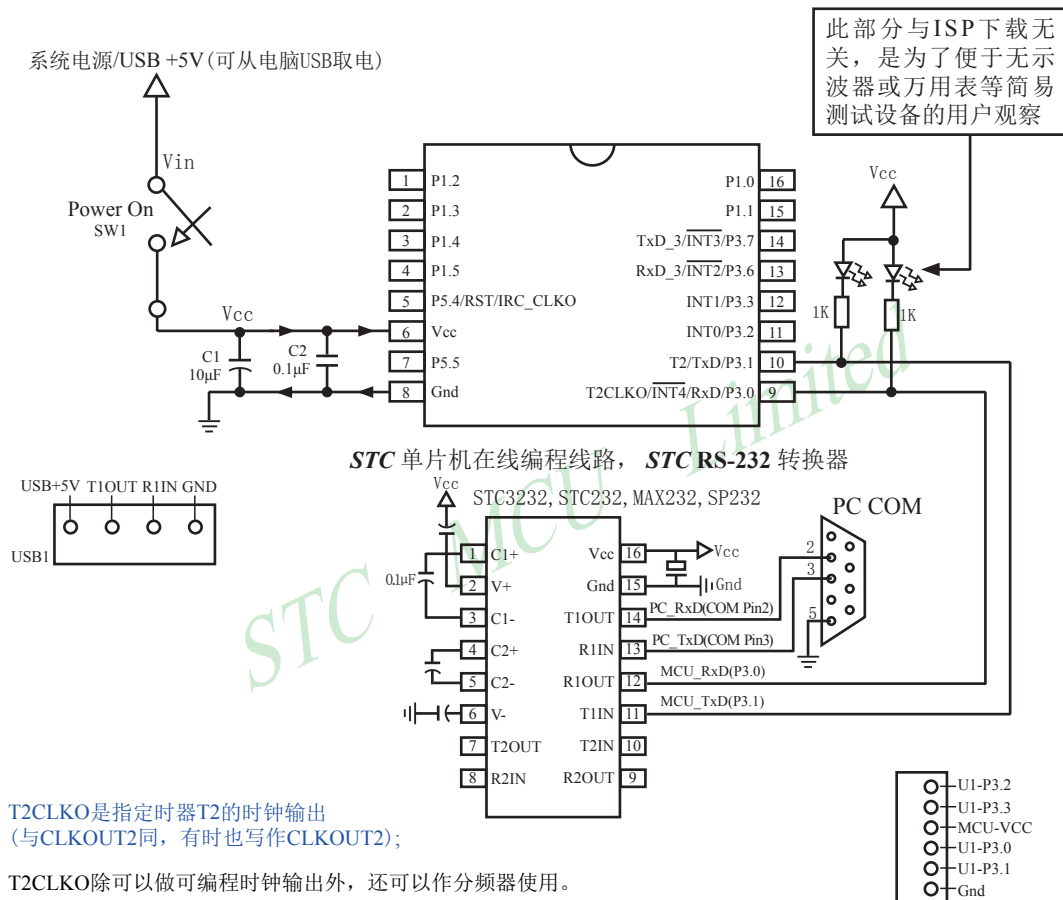
T2CLKO是指定时器T2的时钟输出

(与CLKOUT2同，有时也写作CLKOUT2)；

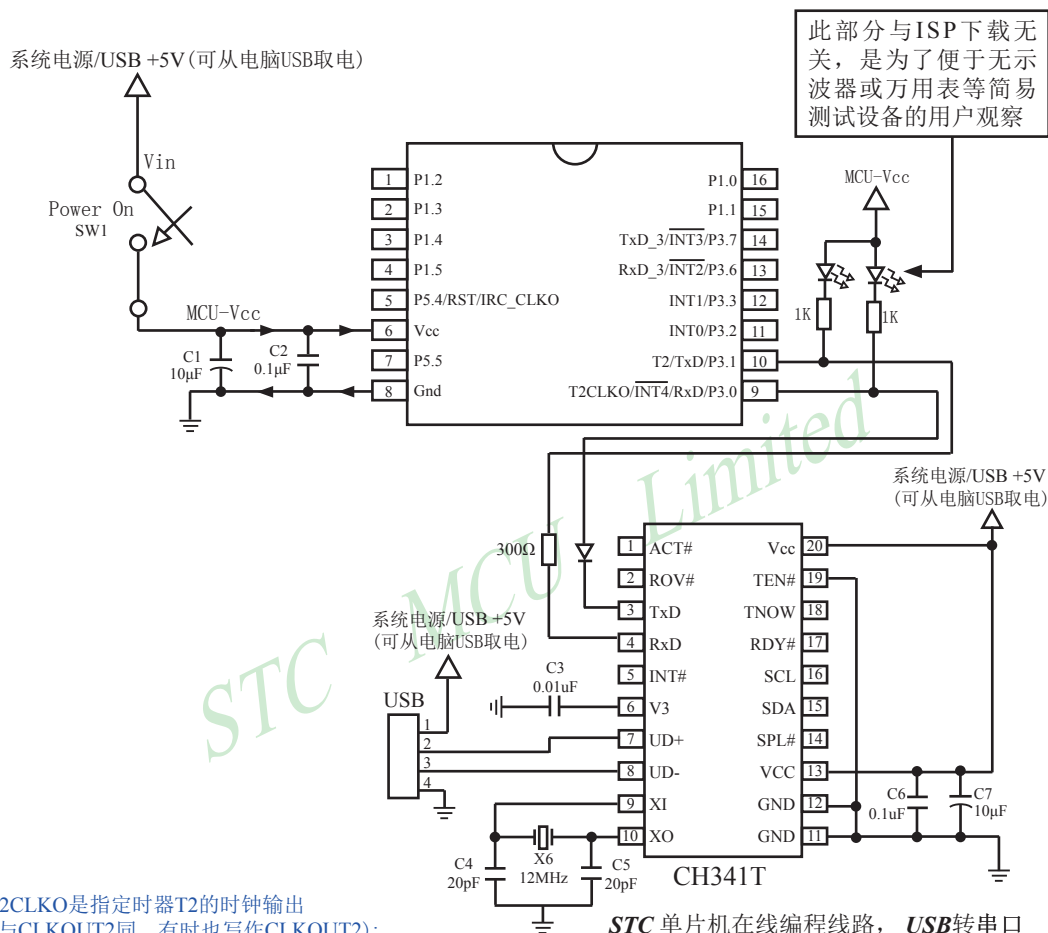
T2CLKO除可以做可编程时钟输出外，还可以作分频器使用。

1.5 STC15F104ESW系列在系统可编程(ISP)典型应用线路图

1.5.1 利用RS-232转换器的典型应用线路图



1.5.2 利用USB转串口的典型应用线路图



T2CLKO是指定时器T2的时钟输出
(与CLKOUT2同,有时也写作CLKOUT2);

T2CLKO除可以做可编程时钟输出外,还可以作分频器使用。

内部高可靠复位，不需要外部复位电路

P5.4/RST/IRC CLK0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 $\pm 1\%$ ($-40^{\circ}\text{C}\sim+85^{\circ}\text{C}$)，常温下温飘5‰，不需要昂贵的外部晶振

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

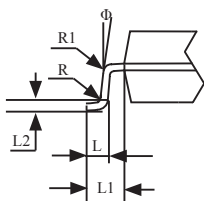
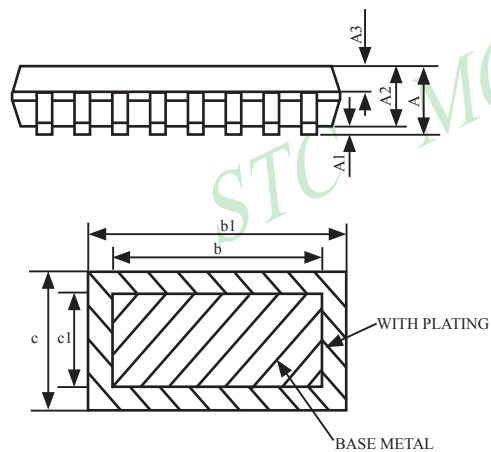
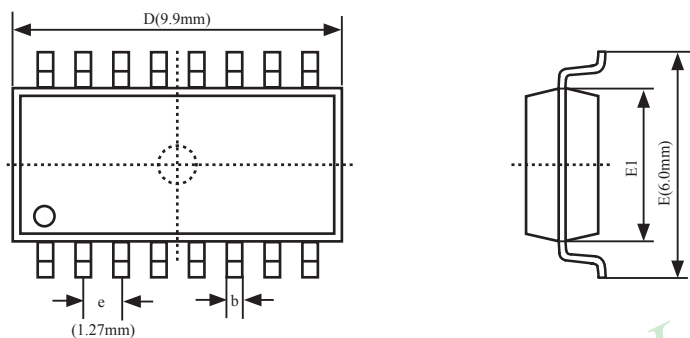
1.6 STC15F104ESW系列管脚说明

管脚	管脚编号 (封装SOP-16/DIP-16)	说明	
P1.0	15	标准I/O口 PORT1[0]	
P1.1	16	标准I/O口 PORT1[1]	
P1.2	1	标准I/O口 PORT1[2]	
P1.3	2	标准I/O口 PORT1[3]	
P1.4	3	标准I/O口 PORT1[4]	
P1.5	4	标准I/O口 PORT1[5]	
P3.0/RxD/ $\overline{\text{INT4}}$ /T2CLKO	9	P3.0	标准I/O口 PORT3[0]
		RxD	串口数据接收端
		$\overline{\text{INT4}}$	外部中断4, 只能下降沿中断, /INT4支持掉电唤醒
		T2CLKO	T2的时钟输出 可通过设置INT_CLKO[2]位/T2CLKO将该管脚配置为T2CLKO
P3.1/TxD/T2	10	P3.1	标准I/O口 PORT3[1]
		TxD	串口数据发送端
		T2	定时器/计数器2的外部输入
P3.2/INT0	11	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	12	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。
P3.6/ $\overline{\text{INT2}}$ /RxD_3	13	P3.6	标准I/O口 PORT3[6]
		$\overline{\text{INT2}}$	外部中断2, 只能下降沿中断 支持掉电唤醒
		RxD_3	串口数据接收端
P3.7/ $\overline{\text{INT3}}$ /TxD_3	14	P3.7	标准I/O口 PORT3[7]
		$\overline{\text{INT3}}$	外部中断3, 只能下降沿中断 支持掉电唤醒
		TxD_3	串口数据发送端
P5.4/RST/ IRC_CLKO	5	P5.4	标准I/O口 PORT5[4]
		RST	复位脚;
		IRC_CLKO	内部R/C振荡时钟输出;输出的频率可为IRC_CLK/1, IRC_CLK/2, IRC_CLK/4
P5.5	7	标准I/O口 PORT5[5]	
Vcc	6	电源正极	
Gnd	8	电源负极, 接地	

1.7 STC15系列单片机封装尺寸图

SOP-16 封装尺寸图

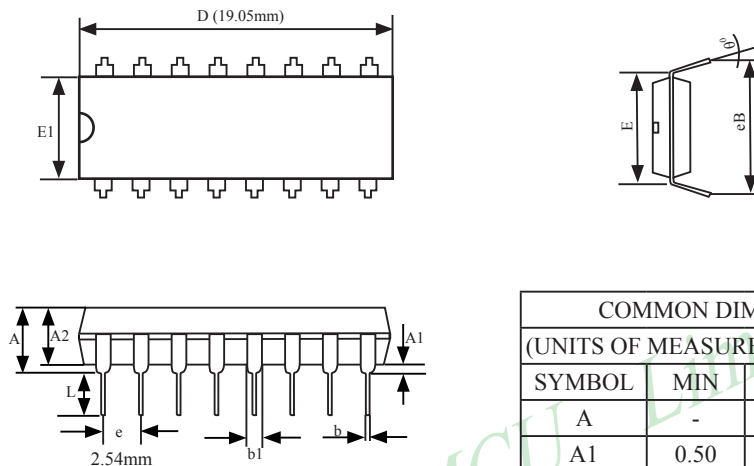
16-PIN SMALL OUTLINE PACKAGE (SOP-16)



COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	1.35	1.60	1.75
A1	0.10	0.15	0.25
A2	1.25	1.45	1.65
A3	0.55	0.65	0.75
b1	0.36	-	0.49
b	0.35	0.40	0.45
c	0.16	-	0.25
c1	0.15	0.20	0.25
D	9.80	9.90	10.00
E	5.80	6.00	6.20
E1	3.80	3.90	4.00
e	1.27		
L	0.45	0.60	0.80
L1	1.04		
L2	0.25		
R	0.07	-	-
R1	0.07	-	-
Φ	6°	8°	10°

DIP-16 封装尺寸图**16-Pin Plastic Dual Inline Package (DIP-16)**

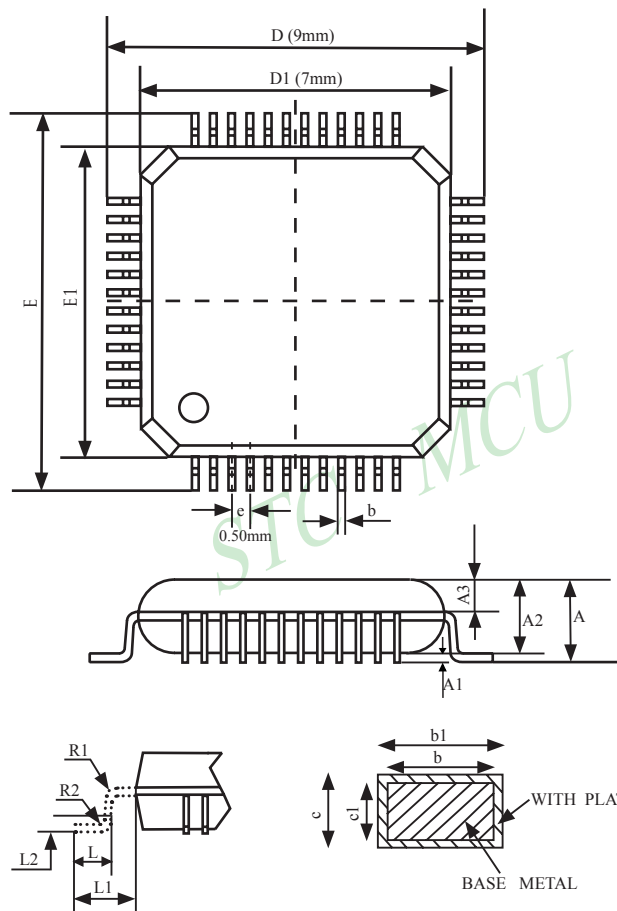
Dimensions in Inches and Millimeters



COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLIMETER)			
SYMBOL	MIN	NOM	MAX
A	-	-	4.80
A1	0.50	-	-
A2	3.10	3.30	3.50
b	0.38	-	0.55
b1	0.38	0.46	0.51
D	18.95	19.05	19.15
E	7.62	7.87	8.25
E1	6.25	6.35	6.45
e	2.54		
eB	7.62	8.80	10.90
L	2.92	3.30	3.81
θ ^{0S}	0	7	15

LQFP-48 封装尺寸图

LQFP-48 OUTLINE PACKAGE

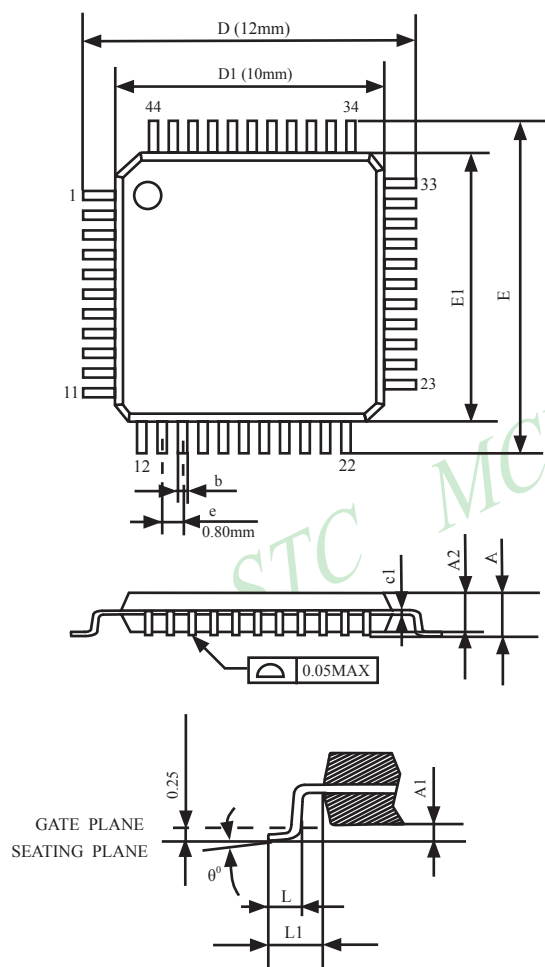


SYMBOL	MIN	NOM	MAX
A	-	-	1.60
A1	0.05	-	0.15
A2	1.35	1.40	1.45
A3	0.59	0.64	0.69
b	0.18	-	0.27
b1	0.17	0.20	0.23
c	0.13	-	0.18
c1	0.12	0.127	0.134
D	8.80	9.00	9.20
D1	6.90	7.00	7.10
E	8.80	9.00	9.20
E1	6.90	7.00	7.10
e	0.50		
L	0.45	0.60	0.75
L1	1.00REF		
L2	0.25		
R1	0.08	-	-
R2	0.08	-	0.20
S	0.20	-	-

VARIATIONS (ALL DIMENSIONS SHOWN IN MM)

LQFP-44 封装尺寸图

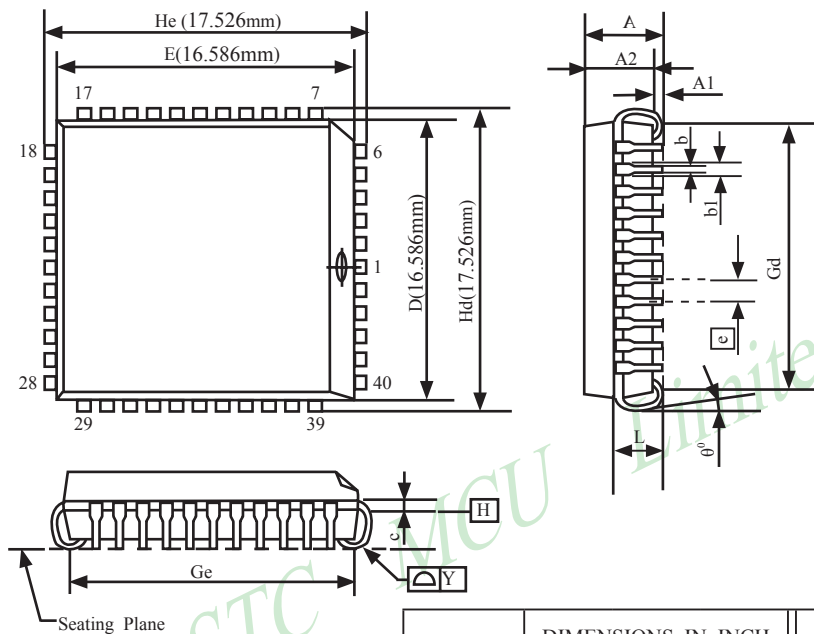
LQFP-44 OUTLINE PACKAGE



VARIATIONS (ALL DIMENSIONS SHOWN IN MM)

SYMBOLS	MIN.	NOM	MAX.
A	-	-	1.60
A1	0.05	-	0.15
A2	1.35	1.40	1.45
c1	0.09	-	0.16
D	12.00		
D1	10.00		
E	12.00		
E1	10.00		
e	0.80		
b(w/o plating)	0.25	0.30	0.35
L	0.45	0.60	0.75
L1	1.00REF		
θ^0	0^0	3.5^0	7^0



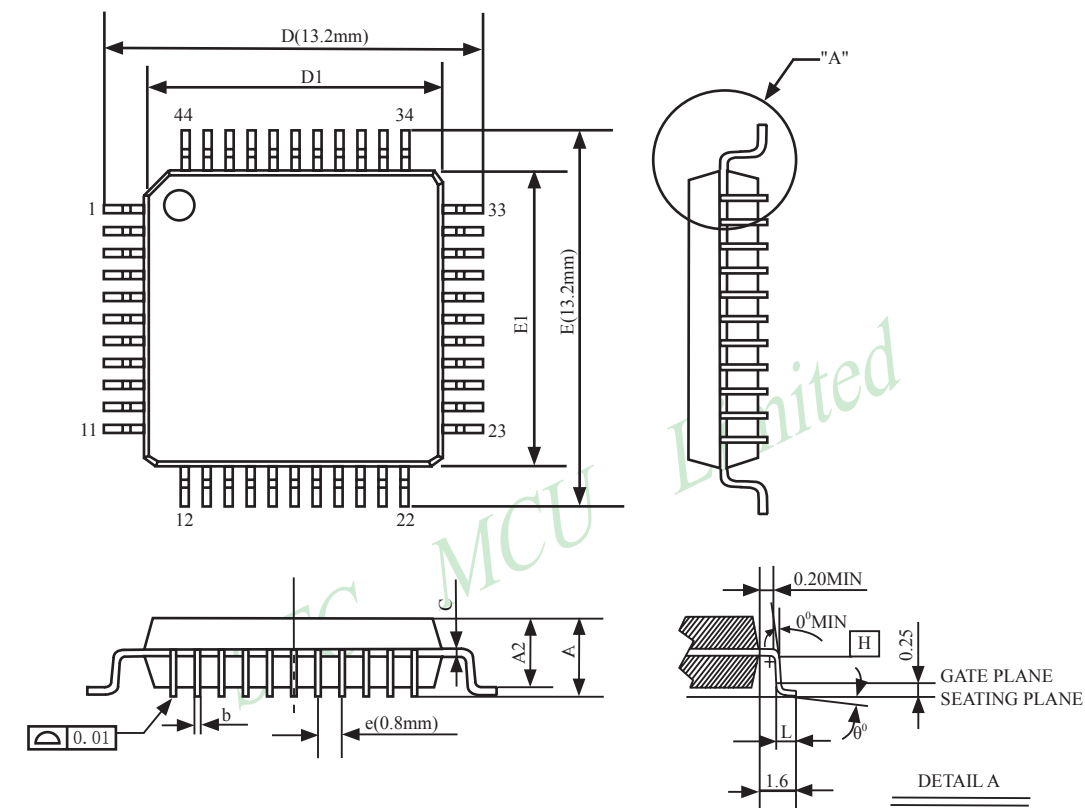
PLCC-44 封装尺寸图 (尽量不要选择此封装)**PLCC-44 OUTLINE PACKAGE**

SYMBOLS	DIMENSIONS IN INCH			DIMENSIONS IN MILLIMETERS		
	MIN	NOM	MAX	MIN	NOM	MAX
A	0.165	-	0.180	4.191	-	4.572
A1	0.020	-	-	0.508	-	-
A2	0.147	-	0.158	3.734	-	4.013
b1	0.026	0.028	0.032	0.660	0.711	0.813
b	0.013	0.017	0.021	0.330	0.432	0.533
c	0.007	0.010	0.0013	0.178	0.254	0.330
D	0.650	0.653	0.656	16.510	16.586	16.662
E	0.650	0.653	0.656	16.510	16.586	16.662
e	0.050BSC			1.270BSC		
G_d	0.590	0.610	0.630	14.986	15.494	16.002
G_e	0.590	0.610	0.630	14.986	15.494	16.002
H_d	0.685	0.690	0.695	17.399	17.526	17.653
H_e	0.685	0.690	0.695	17.399	17.526	17.653
L	0.100	-	0.112	2.540	-	2.845
Y	-	-	0.004	-	-	0.102

1 inch = 1000 mil

PQFP-44 封装尺寸图

PQFP-44 OUTLINE PACKAGE



SYMBOLS	MIN.	NOM	MAX.
A	—	—	2.70
A1	0.25	—	0.50
A2	1.80	2.00	2.20
b(w/o plating)	0.25	0.30	0.35
D	13.00	13.20	13.40
D1	9.9	10.00	10.10
E	13.00	13.20	13.40
E1	9.9	10.00	10.10
L	0.73	0.88	0.93
e	0.80 BSC.		
θ°	0	—	7
C	0.1	0.15	0.2

UNIT:mm

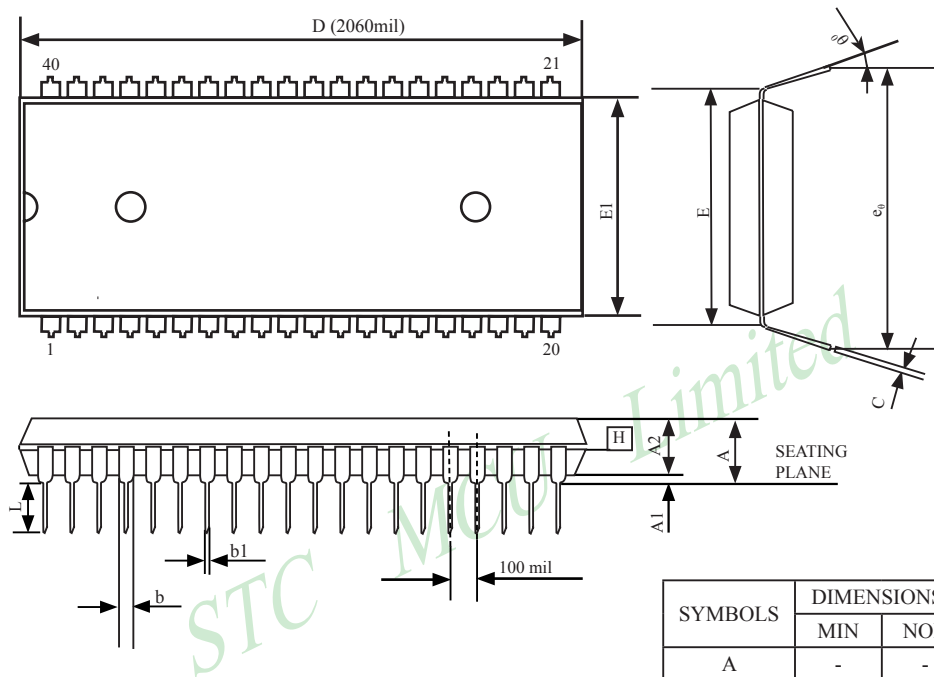
NOTES:

1. JEDEC OUTLINE: M0-108 AA-1

2. DATUM PLANE [H] IS LOCATED AT THE BOTTOM OF THE MOLD PARTING LINE COINCIDENT WITH WHERE THE LAED EXITS THE BODY.

3. DIMENSIONS D1 AND E1 DO NOT INCLUDE MOLD PROTRUSION. ALLOWABLE PROTRUSION IS 0.25mm PER SIDE. DIMENSIONS D1 AND E1 DO INCLUDE MOLD MISMATCH AND ARE DETERMINED AT DATUM PLANE [H].

4. DIMENSION b DOES NOT INCLUDE DAMBAR PROTRUSION.

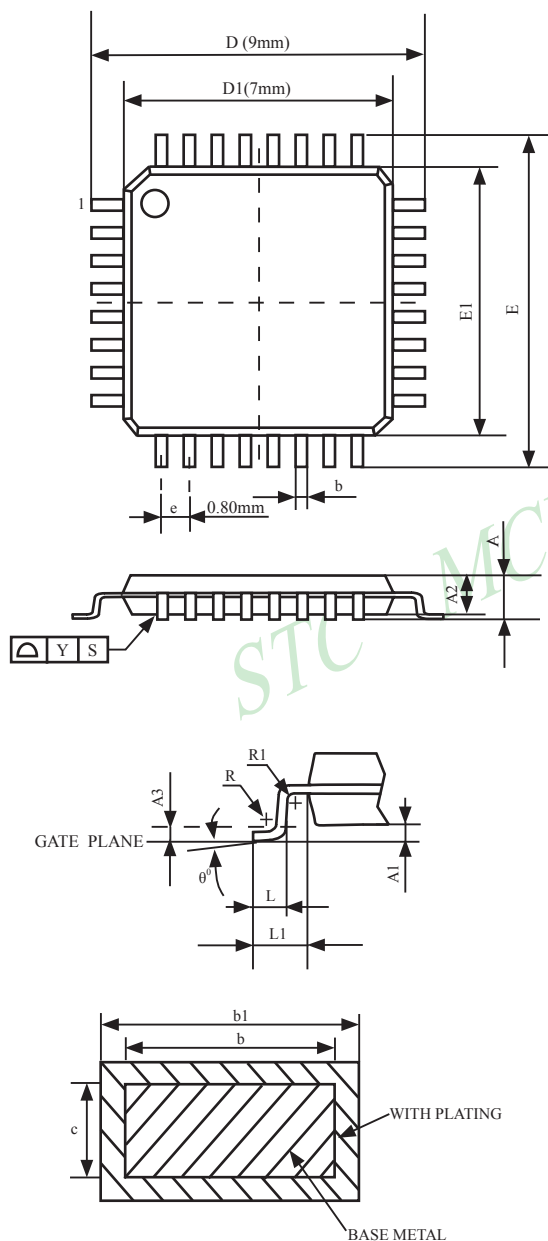
PDIP-40 封装尺寸图**PDIP-40 OUTLINE PACKAGE**

SYMBOLS	DIMENSIONS IN INCH		
	MIN	NOR	MAX
A	-	-	0.190
A1	0.015	-	0.020
A2	0.15	0.155	0.160
C	0.008	-	0.015
D	2.025	2.060	2.070
E	0.600 BSC		
E1	0.540	0.545	0.550
L	0.120	0.130	0.140
b1	0.015	-	0.021
b	0.045	-	0.067
e_0	0.630	0.650	0.690
0	0	7	15

UNIT: INCH 1 inch = 1000mil

LQFP-32 封装尺寸图

LQFP-32 OUTLINE PACKAGE



VARIATIONS (ALL DIMENSIONS SHOWN IN MM)

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

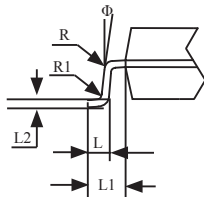
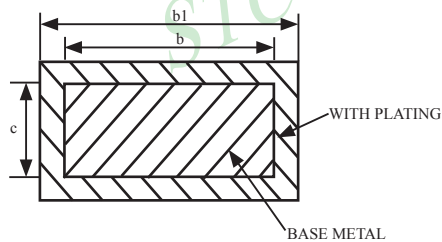
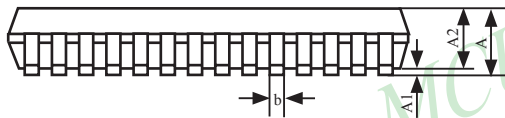
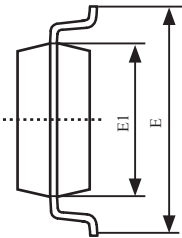
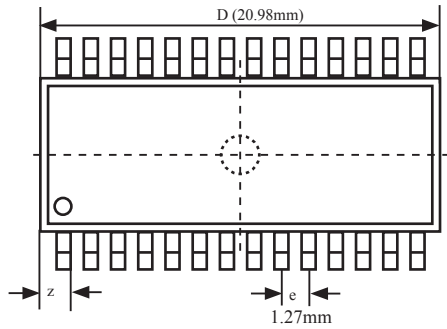
NOTES:

1. All dimensions are in mm
2. Dim D1 AND E1 does not include plastic flash.
Flash: Plastic residual around body edge after de junk/singulation
3. Dim b does not include dambar protrusion/ intrusion.
4. Plating thickness 0.05~0.015 mm.

SOP-32 封装尺寸图

32-Pin Small Outline Package (SOP-32)

Dimensions in Millimeters

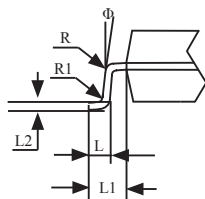
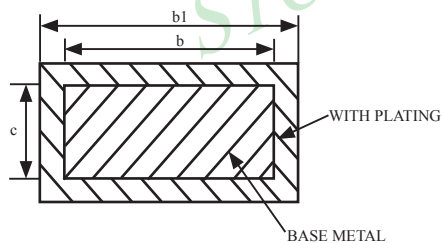
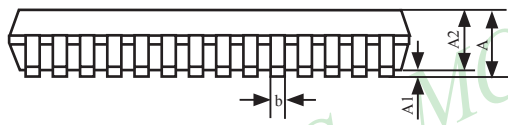
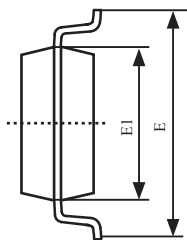
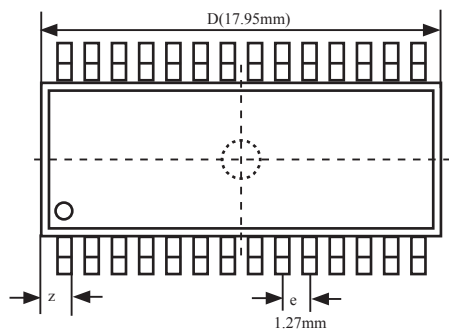


COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b	0.356	0.406	0.456
b1	0.366	0.426	0.486
c	-	0.254	-
D	20.88	20.98	21.08
E	9.980	10.180	10.380
E1	7.390	7.500	7.600
e	1.27		
L	0.700	0.800	0.900
L1	1.303	1.403	1.503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

SOP-28 封装尺寸图

28-Pin Small Outline Package (SOP-28)

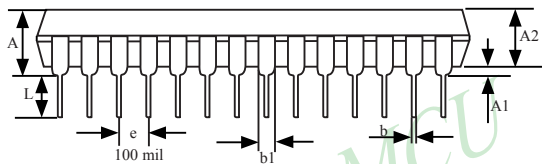
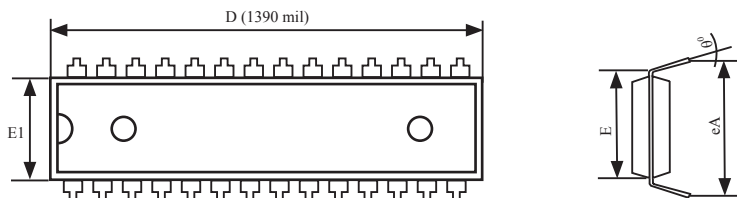
Dimensions in Millimeters



一般尺寸			
(测量单位 = MILLIMETER / 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°	-	10°
z	-	0.745	-

SKDIP-28 封装尺寸图**28-Pin Plastic Dual-In-line Package (SKDIP-28)**

Dimensions in Inches

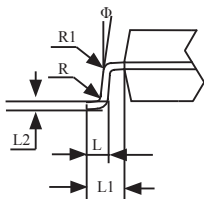
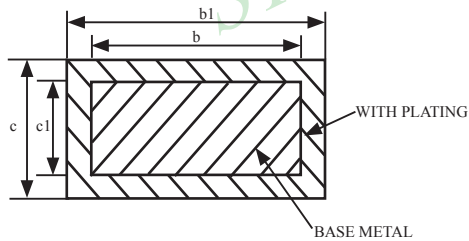
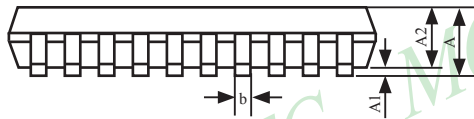
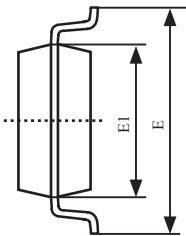
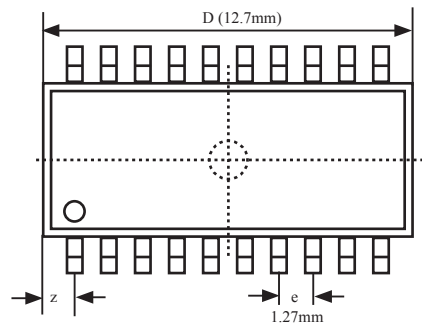


一般尺寸 (测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	-	-	0.210
A1	0.015	-	-
A2	0.125	0.13	0.135
b	-	0.018	-
b1	-	0.060	-
D	1.385	1.390	1.40
E	-	0.310	-
E1	0.283	0.288	0.293
e	-	0.100	-
L	0.115	0.130	0.150
θ°	0	7	15
eA	0.330	0.350	0.370

UNIT: INCH, 1 inch = 1000 mil

SOP-20 封装尺寸图

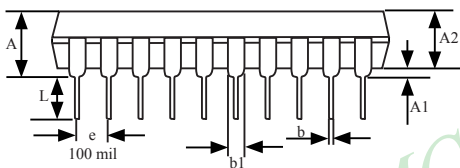
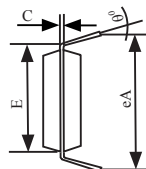
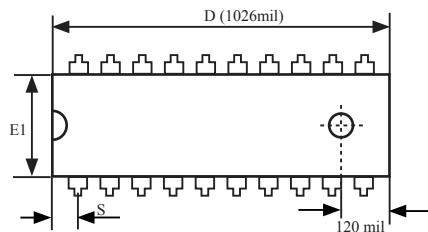
20-Pin Small Outline Package (SOP-20)
Dimensions in Inches and (Millimeters)



一般尺寸			
(测量单位 = MILLIMETER/ 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°	-	10°
z	-	0.660	-

DIP-20 封装尺寸图**20-Pin Plastic Dual Inline Package (DIP-20)**

Dimensions in Inches



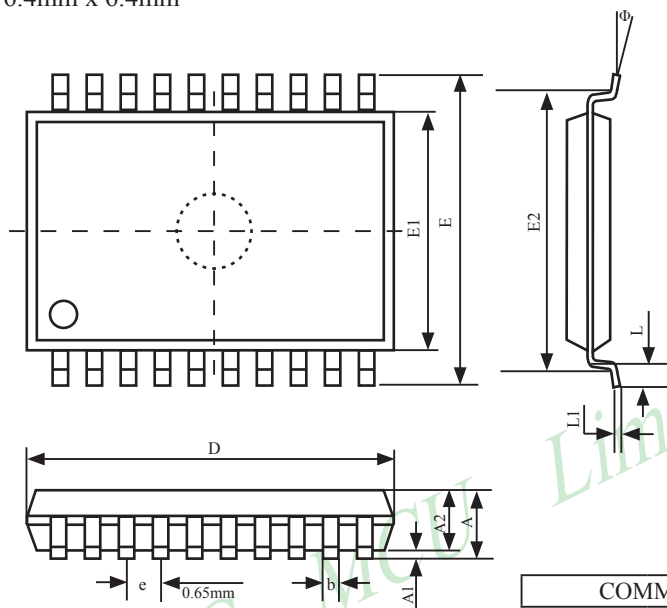
一般尺寸 (测量单位 = INCH)			
符号	MIN.	NOM.	MAX.
A	-	-	0.175
A1	0.015	-	-
A2	0.125	0.13	0.135
b	0.016	0.018	0.020
b1	0.058	0.060	0.064
C	0.008	0.010	0.11
D	1.012	1.026	1.040
E	0.290	0.300	0.310
E1	0.245	0.250	0.255
e	0.090	0.100	0.110
L	0.120	0.130	0.140
θ°	0	-	15
eA	0.355	0.355	0.375
S	-	-	0.075

UNIT: INCH, 1 inch = 1000 mil

LSSOP-20 封装尺寸图

20-Pin Plastic Shrink Small Outline Package (LSSOP-20)

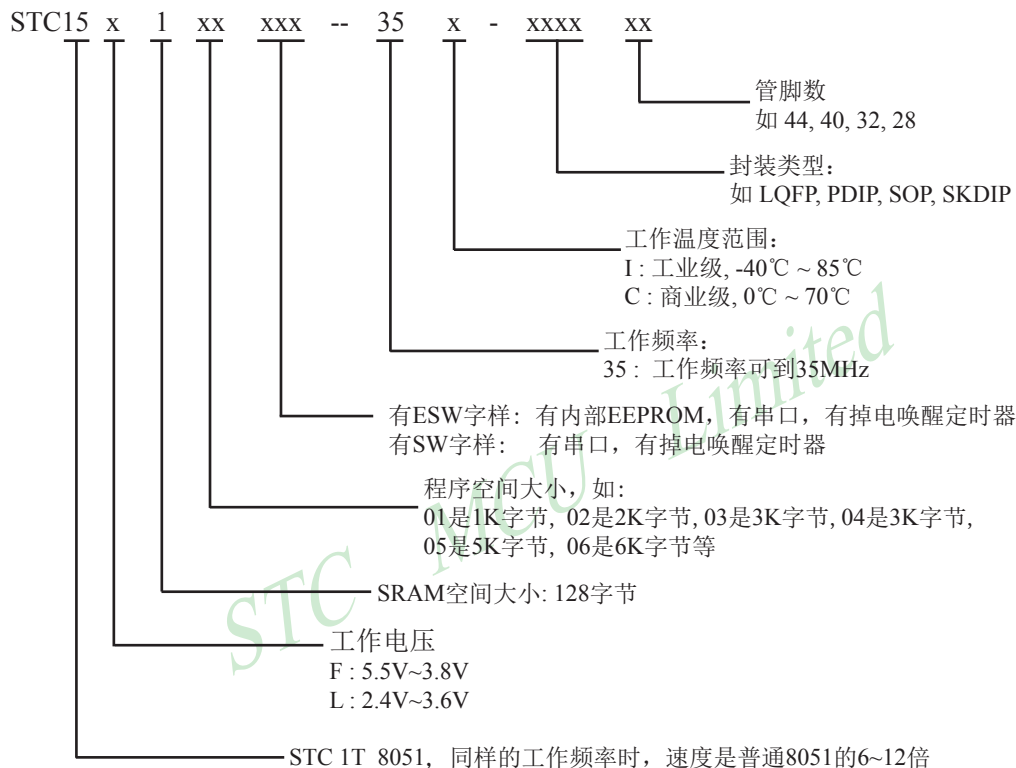
LSSOP-20, 6.4mm x 6.4mm



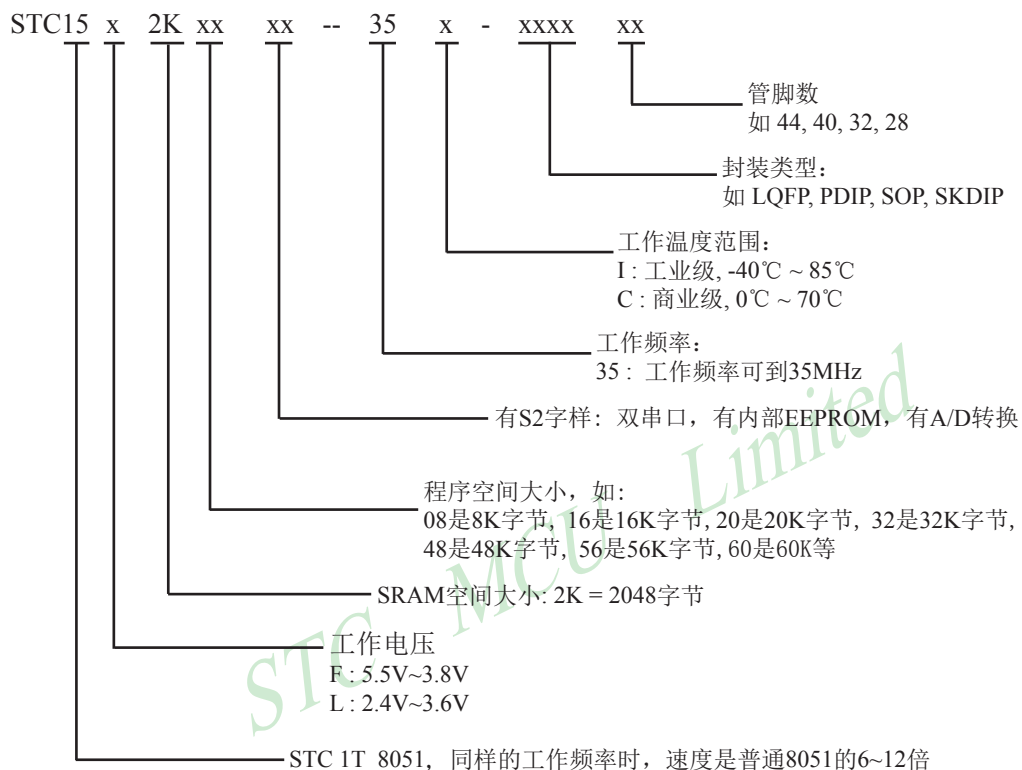
COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	-	-	1.85
A1	0.05	-	-
A2	1.40	1.50	1.60
b	0.17	0.22	0.32
D	6.40	6.50	6.60
E	6.20	6.40	6.60
E1	4.30	4.40	4.50
E2	-	5.72	-
e	0.57	0.65	0.73
L	0.30	0.50	0.70
L1	0.1	0.15	0.25
Φ	0°	-	8°

1.8 STC15系列单片机命名规则

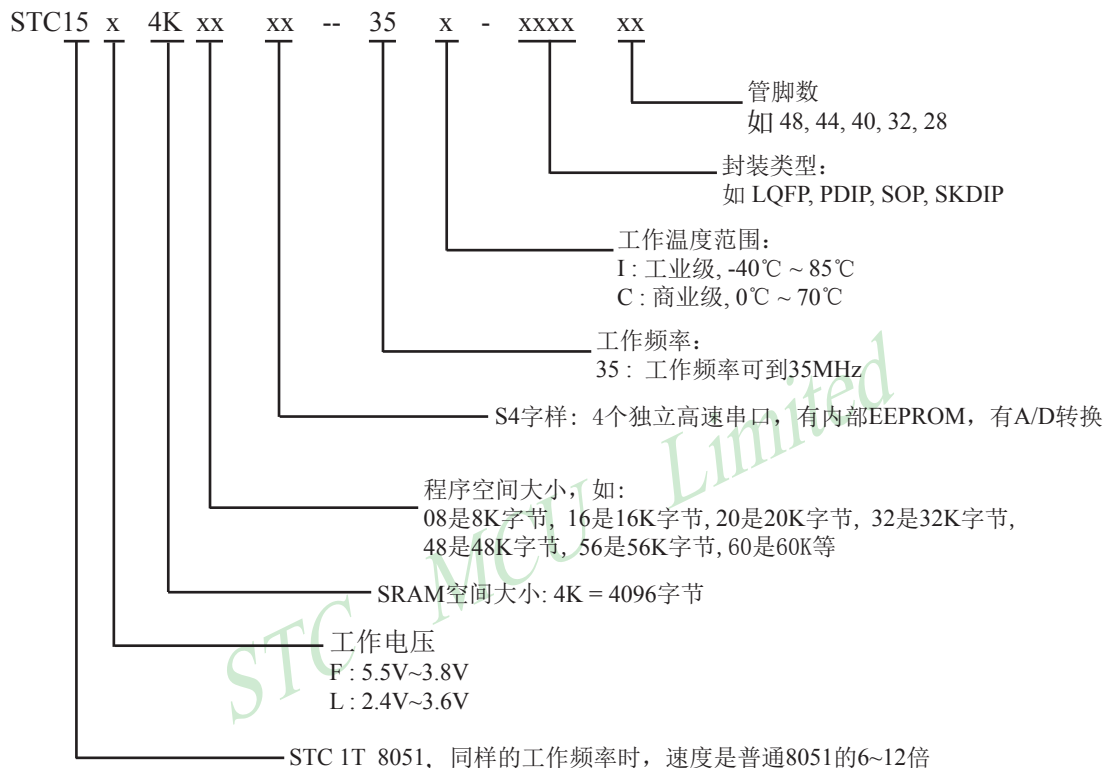
1.8.1 STC15F104ESW系列单片机命名规则



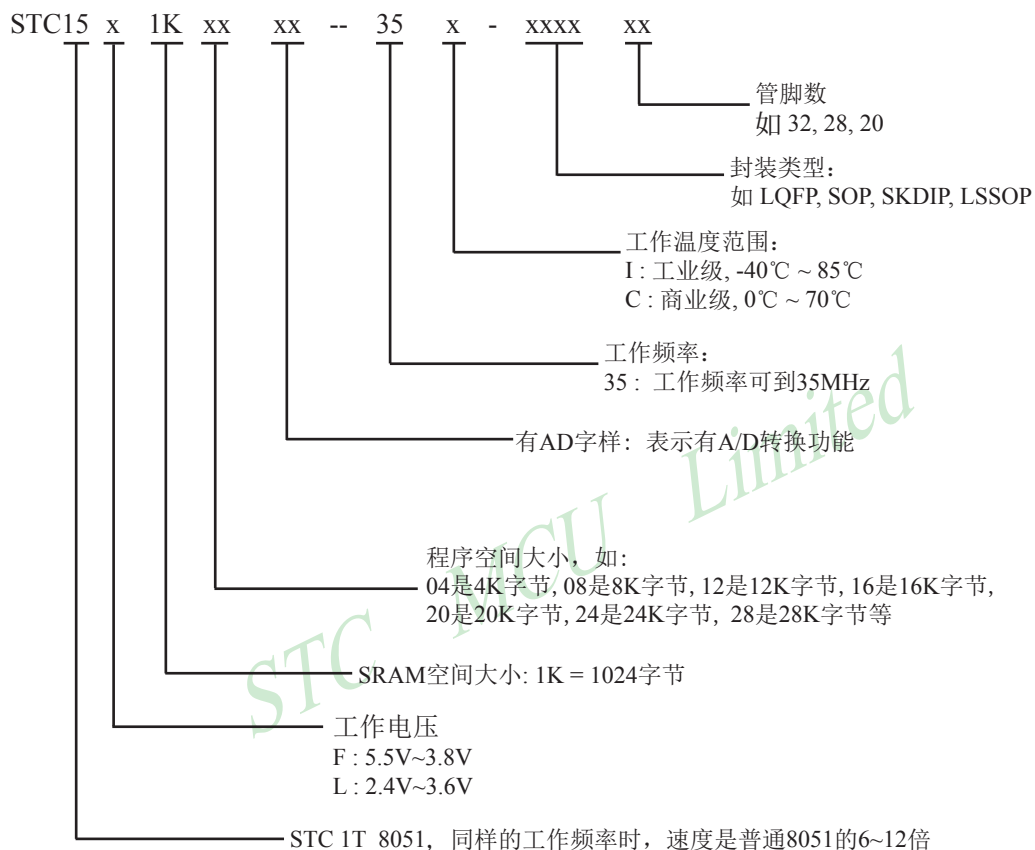
1.8.2 STC15F2K60S2系列单片机命名规则



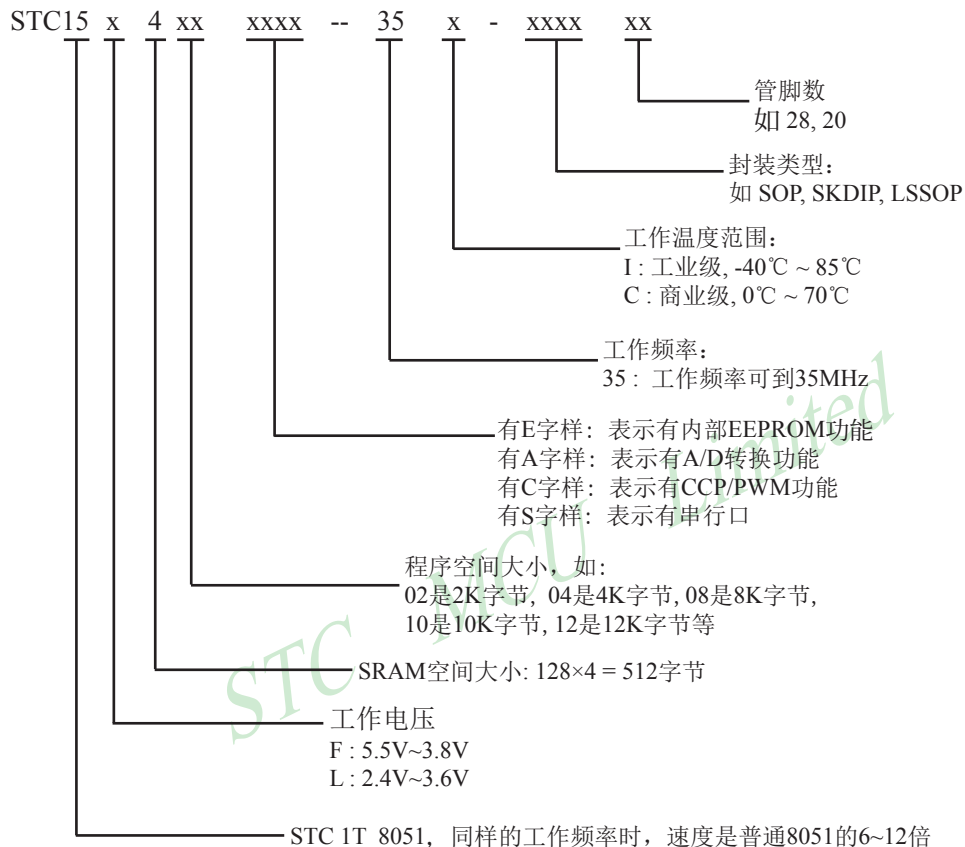
1.8.3 STC15F4K60S4系列单片机命名规则



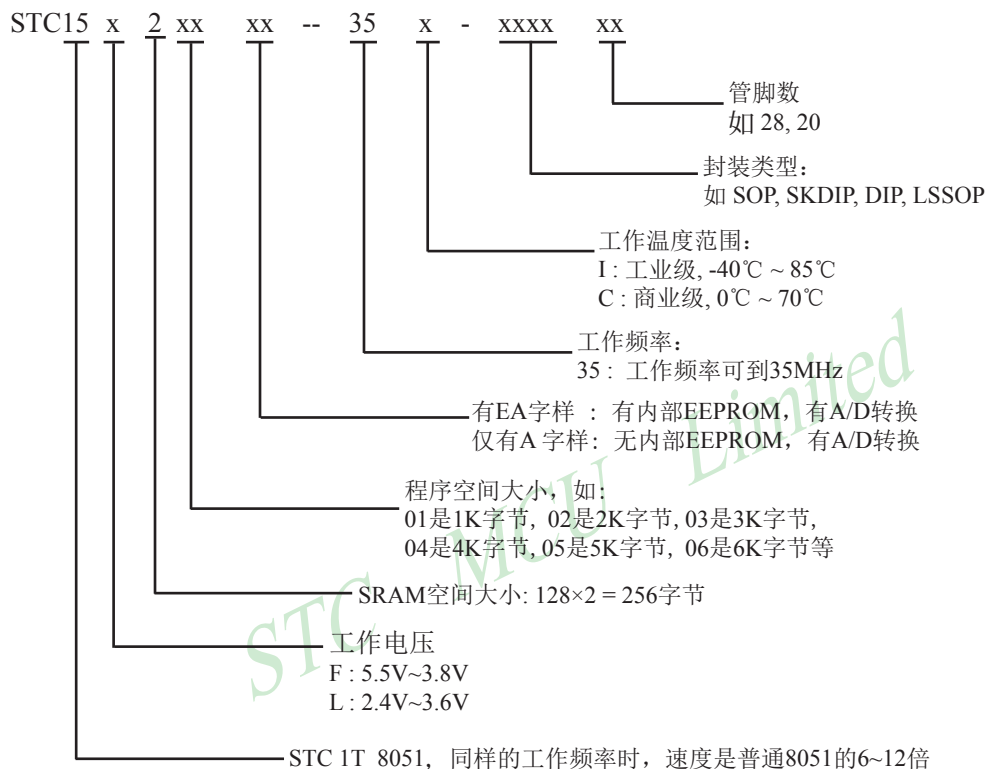
1.8.4 STC15F1K20AD系列单片机命名规则



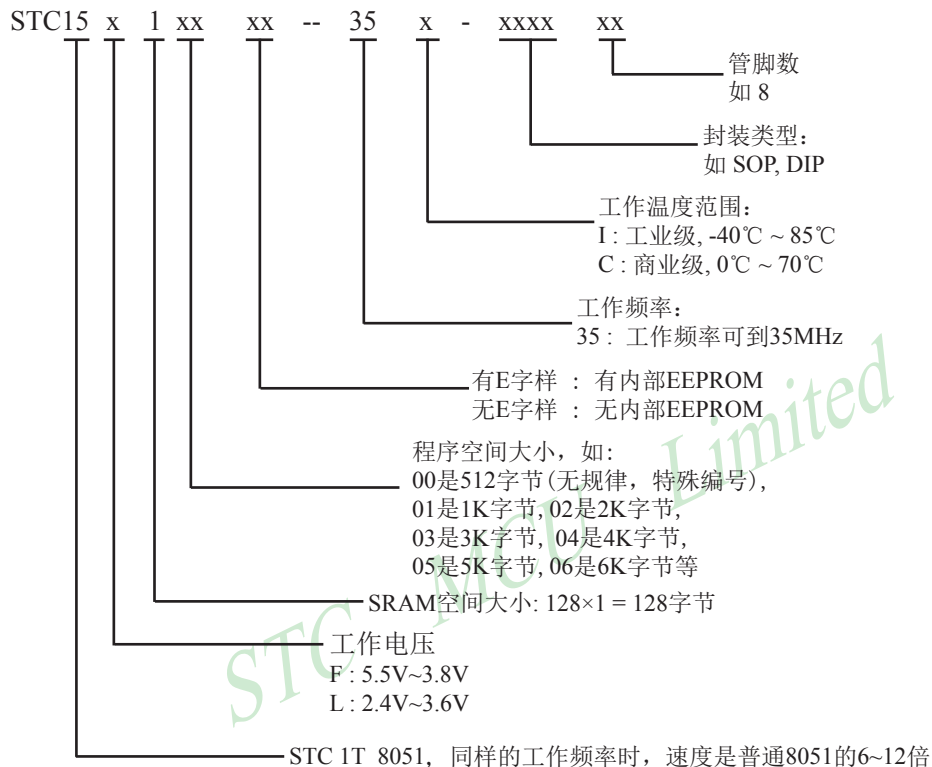
1.8.5 STC15F412EACS系列单片机命名规则



1.8.6 STC15F204EA系列单片机命名规则



1.8.7 STC15F104E系列单片机命名规则



1.9 串行口在多个管脚之间切换及其测试程序(C和汇编)

Mnemonic	Add	Name	7	6	5	4	3	2	1	0	Reset Value
P_SW2	BAH	Peripheral function switch	S1_S1	-	-	-	-	-	-	-	0xxx,xxxx

串口/S1可在3个地方切换, 由 S1_S0 及 S1_S1 控制位来选择

S1_S1 串口/S1可在P1/P3之间来回切换

0 串口/S1在[P3. 0/RxD, P3. 1/TxD]

1 串口/S1在[P3. 6/RxD_3, P3. 7/TxD_3]

请不要再抄袭我们的设计、规格和管脚排列, 再抄袭就很无耻了。

	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						T2H RL_TH2 0000,0000	T2L RL_TL2 0000,0000	0D7H
0C8H	P5 xx11,xxxx	P5M1 xx00,xxxx	P5M0 xx00,xxxx						0CFH
0C0H		WDT_CONTR 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 x0x0,x0x0	SADEN	P_SW2 0xxx,xxxx	IRC_CLKO xxxx,xx00					0BFH
0B0H	P3 11xx,1111	P3M1 00xx,0000	P3M0 00xx,0000						0B7H
0A8H	IE 00x0,xx00	SADDR	WKTCL WKTCL_CNT 0111 1111	WKTCH WKTCH_CNT 0111 1111				IE2 xxxx,x0xx	0AFH
0A0H			AUXR1 P_SW1 xxxx,0000	Don't use	Don't use	Don't use		Don't use	0A7H
098H	SCON 0000,0000	SBUF xxxx,xxxx					Don't use	Don't use	09FH
090H	P1 xx11,1111	P1M1 xx00,0000	P1M0 xx00,0000					CLK_DIV PCON2	097H
088H	TCON xxxx,0000						AUXR xx00,00xx	INT_CLKO AUXR2 x000 00xx	08FH
080H		SP 0000,0111	DPL 0000,0000	DPH 0000,0000				PCON 0011,0000	087H
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

1.C程序：

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 串行口在多个口之间切换举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#define FOSC 18432000L
```

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

```
sfr      P_SW2  = 0xBA;           //外设功能切换寄存器2
```

```
#define S1_S1  0x80              //P_SW2.7
```

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

```
void main()
```

```
{
    P_SW2  &=  ~S1_S1;           //(P3.0/RxD, P3.1/TxD)
```

```
//    P_SW2  |=  S1_S1;           //(P3.6/RxD_3, P3.7/TxD_3)
```

```
    while (1);                  //程序终止
```

```
}
```

2.汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 串行口在多个口之间切换举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#define FOSC 18432000L
```

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

```
P_SW2 EQU 0BAH //外设功能切换寄存器2
```

```
S1_S1 EQU 80H //P_SW2.7
```

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

```
ORG 0000H
LJMP MAIN //复位入口
```

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

```
ORG 0100H
```

```
MAIN:
```

```
MOV SP, #3FH
```

```
ANL P_SW2, #NOT S1_S1 //(P3.0/RxD, P3.1/TxD)
```

```
// ORL P_SW2, #S1_S1 //(P3.6/RxD_3, P3.7/TxD_3)
```

```
SJMP $ //程序终止
```

```
END
```

1.10 每个单片机具有全球唯一身份证号码(ID号)及其测试程序

STC最新一代STC15系列每一个单片机出厂时都具有全球唯一身份证号码(ID号)，用户可以在单片机上电后读取内部RAM单元从F1H - F7H (对于STC15F104E系列是从77H - 7FH)连续7个单元的值来获取此单片机的唯一身份证号码(ID号)，使用“MOV @Ri”指令来读取。如果用户需要用全球唯一ID号进行用户自己的软件加密，建议用户在程序的多个地方有技巧地判断自己的用户程序有无被非法修改，提高解密的难度，防止解密者修改程序，绕过对全球唯一ID号的判断。

除内部RAM的F1H ~ F7H单元的内容为全球唯一ID号外，最新的STC15系列的程序存储器的最后7个字节单元的值也是全球唯一ID号，用户不可修改，但IAP15系列整个程序区是开放的，可以修改，建议利用全球唯一ID号加密时，使用STC15系列，并将EEPROM功能使用上，从EEPROM起始地址0000H开始使用，有效杜绝对全球唯一ID号的攻击。使用程序区的最后7个字节的全球唯一ID号比使用RAM单元 F1H - F7H 的全球唯一ID号进行比较更难被攻击。

//从RAM区和程序区获取全球唯一身份证号码(ID号)的程序举例

1. C程序

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 获取全球唯一身份证号码(ID号)举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#include "reg51.h"
```

```
typedef unsigned char    BYTE;
typedef unsigned int     WORD;
```

```
#define  URMD  0           //0:使用定时器2作为波特率发生器
                        //1:使用定时器1的模式0(16位自动重载模式)作为波特率发生器
                        //2:使用定时器1的模式2(8位自动重载模式)作为波特率发生器
```

```

sfr      T2H      =      0xd6;                //定时器2高8位
sfr      T2L      =      0xd7;                //定时器2低8位

sfr      AUXR     =      0x8e;                //辅助寄存器

#define ID_ADDR_RAM      0xf1                //ID号的存放在RAM区的地址为0F1H

//ID号的存放在程序区的地址为程序空间的最后7字节
//#define ID_ADDR_ROM  0x03f9                //1K程序空间的MCU(如STC15F201EA, STC15F101EA)
//#define ID_ADDR_ROM  0x07f9                //2K程序空间的MCU(如STC15F402EACS,
//STC15F202EA, STC15F102EA)
//#define ID_ADDR_ROM  0x0bf9                //3K程序空间的MCU(如STC15F203EA, STC15F103EA)
//#define ID_ADDR_ROM  0x0ff9                //4K程序空间的MCU(如STC15F804EACS,
//STC15F404EACS, STC15F204EA, STC15F104EA)
//#define ID_ADDR_ROM  0x13f9                //5K程序空间的MCU(如 STC15F205EA, STC15F105EA)
//#define ID_ADDR_ROM  0x1ff9                //8K程序空间的MCU(如STC15F2K08S2, STC15F808EACS,
//STC15F408EACS)
//#define ID_ADDR_ROM  0x27f9                //10K程序空间的MCU(如STC15F410EACS)
//#define ID_ADDR_ROM  0x2ff9                //12K程序空间的MCU(如STC15F812EACS,
//STC15F412EACS)
//#define ID_ADDR_ROM  0x3ff9                //16K程序空间的MCU(如STC15F2K16S2,
//STC15F816EACS)
//#define ID_ADDR_ROM  0x4ff9                //20K程序空间的MCU(如STC15F2K20S2,
//STC15F820EACS)
//#define ID_ADDR_ROM  0x5ff9                //24K程序空间的MCU(如STC15F824EACS)
//#define ID_ADDR_ROM  0x6ff9                //28K程序空间的MCU(如STC15F1K20AD)
//#define ID_ADDR_ROM  0x7ff9                //32K程序空间的MCU(如STC15F2K32S2)
//#define ID_ADDR_ROM  0x9ff9                //40K程序空间的MCU(如STC15F2K40S2)
//#define ID_ADDR_ROM  0xbff9                //48K程序空间的MCU(如STC15F2K48S2)
//#define ID_ADDR_ROM  0xcff9                //52K程序空间的MCU(如STC15F2K52S2)
//#define ID_ADDR_ROM  0xdff9                //56K程序空间的MCU(如STC15F2K56S2)
#define ID_ADDR_ROM  0xeff9                //60K程序空间的MCU(如STC15F104ESW)

//-----

void InitUart();
void SendUart(BYTE dat);

//-----

void main()
{
    BYTE  idata  *iptr;
    BYTE  code   *cptr;
    BYTE  i;

    InitUart();                //串口初始化

```

```

        iptr = ID_ADDR_RAM;                //从RAM区读取ID号
        for (i=0; i<7; i++)                //读7个字节
        {
            SendUart(*iptr++);             //发送ID到串口
        }
        cptr = ID_ADDR_ROM;                //从程序区读取ID号
        for (i=0; i<7; i++)                //读7个字节
        {
            SendUart(*cptr++);             //发送ID到串口
        }

        while (1);                        //程序终止
    }
    /*-----
    串口初始化
    -----*/
    void InitUart()
    {
        SCON = 0x5a;                      //设置串口为8位可变波特率
    #if URMD == 0
        T2L = 0xd8;                        //设置波特率重装值
        T2H = 0xff;                        //115200 bps(65536-18432000/4/115200)
        AUXR = 0x14;                       //T2为1T模式, 并启动定时器2
        AUXR |= 0x01;                      //选择定时器2为串口的波特率发生器
    #elif URMD == 1
        AUXR = 0x40;                       //定时器1为1T模式
        TMOD = 0x00;                       //定时器1为模式0(16位自动重载)
        TL1 = 0xd8;                        //设置波特率重装值
        TH1 = 0xff;                        //115200 bps(65536-18432000/4/115200)
        TR1 = 1;                           //定时器1开始启动
    #else
        TMOD = 0x20;                       //设置定时器1为8位自动重载模式
        AUXR = 0x40;                       //定时器1为1T模式
        TH1 = TL1 = 0xfb;                  //115200 bps(256 - 18432000/32/115200)
        TR1 = 1;
    #endif
    }
    /*-----
    发送串口数据
    -----*/
    void SendUart(BYTE dat)
    {
        while (!TI);                      //等待前面的数据发送完成
        TI = 0;                           //清除发送完成标志
        SBUF = dat;                       //发送串口数据
    }

```


2. 汇编程序

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 获取全球唯一身份证号码(ID号)举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#define URMD 0 //0:使用定时器2作为波特率发生器
//1:使用定时器1的模式0(16位自动重载模式)作为波特率发生器
//2:使用定时器1的模式2(8位自动重载模式)作为波特率发生器

T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位
AUXR DATA 08EH //辅助寄存器
//-----

#define ID_ADDR_RAM 0xf1 //ID号的存放在RAM区的地址为0F1H

//ID号的存放在程序区的地址为程序空间的最后7字节
//1K程序空间的MCU(如STC15F201EA, STC15F101EA)
#define ID_ADDR_ROM 0x03f9 //2K程序空间的MCU(如 STC15F402EACS, STC15F202EA,
// STC15F102EA)
#define ID_ADDR_ROM 0x0bf9 //3K程序空间的MCU如STC15F203EA, STC15F103EA)
#define ID_ADDR_ROM 0x0ff9 //4K程序空间的MCU(如STC15F804EACS,
//STC15F404EACS, STC15F204EA, STC15F104EA)
#define ID_ADDR_ROM 0x13f9 //5K程序空间的MCU(如STC15F205EA, STC15F105EA)
#define ID_ADDR_ROM 0x1ff9 //8K程序空间的MCU(如STC15F2K08S2, STC15F808EACS,
//STC15F408EACS)
#define ID_ADDR_ROM 0x27f9 //10K程序空间的MCU(如 STC15F410EACS)
#define ID_ADDR_ROM 0x2ff9 //12K程序空间的MCU(如STC15F812EACS,
// STC15F412EACS)
#define ID_ADDR_ROM 0x3ff9 //16K程序空间的MCU(如STC15F2K16S2,STC15F816EACS)
#define ID_ADDR_ROM 0x4ff9 //20K程序空间的MCU(如STC15F2K20S2,STC15F820EACS)
#define ID_ADDR_ROM 0x5ff9 //24K程序空间的MCU(如STC15F824EACS)
#define ID_ADDR_ROM 0x6ff9 //28K程序空间的MCU(如 STC15F1K20AD)
#define ID_ADDR_ROM 0x7ff9 //32K程序空间的MCU(如STC15F2K32S2)
#define ID_ADDR_ROM 0x9ff9 //40K程序空间的MCU(如STC15F2K40S2)
#define ID_ADDR_ROM 0xbff9 //48K程序空间的MCU(如STC15F2K48S2)
#define ID_ADDR_ROM 0xcff9 //52K程序空间的MCU(如STC15F2K52S2)

```

```

// #define ID_ADDR_ROM 0xdff9 //56K程序空间的MCU(如STC15F2K56S2)
// #define ID_ADDR_ROM 0xeff9 //60K程序空间的MCU(如STC15F104ESW)

//-----

//-----

        ORG     0000H
        LJMP    MAIN                //复位入口

//-----

        ORG     0100H

MAIN:
        MOV     SP,    #3FH

        LCALL   INIT_UART          //串口初始化

        MOV     R0,    #ID_ADDR_RAM //从RAM区读取ID号
        MOV     R1,    #7          //读7个字节
NEXT1:
        MOV     A,     @R0
        LCALL   SEND_UART          //发送ID到串口
        INC     R0
        DJNZ    R1,    NEXT1

        MOV     DPTR,   #ID_ADDR_ROM //从程序区读取ID号
        MOV     R1,    #7          //读7个字节
NEXT2:
        CLR     A
        MOVC    A,     @A+DPTR
        LCALL   SEND_UART          //发送ID到串口
        INC     DPTR
        DJNZ    R1,    NEXT2

        SJMP    $                //程序终止

/*-----
串口初始化
-----*/
INIT_UART:
        MOV     SCON,   #5AH        //设置串口为8位可变波特率
#if
        URMD == 0
        MOV     T2L,    #0D8H      //设置波特率重装值(65536-18432000/4/115200)
        MOV     T2H,    #0FFH
        MOV     AUXR,   #14H        //T2为1T模式, 并启动定时器2
        ORL     AUXR,   #01H        //选择定时器2为串口的波特率发生器

```

```
#elif    URMD == 1
        MOV    AUXR, #40H           //定时器1为1T模式
        MOV    TMOD, #00H           //定时器1为模式0(16位自动重载)
        MOV    TL1,  #0D8H          //设置波特率重装值(65536-18432000/4/115200)
        MOV    TH1,  #0FFH
        SETB   TR1                   //定时器1开始运行
#else
        MOV    TMOD, #20H           //设置定时器1为8位自动重载模式
        MOV    AUXR, #40H           //定时器1为1T模式
        MOV    TL1,  #0FBH          //115200 bps(256 - 18432000/32/115200)
        MOV    TH1,  #0FBH
        SETB   TR1
#endif

        RET

/*-----
发送串口数据
入口参数: ACC
出口参数: 无
-----*/
SEND_UART:
        JNB    TI,    $             //等待前面的数据发送完成
        CLR    TI              //清除发送完成标志
        MOV    SBUF,  A           //发送串口数据
        RET

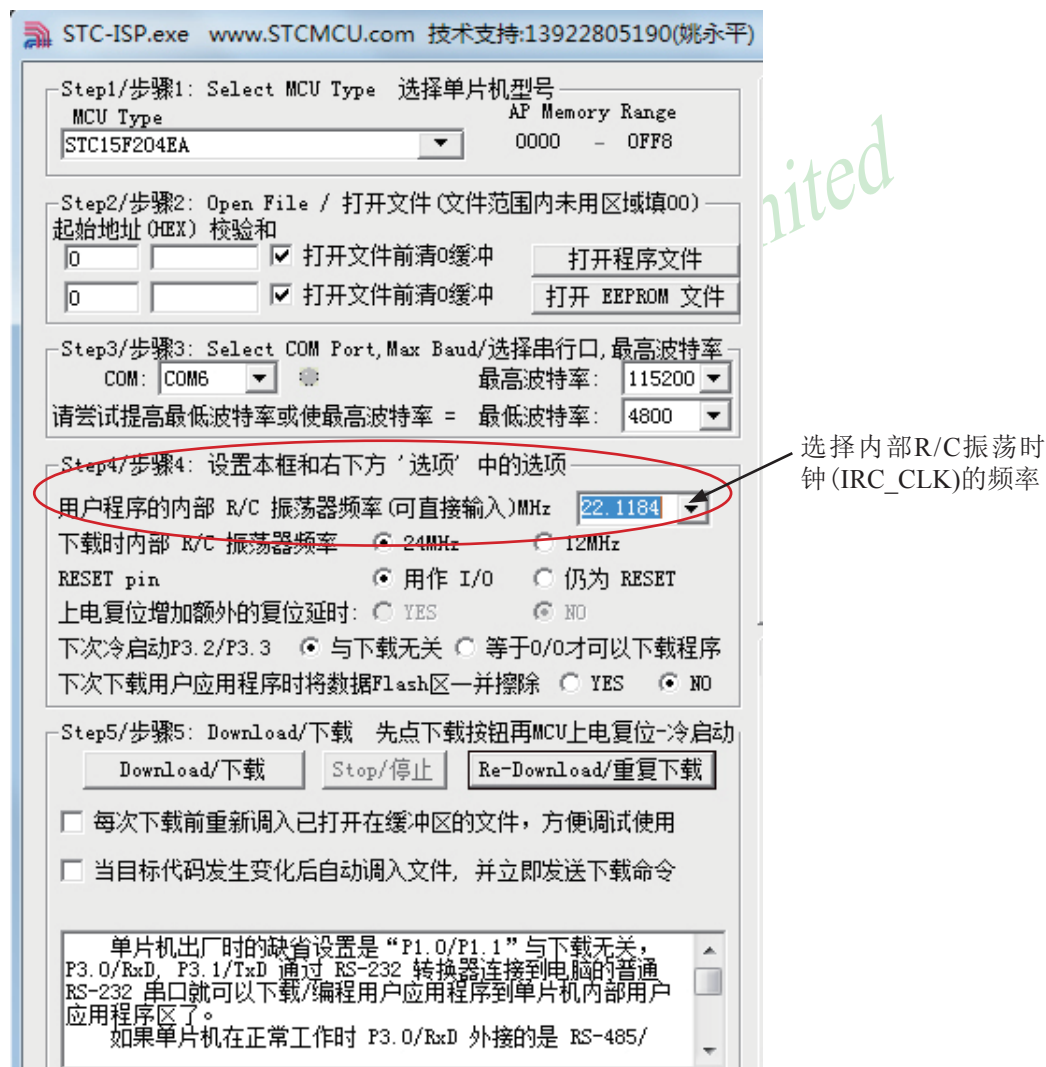
        END
```

第2章 STC15系列的时钟，省电模式及复位

2.1 STC15F104ESW系列单片机的内部可配置时钟

STC15F104ESW系列单片机只有一个时钟源 — 内部高精度R/C时钟， $\pm 1\%$ 温飘（ $-40^{\circ}\text{C}\sim+85^{\circ}\text{C}$ ），常温下温飘5%

2.1.1 在STC-ISP下载/编程工具中设置内部高精度R/C时钟的振荡频率

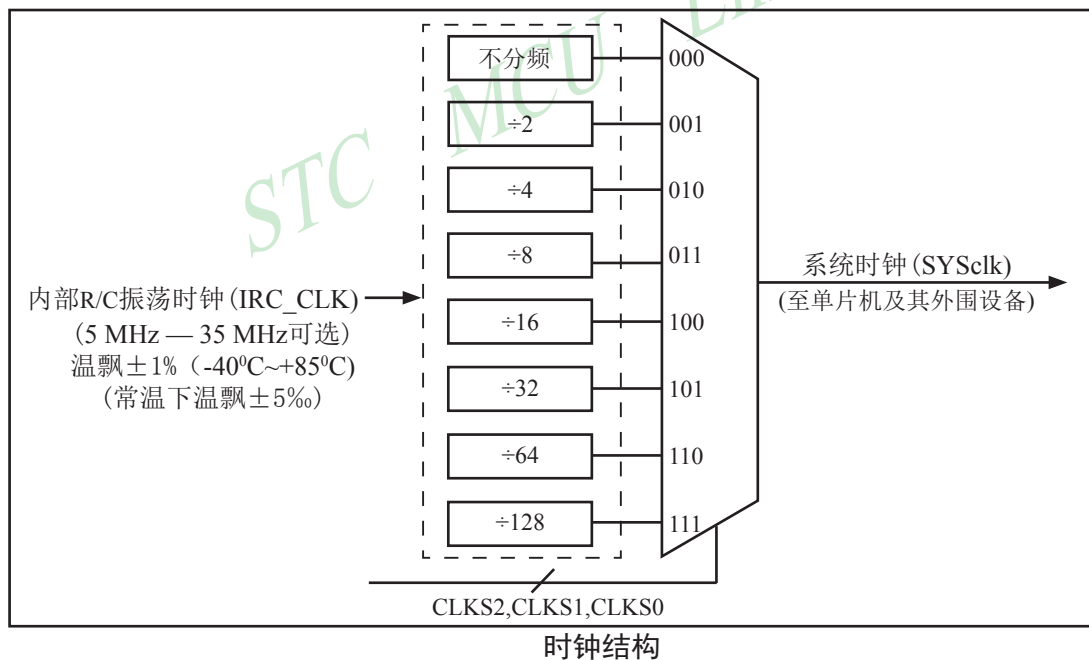


2.1.2 内部时钟分频和分频寄存器

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(PCON2)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 (PCON2)各位的定义如下:

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CLK_DIV (PCON2)	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							



2.1.3 可编程时钟输出(也可作分频器使用)

有2路特可编程时钟输出：IRC_CLKO/P5.4, T2CLKO/P3.0. 只有内部R/C时钟频率为12MHz以下时，现版本的IRC_CLKO/P5.4才能正常输出。

2.1.3.1 与可编程时钟输出有关的特殊功能寄存器

符号	描述	地址	位地址及其符号								复位值
			MSB				LSB				
AUXR	辅助寄存器	8EH	-	-	UART_M0x6	T2R	T2_C \overline{T}	T2x12	-	-	xx00 00xxB
INT_CLKO AUXR2	External Interrupt enable and Clock output register	8FH									x000 00xxB
			-	EX4	EX3	EX2	LVD_WAKE	T2CLKO	-	-	
IRC_CLKO	内部R/C时钟输出寄存器	BBH	-	-	-	-	-	-	IRCS1	IRCS0	xxxx xx00B

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

1. IRC_CLKO : Internal R/C clock output register (Non bit-addressable)

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IRC_CLKO	BBH	name	-	-	-	-	-	-	IRCS1	IRCS0

B7 ~ B2：保留位。

B1-IRCS1	B0-IRCS0	内部R/C振荡时钟的输出频率
0	0	无内部R/C振荡时钟的输出
0	1	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
1	0	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
1	1	内部R/C振荡时钟的输出频率被4分频，输出时钟频率 = IRC_CLK/4

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。

I/O口的输出速度只能达到15MHz附近。

2. INT_CLKO (AUXR2) : External Interrupt Enable and Clock Output register (Non bit-addressable)

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
INT_CLKO AUXR2	8FH	name	-	EX4	EX3	EX2	LVD_WAKE	T2CLKO	-	-

B2 - T2CLKO: 是否允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

1: 允许将P3.0脚配置为定时器2的时钟输出T2CLKO/CLKOUT2, 输出时钟频率= $T2\text{溢出率}/2$

如果 $T2_C/\overline{T}=0$, 定时器/计数器T2是对内部系统时钟计数, 则:

T2工作在1T模式($AUXR.2/T2x12=1$)时的输出频率 = $(SYSclk) / (65536-[RL_TH2, RL_TL2])/2$

T2工作在12T模式($AUXR.2/T2x12=0$)时的输出频率 = $(SYSclk) / 12 / (65536-[RL_TH2, RL_TL2])/2$

如果 $T2_C/\overline{T}=1$, 定时器/计数器T2是对外部脉冲输入(P3. 1/T2)计数, 则:

输出时钟频率 = $(T2_Pin_CLK) / (65536-[RL_TH2, RL_TL2])/2$

0: 不允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

B4 - EX2: 允许外部中断2($\overline{INT2}$)

B5 - EX3: 允许外部中断3($\overline{INT3}$)

B6 - EX4: 允许外部中断4($\overline{INT4}$)

3、辅助特殊功能寄存器: AUXR(地址: 0x8E)

AUXR: Auxiliary register (不可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR	8EH	name	-	-	UART_M0x6	T2R	T2_C/ \overline{T}	T2x12	-	-

B5 - UART_M0x6: 串口模式0的通信速度设置位。

0: UART串口模式0的速度是传统8051单片机串口的速度, 即12分频;

1: UART串口模式0的速度是传统8051单片机串口速度的6倍, 即2分频。

B4 - T2R: 定时器2运行控制位。

0: 不允许定时器2运行;

1: 允许定时器2运行。

B3 - T2_C/ \overline{T} : 控制定时器2用作定时器或计数器。

0, 用作定时器(对内部系统时钟进行计数);

1, 用作计数器(对引脚T2/P3. 1的外部脉冲进行计数)

B2 - T2x12: 定时器2速度控制位

0, 定时器2是传统8051单片机速度, 12分频;

1, 定时器2的速度是传统8051单片机的12倍, 不分频

当串口用T2作为波特率发生器时, 则由T2x12决定串口是12T还是1T.

2.1.3.2 内部R/C时钟输出及测试程序(C和汇编)

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	-	-	-	-	-	-	IRCS1	IRCS0

如何利用IRC_CLKO/P5.4管脚输出时钟

IRC_CLKO/P5.4的时钟输出控制由IRC_CLKO寄存器的IRCS1和IRCS0位控制。通过设置IRCS1(IRC_CLKO.1)和IRCS0(IRC_CLKO.0)可将IRC_CLKO/P5.4管脚配置为内部R/C振荡时钟输出同时还可以设置该内部R/C振荡时钟的输出频率。

新增加的特殊功能寄存器：IRC_CLKO（地址：0xBB）

B1-IRCS1	B0-IRCS0	内部R/C振荡时钟的输出频率
0	0	无内部R/C振荡时钟的输出
0	1	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
1	0	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
1	1	内部R/C振荡时钟的输出频率被4分频，输出时钟频率 = IRC_CLK/4

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。

I/O口的输出速度只能达到15MHz附近。

下面是内部R/C时钟输出的示例程序：

1. C程序：

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列单片机的内部R/C时钟输出 -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

```

```
#include "reg51.h"
```

```
typedef unsigned char    BYTE;
typedef unsigned int     WORD;
```

```
#define FOSC    18432000L
```



```
//-----

sfr      IRC_CLKO = 0xBB;      //IRC时钟输出控制寄存器

//-----

void main()
{
    IRC_CLKO = 0x01;           //0000,0001 P5.4输出频率为SYSclk
//    IRC_CLKO = 0x02;           //0000,0010 P5.4输出频率为SYSclk/2
//    IRC_CLKO = 0x03;           //0000,0011 P5.4输出频率为SYSclk/4

    while (1);                //程序终止
}
```

2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机的内部R/C时钟输出 -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序, -----*/
/* 请在程序或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

IRC_CLKO      DATA    0BBH                      //IRC时钟输出控制寄存器

;-----
;interrupt vector table

        ORG      0000H
        LJMP     MAIN                          //复位入口
;-----

        ORG      0100H
MAIN:
        MOV      SP,      #3FH                  //initial SP
        MOV      IRC_CLKO,    #01H              //0000,0001 P5.4输出频率为SYSclk
//        MOV      IRC_CLKO,    #02H              //0000,0010 P5.4输出频率为SYSclk/2
//        MOV      IRC_CLKO,    #03H              //0000,0011 P5.4输出频率为SYSclk/4
        SJMP     $

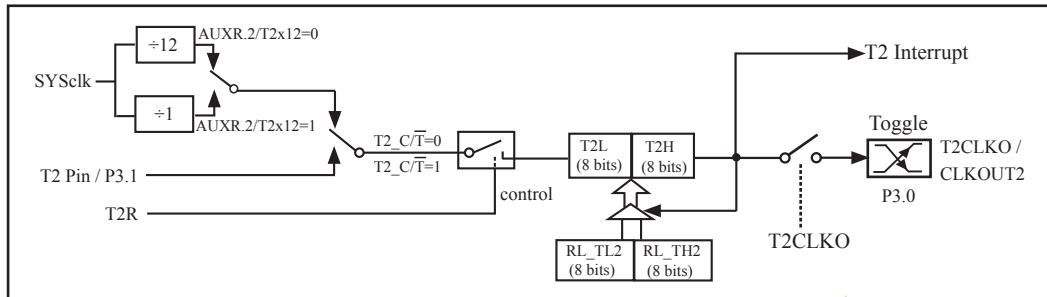
//-----

        END
```

2.1.3.2 定时器2对系统时钟或外部引脚T2的时钟输入进行可编程分频输出及测试程序

T2可以当定时器用，也可以当串口的波特率发生器和可编程时钟输出。

定时器2的原理框图如下：



定时器/计数器2的工作模式: 16位自动重装

如何利用T2CLKO/P3.0管脚输出时钟

AUXR2.2 - T2CLKO: 是否允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

- 1: 允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2,
- 0: 不允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

注意: T2CLKO与CLKOUT2都可表示定时器2(T2)的时钟输出, 下文同。

当T2CLKO/INT_CLKO.2=1时, P3.0管脚配置为定时器2的时钟输出T2CLKO/CLKOUT2。

输出时钟频率 = $T2 \text{ 溢出率} / 2$

如果 $T2_C/\overline{T}=0$, 定时器/计数器T2对内部系统时钟计数, 则:

T2工作在1T模式(AUXR.2/T2x12=1)时的输出时钟频率 = $(SYSclk) / (65536 - [RL_TH2, RL_TL2]) / 2$

T2工作在12T模式(AUXR.2/T2x12=0)时的输出时钟频率 = $(SYSclk) / 12 / (65536 - [RL_TH2, RL_TL2]) / 2$

如果 $T2_C/\overline{T}=1$, 定时器/计数器T2是对外部脉冲输入(P3.1/T2)计数, 则:

输出时钟频率 = $(T2_Pin_CLK) / (65536 - [RL_TH2, RL_TL2]) / 2$

RL_TH2为T2H的重装载寄存器, RL_TL2为T2L的重装载寄存器。

用户在程序中如何具体设置T2CLKO/P3.0管脚输出时钟

1. 对定时器2寄存器T2H/T2L送16位重装载值, $[T2H, T2L] = \#reload_data$
2. 对AUXR寄存器中的T2R位置1, 让定时器2运行
3. 对AUXR2/INT_CLKO寄存器中的T2CLKO位置1, 让定时器2的溢出在P3.0口输出时钟。

注意: 当定时器/计数器2用作可编程时钟输出时, 不要允许相应的定时器中断, 免得CPU反复进中断, 在特殊情况下也可允许定时器/计数器2中断。

下面是定时器2对内部系统时钟或外部引脚T2/P3.1的时钟输入进行可编程时钟分频输出的程序举例(C和汇编):

1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2的可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中,选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"

typedef unsigned char    BYTE;
typedef unsigned int     WORD;

#define FOSC 18432000L

//-----

sfr    AUXR          = 0x8e;           //辅助特殊功能寄存器
sfr    INT_CLKO      = 0x8f;           //唤醒和时钟输出功能寄存器
sfr    T2H           = 0xD6;           //定时器2高8位
sfr    T2L           = 0xD7;           //定时器2低8位

sbit   T2CLKO        = P3^0;           //定时器2的时钟输出脚

#define F38_4KHz      (65536-FOSC/2/38400) //1T模式
//#define F38_4KHz      (65536-FOSC/2/12/38400) //12T模式

//-----

void main()
{
    AUXR  |= 0x04;           //定时器2为1T模式
    //    AUXR  &= ~0x04;       //定时器2为12T模式

```

```

//      AUXR   &=      ~0x08;           //T2_C/T=0, 对内部时钟进行时钟输出
//      AUXR   |=      0x08;           //T2_C/T=1, 对T2(P3.1)引脚的外部时钟进行时钟输出

T2L     =      F38_4KHz;               //初始化计时值
T2H     =      F38_4KHz >> 8;

AUXR    |=      0x10;                 //定时器2开始计时
INT_CLKO =      0x04;                 //使能定时器2的时钟输出功能

while (1);                            //程序终止
}

```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码, 请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码, 请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

AUXR      DATA  08EH      //辅助特殊功能寄存器
INT_CLKO   DATA  08FH      //唤醒和时钟输出功能寄存器
T2H        DATA  0D6H      //定时器2高8位
T2L        DATA  0D7H      //定时器2低8位

T2CLKO     BIT    P3.0      //定时器2的时钟输出脚

F38_4KHz   EQU    0FF10H    //38.4KHz(1T模式下, 65536-18432000/2/38400)
//F38_4KHz EQU    0FFECH    //38.4KHz(12T模式下, (65536-18432000/2/12/38400))

//-----

```

```
ORG    0000H
LJMP   MAIN                      //复位入口

//-----

ORG    0100H
MAIN:
MOV    SP,    #3FH

ORL    AUXR,  #04H              //定时器2为1T模式
// ANL    AUXR,  #0FBH          //定时器2为12T模式

ANL    AUXR,  #0F7H             //T2_C/T=0, 对内部时钟进行时钟输出
// ORL    AUXR,  #08H          //T2_C/T=1, 对T2(P3.1) 引脚的外部时钟进行时钟输出

MOV    T2L,   #LOW F38_4KHz     //初始化计时值
MOV    T2H,   #HIGH F38_4KHz
ORL    AUXR,  #10H              //定时器2开始计时
MOV    INT_CLKO, #04H           //使能定时器2的时钟输出功能

SJMP   $                        //程序终止

;-----

END
```

2.2 STC15F104ESW系列单片机的省电模式

STC15F104ESW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，STC15F104ESW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 (PCON2)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	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

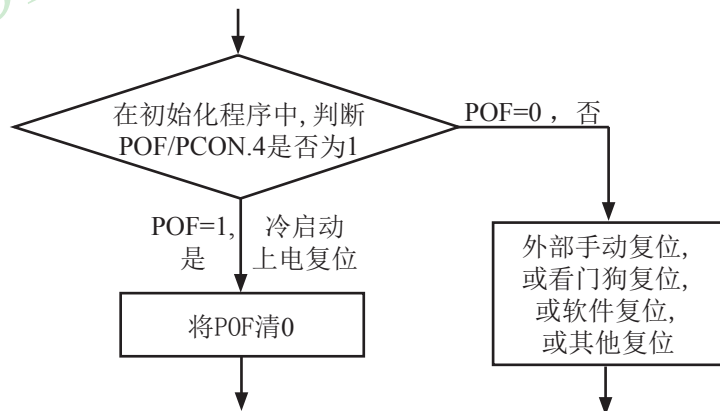
LVDF：低压检测标志位，同时也是低压检测中断请求标志位。

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，上电复位标志位为1，可由软件清0。

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



判断复位种类流程图

PD：将其置1时，进入Power Down模式，可由外部中断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, INT2/P3.6, INT3/P3.7, 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。

SMOD, SMOD0：与电源控制无关，与串口有关，在此不作介绍。

STC MCU Limited

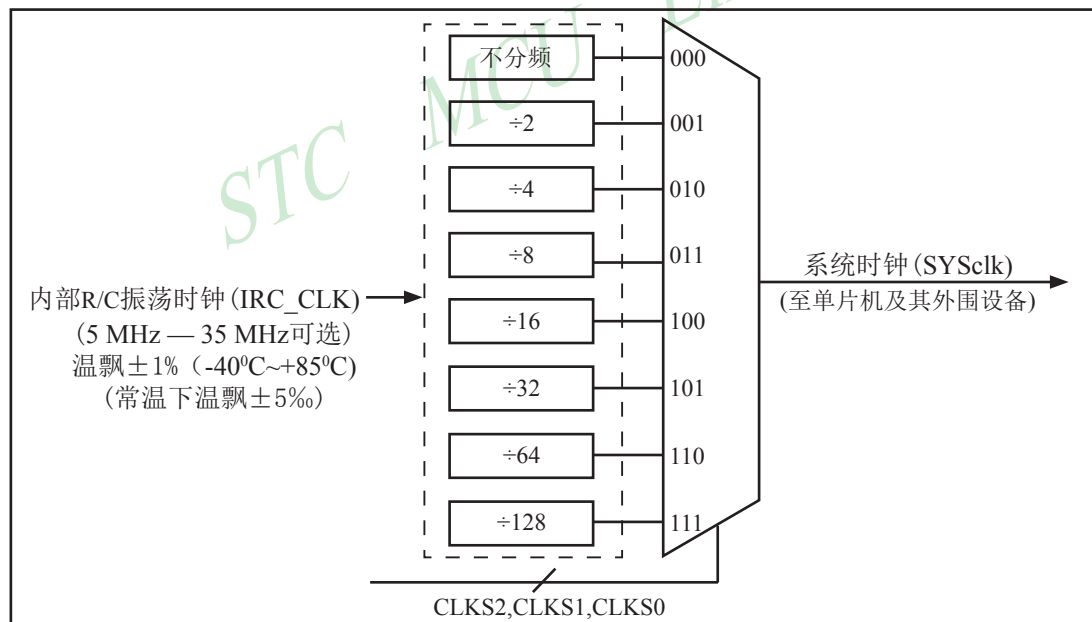
2.2.1 低速模式及其测试程序(C和汇编)

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 (PCON2)各位的定义如下：

SFR Name	SFR Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CLK_DIV (PCON2)	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

1.C程序

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 低速模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

sfr CLK_DIV = 0x97; //时钟分频寄存器

//-----

void main()

```
{
    CLK_DIV = 0x00; //系统时钟为内部R/C振荡时钟
    // CLK_DIV = 0x01; //系统时钟为内部R/C振荡时钟/2
    // CLK_DIV = 0x02; //系统时钟为内部R/C振荡时钟/4
    // CLK_DIV = 0x03; //系统时钟为内部R/C振荡时钟/8
    // CLK_DIV = 0x04; //系统时钟为内部R/C振荡时钟/16
    // CLK_DIV = 0x05; //系统时钟为内部R/C振荡时钟/32
    // CLK_DIV = 0x06; //系统时钟为内部R/C振荡时钟/64
    // CLK_DIV = 0x07; //系统时钟为内部R/C振荡时钟/128

    while (1); //程序终止
}
```

2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 低速模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
CLK_DIV  DATA      097H                      //时钟分频寄存器

//-----
ORG      0000H
LJMP     MAIN                      //复位入口
//-----

MAIN:    ORG      0100H
MOV      SP,    #3FH

MOV      CLK_DIV,    #0           //系统时钟为内部R/C振荡时钟
// MOV    CLK_DIV,    #1           //系统时钟为内部R/C振荡时钟/2
// MOV    CLK_DIV,    #2           //系统时钟为内部R/C振荡时钟/4
// MOV    CLK_DIV,    #3           //系统时钟为内部R/C振荡时钟/8
// MOV    CLK_DIV,    #4           //系统时钟为内部R/C振荡时钟/16
// MOV    CLK_DIV,    #5           //系统时钟为内部R/C振荡时钟/32
// MOV    CLK_DIV,    #6           //系统时钟为内部R/C振荡时钟/64
// MOV    CLK_DIV,    #7           //系统时钟为内部R/C振荡时钟/128

SJMP     $                        //程序终止

;-----

END
```

2.2.2 空闲模式(功耗<1mA)及其测试程序(C和汇编)

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、A/D转换等仍正常运行。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20us, 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。

1. C程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 空闲模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
void main()
{
    while (1)
    {
        PCON |= 0x01;           //将IDL(PCON.0)置1,MCU将进入空闲模式
        _nop_();                //此时CPU无时钟,不执行指令
        _nop_();                //内部中断信号和外部复位信号可以终止空闲模式
        _nop_();
        _nop_();
    }
}
```

2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 空闲模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

//-----

ORG 0000H

LJMP MAIN

//复位入口

//-----

ORG 0100H

MAIN:

MOV SP, #3FH

LOOP:

MOV PCON, #01H

NOP

NOP

NOP

NOP

JMP LOOP

//将IDL(PCON.0)置1,MCU将进入空闲模式

//此时CPU无时钟,不执行指令

//内部中断信号和外部复位信号可以终止空闲模式

END

2.2.3 掉电模式/停机模式及其测试程序(C和汇编)

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/停机模式后, 单片机所使用的时钟(内部系统时钟或外部晶体/时钟)停振, 由于无时钟源, CPU、定时器、看门狗等功能模块停止工作, 外部中断(INT0/INT1/ $\overline{\text{INT2}}$ / $\overline{\text{INT3}}$ / $\overline{\text{INT4}}$)和继续工作。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/停机模式后, 所有I/O口、SFRs(特殊功能寄存器)维持进入掉电模式/停机模式前那一刻的状态不变。如果掉电唤醒专用定时器在进入掉电模式之前被打开(即在进入掉电模式/停机模式之前WKTEN/WKTCH.7=1), 则进入掉电模式/停机模式后, 掉电唤醒专用定时器将开始工作。

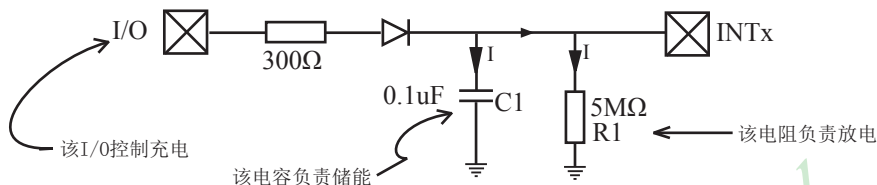
如果STC15F104ESW系列单片机内置掉电唤醒专用定时器被允许(通过软件将WKTCH寄存器中的WKTEN/WKTCH.7位置‘1’, 就可以打开内部掉电唤醒专用定时器), 当MCU进入掉电模式/停机模式时, MCU可由该掉电唤醒专用定时器唤醒。掉电唤醒专用定时器将MCU从掉电模式/停机模式唤醒的执行过程是: 一旦MCU进入掉电模式/停机模式, 内部掉电唤醒专用定时器[WKTCH_CNT, WKTCL_CNT]就从7FFFH开始计数, 直到计数到与{WKTCH[6:0], WKTCL[7:0]}寄存器所设定的计数值相等后就让系统时钟开始振荡; 如果主时钟使用的是内部系统时钟(由用户在ISP烧录程序时自行设置), MCU在等待64个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 就将时钟供给CPU、定时器、看门狗等功能模块工作; 如果主时钟使用的是外部晶体或时钟(由用户在ISP烧录程序时自行设置), MCU在等待1024个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 就将时钟供给CPU、定时器、看门狗、A/D转换等功能模块工作; CPU获得时钟后, 程序从上次设置单片机进入掉电模式语句的下一条语句开始往下执行。掉电唤醒之后, WKTCH_CNT和WKTCL_CNT的内容保持不变, 因此可以通过读[WKTCH, WKTCL]的内容(实际上是读[WKTCH_CNT, WKTCL_CNT]的内容)读出单片机在停机模式/掉电模式所等待的时间。

除掉电唤醒专用定时器外, 还可将掉电模式/停机模式唤醒的中断有: INT0/P3.2, INT1/P3.3 (INT0/INT1上升沿下降沿中断均可), $\overline{\text{INT2}}$ /P3.6, $\overline{\text{INT3}}$ /P3.7, $\overline{\text{INT4}}$ /P3.0 ($\overline{\text{INT2}}$ / $\overline{\text{INT3}}$ / $\overline{\text{INT4}}$ 仅可下降沿中断)。如果掉电模式/停机模式是由外部中断INT0(上升沿+下降沿中断)、INT1(上升沿+下降沿中断)、 $\overline{\text{INT2}}$ (仅可下降沿中断)、 $\overline{\text{INT3}}$ (仅可下降沿中断)或 $\overline{\text{INT4}}$ (仅可下降沿中断)唤醒, 则掉电唤醒之后CPU首先执行设置单片机进入掉电模式的语句的下一条语句(建议在设置单片机进入掉电模式的语句后多加几个NOP空指令), 然后执行相应的中断服务程序。

另外, 在串行中断被允许后, 串行口的接收管脚RxD(可以在RxD/P3.0, RxD_3/P3.6之间切换)如发生由高到低的变化时(起始位接收)也可以将MCU从掉电模式/停机模式唤醒。当MCU由RxD或RxD2唤醒时, 如果主时钟使用的是内部系统时钟(由用户在ISP烧录程序时自行设置), MCU在等待64个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 就将时钟供给CPU工作; 如果主时钟使用的是外部晶体或时钟(由用户在ISP烧录程序时自行设置), MCU在等待1024个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 就将时钟供给CPU工作; CPU获得时钟后, 程序从上次设置单片机进入掉电模式语句的下一条语句开始往下执行。

还有外部复位也可将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。

当用户系统无内置掉电唤醒专用定时器、无外部中断源和RxD下降沿将CPU从掉电模式唤醒时，可用下面的电路能够定时唤醒掉电模式。



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

2.2.3.1 掉电模式/停机模式被唤醒后程序执行流程说明及测试程序(C和汇编)

当STC15F104ESW系列单片机内置掉电唤醒专用定时器被允许(WKTEN=1)，掉电唤醒专用定时器将MCU从掉电模式/停机模式唤醒的执行过程是：一旦MCU进入掉电模式/停机模式，内部掉电唤醒专用定时器[WKTCH_CNT, WKTCL_CNT]就从7FFFH开始计数，直到计数到与{WKTCH[6:0], WKTCL[7:0]}寄存器所设定的计数值相等后就让系统时钟开始振荡；如果主时钟使用的是内部系统时钟(由用户在ISP烧录程序时自行设置)，MCU在等待64个时钟后，就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态，就将时钟供给CPU工作；如果主时钟使用的是外部晶体或时钟(由用户在ISP烧录程序时自行设置)，MCU在等待1024个时钟后，就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态，就将时钟供给CPU工作；CPU获得时钟后，程序从上次设置单片机进入掉电模式语句的下一条语句开始往下执行。

掉电模式/停机模式由中断INT0/P3.2, INT1/P3.3 (INT0/INT1上升沿下降沿中断均可), $\overline{\text{INT2}}$ /P3.6, $\overline{\text{INT3}}$ /P3.7, $\overline{\text{INT4}}$ /P3.0 ($\overline{\text{INT2}}/\overline{\text{INT3}}/\overline{\text{INT4}}$ 仅可下降沿中断)唤醒之后程序的执行流程为：CPU首先执行从上次设置单片机进入掉电模式语句的下一条语句（建议在设置单片机进入掉电模式的语句后多加几个NOP空指令），然后执行相应的中断服务程序。

掉电模式/停机模式由串行口的接收管脚RxD(可以在RxD/P3.0,RxD_3/P3.6之间切换)的下降沿(不产生中断)唤醒后的程序执行流程：当MCU由RxD的下降沿唤醒后，如果主时钟使用的是

内部系统时钟, MCU在等待64个时钟(由用户在ISP烧录程序时自行设置)后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 才将时钟供给CPU工作; 如果主时钟使用的是外部晶体或时钟, MCU在等待1024个时钟(由用户在ISP烧录程序时自行设置)后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 才将时钟供给CPU工作; CPU获得时钟后, 程序从上次设置单片机进入掉电模式语句的下一条语句开始往下执行。

1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 掉电模式中指令执行流程说明-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"
#include "intrins.h"
//-----

void main()
{
    while (1)
    {
        PCON |= 0x02;          //将STOP(PCON.1)置1,MCU将进入掉电模式
        _nop_();
        //当有有效的掉电唤醒源产生时,若使用的是内部振荡器,则立即启动内部振荡器,在64个
        //时钟周期后,将时钟提供给MCU,作为系统时钟若使用的是外部振荡器,则立即启动外
        //部振荡器,在1024个时钟周期后,将时钟提供给MCU,作为系统时钟在时钟信号到达CPU
        //后,若掉电唤醒源是内部32K掉电唤醒定时器、RxD时, CPU直接从此语句开始向下
        //执行程序代码, 而不产生中断
        //若掉电唤醒源是INT0、INT1、INT2、INT3、INT4时, 则CPU首先执行此语句,
        //然后执行中断服务程序

        _nop_();
        _nop_();
    }
}

```


2. 汇编程序：

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 掉电模式中指令执行流程说明-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz
//-----
                ORG      0000H
                LJMP     MAIN                //复位入口

//-----

MAIN:           ORG      0100H

                MOV      SP,    #3FH

LOOP:           MOV      PCON, #02H          //将STOP(PCON.1)置1,MCU将进入掉电模式

                NOP                               //当有有效的掉电唤醒源产生时,若使用的是内部振荡器,
                                                //则立即启动内部振荡器,在64个时钟周期后,
                                                //将时钟提供给MCU,作为系统时钟
                                                //若使用的是外部振荡器,则立即启动外部振荡器,
                                                //在1024个时钟周期后,将时钟提供给MCU,作为系统时钟
                                                //在时钟信号到达CPU后,若掉电唤醒源是内部32K掉电唤醒
                                                //定时器、RxD时, CPU直接从此语句开始向下执行程序
                                                //代码, 而不产生中断
                                                //若掉电唤醒源是INT0、INT1、INT2、INT3、INT4时,
                                                //则CPU首先首先执行此语句,然后执行中断服务程序

                NOP
                NOP
                NOP
                JMP      LOOP

;-----

                END

```


2.2.3.2 用掉电唤醒专用定时器唤醒掉电模式/停机模式的测试程序(C和汇编)

*/*利用内部专用掉电唤醒定时器来唤醒掉电模式的示例程序(C程序)*

1. C程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列 掉电唤醒定时器举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序---*/
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序---*/
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sfr      WKTCL = 0xaa; //掉电唤醒定时器计时低字节
sfr      WKTCH = 0xab; //掉电唤醒定时器计时高字节

sbit     P10 = P1^0;

//-----

void main()
{
    WKTCL = 49; //设置唤醒周期为488us*(49+1) = 24.4ms
    WKTCH = 0x80; //使能掉电唤醒定时器

    while (1)
    {
        PCON = 0x02; //进入掉电模式
        _nop_();
        _nop_();
        P10 = !P10; //掉电唤醒后,取反测试口
    }
}
```

2. 汇编程序：

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列 掉电唤醒定时器举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序----*/
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

WKTCL DATA 0AAH //掉电唤醒定时器计时低字节
WKTCH DATA 0ABH //掉电唤醒定时器计时高字节

//-----

        ORG    0000H
        LJMP   MAIN //复位入口

//-----

MAIN:    ORG    0100H

        MOV    SP,    #3FH

        MOV    WKTCL, #49 //设置唤醒周期为488us*(49+1) = 24.4ms
        MOV    WKTCH, #80H //使能掉电唤醒定时器

LOOP:    MOV    PCON,  #02H //进入掉电模式
        NOP
        NOP
        CPL    P1.0 //掉电唤醒后,取反测试口
        JMP    LOOP

        SJMP   $

;-----

        END
```

2.2.3.3 用外部中断INT0(上升沿+下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT0唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
bit    FLAG;                                //1:上升沿中断 0:下降沿中断
sbit   P10    =    P1^0;

//-----
//中断服务程序
void exint0() interrupt 0                    //INT0中断入口
{
    P10    =    !P10;                        //将测试口取反
    FLAG    =    INT0;                       //保存INT0口的状态, INT0=0(下降沿); INT0=1(上升沿)
}
//-----

void main()
{
    IT0 = 0;                                //设置INT0的中断类型为上升沿和下降沿,上升沿和下降沿均可唤醒
    // IT0 = 1;                                //设置INT0的中断类型为仅下降沿,下降沿唤醒

    EX0 = 1;                                //使能INT0中断
    EA = 1;

    while (1)
    {
        PCON = 0x02;                        //MCU进入掉电模式
        _nop_();                             //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
        _nop_();
    }
}

```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT0唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

                ORG        0000H
                LJMP       MAIN           //复位入口

                ORG        0003H           //INT0中断入口
                LJMP       EXINT0
//-----

MAIN:          ORG        0100H

                MOV        SP,    #3FH

                CLR        IT0           //设置INT0的中断类型为上升沿和下降沿,上升沿和下降沿均可唤醒
//              SETB       IT0           //设置INT0的中断类型为仅下降沿,下降沿唤醒

                SETB       EX0           //使能INT0中断
                SETB       EA

LOOP:          MOV        PCON,    #02H   //MCU进入掉电模式
                NOP                     //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
                NOP
                SJMP       LOOP
//-----
//中断服务程序

EXINT0:        CPL        P1.0           //将测试口取反
                PUSH       PSW
                MOV        C,    INT0     //读取INT0口的状态
                MOV        FLAG,    C    //保存, INT0=0(下降沿); INT0=1(上升沿)
                POP        PSW
                RETI
;-----
                END

```

2.2.3.4 用外部中断INT1(上升沿+下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT1唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
bit    FLAG;           //1:上升沿中断 0:下降沿中断
sbit   P10    =        P1^0;

//-----
//中断服务程序
void exint1() interrupt 2           //INT1中断入口
{
    P10    =        !P10;          //将测试口取反
    FLAG    =        INT1;          //保存INT1口的状态, INT1=0(下降沿); INT1=1(上升沿)
}
//-----

void main()
{
    IT1    =        0;              //设置INT1的中断类型为上升沿和下降沿,上升沿和下降沿均可唤醒
    //      IT1 = 1;                //设置INT1的中断类型为仅下降沿,下降沿唤醒

    EX1 = 1;                        //使能INT1中断
    EA = 1;

    while (1)
    {
        PCON = 0x02;               //MCU进入掉电模式
        _nop_();                    //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
        _nop_();
    }
}

```

2. 汇编程序：

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT1唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

                ORG    0000H
                LJMP   MAIN                //复位入口

                ORG    0013H                //INT1中断入口
                LJMP   EXINT1
//-----

MAIN:
                ORG    0100H
                MOV     SP, #3FH

                CLR     IT1                //设置INT1的中断类型为上升沿和下降沿,上升沿和下降沿均可唤醒
//              SETB    IT1                //设置INT1的中断类型为仅下降沿,下降沿唤醒

                SETB    EX1                //使能INT1中断
                SETB    EA

LOOP:
                MOV     PCON, #02H        //MCU进入掉电模式
                NOP                     //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
                NOP
                SJMP    LOOP

;-----

EXINT1:
                CPL     P1.0                //取反测试口
                PUSH    PSW
                MOV     C, INT1            //读取INT1口的状态
                MOV     FLAG, C            //保存, INT1=0(下降沿); INT0=1(上升沿)
                POP     PSW
                RETI

;-----

                END

```

2.2.3.5 用外部中断INT2(下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT2下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"
#include "intrins.h"
//-----

sfr      INT_CLKO    = 0x8F;    //外部中断与时钟输出控制寄存器
sbit     INT2       = P3^6;    //INT2引脚定义

sbit     P10        = P1^0;
//-----
//中断服务程序
void exint2() interrupt 10
{
    P10    = !    P10;          //将测试口取反

    //      INT_CLKO  &=  0xEF;    //若需要手动清除中断标志,可先关闭中断,
    //                                //此时系统会自动 清除内部的中断标志
    //      INT_CLKO  |=  0x10;    //然后再开中断即可
}
//-----

void main()
{
    INT_CLKO |= 0x10;          //(EX2 = 1)使能INT2下降沿中断
    EA = 1;

    while (1)
    {
        PCON = 0x02;          //MCU进入掉电模式
        _nop_();              //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
        _nop_();
    }
}

```

2. 汇编程序：

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT2下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

INT_CLKO      DATA    08FH          //外部中断与时钟输出控制寄存器
INT2           BIT      P3.6         //INT2引脚定义
//-----

                ORG      0000H
                LJMP     MAIN          //复位入口

                ORG      0053H        //INT2中断入口
                LJMP     EXINT2
//-----

MAIN:           ORG      0100H
                MOV      SP, #3FH
                ORL      INT_CLKO,    #10H  //(EX2 = 1)使能INT2下降沿中断
                SETB     EA

LOOP:           MOV      PCON, #02H      //MCU进入掉电模式
                NOP                      //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务
程序           NOP
                SJMP     LOOP
//-----
//中断服务程序

EXINT2:         CPL      P1.0           //将测试口取反
//              ANL      INT_CLKO,    #0EFH //若需要手动清除中断标志,可先关闭中断,
//                                      //此时系统会自动清除内部的中断标志
//              ORL      INT_CLKO,    #10H //然后再开中断即可
                RETI
;-----

                END

```


2.2.3.6 用外部中断INT3(下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT3下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"
#include "intrins.h"
//-----

sfr      INT_CLKO    =      0x8F;    //外部中断与时钟输出控制寄存器
sbit     INT3        =      P3^7;    //INT3引脚定义

sbit     P10         =      P1^0;

//-----
//中断服务程序
void exint3() interrupt 11
{
    P10      = !      P10;            //将测试口取反

//      INT_CLKO      &=      0xDF;    //若需要手动清除中断标志,可先关闭中断,
//                                      //此时系统会自动清除内部的中断标志

//      INT_CLKO      |=      0x20;    //然后再开中断即可
}
//-----

void main()
{
    INT_CLKO |= 0x20;                //(EX3 = 1)使能INT3下降沿中断
    EA = 1;

    while (1)
    {
        PCON = 0x02;                //MCU进入掉电模式
        _nop_();                    //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
        _nop_();
    }
}

```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT3下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

INT_CLKO    DATA    08FH                //外部中断与时钟输出控制寄存器
INT3        BIT      P3.7                //INT3引脚定义
//-----

                ORG     0000H
                LJMP    MAIN                //复位入口

                ORG     005BH
                LJMP    EXINT3              //INT3中断入口
//-----

MAIN:
                ORG     0100H
                MOV     SP,#3FH
                ORL     INT_CLKO,    #20H    //(EX3 = 1)使能INT3下降沿中断
                SETB    EA

LOOP:
                MOV     PCON,    #02H        //MCU进入掉电模式
                NOP                     //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
                NOP
                SJMP    LOOP
//-----
//中断服务程序
EXINT3:
                CPL     P1.0                //将测试口取反

//                ANL     INT_CLKO,    #0DFH    //若需要手动清除中断标志,可先关闭中断,
//                                           //此时系统会自动清除内部的中断标志
//                ORL     INT_CLKO,    #20H    //然后再开中断即可

                RETI
;-----
                END

```

2.2.3.7 用外部中断INT4(下降沿)唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT4下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
sfr      INT_CLKO    =      0x8F;    //外部中断与时钟输出控制寄存器
sbit     INT4        =      P3^0;    //INT4引脚定义

sbit     P10         =      P1^0;

//-----
//中断服务程序
void exint4() interrupt 16
{
    P10      =      !P10;            //将测试口取反

//      INT_CLKO      &=      0xBF;    //若需要手动清除中断标志,可先关闭中断,
//                                      //此时系统会自动清除内部的中断标志
//      INT_CLKO      |=      0x40;    //然后再开中断即可
}

//-----
void main()
{
    INT_CLKO |= 0x40;                //(EX4 = 1)使能INT4下降沿中断
    EA = 1;

    while (1)
    {
        PCON  =  0x02;                //MCU进入掉电模式
        _nop_();                      //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
        _nop_();
    }
}

```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 INT4下降沿唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

INT_CLKO      DATA    08FH                //外部中断与时钟输出控制寄存器
INT4           BIT      P3.0                //INT4引脚定义

//-----
                ORG      0000H
                LJMP     MAIN                //复位入口

                ORG      0083H                //INT4中断入口
                LJMP     EXINT4

//-----
MAIN:           ORG      0100H
                MOV      SP,    #3FH
                ORL       INT_CLKO,    #40H    //(EX4 = 1)使能INT4下降沿中断
                SETB     EA

LOOP:           MOV      PCON,    #02H        //MCU进入掉电模式
                NOP                      //掉电模式被唤醒后,首先执行此语句,然后再进入中断服务程序
                NOP
                SJMP     LOOP

//-----
//中断服务程序
EXINT4:         CPL      P1.0                //将测试口取反

//             ANL      INT_CLKO,    #0BFH    //若需要手动清除中断标志,可先关闭中断,
//                                           //此时系统会自动清除内部的中断标志
//             ORL       INT_CLKO,    #40H    //然后再开中断即可

                RETI

;-----
                END

```

2.2.3.8 用RxD管脚由高到低的变化唤醒掉电模式/停机模式的测试程序(C和汇编)

1. C程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 RxD串行中断1唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sfr    AUXR    =    0x8e;    //辅助寄存器
sfr    T2H     =    0xd6;    //定时器2高8位
sfr    T2L     =    0xd7;    //定时器2低8位

sfr    P_SW2   =    0xBA;    //外设功能切换寄存器2

#define S1_S1   0x80        //P_SW2.7

sbit   P10     =    P1^0;

//-----

void main()
{
    P_SW2  &=    ~S1_S1;    //(P3.0/RxD, P3.1/TxD),此时P3.0脚的下沿沿有效

//    P_SW2  |=    S1_S1;    //(P3.6/RxD_2, P3.7/TxD_2),此时P3.6脚的下沿沿有效

    SCON    =    0x50;    //8位可变波特率
    T2L     =    (65536 - (FOSC/4/BAUD));    //设置波特率重装值
    T2H     =    (65536 - (FOSC/4/BAUD))>>8;
    AUXR    =    0x14;    //T2为1T模式, 并启动定时器2
    AUXR    |=    0x01;    //选择定时器2为串口的波特率发生器

    ES      =    1;
    EA      =    1;
}

```

```
while (1)
{
    PCON = 0x02;           //MCU进入掉电模式
    _nop_();               //掉电模式被唤醒后,直接从此语句开始向下执行,
                           //不进入中断服务程序

    _nop_();
    P10 = !P10;            //将测试口取反
}

/*-----
UART 中断服务程序
-----*/
void Uart() interrupt 4 using 1
{
    if (RI)
    {
        RI = 0;           //清除RI位
        P0 = SBUF;         //P0显示串口数据
    }
    if (TI)
    {
        TI = 0;           //清除TI位
    }
}
```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited -----*/
/* --- STC15F104ESW 系列 RxD串行中断1唤醒掉电模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

//-----

```

AUXR EQU 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

P_SW2 EQU 0BAH //外设功能切换寄存器2

S1_S1 EQU 80H //P_SW2.7

```

//-----

```

ORG 0000H
LJMP MAIN //复位入口

ORG 0023H
LJMP UART_ISR //中断入口

```

//-----

```

MAIN: ORG 0100H

MOV SP, #3FH

ANL P_SW2, #NOT S1_S1 // (P3.0/RxD, P3.1/TxD), 此时P3.0脚的下降沿有效

// ORL P_SW2, #S1_S1 // (P3.6/RxD_2, P3.7/TxD_2), 此时P3.6脚的下降沿有效

MOV SCON, #50H //8位可变波特率
MOV T2L, #0D8H //设置波特率重装值(65536-18432000/4/115200)
MOV T2H, #0FFH
MOV AUXR, #14H //T2为1T模式, 并启动定时器2
ORL AUXR, #01H //选择定时器2为串口的波特率发生器

```

```
        SETB    ES                //打开串口中断
        SETB    A

LOOP:   MOV     PCON, #02H        //MCU进入掉电模式
        NOP                      //掉电模式被唤醒后,直接从此语句开始向下执行,
                                   //不进入中断服务程序

        NOP
        CPL     P1.0             //掉电唤醒后,取反测试口
        SJMP    LOOP

;-----
;UART 中断服务程序
;-----*/
UART_ISR:
        PUSH    ACC
        PUSH    PSW
        JNB     RI,    CHECKTI    //检测RI位
        CLR     RI        //清除RI位
        MOV     P0,    SBUF       //P0显示串口数据
CHECKTI:
        JNB     TI,    ISR_EXIT   //检测TI位
        CLR     TI        //清除TI位
ISR_EXIT:
        POP     PSW
        POP     ACC
        RETI

;-----

        END
```


2.3 复位

STC15F104ESW系列单片机有6种复位方式：外部RST引脚复位，软件复位，掉电复位/上电复位(并可选择增加额外的复位延时180mS，也叫MAX810专用复位电路，其实就是在上电复位后增加一个180mS复位延时)，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。P5.4/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P5.4/RST管脚已在ISP烧录程序时被设置为复位脚，那P5.4/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20us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 软件复位及其测试程序(C和汇编)

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

1. C程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 软件复位举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
sfr      IAP_CONTR = 0xc7;          //IAP控制寄存器

sbit     P10      =      P1^0;
//-----

void delay()
{
    int i;

    for (i=0; i<10000; i++)
    {
        _nop_();
        _nop_();
        _nop_();
        _nop_();
    }
}

void main()
{
    P10 = !P10;                      //上电P1.0闪烁一次,便于观察
    delay();
    P10 = !P10;
    delay();

    IAP_CONTR = 0x20;                //软件复位,系统重新从用户代码区开始运行程序
    // IAP_CONTR = 0x60;              //软件复位,系统重新从ISP代码区开始运行程序

    while (1);
}
```

2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 软件复位举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

IAP_CONTR DATA 0C7H                                //IAP控制寄存器

//-----

        ORG    0000H
        LJMP   MAIN                                //复位入口
//-----

MAIN:    ORG    0100H

        MOV    SP,    #3FH

        CPL    P1.0                                //上电P1.0闪烁一次,便于观察
        LCALL  DELAY
        CPL    P1.0
        LCALL  DELAY

        MOV    IAP_CONTR,    #20H                //软件复位,系统重新从用户代码区开始运行程序
//        MOV    IAP_CONTR,    #60H                //软件复位,系统重新从ISP代码区开始运行程序

        JMP    $

;-----
DELAY:
        MOV    R0,    #0                            //软件延时
        MOV    R1,    #0

WAIT:    DJNZ   R0,    WAIT
        DJNZ   R1,    WAIT
        RET

;-----

        END

```

2.3.3 掉电复位/上电复位

当电源电压VCC低于掉电复位/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，掉电复位/上电复位结束。

2.3.4 MAX810专用复位电路复位

STC15F104ESW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，则以后掉电复位/上电复位后将再产生约180mS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，STC15F104ESW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)。

STC15F104ESW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/L204EA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 °C	25 °C	85 °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STC15F104ESW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 °C	25 °C	85 °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

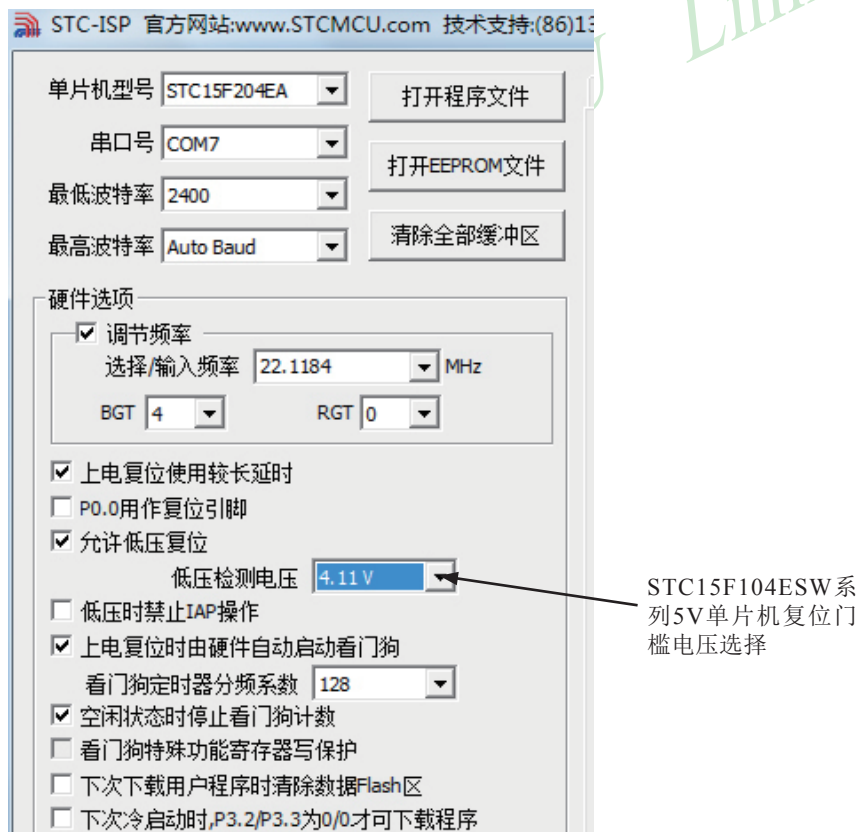
如果用户所使用的是STC15L104ESW系列3.3V单片机，那么用户可以根据单片机的实际工作频率在STC-ISP编程器中选择上表中所列出的低压检测门槛电压作为复位门槛电压。如：常温下工作频率是20MHz以上时，可以选择2.82V电压作为内部低压检测复位门槛电压；常温下工作频率是12MHz以下时，可以选择2.42V电压作为复位门槛电压。

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。

建议在电压偏低时,不要操作EEPROM/IAP,烧录时直接选择“低压禁止IAP操作”。



STC-ISP 官方网站:www.STCMCU.com 技术支持:(86)1

单片机型号 打开程序文件

串口号 打开EEPROM文件

最低波特率 清除全部缓冲区

最高波特率

硬件选项

☒ 调节频率

选择/输入频率 MHz

BGT RGT

☒ 上电复位使用较长延时

☐ P0.0用作复位引脚

☒ 允许低压复位

低压检测电压

☐ 低压时禁止IAP操作

☐ 上电复位时由硬件自动启动看门狗

看门狗定时器分频系数

☒ 空闲状态时停止看门狗计数

☐ 看门狗特殊功能寄存器写保护

☐ 下次下载用户程序时清除数据Flash区

☐ 下次冷启动时,P3.2/P3.3为0/0才可下载程序

下载 停止 重复下载

Limited

STC15L204E系列
3V单片机复位门
槛电压选择

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	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL

LVDF: 低压检测标志位, 同时也是低压检测中断请求标志位。

在正常工作和空闲工作状态时, 如果内部工作电压 V_{cc} 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 V_{cc} 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 V_{cc} 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 V_{cc} 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。

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	EADC	ES	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	PPCA	PLVD	PADC	PS	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。STC15F104ESW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:

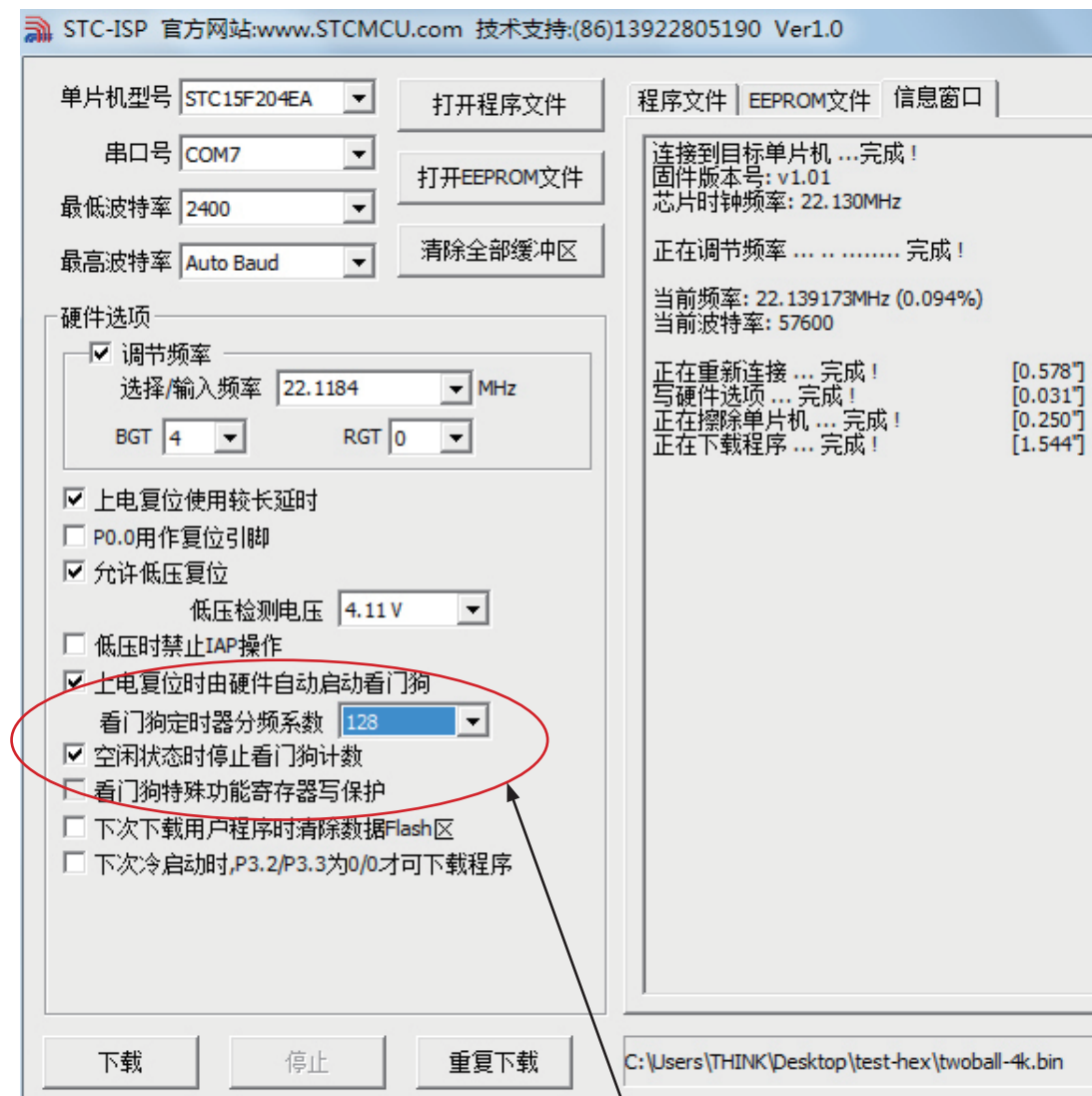
看门狗溢出时间 = $(12 \times \text{Pre-scale} \times 32768) / 12000000 = \text{Pre-scale} \times 393216 / 12000000$

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 = $(12 \times \text{Pre-scale} \times 32768) / 11059200 = \text{Pre-scale} \times 393216 / 11059200$

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



STC-ISP下编程器中看门狗的设置区

看门狗测试程序，在STC的下载板上可以直接测试

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机 看门狗及其溢出时间计算公式-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*--- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/

```

;本演示程序在STC 15系列 ISP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 = $(12 * \text{Pre_scale} * 32768) / \text{Oscillator frequency}$

WDT_CONTR EQU 0C1H ;看门狗地址

WDT_TIME_LED EQU P1.5 ;用 P1.5 控制看门狗溢出时间指示灯,
;看门狗溢出时间可由该指示灯亮的时间长度或熄灭的时间长度表示

WDT_FLAG_LED EQU P1.7

;用 P1.7 控制看门狗溢出复位指示灯,如点亮表示为看门狗溢出复位

Last_WDT_Time_LED_Status EQU 00H ;位变量,存储看门狗溢出时间指示灯的上一次状态位

;WDT 复位时间(所用的Oscillator frequency = 18.432MHz):

;Pre_scale_Word EQU 00111100B ;清0,启动看门狗,预分频数=32, 0.68S

Pre_scale_Word EQU 00111101B ;清0,启动看门狗,预分频数=64, 1.36S

;Pre_scale_Word EQU 00111110B ;清0,启动看门狗,预分频数=128, 2.72S

;Pre_scale_Word EQU 00111111B ;清0,启动看门狗,预分频数=256, 5.44S

ORG 0000H

AJMP MAIN

ORG 0100H

MAIN:

MOV A, WDT_CONTR ;检测是否为看门狗复位

ANL A, #10000000B

JNZ WDT_Reset ;WDT_CONTR.7 = 1,看门狗复位,跳转到看门狗复位程序

;WDT_CONTR.7 = 0,上电复位,冷启动,RAM 单元内容为随机值

SETB Last_WDT_Time_LED_Status ;上电复位,
;初始化看门狗溢出时间指示灯的状态位 = 1

CLR WDT_TIME_LED ;上电复位,点亮看门狗溢出时间指示灯

MOV WDT_CONTR, #Pre_scale_Word ;启动看门狗

WAIT1:

SJMP WAIT1 ;循环执行本语句(停机)，等待看门狗溢出复位

;WDT_CONTR.7 = 1, 看门狗复位，热启动，RAM 单元内容不变，为复位前的值

WDT_Reset: ;看门狗复位，热启动

CLR WDT_FLAG_LED ;是看门狗复位，点亮看门狗溢出复位指示灯

JB Last_WDT_Time_LED_Status, Power_Off_WDT_TIME_LED

;为1熄灭相应的灯，为0亮相应灯

;根据看门狗溢出时间指示灯的上一次状态位设置 WDT_TIME_LED 灯，

;若上次亮本次就熄灭，若上次熄灭本次就亮

CLR WDT_TIME_LED ;上次熄灭本次点亮看门狗溢出时间指示灯

CPL Last_WDT_Time_LED_Status ;将看门狗溢出时间指示灯的上一次状态位取反

WAIT2:

SJMP WAIT2 ;循环执行本语句(停机)，等待看门狗溢出复位

Power_Off_WDT_TIME_LED:

SETB WDT_TIME_LED ;上次亮本次就熄灭看门狗溢出时间指示灯

CPL Last_WDT_Time_LED_Status ;将看门狗溢出时间指示灯的上一次状态位取反

WAIT3:

SJMP WAIT3 ;循环执行本语句(停机)，等待看门狗溢出复位

END

2.3.7 冷启动复位和热启动复位

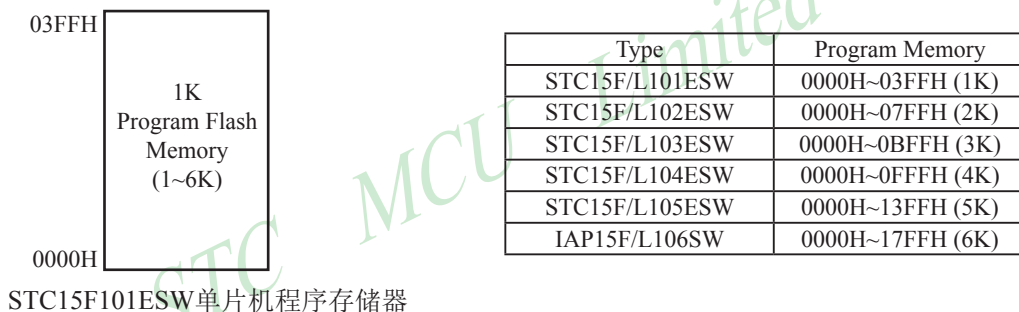
	复位源	现象
热启动复位	内部看门狗复位	会使单片机直接从用户程序区0000H处开始执行用户程序
	通过控制RESET脚产生的硬复位	会使系统从用户程序区0000H处开始直接执行用户程序
	通过对IAP_CONTR寄存器送入20H产生的软复位	会使系统从用户程序区0000H处开始直接执行用户程序
	通过对IAP_CONTR寄存器送入60H产生的软复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用户程序区执行用户程序
冷启动复位	系统停电后再上电引起的硬复位	会使系统从系统ISP监控程序区开始执行程序，检测不到合法的ISP下载命令流后，会软复位到用户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

STC15F104ESW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。STC15F104ESW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。STC15F104ESW系列单片机内部有128字节的数据存储器与特殊功能寄存器(SFRs)存储区。

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。STC15F104ESW系列单片机内部集成了1K~6K字节的Flash程序存储器。STC15F104ESW系列各种型号单片机的程序Flash存储器的地址如下表所示。



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)

STC15系列单片机内部集成了128字节的数据存储器RAM(与传统8051兼容)及特殊功能寄存器区。128字节的RAM(地址范围为00H~7FH)可用于存放程序执行的中间结果和过程数据。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(地址范围为80H~FFH)。

内部RAM的结构如下图所示。



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

PSW：程序状态字寄存器

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80H或更大的值为宜。STC15F104ESW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。STC15F104ESW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。[请不要再抄袭我们的设计、规格和管脚排列,再抄袭就很无耻了。](#)

STC15F104ESW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						T2H RL_TH2 0000,0000	T2L RL_TL2 0000,0000	0D7H
0C8H	P5 xx11,xxxx	P5M1 xx00,xxxx	P5M0 xx00,xxxx						0CFH
0C0H		WDT_CONTR 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 x0x0,x0x0	SADEN	P_SW2 0xxx,xxxx	IRC_CLKO xxxx,xx00					0BFH
0B0H	P3 11xx,1111	P3M1 00xx,0000	P3M0 00xx,0000						0B7H
0A8H	IE 00x0,xx00	SADDR	WKTCL WKTCL_CNT 0111 1111	WKTCH WKTCH_CNT 0111 1111				IE2 xxxx,x0xx	0AFH
0A0H			AUXR1 P_SW1 xxxx,0000	Don't use	Don't use	Don't use		Don't use	0A7H
098H	SCON 0000,0000	SBUF xxxx,xxxx					Don't use	Don't use	09FH
090H	P1 xx11,1111	P1M1 xx00,0000	P1M0 xx00,0000					CLK_DIV PCON2	097H
088H	TCON xxxx,0000						AUXR xx00,00xx	INT_CLKO AUXR2 x000 00xx	08FH
080H		SP 0000,0111	DPL 0000,0000	DPH 0000,0000				PCON 0011,0000	087H
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

可位寻址

不可位寻址

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

符号		描述	地址	位地址及符号										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	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B		
TCON		外部中断0和外部中断1的中断请求控制寄存器	88H	-	-	-	-	IE1	IT1	IE0	IT0	xxxx 0000B		
AUXR		辅助寄存器	8EH	-	-	UART_M0x6	T2R	T2_C \bar{T}	T2x12	-	-	xx00 00xxB		
INT_CLKO	AUXR2	外部中断允许和时钟输出寄存器	8FH	-	EX4	EX3	EX2	LVD_WAKE	T2CLKO	-	-	x000 00xxB		
P1				Port 1	90H	-	-	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	xx11 1111B
P1M1		P1口模式配置寄存器1	91H											xx00 0000B
P1M0		P1口模式配置寄存器0	92H											xx00 0000B
CLK_DIV	PCON2	时钟分频寄存器	97h	-	-	-	-	-	CLKS2	CLKS1	CLKS0	xxxx x000B		
SCON		串口控制寄存器	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000,0000		
SBUF		串口数据缓冲器	99H											xxxx,xxxx
AUXR1	P_SW1	辅助寄存器1	A2H	-	-	-	-	-	GF2	ADRJ	-	DPS	xxxx 0000B	
IE		中断允许寄存器	A8H	EA	ELVD	-	ES	-	EX1	-	EX0	00x0 x0x0B		
SADDR		从机地址控制寄存器	A9H											0000 0000B
WKTCL	WKTCL_CNT	掉电唤醒专用定时器控制寄存器低8位	AAH	-	-	-	-	-	-	-	-	0111 1111B		
WKTCH	WKTCH_CNT	掉电唤醒专用定时器控制寄存器高8位	ABH	WKTEN	-	-	-	-	-	-	-	0111 1111B		
IE2		中断允许寄存器	AFH	-	-	-	-	-	ET2	-	-	xxxx x0xxB		
P3		Port 3	B0H	P3.7	P3.6	-	-	P3.3	P3.2	P3.1	P3.0	11xx 1111B		
P3M1		P3口模式配置寄存器1	B1H											00xx 0000B
P3M0		P3口模式配置寄存器0	B2H											00xx 0000B
IP		中断优先级寄存器	B8H	-	PLVD	-	PS	-	PX1	-	PX0	x0x0 x0x0B		
SADEN		从机地址掩模寄存器	B9H											0000 0000B
P_SW2		外围设备功能切换控制寄存器	BAH	S1_S1	-	-	-	-	-	-	-	0xxx xxxxB		
IRC_CLKO		内部R/C时钟输出寄存器	BBH	-	-	-	-	-	-	IRCS1	IRCS0	xxxx,xx00B		
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

符号	描述	地址	位地址及符号								复位值
			MSB				LSB				
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
P5	Port 5	C8H	-	-	P5.5	P5.4	-	-	-	-	xx11 xxxxB
P5M1	P5口模式配置寄存器1	C9H									xx00 xxxxB
P5M0	P5口模式配置寄存器0	CAH									xx00 xxxxB
PSW	程序状态字寄存器	D0H	CY	AC	F0	RS1	RS0	OV	-	P	0000 00x0B
T2H	定时器2高8位寄存器	D6H									0000 0000B
T2L	定时器2低8位寄存器	D7H									0000 0000B
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80H或更大的值为宜。STC15F104ESW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。

STC15F104ESW系列单片机设计了两个16位的数据指针DPTR0和DPTR1，这两个数据指针共用同一个地址空间，可通过设置DPS/AUXR1.0来选择具体被使用的数据指针。

STC15F104ESW系列8051 单片机 双数据指针 特殊功能寄存器

Mnemonic	Address	Name	7	6	5	4	3	2	1	0	Reset Value
AUXR1	A2H	Auxiliary Register 1	-	-	-	-	GF2	ADJ	0	DPS	xxxx,0000

DPS DPTR registers select bit. DPTR 寄存器选择位

0: DPTR0 is selected DPTR0被选择

1: DPTR1 is selected DPTR1被选择

此系列单片机有两个16-bit数据指针, DPTR0, DPTR1. 当DPS选择位为0时, 选择DPTR0, 当DPS选择位为1时, 选择DPTR1.

AUXR1特殊功能寄存器, 位于A2H单元, 其中的位不可用布尔指令快速访问. 但由于DPS位位于bit0, 故对AUXR1寄存器用INC指令, DPS位便会反转, 由0变成1或由1变成0, 即可实现双数据指针的快速切换.

第4章 STC15F104ESW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

STC15F104ESW系列单片机最多有14个I/O口： P1.0~P1.5, P3.0~P3.3, P3.6~P3.7, P5.4~P5.5。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）、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。STC15F104ESW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，但单片机的整个芯片最大不要超过90mA。

I/O口工作类型设定

P5口设定 <X, X, P5.5, P5.4, X, X, X, X> (P5口地址：C8H)

P5M1 [7 : 0] 寄存器P5M1地址为C9H	P5M0 [7 : 0] 寄存器P5M0地址为CAH	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

注：寄存器P5M1和P5M0的地址分别为C9H和CAH。

举例： MOV P5M1, #00100000B
MOV P5M0, #00110000B
;P5.5为开漏,P5.4为强推挽输出

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

P3M1 [7 : 0] 寄存器P3M1地址为B1H	P3M0 [7 : 0] 寄存器P3M0地址为B2H	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

注：寄存器P3M1和P3M0的地址分别为B1H和B2H。

举例： MOV P3M1, #10001000B
MOV P3M0, #11000000B
;P3.7为开漏,P3.6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

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

P1M1 [7 : 0] 寄存器P1M1地址为91H	P1M0 [7 : 0] 寄存器P1M0地址为92H	I/O 口模式 (P1.x 如做A/D使用, 需先将其设置成开漏或高阻输入)
0	0	准双向口(传统8051 I/O 口模式) , 灌电流可达20mA, 拉电流为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)

注：寄存器P1M1和P1M0的地址分别为91H和92H.

举例： MOV P1M1, #00101000B

 MOV P1M0, #00110000B

 ;P1.5为开漏,P1.4为强推挽输出,P1.3为高阻输入,P1.2/P1.1/P1.0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90mA。即从MCU-VCC流入的电流不超过90mA, 从MCU-Gnd流出电流不超过90mA, 整体流入/流出电流都不能超过90mA.

4.2 管脚P5.4/RST的特别说明

P5. 4/RST即可作普通I/O使用, 还可作复位管脚。当用户ISP编程时将P5. 4/RST设置成普通I/O口用时, 其上电后为准双向口/弱上拉模式。

每次上电时, 单片机会自动判断上一次用户ISP编程时是将P5. 4/RST设置成普通I/O口还是复位脚。如果上一次用户ISP编程时是将P5. 4/RST设置成普通I/O口, 则单片机会将P5. 4/RST上电后的模式设置成准双向口/弱上拉。如果上一次用户ISP编程时是将P5. 4/RST设置成复位脚, 则上电后, P5. 4/RST仍为复位脚。

4.3 与I/O口有关的特殊功能寄存器及其在程序中的地址声明

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

P5 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P5	C8H	name	-	-	P5.5	P5.4	-	-	-	-

P5M1 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P5M1	C9H	name	-	-	P5M1.5	P5M1.4	-	-	-	-

P5M0 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P5M0	CAH	name	-	-	P5M0.5	P5M0.4	-	-	-	-

P3 register (可位寻址)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3	B0H	name	P3.7	P3.6	-	-	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	P3M1.7	P3M1.6	-	-	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	P3M0.7	P3M0.6	-	-	P3M0.3	P3M0.2	P3M0.1	P3M0.0

P1 register (可位寻址)

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

P1M1 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1M1	91H	name	-	-	P1M1.5	P1M1.4	P1M1.3	P1M1.2	P1M1.1	P1M1.0

P1M0 register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1M0	92H	name	-	-	P1M0.5	P1M0.4	P1M0.3	P1M0.2	P1M0.1	P1M0.0

汇编语言：

```
P5      EQU    0C8H          ; or P5      DATA  0C8H
P5M1    EQU    0C9H          ; or P5M1    DATA  0C9H
P5M0    EQU    0CAH
```

;以上为P5口新增功能寄存器的地址声明

```
P3M1    EQU    0B1H          ; or P3M1    DATA  0B1H
P3M0    EQU    0B2H
```

;以上为P3口新增功能寄存器的地址声明

```
P1M1    EQU    091H
P1M0    EQU    092H
```

;以上为P1口新增功能寄存器的地址声明

C语言：

```
sfr      P5      = 0xc8;
sfr      P5M1    = 0xc9;
sfr      P5M0    = 0xca;
```

/*以上为P5新增功能寄存器的C语言地址声明*/

```
sfr      P3M1    = 0xb1;
sfr      P3M0    = 0xb2;
```

/*以上为P3新增功能寄存器的C语言地址声明*/

```
sfr      P1M1    = 0x91;
sfr      P1M0    = 0x92;
```

/*以上为P1新增功能寄存器的C语言地址声明*/

4.4 STC15F104ESW系列单片机P1/P3/P5口的测试程序

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列P1/P3/P5举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

sfr P5	=	0xC8;	//6 bit Port5	P5.7	P5.6	P5.5	P5.4	P5.3	P5.2	P5.1	P5.0	xxx11,xxxx
sfr P5M0	=	0xC9;	//									xx00,xxxx
sfr P5M1	=	0xCA;	//									xx00,xxxx
			//	7	6	5	4	3	2	1	0	Reset Value

```
sbit P10 = P1^0;
sbit P11 = P1^1;
sbit P12 = P1^2;
sbit P13 = P1^3;
sbit P14 = P1^4;
sbit P15 = P1^5;
```

```
sbit P30 = P3^0;
sbit P31 = P3^1;
sbit P32 = P3^2;
sbit P33 = P3^3;
sbit P36 = P3^6;
sbit P37 = P3^7;
```

```
sbit P54 = P5^4;
sbit P55 = P5^5;
```

```
void delay(void);
```

```
void main(void)
{
    P10 = 0;
    delay();
}
```

```
P11    =    0;
delay();
P12    =    0;
delay();
P13    =    0;
delay();
P14    =    0;
delay();
P15    =    0;
delay();

P1      =    0xff;

P30     =    0;
delay();
P31     =    0;
delay();
P32     =    0;
delay();
P33     =    0;
delay();
P36     =    0;
delay();
P37     =    0;
delay();

P3      =    0xff;

P54     =    0;
delay();
P55     =    0;
delay();

P5      =    0xff;

while(1)
{
    P1      =    0x00;
    delay();
    P1      =    0xff;

    P3      =    0x00;
    delay();
    P3      =    0xff;

    P5      =    0x00;
    delay();
    P5      =    0xff;
}
```

$$\}$$

4.5 I/O口各种不同的工作模式结构框图

4.5.1 准双向口输出配置

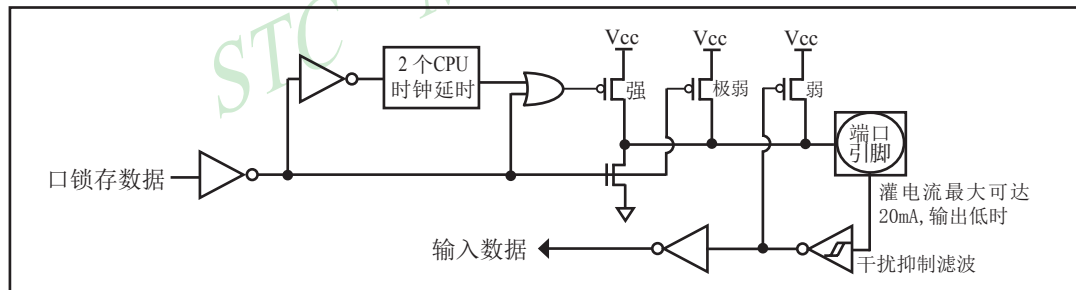
准双向口输出类型可用作输出和输入功能而不需重新配置口线输出状态。这是因为当口线输出为1时驱动能力很弱，允许外部装置将其拉低。当引脚输出为低时，它的驱动能力很强，可吸收相当大的电流。准双向口有3个上拉晶体管适应不同的需要。

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阈值电压以下。

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。



准双向输出

STC15F104ESW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。

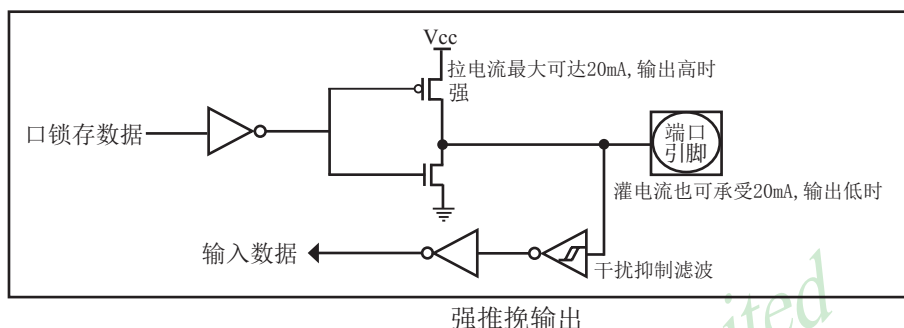
准双向口带有一个施密特触发输入以及一个干扰抑制电路。

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。

4.5.2 强推挽输出配置

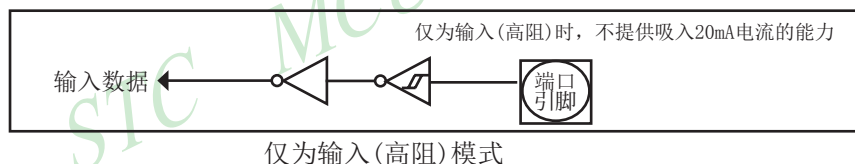
强推挽输出配置的下拉结构与开漏输出以及准双向口的下拉结构相同，但当锁存器为1时提供持续的强上拉。推挽模式一般用于需要更大驱动电流的情况。

强推挽引脚配置如下图所示。



4.5.3 仅为输入（高阻）配置

输入口配置如下图所示。

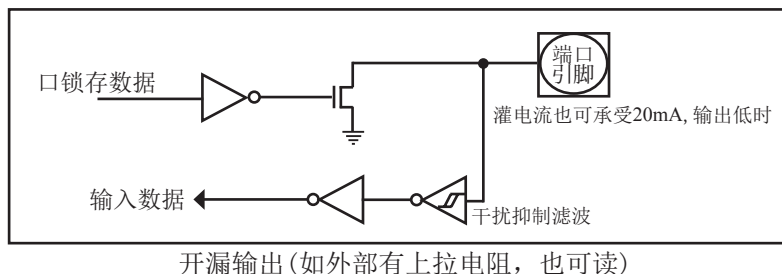


输入口带有一个施密特触发输入以及一个干扰抑制电路。

4.5.4 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。



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

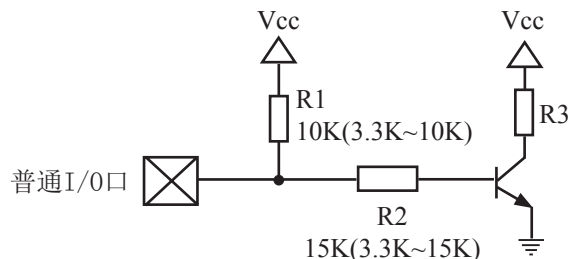
有些是外围接的是NPN三极管,没有加上拉电阻,其实基极串多大电阻,I/O口就应该上拉多大的电阻,或者将该I/O口设置为强推挽输出。

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。

4.6 一种典型三极管控制电路



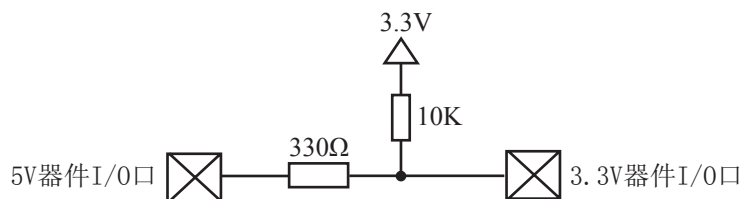
如果用弱上拉控制，建议加上拉电阻R1(3.3K~10K)，如果不加上拉电阻R1(3.3K~10K)，建议R2的值在15K以上，或用强推挽输出。

4.7 典型发光二极管控制电路



4.8 混合电压供电系统3V/5V器件I/O口互连

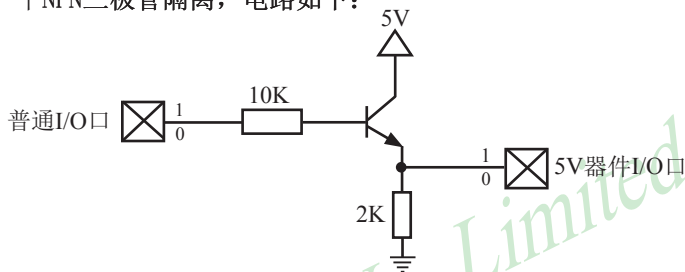
STC15F104ESW系列5V单片机连接3.3V器件时，为防止3.3V器件承受不了5V，可将相应的5V单片机I/O口先串一个330Ω的限流电阻到3.3V器件I/O口，程序初始化时将5V器件的I/O口设置成开漏配置，断开内部上拉电阻，相应的3.3V器件I/O口外部加10K上拉电阻到3.3V器件的Vcc，这样高电平是3.3V，低电平是0V，输入输出一切正常。



STC15L104ESW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。



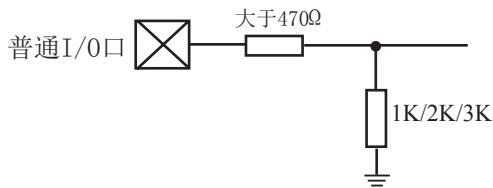
STC15L104ESW系列3V单片机连接5V器件时，为防止3V器件承受不了5V，如果相应的I/O口是输出，可用一个NPN三极管隔离，电路如下：



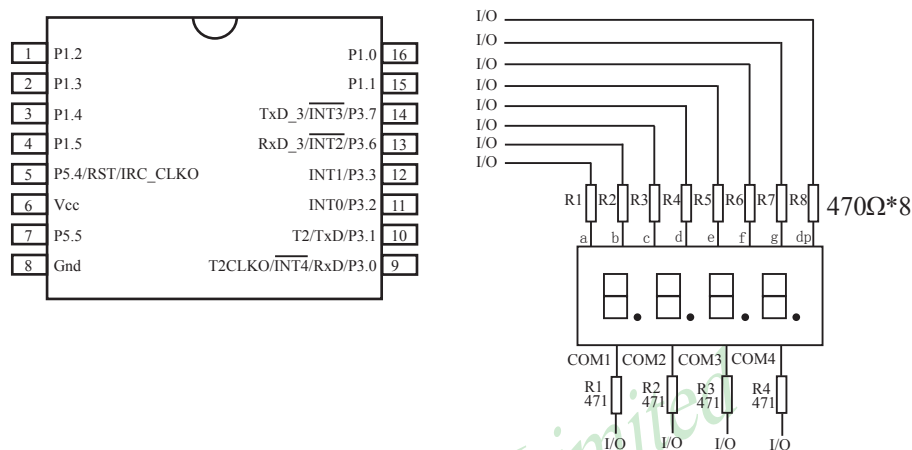
4.9 如何让I/O口上电复位时为低电平

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。

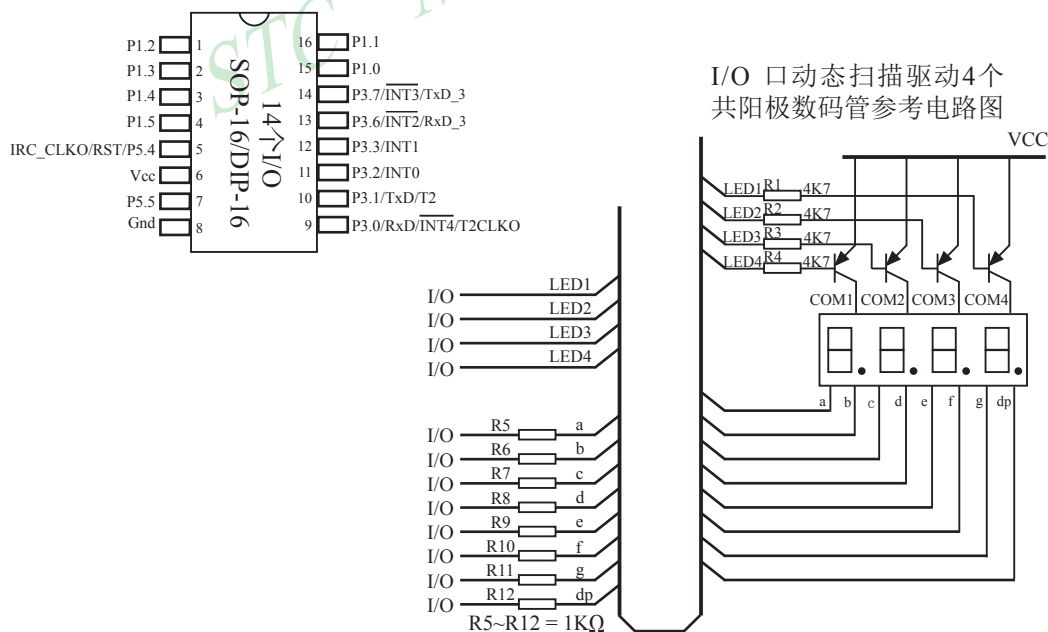


4.10 I/O口直接驱动LED数码管应用线路图



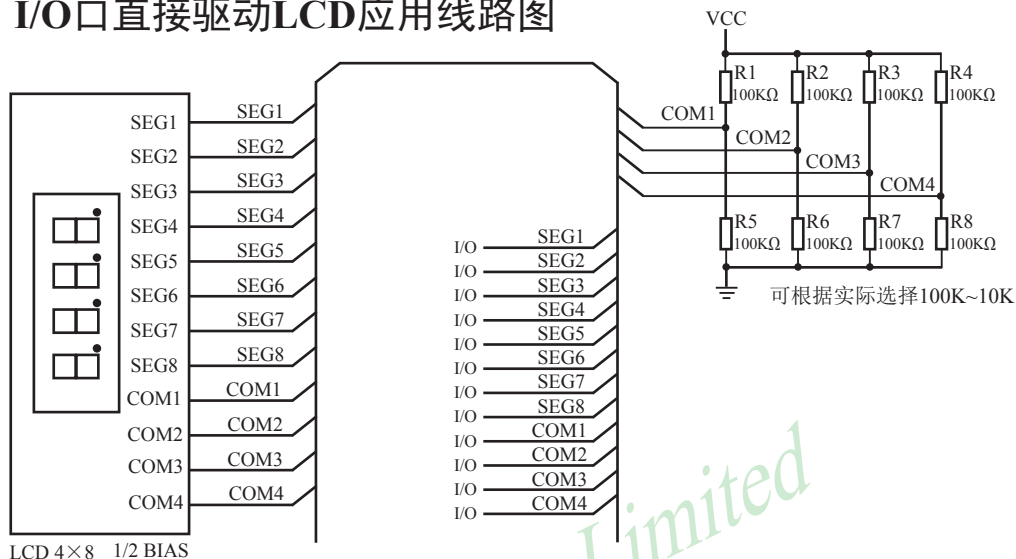
I/O 口动态扫描驱动4个共阴极数码管参考电路图

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

4.11 I/O口直接驱动LCD应用线路图



如何点亮相应的LCD像素：

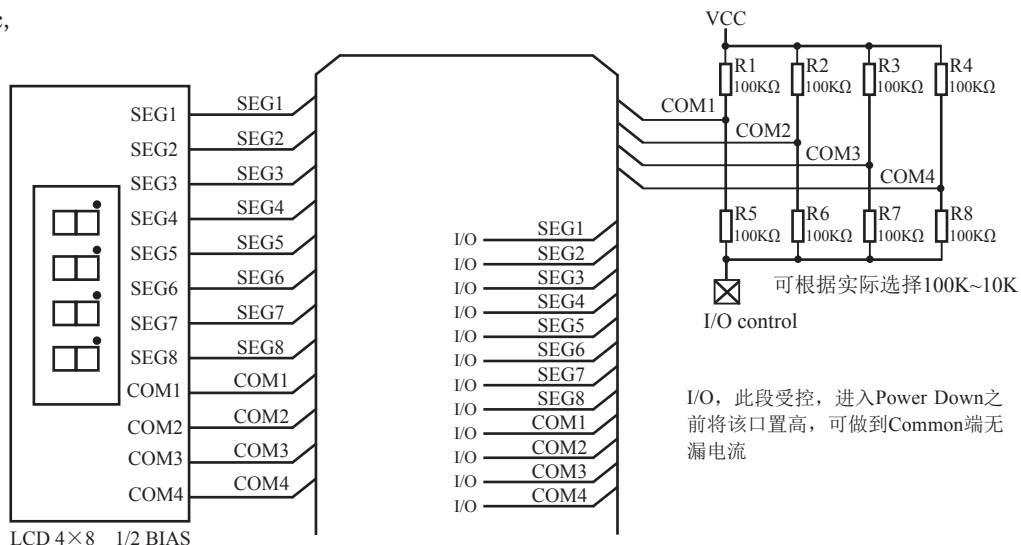
当相应的Common端和相应的Segment端压差大于 $1/2V_{cc}$ 时，相应的像素就显示，当压差小于 $1/2V_{cc}$ 时，相应的像素就不显示

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 $1/2V_{cc}$ ，



4.12 STC15系列单片机I/O口软件模拟I²C接口的测试程序

4.12.1 STC15系列单片机I/O口软件模式I²C接口的主机模式

```

;-----*/
;--- STC MCU Limited. -----*/
;--- STC 1T Series MCU Simulate I2C Master Demo -----*/
;--- Mobile: (86)13922829991 -----*/
;--- Fax: 86-755-82944243 -----*/
;--- Tel: 86-755-82948412 -----*/
;--- Web: www.STCMCU.com -----*/
; If you want to use the program or the program referenced in the */
; article, please specify in which data and procedures from STC */
;-----*/

```

```

SCL    BIT    P1.0
SDA    BIT    P1.1

```

```

;-----

```

```

        ORG    0000H

        MOV     TMOD, #20H      ;初始化串口为(9600,n,8,1)
        MOV     SCON, #5AH
        MOV     A,    #-5       ;-18432000/12/32/9600
        MOV     TH1,   A
        MOV     TL1,   A
        SETB    TR1

MAIN:
        CALL    UART_RXDATA     ;接收下一个串口数据
        MOV     R0,    A         ;临时保存到R0
                                   ;读取I2C设备IDATA 80H的数据
        CALL    I2C_START       ;开始读取
        MOV     A,    #01H
        CALL    I2C_TXBYTE      ;发送地址数据+读信号
        CALL    I2C_RXACK       ;接收ACK
        CALL    I2C_RXBYTE      ;接收数据
        SETB    C
        CALL    I2C_TXACK       ;发送NAK
        CALL    I2C_STOP        ;读取完成

        CALL    UART_TXDATA     ;将读到的数据发送到串口
                                   ;将R0的数据写入I2C设备IDATA 80H
        CALL    I2C_START       ;开始写
        MOV     A,    #00H
        CALL    I2C_TXBYTE      ;发送地址数据+写信号
        CALL    I2C_RXACK       ;接收ACK
        MOV     A,    R0

```

```
CALL    I2C_TXBYTE      ;写数据
CALL    I2C_RXACK       ;接收ACK
CALL    I2C_STOP        ;写完成

JMP     MAIN

;-----
;等待串口数据
;-----
UART_RXDATA:
    JNB    RI,          $      ;等待接收完成标志
    CLR    RI           ;清除标志
    MOV    A,          SBUF    ;保存数据
    RET

;-----
;发送串口数据
;-----
UART_TXDATA:
    JNB    TI,          $      ;等待上一个数据发送完成
    CLR    TI           ;清除标志
    MOV    SBUF, A        ;发送数据
    RET

;-----
;发送I2C起始信号
;-----
I2C_START:
    CLR    SDA          ;数据线下降沿
    CALL   I2C_DELAY     ;延时
    CLR    SCL          ;时钟->低
    CALL   I2C_DELAY     ;延时
    RET

;-----
;发送I2C停止信号
;-----
I2C_STOP:
    CLR    SDA
    SETB   SCL          ;时钟->高
    CALL   I2C_DELAY     ;延时
    SETB   SDA          ;数据线上上升沿
    CALL   I2C_DELAY     ;延时
    RET

;-----
;发送ACK/NAK信号
;-----
```

I2C_TXACK:

```

MOV    SDA,    C           ;送ACK数据
SETB   SCL           ;时钟->高
CALL   I2C_DELAY         ;延时
CLR    SCL           ;时钟->低
CALL   I2C_DELAY         ;延时
SETB   SDA           ;发送完成
RET

```

```

;-----
;接收ACK/NAK信号
;-----

```

I2C_RXACK:

```

SETB   SDA           ;准备读数据
SETB   SCL           ;时钟->高
CALL   I2C_DELAY         ;延时
MOV    C,    SDA       ;读取ACK信号
CLR    SCL           ;时钟->低
CALL   I2C_DELAY         ;延时
RET

```

```

;-----
;接收一字节数据
;-----

```

I2C_TXBYTE:

```

MOV    R7,    #8       ;8位计数
TXNEXT:
RLC    A           ;移出数据位
MOV    SDA,    C       ;数据送数据口
SETB   SCL           ;时钟->高
CALL   I2C_DELAY         ;延时
CLR    SCL           ;时钟->低
CALL   I2C_DELAY         ;延时
DJNZ   R7,    TXNEXT   ;送下一位
RET

```

```

;-----
;发送一字节数据
;-----

```

I2C_RXBYTE:

```

MOV    R7,    #8       ;8位计数
RXNEXT:
SETB   SCL           ;时钟->高
CALL   I2C_DELAY         ;延时
MOV    C,    SDA
RLC    A
CLR    SCL           ;时钟->低
CALL   I2C_DELAY         ;延时

```

```

        DJNZ    R7,      RXNEXT      ;收下一位
        RET

;-----

I2C_DELAY:
        PUSH    0                ;6
        MOV     R0,      #1        ;4
        DJNZ    R0,      $          ;2 6(200K) 1(400K) [18'432'000/400'000=46]
        POP     0                ;4
        RET                    ;3
;-----

        END
```

STC MCU Limited

4.12.2 STC15系列单片机I/O口软件模式I2C接口的从机模式

```

;-----*/
;*/ --- STC MCU Limited. -----*/
;*/ --- STC 1T Series MCU Simulate I2C Slave Demo -----*/
;*/ --- Mobile: (86)13922805190 -----*/
;*/ --- Fax: 86-755-82944243 -----*/
;*/ --- Tel: 86-755-82948412 -----*/
;*/ --- Web: www.STCMCU.com -----*/
;*/ If you want to use the program or the program referenced in the */
;*/ article, please specify in which data and procedures from STC */
;-----*/

```

```

SCL    BIT    P1.0
SDA    BIT    P1.1

```

```

;-----

```

```

    ORG    0

```

```

RESET:

```

```

    SETB   SCL
    SETB   SDA

```

```

    CALL   I2C_WAITSTART      ;等待起始信号
    CALL   I2C_RXBYTE         ;接收地址数据
    CLR    C
    CALL   I2C_TXACK          ;回应ACK
    SETB   C                  ;读/写 IDATA[80H - FFH]
    RRC    A                  ;读/写位->C
    MOV    R0,    A           ;地址送入R0
    JC     READDATA           ;C=1(读) C=0(写)

```

```

WRITEDATA:

```

```

    CALL   I2C_RXBYTE         ;接收数据
    MOV    @R0,    A          ;写入IDATA
    INC    R0                  ;地址+1
    CLR    C
    CALL   I2C_TXACK          ;回应ACK
    CALL   I2C_WAITSTOP       ;等待停止信号
    JMP     RESET

```

```

READDATA:

```

```

    MOV    A,    @R0
    INC    R0
    CALL   I2C_TXBYTE         ;发送IDATA数据
    CALL   I2C_RXACK          ;接收ACK
    CALL   I2C_WAITSTOP       ;等待停止信号
    JMP     RESET

```

```

;-----
;等待起始信号
;-----
I2C_WAITSTART:
    JNB     SCL,    $           ;等待时钟->高
    JB      SDA,    $           ;等待数据线下降沿
    JB      SCL,    $           ;等待时钟->低
    RET

```

```

;-----
;等待结束信号
;-----
I2C_WAITSTOP:
    JNB     SCL,    $           ;等待时钟->高
    JNB     SDA,    $           ;等待数据线上沿
    RET

```

```

;-----
;发送ACK/NAK信号
;-----
I2C_TXACK:
    MOV     SDA,    C           ;送ACK数据
    JNB     SCL,    $           ;等待时钟->高
    JB      SCL,    $           ;等待时钟->低
    SETB    SDA           ;发送完成
    RET

```

```

;-----
;接收ACK/NAK信号
;-----
I2C_RXACK:
    SETB    SDA           ;准备读数据
    JNB     SCL,    $           ;等待时钟->高
    MOV     C,      SDA        ;读取ACK信号
    JB      SCL,    $           ;等待时钟->低
    RET

```

```

;-----
;接收一字节数据
;-----
I2C_RXBYTE:
    MOV     R7,     #8           ;8位计数
RXNEXT:
    JNB     SCL,    $           ;等待时钟->高
    MOV     C,      SDA        ;读取数据口
    RLC     A           ;保存数据
    JB      SCL,    $           ;等待时钟->低
    DJNZ    R7,     RXNEXT      ;收下一位
    RET

```

```
;-----  
;发送一字节数据  
;-----  
I2C_TXBYTE:  
    MOV    R7,    #8            ;8位计数  
TXNEXT:  
    RLC    A            ;移出数据位  
    MOV    SDA,    C      ;数据送数据口  
    JNB    SCL,    $      ;等待时钟->高  
    JB     SCL,    $      ;等待时钟->低  
    DJNZ   R7,    TXNEXT  ;送下一位  
    RET  
  
;-----  
  
    END
```

STC MCU Limited

第5章 指令系统

5.1 寻址方式

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

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

5.1.1 立即寻址

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

如：MOV A, #70H

这条指令的功能是将立即数70H传送到累加器A中

5.1.2 直接寻址

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

如：ANL 70H, #48H

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

5.1.3 间接寻址

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

MOV A, @R1

把数据55H传送到累加器。

5.1.4 寄存器寻址

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

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

如：INC R0 ;(R0)+1 → R0

5.1.5 相对寻址

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

如：JC 80H ;C=1 跳转

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

5.1.6 变址寻址

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

如：MOVC A, @A+DPTR

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

5.1.7 位寻址

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

如：MOV C, 20H ; 片内位单元位操作型指令

5.2 完整指令集对照表(与传统8051对照)

——共111条指令，每条指令的详细执行时间

----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和MUL AB指令的执行速度大幅提升24倍

----共有22条指令，一个时钟就可以执行完成，平均速度快8~12倍

如果按功能分类，STC15F104ESW系列单片机指令系统可分为：

1. 算术操作类指令；
2. 逻辑操作类指令；
3. 数据传送类指令；
4. 布尔变量操作类指令；
5. 控制转移类指令。

按功能分类的指令系统表如下表所示。

指令执行速度效率提升总结(新15系列)：

指令系统共包括111条指令，其中：

执行速度快24倍的	共2条
执行速度快12倍的	共28条
执行速度快8倍的	共19条
执行速度快6倍的	共40条
执行速度快4.8倍的	共8条
执行速度快4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）(新15系列)：

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22条
2个时钟就可执行完成的指令	共37条
3个时钟就可执行完成的指令	共31条
4个时钟就可执行完成的指令	共12条
5个时钟就可执行完成的指令	共8条
6个时钟就可执行完成的指令	共1条

STC15系列将111条指令全部执行完一遍所需的时钟为283个时钟，而传统8051单片机将111条指令全部执行一遍要1944个时钟。可见与传统8051相比较，STC新15系列的指令执行速度大幅提升，平均速度快8~12倍。

算术操作类指令

助记符	功能说明	字节数	传统8051单片机所需时钟	STC15F104ESW系列单片机所需时钟 (采用STC-Y5 CPU内核指令集)	效率提升
ADD A, Rn	寄存器内容加到累加器	1	12	1	12倍
ADD A, direct	直接地址单元中的数据加到累加器	2	12	2	6倍
ADD A, @Ri	间接RAM中的数据加到累加器	1	12	2	6倍
ADD A, #data	立即数加到累加器	2	12	2	6倍
ADDC A, Rn	寄存器带进位加到累加器	1	12	1	12倍
ADDC A, direct	直接地址单元的内容带进位加到累加器	2	12	2	6倍
ADDC A, @Ri	间接RAM内容带进位加到累加器	1	12	2	6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	1	6倍
SUBB A, direct	累加器带借位减直接地址单元的内容	2	12	2	6倍
SUBB A, @Ri	累加器带借位减间接RAM中的内容	1	12	2	6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	1	12倍
INC Rn	寄存器加1	1	12	2	6倍
INC direct	直接地址单元加1	2	12	3	4倍
INC @Ri	间接RAM单元加1	1	12	3	4倍
DEC A	累加器减1	1	12	1	12倍
DEC Rn	寄存器减1	1	12	2	6倍
DEC direct	直接地址单元减1	2	12	3	4倍
DEC @Ri	间接RAM单元减1	1	12	3	4倍
INC DPTR	地址寄存器DPTR加1	1	24	1	24倍
MUL AB	A乘以B	1	48	2	24倍
DIV AB	A除以B	1	48	6	8倍
DA A	累加器十进制调整	1	12	3	4倍

逻辑操作类指令

助记符	功能说明	字节数	传统8051单片机所需时钟	STC15F2K6S02系列单片机所需时钟 (采用STC-Y5 CPU内核指令集)	效率提升
ANL A, Rn	累加器与寄存器相“与”	1	12	1	12倍
ANL A, direct	累加器与直接地址单元相“与”	2	12	2	6倍
ANL A, @Ri	累加器与间接RAM单元相“与”	1	12	2	6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	3	4倍
ANL direct, #data	直接地址单元与立即数相“与”	3	24	3	8倍
ORL A, Rn	累加器与寄存器相“或”	1	12	1	12倍
ORL A, direct	累加器与直接地址单元相“或”	2	12	2	6倍
ORL A, @Ri	累加器与间接RAM单元相“或”	1	12	2	6倍
ORL A, #data	累加器与立即数相“或”	2	12	2	6倍
ORL direct, A	直接地址单元与累加器相“或”	2	12	3	4倍
ORL direct, #data	直接地址单元与立即数相“或”	3	24	3	8倍
XRL A, Rn	累加器与寄存器相“异或”	1	12	1	12倍
XRL A, direct	累加器与直接地址单元相“异或”	2	12	2	6倍
XRL A, @Ri	累加器与间接RAM单元相“异或”	1	12	2	6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	3	4倍
XRL direct, #data	直接地址单元与立即数相“异或”	3	24	3	8倍
CLR A	累加器清“0”	1	12	1	12倍
CPL A	累加器求反	1	12	1	12倍
RL A	累加器循环左移	1	12	1	12倍
RLC A	累加器带进位位循环左移	1	12	1	12倍
RR A	累加器循环右移	1	12	1	12倍
RRC A	累加器带进位位循环右移	1	12	1	12倍
SWAP A	累加器内高低半字节交换	1	12	1	12倍

数据传送类指令

助记符	功能说明	字节数	传统8051单片机所需时钟	STC15F104ESW系列单片机所需时钟 (采用STC-Y5 CPU内核指令集)	效率提升
MOV A, Rn	寄存器内容送入累加器	1	12	1	12倍
MOV A, direct	直接地址单元中的数据送入累加器	2	12	2	6倍
MOV A, @Ri	间接RAM中的数据送入累加器	1	12	2	6倍
MOV A, #data	立即数送入累加器	2	12	2	6倍
MOV Rn, A	累加器内容送入寄存器	1	12	1	12倍
MOV Rn, direct	直接地址单元中的数据送入寄存器	2	24	3	8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	2	6倍
MOV direct, Rn	寄存器内容送入直接地址单元	2	24	2	12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	3	8倍
MOV direct, @Ri	间接RAM中的数据送入直接地址单元	2	24	3	8倍
MOV direct, #data	立即数送入直接地址单元	3	24	3	8倍
MOV @Ri, A	累加器内容送入间接RAM单元	1	12	2	6倍
MOV @Ri, direct	直接地址单元数据送入间接RAM单元	2	24	3	8倍
MOV @Ri, #data	立即数送入间接RAM单元	2	12	2	6倍
MOV DPTR, #data16	16位立即数送入数据指针	3	24	3	8倍
MOVC A, @A+DPTR	以DPTR为基地址变址寻址单元中的数据送入累加器	1	24	5	4.8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	将逻辑上在片外、物理上在片内的扩展RAM(8位地址)的内容送入累加器A中, 读操作	1	24	3	8倍
MOVX @Ri, A	将累加器A的内容送入逻辑上在片外、物理上在片内的扩展RAM(8位地址)中, 写操作	1	24	4	8倍
MOVX A, @DPTR	将逻辑上在片外、物理上在片内的扩展RAM(16位地址)的内容送入累加器A中, 读操作	1	24	2	12倍
MOVX @DPTR, A	将累加器A的内容送入逻辑上在片外、物理上在片内的扩展RAM(16位地址)中, 写操作	1	24	3	8倍
PUSH direct	直接地址单元中的数据压入堆栈	2	24	3	8倍
POP direct	栈底数据弹出送入直接地址单元	2	24	2	12倍
XCH A, Rn	寄存器与累加器交换	1	12	2	6倍
XCH A, direct	直接地址单元与累加器交换	2	12	3	4倍
XCH A, @Ri	间接RAM与累加器交换	1	12	3	4倍
XCHD A, @Ri	间接RAM的低半字节与累加器交换	1	12	3	4倍

布尔变量操作类指令

助记符	功能说明	字节数	传统8051单片机所需时钟	STC15F104ESW系列单片机所需时钟 (采用STC-Y5 CPU内核指令集)	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	3	4倍
SETB C	置1进位位	1	12	1	12倍
SETB bit	置1直接地址位	2	12	3	4倍
CPL C	进位位求反	1	12	1	12倍
CPL bit	直接地址位求反	2	12	3	4倍
ANL C, bit	进位位和直接地址位相“与”	2	24	2	12倍
ANL C, /bit	进位位和直接地址位的反码相“与”	2	24	2	12倍
ORL C, bit	进位位和直接地址位相“或”	2	24	2	12倍
ORL C, /bit	进位位和直接地址位的反码相“或”	2	24	2	12倍
MOV C, bit	直接地址位送入进位位	2	12	2	12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	5	4.8倍
JNB bit, rel	直接地址位为0则转移	3	24	5	4.8倍
JBC bit, rel	直接地址位为1则转移，该位清0	3	24	5	4.8倍

本次指令系统总结更新于2011-10-17日止

控制转移类指令

助记符	功能说明	字节数	传统8051单片机所需时钟	STC15F104ESW系列单片机所需时钟 (采用STC-Y5 CPU内核指令集)	效率提升
ACALL addr11	绝对（短）调用子程序	2	24	4	6倍
LCALL addr16	长调用子程序	3	24	4	6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 rel	相对转移	2	24	3	8倍
JMP @A+DPTR	相对于DPTR的间接转移	1	24	5	4.8倍
JZ rel	累加器为零转移	2	24	4	6倍
JNZ rel	累加器非零转移	2	24	4	6倍
CJNE A, direct, rel	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, rel	累加器与立即数比较，不相等则转移	3	24	4	6倍
CJNE Rn, #data, rel	寄存器与立即数比较，不相等则转移	3	24	4	6倍
CJNE @Ri, #data, rel	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, rel	寄存器减1，非零转移	2	24	4	6倍
DJNZ direct, rel	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

以上指令集均属于 STC-Y5 CPU内核指令集

5.3 传统8051单片机指令定义详解(中文&English)

5.3.1 传统8051单片机指令定义详解

ACALL addr 11

功能：绝对调用

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

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

ACALL SUBRTN

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

指令长度(字节)：2

执行周期：2

二进制编码：

a10	a9	a8	1	0	0	1	0
-----	----	----	---	---	---	---	---

a7	a6	a5	a4	a3	a2	a1	a0
----	----	----	----	----	----	----	----

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

操作：ACALL

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

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

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

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

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

$(PC_{10-0}) \leftarrow$ 页码地址

ADD A, <src-byte>

功能：加法

说明：ADD指令可用于完成把src-byte所表示的源操作数和累加器A的当前值相加。并将结果置于累加器A中。根据运算结果，若第7位有进位则置进位标志为1，否则清零；若第3位有进位则置辅助进位标志为1，否则清零。如果是无符号整数相加则进位置位，显示当前运算结果发生溢出。

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

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

举例：假设累加器A中的数据为0C3H(000011B)，R0的值为0AAH(10101010B)。执行如下指令：

ADD A, R0

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

ADD A, Rn

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：ADD
(A)←(A) + (Rn)

ADD A, direct

指令长度(字节)：2

执行周期：1

二进制编码：

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

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

direct address			
----------------	--	--	--

操作：ADD
(A)←(A) + (direct)

ADD A, @Ri

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：ADD
(A)←(A) + ((Ri))

ADD A, #data

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

immediate data

操作: ADD

 $(A) \leftarrow (A) + \#data$ **ADDC A, <src-byte>**

功能: 带进位的加法。

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

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

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

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

ADDC A,R0

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

ADDC A, Rn

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: ADDC

 $(A) \leftarrow (A) + (C) + (Rn)$ **ADDC A, direct**

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

direct address

操作: ADDC

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

ADDC A, @Ri

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: ADDC

 $(A) \leftarrow (A) + (C) + ((Ri))$ **ADDC A, #data**

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

immediate data			
----------------	--	--	--

操作: ADDC

 $(A) \leftarrow (A) + (C) + \#data$ **AJMP addr 11**

功能: 绝对跳转

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

举例: 假设标号JMPADR位于程序存储器的0123H, 指令

AJMP JMPADR

位于0345H, 执行完该指令后PC值变为0123H。

指令长度(字节): 2

执行周期: 2

二进制编码:

a10	a9	a8	0
-----	----	----	---

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

a7	a6	a5	a4
----	----	----	----

a3	a2	a1	a0
----	----	----	----

注意: 目的地址的A10-A8=a10-a8, A7-A0=a7-a0

操作: AJMP

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

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

功能： 对字节变量进行逻辑与运算

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

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

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

举例： 如果累加器的内容为0C3H(11000011B)，寄存器0的内容为55H(010101011B)，那么指令：

ANL A,R0

执行结果是累加器的内容变为41H(01000001H)。

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

ANL P1, #01110011B

将端口1的位7、位3和位2清零。

ANL A, Rn

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：ANL
 $(A) \leftarrow (A) \wedge (Rn)$

ANL A, direct

指令长度(字节)：2

执行周期：1

二进制编码：

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

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

direct address			
----------------	--	--	--

操作：ANL
 $(A) \leftarrow (A) \wedge (\text{direct})$

ANL A, @Ri

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：ANL
 $(A) \leftarrow (A) \wedge ((Ri))$

ANL A, #data

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

immediate data			
----------------	--	--	--

操作: ANL
 $(A) \leftarrow (A) \wedge \#data$

ANL direct, A

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

direct address			
----------------	--	--	--

操作: ANL
 $(direct) \leftarrow (direct) \wedge (A)$

ANL direct, #data

指令长度(字节): 3

执行周期: 2

二进制编码:

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

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

direct address			
----------------	--	--	--

immediate data			
----------------	--	--	--

操作: ANL
 $(direct) \leftarrow (direct) \wedge \#data$

ANL C, <src-bit>

功能: 对位变量进行逻辑与运算

说明: 如果src-bit表示的布尔变量为逻辑0, 清零进位标志位; 否则, 保持进位标志的当前状态不变。在汇编语言程序中, 操作数前面的“/”符号表示在计算时需要先对被寻址位取反, 然后才作为源操作数, 但源操作数本身不会改变。该指令在执行时不会影响其他各个标志位。

源操作数只能采取直接寻址方式。

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

MOV C, P1.0	;LOAD CARRY WITH INPUT PIN STATE
ANL C, ACC.7	;AND CARRY WITH ACCUM. BIT.7
ANL C, /OV	;AND WITH INVERSE OF OVERFLOW FLAG

ANL C, bit

指令长度(字节): 2

执行周期: 2

二进制编码:

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

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

bit address			
-------------	--	--	--

操作: ANL
 $(C) \leftarrow (C) \wedge (bit)$

ANL C, /bit

指令长度(字节)： 2

执行周期： 2

二进制编码：

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

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

bit address			
-------------	--	--	--

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

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

功能： 若两个操作数不相等则转移

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

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

举例： 设累加器A中值为34H，R7包含的数据为56H。如下指令序列：

```
CJNE    R7,#60H, NOT-EQ
;
;          ...          ; R7 = 60H.
NOT_EQ: JC      REQ_LOW  ; IF R7 < 60H.
;          ...          ; R7 > 60H.
```

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

假设端口1的数据也是34H，那么如下指令：

WAIT: CJNE A,P1,WAIT

清除进位标志并继续往下执行，因为此时累加器的值也为34H，即和P1口的数据相等。（如果P1端口的数据是其他的值，那么程序在此不停地循环，直到P1端口的数据变成34H为止。）

CJNE A, direct, rel

指令长度(字节)： 3

执行周期： 2

二进制编码：

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

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

direct address			
----------------	--	--	--

rel. address			
--------------	--	--	--

操作： $(PC) \leftarrow (PC) + 3$
IF $(A) < > (direct)$
THEN
 $(PC) \leftarrow (PC) + relative\ offset$
IF $(A) < (direct)$
THEN
 $(C) \leftarrow 1$
ELSE
 $(C) \leftarrow 0$

CJNE A, #data, rel

指令长度(字节): 3

执行周期: 2

二进制编码:

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

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

immediata	data
-----------	------

rel. address

操作: $(PC) \leftarrow (PC) + 3$
 IF $(A) <> (data)$
 THEN
 $(PC) \leftarrow (PC) + relative\ offset$
 IF $(A) < (data)$
 THEN
 $(C) \leftarrow 1$
 ELSE
 $(C) \leftarrow 0$

CJNE Rn, #data, rel

指令长度(字节): 3

执行周期: 2

二进制编码:

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

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

immediata	data
-----------	------

rel. address

操作: $(PC) \leftarrow (PC) + 3$
 IF $(Rn) <> (data)$
 THEN
 $(PC) \leftarrow (PC) + relative\ offset$
 IF $(Rn) < (data)$
 THEN
 $(C) \leftarrow 1$
 ELSE
 $(C) \leftarrow 0$

CJNE @Ri, #data, rel

指令长度(字节): 3

执行周期: 2

二进制编码:

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

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

immediate	data
-----------	------

rel. address

操作: $(PC) \leftarrow (PC) + 3$
 IF $((Ri)) <> (data)$
 THEN
 $(PC) \leftarrow (PC) + relative\ offset$
 IF $((Ri)) < (data)$
 THEN
 $(C) \leftarrow 1$
 ELSE
 $(C) \leftarrow 0$

CLR A

功能：清除累加器

说明：该指令用于将累加器A的所有位清零，不影响标志位。

举例：假设累加器A的内容为5CH(01011100B)，那么指令：

CLR A

执行后，累加器的值变为00H(00000000B)。

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：CLR
(A) ← 0

CLR bit

功能：清零指定的位

说明：将bit所代表的位清零，没有标志位会受到影响。CLR可用于进位标志C或者所有可直接寻址的位。

举例：假设端口1的数据为5DH(01011101B)，那么指令

CLR P1.2

执行后，P1端口被设置为59H(01011001B)。

CLR C

指令长度(字节)：1

执行周期：1

二进制编码：

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

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

操作：CLR
(C) ← 0

CLR bit

指令长度(字节)：2

执行周期：1

二进制编码：

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

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

bit address

操作：CLR
(bit) ← 0

CPL A

功能：累加器A求反

说明：将累加器A的每一位都取反，即原来为1的位变为0，原来为0的位变为1。该指令不影响标志位。

举例：设累加器A的内容为5CH(01011100B)，那么指令

CPL A

执行后，累加器的内容变成0A3H(10100011B)。

指令长度(字节)：1

执行周期：1

二进制编码：

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

操作：CPL

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

CPL bit

功能：将bit所表示的位求反

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

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

举例：设P1端口的数据为5BH(01011011B)，那么指令

CLR P1.1

CLR P1.2

执行完后，P1端口被设置为5BH(01011011B)。

CPL C

指令长度(字节)：1

执行周期：1

二进制编码：

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

操作：CPL

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

CPL bit

指令长度(字节)：2

执行周期：1

二进制编码：

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

bit address

操作：CPL

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

DA A

功能： 在加法运算之后，对累加器A进行十进制调整

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

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

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

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

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

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

ADDC A,R3

DA A

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

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

把BCD格式的变量加上01H或99H，可以实现加1或者减1。假设累加器的初始值为30H（表示十进制数30），指令序列

ADD A,#99H

DA A

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

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: DA

-contents of Accumulator are BCD

IF $[(A_{3-0}) > 9] \vee [(AC) = 1]$ THEN $(A_{3-0}) \leftarrow (A_{3-0}) + 6$

AND

IF $[(A_{7-4}) > 9] \vee [(C) = 1]$ THEN $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

DEC byte

功能: 把BYTE所代表的操作数减1

说明: BYTE所代表的变量被减去1。如果原来的值为00H, 那么减去1后, 变成0FFH。没有标志位会受到影响。该指令支持4种操作数寻址方式: 累加器寻址、寄存器寻址、直接寻址和寄存器间接寻址。

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

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

DEC @R0

DEC R0

DEC @R0

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

DEC A

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: DEC

 $(A) \leftarrow (A) - 1$

DEC Rn

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: DEC

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

DEC direct

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

direct address			
----------------	--	--	--

操作: DEC
(direct) ← (direct) - 1**DEC @Ri**

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: DEC
((Ri)) ← ((Ri)) - 1**DIV AB**

功能: 除法

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

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

举例: 假设累加器的值为251 (0FBH或11111011B), 寄存器B的值为18 (12H或00010010B)。则指令

DIV AB

执行后, 累加器的值变成13 (0DH或00001101B), 寄存器B的值变成17 (11H或0001000B), 正好符合 $251 = 13 \times 18 + 17$ 。进位和溢出标志都被清零。

指令长度(字节): 1

执行周期: 4

二进制编码:

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

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

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

DJNZ <byte>, <rel-addr>

功能： 减1，若非0则跳转

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

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

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

```
DJNZ 40H, LABEL_1
DJNZ 50H, LABEL_2
DJNZ 60H, LABEL_3
```

执行之后，程序将跳转到标号LABEL2处执行，且相应的3个RAM单元的内容变成00H、6FH和15H。之所以第1个跳转没被执行，是因为减1后其结果为0，不满足跳转条件。

使用DJNZ指令可以方便地在程序中实现指定次数的循环，此外用一条指令就可以在程序中实现中等长度的时间延迟（2~512个机器周期）。指令序列

```
MOV R2,#8
TOGGLE: CPL P1.7
        DJNZ R2, TOGGLE
```

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

DJNZ Rn,rel

指令长度(字节)： 2

执行周期： 2

二进制编码：

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

rel. address

操作： DJNZ
 $(PC) \leftarrow (PC) + 2$
 $(Rn) \leftarrow (Rn) - 1$
 IF $(Rn) > 0$ or $(Rn) < 0$
 THEN
 $(PC) \leftarrow (PC) + rel$

DJNZ direct, rel

指令长度(字节): 3

执行周期: 2

二进制编码:

操作: DJNZ

 $(PC) \leftarrow (PC) + 2$ $(direct) \leftarrow (direct) - 1$ IF $(direct) > 0$ or $(direct) < 0$

THEN

 $(PC) \leftarrow (PC) + rel$ **INC <byte>**

功能: 加1

说明: INC指令将<byte>所代表的数据加1。如果原来的值为FFH, 则加1后变为00H, 该指令步影响标志位。支持3种寻址模式: 寄存器寻址、直接寻址、寄存器间接寻址。

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

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

INC @R0

INC R0

INC @R0

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

INC A

指令长度(字节): 1

执行周期: 1

二进制编码:

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

操作: INC

 $(A) \leftarrow (A) + 1$ **INC Rn**

指令长度(字节): 1

执行周期: 1

二进制编码:

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

操作: INC

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

INC direct

指令长度(字节): 2

执行周期: 1

二进制编码:

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

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

direct address

操作: INC

 $(\text{direct}) \leftarrow (\text{direct}) + 1$ **INC @Ri**

指令长度(字节): 1

执行周期: 1

二进制编码:

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

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

操作: INC

 $((Ri)) \leftarrow ((Ri)) + 1$ **INC DPTR**

功能: 数据指针加1

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

该指令是唯一1条16位寄存器递增指令。

举例: 假设寄存器DPH和DPL的内容分别为12H和0FEH, 则指令序列

INC DPTR

INC DPTR

INC DPTR

执行完毕后, DPH和DPL变成13H和01H

指令长度(字节): 1

执行周期: 2

二进制编码:

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

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

操作: INC

 $(DPTR) \leftarrow (DPTR) + 1$

JB bit, rel

功能：若位数据为1则跳转

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

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

JB P1.2, LABEL1

JB ACC.2, LABEL2

将导致程序转到标号LABEL2处去执行

指令长度(字节)：3

执行周期：2

二进制编码：

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

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

bit address			
-------------	--	--	--

rel. address			
--------------	--	--	--

操作：JB

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

IF (bit) = 1

THEN

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

JBC bit, rel

功能：若位数据为1则跳转并将其清零

说明：如果bit代表的位数据为1，则将其清零并跳转到rel所指定的地址处去执行。如果bit代表的位数据为0，则继续执行下一条指令。跳转的目标地址按照如下方式计算：先增加PC的值，使其指向下一条指令的首字节地址，然后把rel所代表的有符号的相对偏移量（指令的第3个字节）加到PC上去，新的PC值即为目标地址，而且该操作不会影响标志位。

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

举例：假设累加器的内容为56H(01010110B)，则指令序列

JBC ACC.3, LABEL1

JBC ACC.2, LABEL2

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

指令长度(字节)：3

执行周期：2

二进制编码：

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

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

bit address			
-------------	--	--	--

rel. address			
--------------	--	--	--

操作：JBC

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

IF (bit) = 1

THEN

(bit) \leftarrow 0

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

JC rel

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

指令长度(字节)： 2

执行周期： 2

二进制编码：

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

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

JMP @A+DPTR

- 功能： 间接跳转。
- 说明： 把累加器A中的8位无符号数据和16位的数据指针的值相加，其和作为下一条将要执行的指令的地址，传送给程序计数器PC。执行16位的加法时，低字节DPL的进位会传到高字节DPH。累加器A和数据指针DPTR 的内容都不会发生变化。不影响任何标志位。
- 举例： 假设累加器A中的值是偶数（从0到6）。下面的指令序列将使得程序跳转到位于跳转表JMP\_TBL 的4条AJMP指令中的某一条去执行：

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

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

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

指令长度(字节)： 1

执行周期： 2

二进制编码：

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

操作： JMP  
(PC) ← (A) + (DPTR)

**JNB bit, rel**

- 功能：** 如果bit所代表的位不为1则跳转。
- 说明：** 如果bit所表示的位为0，则转移到rel所代表的地址去执行；否则，继续执行下一条指令。跳转的目标地址如此计算：先增加PC的值，使其指向下一条指令的首字节地址，然后把rel所代表的有符号的相对偏移量（指令的第3个字节）加到PC上去，新的PC值即为目标地址。该指令只是测试相应的位数据，但不会改变其数值，而且该操作不会影响标志位。
- 举例：** 假设端口1的输入数据为110010108，累加器的值为56H（01010110B）。则指令序列
- ```
JNB    P1.3, LABEL1
JNB    ACC.3, LABEL2
```
- 执行后将导致程序转到标号LABEL2处去执行。

指令长度(字节)： 3

执行周期： 2

二进制编码：

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

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

bit address

rel. address

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

JNC rel

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

指令长度(字节)： 2

执行周期： 2

二进制编码：

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

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

**JNZ rel**

**功能：** 如果累加器的内容非0则跳转

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

**举例：** 设累加器的初始值为00H，则指令序列

```
JNZ LABEL1
INC A
JNZ LAEEL2
```

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

指令长度(字节)： 2

执行周期： 2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

**操作：** JNZ  
 $(PC) \leftarrow (PC) + 2$   
 IF  $(A) \neq 0$   
 THEN  $(PC) \leftarrow (PC) + rel$

**JZ rel**

**功能：** 若累加器的内容为0则跳转

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

**举例：** 设累加器的初始值为01H，则指令序列

```
JZ LABEL1
DEC A
JZ LAEEL2
```

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

指令长度(字节)： 2

执行周期： 2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

**操作：** JZ  
 $(PC) \leftarrow (PC) + 2$   
 IF  $(A) = 0$   
 THEN  $(PC) \leftarrow (PC) + rel$

**LCALL addr16**

功能：长调用

说明：LCALL用于调用addr16所指地址处的子例程。首先将PC的值增加3，使得PC指向紧随LCALL的下一条指令的地址，然后把16位PC的低8位和高8位依次压入栈（低位字节在先），同时把栈指针加2。然后再把LCALL指令的第2字节和第3字节的数据分别装入PC的高位字节DPH和低位字节DPL，程序从新的PC所对应的地址处开始执行。因而子例程可以位于64KB程序存储空间的任何地址处。该操作不影响标志位。

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

LCALL SUBRTN

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

指令长度(字节)：3

执行周期：2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

addr15-addr8

addr7-addr0

操作：LCALL

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

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

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

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

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

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

**LJMP addr16**

功能：长跳转

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

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

LJMP JMPADR

执行完毕后，PC的当前值变为1234H。

指令长度(字节)：3

执行周期：2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

addr15-addr8

addr7-addr0

操作：LJMP

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



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

功能：传送字节变量

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

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

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

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

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

**MOV A,Rn**

指令长度(字节)：1

执行周期：1

二进制编码：

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

操作：MOV  
(A) ← (Rn)

**\*MOV A,direct**

指令长度(字节)：2

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作：MOV  
(A) ← (direct)

注意：MOV A, ACC是无效指令。

**MOV A,@Ri**

指令长度(字节)：1

执行周期：1

二进制编码：

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

操作：MOV  
(A) ← ((Ri))

**MOV A,#data**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

操作: MOV  
(A)←#data**MOV Rn,A**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

操作: MOV  
(Rn)←(A)**MOV Rn,direct**

指令长度(字节): 2

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|              |
|--------------|
| direct addr. |
|--------------|

操作: MOV  
(Rn)←(direct)**MOV Rn,#data**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

操作: MOV  
(Rn)←#data**MOV direct,A**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: MOV  
(direct)←(A)**MOV direct,Rn**

指令长度(字节): 2

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: MOV  
(direct)←(Rn)

**MOV direct, direct**

指令长度(字节): 3

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                 |  |  |  |
|-----------------|--|--|--|
| dir.addr. (src) |  |  |  |
|-----------------|--|--|--|

操作: MOV  
(direct)←(direct)**MOV direct, @Ri**

指令长度(字节): 2

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

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

|              |  |  |  |
|--------------|--|--|--|
| direct addr. |  |  |  |
|--------------|--|--|--|

操作: MOV  
(direct)←((Ri))**MOV direct,#data**

指令长度(字节): 3

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

操作: MOV  
(direct) ← #data**MOV @Ri, A**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

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

操作: MOV  
((Ri)) ← (A)**MOV @Ri, direct**

指令长度(字节): 2

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

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

|              |  |  |  |
|--------------|--|--|--|
| direct addr. |  |  |  |
|--------------|--|--|--|

操作: MOV  
((Ri)) ← (direct)**MOV @Ri, #data**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

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

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

操作: MOV  
((Ri)) ← #data

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

功能： 传送位变量

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

举例： 假设进位标志C的初值为1，端口P2中的数据是11000101B，端口1的数据被设置为35H(00110101B)。则指令序列

```
MOV P1.3, C
MOV C, P3.3
MOV P1.2, C
```

执行后，进位标志被清零，端口1的数据变为39H（00111001B）。

**MOV C,bit**

指令长度(字节)： 2

执行周期： 1

二进制编码： 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

操作： MOV  
(C) ← (bit)

**MOV bit,C**

指令长度(字节)： 2

执行周期： 2

二进制编码： 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

操作： MOV  
(bit)← (C)

**MOV DPTR , #data 16**

功能： 将16位的常数存放到数据指针

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

该指令是唯一一条能一次性移动16位数据的指令。

举例： 指令：

```
MOV DPTR, #1234H
```

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

指令长度(字节)： 3

执行周期： 2

二进制编码： 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|                     |
|---------------------|
| immediate data 15-8 |
|---------------------|

操作： MOV  
(DPTR) ← #data<sub>15-0</sub>  
DPH DPL ← #data<sub>15-8</sub> #data<sub>7-0</sub>

**MOVC A, @A+ <base-reg>**

**功能：** 把程序存储器中的代码字节数据（常数数据）转送至累加器A

**说明：** MOVC指令将程序存储器中的代码字节或常数字节传送到累加器A。被传送的数据字节的地址是由累加器中的无符号8位数据和16位基址寄存器（DPTR或PC）的数值相加产生的。如果以PC为基址寄存器，则在累加器内容加到PC之前，PC需要先增加到指向紧邻MOVC之后的语句的地址；如果是以DPTR为基址寄存器，则没有此问题。在执行16位的加法时，低8位产生的进位会传递给高8位。本指令不影响标志位。

**举例：** 假设累加器A的值处于0~4之间，如下子例程将累加器A中的值转换为用DB伪指令（定义字节）定义的4个值之一。

```
REL-PC: INC A
 MOVC A, @A+PC
 RET
 DB 66H
 DB 77H
 DB 88H
 DB 99H
```

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

**MOVC A, @A+DPTR**

指令长度(字节)： 1

执行周期： 2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作： MOVC  
(A) ← ((A)+(DPTR))

**MOVC A, @A+PC**

指令长度(字节)： 1

执行周期： 2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作： MOVC  
(PC) ← (PC)+1  
(A) ← ((A)+(PC))

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

功能：外部传送

说明：MOVX指令用于在累加器和外部数据存储器之间传递数据。因此在传送指令MOV后附加了X。MOVX又分为两种类型，它们之间的区别在于访问外部数据RAM的间接地址是8位的还是16位的。

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

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

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

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

```
MOVX A, @R1
MOVX @R0, A
```

将数据56H复制到累加器A以及外部RAM的12H单元中。

**MOVX A, @Ri**

指令长度(字节)：1

执行周期：2

二进制编码：

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

操作：MOVX  
(A) ← ((Ri))

**MOVX A, @DPTR**

指令长度(字节)：1

执行周期：2

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

操作：MOVX  
(A) ← ((DPTR))

**MOVX @Ri, A**

指令长度(字节): 1

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

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

操作: MOVX  
((Ri))←(A)**MOVX @DPTR, A**

指令长度(字节): 1

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

操作: MOVX  
(DPTR)←(A)**MUL AB**

功能: 乘法

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

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

MUL AB

求得乘积12 800(3200H), 所以寄存器B的值变成32H(00110010B), 累加器被清零, 溢出标志被置位, 进位标志被清零。

指令长度(字节): 1

执行周期: 4

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

操作: MUL  
(A)<sub>7-0</sub>←(A)×(B)  
(B)<sub>15-8</sub>

## NOP

**功能：** 空操作

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

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

```
CLR P2.7
NOP
NOP
NOP
NOP
SETB P2.7
```

**指令长度(字节)：** 1

**执行周期：** 1

**二进制编码：**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**操作：** NOP  
(PC) ← (PC)+1

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

**功能：** 两个字节变量的逻辑或运算

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

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

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

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

```
ORL A, R0
```

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

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

之后，把P1口的第5、4、1位置1。



**ORL A,Rn**

指令长度(字节): 1

执行周期: 1

二进制编码: 

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

操作: ORL  
 $(A) \leftarrow (A) \vee (Rn)$ **ORL A,direct**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: ORL  
 $(A) \leftarrow (A) \vee (\text{direct})$ **ORL A,@Ri**

指令长度(字节): 1

执行周期: 1

二进制编码: 

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

操作: ORL  
 $(A) \leftarrow (A) \vee ((Ri))$ **ORL A,#data**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

操作: ORL  
 $(A) \leftarrow (A) \vee \#data$ **ORL direct, A**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: ORL  
 $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$ **ORL direct, #data**

指令长度(字节): 3

执行周期: 2

二进制编码: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

|                |
|----------------|
| immediate data |
|----------------|

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

**ORL C, <src-bit>**

功能： 位变量的逻辑或运算

说明： 如果<src-bit>所表示的位变量为1，则置位进位标志；否则，保持进位标志的当前状态不变。在汇编语言中，位于源操作数之前的“/”表示将源操作数取反后使用，但源操作数本身不发生变化。在执行本指令时，不影响其他标志位。

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

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

**ORL C, bit**

指令长度(字节)： 2

执行周期： 2

二进制编码： 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

操作： ORL  
 $(C) \leftarrow (C) \vee (\text{bit})$

**ORL C, /bit**

指令长度(字节)： 2

执行周期： 2

二进制编码： 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

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

**POP direct**

功能： 出栈

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

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

```
POP DPH
POP DPL
之后，栈指针的值变成30H，数据指针变为0123H。此时指令
POP SP
将把栈指针变为20H。
```

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

指令长度(字节)： 2

执行周期： 2

二进制编码： 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作： POP  
 $(\text{direct}) \leftarrow ((\text{SP}))$   
 $(\text{SP}) \leftarrow (\text{SP}) - 1$

**PUSH direct**

- 功能：压栈
- 说明：栈指针首先加1，然后将direct所表示的变量内容复制到由栈指针指定的内部RAM存储单元中去。该操作不影响标志位。
- 举例：设在进入中断服务程序时栈指针的值为09H，数据指针DPTR的值为0123H。则执行如下指令序列
- PUSH DPL  
PUSH DPH
- 之后，栈指针变为0BH，并把数据23H和01H分别存入内部RAM的0AH和0BH存储单元之中。

指令长度(字节)： 2  
执行周期： 2  
二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作： PUSH  
(SP) ← (SP) + 1  
((SP)) ← (direct)

**RET**

- 功能：从子例程返回
- 说明：执行RET指令时，首先将PC值的高位字节和低位字节从栈中弹出，栈指针减2。然后，程序从形成的PC值所对应的地址处开始执行，一般情况下，该指令和ACALL或LCALL配合使用。改指令的执行不影响标志位。
- 举例：设栈指针的初值为0BH，内部RAM的0AH和0BH存储单元中的数据分别为23H和01H。则指令：
- RET
- 执行后，栈指针变为09H。程序将从0123H地址处继续执行。

指令长度(字节)： 1  
执行周期： 2  
二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

操作： RET  
(PC<sub>15-8</sub>) ← ((SP))  
(SP) ← (SP) -1  
(PC<sub>7-0</sub>) ← ((SP))  
(SP) ← (SP) -1

**RETI**

**功能：** 中断返回

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

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

RETI

执行完毕后，栈指针变成09H，中断返回后程序继续从0123H地址开始执行。

**指令长度(字节)：** 1

**执行周期：** 2

**二进制编码：**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

**操作：** RETI

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

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

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

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

**RL A**

**功能：** 将累加器A中的数据位循环左移

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

**举例：** 设累加器的内容为0C5H（11000101B），则指令

RL A

执行后，累加器的内容变成8BH（10001011B），且标志位不受影响。

**指令长度(字节)：** 1

**执行周期：** 1

**二进制编码：**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

**操作：** RL

$(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$

$(A_0) \leftarrow (A_7)$

## RLC A

**功能：**带进位循环左移

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

**举例：**假设累加器A的值为0C5H(11000101B)，则指令

RLC A

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

指令长度(字节)：1

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作：RLC

$(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$

$(A_0) \leftarrow (C)$

$(C) \leftarrow (A_7)$

## RR A

**功能：**将累加器的数据位循环右移

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

**举例：**设累加器的内容为0C5H（11000101B），则指令

RR A

执行后累加器的内容变成0E2H（11100010B），标志位不受影响。

指令长度(字节)：1

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作：RR

$(A_n) \leftarrow (A_{n+1}) \quad n = 0-6$

$(A_7) \leftarrow (A_0)$

RRC A

功能：带进位循环右移

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

举例：假设累加器的值为0C5H(11000101B)，进位标志为0，则指令

RRC A

执行后，将把累加器的数据变为62H(01100010B)，进位标志被置位。

指令长度(字节)：1

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作：RRC

$(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$

$(A_7) \leftarrow (C)$

$(C) \leftarrow (A_0)$

SETB <bit>

功能：置位

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

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

SETB C

SETB P1.0

执行后，进位标志变为1，端口1的输出状态变成35H(00110101B)。

SETB C

指令长度(字节)：1

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

操作：SETB

$(C) \leftarrow 1$

SETB bit

指令长度(字节)：2

执行周期：1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

操作：SETB

$(bit) \leftarrow 1$

## SJMP rel

功能: 短跳转

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

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

SJMP RELADR

汇编后位于0100H。当执行完该指令后, PC值变成0123H。

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

指令长度(字节): 2

执行周期: 2

二进制编码: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

操作: SJMP

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

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

## SUBB A, <src-byte>

功能: 带借位的减法

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

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

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

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

SUBB A, R2

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

注意: 0C9H减去54H应该是75H, 但在上面的计算中, 由于在SUBB指令执行前, 进位标志C已经被置位, 因而最终结果还需要减去进位标志, 得到74H。因此, 如果在进行单精度或者多精度减法运算前, 进位标志C的状态未知, 那么应改采用CLR C指令把进位标志C清零。

**SUBB A, Rn**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

操作: SUBB

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

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

操作: SUBB

 $(A) \leftarrow (A) - (C) - (\text{direct})$ **SUBB A, @Ri**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

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

操作: SUBB

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

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

操作: SUBB

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

功能: 交换累加器的高低半字节

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

举例: 设累加器的内容为0C5H(11000101B), 则指令

SWAP A

执行后, 累加器的内容变成5CH(01011100B)。

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

操作: SWAP

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



**XCH A, <byte>**

**功能：** 交换累加器和字节变量的内容

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

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

XCH A, @R0

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

**XCH A, Rn**

指令长度(字节)： 1

执行周期： 1

二进制编码：

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

操作： XCH  
(A)  $\longleftrightarrow$  (Rn)

**XCH A, direct**

指令长度(字节)： 2

执行周期： 1

二进制编码：

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作： XCH  
(A)  $\longleftrightarrow$  (direct)

**XCH A, @Ri**

指令长度(字节)： 1

执行周期： 1

二进制编码：

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

操作： XCH  
(A)  $\longleftrightarrow$  ((Ri))

**XCHD    A, @Ri**

- 功能：** 交换累加器和@Ri对应单元中的数据的低4位
- 说明：** XCHD指令将累加器内容的低半字节（位0~3，一般是十六进制数或BCD码）和间接寻址的内部RAM单元的数据进行交换，各自的高半字（位7~4）节不受影响。另外，该指令不影响标志位。
- 举例：** 设R0保存了地址20H，累加器的内容为36H (00110110B)。内部RAM的20H单元存储的数据为75H (011110101B)。则指令：
- XCHD        A, @R0
- 执行后，内部RAM 20H单元的内容变成76H (01110110B)，累加器的内容变为35H(00110101B)。

指令长度(字节)： 1

执行周期： 1

二进制编码：

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

操作： XCHD  
          (A<sub>3-0</sub>)  $\longleftrightarrow$  (Ri<sub>3-0</sub>)

**XRL    <dest-byte>, <src-byte>**

- 功能：** 字节变量的逻辑异或
- 说明：** XRL指令将<dest-byte>和<src-byte>所代表的字节变量逐位进行逻辑异或运算，结果保存在<dest-byte>所代表的字节变量里。该指令不影响标志位。
- 两个操作数组合起来共支持6种寻址方式：当目的操作数为累加器时，源操作数可以采用寄存器寻址、直接寻址、寄存器间接寻址和立即数寻址；当目的操作数是可直接寻址的数据时，源操作数可以是累加器或者立即数。
- 注意：如果该指令被用来修改输出引脚上的状态，那么dest-byte所代表的数据就是从端口输出数据锁存器中获取的数据，而不是从引脚上读取的数据。
- 举例：**        如果累加器和寄存器0的内容分别为0C3H (11000011B)和0AAH(10101010B)，则指令：
- XRL        A, R0
- 执行后，累加器的内容变成69H (01101001B)。
- 当目的操作数是可直接寻址字节数据时，该指令可把任何RAM单元或者寄存器中的各个位取反。具体哪些位会被取反，在运行过程当中确定。指令：
- XRL        P1, #00110001B
- 执行后，P1口的位5、4、0被取反。

**XRL A, Rn**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

操作: XRL

 $(A) \leftarrow (A) \vee (Rn)$ **XRL A, direct**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: XRL

 $(A) \leftarrow (A) \vee (\text{direct})$ **XRL A, @Ri**

指令长度(字节): 1

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

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

操作: XRL

 $(A) \leftarrow (A) \vee ((Ri))$ **XRL A, #data**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

操作: XRL

 $(A) \leftarrow (A) \vee \#data$ **XRL direct, A**

指令长度(字节): 2

执行周期: 1

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

操作: XRL

 $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$ **XRL direct, #dataw**

指令长度(字节): 3

执行周期: 2

二进制编码: 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

|                |
|----------------|
| immediate data |
|----------------|

操作: XRL

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

### 5.3.2 Instruction Definitions of Traditional 8051 MCU

#### ACALL addr 11

**Function:** Absolute Call

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

**Example:** Initially SP equals 07H. The label “SUBRTN” is at program memory location 0345H. After executing the instruction,  
ACALL SUBRTN  
at location 0123H, SP will contain 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC will contain 0345H.

**Bytes:** 2

**Cycles:** 2

**Encoding:**

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

**Operation:**

ACALL

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

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

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

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

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

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

#### ADD A,<src-byte>

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b>    | Add                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Description:</b> | ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.<br><br>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.<br><br>Four source operand addressing modes are allowed: register, direct register-indirect, or immediate. |
| <b>Example:</b>     | The Accumulator holds 0C3H(11000011B) and register 0 holds 0AAH (10101010B). The instruction,<br>ADD A,R0<br>will leave 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

**ADD A,Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** ADD  
 $(A) \leftarrow (A) + (Rn)$ **ADD A,direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** ADD  
 $(A) \leftarrow (A) + (\text{direct})$ **ADD A,@Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

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

**Operation:** ADD  
 $(A) \leftarrow (A) + ((Ri))$ **ADD A,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

**Operation:** ADD  
 $(A) \leftarrow (A) + \#data$ **ADDC A,<src-byte>****Function:** Add with Carry**Description:** ADDC simultaneously adds the byte variable indicated, the Carry flag and the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

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

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

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

**ADDC A,Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** ADDC  
 $(A) \leftarrow (A) + (C) + (Rn)$ **ADDC A,direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** ADDC  
 $(A) \leftarrow (A) + (C) + (\text{direct})$ **ADDC A,@Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

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

**Operation:** ADDC  
 $(A) \leftarrow (A) + (C) + ((Ri))$ **ADDC A,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

**Operation:** ADDC  
 $(A) \leftarrow (A) + (C) + \#data$ **AJMP addr 11****Function:** Absolute Jump**Description:** AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2K block of program memory as the first byte of the instruction following AJMP.**Example:** The label “JMPADR” is at program memory location 0123H. The instruction, AJMP JMPADR is at location 0345H and will load the PC with 0123H.**Bytes:** 2**Cycles:** 2**Encoding:**

|     |    |    |   |
|-----|----|----|---|
| a10 | a9 | a8 | 0 |
|-----|----|----|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
|---|---|---|---|

|    |    |    |    |
|----|----|----|----|
| a7 | a6 | a5 | a4 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| a3 | a2 | a1 | a0 |
|----|----|----|----|

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

**ANL <dest-byte> , <src-byte>****Function:** Logical-AND for byte variables**Description:** ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

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

*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  
 $(A) \leftarrow (A) \wedge (Rn)$ **ANL A,direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** ANL  
 $(A) \leftarrow (A) \wedge (\text{direct})$ **ANL A,@Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

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

**Operation:** ANL  
 $(A) \leftarrow (A) \wedge ((Ri))$

**ANL A,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

**Operation:** ANL  
(A)←(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 affected*. No other flgs are affected.

Only direct addressing is allowed for the source operand.

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

```

MOV C, P1.0 ;LOAD CARRY WITH INPUT PIN STATE
ANL C, ACC.7 ;AND CARRY WITH ACCUM. BIT.7
ANL C, /OV ;AND WITH INVERSE OF OVERFLOW FLAG

```

**ANL C,bit****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** ANL  
(C) ← (C) ∧ (bit)



**ANL C, /bit****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** ANL  
 $(C) \leftarrow (C) \wedge (\text{bit})$ **CJNE <dest-byte>, <src-byte>, rel****Function:** Compare and Jump if Not Equal**Description:** CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

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

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

|         |      |                  |                |
|---------|------|------------------|----------------|
|         | CJNE | R7, #60H, NOT-EQ |                |
| ;       | ...  | ...              | ; R7 = 60H.    |
| NOT_EQ: | JC   | REQ LOW          | ; IF R7 < 60H. |
| ;       | ...  | ...              | ; R7 > 60H.    |

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

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

WAIT: CJNE A, P1, WAIT

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

**CJNE A, direct, rel****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

|              |
|--------------|
| rel. address |
|--------------|

**Operation:**  $(PC) \leftarrow (PC) + 3$   
IF  $(A) <> (direct)$   
THEN  
 $(PC) \leftarrow (PC) + \text{relative offset}$   
IF  $(A) < (direct)$   
THEN  
 $(C) \leftarrow 1$   
ELSE  
 $(C) \leftarrow 0$

**CJNE A,#data,rel****Bytes:** 3**Cycles:** 2

**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediata data |  |  |  |
|----------------|--|--|--|

|              |  |  |  |
|--------------|--|--|--|
| rel. address |  |  |  |
|--------------|--|--|--|

**Operation:**  $(PC) \leftarrow (PC) + 3$   
 IF  $(A) <> (data)$   
 THEN  
      $(PC) \leftarrow (PC) + \text{relative offset}$   
 IF  $(A) < (data)$   
 THEN  
      $(C) \leftarrow 1$   
 ELSE  
      $(C) \leftarrow 0$

**CJNE Rn,#data,rel****Bytes:** 3**Cycles:** 2

**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediata data |  |  |  |
|----------------|--|--|--|

|              |  |  |  |
|--------------|--|--|--|
| rel. address |  |  |  |
|--------------|--|--|--|

**Operation:**  $(PC) \leftarrow (PC) + 3$   
 IF  $(Rn) <> (data)$   
 THEN  
      $(PC) \leftarrow (PC) + \text{relative offset}$   
 IF  $(Rn) < (data)$   
 THEN  
      $(C) \leftarrow 1$   
 ELSE  
      $(C) \leftarrow 0$

**CJNE @Ri,#data,rel****Bytes:** 3**Cycles:** 2

**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

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

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

|              |  |  |  |
|--------------|--|--|--|
| rel. address |  |  |  |
|--------------|--|--|--|

**Operation:**  $(PC) \leftarrow (PC) + 3$   
 IF  $((Ri)) <> (data)$   
 THEN  
      $(PC) \leftarrow (PC) + \text{relative offset}$   
 IF  $((Ri)) < (data)$   
 THEN  
      $(C) \leftarrow 1$   
 ELSE  
      $(C) \leftarrow 0$

**CLR A****Function:** Clear Accumulator**Description:** The Accumulator is cleared (all bits set on zero). No flags are affected.

**Example:** The Accumulator contains 5CH (01011100B). The instruction,  
CLR A  
will leave the Accumulator set to 00H (00000000B).

**Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

**Operation:** CLR  
(A) ← 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) ←  $\overline{(A)}$ **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 (0101101B). 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) ←  $\overline{(C)}$ **CPL bit****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** CPL  
(bit) ←  $\overline{(\text{bit})}$

**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  $[(A_{3-0}) > 9] \vee [(AC) = 1]$   
   THEN  $(A_{3-0}) \leftarrow (A_{3-0}) + 6$   
       AND  
 IF  $[(A_{7-4}) > 9] \vee [(C) = 1]$   
   THEN  $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

**DEC byte****Function:** Decrement**Description:** The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH.

No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

*Note:* When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.**Example:** Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence,

DEC @R0

DEC R0

DEC @R0

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

**DEC A****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

**Operation:** DEC  
 $(A) \leftarrow (A) - 1$ **DEC Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

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

**DEC direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** DEC  
(direct) $\leftarrow$ -(direct) - 1**DEC @Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
|---|---|---|---|

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

**Operation:** DEC  
((Ri)) $\leftarrow$ ((Ri)) - 1**DIV AB****Function:** Divide**Description:** DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.**Exception:** if B had originally contained 00H, the values returned in the Accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.**Example:** The Accumulator contains 251(OFBH or 11111011B) and B contains 18(12H or 00010010B). The instruction,

DIV AB

will leave 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010010B) in B, since  $251 = (13 \times 18) + 17$ . Carry and OV will both be cleared.**Bytes:** 1**Cycles:** 4**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

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

**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 affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

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

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

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

```

DJNZ 40H, LABEL_1
DJNZ 50H, LABEL_2
DJNZ 60H, LABEL_3

```

will cause a jump to the instruction at label LABEL\_2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was not taken because the result was zero.

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

```

MOV R2, #8
TOGGLE: CPL P1.7
 DJNZ R2, TOGGLE

```

will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

**DJNZ Rn,rel****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|              |  |  |  |
|--------------|--|--|--|
| rel. address |  |  |  |
|--------------|--|--|--|

**Operation:** DJNZ  
 $(PC) \leftarrow (PC) + 2$   
 $(Rn) \leftarrow (Rn) - 1$   
 IF  $(Rn) > 0$  or  $(Rn) < 0$   
 THEN  
 $(PC) \leftarrow (PC) + rel$

**DJNZ direct, rel****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

|              |  |  |  |
|--------------|--|--|--|
| rel. address |  |  |  |
|--------------|--|--|--|



**Operation:** DJNZ  
 $(PC) \leftarrow (PC) + 2$   
 $(direct) \leftarrow (direct) - 1$   
 IF  $(direct) > 0$  or  $(direct) < 0$   
 THEN  
 $(PC) \leftarrow (PC) + rel$

## INC <byte>

**Function:** Increment

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

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

**Example:** Register 0 contains 7EH (01111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence,

```
INC @R0
INC R0
INC @R0
```

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

## INC A

**Bytes:** 1

**Cycles:** 1

**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**Operation:** INC  
 $(A) \leftarrow (A) + 1$

## INC Rn

**Bytes:** 1

**Cycles:** 1

**Encoding:**

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

**Operation:** INC  
 $(Rn) \leftarrow (Rn) + 1$

## INC direct

**Bytes:** 2

**Cycles:** 1

**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** INC  
 $(direct) \leftarrow (direct) + 1$

**INC @Ri****Bytes:** 1**Cycles:** 1**Encoding:**

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

**Operation:** INC  
 $((Ri)) \leftarrow ((Ri)) + 1$ **INC DPTR****Function:** Increment Data Pointer**Description:** Increment the 16-bit data pointer by 1. A 16-bit increment (modulo  $2^{16}$ ) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H will increment the high-order-byte (DPH). No flags are affected.  
This is the only 16-bit register which can be incremented.**Example:** Register DPH and DPL contains 12H and 0FEH, respectively. The instruction sequence,  
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  
 $(DPTR) \leftarrow (DPTR) + 1$ **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 |
|---|---|---|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

|              |
|--------------|
| rel. address |
|--------------|

**Operation:** JB  
 $(PC) \leftarrow (PC) + 3$   
IF (bit) = 1  
THEN  
 $(PC) \leftarrow (PC) + rel$

**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 will not be cleared if it is already a zero.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

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

**Example:** The Accumulator holds 56H (01010110B). The instruction sequence,

JBC ACC.3, LABEL1

JBC ACC.2, LABEL2

will cause program execution to continue at the instruction identified by the label LABEL2, with the Accumulator modified to 52H (01010010B).

**Bytes:** 3**Cycles:** 2**Encoding:**

0 0 0 1

0 0 0 0

bit address

rel. address

**Operation:**

JBC

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

IF (bit) = 1

THEN

 $(bit) \leftarrow 0$  $(PC) \leftarrow (PC) + rel$ **JC rel****Function:** Jump if Carry is set**Description:** If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.**Example:** The carry flag is cleared. The instruction sequence,

JC LABEL1

CPL C

JC 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

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

IF (C) = 1

THEN

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

**JMP @A+DPTR****Function:** Jump indirect**Description:** Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo  $2^{16}$ ): 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)  $\leftarrow$  (A) + (DPTR)**JNB bit, rel****Function:** Jump if Bit is not set**Description:** If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.**Example:** The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The instruction sequence,

```

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

```

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

**Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

|              |
|--------------|
| rel. address |
|--------------|

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

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

```

JNC LABEL1
CPL C
JNC LABEL2

```

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

**Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

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

**JNZ rel****Function:** Jump if Accumulator Not Zero**Description:** If any bit of the Accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.**Example:** The Accumulator originally holds 00H. The instruction sequence,

```

JNZ LABEL1
INC A
JNZ 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  
 $(PC) \leftarrow (PC) + 2$   
 IF  $(A) \neq 0$   
 THEN  $(PC) \leftarrow (PC) + rel$

**JZ rel****Function:** Jump if Accumulator Zero**Description:** If all bits of the Accumulator are zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.**Example:** The Accumulator originally contains 01H. The instruction sequence,

JZ LABEL1

DEC A

JZ LAEEL2

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

**Bytes:** 2**Cycles:** 2**Encoding:**

0 1 1 0

0 0 0 0

rel. address

**Operation:**

JZ

 $(PC) \leftarrow (PC) + 2$ IF  $(A) = 0$ THEN  $(PC) \leftarrow (PC) + \text{rel}$ **LCALL addr16****Function:** Long call**Description:** LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K-byte program memory address space. No flags are affected.**Example:** Initially the Stack Pointer equals 07H. The label “SUT2N” is assigned to program memory location 1234H. After executing the instruction,

LCALL SUT2N

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

 $(PC) \leftarrow (PC) + 3$  $(SP) \leftarrow (SP) + 1$  $((SP)) \leftarrow (PC_{7-0})$  $(SP) \leftarrow (SP) + 1$  $((SP)) \leftarrow (PC_{15-8})$  $(PC) \leftarrow \text{addr}_{15-0}$

**LJMP addr16****Function:** Long Jump**Description:** LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.**Example:** The label “JMPADR” is assigned to the instruction at program memory location 1234H. The instruction,

LJMP JMPADR

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

**Bytes:** 3**Cycles:** 2**Encoding:**

0 0 0 0 0 0 1 0

addr15-addr8

addr7-addr0

**Operation:**

LJMP

(PC) ← addr<sub>15-0</sub>**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&lt;= 30H

MOV A, @R0 ;A&lt;= 40H

MOV R1, A ;R1&lt;= 40H

MOV B, @R1 ;B&lt;= 10H

MOV @R1, P1 ;RAM (40H) &lt;= 0CAH

MOV P2, P1 ;P2 #0CAH

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

**MOV A,Rn****Bytes:** 1**Cycles:** 1**Encoding:**

1 1 1 0 1 r r r

**Operation:**

MOV

(A) ← (Rn)

**\*MOV A,direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(A)←(direct)**\*MOV A,ACC is not a valid instruction****MOV A,@Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
|---|---|---|---|

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

**Operation:** MOV  
(A)←((Ri))**MOV A,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(A)←#data**MOV Rn,A****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** MOV  
(Rn)←(A)**MOV Rn,direct****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|              |  |  |  |
|--------------|--|--|--|
| direct addr. |  |  |  |
|--------------|--|--|--|

**Operation:** MOV  
(Rn)←(direct)**MOV Rn,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(Rn)←#data



**MOV direct, A****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(direct) ← (A)**MOV direct, Rn****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(direct) ← (Rn)**MOV direct, direct****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                 |  |  |  |
|-----------------|--|--|--|
| dir.addr. (src) |  |  |  |
|-----------------|--|--|--|

**Operation:** MOV  
(direct) ← (direct)**MOV direct, @Ri****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

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

|              |  |  |  |
|--------------|--|--|--|
| direct addr. |  |  |  |
|--------------|--|--|--|

**Operation:** MOV  
(direct) ← ((Ri))**MOV direct, #data****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** MOV  
(direct) ← #data**MOV @Ri, A****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
|---|---|---|---|

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

**Operation:** MOV  
((Ri)) ← (A)

**MOV @Ri, direct****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

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

|              |
|--------------|
| direct addr. |
|--------------|

**Operation:** MOV  
((Ri)) ← (direct)**MOV @Ri, #data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
|---|---|---|---|

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

|                |
|----------------|
| immediate data |
|----------------|

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

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

**MOV C, bit****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** MOV  
(C) ← (bit)**MOV bit, C****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** MOV  
(bit) ← (C)

**MOV DPTR, #data 16****Function:** Load Data Pointer with a 16-bit constant**Description:** The Data Pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.  
This is the only instruction which moves 16 bits of data at once.**Example:** The instruction,  
MOV DPTR, #1234H  
will load the value 1234H into the Data Pointer: DPH will hold 12H and DPL will hold 34H.**Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|                     |  |  |  |
|---------------------|--|--|--|
| immediate data 15-8 |  |  |  |
|---------------------|--|--|--|

**Operation:** MOV  
(DPTR) ← #data<sub>15-0</sub>  
DPH DPL ← #data<sub>15-8</sub> #data<sub>7-0</sub>**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 defined by the DB (define byte) directive.REL-PC: INC A  
MOVC A, @A+PC  
RET  
DB 66H  
DB 77H  
DB 88H  
DB 99H

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

**MOVC A, @A+DPTR****Bytes:** 1**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

**Operation:** MOVC  
(A) ← ((A)+(DPTR))

**MOVC A,@A+PC****Bytes:** 1**Cycles:** 2**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

**Operation:** MOVC  
(PC) ← (PC)+1  
(A) ← ((A)+(PC))**MOVX <dest-byte> , <src-byte>****Function:** Move External**Description:** The MOVX instructions transfer data between the Accumulator and a byte of external data memory, hence the “X” appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

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

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

**Example:** An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX A, @R1
MOVX @R0, A
```

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

**MOVX A,@Ri****Bytes:** 1**Cycles:** 2**Encoding:**

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

**Operation:** MOVX  
(A) ← ((Ri))

**MOVX A,@DPTR****Bytes:** 1**Cycles:** 2**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**Operation:** MOVX  
(A) ← ((DPTR))**MOVX @Ri,A****Bytes:** 1**Cycles:** 2**Encoding:**

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

**Operation:** MOVX  
((Ri))← (A)**MOVX @DPTR,A****Bytes:** 1**Cycles:** 2**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**Operation:** MOVX  
(DPTR)←(A)**MUL AB****Function:** Multiply**Description:** MUL AB multiplies the unsigned eight-bit integers in the Accumulator and register B. The low-order byte of the sixteen-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared**Example:** Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

MUL AB

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

**Bytes:** 1**Cycles:** 4**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**Operation:** MUL  
(A)<sub>7:0</sub> ← (A)×(B)  
(B)<sub>15:8</sub>

NOP

**Function:** No Operation

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

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

```
CLR P2.7
NOP
NOP
NOP
NOP
SETB P2.7
```

**Bytes:** 1

**Cycles:** 1

**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**Operation:** NOP  
(PC) ← (PC)+1

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

**Function:** Logical-OR for byte variables

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

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

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

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

```
ORL A, R0
```

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

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

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

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

**ORL A,Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** ORL  
 $(A) \leftarrow (A) \vee (Rn)$ **ORL A,direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** ORL  
 $(A) \leftarrow (A) \vee (\text{direct})$ **ORL A,@Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

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

**Operation:** ORL  
 $(A) \leftarrow (A) \vee ((Ri))$ **ORL A,#data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

**Operation:** ORL  
 $(A) \leftarrow (A) \vee \#data$ **ORL direct, A****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** ORL  
 $(\text{direct}) \leftarrow (\text{direct}) \vee (A)$ **ORL direct, #data****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

|                |
|----------------|
| immediate data |
|----------------|

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

**ORL C, <src-bit>****Function:** Logical-OR for bit variables**Description:** Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (“/”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.**Example:** Set the carry flag if and only if P1.0 = 1, ACC. 7 = 1, or OV = 0:

```

MOV C, P1.0 ;LOAD CARRY WITH INPUT PIN P1.0
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  
 $(C) \leftarrow (C) \vee (\text{bit})$ **ORL C, /bit****Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** ORL  
 $(C) \leftarrow (C) \vee \overline{(\text{bit})}$ **POP direct****Function:** Pop from stack**Description:** The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

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

```

POP DPH
POP DPL

```

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

```

POP SP

```

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

**Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** POP  
 $(\text{direct}) \leftarrow ((\text{SP}))$   
 $(\text{SP}) \leftarrow (\text{SP}) - 1$



**PUSH direct****Function:** Push onto stack**Description:** The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.**Example:** On entering 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  
 $(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$   
 $(A_0) \leftarrow (A_7)$

**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  
 $(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$   
 $(A_0) \leftarrow (C)$   
 $(C) \leftarrow (A_7)$ **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  
 $(A_n) \leftarrow (A_{n+1}) \quad n = 0 - 6$   
 $(A_7) \leftarrow (A_0)$ **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  
 $(A_{n+1}) \leftarrow (A_n) \quad n = 0-6$   
 $(A_7) \leftarrow (C)$   
 $(C) \leftarrow (A_0)$

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

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

**SETB C****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

**Operation:** SETB  
(C) ← 1**SETB bit****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|             |
|-------------|
| bit address |
|-------------|

**Operation:** SETB  
(bit) ← 1**SJMP rel****Function:** Short Jump**Description:** Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128bytes preceding this instruction to 127 bytes following it.

**Example:** The label “RELADR” is assigned to an instruction at program memory location 0123H. The instruction,  
SJMP RELADR  
will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.  
(Note: Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H - 0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be an one-instruction infinite loop).

**Bytes:** 2**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
|---|---|---|---|

|              |
|--------------|
| rel. address |
|--------------|

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

**SUBB A, <src-byte>****Function:** Subtract with borrow

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

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

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

**Example:** The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

SUBB A, R2

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

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

**SUBB A, Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** SUBB  
 $(A) \leftarrow (A) - (C) - (Rn)$

**SUBB A, direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** SUBB  
 $(A) \leftarrow (A) - (C) - (\text{direct})$

**SUBB A, @Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
|---|---|---|---|

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

**Operation:** SUBB  
 $(A) \leftarrow (A) - (C) - ((Ri))$

**SUBB A, #data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| immediate data |
|----------------|

**Operation:** SUBB  
(A) ← (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<sub>3-0</sub>) ↔ (A<sub>7-4</sub>)**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) ↔ (Rn)**XCH A, direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

|                |
|----------------|
| direct address |
|----------------|

**Operation:** XCH  
(A) ↔ (direct)

**XCH A, @Ri****Bytes:** 1**Cycles:** 1**Encoding:**

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

**Operation:** XCH  
(A)  $\longleftrightarrow$  ((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<sub>3-0</sub>)  $\longleftrightarrow$  (Ri<sub>3-0</sub>)**XRL <dest-byte>, <src-byte>****Function:** Logical Exclusive-OR for byte variables**Description:** XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

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

*(Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.)***Example:** If the Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,

XRL A, R0

will leave the Accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combination of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The instruction,

XRL P1, #00110001B

will complement bits 5, 4 and 0 of output Port 1.

**XRL A, Rn****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 1 | r | r | r |
|---|---|---|---|

**Operation:** XRL  
(A)  $\leftarrow$  (A)  $\wedge$  (Rn)**XRL A, direct****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** XRL  
(A)  $\leftarrow$  (A)  $\wedge$  (direct)**XRL A, @Ri****Bytes:** 1**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

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

**Operation:** XRL  
(A)  $\leftarrow$  (A)  $\wedge$  ((Ri))**XRL A, #data****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

**Operation:** XRL  
(A)  $\leftarrow$  (A)  $\wedge$  #data**XRL direct, A****Bytes:** 2**Cycles:** 1**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

**Operation:** XRL  
(direct)  $\leftarrow$  (direct)  $\wedge$  (A)**XRL direct, #data****Bytes:** 3**Cycles:** 2**Encoding:**

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
|---|---|---|---|

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
|---|---|---|---|

|                |  |  |  |
|----------------|--|--|--|
| direct address |  |  |  |
|----------------|--|--|--|

|                |  |  |  |
|----------------|--|--|--|
| immediate data |  |  |  |
|----------------|--|--|--|

**Operation:** XRL  
(direct)  $\leftarrow$  (direct)  $\wedge$  # data



## 第6章 中断系统

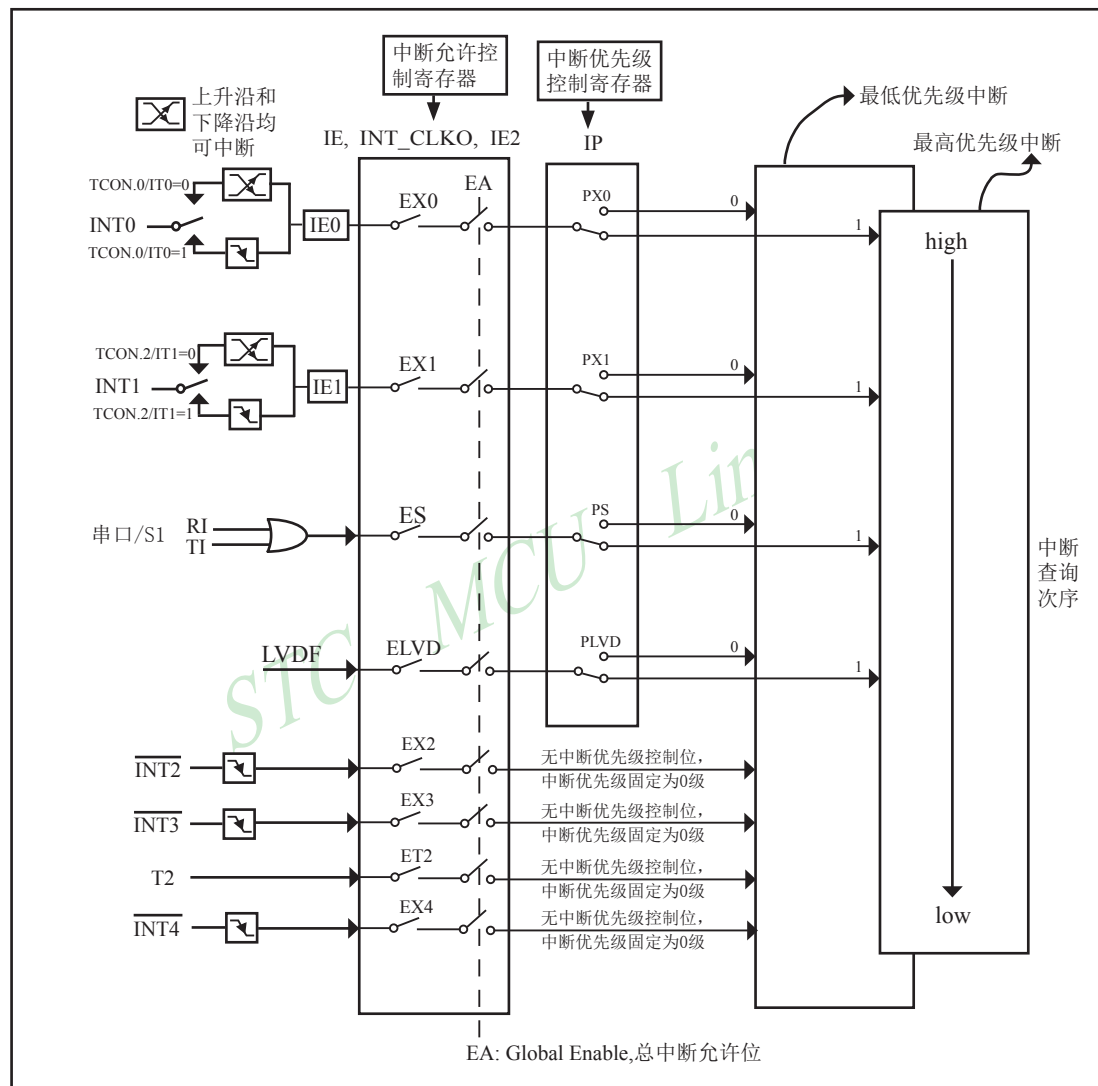
中断系统是为使CPU具有对外界紧急事件的实时处理能力而设置的。

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

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

STC15F104ESW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)、外部中断1(INT1)、串口中断、低压检测(LVD)中断、外部中断2(INT2)、外部中断3(INT3)，定时器T2中断以及外部中断4(INT4)。除外部中断2(INT2)、外部中断3(INT3)、定时器T2中断及外部中断4(INT4)固定是最低优先级中断外，其它的中断都具有2个中断优先级，可实现2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。

## 6.1 中断结构图



STC15F104ESW系列中断结构图

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)脚检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。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(INT4)也可以用于将单片机从掉电模式唤醒。

定时器2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ET2=0,该中断请求标志位会自动地被清0。

当串行口发送或接收完成时,相应的中断请求标志位TI或RI就会被置位,如果串口中断被打开,向CPU请求中断,单片机转去执行该串口中断。中断响应后,TI或RI需由软件清零。

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。

各个中断触发行行为总结如下表所示:

表6-2 中断触发

| 中断源                                 | 触发行行为                                |
|-------------------------------------|--------------------------------------|
| INT0<br>(外部中断0)                     | (IT0 = 1): 下降沿; (IT0 = 0): 上升沿和下降沿均可 |
| INT1<br>(外部中断1)                     | (IT1 = 1): 下降沿; (IT1 = 0): 上升沿和下降沿均可 |
| UART                                | 发送或接受完成                              |
| LVD                                 | 电源电压下降到低于LVD检测电压                     |
| $\overline{\text{INT2}}$<br>(外部中断2) | 下降沿                                  |
| $\overline{\text{INT3}}$<br>(外部中断3) | 下降沿                                  |
| Timer2                              | 定时器2溢出                               |
| $\overline{\text{INT4}}$<br>(外部中断4) | 下降沿                                  |

6.2 中断向量入口地址/查询次序/优先级/请求标志/允许位表

中断向量入口地址/查询次序/优先级/请求标志位/允许位

| 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 | 优先级0(最低) | 优先级1(最高) | 中断请求标志位           | 中断允许控制位                         |
|-------------------------------------|--------|-------------|---------|----------|----------|-------------------|---------------------------------|
| INT0<br>(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<br>(外部中断1)                     | 0013H  | 2           | PX1     | 0        | 1        | IE1               | EX1/EA                          |
| Timer1                              | 001BH  | 3           | PT1     | 0        | 1        | TF1               | ET1/EA                          |
| S1(UART)                            | 0023BH | 4           | PS      | 0        | 1        | RI+TI             | ES/EA                           |
| ADC                                 | 002BH  | 5           | PADC    | 0        | 1        | ADC_FLAG          | EADC/EA                         |
| LVD                                 | 0033H  | 6           | PLVD    | 0        | 1        | LVDF              | ELVD/EA                         |
| PCA                                 | 003BH  | 7           | PPCA    | 0        | 1        | CF+CCF0+CCF1+CCF2 | (ECF+ECCF0+ECCF1+ECCF2+ELVD)/EA |
| S2(UART2)                           | 0043H  | 8           | PS2     | 0        | 1        | S2RI+S2TI         | ES2/EA                          |
| SPI                                 | 004BH  | 9           | PSPI    | 0        | 1        | SPIF              | ESPI/EA                         |
| $\overline{\text{INT2}}$<br>(外部中断2) | 0053H  | 10          | 0       | 0        |          |                   | EX2/EA                          |
| $\overline{\text{INT3}}$<br>(外部中断3) | 005BH  | 11          | 0       | 0        |          |                   | EX3/EA                          |
| Timer 2                             | 0063H  | 12          | 0       | 0        |          |                   | ET2/EA                          |
| -                                   | 006BH  | 13          |         |          |          |                   |                                 |
| System Reserved                     | 0073H  | 14          |         |          |          |                   |                                 |
| System Reserved                     | 007BH  | 15          |         |          |          |                   |                                 |
| $\overline{\text{INT4}}$<br>(外部中断4) | 0083H  | 16(lowest)  | 0       | 0        |          |                   | EX4/EA                          |

## 6.3 在Keil C中如何声明中断函数

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_Routine(void) interrupt 1;
void Int1_Routine(void) interrupt 2;
void Timer1_Routine(void) interrupt 3;
void UART_Routine(void) interrupt 4;
void ADC_Routine(void) interrupt 5;
void LVD_Routine(void) interrupt 6;
void PCA_Routine(void) interrupt 7;
void UART2_Routine(void) interrupt 8;
void SPI_Routine(void) interrupt 9;
void Int2_Routine(void) interrupt 10;
void Int3_Routine(void) interrupt 11;
void Timer2_Routine(void) interrupt 12;
void Int4_Routine(void) interrupt 16;
void S3_Routine(void) interrupt 17;
void S4_Routine(void) interrupt 18;
void Timer3_Routine(void) interrupt 19;
void Timer4_Routine(void) interrupt 20;
```

## 6.4 中断寄存器

| 符号                | 描述                     | 地址  | 位地址及符号 |       |           |     |                     |        |     |     | 复位值        |
|-------------------|------------------------|-----|--------|-------|-----------|-----|---------------------|--------|-----|-----|------------|
|                   |                        |     | MSB    |       |           |     | LSB                 |        |     |     |            |
| IE                | Interrupt Enable       | A8H | EA     | ELVD  | -         | ES  | -                   | EX1    | -   | EX0 | 00x0 x0x0B |
| IE2               | Interrupt Enable 2     | AFH | -      | -     | -         | -   | -                   | ET2    | -   | -   | xxxx x0xxB |
| INT_CLKO<br>AUXR2 | 外部中断允许和时钟输出寄存器         | 8FH | -      | EX4   | EX3       | EX2 | LVD_WAKE            | T2CLKO | -   | -   | x000 00xxB |
|                   |                        |     | -      | -     | -         | -   | -                   | -      | -   | -   |            |
| IP                | Interrupt Priority Low | B8H | -      | PLVD  | -         | PS  | -                   | PX1    | -   | PX0 | x0x0 x0x0B |
| TCON              | 外部中断0和外部中断1的中断请求控制寄存器  | 88H | -      | -     | -         | -   | IE1                 | IT1    | IE0 | IT0 | xxxx 0000B |
| SCON              | Serial Control         | 98H | SM0/FE | SM1   | SM2       | REN | TB8                 | RB8    | TI  | RI  | 0000 0000B |
| PCON              | Power Control register | 87H | SMOD   | SMOD0 | LVDf      | POF | GF1                 | GF0    | PD  | IDL | 0011 0000B |
| AUXR              | 辅助寄存器                  | 8EH | -      | -     | UART_M0x6 | T2R | T2_C $\overline{T}$ | T2x12  | 1   | -   | xx00 00xxB |

上表中列出了与STC15F104ESW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、IE2和INT\_CLKO

STC15F104ESW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 | -  | ES | -  | EX1 | -  | 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。

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

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。

EX0：外部中断0中断允许位，EX0=1允许中断，EX0=0禁止中断。

IE2：中断允许寄存器（不可位寻址）

| SFR name | Address | bit  | B7 | B6 | B5 | B4 | B3 | B2  | B1 | B0 |
|----------|---------|------|----|----|----|----|----|-----|----|----|
| IE2      | AFH     | name | -  | -  | -  | -  | -  | ET2 | -  | -  |

ET2：定时器2的中断允许位。

ET2=1,允许定时器2产生中断；

ET2=0,禁止定时器2产生中断

INT\_CLKO (AUXR2)是STC15F104ESW系列单片机新增寄存器，地址是8FH，INT\_CLKO (AUXR2)格式如下：

INT\_CLKO (AUXR2)：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<br>(AUXR2) | 8FH     | name | -  | EX4 | EX3 | EX2 | LVD_WAKE | T2CLKO | -  | -  |

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。

LVD\_WAKE, T2CLKO与中断无关，在此不作介绍。

STC15F104ESW系列单片机复位以后，IE、IE2和INT\_CLKO(AUXR2)被清0，由用户程序置“1”或清“0”IE、IE2和INT\_CLKO (AUXR2)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IE2和INT\_CLKO(不可位寻址)的内容只可用字节操作指令(即MOV IE2, #DATA或MOV INT\_CLKO, #DATA)来解决。

## 2. 中断优先级控制寄存器IP

传统8051单片机具有两个中断优先级，即高优先级和低优先级，可以实现两级中断嵌套。STC15F104ESW系列单片机通过设置特殊功能寄存器(IP和IP2)中的相应位，可将部分中断设有2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2个优先级中断。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

STC15F104ESW系列单片机的片内各优先级控制寄存器的格式如下：

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 | -  | PS | -  | PX1 | -  | PX0 |

PLVD：低压检测中断优先级控制位。

当PLVD=0时，低压检测中断为最低优先级中断(优先级0)

当PLVD=1时，低压检测中断为最高优先级中断(优先级1)

- PS: 串口中断优先级控制位。  
 当PS=0时，串口中断为最低优先级中断(优先级0)  
 当PS=1时，串口中断为最高优先级中断(优先级1)
- PX1: 外部中断1优先级控制位。  
 当PX1=0时，外部中断1为最低优先级中断(优先级0)  
 当PX1=1时，外部中断1为最高优先级中断(优先级1)
- PX0: 外部中断0优先级控制位。  
 当PX0=0时，外部中断0为最低优先级中断(优先级0)  
 当PX0=1时，外部中断0为最高优先级中断(优先级1)

### 3. 外部中断0和外部中断1的中断请求控制寄存器TCON

TCON为外部中断0和外部中断1的中断请求控制寄存器，TCON格式如下：

TCON：外部中断0和外部中断1的中断请求控制寄存器（可位寻址）

| SFR name | Address | bit  | B7 | B6 | B5 | B4 | B3  | B2  | B1  | B0  |
|----------|---------|------|----|----|----|----|-----|-----|-----|-----|
| TCON     | 88H     | name | -  | -  | -  | -  | IE1 | IT1 | IE0 | IT0 |

- 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. 串行口控制寄存器SCON

SCON为串行口控制寄存器，SCON格式如下：

SCON：串行口控制寄存器（可位寻址）

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

- RI: 串行口接收中断标志。若串行口允许接收且以方式0工作，则每当接收到第8位数据时置1；若以方式1、2、3工作且SM2=0时，则每当接收到停止位的中间时置1；当串行口以方式2或方式3工作且SM2=1时，则仅当接收到的第9位数据RB8为1后，同时还要接收到停止位的中间时置1。RI为1表示串行口正向CPU申请中断(接收中断)，RI必须由用户的中断服务程序清零。
- TI: 串行口发送中断标志。串行口以方式0发送时，每当发送完8位数据，由硬件置1；若以方式1、方式2或方式3发送时，在发送停止位的开始时置1。TI=1表示串行口正在向CPU申请中断(发送中断)。值得注意的是，CPU响应发送中断请求，转向执行中断服务程序时并不将TI清零，TI必须由用户在中断服务程序中清零。

SCON寄存器的其他位与中断无关，在此不作介绍。



在中断允许寄存器IE中，串行口中断相应的允许位是ES/IE.4

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 | -  | ES | -  | EX1 | -  | 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。

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

## 5. 低压检测中断相关寄存器：电源控制寄存器PCON

PCON为电源控制寄存器，PCON格式如下：

PCON：电源控制寄存器

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

LVDF：低压检测标志位，同时也是低压检测中断请求标志位。

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 | -  | ES | -  | EX1 | -  | 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。

## 6.5 中断优先级

除外部中断2 ( $\overline{\text{INT2}}$ )、外部中断3 ( $\overline{\text{INT3}}$ )、定时器T2中断及外部中断4 ( $\overline{\text{INT4}}$ )外，STC15F104ESW系列单片机的所有的中断都具有2个中断优先级。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，STC15F104ESW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. UART                      |           |
| 5. ADC interrupt             |           |
| 6. LVD                       |           |
| 7. PCA                       |           |
| 8. UART2                     |           |
| 9. SPI                       |           |
| 10. $\overline{\text{INT2}}$ |           |
| 11. $\overline{\text{INT3}}$ |           |
| 12. Timer 2                  |           |
| 13.                          |           |
| 14.                          |           |
| 15.                          |           |
| 16. $\overline{\text{INT4}}$ |           |
| 17. UART3                    |           |
| 18. UART4                    |           |
| 19. Timer 3                  |           |
| 20. Timer 4                  | (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_Routine(void) interrupt 1;
void Int1_Routine(void) interrupt 2;
void Timer1_Routine(void) interrupt 3;
void UART_Routine(void) interrupt 4;
void ADC_Routine(void) interrupt 5;
void LVD_Routine(void) interrupt 6;
void PCA_Routine(void) interrupt 7;
void UART2_Routine(void) interrupt 8;
void SPI_Routine(void) interrupt 9;
void Int2_Routine(void) interrupt 10;
void Int3_Routine(void) interrupt 11;
void Timer2_Routine(void) interrupt 12;
void Int4_Routine(void) interrupt 16;
void S3_Routine(void) interrupt 17;
void S4_Routine(void) interrupt 18;
void Timer3_Routine(void) interrupt 19;
void Timer4_Routine(void) interrupt 20;
```

## 6.6 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 |
| S1(UART)             | 0023H |
| ADC interrupt        | 002BH |
| LVD                  | 0033H |
| PCA                  | 003BH |
| S2(UART2)            | 0043H |
| SPI                  | 004BH |
| External Interrupt 2 | 0053H |
| External Interrupt 3 | 005BH |
| Timer 2              | 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)。

**注意:**不能用RET指令代替RETI指令

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

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

## 6.7 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 $IT_x = 0(x = 0,1)$ ，那么系统在 $INT_x(x = 0,1)$ 脚探测到上升沿或下降沿后均可产生外部中断。如果 $IT_x = 1(x = 0,1)$ ，那么系统在 $INT_x(x = 0,1)$ 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( $\overline{INT2}$ )、外部中断3( $\overline{INT3}$ )及外部中断4( $\overline{INT4}$ )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 $EX_n=0(n=2,3,4)$ ，这些中断请求标志位会自动地被清0。[这些中断请求标志位也可以通过软件禁止相应的中断允许控制位将其清“0”\(特殊应用\)](#)。外部中断2( $\overline{INT2}$ )、外部中断3( $\overline{INT3}$ )及外部中断4( $\overline{INT4}$ )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。

## 6.8 中断的测试程序(C和汇编)

### 6.8.1 外部中断0(INT0)的测试程序

#### 6.8.1.1 外部中断INT0(上升沿+下降沿)的测试程序(C和汇编)

##### 1.C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT0中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

bit FLAG;
sbit P10 = P1^0; //1:上升沿中断 0:下降沿中断

//-----
//外部中断服务程序
void exint0() interrupt 0 //INT0中断入口
{
 P10 = !P10; //将测试口取反
 FLAG = INT0; //保存INT0口的状态, INT0=0(下降沿); INT0=1(上升沿)
}

//-----
void main()
{
 INT0 = 1;
 IT0 = 0; //设置INT0的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 EX0 = 1; //使能INT0中断
 EA = 1;

 while (1);
}

```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT0中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译  
 //假定测试芯片的工作频率为18.432MHz

```

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

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0003H
 LJMP EXINT0 //INT0中断入口
//-----

MAIN: ORG 0100H

 MOV SP, #3FH

 CLR IT0 //设置INT0的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 SETB EX0 //使能INT0中断
 SETB EA
 SJMP $

//-----
//外部中断服务程序

EXINT0:
 CPL P1.0 //将测试口取反
 PUSH PSW
 MOV C, INT0 //读取INT0口的状态
 MOV FLAG, C //保存, INT0=0(下降沿); INT0=1(上升沿)
 POP PSW
 RETI

;-----

 END

```

### 6.8.1.2 外部中断INT0(下降沿) 的测试程序(C和汇编)

#### 1.C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT0中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sbit P10 = P1^0;

//-----
//外部中断服务程序
void exint0() interrupt 0 //INT0中断入口
{
 P10 = !P10; //将测试口取反
}

//-----

void main()
{
 INT0 = 1;
 IT0 = 1; //设置INT0的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 EX0 = 1; //使能INT0中断
 EA = 1;

 while (1);
}

```



## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT0中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0003H //INT0中断入口
 LJMP EXINT0

//-----

MAIN: ORG 0100H
 MOV SP, #3FH

 SETB IT0 //设置INT0的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 SETB EX0 //使能INT0中断
 SETB EA
 SJMP $

//-----
//外部中断服务程序

EXINT0:
 CPL P1.0 //将测试口取反
 RETI

;-----

 END

```

## 6.8.2 外部中断1(INT1)的测试程序

### 6.8.2.1 外部中断INT1(上升沿+下降沿)的测试程序(C和汇编)

#### 1.C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT1中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

bit FLAG; //1:上升沿中断 0:下降沿中断
sbit P10 = P1^0;

//-----
//外部中断服务程序
void exint1() interrupt 2 //INT1中断入口
{
 P10 = !P10; //将测试口取反
 FLAG = INT1; //保存INT1口的状态, INT1=0(下降沿); INT1=1(上升沿)
}

//-----
void main()
{
 INT1 = 1;
 IT1 = 0; //设置INT1的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 EX1 = 1; //使能INT1中断
 EA = 1;

 while (1);
}

```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT1中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

FLAG BIT 20H.0 //1:上升沿中断 0:下降沿中断

//-----

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0013H
 LJMP EXINT1 //INT1中断入口

//-----

MAIN: ORG 0100H

 MOV SP, #3FH

 CLR IT1 //设置INT1的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 SETB EX1 //使能INT1中断
 SETB EA
 SJMP $

//-----
//外部中断服务程序

EXINT1:
 CPL P1.0 //将测试口取反
 PUSH PSW
 MOV C, INT1 //读取INT1口的状态
 MOV FLAG, C //保存, INT1=0(下降沿); INT0=1(上升沿)
 POP PSW
 RETI

;-----

 END

```

### 6.8.2.2 外部中断INT1(下降沿) 的测试程序(C和汇编)

#### 1.C程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT1中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sbit P10 = P1^0;

//-----
//外部中断服务程序
void exint1() interrupt 2 //INT1中断入口
{
 P10 = !P10; //将测试口取反
}

//-----

void main()
{
 INT1 = 1;
 IT1 = 1; //设置INT1的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 EX1 = 1; //使能INT1中断
 EA = 1;

 while (1);
}
```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 INT1中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译  
 //假定测试芯片的工作频率为18.432MHz

```

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0013H
 LJMP EXINT1 //INT1中断入口

//-----

MAIN: ORG 0100H
 MOV SP, #3FH

 SETB IT1 //设置INT1的中断类型 (1:仅下降沿 0:上升沿和下降沿)
 SETB EX1 //使能INT1中断
 SETB EA
 SJMP $

//-----
//外部中断服务程序

EXINT1:
 CPL P1.0 //将测试口取反
 RETI

;-----

 END

```

## 6.8.3 外部中断2(INT2)(下降沿中断)的测试程序(C和汇编)

### 1.C程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机外部中断2 (INT2) (下降沿) -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sfr INT_CLKO = 0x8f; //外部中断与时钟输出控制寄存器
sbit P10 = P1^0;

//-----

//外部中断服务程序
void exint2() interrupt 10 //INT2中断入口
{
 P10 = !P10; //将测试口取反

 // INT_CLKO &= 0xEF; //若需要手动清除中断标志,可先关闭中断,
 // //此时系统会自动清除内部的中断标志
 // INT_CLKO |= 0x10; //然后再开中断即可
}

void main()
{
 INT_CLKO |= 0x10; //(EX2 = 1)使能INT2中断
 EA = 1;

 while (1);
}
```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机外部中断2 (INT2) (下降沿) -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

INT_CLKO DATA 08FH //外部中断与时钟输出控制寄存器
//-----

ORG 0000H
LJMP MAIN //复位入口

ORG 0053H //INT2中断入口
LJMP EXINT2

//-----
MAIN:
ORG 0100H
MOV SP, #3FH

ORL INT_CLKO, #10H //(EX2 = 1)使能INT2中断

SETB EA

SJMP $

//-----
//外部中断服务程序

EXINT2:
CPL P1.0 //将测试口取反

// ANL INT_CLKO, #0EFH //若需要手动清除中断标志,可先关闭中断,
// ORL INT_CLKO, #10H //此时系统会自动清除内部的中断标志
// //然后再开中断即可

RETI
;-----
END

```

## 6.8.4 外部中断3( $\overline{\text{INT3}}$ ) (下降沿中断) 的测试程序(C和汇编)

### 1.C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机外部中断3 ($\overline{\text{INT3}}$) (下降沿) -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sfr INT_CLKO = 0x8f; //外部中断与时钟输出控制寄存器
sbit P10 = P1^0;

//-----
//外部中断服务程序
void exint3() interrupt 11 //INT3中断入口
{
 P10 = !P10; //将测试口取反

// INT_CLKO &= 0xDF; //若需要手动清除中断标志,可先关闭中断,
// //此时系统会自动清除内部的中断标志
// INT_CLKO |= 0x20; //然后再开中断即可
}

void main()
{
 INT_CLKO |= 0x20; //(EX3 = 1)使能INT3中断
 EA = 1;

 while (1);
}

```



## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机外部中断3 (INT3) (下降沿) -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

INT_CLKO DATA 08FH //外部中断与时钟输出控制寄存器
//-----

ORG 0000H
LJMP MAIN //复位入口

ORG 005BH //INT3中断入口
LJMP EXINT3
//-----

MAIN:
ORG 0100H
MOV SP, #3FH

ORL INT_CLKO, #20H //(EX3 = 1)使能INT3中断

SETB EA
SJMP $

//-----
//外部中断服务程序

EXINT3:
CPL P1.0 //将测试口取反

// ANL INT_CLKO, #0DFH //若需要手动清除中断标志,可先关闭中断,
// ORL INT_CLKO, #20H //此时系统会自动清除内部的中断标志
// //然后再开中断即可

RETI
;-----

END

```

## 6.8.5 外部中断4( $\overline{\text{INT4}}$ ) (下降沿中断) 的测试程序(C和汇编)

### 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机外部中断4 ($\overline{\text{INT4}}$) (下降沿) -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----

sfr INT_CLKO = 0x8f; //外部中断与时钟输出控制寄存器
sbit P10 = P1^0;

//-----

//外部中断服务程序
void exint4() interrupt 16 //INT3中断入口
{
 P10 = !P10; //将测试口取反

 // INT_CLKO &= 0xBF; //若需要手动清除中断标志,可先关闭中断,
 // //此时系统会自动清除内部的中断标志
 // INT_CLKO |= 0x40; //然后再开中断即可
}

void main()
{
 INT_CLKO |= 0x40; //(EX4 = 1)使能INT4中断
 EA = 1;

 while (1);
}

```

## 6.8.6 T2扩展为外部下降沿中断的测试程序(C和汇编)

### ——利用T2的外部计数方式

#### 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 T2扩展为外部下降沿中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
sfr IE2 = 0xaf; //中断使能寄存器2
sfr AUXR = 0x8e; //辅助寄存器
sfr T2H = 0xd6; //定时器2高8位
sfr T2L = 0xd7; //定时器2低8位

sbit P10 = P1^0;

//-----
//中断服务程序
void t2int() interrupt 12 //中断入口
{
 P10 = !P10; //将测试口取反

 // IE2 &= ~0x04; //若需要手动清除中断标志,可先关闭中断,
 // //此时系统会自动清除内部的中断标志
 // IE2 |= 0x04; //然后再开中断即可
}

void main()
{
 AUXR |= 0x04; //定时器2为1T模式

```

```
AUXR |= 0x08; //T2_C/T=1, T2(P3.1)引脚为时钟源
T2H = T2L = 0xff; //初始化计时值
AUXR |= 0x10; //定时器2开始计时

IE2 |= 0x04; //开定时器2中断

EA = 1;

while (1);

}
```

## 2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 T2扩展为外部下降沿中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译  
//假定测试芯片的工作频率为18.432MHz

```
IE2 DATA 0AFH //中断使能寄存器2
AUXR DATA 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位
```

//-----

```
ORG 0000H
LJMP MAIN //复位入口
```

```
ORG 0063H //中断入口
LJMP T2INT
```

//-----

```
ORG 0100H
```

MAIN:

```
MOV SP, #3FH

ORL AUXR, #04H //定时器2为1T模式
ORL AUXR, #08H //T2_C/T=1, T2(P3.1)引脚为时钟源

MOV A, #0FFH //初始化计时值
MOV T2L, A
MOV T2H, A

ORL AUXR, #10H //定时器2开始计时

ORL IE2, #04H //开定时器2中断

SETB EA

SJMP $

//-----
//外部中断服务程序

T2INT:
CPL P1.0 //将测试口取反

// ANL IE2, #0FBH //若需要手动清除中断标志,可先关闭中断,
// //此时系统会自动清除内部的中断标志
// ORL IE2, #04H //然后再开中断即可

RETI

;-----

END
```

## 第7章 定时器/计数器T2及其应用

STC15F104ESW系列单片机内部设置了1个16位定时器/计数器T2。定时器/计数器T2都具有计数方式和定时方式两种工作方式，用特殊功能寄存器AUXR中的控制位— T2\_C/T来选择T2为定时器还是计数器。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(T2为P3.1)，则为计数方式，每来一个脉冲加1。

当定时器/计数器T2工作在定时模式时，特殊功能寄存器AUXR中T2x12决定了是系统时钟/12还是系统时钟/1(不分频)后让T2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T2的工作模式固定为16位自动重装载模式。T2可以当定时器使用，也可以当串口的波特率发生器和可编程时钟输出。

### 7.1 定时器/计数器T2的相关寄存器

| 符号                | 描述                        | 地址  | 位地址及其符号 |      |           |     |          |        |   |     | 复位值        |
|-------------------|---------------------------|-----|---------|------|-----------|-----|----------|--------|---|-----|------------|
|                   |                           |     | MSB     |      |           |     | LSB      |        |   |     |            |
| T2H               | 定时器2高8位寄存器                | D6H |         |      |           |     |          |        |   |     | 0000 0000B |
| T2L               | 定时器2低8位寄存器                | D7H |         |      |           |     |          |        |   |     | 0000 0000B |
| AUXR              | 辅助寄存器                     | 8EH | -       | -    | UART_M0x6 | T2R | T2_C/T   | T2x12  | - | -   | xx00 00xxB |
| INT_CLKO<br>AUXR2 | 外部中断允许和时钟输出寄存器            | 8FH | -       | EX4  | EX3       | EX2 | LVD_WAKE | T2CLKO | - | -   | x000 00xxB |
| IE                | Interrupt Enable          | A8H | EA      | ELVD | -         | ES  | -        | EX1    | - | EX0 | 00x0 x0x0B |
| IE2               | Interrupt Enable register | AFH | -       | -    | -         | -   | -        | ET2    | - | -   | xxxx,x0xx  |

## 1、定时器2的控制寄存器：辅助寄存器AUXR

STC15F104ESW系列单片机是 1T 的 8051 单片机，定时器 2 复位后是传统 8051 的速度，即 12 分频，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 T2 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 | -  | -  | UART_M0x6 | T2R | T2_C/T | T2x12 | -  | -  |

UART\_M0x6：串口模式 0 的通信速度设置位。

0，UART 串口模式 0 的速度是传统 8051 单片机串口的速度，12 分频；

1，UART 串口模式 0 的速度是传统 8051 单片机串口速度的 6 倍，2 分频

T2R：定时器 2 允许控制位

0，不允许定时器 2 运行；

1，允许定时器 2 运行

T2\_C/T：控制定时器 2 用作定时器或计数器。

0，用作定时器(对内部系统时钟进行计数)；

1，用作计数器(对引脚 T2/P3.1 的外部脉冲进行计数)

T2x12：定时器 2 速度控制位

0，定时器 2 是传统 8051 速度，12 分频；

1，定时器 2 的速度是传统 8051 的 12 倍，不分频

当串口用 T2 作为波特率发生器时，则由 T2x12 决定串口是 12T 还是 1T。

## 2、T2 的时钟输出允许控制位 T2CLKO

T2CLKO/P3.0 的时钟输出控制由 INT\_CLKO(AUXR2) 寄存器的 T2CLKO 位控制。T2CLKO/CLKOUT2 的输出时钟频率由定时器 2 控制，不要允许相应的定时器中断，免得 CPU 反复进中断。定时器 2 的工作模式固定为模式 0 (16 位自动重装载模式)，在此模式下定时器 2 可用作可编程时钟输出。

INT\_CLKO (AUXR2) 格式如下：

INT\_CLKO (AUXR2)：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<br>AUXR2 | 8FH     | name | -  | EX4 | EX3 | EX2 | LVD_WAKE | T2CLKO | -  | -  |

T2CLKO：是否允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

1：允许将P3.0脚配置为定时器2的时钟输出T2CLKO/CLKOUT2，输出时钟频率= $T2\text{溢出率}/2$

如果 $T2\_C/\overline{T}=0$ ，定时器/计数器T2是对内部系统时钟计数，则：

T2工作在1T模式( $AUXR.2/T2x12=1$ )时的输出频率 =  $(SYSclk) / (65536-[RL\_TH2, RL\_TL2])/2$

T2工作在12T模式( $AUXR.2/T2x12=0$ )时的输出频率 =  $(SYSclk) / 12 / (65536-[RL\_TH2, RL\_TL2])/2$

如果 $T2\_C/\overline{T}=1$ ，定时器/计数器T2是对外部脉冲输入(P3.1/T2)计数，则：

输出时钟频率 =  $(T2\_Pin\_CLK) / (65536-[RL\_TH2, RL\_TL2])/2$

0：不允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

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

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

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

### 3、T2的中断允许控制位ET2

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 | ES | ET1 | EX1 | ET0 | EX0 |

EA：CPU的总中断允许控制位，EA=1，CPU开放中断，EA=0，CPU屏蔽所有的中断申请。

EA的作用是使中断允许形成多级控制。即各中断源首先受EA控制；其次还受各中断源自己的中断允许控制位控制。

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。

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

IE2：中断允许寄存器（不可位寻址）

| SFR name | Address | bit  | B7 | B6 | B5 | B4 | B3 | B2  | B1 | B0 |
|----------|---------|------|----|----|----|----|----|-----|----|----|
| IE2      | AFH     | name | -  | -  | -  | -  | -  | ET2 | -  | -  |

ET2：定时器2的中断允许位。

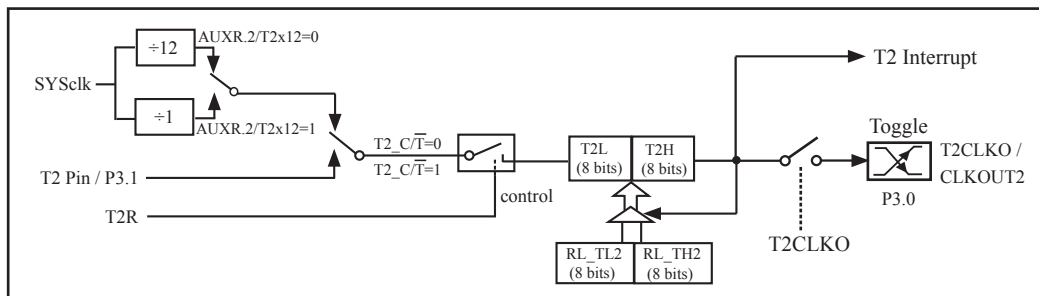
ET2=1,允许定时器2产生中断；

ET2=0,禁止定时器2产生中断



## 7.2 定时器/计数器2作定时器及其测试程序(C和汇编)

定时器/计数器2的原理框图如下:



定时器/计数器2的工作模式: 16位自动重装

STC创新设计, 请不要抄袭, 再抄袭就很无耻了

T2R/AUXR. 4为AUXR寄存器内的控制位, AUXR寄存器各位的具体功能描述见上节AUXR寄存器的介绍。

当 $T2\_C/\overline{T}=0$ 时, 多路开关连接到系统时钟输出, T2对内部系统时钟计数, T2工作在定时方式。当 $T2\_C/\overline{T}=1$ 时, 多路开关连接到外部脉冲输入P3.1/T2, 即T2工作在计数方式。

STC15F104ESW系列单片机的定时器2有两种计数速率: 一种是12T模式, 每12个时钟加1, 与传统8051单片机相同; 另外一种1T模式, 每个时钟加1, 速度是传统8051单片机的12倍。T2的速率由特殊功能寄存器AUXR中的 $T2x12$ 决定, 如果 $T2x12=0$ , T2则工作在12T模式; 如果 $T2x12=1$ , T2则工作在1T模式。

定时器2有2个隐藏的寄存器RL\_TH2和RL\_TL2。RL\_TH2与T2H共有同一个地址, RL\_TL2与T2L共有同一个地址。当 $T2R=0$ 即定时器/计数器2被禁止工作时, 对T2L写入的内容会同时写入RL\_TL2, 对T2H写入的内容也会同时写入RL\_TH2。当 $T2R=1$ 即定时器/计数器2被允许工作时, 对T2L写入内容, 实际上不是写入当前寄存器T2L中, 而是写入隐藏的寄存器RL\_TL2中; 对T2H写入内容, 实际上也不是写入当前寄存器T2H中, 而是写入隐藏的寄存器RL\_TH2。当读T2H和T2L的内容时, 所读的内容就是T2H和T2L的内容, 而不是RL\_TH2和RL\_TL2的内容。

这样可以巧妙地实现16位重装定时器。[T2L, T2H]的溢出不仅置位被隐藏的中断请求标志位(定时器2的中断请求标志位对用户不可见), 使CPU转去执行定时器2中断程序, 而且会自动将[RL\_TL2, RL\_TH2]的内容重新装入[T2L, T2H]。

## 7.2.1 定时器2的16位自动重装载模式的测试程序(C和汇编)

### 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2的16位自动重装载模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

//-----

/* define constants */
#define FOSC 18432000L

#define T38_4KHz (256-18432000/12/38400/2) //38.4KHz

/* define SFR */

sfr IE2 = 0xAF; // (IE2.2) timer2 interrupt control bit
sfr T2 = 0x9C;
sfr AUXR = 0x8E;

sbit TEST_PIN = P0^0; //test pin

//-----

/* Timer2 interrupt routine */
void t2_isr() interrupt 12 using 1
{
 TEST_PIN = !TEST_PIN;
}

//-----

```

```

/* main program */
void main()
{
 T2 = T38_4KHz; //set timer2 reload value
 AUXR |= 0x10; //timer2 start run
 IE2 |= 0x04; //enable timer2 interrupt
 EA = 1; //open global interrupt switch

 while (1); //loop
}

```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2的16位自动重装载模式举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

IE2 DATA 0AFH //中断使能寄存器2
AUXR DATA 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

F38_4KHz EQU 0FF10H //38.4KHz(1T模式下, 65536-18432000/2/38400)

```

//-----

```

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0063H
 LJMP T2INT //中断入口

```

//-----

```

 ORG 0100H
MAIN:
 MOV SP, #3FH

 ORL AUXR, #04H //定时器2为1T模式

 MOV T2L, #LOW F38_4KHz //初始化计时值
 MOV T2H, #HIGH F38_4KHz

 ORL AUXR, #10H //定时器2开始计时

 ORL IE2, #04H //开定时器2中断

 SETB EA

 SJMP $

//-----
//外部中断服务程序

T2INT:
 CPL P1.0 //将测试口取反

// ANL IE2, #0FBH //若需要手动清除中断标志,可先关闭中断,
// //此时系统会自动清除内部的中断标志
// ORL IE2, #04H //然后再开中断即可

 RETI

;-----

 END
```

## 7.2.2 定时器2扩展为外部下降沿中断的测试程序(C和汇编)

;定时器2中断(下降沿中断)的测试程序, 定时器/计数器2工作在计数模式中的16位自动重装载模式

### 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 T2扩展为外部下降沿中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

//-----
sfr IE2 = 0xaf; //中断使能寄存器2
sfr AUXR = 0x8e; //辅助寄存器
sfr T2H = 0xd6; //定时器2高8位
sfr T2L = 0xd7; //定时器2低8位
sbit P10 = P1^0;
//-----

//中断服务程序
void t2int() interrupt 12 //中断入口
{
 P10 = !P10; //将测试口取反

 // IE2 &= ~0x04; //若需要手动清除中断标志,可先关闭中断,
 // //此时系统会自动清除内部的中断标志

 // IE2 |= 0x04; //然后再开中断即可
}

void main()
{
 AUXR |= 0x04; //定时器2为1T模式
 AUXR |= 0x08; //T2_C/T=1, T2(P3.1)引脚为时钟源
 T2H = T2L = 0xff; //初始化计时值
 AUXR |= 0x10; //定时器2开始计时

 IE2 |= 0x04; //开定时器2中断

 EA = 1;

 while (1);
}

```

## 2. 汇编程序：

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 T2扩展为外部下降沿中断举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序 */
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

IE2 DATA 0AFH //中断使能寄存器2
AUXR DATA 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

//-----

 ORG 0000H
 LJMP MAIN //复位入口

 ORG 0063H
 LJMP T2INT //中断入口

//-----

MAIN: ORG 0100H

 MOV SP, #3FH

 ORL AUXR, #04H //定时器2为1T模式
 ORL AUXR, #08H //T2_C/T=1, T2(P3.1)引脚为时钟源

 MOV A, #0FFH //初始化计时值
 MOV T2L, A
 MOV T2H, A

 ORL AUXR, #10H //定时器2开始计时

 ORL IE2, #04H //开定时器2中断

 SETB EA

 SJMP $

```

//-----

//外部中断服务程序

T2INT:

CPL P1.0

//将测试口取反

// ANL IE2, #0FBH

//若需要手动清除中断标志,可先关闭中断,

//此时系统会自动清除内部的中断标志

// ORL IE2, #04H

//然后再开中断即可

RETI

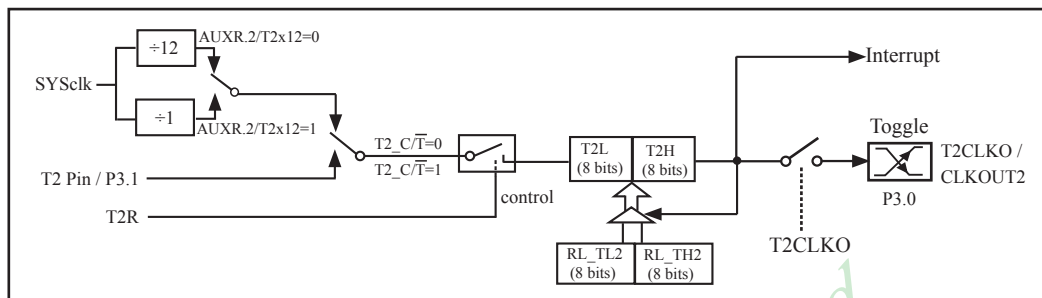
;-----

END

STC MCU Limited

## 7.3 定时器/计数器2对内部系统时钟或外部引脚T2的时钟输入进行可编程时钟分频输出及其测试程序(C和汇编)

定时器/计数器2的原理框图如下：



定时器/计数器2的工作模式: 16位自动重装

STC创新设计，请不要抄袭，再抄袭就很无耻了

定时器T2除可当定时器/计数器使用外，还可作可编程时钟输出。当定时器/计数器2用作可编程时钟输出时，不要允许相应的定时器中断，免得CPU反复进中断。

当T2CLKO/INT\_CLKO.2=1时，P3.0管脚配置为定时器2的时钟输出T2CLKO/CLKOUT2。

输出时钟频率 = T2 溢出率 / 2

如果T2\_C/T=0，定时器/计数器T2对内部系统时钟计数，则：

T2工作在1T模式(AUXR.2/T2x12=1)时的输出时钟频率 = (SYSclk)/(65536-[RL\_TH2, RL\_TL2])/2

T2工作在12T模式(AUXR.2/T2x12=0)时的输出时钟频率 = (SYSclk)/12/(65536-[RL\_TH2, RL\_TL2])/2

如果T2\_C/T=1，定时器/计数器T2是对外部脉冲输入(P3.1/T2)计数，则：

输出时钟频率 = (T2\_Pin\_CLK) / (65536-[RL\_TH2, RL\_TL2])/2

上面所有的式子中RL\_TH2是T2H的重装载寄存器，RL\_TL2是T2L的重装载寄存器。



下面是定时器2对内部系统时钟或外部引脚T2/P3.1的时钟输入进行可编程时钟分频输出的程序举例(C和汇编):

### 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2的可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使用了STC的资料及程序 */
/* 如果要在文章中应用此代码,请在文章中注明使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中,选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

#define FOSC 18432000L

//-----

sfr AUXR = 0x8e; //辅助特殊功能寄存器
sfr INT_CLKO = 0x8f; //唤醒和时钟输出功能寄存器
sfr T2H = 0xD6; //定时器2高8位
sfr T2L = 0xD7; //定时器2低8位

sbit T2CLKO = P3^0; //定时器2的时钟输出脚

#define F38_4KHz (65536-FOSC/2/38400) //1T模式
//#define F38_4KHz (65536-FOSC/2/12/38400) //12T模式

//-----

void main()
{
 AUXR |= 0x04; //定时器2为1T模式
 // AUXR &= ~0x04; //定时器2为12T模式

```

```

// AUXR &= ~0x08; //T2_C/T=0, 对内部时钟进行时钟输出
// AUXR |= 0x08; //T2_C/T=1, 对T2(P3.1)引脚的外部时钟进行时钟输出

T2L = F38_4KHz; /初始化计时值
T2H = F38_4KHz >> 8;

AUXR |= 0x10; //定时器2开始计时
INT_CLKO = 0x04; //使能定时器2的时钟输出功能

while (1); //程序终止
}

```

## 2. 汇编程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码, 请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码, 请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

AUXR DATA 08EH //辅助特殊功能寄存器
INT_CLKO DATA 08FH //唤醒和时钟输出功能寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

T2CLKO BIT P3.0 //定时器2的时钟输出脚

F38_4KHz EQU 0FF10H //38.4KHz(1T模式下, 65536-18432000/2/38400)
//F38_4KHz EQU 0FFECH //38.4KHz(12T模式下, (65536-18432000/2/12/38400))

//-----

```

```
ORG 0000H
LJMP MAIN //复位入口

//-----

ORG 0100H
MAIN:
MOV SP, #3FH

ORL AUXR, #04H //定时器2为1T模式
// ANL AUXR, #0FBH //定时器2为12T模式

ANL AUXR, #0F7H //T2_C/T=0, 对内部时钟进行时钟输出
// ORL AUXR, #08H //T2_C/T=1, 对T2(P3.1) 引脚的外部时钟进行时钟输出

MOV T2L, #LOW F38_4KHz //初始化计时值
MOV T2H, #HIGH F38_4KHz
ORL AUXR, #10H //定时器2开始计时
MOV INT_CLKO, #04H //使能定时器2的时钟输出功能

SJMP $ //程序终止

;-----

END
```

## 7.4 定时器/计数器2作串行口波特率发生器及其测试程序

定时器T2除可当定时器/计数器和可编程时钟输出使用外，还可作串行口波特率发生器。

串行口如果工作在模式1（8位UART，波特率可变）和模式3（9位UART，波特率可变）时，其可变的波特率可以由定时器T2产生。此时，

$$\text{串行口的波特率} = (\text{定时器T2的溢出率}) / 4.$$

（注意：此时波特率也与SMOD无关。）

当T2工作在1T模式(AUXR.2/T2x12=1)时，定时器T2的溢出率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时，串行口的波特率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

当T2工作在12T模式(AUXR.2/T2x12=0)时，定时器2的溢出率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时，串行口的波特率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

### 1. C程序：

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2用作串口的波特率发生器举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序----*/
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#include "reg51.h"
```

```
#include "intrins.h"
```

```
typedef unsigned char BYTE;
typedef unsigned int WORD;
```

```
#define FOSC 18432000L //系统频率
#define BAUD 115200 //串口波特率

#define NONE_PARITY 0 //无校验
#define ODD_PARITY 1 //奇校验
#define EVEN_PARITY 2 //偶校验
#define MARK_PARITY 3 //标记校验
#define SPACE_PARITY 4 //空白校验
```

```

#define PARITYBIT EVEN_PARITY //定义校验位

sfr AUXR = 0x8e; //辅助寄存器
sfr T2H = 0xd6; //定时器2高8位
sfr T2L = 0xd7; //定时器2低8位

sbit P22 = P2^2;

bit busy;

void SendData(BYTE dat);
void SendString(char *s);
void main()
{
 #if (PARITYBIT == NONE_PARITY)
 SCON = 0x50; //8位可变波特率
 #elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
 SCON = 0xda; //9位可变波特率,校验位初始为1
 #elif (PARITYBIT == SPACE_PARITY)
 SCON = 0xd2; //9位可变波特率,校验位初始为0
 #endif

 T2L = (65536 - (FOSC/4/BAUD)); //设置波特率重装值
 T2H = (65536 - (FOSC/4/BAUD))>>8;
 AUXR = 0x14; //T2为1T模式,并启动定时器2
 AUXR |= 0x01; //选择定时器2为串口的波特率发生器
 ES = 1; //使能串口中断
 EA = 1;

 SendString("STC15F104ESW\r\nUart Test !\r\n");
 while(1);
}

/*-----
UART 中断服务程序
-----*/
void Uart() interrupt 4 using 1
{
 if (RI)
 {
 RI = 0; //清除RI位
 P0 = SBUF; //P0显示串口数据
 P22 = RB8; //P2.2显示校验位
 }
 if (TI)
 {
 TI = 0; //清除TI位
 busy = 0; //清忙标志
 }
}

```

```
/*-----
发送串口数据
-----*/
void SendData(BYTE dat)
{
 while (busy); //等待前面的数据发送完成
 ACC = dat; //获取校验位P (PSW.0)
 if (P) //根据P来设置校验位
 {
 #if (PARITYBIT == ODD_PARITY)
 TB8 = 0; //设置校验位为0
 #elif (PARITYBIT == EVEN_PARITY)
 TB8 = 1; //设置校验位为1
 #endif
 }
 else
 {
 #if (PARITYBIT == ODD_PARITY)
 TB8 = 1; //设置校验位为1
 #elif (PARITYBIT == EVEN_PARITY)
 TB8 = 0; //设置校验位为0
 #endif
 }
 busy = 1;
 SBUF = ACC; //写数据到UART数据寄存器
}

/*-----
发送字符串
-----*/
void SendString(char *s)
{
 while (*s) //检测字符串结束标志
 {
 SendData(*s++); //发送当前字符
 }
}
```

## 2. 汇编程序:

```

/*----- */
/* --- STC MCU Limited. ----- */
/* --- STC15F104ESW 系列 定时器2用作串口的波特率发生器举例----- */
/* --- Mobile: (86)13922809991 ----- */
/* --- Fax: 86-755-82905966 ----- */
/* --- Tel: 86-755-82948412 ----- */
/* --- Web: www.STCMCU.com ----- */
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序--- */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序--- */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可----- */
/*----- */

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

#define NONE_PARITY 0 //无校验
#define ODD_PARITY 1 //奇校验
#define EVEN_PARITY 2 //偶校验
#define MARK_PARITY 3 //标记校验
#define SPACE_PARITY 4 //空白校验

```

```

#define PARITYBIT EVEN_PARITY //定义校验位

```

//-----

```

AUXR EQU 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

```

//-----

```

BUSY BIT 20H.0 //忙标志位

```

//-----

```

ORG 0000H
LJMP MAIN

```

```

ORG 0023H
LJMP UART_ISR

```

//-----

```

ORG 0100H

```

MAIN:

```

CLR BUSY
CLR EA
MOV SP, #3FH

```

```

#if (PARITYBIT == NONE_PARITY)

```

```

 MOV SCON, #50H //8位可变波特率

```

```

#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)

```

```

 MOV SCON, #0DAH //9位可变波特率,校验位初始为1
#elif (PARITYBIT == SPACE_PARITY)
 MOV SCON, #0D2H //9位可变波特率,校验位初始为0
#endif

//-----
 MOV T2L, #0D8H //设置波特率重装值(65536-18432000/4/115200)
 MOV T2H, #0FFH
 MOV AUXR, #14H //T2为1T模式,并启动定时器2
 ORL AUXR, #01H //选择定时器2为串口的波特率发生器
 SETB ES //使能串口中断
 SETB EA

 MOV DPTR, #TESTSTR //发送测试字符串
 LCALL SENDSTRING

 SJMP $
;-----
TESTSTR:
 DB "STC15F104ESW Uart1 Test !",0DH,0AH,0

; /*-----
; UART 中断服务程序
; -----*/
UART_ISR:
 PUSH ACC
 PUSH PSW
 JNB RI, CHECKTI //检测RI位
 CLR RI //清除RI位
 MOV P0, SBUF //P0显示串口数据
 MOV C, RB8
 MOV P2.2, C //P2.2显示校验位
CHECKTI:
 JNB TI, ISR_EXIT //检测TI位
 CLR TI //清除TI位
 CLR BUSY //清忙标志
ISR_EXIT:
 POP PSW
 POP ACC
 RETI

; /*-----
; 发送串口数据
; -----*/
SENDDATA:
 JB BUSY, $ //等待前面的数据发送完成
 MOV ACC, A //获取校验位P (PSW.0)
 JNB P, EVEN1INACC //根据P来设置校验位

```



```

ODD1INACC:
#if (PARITYBIT == ODD_PARITY)
 CLR TB8 //设置校验位为0
#elif (PARITYBIT == EVEN_PARITY)
 SETB TB8 //设置校验位为1
#endif
 SJMP PARITYBITOK
EVEN1INACC:
#if (PARITYBIT == ODD_PARITY)
 SETB TB8 //设置校验位为1
#elif (PARITYBIT == EVEN_PARITY)
 CLR TB8 //设置校验位为0
#endif
PARITYBITOK: //校验位设置完成
 SETB BUSY
 MOV SBUF, A //写数据到UART数据寄存器
 RET

; /*-----
; 发送字符串
; -----*/
SENDSTRING:
 CLR A
 MOVC A, @A+DPTR //读取字符
 JZ STRINGEND //检测字符串结束标志
 INC DPTR //字符串地址+1
 LCALL SENDDATA //发送当前字符
 SJMP SENDSTRING
STRINGEND:
 RET
; -----
END

```

## 7.5 如何将定时器T2的速度提高12倍

### STC15F104ESW 系列单片机的AUXR寄存器

| SFR name | Address | bit  | B7 | B6 | B5        | B4  | B3              | B2    | B1 | B0 |
|----------|---------|------|----|----|-----------|-----|-----------------|-------|----|----|
| AUXR     | 8EH     | name | -  | -  | UART_M0x6 | T2R | T2_C/ $\bar{T}$ | T2x12 | -  | -  |

定时器0、定时器1和定时器2:

STC15F104ESW系列是1T的8051单片机，定时器2复位后是传统8051的速度，即12分频，但也可不进行12分频，实现真正的1T。

T2x12: 定时器2速度控制位

0，定时器2是传统8051速度，12分频；

1，定时器2的速度是传统8051的12倍，不分频

当串口用T2作为波特率发生器时，则由T2x12决定串口或串口2是12T

UART串口的模式0:

STC15F104ESW系列是1T的8051单片机，为了兼容传统8051，UART串口复位后是兼容传统8051的

UART\_M0x6: 0，UART串口的模式0是传统12T的8051速度，12分频；

1，UART串口的模式0的速度是传统12T的8051的6倍，2分频

如果用定时器T1做波特率发生器时，UART串口的速度由T1的溢出率决定

T2R: 定时器2允许控制位

0，不允许定时器2运行；

1，允许定时器2运行

T2\_C/ $\bar{T}$ : 控制定时器2用作定时器或计数器。

0，用作定时器(对内部系统时钟进行计数)；

1，用作计数器(对引脚T2/P3.1的外部脉冲进行计数)

7.6 可编程时钟输出(也可作分频器使用)

有2路种可编程时钟输出: IRC\_CLKO/P5.4, T2CLKO/P3.0. 只有内部R/C时钟频率为12MHz以下时, 现版本的IRC\_CLKO/P5.4才能正常输出。

7.6.1 与可编程时钟输出有关的特殊功能寄存器

| 符号                | 描述                                                  | 地址  | 位地址及其符号 |     |           |     |          |        |       |       | 复位值        |
|-------------------|-----------------------------------------------------|-----|---------|-----|-----------|-----|----------|--------|-------|-------|------------|
|                   |                                                     |     | MSB     |     |           |     | LSB      |        |       |       |            |
| AUXR              | 辅助寄存器                                               | 8EH | -       | -   | UART_M0x6 | T2R | T2_C/T   | T2x12  | -     | -     | xx00 00xxB |
| INT_CLKO<br>AUXR2 | External Interrupt enable and Clock output register | 8FH |         |     |           |     |          |        |       |       | x000 00xxB |
|                   |                                                     |     | -       | EX4 | EX3       | EX2 | LVD_WAKE | T2CLKO | -     | -     |            |
|                   |                                                     |     |         |     |           |     |          |        |       |       |            |
| IRC_CLKO          | 内部R/C时钟输出寄存器                                        | BBH | -       | -   | -         | -   | -        | -      | IRCS1 | IRCS0 | xxxx xx00B |

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

1. IRC\_CLKO : Internal R/C clock output register (Non bit-addressable)

| SFR Name | SFR Address | bit  | B7 | B6 | B5 | B4 | B3 | B2 | B1    | B0    |
|----------|-------------|------|----|----|----|----|----|----|-------|-------|
| IRC_CLKO | BBH         | name | -  | -  | -  | -  | -  | -  | IRCS1 | IRCS0 |

B7~B2: 保留位。

| B1-IRCS1 | B0-IRCS0 | 内部R/C振荡时钟的输出频率                         |
|----------|----------|----------------------------------------|
| 0        | 0        | 无内部R/C振荡时钟的输出                          |
| 0        | 1        | 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 |
| 1        | 0        | 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 |
| 1        | 1        | 内部R/C振荡时钟的输出频率被4分频, 输出时钟频率 = IRC_CLK/4 |

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。

I/O口的输出速度只能达到15MHz附近。

## 2. INT\_CLKO (AUXR2) : External Interrupt Enable and Clock Output register (Non bit-addressable)

| SFR Name          | SFR Address | bit  | B7 | B6  | B5  | B4  | B3       | B2     | B1 | B0 |
|-------------------|-------------|------|----|-----|-----|-----|----------|--------|----|----|
| INT_CLKO<br>AUXR2 | 8FH         | name | -  | EX4 | EX3 | EX2 | LVD_WAKE | T2CLKO | -  | -  |

B2 - T2CLKO：是否允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

1：允许将P3.0脚配置为定时器2的时钟输出T2CLKO/CLKOUT2，输出时钟频率= $T2\text{溢出率}/2$

如果 $T2\_C/\overline{T}=0$ ，定时器/计数器T2是对内部系统时钟计数，则：

T2工作在1T模式( $AUXR.2/T2x12=1$ )时的输出频率 =  $(SYSclk) / (65536-[RL\_TH2, RL\_TL2])/2$

T2工作在12T模式( $AUXR.2/T2x12=0$ )时的输出频率 =  $(SYSclk) / 12 / (65536-[RL\_TH2, RL\_TL2])/2$

如果 $T2\_C/\overline{T}=1$ ，定时器/计数器T2是对外部脉冲输入(P3. 1/T2)计数，则：

输出时钟频率 =  $(T2\_Pin\_CLK) / (65536-[RL\_TH2, RL\_TL2])/2$

0：不允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

B4 - EX2：允许外部中断2( $\overline{INT2}$ )

B5 - EX3：允许外部中断3( $\overline{INT3}$ )

B6 - EX4：允许外部中断4( $\overline{INT4}$ )

## 3、辅助特殊功能寄存器：AUXR(地址：0x8E)

AUXR : Auxiliary register (不可位寻址)

| SFR name | Address | bit  | B7 | B6 | B5        | B4  | B3                   | B2    | B1 | B0 |
|----------|---------|------|----|----|-----------|-----|----------------------|-------|----|----|
| AUXR     | 8EH     | name | -  | -  | UART_M0x6 | T2R | T2_C/ $\overline{T}$ | T2x12 | -  | -  |

B5 - UART\_M0x6：串口模式0的通信速度设置位。

0：UART串口模式0的速度是传统8051单片机串口的速度，即12分频；

1：UART串口模式0的速度是传统8051单片机串口速度的6倍，即2分频。

B4 - T2R：定时器2运行控制位。

0：不允许定时器2运行；

1：允许定时器2运行。

B3 - T2\_C/ $\overline{T}$ ：控制定时器2用作定时器或计数器。

0，用作定时器(对内部系统时钟进行计数)；

1，用作计数器(对引脚T2/P3. 1的外部脉冲进行计数)

B2 - T2x12：定时器2速度控制位

0，定时器2是传统8051单片机速度，12分频；

1，定时器2的速度是传统8051单片机的12倍，不分频

当串口用T2作为波特率发生器时，则由T2x12决定串口是12T还是1T。

### 7.6.2 内部R/C时钟输出及其测试程序(C和汇编)

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 | -  | -  | -  | -  | -  | -  | IRCS1 | IRCS0 |

如何利用IRC\_CLKO/P5.4管脚输出时钟

IRC\_CLKO/P5.4的时钟输出控制由IRC\_CLKO寄存器的IRCS1和IRCS0位控制。通过设置IRCS1(IRC\_CLKO.1)和IRCS0(IRC\_CLKO.0)可将IRC\_CLKO/P5.4管脚配置为内部R/C振荡时钟输出同时还可以设置该内部R/C振荡时钟的输出频率。

新增加的特殊功能寄存器：IRC\_CLKO（地址：0xBB）

| B1-IRCS1 | B0-IRCS0 | 内部R/C振荡时钟的输出频率                        |
|----------|----------|---------------------------------------|
| 0        | 0        | 无内部R/C振荡时钟的输出                         |
| 0        | 1        | 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 |
| 1        | 0        | 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 |
| 1        | 1        | 内部R/C振荡时钟的输出频率被4分频，输出时钟频率 = IRC_CLK/4 |

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。

I/O口的输出速度只能达到15MHz附近。

下面是内部R/C时钟输出的示例程序：

1. C程序：

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列单片机的内部R/C时钟输出 -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/
//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz
```

```
#include "reg51.h"
```

```
typedef unsigned char BYTE;
typedef unsigned int WORD;
```

```
#define FOSC 18432000L
```

```
//-----

sfr IRC_CLKO = 0xBB; //IRC时钟输出控制寄存器

//-----

void main()
{
 IRC_CLKO = 0x01; //0000,0001 P5.4输出频率为SYSclk
 // IRC_CLKO = 0x02; //0000,0010 P5.4输出频率为SYSclk/2
 // IRC_CLKO = 0x03; //0000,0011 P5.4输出频率为SYSclk/4

 while (1); //程序终止
}
```

## 2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机的内部R/C时钟输出 -----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序, -----*/
/* 请在程序或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

IRC_CLKO DATA 0BBH //IRC时钟输出控制寄存器

;-----
;interrupt vector table

 ORG 0000H
 LJMP MAIN //复位入口
;-----

 ORG 0100H
MAIN:
 MOV SP, #3FH //initial SP
 MOV IRC_CLKO, #01H //0000,0001 P5.4输出频率为SYSclk
// MOV IRC_CLKO, #02H //0000,0010 P5.4输出频率为SYSclk/2
// MOV IRC_CLKO, #03H //0000,0011 P5.4输出频率为SYSclk/4
 SJMP $

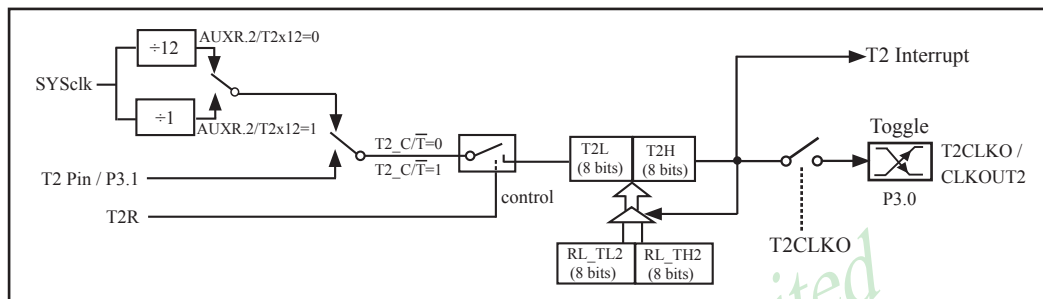
//-----

 END
```

## 7.6.3 定时器2对系统时钟或外部引脚T2的时钟输入进行可编程分频输出 ——及测试程序(C和汇编)

T2可以当定时器用,也可以当串口的波特率发生器和可编程时钟输出。

定时器2的原理框图如下:



定时器/计数器2的工作模式: 16位自动重装

STC创新设计, 请不要再抄袭, 再抄袭就很无耻了

如何利用T2CLKO/P3.0管脚输出时钟

AUXR2.2 - T2CLKO: 是否允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

- 1: 允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2,
- 0: 不允许将P3.0脚配置为定时器2(T2)的时钟输出T2CLKO/CLKOUT2

注意: T2CLKO与CLKOUT2都可表示定时器2(T2)的时钟输出, 下文同。

当T2CLKO/INT\_CLKO.2=1时, P3.0管脚配置为定时器2的时钟输出T2CLKO/CLKOUT2。

输出时钟频率 = T2 溢出率/2

如果T2\_C/T=0, 定时器/计数器T2对内部系统时钟计数, 则:

T2工作在1T模式(AUXR.2/T2x12=1)时的输出时钟频率 = (SYSclk)/(65536-[RL\_TH2, RL\_TL2])/2

T2工作在12T模式(AUXR.2/T2x12=0)时的输出时钟频率 = (SYSclk)/12/(65536-[RL\_TH2, RL\_TL2])/2

如果T2\_C/T=1, 定时器/计数器T2是对外部脉冲输入 (P3.1/T2) 计数, 则:

输出时钟频率 = (T2\_Pin\_CLK) / (65536-[RL\_TH2, RL\_TL2])/2

RL\_TH2为T2H的重装载寄存器, RL\_TL2为T2L的重装载寄存器。

用户在程序中如何具体设置T2CLKO/P3.0管脚输出时钟

1. 对定时器2寄存器T2H/T2L送16位重装载值, [T2H,T2L] = #reload\_data
2. 对AUXR寄存器中的T2R位置1, 让定时器2运行
3. 对AUXR2/INT\_CLKO寄存器中的T2CLKO位置1, 让定时器2的溢出在P3.0口输出时钟。

注意: 当定时器/计数器2用作可编程时钟输出时, 不要允许相应的定时器中断, 免得CPU反复进中断, 在特殊情况下也可允许定时器/计数器2中断。

下面是定时器2对内部系统时钟或外部引脚T2/P3. 1的时钟输入进行可编程时钟分频输出的程序举例(C和汇编)：

### 1. C程序：

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2的可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码, 请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码, 请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18. 432MHz

#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

#define FOSC 18432000L

//-----

sfr AUXR = 0x8e; //辅助特殊功能寄存器
sfr INT_CLKO = 0x8f; //唤醒和时钟输出功能寄存器
sfr T2H = 0xD6; //定时器2高8位
sfr T2L = 0xD7; //定时器2低8位

sbit T2CLKO = P3^0; //定时器2的时钟输出脚

#define F38_4KHz (65536-FOSC/2/38400) //1T模式
//#define F38_4KHz (65536-FOSC/2/12/38400) //12T模式

//-----

void main()
{
 AUXR |= 0x04; //定时器2为1T模式
 // AUXR &= ~0x04; //定时器2为12T模式

```



```
// AUXR &= ~0x08; //T2_C/T=0, 对内部时钟进行时钟输出
// AUXR |= 0x08; //T2_C/T=1, 对T2(P3.1)引脚的外部时钟进行时钟输出

 T2L = F38_4KHz; //初始化计时值
 T2H = F38_4KHz >> 8;

 AUXR |= 0x10; //定时器2开始计时
 INT_CLKO = 0x04; //使能定时器2的时钟输出功能

 while (1); //程序终止
}
```

## 2. 汇编程序:

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- STC15F104ESW 系列 定时器2可编程时钟分频输出举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码, 请在程序中注明使使用了STC的资料及程序 */
/* 如果要在文章中应用此代码, 请在文章中注明使使用了STC的资料及程序 */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

|            |      |        |                                               |
|------------|------|--------|-----------------------------------------------|
| AUXR       | DATA | 08EH   | //辅助特殊功能寄存器                                   |
| INT_CLKO   | DATA | 08FH   | //唤醒和时钟输出功能寄存器                                |
| T2H        | DATA | 0D6H   | //定时器2高8位                                     |
| T2L        | DATA | 0D7H   | //定时器2低8位                                     |
| T2CLKO     | BIT  | P3.0   | //定时器2的时钟输出脚                                  |
| F38_4KHz   | EQU  | 0FF10H | //38.4KHz(1T模式下, 65536-18432000/2/38400)      |
| //F38_4KHz | EQU  | 0FFECH | //38.4KHz(12T模式下, (65536-18432000/2/12/38400) |
| //-----    |      |        |                                               |

```
ORG 0000H
LJMP MAIN //复位入口

//-----

ORG 0100H
MAIN:
MOV SP, #3FH

ORL AUXR, #04H //定时器2为1T模式
// ANL AUXR, #0FBH //定时器2为12T模式

ANL AUXR, #0F7H //T2_C/T=0, 对内部时钟进行时钟输出
// ORL AUXR, #08H //T2_C/T=1, 对T2(P3.1) 引脚的外部时钟进行时钟输出

MOV T2L, #LOW F38_4KHz //初始化计时值
MOV T2H, #HIGH F38_4KHz
ORL AUXR, #10H //定时器2开始计时
MOV INT_CLKO, #04H //使能定时器2的时钟输出功能

SJMP $ //程序终止

;-----

END
```

## 7.7 掉电唤醒专用定时器，进入掉电模式后可将单片机唤醒 ——及测试程序(C和汇编)

STC15F104ESW系列单片机新增了内部掉电唤醒定时器，在进入停机模式/掉电模式后，除了可以通过外部中断源进行唤醒外，还可以在无外部中断源的情况下通过使能内部掉电唤醒定时器定期唤醒CPU，使其恢复到正常工作状态。

STC15F104ESW系列单片机由特殊功能寄存器WKTCH和WKTCL进行管理和控制。

**WKTCL**(不可位寻址)

| SFR name | Address | bit  | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | Reset Value |
|----------|---------|------|----|----|----|----|----|----|----|----|-------------|
| WKTCL    | AAH     | name |    |    |    |    |    |    |    |    | 1111 1110B  |

**WKTCH**(不可位寻址)

| SFR name | Address | bit  | B7    | B6 | B5 | B4 | B3 | B2 | B1 | B0 | Reset Value |
|----------|---------|------|-------|----|----|----|----|----|----|----|-------------|
| WKTCH    | ABH     | name | WKTEN |    |    |    |    |    |    |    | 0111 1111B  |

内部掉电唤醒定时器是一个15位定时器，{WKTCH[6:0], WKTCL[7:0]}构成最长15位计数值(32768个)，定时从0开始计数。

WKTEN：内部停机唤醒定时器的使能控制位。

WKTEN=1，允许内部停机唤醒定时器；

WKTEN=0，禁止内部停机唤醒定时器；

STC15F104ESW系列除增加了特殊功能寄存器WKTCL和WKTCH，还设计了2个隐藏的特殊功能寄存器WKTCL\_CNT和WKTCH\_CNT来控制内部掉电唤醒专用定时器。WKTCL\_CNT与WKTCL共用同一个地址，WKTCH\_CNT与WKTCH共用同一个地址，WKTCL\_CNT和WKTCH\_CNT是隐藏的，对用户不可见。WKTCL\_CNT和WKTCH\_CNT实际上是作计数器使用，而WKTCL和WKTCH实际上作比较器使用。当用户对WKTCL和WKTCH写入内容时，该内容只写入寄存器WKTCL和WKTCH中，而不会写入WKTCL\_CNT和WKTCH\_CNT中。当用户读寄存器WKTCL和WKTCH中的内容时，实际上读的是寄存器WKTCL\_CNT和WKTCH\_CNT中的内容，而不是WKTCL和WKTCH中的内容。

特殊功能寄存器WKTCL\_CNT和WKTCH\_CNT的格式如下所示：

**WKTCL\_CNT**

| SFR name  | Address | bit  | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | Reset Value |
|-----------|---------|------|----|----|----|----|----|----|----|----|-------------|
| WKTCL_CNT | AAH     | name |    |    |    |    |    |    |    |    | 1111 1111B  |

**WKTCH\_CNT**

| SFR name  | Address | bit  | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | Reset Value |
|-----------|---------|------|----|----|----|----|----|----|----|----|-------------|
| WKTCH_CNT | ABH     | name | -  |    |    |    |    |    |    |    | x111 1111B  |

通过软件将WKTCH寄存器中的WKTEN(Power Down Wakeup Timer Enable)位置‘1’，使能内部掉电唤醒专用定时器。一旦MCU进入Power Down Mode, 内部掉电唤醒专用定时器[WKTCH\_CNT, WKTCL\_CNT]就从7FFFH开始计数, 直到计数到与{WKTCH[6:0], WKTCL[7:0]}寄存器所设定的计数值相等后就让系统时钟开始振荡。如果主时钟使用的是内部系统时钟(由用户在ISP烧录程序时自行设置), MCU在等待64个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 就将时钟供给CPU工作。如果主时钟使用的是外部晶体或时钟(由用户在ISP烧录程序时自行设置), MCU在等待1024个时钟后, 就认为此时系统时钟从开始起振的不稳定状态已经过渡到稳定状态, 才将时钟供给CPU工作。CPU获得时钟后, 程序从上次设置单片机进入掉电模式语句的下一条语句开始往下执行。掉电唤醒之后, WKTCH\_CNT和WKTCL\_CNT的内容保持不变, 因此可以通过读[WKTCH, WKTCL]的内容(实际上是读[WKTCH\_CNT, WKTCL\_CNT]的内容)读出单片机在停机模式/掉电模式所等待的时间。

这里请注意: 用户在设置寄存器{WKTCH[6:0], WKTCL[7:0]}的计数值时, 要按照所需要的计数次数, 在计数次数的基础上减1所得的数值才是{WKTCH, WKTCL}的计数值。如用户需计数10次, 则将9写入寄存器{WKTCH[6:0], WKTCL[7:0]}中。同样, 如果用户需计数32768次, 则应对{WKTCH[6:0], WKTCL[7:0]}写入7FFFH(即32767)。

内部定时器计数一次的时间约为488us, 当然误差较大。

内部掉电唤醒专用定时器最短计数时间约为488uS

内部掉电唤醒专用定时器最长计数时间约为 $488\text{us} \times 32768 = 15.99\text{S}$

例如: {设定WKTCH[6:0], WKTCL[7:0]}寄存器的值等于10, 则从系统掉电到启动系统振荡器,

所需要等待的时间为  $488\text{uS} \times 10 = 4880\text{uS}$

设定{WKTCH[6:0], WKTCL[7:0]}寄存器的值等于32768(最大值 =  $32768 = 2^{15}$ ), 则从系统掉电到启动系统振荡器, 所需要等待的时间为  $488\text{uS} \times 32768 = 15.99\text{S}$

|                                   |               |          |
|-----------------------------------|---------------|----------|
| {WKTCH[6:0], WKTCL[7:0]} = 0,     | 488uS x 1     | = 488uS  |
| {WKTCH[6:0], WKTCL[7:0]} = 9,     | 488uS x 10    | = 4.88mS |
| {WKTCH[6:0], WKTCL[7:0]} = 99,    | 488uS x 100   | = 48.8mS |
| {WKTCH[6:0], WKTCL[7:0]} = 999,   | 488uS x 1000  | = 488mS  |
| {WKTCH[6:0], WKTCL[7:0]} = 4095,  | 488uS x 4096  | = 2.0S   |
| {WKTCH[6:0], WKTCL[7:0]} = 32767, | 488uS x 32768 | = 15.99S |

掉电模式功耗: 单片机在掉电模式下的典型功耗为2uA。

如果掉电唤醒定时器被允许(WKTEN=1), 同时用户也将外部中断打开了。进入掉地模式后, 当外部中断提前将单片机从停机模式唤醒时, 可以通过读WKTCL和WKTCH的内容(实际是读WKTCL\_CNT和WKTCH\_CNT中的内容), 可以读出单片机在停机模式/掉电模式等待的时间。

## /\*利用内部专用掉电唤醒定时器来唤醒掉电模式的示例程序（C程序）

## 1. C程序:

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列 掉电唤醒定时器举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序---*/
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序---*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#include "reg51.h"
```

```
#include "intrins.h"
```

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

```
sfr WKTCL = 0xaa;
```

```
sfr WKTCH = 0xab;
```

//掉电唤醒定时器计时低字节

//掉电唤醒定时器计时高字节

```
sbit P10 = P1^0;
```

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

```
void main()
```

```
{
```

```
 WKTCL = 49;
```

//设置唤醒周期为488us\*(49+1) = 24.4ms

```
 WKTCH = 0x80;
```

//使能掉电唤醒定时器

```
 while (1)
```

```
 {
```

```
 PCON = 0x02;
```

//进入掉电模式

```
 nop();
```

```
 nop();
```

```
 P10 = !P10;
```

//掉电唤醒后,取反测试口

```
 }
```

```
}
```

## 2. 汇编程序：

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC15F104ESW 系列 掉电唤醒定时器举例-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序----*/
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
WKTCL DATA 0AAH //掉电唤醒定时器计时低字节
WKTCH DATA 0ABH //掉电唤醒定时器计时高字节

//-----

 ORG 0000H
 LJMP MAIN //复位入口

//-----

MAIN: ORG 0100H

 MOV SP, #3FH

 MOV WKTCL, #49 //设置唤醒周期为488us*(49+1) = 24.4ms
 MOV WKTCH, #80H //使能掉电唤醒定时器

LOOP: MOV PCON, #02H //进入掉电模式
 NOP
 NOP
 CPL P1.0 //掉电唤醒后,取反测试口
 JMP LOOP

 SJMP $

;-----

 END
```

## 第8章 串行口通信

STC15F104ESW系列单片机具有1个采用UART(Universal Asynchronous Receiver/Transmitter)工作方式的全双工串行通信接口。串行口由2个数据缓冲器、一个移位寄存器、一个串行控制寄存器和一个波特率发生器等组成。串行口的数据缓冲器由2个互相独立的接收、发送缓冲器构成，可以同时发送和接收数据。发送缓冲器只能写入而不能读出，接收缓冲器只能读出而不能写入，因而两个缓冲器可以共用一个地址码。串行口的两个缓冲器统称串行通信特殊功能寄存器SBUF，它们共用的地址码是99H；

STC15F104ESW系列单片机的串行口有4种工作方式，其中两种方式的波特率是可变的，另两种是固定的，以供不同应用场合选用。用户可用软件设置不同的波特率和选择不同的工作方式。主机可通过查询或中断方式对接收/发送进行程序处理，使用十分灵活。

STC15F104ESW系列单片机串行口对应的硬件部分是TxD和RxD。串行口可以在多个口之间进行切换。通过设置特殊功能寄存器P\_SW2中的位S1\_S1/P\_SW2.7，可以将串行口从[RxD/P3.0,TxD/P3.1]切换到[RxD\_3/P3.6,TxD\_3/P3.7]。

STC15F104ESW系列单片机的串行通信口，除用于数据通信外，还可方便地构成一个或多个并行I/O口，或作串一并转换，或用于扩展串行外设等。

### 8.1 串行口的相关寄存器

| 符号    | 描述                         | 地址  | 位地址及符号 |       |           |     |        |       |    |     | 复位值        |
|-------|----------------------------|-----|--------|-------|-----------|-----|--------|-------|----|-----|------------|
|       |                            |     | MSB    |       |           |     | LSB    |       |    |     |            |
| T2H   | 定时器2高8位寄存器                 | D6H |        |       |           |     |        |       |    |     | 0000 0000B |
| T2L   | 定时器2低8位寄存器                 | D7H |        |       |           |     |        |       |    |     | 0000 0000B |
| AUXR  | 辅助寄存器                      | 8EH | -      | -     | UART_M0x6 | T2R | T2_C/T | T2x12 | -  | -   | xx00 00xxB |
| SCON  | Serial Control             | 98H | SM0/FE | SM1   | SM2       | REN | TB8    | RB8   | TI | RI  | 0000 0000B |
| SBUF  | Serial Buffer              | 99H |        |       |           |     |        |       |    |     | xxxx xxxxB |
| PCON  | Power Control              | 87H | SMOD   | SMOD0 | LVDF      | POF | GF1    | GF0   | PD | IDL | 0011 0000B |
| IE    | Interrupt Enable           | A8H | EA     | ELVD  | -         | ES  | -      | EX1   | -  | EX0 | 00x0 x0x0B |
| IP    | Interrupt Priority Low     | B8H | -      | PLVD  | -         | PS  | -      | PX1   | -  | PX0 | x0x0 x0x0B |
| SADEN | Slave Address Mask         | B9H |        |       |           |     |        |       |    |     | 0000 0000B |
| SADDR | Slave Address              | A9H |        |       |           |     |        |       |    |     | 0000 0000B |
| P_SW2 | Peripheral function switch | BAH | S1_S1  | -     | -         | -   | -      | -     | -  | -   | 0xxx xxxxB |

1. 串行口的控制寄存器SCON和PCON

STC15F104ESW系列单片机的串行口设有两个控制寄存器：串行控制寄存器SCON和波特率选择特殊功能寄存器PCON。

串行控制寄存器SCON用于选择串行通信的工作方式和某些控制功能。其格式如下：

SCON：串行控制寄存器（可位寻址）

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

SM0/FE：当PCON寄存器中的SMOD0/PCON.6位为1时，该位用于帧错误检测。当检测到一个无效停止位时，通过UART接收器设置该位。它必须由软件清零。

当PCON寄存器中的SMOD0/PCON.6位为0时，该位和SM1一起指定串行通信的工作方式，如下表所示。

其中SM0、SM1按下列组合确定串行口的工作方式：

| SM0                                                                                                                                               | SM1 | 工作方式 | 功能说明           | 波特率                                                                   |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-----|------|----------------|-----------------------------------------------------------------------|
| 0                                                                                                                                                 | 0   | 方式0  | 同步移位串行方式：移位寄存器 | 当UART_M0x6 = 0时，波特率是SYSclk/12，<br>当UART_M0x6 = 1时，波特率是SYSclk / 2      |
| 0                                                                                                                                                 | 1   | 方式1  | 8位UART，波特率可变   | 串行口用定时器2作为其波特率发生器时，<br>波特率=(定时器T2的溢速率)/4。<br>注意：此波特率与SMOD无关。          |
| 1                                                                                                                                                 | 0   | 方式2  | 9位UART         | (2 <sup>SMOD</sup> / 64) x SYSclk系统工作时钟频率                             |
| 1                                                                                                                                                 | 1   | 方式3  | 9位UART，波特率可变   | 串行口用定时器2作为其波特率发生器时，<br>波特率=(定时器1的溢速率或定时器T2的溢速率)/4。<br>注意：此波特率与SMOD无关。 |
| 当AUXR.2/T2x12 = 0时，定时器T2的溢速率 = SYSclk / 12 / ( 65536 - [RL_TH2,RL_TL2] );<br>当AUXR.2/T2x12 = 1时，定时器T2的溢速率 = SYSclk / ( 65536 - [RL_TH2,RL_TL2] ); |     |      |                |                                                                       |

SM2：允许方式2或方式3多机通信控制位。

在方式2或方式3时，如果SM2位为1且REN位为1，则接收机处于地址帧筛选状态。此时可以利用接收到的第9位（即RB8）来筛选地址帧：若RB8=1，说明该帧是地址帧，地址信息可以进入SBUF，并使RI为1，进而在中断服务程序中再进行地址号比较；若RB8=0，说明该帧不是地址帧，应丢掉且保持RI=0。在方式2或方式3中，如果SM2位为0且REN位为1，接收机处于地址帧筛选被禁止状态。不论收到的RB8为0或1，均可使接收到的信息进入SBUF，并使RI=1，此时RB8通常为校验位。

方式1和方式0是非多机通信方式，在这两种方式时，要设置SM2 应为0。

REN：允许/禁止串行接收控制位。由软件置位REN，即REN=1为允许串行接收状态，可启动串行接收器RxD，开始接收信息。软件复位REN，即REN=0，则禁止接收。

TB8：在方式2或方式3，它为要发送的第9位数据，按需要由软件置位或清0。例如，可用作数据的校验位或多机通信中表示地址帧/数据帧的标志位。在方式0和方式1中，该位不用。



**RB8:** 在方式2或方式3, 是接收到的第9位数据, 作为奇偶校验位或地址帧/数据帧的标志位。方式0中不用RB8(置SM2=0). 方式1中也不用RB8(置SM2=0, RB8是接收到的停止位)。

**TI:** 发送中断请求中断标志位。在方式0, 当串行发送数据第8位结束时, 由内部硬件自动置位, 即TI=1, 向主机请求中断, 响应中断后TI必须用软件清零, 即TI=0。在其他方式中, 则在停止位开始发送时由内部硬件置位, 即TI=1, 响应中断后TI必须用软件清零。

**RI:** 接收中断请求标志位。在方式0, 当串行接收到第8位结束时由内部硬件自动置位RI=1, 向主机请求中断, 响应中断后RI必须用软件清零, 即RI=0。在其他方式中, 串行接收到停止位的中间时刻由内部硬件置位, 即RI=1, 向CPU发中断申请, 响应中断后RI必须由软件清零。

SCON的所有位可通过整机复位信号复位为全“0”。SCON的字节地址为98H, 可位寻址, 各位地址为98H~9FH, 可用软件实现位设置。

串行通信的中断请求: 当一帧发送完成, 内部硬件自动置位TI, 即TI=1, 请求中断处理; 当接收完一帧信息时, 内部硬件自动置位RI, 即RI=1, 请求中断处理。由于TI和RI以“或逻辑”关系向主机请求中断, 所以主机响应中断时事先并不知道是TI还是RI请求的中断, 必须在中断服务程序中查询TI和RI进行判别, 然后分别处理。因此, 两个中断请求标志位均不能由硬件自动置位, 必须通过软件清0, 否则将出现一次请求多次响应的错误。

电源控制寄存器PCON中的SMOD/PCON. 7用于设置方式1、方式2、方式3的波特率是否加倍。

电源控制寄存器PCON格式如下:

PCON: 电源控制寄存器 (不可位寻址)

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

**SMOD:** 波特率选择位。当用软件置位SMOD, 即SMOD=1, 则使串行通信方式2的波特率加倍; SMOD=0, 则各工作方式的波特率加倍。复位时SMOD=0。

**SMOD0:** 帧错误检测有效控制位。当SMOD0=1, SCON寄存器中的SM0/FE位用于FE(帧错误检测)功能; 当SMOD0=0, SCON寄存器中的SM0/FE位用于SM0功能, 和SM1一起指定串行口的工作方式。复位时SMOD0=0

## 2. 串行口数据缓冲寄存器SBUF

STC15F系列单片机的串行口缓冲寄存器(SBUF)的地址是99H，实际是2个缓冲器，写SBUF的操作完成待发送数据的加载，读SBUF的操作可获得已接收到的数据。两个操作分别对应两个不同的寄存器，1个是只写寄存器，1个是只读寄存器。

串行通道内设有数据寄存器。在所有的串行通信方式中，在写入SBUF信号(MOV SBUF, A)的控制下，把数据装入相同的9位移位寄存器，前面8位为数据字节，其最低位为移位寄存器的输出位。根据不同的工作方式会自动将“1”或TB8的值装入移位寄存器的第9位，并进行发送。

串行通道的接收寄存器是一个输入移位寄存器。在方式0时它的字长为8位，其他方式时为9位。当一帧接收完毕，移位寄存器中的数据字节装入串行数据缓冲器SBUF中，其第9位则装入SCON寄存器中的RB8位。如果由于SM2使得已接收到的数据无效时，RB8和SBUF中内容不变。

由于接收通道内设有输入移位寄存器和SBUF缓冲器，从而能使一帧接收完将数据由移位寄存器装入SBUF后，可立即开始接收下一帧信息，主机应在该帧接收结束前从SBUF缓冲器中将数据取走，否则前一帧数据将丢失。SBUF以并行方式送往内部数据总线。

## 3. 辅助寄存器AUXR

辅助寄存器AUXR的格式及各位含义如下：

AUXR：辅助寄存器（不可位寻址）

| SFR name | Address | bit  | B7 | B6 | B5        | B4  | B3     | B2    | B1 | B0 |
|----------|---------|------|----|----|-----------|-----|--------|-------|----|----|
| AUXR     | 8EH     | name | -  | -  | UART_M0x6 | T2R | T2_C/T | T2x12 | -  | -  |

UART\_M0x6：串口模式0的通信速度设置位。

0，UART串口模式0的速度是传统8051单片机串口的速度，12分频；

1，UART串口模式0的速度是传统8051单片机串口速度的6倍，2分频

T2R：定时器2允许控制位

0，不允许定时器2运行；

1，允许定时器2运行

T2\_C/T：控制定时器2用作定时器或计数器。

0，用作定时器(对内部系统时钟进行计数)；

1，用作计数器(对引脚T2/P3.1的外部脉冲进行计数)

T2x12：定时器2速度控制位

0，定时器2是传统8051速度，12分频；

1，定时器2的速度是传统8051的12倍，不分频

当串口用T2作为波特率发生器时，则由T2x12决定串口或串口2是12T还是1T。

4. 定时器2的寄存器T2H, T2L

定时器2寄存器T2H(地址为D6H, 复位值为00H)及寄存器T2L(地址为D7H, 复位值为00H)用于保存重装时间常数。

5. 从机地址控制寄存器SADEN和SADDR

为了方便多机通信, STC15F系列单片机设置了从机地址控制寄存器SADEN和SADDR。其中SADEN是从机地址掩模寄存器(地址为B9H, 复位值为00H), SADDR是从机地址寄存器(地址为A9H, 复位值为00H)。

6. 与串行口中断相关的寄存器位ES和PS

串行口中断允许位ES位于中断允许寄存器IE中, 中断允许寄存器的格式如下:

IE: 中断允许寄存器 (可位寻址)

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

EA: CPU的总中断允许控制位, EA=1, CPU开放中断, EA=0, CPU屏蔽所有的中断申请。  
EA的作用是使中断允许形成多级控制。即各中断源首先受EA控制;其次还受各中断源自己的中断允许控制位控制。

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

IP: 中断优先级控制寄存器低 (可位寻址)

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

PS: 串行口中断优先级控制位。  
当PS=0时, 串行口中断为最低优先级中断(优先级0)  
当PS=1时, 串行口中断为最高优先级中断(优先级1)

7. 将串口进行切换的寄存器P\_SW2

| Mnemonic | Add | Name                       | 7     | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Reset Value |
|----------|-----|----------------------------|-------|---|---|---|---|---|---|---|-------------|
| P_SW2    | BAH | Peripheral function switch | S1_S1 | - | - |   |   |   | - | - | 0xxx,xxxx   |

|                                       |                                  |
|---------------------------------------|----------------------------------|
| 串口/S1可在3个地方切换, 由 S1_S0 及 S1_S1 控制位来选择 |                                  |
| S1_S1                                 | 串口/S1可在P1/P3之间来回切换               |
| 0                                     | 串口/S1在[P3. 0/RxD, P3. 1/TxD]     |
| 1                                     | 串口/S1在[P3. 6/RxD_3, P3. 7/TxD_3] |

## 8.2 串行口工作模式

STC15F104ESW系列单片机的串行通信接口有4种工作模式，可通过软件编程对SCON中的SM0、SM1的设置进行选择。其中模式1、模式2和模式3为异步通信，每个发送和接收的字符都带有1个启动位和1个停止位。在模式0中，串行口被作为1个简单的移位寄存器使用。

### 8.2.1 串行口工作模式0：同步移位寄存器(建议初学者不学)

在模式0状态，串行通信接口工作在同步移位寄存器模式，当串行口模式0的通信速度设置位UART\_M0x6/AUXR.5 = 0时，其波特率固定为SYSclk/12。当串行口模式0的通信速度设置位UART\_M0x6/AUXR.5 = 1时，其波特率固定为SYSclk/2。串行口数据由RxD/P3.0端输入，同步移位脉冲（SHIFTCLOCK）由TxD/P3.1输出，发送、接收的是8位数据，低位在先。

模式0的发送过程：当主机执行将数据写入发送缓冲器SBUF指令时启动发送，串行口即将8位数据以SYSclk/12或SYSclk/2(由UART\_M0x6/AUXR.5确定是12分频还是2分频)的波特率从RxD管脚输出(从低位到高位)，发送完中断标志TI置“1”，TxD管脚输出同步移位脉冲（SHIFTCLOCK）。波形如图8-1中“发送”所示。

当写信号有效后，相隔一个时钟，发送控制端SEND有效(高电平)，允许RxD发送数据，同时允许TxD输出同步移位脉冲。一帧(8位)数据发送完毕时，各控制端均恢复原状态，只有TI保持高电平，呈中断申请状态。在再次发送数据前，必须用软件将TI清0。

模式0接收过程：模式0接收时，复位接收中断请求标志RI，即RI=0，置位允许接收控制位REN=1时启动串行模式0接收过程。启动接收过程后，RxD为串行输入端，TxD为同步脉冲输出端。串行接收的波特率为SYSclk/12或SYSclk/2(由UART\_M0x6/AUXR.5确定是12分频还是2分频)。其时序图如图8-1中“接收”所示。

当接收完成一帧数据(8位)后，控制信号复位，中断标志RI被置“1”，呈中断申请状态。当再次接收时，必须通过软件将RI清0

工作于模式0时，必须清0多机通信控制位SM2，使不影响TB8位和RB8位。由于波特率固定为SYSclk/12或SYSclk/2，无需定时器提供，直接由单片机的时钟作为同步移位脉冲。

串行口工作模式0的示意图如图8-1所示

由示意图中可见，由TX和RX控制单元分别产生中断请求信号并置位TI=1或RI=1，经“或门”“送主机请求中断，所以主机响应中断后必须软件判别是TI还是RI请求中断，必须软件清0中断请求标志位TI或RI。

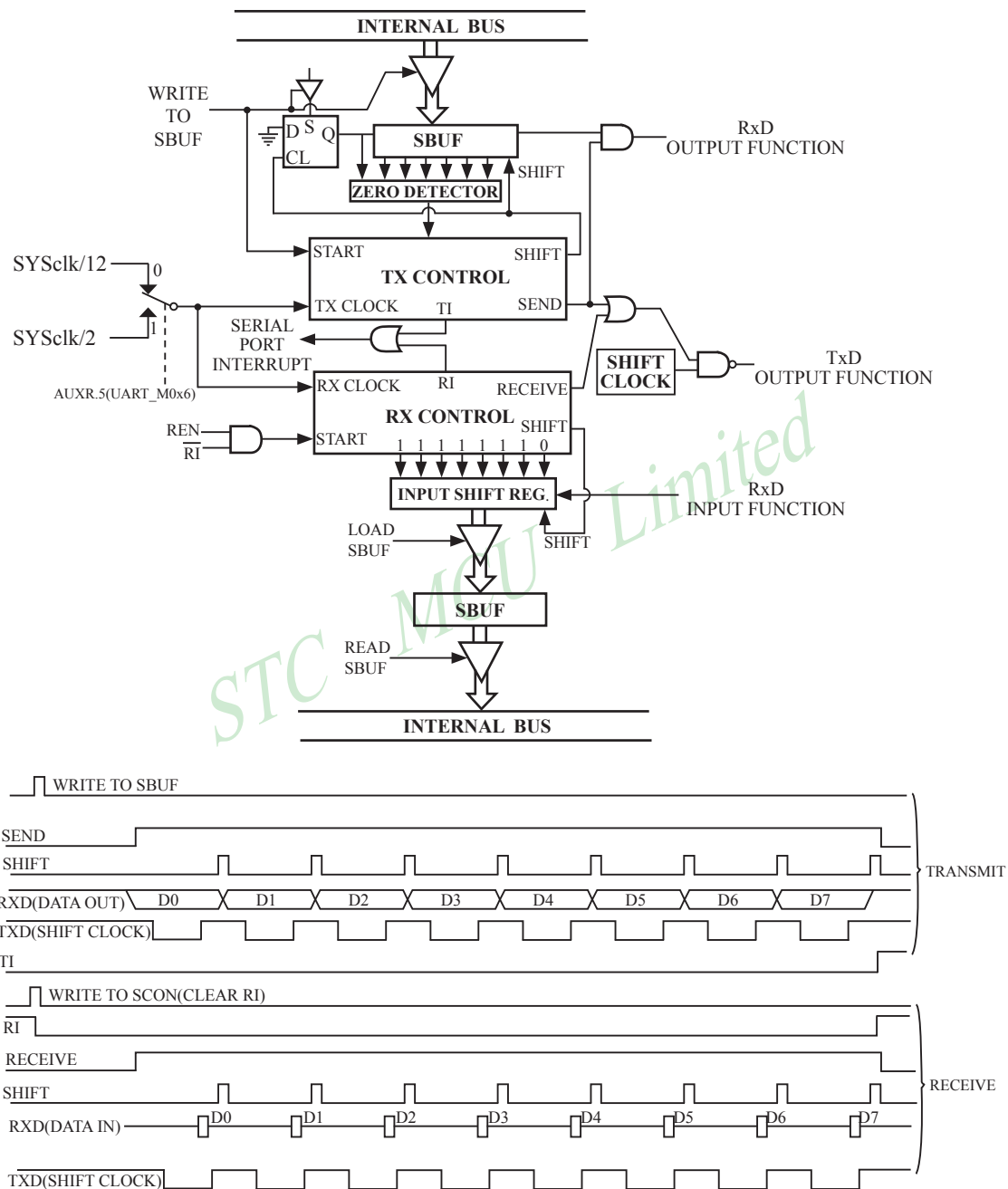


图8-1 串行口模式0功能结构及时序示意图

## 8.2.2 串行口工作模式1：8位UART，波特率可变

当软件设置SCON的SM0、SM1为“01”时，串行口则以模式1工作。此模式为8位UART格式，一帧信息为10位：1位起始位，8位数据位（低位在先）和1位停止位。波特率可变，即可根据需要进行设置。TxD/P3.1为发送信息，RxD/P3.0为接收端接收信息，串行口为全双工接受/发送串行口。

图8-2为串行模式1的功能结构示意图及接收/发送时序图

模式1的发送过程：串行通信模式发送时，数据由串行发送端TxD输出。当主机执行一条写“SBUF”的指令就启动串行通信的发送，写“SBUF”信号还把“1”装入发送移位寄存器的第9位，并通知TX控制单元开始发送。发送各位的定时是由16分频计数器同步。

移位寄存器将数据不断右移送TxD端口发送，在数据的左边不断移入“0”作补充。当数据的最高位移到移位寄存器的输出位置，紧跟其后的是第9位“1”，在它的左边各位全为“0”，这个状态条件，使TX控制单元作最后一次移位输出，然后使允许发送信号“SEND”失效，完成一帧信息的发送，并置位中断请求位TI，即TI=1，向主机请求中断处理。

模式1的接收过程：当软件置位接收允许标志位REN，即REN=1时，接收器便以选定波特率的16分频的速率采样串行接收端口RxD，当检测到RxD端口从“1”→“0”的负跳变时就启动接收器准备接收数据，并立即复位16分频计数器，将1FFH植装入移位寄存器。复位16分频计数器是使它与输入位时间同步。

16分频计数器的16个状态是将1波特率（每位接收时间）均为16等份，在每位时间的7、8、9状态由检测器对RxD端口进行采样，所接收的值是这次采样直经“三中取二”的值，即3次采样至少2次相同的值，以此消除干扰影响，提高可靠性。在起始位，如果接收到的值不为“0”（低电平），则起始位无效，复位接收电路，并重新检测“1”→“0”的跳变。如果接收到的起始位有效，则将它输入移位寄存器，并接收本帧的其余信息。

接收的数据从接收移位寄存器的右边移入，已装入的1FFH向左边移出，当起始位“0”移到移位寄存器的最左边时，使RX控制器作最后一次移位，完成一帧的接收。若同时满足以下两个条件：

- RI=0;
- SM2=0或接收到的停止位为1。

则接收到的数据有效，实现装载入SBUF，停止位进入RB8，置位RI，即RI=1，向主机请求中断，若上述两条件不能同时满足，则接收到的数据作废并丢失，无论条件满足与否，接收器重又检测RxD端口上的“1”→“0”的跳变，继续下一帧的接收。接收有效，在响应中断后，必须由软件清0，即RI=0。通常情况下，串行通信工作于模式1时，SM2设置为“0”。

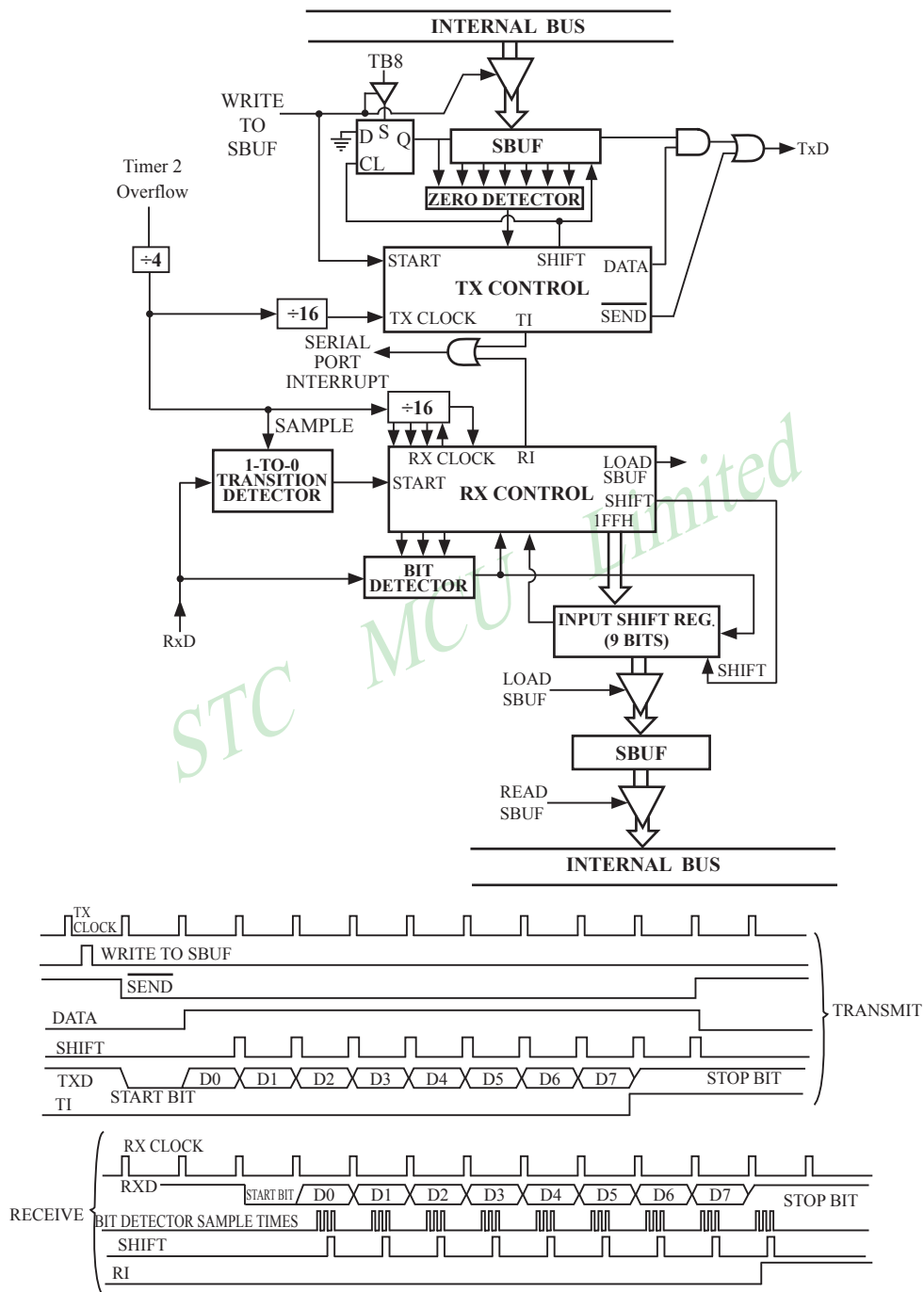


图8-2 串行口模式1功能结构示意图及接收/发送时序图

串行通信模式1的波特率是可变的，可变的波特率由定时器/计数器2产生。

当串行口用定时器2作为其波特率发生器时，

串行口的波特率=(定时器T2的溢出率)/4.

STC创新设计，请不要抄袭，再抄袭就很无耻了

(注意：此时波特率也与SMOD无关。)

当T2工作在1T模式(AUXR.2/T2x12=1)时，定时器2的溢出率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时，串行口的波特率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

当T2工作在12T模式(AUXR.2/T2x12=0)时，定时器2的溢出率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时，串行口的波特率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

RL\_TH2是T2H的自动重装载寄存器，RL\_TL2是T2L的自动重装载寄存器，

STC MCU Limited



### 8.2.3 串行口工作模式2：9位UART，波特率固定(建议不学习)

当SM0、SM1两位为10时，串行口工作在模式2。串行口工作模式2为9位数据异步通信UART模式，其一帧的信息由11位组成：1位起始位，8位数据位(低位在先)，1位可编程位(第9位数据)和1位停止位。发送时可编程位(第9位数据)由SCON中的TB8提供，可软件设置为1或0，或者可将PSW中的奇/偶校验位P值装入TB8(TB8既可作为多机通信中的地址数据标志位，又可作为数据的奇偶校验位)。接收时第9位数据装入SCON的RB8。TxD/P3.1为发送端口，Rx/D/P3.0为接收端口，以全双工模式进行接收/发送。

模式2的波特率为：

串行通信模式2波特率= $2^{SMOD}/64 \times (\text{SYSclk系统工作时钟频率})$

上述波特率可通过软件对PCON中的SMOD位进行设置，当SMOD=1时，选择1/32(SYSclk)；当SMOD=0时，选择1/64(SYSclk)，故而称SMOD为波特率加倍位。可见，模式2的波特率基本上是固定的。

图8-3为串行通信模式2的功能结构示意图及其接收/发送时序图。

由图8-3可知，模式2和模式1相比，除波特率发生源略有不同，发送时由TB8提供给移位寄存器第9数据位不同外，其余功能结构均基本相同，其接收/发送操作过程及时序也基本相同。

当接收器接收完一帧信息后必须同时满足下列条件：

- RI=0
- SM2=0或者SM2=1，并且接收到的第9数据位RB8=1。

当上述两条件同时满足时，才将接收到的移位寄存器的数据装入SBUF和RB8中，并置位RI=1，向主机请求中断处理。如果上述条件有一个不满足，则刚接收到移位寄存器中的数据无效而丢失，也不置位RI。无论上述条件满足与否，接收器又重新开始检测Rx/D输入端口的跳变信息，接收下一帧的输入信息。

在模式2中，接收到的停止位与SBUF、RB8和RI无关。

通过软件对SCON中的SM2、TB8的设置以及通信协议的约定，为多机通信提供了方便。

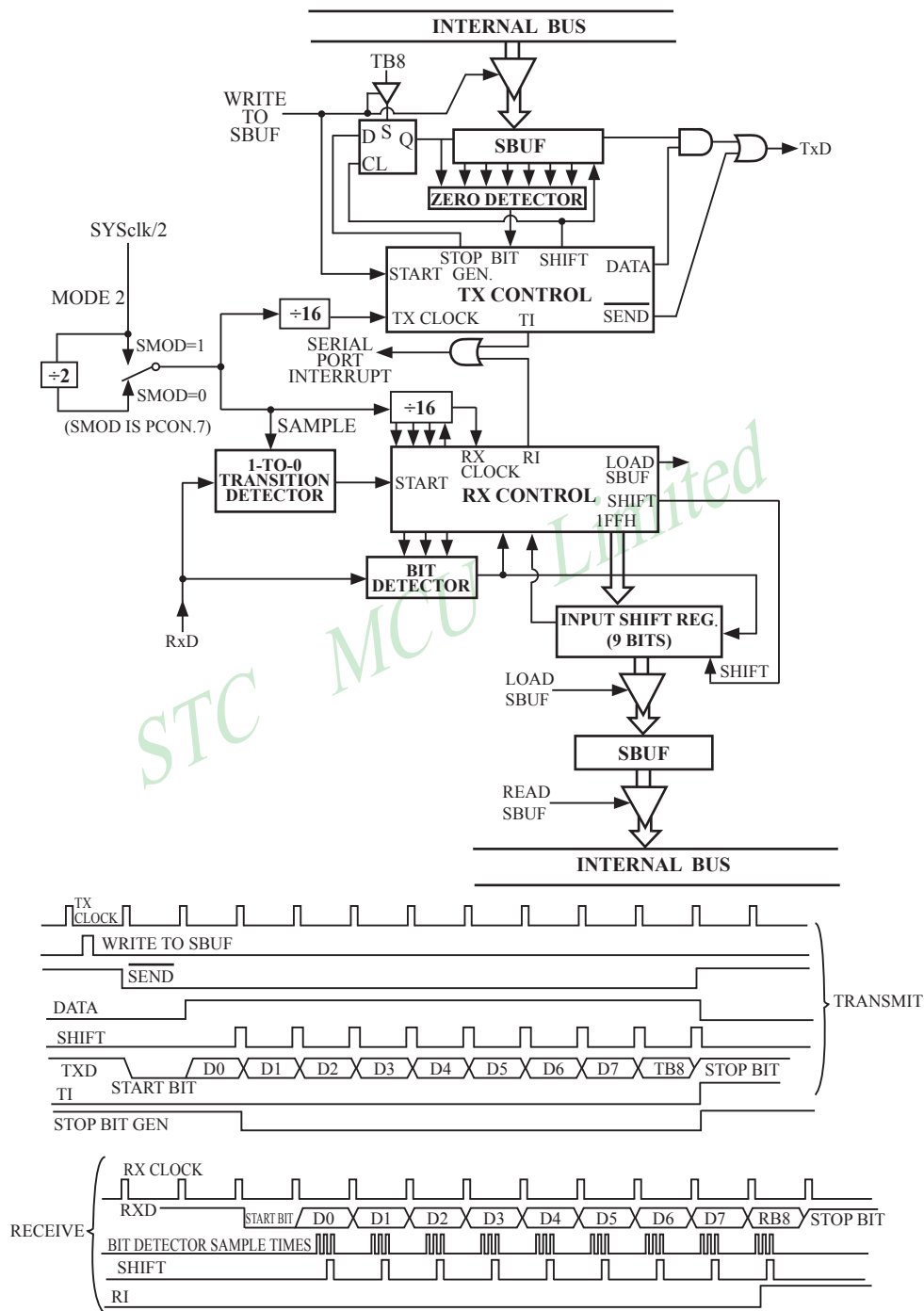


图8-3 串行口模式2功能结构示意图及接收/发送时序图

## 8.2.4 串行口工作模式3：9位UART，波特率可变

当SM0、SM1两位为11时，串行口工作在模式3。串行通信模式3为9位数据异步通信UART模式，其一帧的信息由11位组成：1位起始位，8位数据位（低位在先），1位可编程位（第9位数据）和1位停止位。发送时可编程位（第9位数据）由SCON中的TB8提供，可软件设置为1或0，或者可将PSW中的奇/偶校验位P值装入TB8（TB8既可作为多机通信中的地址数据标志位，又可作为数据的奇偶校验位）。接收时第9位数据装入SCON的RB8。TxD/P3.1为发送端口，RxD/P3.0为接收端口，以全双工模式进行接收/发送。

图8-4为串行口工作模式3的功能结构示意图及其接收/发送时序图。

由图8-4可知，模式3和模式1相比，除发送时由TB8提供给移位寄存器第9数据位不同外，其余功能结构均基本相同，其接收‘发送操作过程及时序也基本相同。

当接收器接收完一帧信息后必须同时满足下列条件：

- RI=0
- SM2=0或者SM2=1，并且接收到的第9数据位RB8=1。

当上述两条件同时满足时，才将接收到的移位寄存器的数据装入SBUF和RB8中，并置位RI=1，向主机请求中断处理。如果上述条件有一个不满足，则刚接收到移位寄存器中的数据无效而丢失，也不置位RI。无论上述条件满足与否，接收器又重新开始检测RxD输入端口的跳变信息，接收下一帧的输入信息。

在模式3中，接收到的停止位与SBUF、RB8和RI无关。

通过软件对SCON中的SM2、TB8的设置以及通信协议的约定，为多机通信提供了方便。

串行通信模式3的波特率也是可变的，可变的波特由定时器/计数器2产生。

模式3的波特率为：

当串行口用定时器2作为其波特率发生器时，

串行口的波特率=(定时器T2的溢出率)/4。

（注意：此时波特率也与SMOD无关。）

STC创新设计，请不要抄袭，再抄袭就很无耻了

当T2工作在1T模式(AUXR.2/T2x12=1)时，定时器2的溢出率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ；

即此时，串行口的波特率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

当T2工作在12T模式(AUXR.2/T2x12=0)时，定时器2的溢出率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ；

即此时，串行口的波特率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}]) / 4$

RL\_TH2是T2H的自动重装载寄存器，RL\_TL2是T2L的自动重装载寄存器，

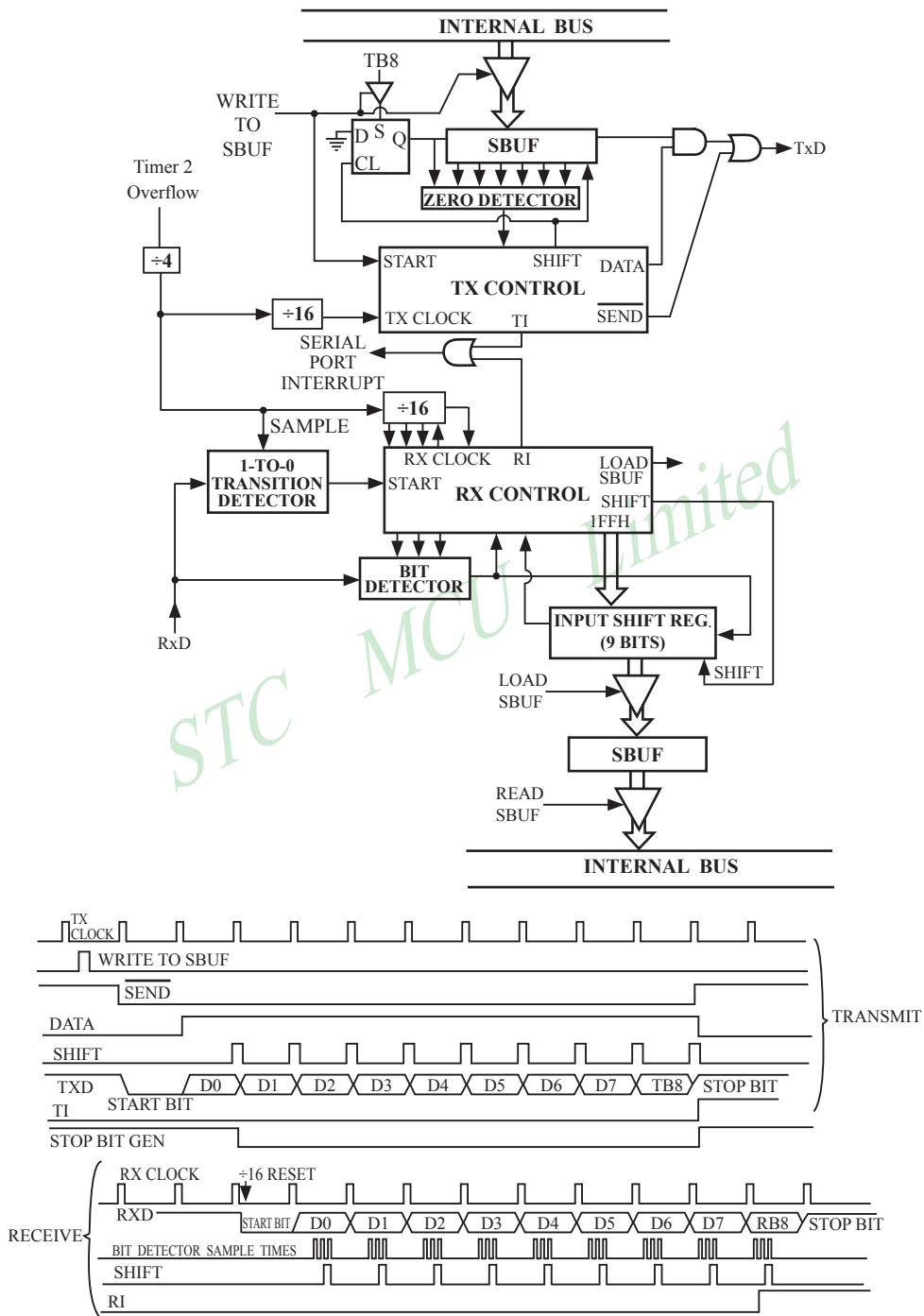


图8-4 串行口模式3功能结构示意图及接收/发送时序图

## 8.3 串行口的波特率的设置

STC15F104ESW系列单片机串行口的波特率随所选工作模式的不同而异,对于工作模式0和模式2,其波特率与系统时钟频率SYSclk和PCON中的波特率选择位SMOD有关,而模式1和模式3的波特率除与SYSclk外,还与定时器/计数器2设置有关。通过对定时器/计数器2的设置,可选择不同的波特率,所以这种波特率是可变的。串行口只能选择定时器2作为其波特率发生器。

串行通信模式0,其波特率与系统时钟频率SYSclk有关。

当模式0的通信速度设置位UART\_M0x6/AUXR.5 = 0时,其波特率 = SYSclk/12。

当模式0的通信速度设置位UART\_M0x6/AUXR.5 = 1时,其波特率 = SYSclk/2。

一旦SYSclk选定且UART\_M0x6/AUXR.5设置好,则串行通信工作模式0的波特率固定不变。

串行通信工作模式2,其波特率除与SYSclk有关外,还与SMOD位有关。

其基本表达式为: 串行通信模式2波特率 =  $2^{\text{SMOD}} / 64 \times (\text{SYSclk系统工作时钟频率})$

当SMOD=1时,波特率 =  $2 / 64 (\text{SYSclk}) = 1 / 32 (\text{SYSclk})$ ;

当SMOD=0时,波特率 =  $1 / 64 (\text{SYSclk})$ 。

当SYSclk选定后,通过软件设置PCON中的SMOD位,可选择两种波特率。所以,这种模式的波特率基本固定。

串行通信模式1和3,其波特率是可变的:

串行口用定时器2作为其波特率发生器,

串行口的波特率 = (定时器T2的溢出率) / 4.

(注意:此时波特率也与SMOD无关。)

当T2工作在1T模式(AUXR.2/T2x12=1)时,定时器2的溢出率 =  $\text{SYSclk} / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时, 串行口的波特率 =  $\text{SYSclk} / (65536 - [\text{T2H}, \text{T2L}]) / 4$

当T2工作在12T模式(AUXR.2/T2x12=0)时,定时器2的溢出率 =  $\text{SYSclk} / 12 / (65536 - [\text{RL\_TH2}, \text{RL\_TL2}])$ ;

即此时, 串行口的波特率 =  $\text{SYSclk} / 12 / (65536 - [\text{T2H}, \text{T2L}]) / 4$

通过对定时器/计数器2的设置,可灵活地选择不同的波特率。在实际应用中多半选用串行模式1或串行模式3。显然,为选择波特率,关键在于定时器/计数器2的溢出率的计算。

STC创新设计, 请不要抄袭, 再抄袭就很无耻了

用户在程序中如何具体使用串口和定时器T2

1. 设置串口的工作模式，SCON寄存器中的SM0和SM1两位决定了串口的4种工作模式。
2. 设置串口的波特率，使用定时器2寄存器T2H及T2L
3. 设置寄存器AUXR中的位T2x12/AUXR. 2, 确定定时器2速度是1T还是12T
4. 启动定时器2，让T2R位为1，T2H/T2L定时器2寄存器就立即开始计数。
5. 设置串口的中断优先级，及打开中断相应的控制位是：

PS, ES, EA

6. 如要串口接收，将REN置1即可

如要串口发送，将数据送入SBUF即可，

接收完成标志RI, 发送完成标志TI, 要由软件清0。

当串口工作在模式1和模式3时，计算相应的波特率需要设置的重装载数, 结果送入T2H/T2L寄存器：

计算自动重装数 RELOAD：

1. 计算 RELOAD

计算公式： $RELOAD = 65536 - INT(SYSc1k/Baud0/4 + 0.5)$

计算出的RELOAD 数直接送T2H/T2L寄存器

式中：INT() 表示取整运算即舍去小数，在式中加 0.5 可以达到四舍五入的目的

SYSc1k = 晶振频率

Baud0 = 标准波特率

2. 设置AUXR. 2/T2x12=1, 定时器2工作在1T模式

3. 计算用 RELOAD 产生的波特率：

$Baud = SYSc1k / (65536 - RELOAD) / 4$

4. 计算误差

$error = (Baud - Baud0) / Baud0 * 100\%$

5. 如果误差绝对值 > 3% 要更换波特率或者更换晶体频率，重复步骤 1-4

例：SYSc1k = 22.1184MHz, Baud0 = 57600

1.  $RELOAD = 65536 - INT(22118400/57600/4 + 0.5)$   
 $= 65536 - INT(96.5)$   
 $= 65536 - 96$   
 $= 65440$   
 $= 0FFA0H$

2. 设置AUXR. 2/T2x12=1, 定时器2工作在1T模式

3.  $Baud = 22118400 / (65536 - 65440) / 4$   
 $= 57600$

4. 误差等于零

例:  $\text{SYSclk} = 12\text{MHz}$ ,  $\text{Baud0} = 57600$

1.  $\text{RELOAD} = 65536 - \text{INT}(12000000/57600/4 + 0.5)$

$= 65536 - \text{INT}(52.0833 + 0.5)$

$= 65536 - \text{INT}(52.5833)$

$= 65536 - 52$

$= 65484$

$= 0\text{FFCCH}$

2. 设置AUXR. 2/T2x12=1, 定时器2工作在1T模式

3.  $\text{Baud} = 12000000/(65536-65484)/4$

$= 57692$

4.  $\text{error} = (57692 - 57600)/57600 * 100\%$

$= 0.16\%$

STC MCU Limited

## 8.4 串行口的测试程序(C和汇编)

### ——定时器2作串口波特率发生器

#### 1. C程序:

```

/*----- */
/* --- STC MCU Limited. ----- */
/* --- STC15F104ESW 系列 定时器2用作串口的波特率发生器举例----- */
/* --- Mobile: (86)13922809991 ----- */
/* --- Fax: 86-755-82905966 ----- */
/* --- Tel: 86-755-82948412 ----- */
/* --- Web: www.STCMCU.com ----- */
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序---- */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序---- */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可----- */
/*----- */

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译
//假定测试芯片的工作频率为18.432MHz

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

typedef unsigned char BYTE;
typedef unsigned int WORD;

#define FOSC 18432000L //系统频率
#define BAUD 115200 //串口波特率

#define NONE_PARITY 0 //无校验
#define ODD_PARITY 1 //奇校验
#define EVEN_PARITY 2 //偶校验
#define MARK_PARITY 3 //标记校验
#define SPACE_PARITY 4 //空白校验

#define PARITYBIT EVEN_PARITY //定义校验位

sfr AUXR = 0x8e; //辅助寄存器
sfr T2H = 0xd6; //定时器2高8位
sfr T2L = 0xd7; //定时器2低8位

sbit P22 = P2^2;

bit b usy;

void SendData(BYTE dat);
void SendString(char *s);

```



```

void main()
{
 #if (PARITYBIT == NONE_PARITY)
 SCON = 0x50; //8位可变波特率
 #elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
 SCON = 0xda; //9位可变波特率,校验位初始为1
 #elif (PARITYBIT == SPACE_PARITY)
 SCON = 0xd2; //9位可变波特率,校验位初始为0
 #endif

 T2L = (65536 - (FOSC/4/BAUD)); //设置波特率重装值
 T2H = (65536 - (FOSC/4/BAUD))>>8;
 AUXR = 0x14; //T2为1T模式,并启动定时器2
 AUXR |= 0x01; //选择定时器2为串口的波特率发生器
 ES = 1; //使能串口中断
 EA = 1;

 SendString("STC15F104ESW\r\nUart Test !\r\n");
 while(1);
}

/*-----
UART 中断服务程序
-----*/
void Uart() interrupt 4 using 1
{
 if (RI)
 {
 RI = 0; //清除RI位
 P0 = SBUF; //P0显示串口数据
 P22 = RB8; //P2.2显示校验位
 }
 if (TI)
 {
 TI = 0; //清除TI位
 busy = 0; //清忙标志
 }
}

/*-----
发送串口数据
-----*/
void SendData(BYTE dat)
{
 while (busy); //等待前面的数据发送完成
 ACC = dat; //获取校验位P (PSW.0)
 if (P) //根据P来设置校验位
 {
 #if (PARITYBIT == ODD_PARITY)

```

```
 TB8 = 0; //设置校验位为0
 #elif (PARITYBIT == EVEN_PARITY)
 TB8 = 1; //设置校验位为1
 #endif
 }
 else
 {
 #if (PARITYBIT == ODD_PARITY)
 TB8 = 1; //设置校验位为1
 #elif (PARITYBIT == EVEN_PARITY)
 TB8 = 0; //设置校验位为0
 #endif
 }
 busy = 1;
 SBUF = ACC; //写数据到UART数据寄存器
}

/*-----
发送字符串
-----*/
void SendString(char *s)
{
 while (*s) //检测字符串结束标志
 {
 SendData(*s++); //发送当前字符
 }
}
```

## 2. 汇编程序:

```

/*----- */
/* --- STC MCU Limited. ----- */
/* --- STC15F104ESW 系列 定时器2用作串口的波特率发生器举例----- */
/* --- Mobile: (86)13922809991 ----- */
/* --- Fax: 86-755-82905966 ----- */
/* --- Tel: 86-755-82948412 ----- */
/* --- Web: www.STCMCU.com ----- */
/* 如果要在程序中使用此代码,请在程序中注明使使用了STC的资料及程序--- */
/* 如果要在文章中应用此代码,请在文章中注明使使用了STC的资料及程序--- */
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可----- */
/*----- */

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

#define NONE_PARITY 0 //无校验
#define ODD_PARITY 1 //奇校验
#define EVEN_PARITY 2 //偶校验
#define MARK_PARITY 3 //标记校验
#define SPACE_PARITY 4 //空白校验

```

```

#define PARITYBIT EVEN_PARITY //定义校验位

```

//-----

```

AUXR EQU 08EH //辅助寄存器
T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位

```

//-----

```

BUSY BIT 20H.0 //忙标志位

```

//-----

```

ORG 0000H
LJMP MAIN

```

```

ORG 0023H
LJMP UART_ISR

```

//-----

```

ORG 0100H

```

MAIN:

```

CLR BUSY
CLR EA
MOV SP, #3FH

```

```

#if (PARITYBIT == NONE_PARITY)

```

```

 MOV SCON, #50H //8位可变波特率

```

```

#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)

```

```

 MOV SCON, #0DAH //9位可变波特率,校验位初始为1
#elif (PARITYBIT == SPACE_PARITY)
 MOV SCON, #0D2H //9位可变波特率,校验位初始为0
#endif

//-----
 MOV T2L, #0D8H //设置波特率重装值(65536-18432000/4/115200)
 MOV T2H, #0FFH
 MOV AUXR, #14H //T2为1T模式,并启动定时器2
 ORL AUXR, #01H //选择定时器2为串口的波特率发生器
 SETB ES //使能串口中断
 SETB EA

 MOV DPTR, #TESTSTR //发送测试字符串
 LCALL SENDSTRING

 SJMP $
;-----
TESTSTR:
 DB "STC15F104ESW Uart1 Test !",0DH,0AH,0

; /*-----
; UART 中断服务程序
; -----*/
UART_ISR:
 PUSH ACC
 PUSH PSW
 JNB RI, CHECKTI //检测RI位
 CLR RI //清除RI位
 MOV P0, SBUF //P0显示串口数据
 MOV C, RB8
 MOV P2.2, C //P2.2显示校验位
CHECKTI:
 JNB TI, ISR_EXIT //检测TI位
 CLR TI //清除TI位
 CLR BUSY //清忙标志
ISR_EXIT:
 POP PSW
 POP ACC
 RETI

; /*-----
; 发送串口数据
; -----*/
SENDDATA:
 JB BUSY, $ //等待前面的数据发送完成
 MOV ACC, A //获取校验位P (PSW.0)
 JNB P, EVEN1INACC //根据P来设置校验位

```

ODDIINACC:

```
#if (PARITYBIT == ODD_PARITY)
 CLR TB8 //设置校验位为0
```

```
#elif (PARITYBIT == EVEN_PARITY)
 SETB TB8 //设置校验位为1
```

```
#endif
```

```
 SJMP PARITYBITOK
```

EVENIINACC:

```
#if (PARITYBIT == ODD_PARITY)
 SETB TB8 //设置校验位为1
```

```
#elif (PARITYBIT == EVEN_PARITY)
 CLR TB8 //设置校验位为0
```

```
#endif
```

PARITYBITOK: //校验位设置完成

```
 SETB BUSY
```

```
 MOV SBUF, A //写数据到UART数据寄存器
```

```
 RET
```

```
;/*-----
```

;发送字符串

```
//-----*/
```

SENDSTRING:

```
 CLR A
```

```
 MOVC A, @A+DPTR //读取字符
```

```
 JZ STRINGEND //检测字符串结束标志
```

```
 INC DPTR //字符串地址+1
```

```
 LCALL SENDDATA //发送当前字符
```

```
 SJMP SENDSTRING
```

STRINGEND:

```
 RET
```

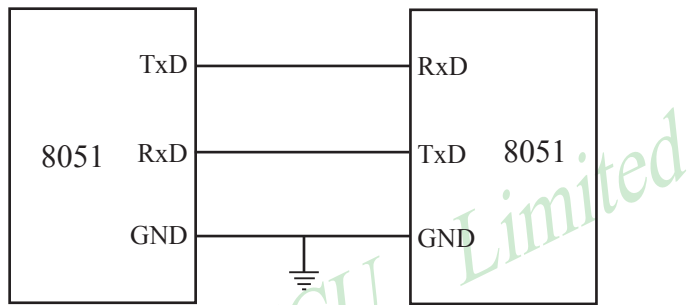
```
//-----
```

```
END
```

8.5 双机通信

STC15F104ESW系列单片机的串行通信根据其应用可分为双机通信和多机通信两种。下面先介绍双机通信。

如果两个8051应用系统相距很近，可将它们的串行端口直接相连（TXD—RXD，RXD—TXD，GND—GND—地），即可实现双机通信。为了增加通信距离，减少通道及电源干扰，可采用RS—232C或RS—422、RS—485标准进行双机通信，两通信系统之间采用光—电隔离技术，以减少通道及电源的干扰，提高通信可靠性。



为确保通信成功，通信双方必须在软件上有系列的约定通常称为软件通信“协议”。现举例简介双机异步通信软件“协议”如下：

通信双方均选用2400波特的传输速率，设系统的主频SYSclk=6MHz,甲机发送数据，乙机接收数据。在双机开始通信时，先由甲机发送一个呼叫信号（例如“06H”），以询问乙机是否可以接收数据；乙机接收到呼叫信号后，若同意接收数据，则发回“00H”作为应答信号，否则发“05H”表示暂不能接收数据，；甲机只有在接收到乙机的应答信号“00H”后才可将存储在外数据存储器中的内容逐一发送给乙机，否则继续向乙机发呼叫信号，直到乙机同意接收。其发送数据格式如下：

|      |     |     |     |     |     |       |
|------|-----|-----|-----|-----|-----|-------|
| 字节数n | 数据1 | 数据2 | 数据3 | ... | 数据n | 累加校验和 |
|------|-----|-----|-----|-----|-----|-------|

字节数n：甲机向乙机发送的数据个数；

数据1~数据n：甲机将向乙机发送的n帧数据；

累加校验和：为字节数n、数据1、...、数据n,这(n+1)个字节内容的算术累加和。

乙机根据接收到的“校验和”判断已接收到的n个数据是否正确。若接收正确,向甲机回发“0FH”信号,否则回发“FOH”信号。甲机只有在接收到乙机发回的“0FH”信号才算完成发送任务，返回被调用的程序，否则继续呼叫，重发数据。

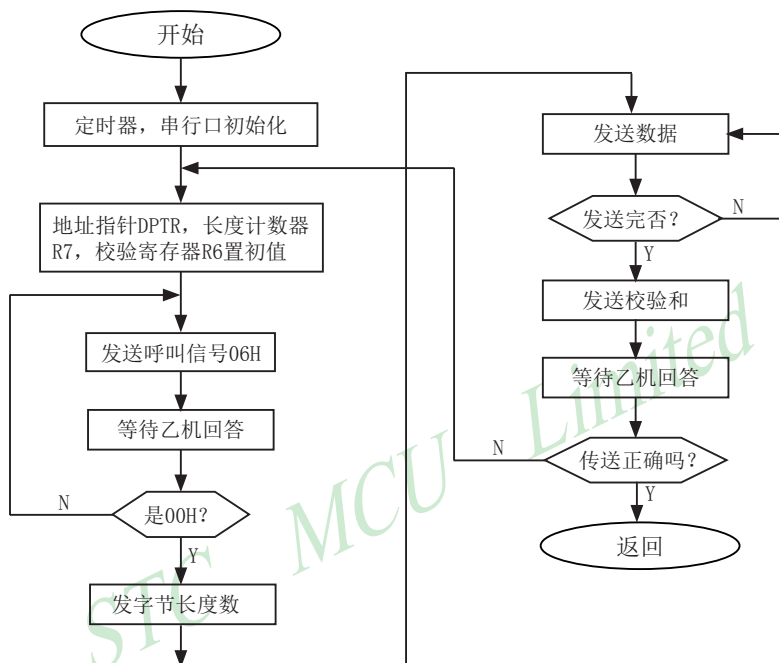
不同的通信要求，软件“协议”内容也不一样，有关需甲、乙双方共同遵守的约定应尽量完善，以防止通信不能正确判别而失败。

STC15F系列单片机的串行通信，可直接采用查询法，也可采用自动中断法。

## (1) 查询方式双机通信软件举例

## ①甲机发送子程序段

下图为甲机发送子程序流程图。



甲机发送程序设置:

- 波特率设置: 选用定时器/计数器1定时模式、工作方式2, 计数常数F3H, SMOD=1。波特率为2400 (位/秒);
- 串行通信设置: 异步通信方式1, 允许接收;
- 内部RAM和工作寄存器设置: 31H和30H单元存放发送的数据块首地址; 2FH单元存放发送的数据块个数; R6为累加和寄存器。

甲机发送子程序清单：

START:

```

MOV TMOD, #20H ; 设置定时器/计数器1定时、工作方式2
MOV TH1, #0F3H ; 设置定时计数常数
MOV TL1, #0F3H ;
MOV SCON, #50H ; 串口初始化
MOV PCON, #80H ; 设置SMOD=1
SETB TR1 ; 启动定时

```

ST-RAM:

```

MOV DPH, 31H ; 设置外部RAM数据指针
MOV DPL, 30H ; DPTR初值
MOV R7, 2FH ; 发送数据块数送R7
MOV R6, #00H ; 累加和寄存器R6清0

```

TX-ACK:

```

MOV A, #06H ;
MOV SBUF, A ; } 发送呼叫信号“06H”

```

WAIT1:

```

JBC T1, RX-YES ; 等待发送完呼叫信号
SJMP WAIT1 ; 未发送完转WAIT1

```

RX-YES:

```

JBC RI, NEXT1 ; 判断乙机回答信号
SJMP RX-YES ; 未收到回答信号，则等待

```

NEXT1:

```

MOV A, SBUF ; 接收回答信号送A
CJNE A, #00H, TX-ACK ; 判断是否“00H”，否则重发呼叫信号

```

TX-BYT:

```

MOV A, R7 ;
MOV SBUF, A ; } 发送数据块数n
ADD A, R6
MOV R6, A

```

WAIT2:

```

JBC T1, TX-NES ;
JMP WAIT2 ; } 等待发送完

```

TX-NES:

```

MOVX A, @DPTR ; 从外部RAM取发送数据
MOV SBUF, A ; 发送数据块
ADD A, R6
MOV R6, A
INC DPTR ; DPTR指针加1

```



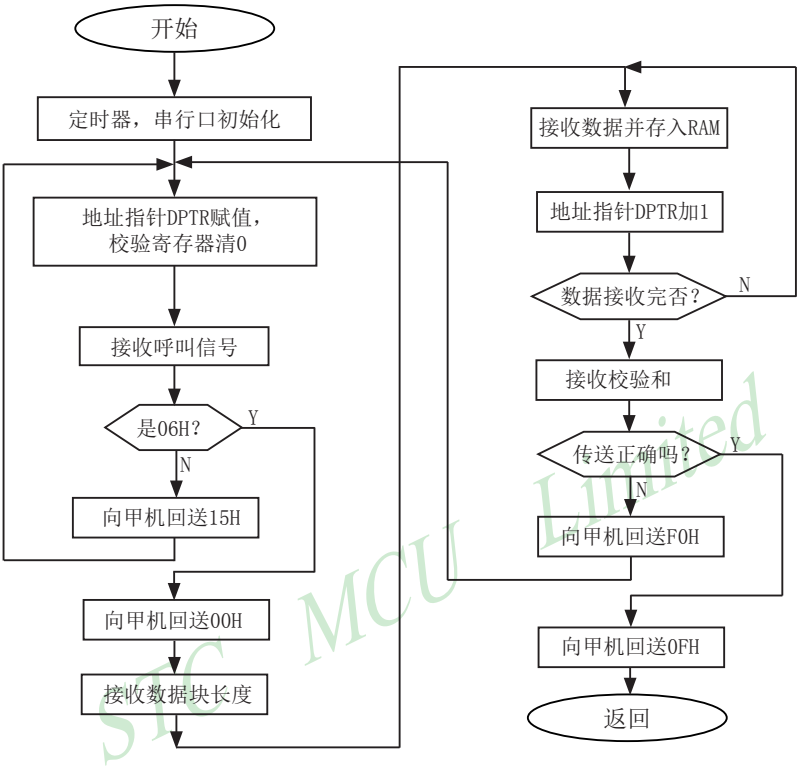
```
WAIT3:
 JBC TI, NEXT2 ; 判断一数据块发送完否
 SJMP WAIT3 ; 等待发送完
NEXT2:
 DJNZ R7, TX-NES ; 判断发送全部结束否
TX-SUM:
 MOV A, R6 ; 发送累加和给乙机
 MOV SBUF, A
WAIT4:
 JBC TI, RX-0FH ;
 SJMP WAIT4 ; } 等待发送完
RX-0FH:
 JBC RI, IF-0FH ;
 SJMP RX-0FH ; } 等待接收乙机回答信号
IF-0FH:
 MOV A, SBUF; ;
 CJNE A, #0FH, ST-RAM ; } 判断传输是否正确, 否则重新发送
 RET ; 返回
```

#### 乙机接收子程序段

接收程序段的设置:

- (a) 波特率设置初始化: 同发送程序;
- (b) 串行通信初始化: 同发送程序;
- (c) 寄存器设置:
  - 内部RAM 31H、30H单元存放接收数据缓冲区首地址。
  - R7——数据块个数寄存器。
  - R6——累加和寄存器。
- (d) 向甲机回答信号: “0FH”为接收正确, “F0H”为传送出错, “00H”为同意接收数据, “05H”为暂不接收。

下图为双机通信查询方式乙机接收子程序流程图。



接收子程序清单:

TART:

|      |       |       |                  |
|------|-------|-------|------------------|
| MOV  | TMOD, | #20H  |                  |
| MOV  | TH1,  | #0F3H |                  |
|      |       |       | ;                |
|      |       |       | }                |
| MOV  | TL1,  | #0F3H | ; 定时器/计数器1设置     |
| SETB | TR1   |       | ; 启动定时器/计数器1     |
| MOV  | SCON, | #50H  | ; 置串行通信方式1, 允许接收 |

ST-RAM:

|     |       |      |             |
|-----|-------|------|-------------|
| MOV | PCON, | #80H |             |
|     |       |      | ; } SMOD置位  |
| MOV | DPH,  | 31H  |             |
|     |       |      | ; }         |
| MOV | DPL,  | 30H  | ; 设置DPTR首地址 |
| MOV | R6,   | #00H | ; 校验和寄存器清0  |

RX-ACK:

JBC RI, IF-06H ; 判断接收呼叫信号  
 SJMP RX-ACK ; 等待接收呼叫信号

IF-06H:

MOV A, SBUF ; 呼叫信号送A  
 CJNEA #06H, TX-05H ; 判断呼叫信号正确否?

TX-00H:

MOV A, #00H ;  
 MOV SBUF, A ; } 向甲机发送“00H”，同意接收

WAIT1:

JBC TI, RX-BYS ; 等待应答信号发送完  
 SJMP WAIT1

TX-05H:

MOV A, #05H ; 向甲机发送“05H”呼叫  
 MOV SBUF, A ; 不正确信号

WAIT2:

JBC TI, HAVE1 ; 等待发送完  
 SJMP WAIT2

HAVE1:

LJMP RX-ACK ; 因呼叫错，返回重新接收呼叫

RX-BYS:

JBC RI, HAVE2 ; 等待接收数据块个数  
 SJMP RX-BYS ;

HAVE2:

MOV A, SBUF ;  
 MOV R7, A ; 数据块个数帧送R7,R6  
 MOV R6, A ;

RX-NES:

JBC RI, HAVE3 ;  
 SJMP RX-NES ; } 接收数据帧

HAVE3:

MOV A, SBUF ;  
 MOVX @DPTR, A ; 接收到的数据存入外部RAM  
 INC DPTR ;  
 ADD A, R6 ;  
 MOV R6, A ; } 形成累加和  
 DJNZ R7, RX-NES ; 判断数据是否接收完

RX-SUM:

```

 JBC RI, HAVE4 ;
 SJMP RX-SUM ; } 等待接收校验和

```

HAVE4:

```

 MOV A, SBUF ;
 CJNE A, R6, TX-ERR ; } 判断传输是否正确

```

TX-RIT:

```

 MOV A, #0FH ;
 MOV SBUF, A ; } 向甲机发送接收正确信息

```

WAIT3:

```

 JBC TI, GOOD ;
 SJMP WAIT3 ; } 等待发送结束

```

TX-ERR:

```

 MOV A, #0F0H ; 向甲机发送传输有误信号
 MOV SBUF, A

```

WAIT4:

```

 JBC TI, AGAIN ; 等待发送完
 SJMP WAIT4

```

AGAIN:

```

 LJMP ST-RAM ; 返回重新开始接收

```

GOOD:

```

 RET ; 传输正确返回

```

## (2) 中断方式双机通信软件举例

在很多应用场合，双机通信的双方或一方采用中断方式以提高通信效率。由于STC15F系列单片机的串行通信是双工的，且中断系统只提供一个中断矢量入口地址，所以实际上是中断和查询必须相结合，即接收/发送均可各自请求中断，响应中断时主机并不知道是谁请求中断，统一转入同一个中断矢量入口，必须由中断服务程序查询确定并转入对应的服务程序进行处理。

这里，任以上述协议为例，甲方（发送方）任以查询方式通信（从略），乙方（接收方）则改用中断—查询方式进行通信。

在中断接收服务程序中，需设置三个标志位来判断所接收的信息是呼叫信号还是数据块个数，是数据还是校验和。增设寄存器：内部RAM32H单元为数据块个数寄存器，33H单元为校验和寄存器，位地址7FH、7EH、7DH为标志位。

## 乙机接收中断服务程序清单

采用中断方式时,应在主程序中安排定时器/计数器、串行通信等初始化程序。通信接收的数据存放在外部RAM的首地址也需在主程序中确定。

主程序:

```
ORG 0000H
AJMP START ; 转至主程序起始处
ORG 0023H
LIMP SERVE ; 转中断服务程序处
.
.
.
```

START:

```
MOV TMOD, #20H ; 定义定时器/计数器1定时、工作方式2
MOV TH1, #0F3H ;
MOV TL1, #0F3H ; } 设置波特率为2400位/秒
MOV SCON, #50H ; 设置串行通信方式1, 允许接收
MOV PCON, #80H ; 设置SMOD=1
SETB TR1 ; 启动定时器
SETB 7FH ;
SETB 7EH ; 设置标志位为1
SETB 7DH ;
MOV 31H, #10H ; 规定接收的数据存储于外部RAM的
MOV 30H, #00H ; } 起始地址1000H
MOV 33H, #00H ; 累加和单元清0
SETB EA ;
SETB ES ; } 开中断
.
.
.
```

中断服务程序：

SERVE:

```

CLR EA ; 关中断
CLR RI ; 清除接收中断请求标志
PUSH DPH ;
PUSH DPL ; 现场保护
PUSH A ;
JB 7FH, RXACK ; 判断是否是呼叫信号
JB 7EH, RXBYS ; 判断是否是数据块数据
JB 7DH, RXDATA ; 判断是否是接收数据帧

```

RXSUM:

```

MOV A, SBUF ; 接收到的校验和
CJNE A, 33H, TXERR ; 判断传输是否正确

```

TXRI:

```

MOV A, #0FH ;
MOV SBUF, A ; } 向甲机发送接收正确信号“0FH”

```

WAIT1:

```

JNB TI, WAIT1 ; 等待发送完毕
CLR TI ; 清除发送中断请求标志位
SJMP AGAIN ; 转结束处理

```

TXERR:

```

MOV A, #0F0H ;
MOV SBUF, A ; } 向甲机发送接收出错信号“F0H”

```

WAIT2:

```

JNB TI, WAIT2 ; 等待发送完毕
CLR TI ; 清除发送中断请求标志
SJMP AGAIN ; 转结束处理

```

RXACK:

```

MOV A, SBUF ; 判断是否是呼叫信号“06H”
XRL A, #06H ; 异或逻辑处理
JZ TXREE ; 是呼叫，则转TXREE

```

TXNACK:

```

MOV A, #05H ; 接收到的不是呼叫信号，则向甲机发送
MOV SBUF, A ; “05H”，要求重发呼叫

```

## WAIT3:

```

JNB TI, WAIT3 ; 等待发送结束
CLR TI
SJMP RETURN ; 转恢复现场处理

```

## TXREE:

```

MOV A, #00H ; 接收到的是呼叫信号, 发送“00H”
MOV SBUF, A ; 接收到的是呼叫信号, 发送“00H”

```

## WAIT4:

```

JNB TI, WAIT4 ; 等待发送完毕
CLR TI ; 清除TI标志
CLR 7FH ; 清除呼叫标志
SJMP RETURN ; 转恢复现场处理

```

## RXBYS:

```

MOV A, SBUF ; 接收到数据块数
MOV 32H, A ; 存入32H单元
ADD A, 33H ; }
MOV 33H, A ; } 形成累加和
CLR 7EH ; 清除数据块数标志
SJMP RETURN ; 转恢复现场处理

```

## RXDATA:

```

MOV DPH, 31H ; }
MOV DPL, 30H ; } 设置存储数据地址指针
MOV A, SBUF ; 读取数据帧
MOVX @DPTR, A ; 将数据存外部RAM
INC DPTR ; 地址指针加1
MOV 31H, DPH ; }
MOV 30H, DPL ; } 保存地址指针值
ADD A, 33H ; }
MOV 33H, A ; } 形成累加和
DJNZ 32H, RETURN ; 判断数据接收完否
CLR 7DH ; 清数据接收完标志
SJMP RETURN ; 转恢复现场处理

```

AGAIN:

|      |           |   |                |
|------|-----------|---|----------------|
| SETB | 7FH       | ; |                |
| SETB | 7EH       | ; | 恢复标志位          |
| SETB | 7DH       | ; |                |
| MOV  | 33H, #00H | ; | 累加和单元清0        |
| MOV  | 31H, #10H | ; | } 恢复接收数据缓冲区首地址 |
| MOV  | 30H, #00H | ; |                |

RETURN:

|      |     |   |      |
|------|-----|---|------|
| POP  | A   | ; |      |
| POP  | DPL | ; | 恢复现场 |
| POP  | DPH | ; |      |
| SETB | EA  | ; | 开中断  |
| RET1 |     | ; | 返回   |

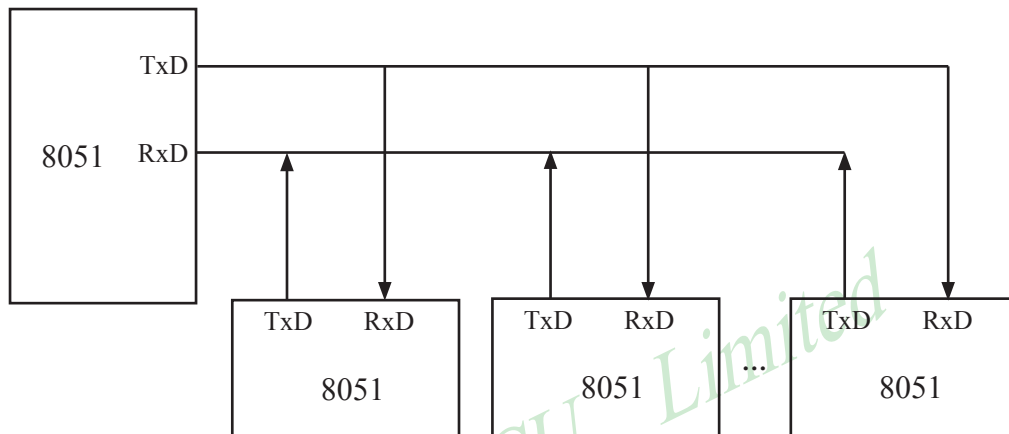
上述程序清单中，ORG为程序段说明伪指令，在程序汇编时，它向汇编程序说明该程序段的起始地址。

在实际应用中情况多种多样，而且是两台独立的计算机之间进行信息传输。因此，应周密考虑通信协议，以保证通信的正确性和成功率



## 8.6 多机通信

在很多实际应用系统中，需要多台微计算机协调工作。STC15F系列单片机的串行通信方式2和方式3具有多机通信功能，可构成各种分布式通信系统。下图为全双工主从式多机通信系统的连接框图。



上图为一台主机和几台从机组成的全双工多机通信系统。主机可与任一从机通信，而从机之间的通信必须通过知己转发。

### （1）多机通信的基本原理

在多机通信系统中，为保证主机（发送）与多台从机（接收）之间能可靠通信，串行通信必须具备识别能力。MCS-51系列单片机的串行通信控制寄存器SCON中设有多机通信选择位SM2。当程序设置SM2=1，串行通信工作于方式2或方式8，发送端通过对TB8的设置以区别于发送的是地址帧（TB8=1）还是数据帧（TB8=0），接收端通过对接收到RB8进行识别：当SM2=1，若接收到RB8=1，则被确认为呼叫地址帧，将该帧内容装入SBUF中，并置位RI=1，向CPU请求中断，进行地址呼叫处理；若RB8=0为数据帧，将不予理睬，接收的信息被丢弃。若SM2=0，则无论是地址帧还是数据帧均接收，并置位RI=1，向CPU请求中断，将该帧内容装入SBUF。据此原理，可实现多机通信。

对于上图的从机式多机通信系统，从机的地址为0，1，2，…，n。实现多机通信的过程如下：

- ① 置全部从机的SM2=1，处于只接收地址帧状态。
- ② 主机首先发送呼叫地址帧信息，将TB8设置为1，以表示发送的是 呼叫地址帧。

③ 所有从机接收到呼叫地址帧后，各自将接收到的主机呼叫的地址与本机的地址相比较：若比较结果相等，则为被寻址从机，清除SM2=0，准备接收从主机发送的数据帧，直至全部数据传输完；若比较不相等，则为非寻址从机，任维持SM2=1不变，对其后发来的数据帧不予理睬，即接收到的数据帧内容不装入SBUF，不置位，RI=0，不会产生中断请求，直至被寻址为止。

- ④ 主机在发送完呼叫地址帧后，接着发送一连串的数据帧，其中的TB8=0，以表示为数据帧。
- ⑤ 当主机改变从机通信时间则再发呼叫地址帧，寻呼其他从机，原先被寻址的从机经分析得知主机在寻呼其他从机时，恢复其SM2=1，对其后主机发送的数据帧不予理睬。

上述过程均在软件控制下实现。

## （2）多机通信协议简述

由于串行通信是在二台或多台各自完全独立的系统之间进行信息传

输这就需要根据时间通信要求制定某些约定，作为通信规范遵照执行，协议要求严格、完善，不同的通信要求，协议的内容也不相同。在多机通信系统中要考虑的问题较多，协议内容比较复杂。这里仅例举几条作一说明。

上图的主从式多机通信系统，允许配置255台从机，各从机的地址分别为00H~FEH。

- ① 约定地址FFH为全部从机的控制命令，命令各从机恢复SM2=1状态，准备接收主机的地址呼叫。
- ② 主机和从机的联络过程约定：主机首先发送地址呼叫帧，被寻址的从机回送本机地址给主机，经验证地址相符后主机再向被寻址的从机发送命令字，被寻址的从机根据命令字要求回送本机的状态，若主机判断状态正常，主机即开始发送或接收数据帧，发送或接收的第一帧为传输数据块长度。
- ③ 约定主机发送的命令字为：
- 00H：要求从机接收数据块；
- 01H：要求从机发送数据块；
- ：
- ：
- ：

其他：非法命令。

- ④ 从机的状态字格式约定为：

| B7  | B6 | B5 | B4 | B3 | B2 | B1   | B0   |
|-----|----|----|----|----|----|------|------|
| ERR | 0  | 0  | 0  | 0  | 0  | TRDY | RRDY |

定义：若ERR=1，从机接收到非法命令；

若TRDY=1，从机发送准备就绪；

若RRDY=1，从机接收准备就绪；

- ⑤ 其他：如传输出错措施等。

### (3) 程序举例

在实际应用中如传输波特率不太高，系统实时性有一定要求以及希望提高通信效率，则多半采用中断控制方式，但程序调试较困难，这就要求提高程序编制的正确性。采用查询方式，则程序调试较方便。这里仅以中断控制方式为例简单介绍主—从机之间一对一通信软件。

#### ① 主机发送程序

该主机要发送的数据存放在内部RAM中，数据块的首地址为51H，数据块长度存放做50H单元中，有关发送前的初始化、参数设置等采用子程序格式，所有信息发送均由中断服务程序完成。当主机需要发送时，在完成发送子程序的调用之后，随即返回主程序继续执行。以后只需查询PSW·5的F0标志位的状态即可知道数据是否发送完毕。

要求主机向#5从机发送数据，中断服务程序选用工作寄存器区1的R0~R7。

主机发送程序清单：

```
ORG 0000H
AJMP MAIN ; 转主程序
ORG 0023H ; 发送中断服务程序入口
LJMP SERVE ; 转中断服务程序
:
:
MAIN: ; 主程序
:
:
ORG 1000H ; 发送子程序入口
TXCALL:
MOV TMOD, #20H ; 设置定时器/计数器1定时、方式2
MOV TH1, #0F3H ; 设置波特率为2400位/秒
MOV TL1, #0F3H ; 置位SMOD
MOV PCON, #80H ;
SETB TR1 ; 启动定时器/计数器1
MOV SCON, #0D8H ; 串行方式8，允许接收，TB8=1
SETB EA ; 开中断总控制位
CLR ES ; 禁止串行通信中断
TXADDR:
MOV SBUF, #05H ; 发送呼叫从机地址
WAIT1:
JNB TI, WAIT1 ; 等待发送完毕
CLR TI ; 复位发送中断请求标志
```

## RXADDR:

|      |       |              |                        |
|------|-------|--------------|------------------------|
| JNB  | RI,   | RXADDR       | ; 等待从机回答本机地址           |
| CLR  | TI    |              | ; 复位接收中断请求标志           |
| MOV  | A,    | SBUF         | ; 读取从机回答的本机地址          |
| CJNE | A,    | #05H, TXADDR | ; 判断呼叫地址符否, 否则重发       |
| CLR  | TB8   |              | ; 地址相符, 复位TB8=0, 准备发数据 |
| CLR  | PSW.5 |              | ; 复位F0=0标志位            |
| MOV  | 08H,  | #50H         | ; 发送数据地址指针送R0          |
| MOV  | 0CH,  | 50H          | ; 数据块长度送R4             |
| INC  | 0CH   |              | ; 数据块长度加1              |
| SETB | ES    |              | ; 允许串行通信中断             |
| RET  |       |              | ; 返回主程序                |

⋮

## SERVE:

|      |     |                      |
|------|-----|----------------------|
| CLR  | TI  | ; 中断服务程序段, 清中断请求标志TI |
| PUSH | PSW |                      |
| PUSH | A   | } 现场入栈保护             |
| CLR  | RS1 |                      |
| SETB | RS0 | } 选择工作寄存器区1          |
|      |     |                      |

## TXDATA:

|     |       |     |              |
|-----|-------|-----|--------------|
| MOV | SBUF, | @R0 | ; 发送数据块长度及数据 |
|-----|-------|-----|--------------|

## WAIT2:

|      |       |        |                |
|------|-------|--------|----------------|
| JNB  | TI,   | WAIT2  | ; 等待发送完毕       |
| CLR  | TI    |        | ; 复位TI=0       |
| INC  | R0    |        | ; 地址指针加1       |
| DJNZ | R4,   | RETURN | ; 数据块未发送完, 转返回 |
| SETB | PSW.5 |        | ; 已发送完毕置位F0=1  |
| CLR  | ES    |        | ; 关闭串行中断       |

## RETURN:

|      |     |        |
|------|-----|--------|
| POP  | A   | } 恢复现场 |
| POP  | PSW |        |
| RETI |     |        |

## ②从机接收程序

主机发送的地址呼叫帧，所有的从机均接收，若不是呼叫本机地址即从中断返回；若是本机地址，则回送本机地址给主机作为应答，并开始接收主机发送来的数据块长度帧，并存放于内部RAM的60H单元中，紧接着接收的数据帧存放于61H为首地址的内部RAM单元中，程序中还选用20·0H、20·1H位作标志位，用来判断接收的是地址、数据块长度还是数据，选用了2FH、2EH两个字节单元用于存放数据字节数和存储数据指针。#5从机的接收程序如下，供参考。

#5从机接收程序清单：

```

ORG 0000H
AJMP START ; 转主程序段
ORG 0023H
LJMP SERVE ; 从中断入口转中断服务程序
ORG 0100H

START:
MOV TMOD, #20H ; 主程序段：初始化程序，设置定时
MOV TH1, #0F3H ; 器/计数器1定时、工作方式2，设
MOV TL1, #0F3H ; 置波特率为2400位/秒的有关初值
MOV PCON, #80H ; 置位SMOD
MOV SCON, #0F0H ; 设置串行方式3，允许接收，SM2=1
SETB TR1 ; 启动定时器/计数器1
SETB 20·0 ; }
SETB 20·1 ; } 置标志位为1
SETB EA ; }
SETB ES ; } 开中断
:
:
ORG 1000H

SERVE:
CLR RI ; 清接收请求中断标志RI=0
PUSH A ; }
PUSH PSW ; } 现场保护
CLR RS1 ; }
SETB RS0 ; } 选择工作寄存器区1
JB 20·0H, ISADDR ; 判断是否是地址帧
JB 20·1H, ISBYTE ; 判断是否是数据块长度帧

```

## ISDATA:

```

MOV R0, 2EH ; 数据指针送R0
MOV A, SBUF ; 接收数据
MOV @R0, A
INC 2EH ; 数据指针加1
DJNZ 2FH, RETURN ; 判断数据接收完否?
SETB 20·0H ;
SETB 20·1H ; 恢复标志位
SETB SM2 ;
SJMP RETURN ; 转入恢复现场，返回

```

## ISADDR:

```

MOV A, SBUF ; 是地址呼叫，判断与本机地址
 }
CJNE A, #05H, RETURN ; 相符否，不符则转返回
MOV SBUF, #01H ; 相符，发回答信号“01H”

```

## WAIT:

```

JNB TI, WAIT ; 等待发送结束
CLR TI
CLR 20·0H ; 清0TI, 20·0, SM2
CLR SM2 ; 清0TI, 20·0, SM2
SJMP RETURN ; 转返回

```

## ISBYTES:

```

MOV A, SBUF ; 接收数据块长度帧
MOV R0, #60H ;
MOV @R0, A ; 将数据块长度存入内部RAM
MOV 2FH, A ; 60H单元及2FH单元
MOV 2EH, #61H ; 置首地址61H于2EH单元
CLR 20·1H ; 清20·1H标志，表示以后接收的为数据

```

## RETURN:

```

POP PSW ; }
POP A ; } 恢复现场
RETI ; 返回

```

多机通信方式可多种多样，上例仅以最简单的住一从式作了简单介绍，仅供参考。

对于串行通信工作方式0的同步方式，常用于通过移位寄存器进行扩展并行I/O口，或配置某些串行通信接口的外部设备。例如，串行打印机、显示器等。这里就不一一举例了。

## 第9章 STC15F104ESW系列单片机EEPROM的应用

STC15F104ESW系列单片机内部集成了EEPROM，其与程序空间是分开的。利用ISP/IAP技术可将内部Data Flash当EEPROM，擦写次数在10万次以上。EEPROM可分为若干个扇区，每个扇区包含512字节。使用时，建议同一次修改的数据放在同一个扇区，不是同一次修改的数据放在不同的扇区，不一定要用满。数据存储器的擦除操作是按扇区进行的。

EEPROM可用于保存一些需要在应用过程中修改并且掉电不丢失的参数数据。在用户程序中，可以对EEPROM进行字节读/字节编程/扇区擦除操作。在工作电压Vcc偏低时，建议不要进行EEPROM/IAP操作。

### 9.1 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 | SMOD   | SMOD0 | LVDF  | POF      | GF1 | GF0 | PD  | IDL | 0011 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，IAP15F106SW/IAP15L106SW除外，IAP15F106SW/IAP15L106SW可在用户应用程序区修改用户应用程序区。  
特别声明：EEPROM也可以用MOVC指令读(MOVC访问的是程序存储器)，但起始地址不再是0000H, 而是程序存储空间结束地址的下一个地址。

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。

在每次触发前，需重新送字节读/字节编程/扇区擦除命令，在命令不改变时, 不需重新送命令



## 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<br>(2个时钟)     | Program/编程<br>(=55us) | Sector Erase<br>扇区擦除<br>(=21ms) | Recommended System Clock<br>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 | SMOD | SMOD0 | LVDF | POF | GF1 | GF0 | PD | IDL |

LVDF: 低压检测标志位, 当工作电压Vcc低于低压检测门槛电压时，该位置1。该位要由软件清0

当低压检测电路发现工作电压Vcc偏低时，不要进行EEPROM/IAP操作。

5V单片机的低压检测门槛电压：

| -40 °C | 25 °C | 85 °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 °C | 25 °C | 85 °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  |

## 9.2 STC15F104ESW系列单片机EEPROM空间大小及地址

| STC15F104ESW系列单片机内部EEPROM选型一览表<br>STC15L104ESW系列单片机内部EEPROM选型一览表 |           |     |                       |                        | STC15F104ESW系列单片机内部EEPROM还可以用MOVC指令读，但此时首地址不再是0000H，而是程序存储空间结束地址的下一个地址 |
|------------------------------------------------------------------|-----------|-----|-----------------------|------------------------|------------------------------------------------------------------------|
| 型号                                                               | EEPROM字节数 | 扇区数 | 用IAP字节读时EEPROM起始扇区首地址 | 用IAP字节读时EEPROM结束扇区末尾地址 |                                                                        |
| STC15F101ESW<br>STC15L101ESW                                     | 2K        | 4   | 0000h                 | 07FFh                  |                                                                        |
| STC15F102ESW<br>STC15L102ESW                                     | 2K        | 4   | 0000h                 | 07FFh                  |                                                                        |
| STC15F103ESW<br>STC15L103ESW                                     | 2K        | 4   | 0000h                 | 07FFh                  |                                                                        |
| STC15F104ESW<br>STC15L104ESW                                     | 1K        | 2   | 0000h                 | 03FFh                  |                                                                        |
| STC15F105ESW<br>STC15L105ESW                                     | 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                        |           |     |                       |                        |                                                                        |
| IAP15F106SW<br>IAP15L106SW                                       | -         | 12  | 0000h                 | 17FFh                  | 没有专门的EEPROM，但可在程序区修改程序，使用时不要将自己的有效程序擦除。                                |

| STC15F2K20S2单片机的内部EEPROM地址表<br>STC15L2K20S2单片机的内部EEPROM地址表 |       |       |       |       |       |       |       | 每个扇区<br>512字节<br><br>建议同一次修改的数据放在同一扇区，不是同一次修改的数据放在不同的扇区，不必用满，当然可全用 |
|------------------------------------------------------------|-------|-------|-------|-------|-------|-------|-------|--------------------------------------------------------------------|
| 第一扇区                                                       |       | 第二扇区  |       | 第三扇区  |       | 第四扇区  |       |                                                                    |
| 起始地址                                                       | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  |                                                                    |
| 0000h                                                      | 1FFh  | 200h  | 3FFh  | 400h  | 5FFh  | 600h  | 7FFh  |                                                                    |
| 第五扇区                                                       |       | 第六扇区  |       | 第七扇区  |       | 第八扇区  |       |                                                                    |
| 起始地址                                                       | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  |                                                                    |
| 800h                                                       | 9FFh  | A00h  | BFFh  | C00h  | DFFh  | E00h  | FFFh  |                                                                    |
| 第九扇区                                                       |       | 第十扇区  |       | 第十一扇区 |       | 第十二扇区 |       |                                                                    |
| 起始地址                                                       | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  | 起始地址  | 结束地址  |                                                                    |
| 1000h                                                      | 11FFh | 1200h | 13FFh | 1400h | 15FFh | 1600h | 17FFh |                                                                    |

## 9.3 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,<br>;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,也可以用MOV指令读,但起始地址不再是0000H,而是程序存储空间结束地址的下一个地址

|     |            |                    |              |                           |
|-----|------------|--------------------|--------------|---------------------------|
| MOV | IAP_ADDRH, | #BYTE_ADDR_HIGH    | ;送地址高字节      | } 地址需要改变时<br>才需重新送地址      |
| MOV | IAP_ADDRL, | #BYTE_ADDR_LOW     | ;送地址低字节      |                           |
| MOV | IAP_CONTR, | #WAIT_TIME         | ;设置等待时间      | } 此两句可以合成一句,<br>并且只送一次就够了 |
| ORL | IAP_CONTR, | #10000000B         | ;允许ISP/IAP操作 |                           |
| MOV | IAP_CMD,   | #ISP_IAP_BYTE_READ |              |                           |

;送字节读命令,现有A版本每次触发前需重新送命令。

;在命令不需改变时,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FF,指向非EEPROM区
;MOV IAP_ADDRL, #0FFh ;送地址低字节单元为FF,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
MOV IAP_ADDRL, #BYTE_ADDR_LOW ;送地址低字节 } 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
ORL IAP_CONTR, #10000000B ;允许ISP/IAP操作 } 此两句可合成一句,并且只送一次就够了

MOV IAP_CMD, #ISP_IAP_BYTE_PROGRAM
 ;送字节编程命令,现有A版本每次触发前需重新送命令。
 ;在命令不需改变时,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 ;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FF,指向非EEPROM区,防止误操作
;MOV IAP_ADDRL, #0FFh ;送地址低字节单元为FF,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
ORL IAP_CONTR, #10000000B ;允许ISP/IAP
 } 此两句可以合成一句,并且只
MOV IAP_CMD, #ISP_IAP_SECTOR_ERASE ;送扇区擦除命令, 现有A版本每次触发前需重新送命令。
 ;在命令不需改变时, 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FF, 指向非EEPROM区
;MOV IAP_ADDRL, #0FFh ;送地址低字节单元为FF, 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。

## 9.4 EEPROM测试程序(C和汇编)

### 9.4.1 EEPROM测试程序(不用串口送出数据)(C和汇编)

#### 1. C程序:

;**STC15F104ESW系列单片机EEPROM/IAP 功能测试程序演示**

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机 EEPROM/IAP功能-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序, -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
void IapIdle();
```

```
BYTE IapReadByte(WORD addr);
```

```
#include "reg51.h"
```

```
#include "intrins.h"
```

```
typedef unsigned char BYTE;
```

```
typedef unsigned int WORD;
```

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

```
sfr IAP_DATA = 0xC2; //IAP数据寄存器
sfr IAP_ADDRH = 0xC3; //IAP地址寄存器高字节
sfr IAP_ADDRL = 0xC4; //IAP地址寄存器低字节
sfr IAP_CMD = 0xC5; //IAP命令寄存器
sfr IAP_TRIG = 0xC6; //IAP命令触发寄存器
sfr IAP_CONTR = 0xC7; //IAP控制寄存器
```

```
#define CMD_IDLE 0 //空闲模式
#define CMD_READ 1 //IAP字节读命令
#define CMD_PROGRAM 2 //IAP字节编程命令
#define CMD_ERASE 3 //IAP扇区擦除命令
```

```
//#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

//测试地址
#define IAP_ADDRESS 0x0400

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

void main()
{
 WORD i;

 P1 = 0xfe; //1111,1110 系统OK
 Delay(10); //延时
 IapEraseSector(IAP_ADDRESS); //扇区擦除
 for (i=0; i<512; i++) //检测是否擦除成功(全FF检测)
 {
 if (IapReadByte(IAP_ADDRESS+i) != 0xff)
 goto Error; //如果出错, 则退出
 }
 P1 = 0xfc; //1111,1100 擦除成功
 Delay(10); //延时
 for (i=0; i<512; i++) //编程512字节
 {
 IapProgramByte(IAP_ADDRESS+i, (BYTE)i);
 }
 P1 = 0xf8; //1111,1000 编程完成
 Delay(10); //延时
 for (i=0; i<512; i++) //校验512字节
 {
 if (IapReadByte(IAP_ADDRESS+i) != (BYTE)i)
 goto Error; //如果校验错误, 则退出
 }
 P1 = 0xf0; //1111,0000 测试完成
 while (1);

Error:
 P1 &= 0x7f; //0xxx,xxxx IAP操作失败
 while (1);
}

/*-----
软件延时
-----*/

```

```

void Delay(BYTE n)
{
 WORD x;

 while (n--)
 {
 x = 0;
 while (++x);
 }
}

/*-----
关闭IAP
-----*/
void IapIdle()
{
 IAP_CONTR = 0; //关闭IAP功能
 IAP_CMD = 0; //清除命令寄存器
 IAP_TRIG = 0; //清除触发寄存器
 IAP_ADDRH = 0x80; //将地址设置到非IAP区域
 IAP_ADDRL = 0;
}

/*-----
从ISP/IAP/EEPROM区域读取一字节
-----*/
BYTE IapReadByte(WORD addr)
{
 BYTE dat; //数据缓冲区

 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_READ; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 nop(); //等待ISP/IAP/EEPROM操作完成
 dat = IAP_DATA; //读ISP/IAP/EEPROM数据
 IapIdle(); //关闭IAP功能

 return dat; //返回
}

```

/\*-----\*/

写一字节数据到ISP/IAP/EEPROM区域

-----\*/

void IapProgramByte(WORD addr, BYTE dat)

```
{
 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_PROGRAM; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_DATA = dat; //写ISP/IAP/EEPROM数据
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 _nop(); //等待ISP/IAP/EEPROM操作完成
 IapIdle();
}
```

/\*-----\*/

扇区擦除

-----\*/

void IapEraseSector(WORD addr)

```
{
 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_ERASE; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 _nop(); //等待ISP/IAP/EEPROM操作完成
 IapIdle();
}
```

## 2. 汇编程序：

## ;STC15F104ESW系列单片机EEPROM/IAP 功能测试程序演示

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机 EEPROM/IAP功能-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序或文章中注明使用了STC的资料及程序 -----*/
/*--- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

IAP_DATA EQU 0C2H //IAP数据寄存器
IAP_ADDRH EQU 0C3H //IAP地址寄存器高字
IAP_ADDRL EQU 0C4H //IAP地址寄存器低字
IAP_CMD EQU 0C5H //IAP命令寄存器
IAP_TRIG EQU 0C6H //IAP命令触发寄存器
IAP_CONTR EQU 0C7H //IAP控制寄存器

```

```

CMD_IDLE EQU 0 //空闲模式
CMD_READ EQU 1 //IAP字节读命令
CMD_PROGRAM EQU 2 //IAP字节编程命令
CMD_ERASE EQU 3 //IAP扇区擦除命令

```

```

;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

```

//测试地址

IAP\_ADDRESS EQU 0400H

//-----

```

ORG 0000H
LJMP MAIN

```

;-----

```

ORG 0100H

```

MAIN:

```

MOV P1, #0FEH //1111,1110 系统OK
LCALL DELAY //延时

```

```

;-----
MOV DPTR, #IAP_ADDRESS //设置ISP/IAP/EEPROM地址
LCALL IAP_ERASE //扇区擦除
;-----
MOV DPTR, #IAP_ADDRESS //设置ISP/IAP/EEPROM地址
MOV R0, #0 //检测512字节
MOV R1, #2
CHECK1: //检测是否擦除成功(全FF检测)
LCALL IAP_READ //读IAP数据
CJNE A, #0FFH, ERROR //如果出错,则退出
INC DPTR //IAP地址+1
DJNZ R0, CHECK1
DJNZ R1, CHECK1
;-----
MOV P1, #0FCH //1111,1100 擦除成功
LCALL DELAY //延时
;-----
MOV DPTR, #IAP_ADDRESS //设置ISP/IAP/EEPROM地址
MOV R0, #0 //编程512字节
MOV R1, #2
MOV R2, #0
NEXT:
MOV A, R2 //准备数据
LCALL IAP_PROGRAM //字节编程
INC DPTR //IAP地址+1
INC R2 //修改测试数据
DJNZ R0, NEXT
DJNZ R1, NEXT
;-----
MOV P1, #0F8H //1111,1000 编程完成
LCALL DELAY //延时
;-----
MOV DPTR,#IAP_ADDRESS //设置ISP/IAP/EEPROM地址
MOV R0,#0 //校验512字节
MOV R1,#2
MOV R2,#0
CHECK2:
LCALL IAP_READ //读IAP数据
CJNE A, 2, ERROR //如果出错,则退出
INC DPTR //IAP地址+1
INC R2
DJNZ R0, CHECK2
DJNZ R1, CHECK2
;-----
MOV P1, #0F0H //1111,0000 测试完成
SJMP $
;-----

```

ERROR:

```

MOV P0, R0
MOV P2, R1
MOV P3, R2
CLR P1.7 //0xxx,xxxx IAP 测试失败
SJMP $

```

/\*-----\*/

软件延时

-----\*/

DELAY:

```

CLR A
MOV R0, A
MOV R1, A
MOV R2, #20H

```

DELAY1:

```

DJNZ R0, DELAY1
DJNZ R1, DELAY1
DJNZ R2, DELAY1
RET

```

/\*-----\*/

关闭IAP

-----\*/

IAP\_IDLE:

```

MOV IAP_CONTR, #0 //关闭IAP功能
MOV IAP_CMD, #0 //清除命令寄存器
MOV IAP_TRIG, #0 //清除触发寄存器
MOV IAP_ADDRH, #80H //将地址设置到非IAP区域
MOV IAP_ADDRL, #0
RET

```

/\*-----\*/

从ISP/IAP/EEPROM区域读取一字节

-----\*/

IAP\_READ:

```

MOV IAP_CONTR, #ENABLE_IAP //使能IAP
MOV IAP_CMD, #CMD_READ //设置IAP命令
MOV IAP_ADDRL,DPL //设置IAP低地址
MOV IAP_ADDRH, DPH //设置IAP高地址
MOV IAP_TRIG, #5AH //写触发命令(0x5a)
MOV IAP_TRIG, #0A5H //写触发命令(0xa5)
NOP //等待ISP/IAP/EEPROM操作完成
MOV A, IAP_DATA //读IAP数据

```

```
LCALL IAP_IDLE //关闭IAP功能
RET

/*-----
写一字节数据到ISP/IAP/EEPROM区域
-----*/
IAP_PROGRAM:
MOV IAP_CONTR, #ENABLE_IAP //使能IAP
MOV IAP_CMD, #CMD_PROGRAM //设置IAP命令
MOV IAP_ADDRL, DPL //设置IAP低地址
MOV IAP_ADDRH, DPH //设置IAP高地址
MOV IAP_DATA, A //写IAP数据
MOV IAP_TRIG, #5AH //写触发命令(0x5a)
MOV IAP_TRIG, #0A5H //写触发命令(0xa5)
NOP //等待ISP/IAP/EEPROM操作完成
LCALL IAP_IDLE //关闭IAP功能
RET

/*-----
扇区擦除
-----*/
IAP_ERASE:
MOV IAP_CONTR, #ENABLE_IAP //使能IAP
MOV IAP_CMD, #CMD_ERASE //设置IAP命令
MOV IAP_ADDRL, DPL //设置IAP低地址
MOV IAP_ADDRH, DPH //设置IAP高地址
MOV IAP_TRIG, #5AH //写触发命令(0x5a)
MOV IAP_TRIG, #0A5H //写触发命令(0xa5)
NOP //等待ISP/IAP/EEPROM操作完成
LCALL IAP_IDLE //关闭IAP功能
RET

END
```

## 9.4.2 EEPROM测试程序(使用串口送出数据)(C和汇编)

### 1. C程序:

;STC15F104ESW系列单片机EEPROM/IAP 功能测试程序演示

```
/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机 EEPROM/IAP功能-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序, -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*--- 在 Keil C 开发环境中, 选择 Intel 8052 编译即可-----*/
/*-----*/
```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```
#include "reg51.h"
```

```
#include "intrins.h"
```

```
typedef unsigned char BYTE;
```

```
typedef unsigned int WORD;
```

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

```
sfr IAP_DATA = 0xC2; //IAP数据寄存器
sfr IAP_ADDRH = 0xC3; //IAP地址寄存器高字节
sfr IAP_ADDRL = 0xC4; //IAP地址寄存器低字节
sfr IAP_CMD = 0xC5; //IAP命令寄存器
sfr IAP_TRIG = 0xC6; //IAP命令触发寄存器
sfr IAP_CONTR = 0xC7; //IAP控制寄存器
```

```
#define CMD_IDLE 0 //空闲模式
```

```
#define CMD_READ 1 //IAP字节读命令
```

```
#define CMD_PROGRAM 2 //IAP字节编程命令
```

```
#define CMD_ERASE 3 //IAP扇区擦除命令
```

```
#define URMD 0 //0:使用定时器2作为波特率发生器
 //1:使用定时器1的模式0(16位自动重载模式)作为波特率发生器
 //2:使用定时器1的模式2(8位自动重载模式)作为波特率发生器
```

```
sfr T2H = 0xd6; //定时器2高8位
```

```
sfr T2L = 0xd7; //定时器2低8位
```

```
sfr AUXR = 0x8e; //辅助寄存器
```



```

#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

//测试地址
#define IAP_ADDRESS 0x0400

void Delay(BYTE n);
void IapIdle();
BYTE IapReadByte(WORD addr);
void IapProgramByte(WORD addr, BYTE dat);
void IapEraseSector(WORD addr);
void InitUart();
BYTE SendData(BYTE dat);

void main()
{
 WORD i;

 P1 = 0xfe; //1111,1110 系统OK
 InitUart(); //初始化串口
 Delay(10); //延时
 IapEraseSector(IAP_ADDRESS); //扇区擦除
 for (i=0; i<512; i++) //检测是否擦除成功(全FF检测)
 {
 if (SendData(IapReadByte(IAP_ADDRESS+i)) != 0xff)
 goto Error; //如果出错,则退出
 }
 P1 = 0xfe; //1111,1100 擦除成功
 Delay(10); //延时
 for (i=0; i<512; i++) //编程512字节
 {
 IapProgramByte(IAP_ADDRESS+i, (BYTE)i);
 }
 P1 = 0xf8; //1111,1000 编程完成
 Delay(10); //延时
 for (i=0; i<512; i++) //校验512字节
 {
 if (SendData(IapReadByte(IAP_ADDRESS+i)) != (BYTE)i)
 goto Error; //如果校验错误,则退出
 }
 P1 = 0xf0; //1111,0000 测试完成
 while (1);
}

```

Error:

```
 P1 &= 0x7f; //0xxx,xxxx IAP操作失败
 while (1);

 }

 /*-----
 软件延时
 -----*/
 void Delay(BYTE n)
 {
 WORD x;

 while (n--)
 {
 x = 0;
 while (++x);
 }
 }

 /*-----
 关闭IAP
 -----*/
 void IapIdle()
 {
 IAP_CONTR = 0; //关闭IAP功能
 IAP_CMD = 0; //清除命令寄存器
 IAP_TRIG = 0; //清除触发寄存器
 IAP_ADDRH = 0x80; //将地址设置到非IAP区域
 IAP_ADDRL = 0;

 }

 /*-----
 从ISP/IAP/EEPROM区域读取一字节
 -----*/
 BYTE IapReadByte(WORD addr)
 {
 BYTE dat; //数据缓冲区

 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_READ; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 nop(); //等待ISP/IAP/EEPROM操作完成
 dat = IAP_DATA; //读ISP/IAP/EEPROM数据
 IapIdle(); //关闭IAP功能

 return dat; //返回
 }
}
```

```

/*-----
写一字节数据到ISP/IAP/EEPROM区域
-----*/
void IapProgramByte(WORD addr, BYTE dat)
{
 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_PROGRAM; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_DATA = dat; //写ISP/IAP/EEPROM数据
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 nop(); //等待ISP/IAP/EEPROM操作完成
 IapIdle();
}

/*-----
扇区擦除
-----*/
void IapEraseSector(WORD addr)
{
 IAP_CONTR = ENABLE_IAP; //使能IAP
 IAP_CMD = CMD_ERASE; //设置IAP命令
 IAP_ADDRL = addr; //设置IAP低地址
 IAP_ADDRH = addr >> 8; //设置IAP高地址
 IAP_TRIG = 0x5a; //写触发命令(0x5a)
 IAP_TRIG = 0xa5; //写触发命令(0xa5)
 nop(); //等待ISP/IAP/EEPROM操作完成
 IapIdle();
}

/*-----
初始化串口
-----*/
void InitUart()
{
 SCON = 0x5a; //设置串口为8位可变波特率
#if URMD == 0
 T2L = 0xd8; //设置波特率重装值
 T2H = 0xff; //115200 bps(65536-18432000/4/115200)
 AUXR = 0x14; //T2为1T模式, 并启动定时器2
 AUXR |= 0x01; //选择定时器2为串口的波特率发生器
#elif URMD == 1
 AUXR = 0x40; //定时器1为1T模式
 TMOD = 0x00; //定时器1为模式0(16位自动重载)
 TL1 = 0xd8; //设置波特率重装值
 TH1 = 0xff; //115200 bps(65536-18432000/4/115200)
 TR1 = 1; //定时器1开始启动

```

```

#else
 TMOD = 0x20; //设置定时器1为8位自动重载模式
 AUXR = 0x40; //定时器1为1T模式
 TH1 = TL1 = 0xfb; //115200 bps(256 - 18432000/32/115200)
 TR1 = 1;

#endif
}

/*-----
发送串口数据
-----*/
BYTE SendData(BYTE dat)
{
 while (!TI); //等待前一个数据发送完成
 TI = 0; //清除发送标志
 SBUF = dat; //发送当前数据

 return dat;
}

```

## 2. 汇编程序：

;**STC15F104ESW系列单片机EEPROM/IAP 功能测试程序演示**

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 15 系列单片机 EEPROM/IAP功能-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序中或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/

```

//本示例在Keil开发环境下请选择Intel的8052芯片型号进行编译

//假定测试芯片的工作频率为18.432MHz

```

#define URMD 0 //0:使用定时器2作为波特率发生器
 //1:使用定时器1的模式0(16位自动重载模式)作为波特率发生器
 //2:使用定时器1的模式2(8位自动重载模式)作为波特率发生器

T2H DATA 0D6H //定时器2高8位
T2L DATA 0D7H //定时器2低8位
AUXR DATA 08EH //辅助寄存器

IAP_DATA EQU 0C2H //IAP数据寄存器
IAP_ADDRH EQU 0C3H //IAP地址寄存器高字

```

|                       |                    |      |                      |
|-----------------------|--------------------|------|----------------------|
| IAP_ADDR_L            | EQU                | 0C4H | //IAP地址寄存器低字         |
| IAP_CMD               | EQU                | 0C5H | //IAP命令寄存器           |
| IAP_TRIG              | EQU                | 0C6H | //IAP命令触发寄存器         |
| IAP_CONTR             | EQU                | 0C7H | //IAP控制寄存器           |
|                       |                    |      |                      |
| CMD_IDLE              | EQU                | 0    | //空闲模式               |
| CMD_READ              | EQU                | 1    | //IAP字节读命令           |
| CMD_PROGRAM           | EQU                | 2    | //IAP字节编程命令          |
| CMD_ERASE             | EQU                | 3    | //IAP扇区擦除命令          |
|                       |                    |      |                      |
| ;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     |
|                       |                    |      |                      |
| //测试地址                |                    |      |                      |
| IAP_ADDRESS EQU 0400H |                    |      |                      |
| //-----               |                    |      |                      |
| ORG                   | 0000H              |      |                      |
| LJMP                  | MAIN               |      |                      |
| ;-----                |                    |      |                      |
| ORG                   | 0100H              |      |                      |
| MAIN:                 |                    |      |                      |
| LCALL                 | INIT_UART          |      | //初始化串口              |
| MOV                   | P1, #0FEH          |      | //1111,1110 系统OK     |
| LCALL                 | DELAY              |      | //延时                 |
| ;-----                |                    |      |                      |
| MOV                   | DPTR, #IAP_ADDRESS |      | //设置ISP/IAP/EEPROM地址 |
| LCALL                 | IAP_ERASE          |      | //扇区擦除               |
| ;-----                |                    |      |                      |
| MOV                   | DPTR, #IAP_ADDRESS |      | //设置ISP/IAP/EEPROM地址 |
| MOV                   | R0, #0             |      | //检测512字节            |
| MOV                   | R1, #2             |      |                      |
| CHECK1:               |                    |      | //检测是否擦除成功(全FF检测)    |
| LCALL                 | IAP_READ           |      | //读IAP数据             |
| LCALL                 | SEND_DATA          |      |                      |
| CJNE                  | A, #0FFH, ERROR    |      | //如果出错,则退出           |
| INC                   | DPTR               |      | //IAP地址+1            |
| DJNZ                  | R0, CHECK1         |      |                      |
| DJNZ                  | R1, CHECK1         |      |                      |
| ;-----                |                    |      |                      |
| MOV                   | P1, #0FCH          |      | //1111,1100 擦除成功     |
| LCALL                 | DELAY              |      | //延时                 |
| ;-----                |                    |      |                      |
| MOV                   | DPTR, #IAP_ADDRESS |      | //设置ISP/IAP/EEPROM地址 |

```

 MOV R0, #0 //编程512字节
 MOV R1, #2
 MOV R2, #0
NEXT:
 MOV A,R2 //准备数据
 LCALL IAP_PROGRAM //字节编程
 INC DPTR //IAP地址+1
 INC R2 //修改测试数据
 DJNZ R0, NEXT
 DJNZ R1, NEXT
;-----
 MOV P1, #0F8H //1111,1000 编程完成
 LCALL DELAY //延时
;-----
 MOV DPTR, #IAP_ADDRESS //设置ISP/IAP/EEPROM地址
 MOV R0, #0 //校验512字节
 MOV R1, #2
 MOV R2, #0
CHECK2:
 LCALL IAP_READ //读IAP数据
 LCALL SEND_DATA
 CJNE A,2, ERROR //如果出错,则退出
 INC DPTR //IAP地址+1
 INC R2
 DJNZ R0, CHECK2
 DJNZ R1, CHECK2
;-----
 MOV P1, #0F0H //1111,0000 测试完成
 SJMP $
;-----
ERROR:
 MOV P0, R0
 MOV P2, R1
 MOV P3, R2
 CLR P1.7 //0xxx,xxxx IAP 测试失败
 SJMP $

/*-----
软件延时
-----*/
DELAY:
 CLR A
 MOV R0, A
 MOV R1, A
 MOV R2, #20H
DELAY1:
 DJNZ R0, DELAY1
 DJNZ R1, DELAY1
 DJNZ R2, DELAY1
 RET

```

/\*-----

关闭IAP

-----\*/

IAP\_IDLE:

|     |            |      |                |
|-----|------------|------|----------------|
| MOV | IAP_CONTR, | #0   | //关闭IAP功能      |
| MOV | IAP_CMD,   | #0   | //清除命令寄存器      |
| MOV | IAP_TRIG,  | #0   | //清除触发寄存器      |
| MOV | IAP_ADDRH, | #80H | //将地址设置到非IAP区域 |
| MOV | IAP_ADDRL, | #0   |                |
| RET |            |      |                |

/\*-----

从ISP/IAP/EEPROM区域读取一字节

-----\*/

IAP\_READ:

|       |            |             |                        |
|-------|------------|-------------|------------------------|
| MOV   | IAP_CONTR, | #ENABLE_IAP | //使能IAP                |
| MOV   | IAP_CMD,   | #CMD_READ   | //设置IAP命令              |
| MOV   | IAP_ADDRL, | DPL         | //设置IAP低地址             |
| MOV   | IAP_ADDRH, | DPH         | //设置IAP高地址             |
| MOV   | IAP_TRIG,  | #5AH        | //写触发命令(0x5a)          |
| MOV   | IAP_TRIG,  | #0A5H       | //写触发命令(0xa5)          |
| NOP   |            |             | //等待ISP/IAP/EEPROM操作完成 |
| MOV   | A,         | IAP_DATA    | //读IAP数据               |
| LCALL | IAP_IDLE   |             | //关闭IAP功能              |
| RET   |            |             |                        |

/\*-----

写一字节数据到ISP/IAP/EEPROM区域

-----\*/

IAP\_PROGRAM:

|       |            |              |                        |
|-------|------------|--------------|------------------------|
| MOV   | IAP_CONTR, | #ENABLE_IAP  | //使能IAP                |
| MOV   | IAP_CMD,   | #CMD_PROGRAM | //设置IAP命令              |
| MOV   | IAP_ADDRL, | DPL          | //设置IAP低地址             |
| MOV   | IAP_ADDRH, | DPH          | //设置IAP高地址             |
| MOV   | IAP_DATA,  | A            | //写IAP数据               |
| MOV   | IAP_TRIG,  | #5AH         | //写触发命令(0x5a)          |
| MOV   | IAP_TRIG,  | #0A5H        | //写触发命令(0xa5)          |
| NOP   |            |              | //等待ISP/IAP/EEPROM操作完成 |
| LCALL | IAP_IDLE   |              | //关闭IAP功能              |
| RET   |            |              |                        |

/\*-----

扇区擦除

-----\*/

IAP\_ERASE:

|     |            |             |            |
|-----|------------|-------------|------------|
| MOV | IAP_CONTR, | #ENABLE_IAP | //使能IAP    |
| MOV | IAP_CMD,   | #CMD_ERASE  | //设置IAP命令  |
| MOV | IAP_ADDRL, | DPL         | //设置IAP低地址 |
| MOV | IAP_ADDRH, | DPH         | //设置IAP高地址 |

```

MOV IAP_TRIG, #5AH //写触发命令(0x5a)
MOV IAP_TRIG, #0A5H //写触发命令(0xa5)
NOP //等待ISP/IAP/EEPROM操作完成
LCALL IAP_IDLE //关闭IAP功能
RET

;/*-----
;初始化串口
;-----*/
INIT_UART:
 MOV SCON, #5AH ;设置串口为8位可变波特率
#if URMD == 0
 MOV T2L, #0D8H ;设置波特率重装值(65536-18432000/4/115200)
 MOV T2H, #0FFH
 MOV AUXR, #14H ;T2为1T模式, 并启动定时器2
 ORL AUXR, #01H ;选择定时器2为串口的波特率发生器
#elif URMD == 1
 MOV AUXR, #40H ;定时器1为1T模式
 MOV TMOD, #00H ;定时器1为模式0(16位自动重载)
 MOV TL1, #0D8H ;设置波特率重装值(65536-18432000/4/115200)
 MOV TH1, #0FFH
 SETB TR1 ;定时器1开始运行
#else
 MOV TMOD, #20H ;设置定时器1为8位自动重装模式
 MOV AUXR, #40H ;定时器1为1T模式
 MOV TL1, #0FBH ;115200 bps(256 - 18432000/32/115200)
 MOV TH1, #0FBH
 SETB TR1
#endif
 RET

;/*-----
;发送串口数据
;-----*/
SEND_DATA:
 JNB TI, $;等待前一个数据发送完成
 CLR TI ;清除发送标志
 MOV SBUF, A ;发送当前数据
 RET

END

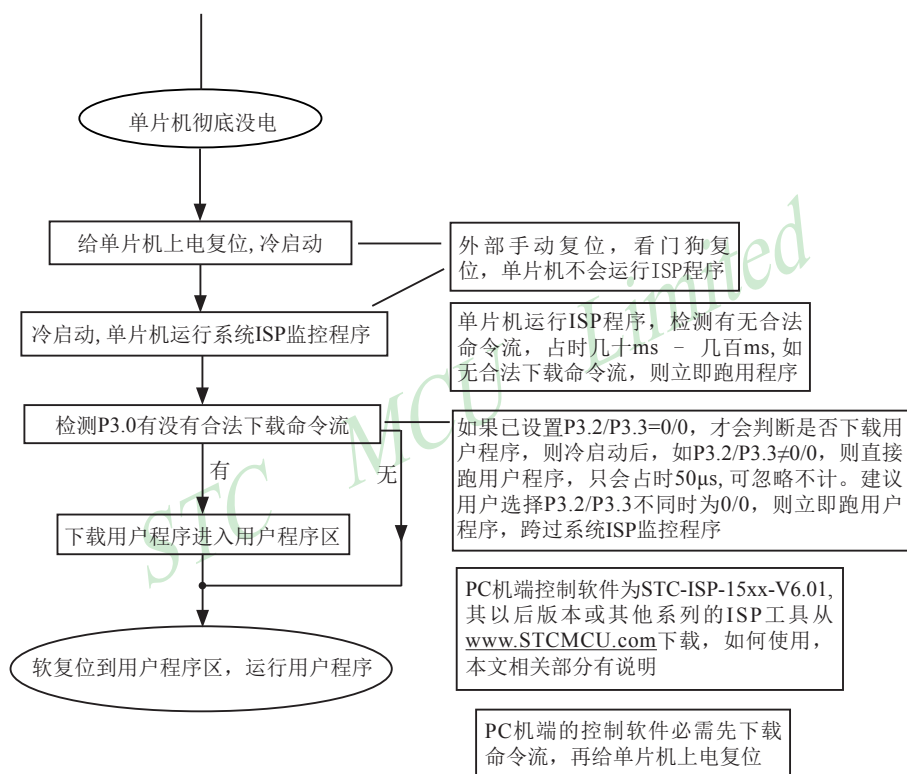
```



## 第10章 STC15系列单片机开发/编程工具说明

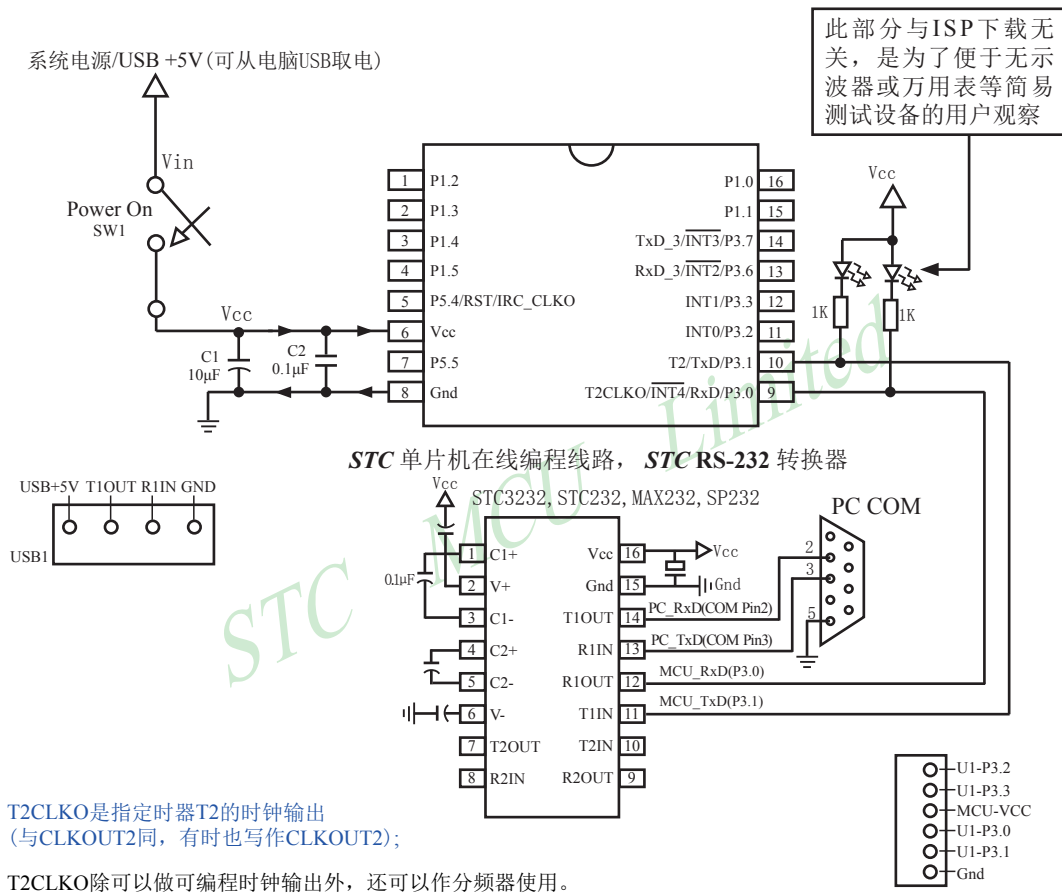
### 10.1 在系统可编程(ISP)原理, 官方演示工具使用说明

#### 10.1.1 在系统可编程(ISP)原理使用说明



## 10.1.2 STC15F104ESW系列在系统可编程(ISP)典型应用线路图

### 10.1.2.1 利用RS-232转换器的典型应用线路图



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

内部高可靠复位, 不需要外部复位电路

P5.4/RST/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°C~+85°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, STC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。

STC15F104ESW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](http://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。

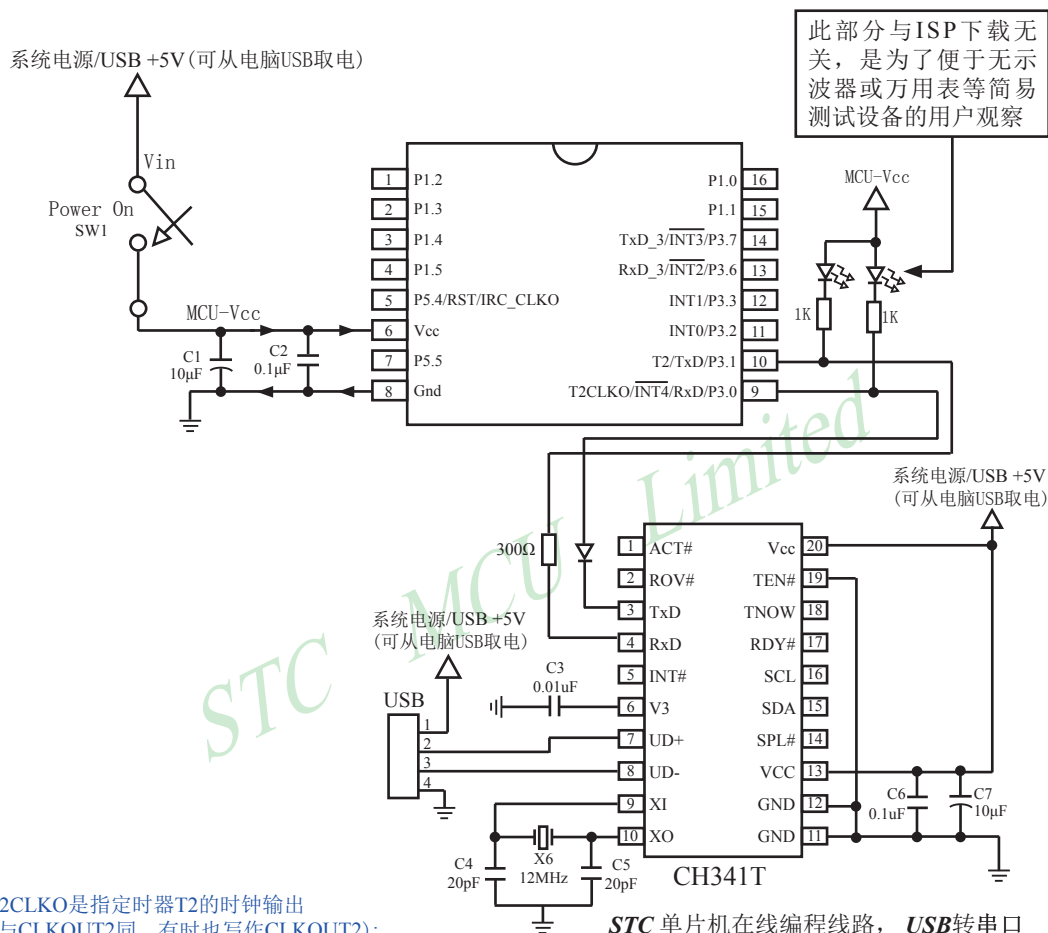
(2). 使用STC-ISP下载工具(软件),请随时更新,STC-ISP下载工具目前已更新到V4.88版本以上

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(冷启动)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”

#### 10.1.2.2 利用USB转串口的典型应用线路图



T2CLKO是指定时器T2的时钟输出  
(与CLKOUT2同,有时也写作CLKOUT2);

T2CLKO除可以做可编程时钟输出外,还可以作分频器使用。

内部高可靠复位，不需要外部复位电路

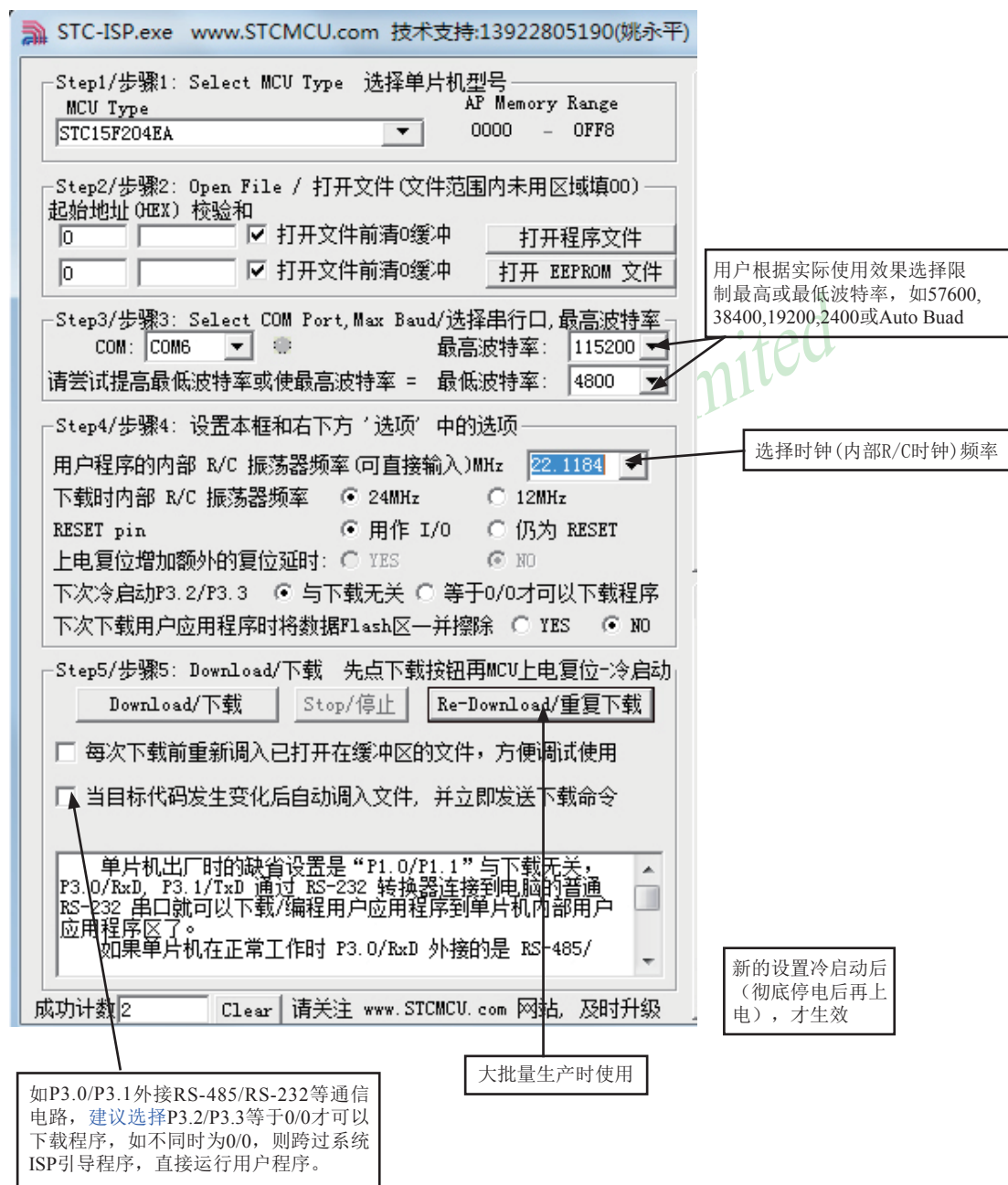
P5.4/RST/IRC\_CLKO脚出厂时默认为I/O口，可以通过 STC-ISP 编程器将其设置为RST复位脚。

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

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

## 10.1.3 电脑端的STC-ISP下载控制软件界面使用说明

### 10.1.3.1 STC-ISP下载控制软件Ver4.88的界面使用说明



Step1/步骤1: 选择你所使用的单片机型号, 如STC15F104ESW等

Step2/步骤2: 打开文件, 要烧录用户程序, 必须调入用户的程序代码 (\*.bin, \*.hex)

Step3/步骤3: 选择串行口, 你所使用的电脑串口, 如串行口1—COM1, 串行口2—COM2, ...

有些新式笔记本电脑没有RS-232串行口, 可买一条USB-RS232转接器, 人民币50元左右。

有些USB-RS232转接器, 不能兼容, 可让STC帮你购买经过测试的转换器。

Step4/步骤4: 选择内部R/C振荡时钟频率

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:

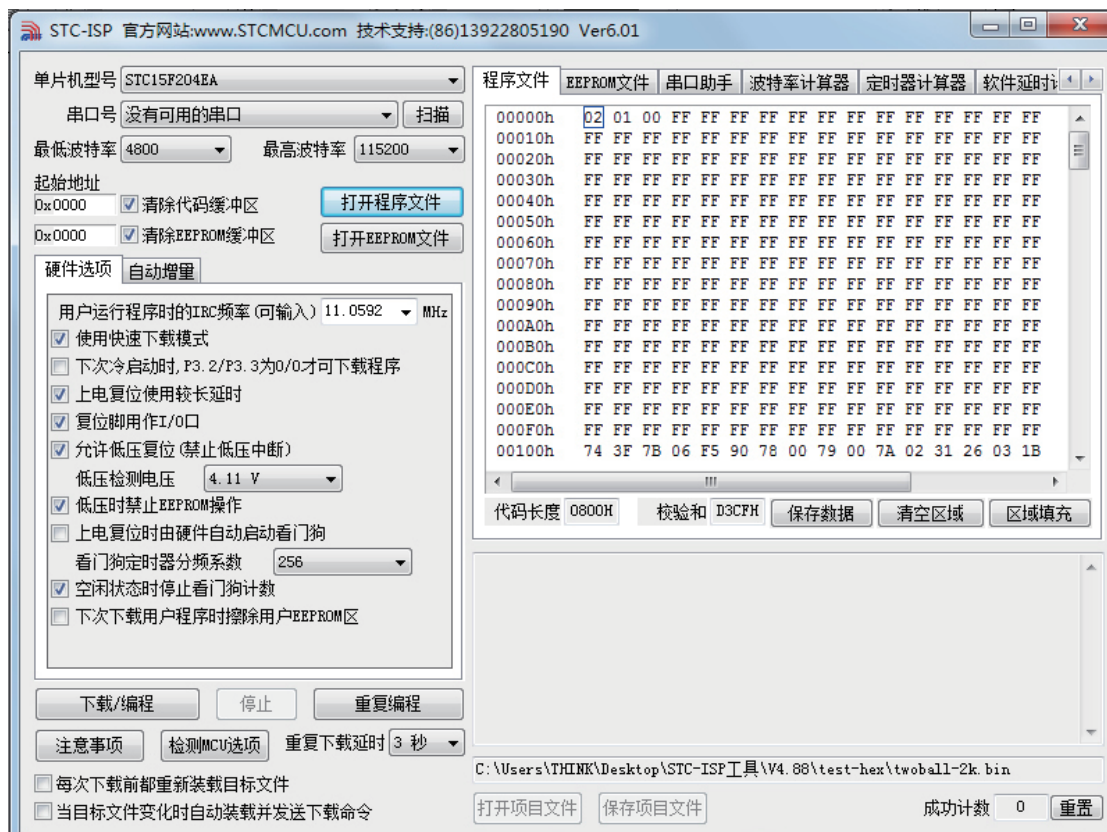
- (1). MCU/单片机 RXD(P3.0) --- RS-232转换器 --- PC/电脑 TXD(COM Port Pin3)
- (2). MCU/单片机 TXD(P3.1) --- RS-232转换器 --- PC/电脑 RXD(COM Port Pin2)
- (3). MCU/单片机 GND ----- PC/电脑 GND(COM Port Pin5)
- (4). 如果您的系统P3.0/P3.1连接到 RS-485 电路, 推荐

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).

### 10.1.3.2 最新STC15系列单片机的ISP下载控制软件Ver6.01的界面使用说明



最新的STC15xx系列单片机的ISP下载控制软件V6.01的界面如上图所示。该软件新增了许多新功能(如扫描当前系统中可用的串口、波特率计算器、软件延时计算器等),界面也与V4.88的软件界面有很大不同。下文将详细介绍该STC-ISP-15xx-V6.01软件的各个功能。



STC-ISP 官方网站:www.STCMCU.com 技术支持:(86)1

单片机型号 **STC15F204EA** 选择STC15系列单片机的型号

串口号 **没有可用的串口** 扫描 扫描当前系统中可用的串口

最低波特率 **4800** 最高波特率 **115200** 用户根据实际使用效果选择限制最高或最低波特率，如57600, 38400, 19200, 2400或Auto Baud

起始地址 **0x0000** ☒ 清除代码缓冲区 打开程序文件 打开用户的程序代码文件

**0x0000** ☒ 清除EEPROM缓冲区 打开EEPROM文件 打开EEPROM数据文件

硬件选项 **自动增量**

用户运行程序时的IRC频率(可输入) **11.0592** MHz 选择时钟(内部R/C时钟)频率(可输入)

☒ 使用快速下载模式 是否使用较快速度的内部振荡器频率进行下载  
选择: 使用较快频率的内部振荡器  
不选择: 使用较慢频率的内部振荡器

☐ 下次冷启动时, P3.2/P3.3为0/0才可下载程序 下次是否需要P3.1和P3.2同时为低电平时才可下载程序  
选择: P3.1和P3.2同时为低电平时才可下载程序  
不选择: 下载时不检测P3.1和P3.2的电平

☒ 上电复位使用较长延时 上电复位时, 是否需要额外的复位延时  
选择: 需要额外的复位延时  
不选择: 一般长度的复位延时

☒ 复位脚用作I/O口 是否需要将复位引脚当作普通I/O口来使用  
选择: 复位引脚当作普通I/O口  
不选择: 复位引脚仍为复位脚

☒ 允许低压复位(禁止低压中断) 当电压低于设定的低压检测门槛电压时, 芯片是复位还是中断  
选择: 检测到低压时复位  
不选择: 检测到低压时不复位而产生低压中断  
建议: 当振荡器频率高于20MHz时  
对于3V的芯片, 低压检测门槛电压建议选择2.5V以上  
对于5V的芯片, 低压检测门槛电压建议选择4.11V以上

低压检测电压 **4.11 V**

☒ 低压时禁止EEPROM操作

☐ 上电复位时由硬件自动启动看门狗

看门狗定时器分频系数 **256**

☒ 空闲状态时停止看门狗计数

☐ 下次下载用户程序时擦除用户EEPROM区

**下载/编程** **停止** **重复编程**

**注意事项** **检测MCU选项** **重复下载延时 **3** 秒**

☐ 每次下载前都重新装载目标文件

☐ 当目标文件变化时自动装载并发送下载命令

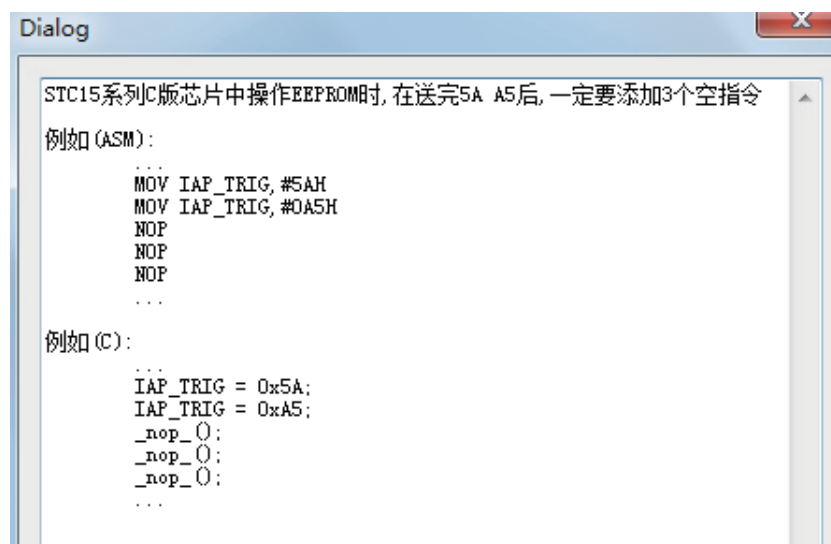
如P3.0/P3.1外接RS-485/RS-232等通信电路, 建议选择P3.2/P3.3等于0/0才可以下载程序, 如不同时为0/0, 则跨过系统ISP引导程序, 直接运行用户程序。

大批量生产时使用

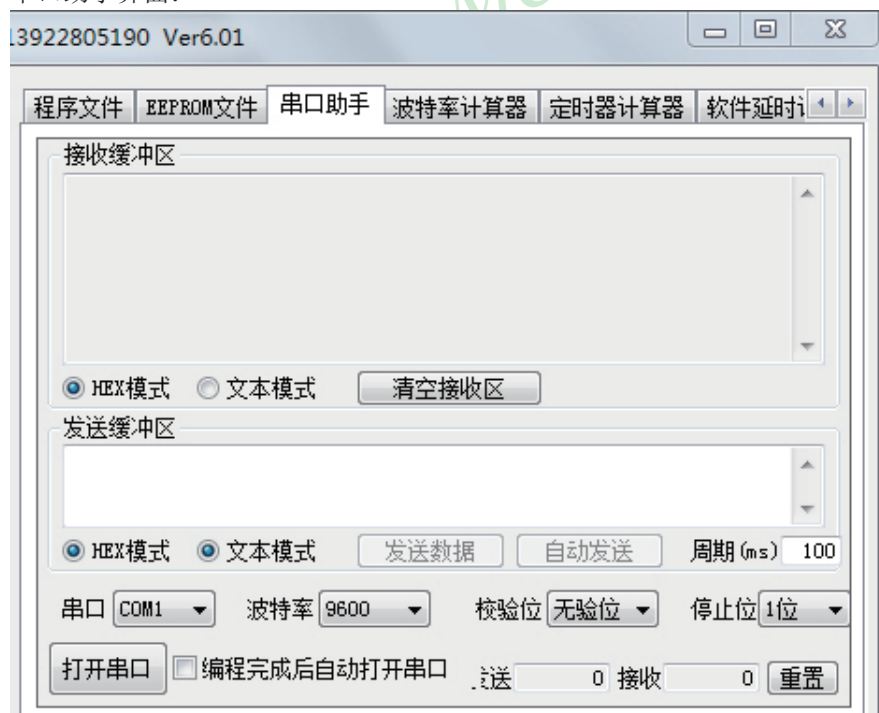
新的设置冷启动后(彻底停电后再上电), 才生效



点击界面上的注意事项按钮后出现下面的对话框:



串口助手界面:



波特率计算器界面：

3922805190 Ver6.01

程序文件 | EEPROM文件 | 串口助手 | **波特率计算器** | 定时器计算器 | 软件延时

系统频率 11.0592 MHz      UART选择 串口1

波特率 3600      UART数据位 8位数据

☐ 波特率倍速 (SMOD)      波特率发生器 定时器1 (16位自动重载)

误差 0.00%      定时器时钟 1T (FOSC)

```

void UartInit(void) //9600bps@11.0592MHz
{
 SCON = 0x50; //8位数据,可变波特率
 AUXR |= 0x40; //定时器1时钟为Fosc,即1T
 AUXR &= 0xFE; //串口1选择定时器1为波特率发生器
 TMOD &= 0x0F; //设定定时器1为16位自动重载方式
 TL1 = 0xE0; //设定定时器1初值
 TH1 = 0xFE; //设定定时器1初值
 ET1 = 0; //禁止定时器1中断
 TR1 = 1; //启动定时器1
}

```

生成C代码    生成ASM代码    复制代码

定时器计算器界面：

13922805190 Ver6.01

程序文件 | EEPROM文件 | 串口助手 | 波特率计算器 | **定时器计算器** | 软件延时

系统频率 11.0592 MHz      选择定时器 定时器0

定时长度 100 微秒      定时器模式 16位自动重载 (15系列)

误差 0.01%      定时器时钟 1T (FOSC)

```

void Timer0Init(void) //100微秒@11.0592MHz
{
 AUXR |= 0x80; //定时器时钟1T模式
 TMOD &= 0xF0; //设定定时器模式
 TLO = 0xAE; //设定定时器初值
 TH0 = 0xFB; //设定定时器初值
 TFO = 0; //清除TFO标志
 TR0 = 1; //定时器0开始计时
}

```

生成C代码    生成ASM代码    复制代码

软件延时计算器界面:

3922805190 Ver6.01

EEPROM文件 串口助手 波特率计算器 定时器计算器 软件延时计算器

系统频率 11.0592 MHz

定时长度 100 微秒

8051指令集 STC-Y5

```
void Delay100us() // @11.0592MHz
{
 unsigned char i, j;

 nop();
 nop();
 i = 2;
 j = 15;
 do
 {
 while (--j);
 } while (--i);
}
```

生成C代码 生成ASM代码 复制代码

## 10.1.4 STC-ISP(最方便的在线升级软件)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 15系列ISP下载编程工具作为编程工具

STC-ISP Ver 3.0A PCB板可以焊接3种电路，分别支持STC15系列8Pin / 16Pin / 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 /16Pin /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 15系列ISP下载编程工具(其实就是单片机通过RS-232转换器连接到电脑)完成下载编程用户程序的工作：

关于硬件连接：

- (1). 根据单片机的工作电压选择单片机电源电压
  - A. 5V单片机，短接JP1的MCU-VCC，+5V电源管脚
  - B. 3V单片机，短接JP1的MCU-VCC，3.3V电源管脚
- (2). 连接线(STC提供)
  - A. 将一端有9芯连接座的插头插入PC/电脑RS-232串行接口插座用于通信
  - B. 将同一端的USB插头插入PC/电脑USB接口用于取电
  - C. 将只有一个USB插头的一端插入STC-ISP Ver 3.0A PCB板USB1插座用于RS-232通信和供电，此时USB +5V Power灯亮(D43, USB接口有电)
- (3). 其他插座不需连接
- (4). SW1开关处于非按下状态，此时MCU-VCC Power灯不亮(D41)，没有给单片机通电
- (5). SW3开关
  - 处于非按下状态，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(STC15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。

## 10.1.5 若无RS-232转换器，如何用STC的ISP下载板做RS-232通信转换

利用STC 15系列ISP下载编程工具(其实就是单片机通过RS-232转换器连接到电脑)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也从用户系统引到STC 15系列ISP下载编程工具(其实就是单片机通过RS-232转换器连接到电脑)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。

## 10.1.6 如何解决VB版ISP工具在XP或WIN7下控件过期或不能注册的问题

本节用于解决VB版本的ISP下载工具在XP或者WIN7下由于控件版本过期或者不能注册而导致不能下载的问题。

### 1、控件版本过期

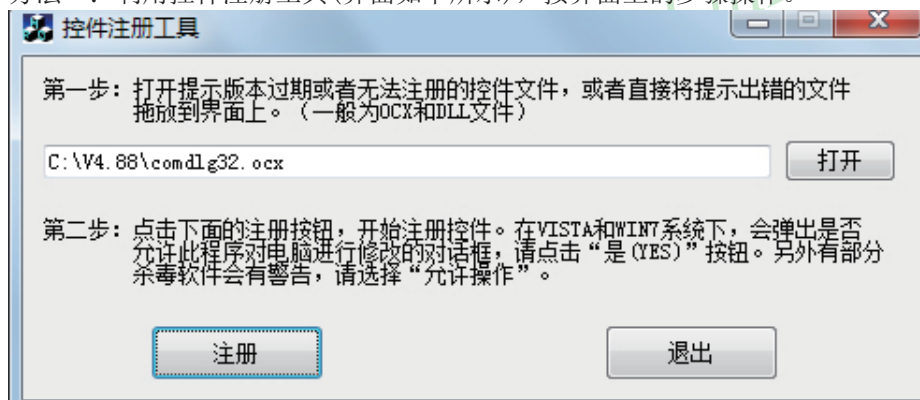
在打开STC的ISP下载界面时若出现如下画面



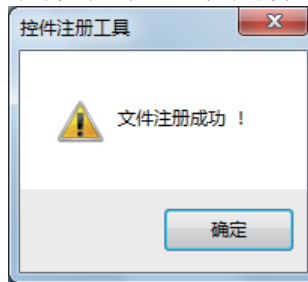
或者与之类似的加载失败和版本过期的错误提示时，表示该ISP下载工具的控件已经过期。

解决方法：

方法一：利用控件注册工具(界面如下所示)，按界面上的步骤操作。



点击注册后会出现下图所示的界面提示，表示控件版本过期的问题已解决。重新运行STC-ISP下载工具即可正常下载了。



方法二：将解压目录下的“comdlg32.ocx”文件复制到“c:\windows\system32\”下，覆盖原文件，然后再次运行应用程序。

## 2、控件不能正确注册

一般这种错误只会出现在windows7和vista系统，当打开STC的ISP下载界面时若出现如下画面



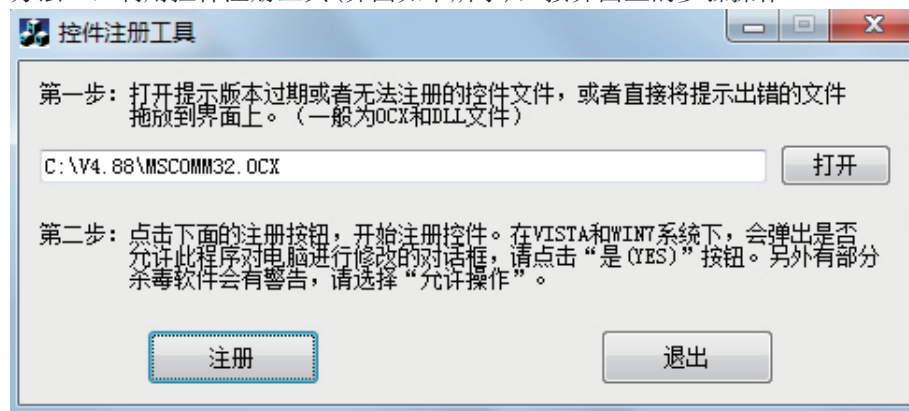
或者如下画面时



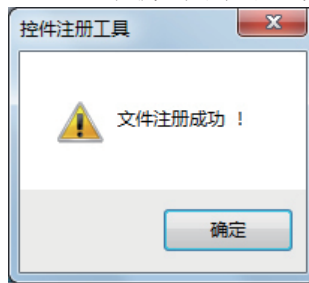
表示出现控件不能正确注册的问题。

解决方法：

方法一：利用控件注册工具(界面如下所示)，按界面上的步骤操作。



点击注册后会出现下图所示的界面提示，表示控件不能正确注册的问题已解决。重新运行STC-ISP下载工具即可正常下载了。



方法二：以管理员身份运行可执行程序即可，具体操作如下：

- ① 进入 STC的ISP下载程序所在的目录，找到可执行文件（例如STC\_ISP\_V488.EXE）
- ② 右键点击可执行文件
- ③ 在右键菜单中选择“以管理员身份运行程序”，从而可以带到注册控件的目的
- ④ 下次再运行程序时便可直接打开了



## 10.2 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: STC15F104ESW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

STC15F104ESW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。

## 10.3 自定义下载演示程序(实现不停电下载)

```

/*-----*/
/* --- STC MCU Limited. -----*/
/* --- 演示STC 1T 系列单片机 利用软件实现自定义下载-----*/
/* --- Mobile: (86)13922809991 -----*/
/* --- Fax: 86-755-82905966 -----*/
/* --- Tel: 86-755-82948412 -----*/
/* --- Web: www.STCMCU.com -----*/
/* 如果要在程序中使用或在文章中引用该程序， -----*/
/* 请在程序或文章中注明使用了STC的资料及程序 -----*/
/*---- 在 Keil C 开发环境中，选择 Intel 8052 编译即可-----*/
/*-----*/

#include <reg51.h>
#include <intrins.h>

sfr IAP_CONTR = 0xc7;
sbit MCU_Start_Led = P1^7;

#define Self_Define_ISP_Download_Command 0x22
#define RELOAD_COUNT 0xfb //18.432MHz,12T,SMOD=0,9600bps
//define RELOAD_COUNT 0xf6 //18.432MHz,12T,SMOD=0,4800bps
//define RELOAD_COUNT 0xec //18.432MHz,12T,SMOD=0,2400bps
//define RELOAD_COUNT 0xd8 //18.432MHz,12T,SMOD=0,1200bps

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

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

 serial_port_initial(); //Initial UART
 display_MCU_Start_Led(); //Turn on the work LED
 send_UART(0x34); //Send UART test data
 send_UART(0xa7); // Send UART test data
 while (1);
}

void send_UART(unsigned char i)
{
 ES = 0; //Disable serial interrupt
 TI = 0; //Clear TI flag

```

```
 SBUF = i; //send this data
 while (!TI); //wait for the data is sent
 TI = 0; //clear TI flag
 ES = 1; //enable serial interrupt
 }

void UART_Interrupt)Receive(void) interrupt 4 using 1
{
 unsigned char k = 0;
 if (RI)
 {
 RI = 0;
 k = SBUF;
 if (k == Self_Define_ISP_Command) //check the serial data
 {
 delay(); //delay 1s
 delay(); //delay 1s
 soft_reset_to_ISP_Monitor();
 }
 }
 if (TI)
 {
 TI = 0;
 }
}

void soft_reset_to_ISP_Monitor(void)
{
 IAP_CONTR = 0x60; //0110,0000 soft reset system to run ISP monitor
}

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

```
void display_MCU_Start_Led(void)
{
 unsigned char i = 0;
 for (i=0; i<3; i++)
 {
 MCU_Start_Led = 0; //Turn on work LED
 dejay();
 MCU_Start_Led = 1; //Turn off work LED
 dejay();
 MCU_Start_Led = 0; //Turn on work LED
 }
}
```

STC MCU Limited

## 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" embedded 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. A further step may require a "linker" to combine portions of program from separate files and to set the address in memory at which the program may execute. We begin with a few definitions.

An assembly language program is 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.

An assembler is a program that translates 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 partial 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.

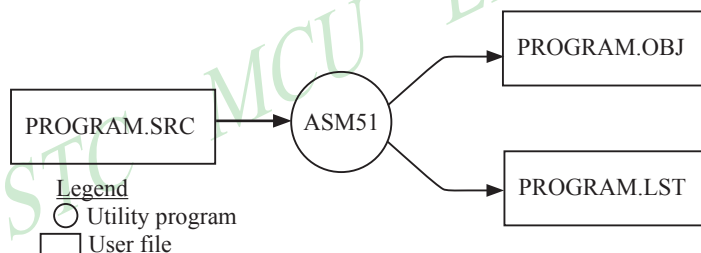


Figure 1 Assembling a source program

### 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 symbol 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] [...] [:comment]
```

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 characters. 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 used 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

Instruction 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 possibilities for the operand field are numerous, and we shall elaborate on these at length. But first, the comment field.

## Comment Field

Remarks to clarify the program go into comment field at the end of each line. Comments must begin with a semicolon (;). Each lines may be comment lines by beginning them with a semicolon. Subroutines and large sections of a program generally begin with a comment block—several 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 instructions, 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 pre-defined 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 evaluating 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 special 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 always 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 categories 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 category.

### 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 definition 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, etc.), 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 create 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 do 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 the 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 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 continued. 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

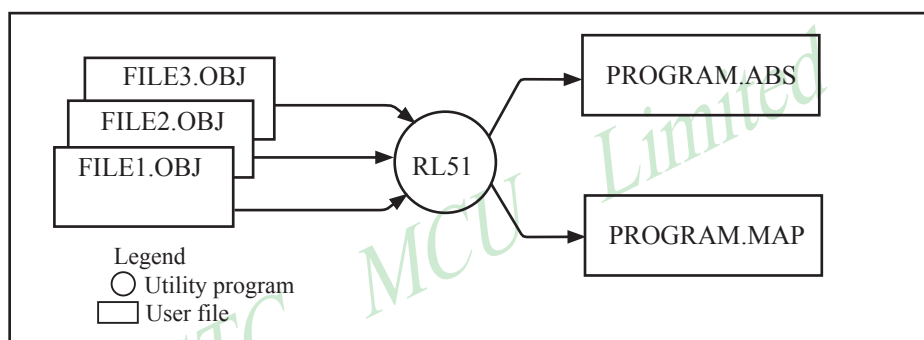
Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any affect on the program itself. They can be entered on the invocation line when a program is assembled, or they can be placed in the source file. Assembler controls appearing in the source file must be preceded with a dollar 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 beginning 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 continuation 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/<br>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<br>(file)                  | P                   | NOERRORPRINT       | EP      | Designates a file to receive error messages in addition to the listing file (defaults 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                                                        |
| INCLUDED(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 listing file                                                  |
| MACRO<br>(men_percent)                | 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 evaluate 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                                                      |
| PAGELNGTH<br>(N)                      | P                   | PAGELNGT(60)       | PL      | Sets maximum 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<br>(rb,...)              | P                   | REGISTERBANK(0)    | RB      | Indicates one or more banks used in program module                                                            |
| NOREGISTER-<br>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<br>(path)                   | P                   | same as source     | WF      | Designates alternate path for temporary 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, etc., 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 is 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 compiler handles on behalf of the programmer
- Easier to write, especially for large and complex programs
- Produces more readable program source codes

Nevertheless, 8051 C, being very similar to the conventional C language, also suffers from the following disadvantages:

- Processes the disadvantages of high-level, structured programming languages.
- Generally generates larger machine codes
- Programmer has less control and less ability to directly interact with hardware

To compare between 8051 C and assembly language, consider the solutions to the Example—Write a program using Timer 0 to create a 1KHz square wave on P1.0.

A solution written below in 8051 C language:

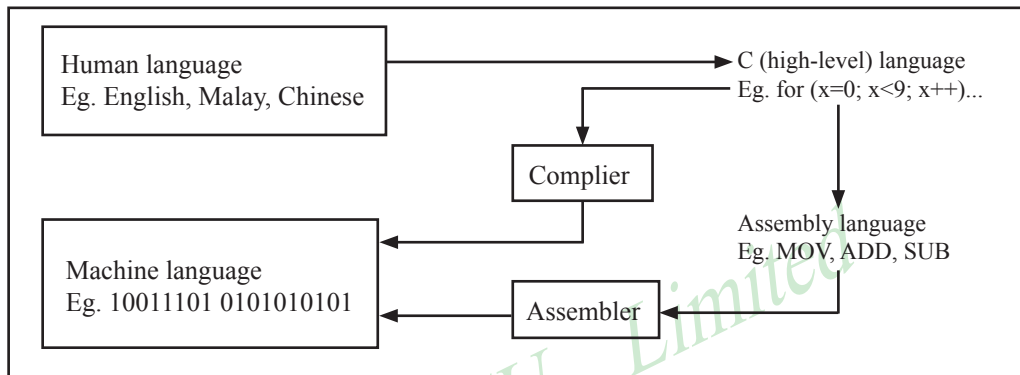
```
sbit portbit = P1^0; /*Use variable portbit to refer to P1.0*/
main ()
{
 TMOD = 1;
 while (1)
 {
 TH0 = 0xFE;
 TL0 = 0xC;
 TR0 = 1;
 while (TF0 != 1);
 TR0 = 0;
 TF0 = 0;
 portbit = !(P1.^0);
 }
}
```

A solution written below in assembly language:

|       |      |            |                            |
|-------|------|------------|----------------------------|
|       | ORG  | 8100H      |                            |
|       | MOV  | TMOD, #01H | ;16-bit timer mode         |
| LOOP: | MOV  | TH0, #0FEH | ;500 (high byte)           |
|       | MOV  | TL0, #0CH  | ;500 (low byte)            |
|       | SETB | TR0        | ;start timer               |
| WAIT: | JNB  | TF0, WAIT  | ;wait for overflow         |
|       | CLR  | TR0        | ;stop timer                |
|       | CLR  | TF0        | ;clear timer overflow flag |
|       | CPL  | P1.0       | ;toggle port bit           |
|       | SJMP | LOOP       | ;repeat                    |
|       | END  |            |                            |

Notice that both the assembly and C language solutions for the above example require almost the same number of lines. However, the difference lies in the readability of these programs. The C version seems more human than assembly, and is hence more readable. This often helps facilitate the human programmer's efforts to write even very complex programs. The assembly language version is more closely related to the machine code, and though less readable, often results in more compact machine code. As with this example, the resultant machine code from the assembly version takes 83 bytes while that of the C version requires 149 bytes, an increase of 79.5%!

The human programmer's choice of either high-level C language or assembly language for talking to the 8051, whose language is machine language, presents an interesting picture, as shown in following figure.



Conversion between human, high-level, assembly, and machine language

## 8051 C COMPILERS

We saw in the above figure that a compiler 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 compiler 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 compiler faces a greater task to bridge that difference.

Currently, there exist various 8051 C compiler, which offer almost similar functions. All our examples and programs have been compiled and tested with Keil's  $\mu$  Vision 2 IDE by Keil Software, an integrated 8051 program development environment 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     | $\pm 1.175494\text{E}-38$ to $\pm 3.402823\text{E}+38$ |
| 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 registers (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 indicates 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 caret 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 errmsg[] = "An error occurred" ;
```

declares a char array called errmsg 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 internal **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 array 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 individual 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

|                           |                                                              |
|---------------------------|--------------------------------------------------------------|
| <code>data_type</code>    | refers to which type of data that the pointer is pointing to |
| <code>*</code>            | denotes that this is a pointer variable                      |
| <code>pointer_name</code> | is the name of the pointer                                   |

As an example, the following declarations:

```
int * numPtr
int num;
numPtr = #
```

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 = #
```

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 = #
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 address 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 specifies that pointer numPtr should reside in external data memory (**xdata**), and we say that the pointer's memory type is **xdata**.

## Typed Pointers

We can go even further when declaring pointers. Consider the example:

```
int data * xdata numPtr = #
```

The above statement declares the same pointer numPtr to reside in external data memory (**xdata**), and this pointer points to data of type int that is itself stored in the variable num in internal data memory (**data**). The memory type specifier, **data**, before the \* specifies the *data memory type* while the memory type specifier, **xdata**, after the \* specifies the pointer memory type.

Pointer declarations where the data memory types are explicitly specified are called typed pointers. Typed pointers have the property that you specify in your code where the data pointed by pointers should reside. The size of typed pointers depends on the data memory type and could be one or two bytes.

## Untyped Pointers

When we do not explicitly state the data memory type when declaring pointers, we get untyped pointers, which are generic pointers that can point to data residing in any type of memory. Untyped pointers have the advantage that they can be used to point to any data independent of the type of memory in which the data is stored. All untyped pointers consist of 3 bytes, and are hence larger than typed pointers. Untyped pointers are also generally slower because the data memory type is not determined or known until the compiled program is run at runtime. The first byte of untyped pointers refers to the data memory type, which is simply a number according to the following table. The second and third bytes are, respectively, the higher-order and lower-order bytes of the address being pointed to.

An untyped pointer is declared just like normal C, where:

```
int * xdata numPtr = #
```

does not explicitly specify the memory type of the data pointed to by the pointer. In this case, we are using untyped pointers.

| Data memory type values stored in first byte of untyped pointers |                  |
|------------------------------------------------------------------|------------------|
| Value                                                            | Data Memory Type |
| 1                                                                | idata            |
| 2                                                                | xdata            |
| 3                                                                | pdata            |
| 4                                                                | data/bdata       |
| 5                                                                | code             |

## FUNCTIONS

In programming the 8051 in assembly, we learnt the advantages of using subroutines to group together common and frequently used instructions. The same concept appears in 8051 C, but instead of calling them subroutines, we call them **functions**. As in conventional C, a function must be declared and defined. A function definition includes a list of the number and types of inputs, and the type of the output (return type), plus 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 actually 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 arguments. 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 technicalities. 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 STC15F104ESW系列单片机电气特性

### Absolute Maximum Ratings

| Parameter                 | Symbol    | Min  | Max       | Unit |
|---------------------------|-----------|------|-----------|------|
| Storage temperature       | TST       | -55  | +125      | °C   |
| Operating temperature (I) | TA        | -40  | +85       | °C   |
| Operating temperature (C) | TA        | 0    | +70       | °C   |
| 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)<br>(Quasi-output)             | 200           | 270   | -    | uA   | 5V             |
| IOH2 | Sourcing Current for output high (P0,P1,P2,P3)<br>(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)<br>(Quasi-output) | 140           | 170  | -    | uA   | 3.3V            |
| IOH2 | Sourcing Current for output high (P0,P1,P2,P3)<br>(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       |

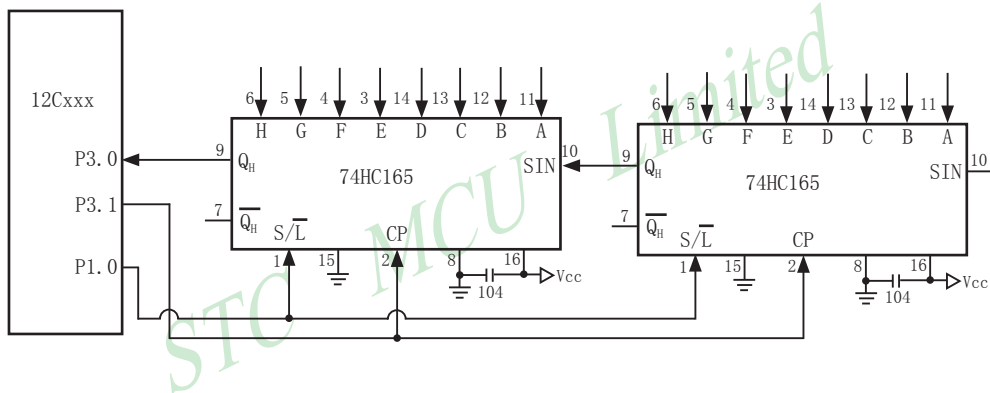
## 附录D：用串口扩展I/O接口

STC15系列单片机串行口的方式0可用于I/O扩展。如果在应用系统中，串行口未被占用，那么将它用来扩展并行I/O口是一种经济、实用的方法。

在操作方式0时，串行口作同步移位寄存器，其波特率是固定的，为 $\text{SYSclk}/12$ （ $\text{SYSclk}$ 为系统时钟频率）。数据由RXD端（P3.0）出入，同步移位时钟由TXD端（P3.1）输出。发送、接收的是8位数据，低位在先。

### 一、用74HC165扩展并行输入口

下图是利用两片74HC165扩展二个8位并行输入口的接口电路图。



74HC165是8位并行置入移位寄存器。当移位/置入端(S/L)由高到低跳变时，并行输入端的数据置入寄存器；当S/L=1，且时钟禁止端（第15脚）为低电平时，允许时钟输入，这时在时钟脉冲的作用下，数据将由 $Q_A$ 到 $Q_H$ 方向移位。

上图中，TXD(P3.1)作为移位脉冲输出端与所有74HC165的移位脉冲输入端CP相连；RXD(P3.0)作为串行输入端与74HC165的串行输出端 $Q_H$ 相连；P1.0用来控制74HC165的移位与置入而同S/L相连；74HC165的时钟禁止端（15脚）接地，表示允许时钟输入。当扩展多个8位输入口时，两芯片的首尾（ $Q_H$ 与 $S_{IN}$ ）相连。

下面的程序是从16位扩展口读入5组数据（每组二个字节），并把它们转存到内部RAM 20H开始的单元中。

```

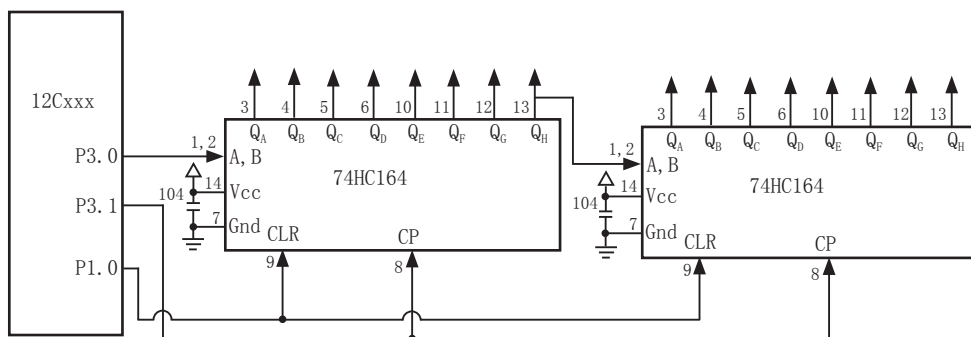
MOV R7, #05H ; 设置读入组数
MOV RO, #20H ; 设置内部RAM数据区首址
START: CLR P1.0 ; 并行置入数据, S/L=0
 SETB P1.0 ; 允许串行移位S/L=1
MOV R1, #02H ; 设置每组字节数, 即外扩74LS165的个数
RXDATA: MOV SCON, #00010000B ; 设串行方式0, 允许接收, 启动接收过程
WAIT: JNB RI, WAIT ; 未接收完一帧, 循环等待
 CLR RI ; 清RI标志, 准备下次接收
 MOV A, SBUF ; 读入数据
 MOV @RO, A ; 送至RAM缓冲区
 INC RO ; 指向下一个地址
 DJNZ R1, RXDATA ; 为读完一组数据, 继续
 DJNZ R7, START ; 5组数据未读完重新并行置入
 ; 对数据进行处理

```

上面的程序对串行接收过程采用的是查询等待的控制方式, 如有必要, 也可改用中断方式。从理论上讲, 按上图方法扩展的输入口几乎是无限的, 但扩展的越多, 口的操作速度也就越慢。

## 二、用74HC164扩展并行输出口

74HC164是8位串入并出移位寄存器。下图是利用74HC164扩展二个8位输出口的接口电路。



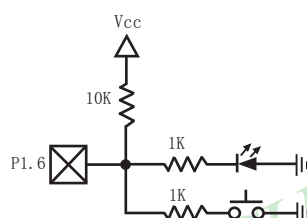
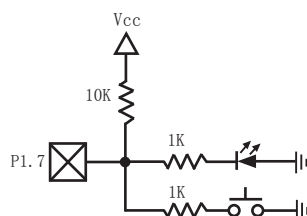
当单片机串行口工作在方式0的发送状态时，串行数据由P3.0（RXD）送出，移位时钟由P3.1（TXD）送出。在移位时钟的作用下，串行口发送缓冲器的数据一位一位地移入74HC164中。需要指出的是，由于74HC164无并行输出控制端，因而在串行输入过程中，其输出端的状态会不断变化，故在某些应用场合，在74HC164的输出端应加接输出三态门控制，以便保证串行输入结束后再输出数据。

下面是将RAM缓冲区30H、31H的内容串行口由74HC164并行输出的子程序。

```
START: MOV R7, #02H ; 设置要发送的字节个数
 MOV R0, #30H ; 设置地址指针
 MOV SCON, #00H ; 设置串行口方式0
SEND: MOV A, @R0
 MOV SBUF, A ; 启动串行口发送过程
WAIT: JNB TI, WAIT ; 一帧数据未发送完，循等待
 CLR TI
 INC R0 ; 取下一个数
 DJNZ R7, SEND
 RET
```



## 附录E：一个I/O口驱动发光二极管并扫描按键



利用STC15系列单片机的I/O口可设置成弱上拉, 强上拉(推挽)输出, 仅为输入(高阻), 开漏四种模式的特性, 可以利用STC15系列单片机的I/O口同时作为发光二极管驱动及按键检测用, 可以大幅节省I/O口。

当驱动发光二极管时, 将该I/O口设置成强推挽输出, 输出高即可点亮发光二极管。  
当检测按键时, 将该I/O口设置成弱上拉输入, 再读外部口的状态, 即可检测按键。

## 附录F：STC15系列单片机取代传统8051注意事项

STC15F104ESW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STC15F104ESW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STC15F104ESW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STC15F104ESW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STC15F104ESW系列

看门狗：

最新STC15F104ESW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STC15F104ESW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STC15F104ESW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<br>IAP_DATA<br>STC89xx 系列<br>ISP_DATA   | C2h<br>E2h | ISP/IAP Flash Data Register    |       |      |       |          |   |     |     |     | 1111,1111   |
| STC15Fxx 系列<br>IAP_ADDRH<br>STC89xx 系列<br>ISP_ADDRH | C3h<br>E3h | ISP/IAP Flash Address High     |       |      |       |          |   |     |     |     | 0000,0000   |
| STC15Fxx 系列<br>IAP_ADDRL<br>STC89xx 系列<br>ISP_ADDRL | C4h<br>E4h | ISP/IAP Flash Address Low      |       |      |       |          |   |     |     |     | 0000,0000   |
| STC15Fxx 系列<br>IAP_CMD<br>STC89xx 系列<br>ISP_CMD     | C5h<br>E5h | ISP/IAP Flash Command Register | -     | -    | -     | -        | - | -   | MS1 | MS0 | xxxx,xx00   |
| STC15Fxx 系列<br>IAP_TRIG<br>STC89xx 系列<br>ISP_TRIG   | C6h<br>E6h | ISP/IAP Flash Command Trigger  |       |      |       |          |   |     |     |     | xxxx,xxxx   |
| STC15Fxx系列<br>IAP_CONTR<br>STC89xx 系列<br>ISP_CONTR  | C7h<br>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STC15F104ESW系列单片机只有定时器/计数器2, 定时器/计数器2的工作模式固定为16位自动重装载模式。而传统STC89系列单片机设有定时器2, 但它集成了的定时器/计数器0和定时器/计数器1, 它们有四种工作模式: 模式0是13位定时/计数器模式; 模式1是16位定时器/计数器模式; 模式2是8位自动重装载模式; 模式3是两个8位定时器/计数器模式。

### 外部时钟和内部时钟:

最新STC15F104ESW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STC15F104ESW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STC15F104ESW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STC15F104ESW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STC15F104ESW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。另外最新STC15F104ESW系列单片机还内置了掉电唤醒专用定时器

## 附录G：STC15F104ESW系列对指令系统的提升

- 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和MUL AB指令的执行速度大幅提升24倍
- 共有22条指令，一个时钟就可以执行完成，平均速度快8~12倍

如果按功能分类，STC15F104ESW系列单片机指令系统可分为：

1. 算术操作类指令；
2. 逻辑操作类指令；
3. 数据传送类指令；
4. 布尔变量操作类指令；
5. 控制转移类指令。

按功能分类的指令系统表如下表所示。

### 指令执行速度效率提升总结(新15系列)：

指令系统共包括111条指令，其中：

|            |      |
|------------|------|
| 执行速度快24倍的  | 共2条  |
| 执行速度快12倍的  | 共28条 |
| 执行速度快8倍的   | 共19条 |
| 执行速度快6倍的   | 共40条 |
| 执行速度快4.8倍的 | 共8条  |
| 执行速度快4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）(新15系列)：

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22条 |
| 2个时钟就可执行完成的指令 | 共37条 |
| 3个时钟就可执行完成的指令 | 共31条 |
| 4个时钟就可执行完成的指令 | 共12条 |
| 5个时钟就可执行完成的指令 | 共8条  |
| 6个时钟就可执行完成的指令 | 共1条  |

STC15系列将111条指令全部执行完一遍所需的时钟为283个时钟，而传统8051单片机将111条指令全部执行一遍要1944个时钟。可见与传统8051相比较，STC新15系列的指令执行速度大幅提升，平均速度快8~12倍。

## 算术操作类指令

| 助记符            | 功能说明              | 字节数 | 传统8051单片机所需时钟 | STC15F104ESW系列单片机所需时钟<br>(采用STC-Y5 CPU内核指令集) | 效率提升 |
|----------------|-------------------|-----|---------------|----------------------------------------------|------|
| ADD A, Rn      | 寄存器内容加到累加器        | 1   | 12            | 1                                            | 12倍  |
| ADD A, direct  | 直接地址单元中的数据加到累加器   | 2   | 12            | 2                                            | 6倍   |
| ADD A, @Ri     | 间接RAM中的数据加到累加器    | 1   | 12            | 2                                            | 6倍   |
| ADD A, #data   | 立即数加到累加器          | 2   | 12            | 2                                            | 6倍   |
| ADDC A, Rn     | 寄存器带进位加到累加器       | 1   | 12            | 1                                            | 12倍  |
| ADDC A, direct | 直接地址单元的内容带进位加到累加器 | 2   | 12            | 2                                            | 6倍   |
| ADDC A, @Ri    | 间接RAM内容带进位加到累加器   | 1   | 12            | 2                                            | 6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            | 1                                            | 6倍   |
| SUBB A, direct | 累加器带借位减直接地址单元的内容  | 2   | 12            | 2                                            | 6倍   |
| SUBB A, @Ri    | 累加器带借位减间接RAM中的内容  | 1   | 12            | 2                                            | 6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            | 1                                            | 12倍  |
| INC Rn         | 寄存器加1             | 1   | 12            | 2                                            | 6倍   |
| INC direct     | 直接地址单元加1          | 2   | 12            | 3                                            | 4倍   |
| INC @Ri        | 间接RAM单元加1         | 1   | 12            | 3                                            | 4倍   |
| DEC A          | 累加器减1             | 1   | 12            | 1                                            | 12倍  |
| DEC Rn         | 寄存器减1             | 1   | 12            | 2                                            | 6倍   |
| DEC direct     | 直接地址单元减1          | 2   | 12            | 3                                            | 4倍   |
| DEC @Ri        | 间接RAM单元减1         | 1   | 12            | 3                                            | 4倍   |
| INC DPTR       | 地址寄存器DPTR加1       | 1   | 24            | 1                                            | 24倍  |
| MUL AB         | A乘以B              | 1   | 48            | 2                                            | 24倍  |
| DIV AB         | A除以B              | 1   | 48            | 6                                            | 8倍   |
| DA A           | 累加器十进制调整          | 1   | 12            | 3                                            | 4倍   |

## 逻辑操作类指令

| 助记符               | 功能说明             | 字节数 | 传统8051单片机所需时钟 | STC15F2K6S02系列单片机所需时钟<br>(采用STC-Y5 CPU内核指令集) | 效率提升 |
|-------------------|------------------|-----|---------------|----------------------------------------------|------|
| ANL A, Rn         | 累加器与寄存器相“与”      | 1   | 12            | 1                                            | 12倍  |
| ANL A, direct     | 累加器与直接地址单元相“与”   | 2   | 12            | 2                                            | 6倍   |
| ANL A, @Ri        | 累加器与间接RAM单元相“与”  | 1   | 12            | 2                                            | 6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            | 3                                            | 4倍   |
| ANL direct, #data | 直接地址单元与立即数相“与”   | 3   | 24            | 3                                            | 8倍   |
| ORL A, Rn         | 累加器与寄存器相“或”      | 1   | 12            | 1                                            | 12倍  |
| ORL A, direct     | 累加器与直接地址单元相“或”   | 2   | 12            | 2                                            | 6倍   |
| ORL A, @Ri        | 累加器与间接RAM单元相“或”  | 1   | 12            | 2                                            | 6倍   |
| ORL A, #data      | 累加器与立即数相“或”      | 2   | 12            | 2                                            | 6倍   |
| ORL direct, A     | 直接地址单元与累加器相“或”   | 2   | 12            | 3                                            | 4倍   |
| ORL direct, #data | 直接地址单元与立即数相“或”   | 3   | 24            | 3                                            | 8倍   |
| XRL A, Rn         | 累加器与寄存器相“异或”     | 1   | 12            | 1                                            | 12倍  |
| XRL A, direct     | 累加器与直接地址单元相“异或”  | 2   | 12            | 2                                            | 6倍   |
| XRL A, @Ri        | 累加器与间接RAM单元相“异或” | 1   | 12            | 2                                            | 6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            | 3                                            | 4倍   |
| XRL direct, #data | 直接地址单元与立即数相“异或”  | 3   | 24            | 3                                            | 8倍   |
| CLR A             | 累加器清“0”          | 1   | 12            | 1                                            | 12倍  |
| CPL A             | 累加器求反            | 1   | 12            | 1                                            | 12倍  |
| RL A              | 累加器循环左移          | 1   | 12            | 1                                            | 12倍  |
| RLC A             | 累加器带进位位循环左移      | 1   | 12            | 1                                            | 12倍  |
| RR A              | 累加器循环右移          | 1   | 12            | 1                                            | 12倍  |
| RRC A             | 累加器带进位位循环右移      | 1   | 12            | 1                                            | 12倍  |
| SWAP A            | 累加器内高低半字节交换      | 1   | 12            | 1                                            | 12倍  |

## 数据传送类指令

| 助记符                | 功能说明                                      | 字节数 | 传统8051单片机所需时钟 | STC15F104ESW系列单片机所需时钟<br>(采用STC-Y5 CPU内核指令集) | 效率提升 |
|--------------------|-------------------------------------------|-----|---------------|----------------------------------------------|------|
| MOV A, Rn          | 寄存器内容送入累加器                                | 1   | 12            | 1                                            | 12倍  |
| MOV A, direct      | 直接地址单元中的数据送入累加器                           | 2   | 12            | 2                                            | 6倍   |
| MOV A, @Ri         | 间接RAM中的数据送入累加器                            | 1   | 12            | 2                                            | 6倍   |
| MOV A, #data       | 立即数送入累加器                                  | 2   | 12            | 2                                            | 6倍   |
| MOV Rn, A          | 累加器内容送入寄存器                                | 1   | 12            | 1                                            | 12倍  |
| MOV Rn, direct     | 直接地址单元中的数据送入寄存器                           | 2   | 24            | 3                                            | 8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            | 2                                            | 6倍   |
| MOV direct, Rn     | 寄存器内容送入直接地址单元                             | 2   | 24            | 2                                            | 12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            | 3                                            | 8倍   |
| MOV direct, @Ri    | 间接RAM中的数据送入直接地址单元                         | 2   | 24            | 3                                            | 8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            | 2                                            | 6倍   |
| MOV @Ri, direct    | 直接地址单元数据送入间接RAM单元                         | 2   | 24            | 3                                            | 8倍   |
| MOV @Ri, #data     | 立即数送入间接RAM单元                              | 2   | 12            | 2                                            | 6倍   |
| MOV DPTR, #data16  | 16位立即数送入数据指针                              | 3   | 24            | 3                                            | 8倍   |
| MOVC A, @A+DPTR    | 以DPTR为基地址变址寻址单元中的数据送入累加器                  | 1   | 24            | 5                                            | 4.8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        | 将逻辑上在片外、物理上在片内的扩展RAM(8位地址)的内容送入累加器A中，读操作  | 1   | 24            | 3                                            | 8倍   |
| MOVX @Ri, A        | 将累加器A的内容送入逻辑上在片外、物理上在片内的扩展RAM(8位地址)中，写操作  | 1   | 24            | 4                                            | 8倍   |
| MOVX A, @DPTR      | 将逻辑上在片外、物理上在片内的扩展RAM(16位地址)的内容送入累加器A中，读操作 | 1   | 24            | 2                                            | 12倍  |
| MOVX @DPTR, A      | 将累加器A的内容送入逻辑上在片外、物理上在片内的扩展RAM(16位地址)中，写操作 | 1   | 24            | 3                                            | 8倍   |
| PUSH direct        | 直接地址单元中的数据压入堆栈                            | 2   | 24            | 3                                            | 8倍   |
| POP direcct        | 栈底数据弹出送入直接地址单元                            | 2   | 24            | 2                                            | 12倍  |
| XCH A, Rn          | 寄存器与累加器交换                                 | 1   | 12            | 2                                            | 6倍   |
| XCH A, direct      | 直接地址单元与累加器交换                              | 2   | 12            | 3                                            | 4倍   |
| XCH A, @Ri         | 间接RAM与累加器交换                               | 1   | 12            | 3                                            | 4倍   |
| XCHD A, @Ri        | 间接RAM的低半字节与累加器交换                          | 1   | 12            | 3                                            | 4倍   |



## 布尔变量操作类指令

| 助记符          | 功能说明             | 字节数 | 传统8051单片机所需时钟 | STC15F104ESW系列单片机所需时钟<br>(采用STC-Y5 CPU内核指令集) | 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            | 3                                            | 4倍   |
| SETB C       | 置1进位位            | 1   | 12            | 1                                            | 12倍  |
| SETB bit     | 置1直接地址位          | 2   | 12            | 3                                            | 4倍   |
| CPL C        | 进位位求反            | 1   | 12            | 1                                            | 12倍  |
| CPL bit      | 直接地址位求反          | 2   | 12            | 3                                            | 4倍   |
| ANL C, bit   | 进位位和直接地址位相“与”    | 2   | 24            | 2                                            | 12倍  |
| ANL C, /bit  | 进位位和直接地址位的反码相“与” | 2   | 24            | 2                                            | 12倍  |
| ORL C, bit   | 进位位和直接地址位相“或”    | 2   | 24            | 2                                            | 12倍  |
| ORL C, /bit  | 进位位和直接地址位的反码相“或” | 2   | 24            | 2                                            | 12倍  |
| MOV C, bit   | 直接地址位送入进位位       | 2   | 12            | 2                                            | 12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            | 5                                            | 4.8倍 |
| JNB bit, rel | 直接地址位为0则转移       | 3   | 24            | 5                                            | 4.8倍 |
| JBC bit, rel | 直接地址位为1则转移, 该位清0 | 3   | 24            | 5                                            | 4.8倍 |

本次指令系统总结更新于2011-10-17日止

## 控制转移类指令

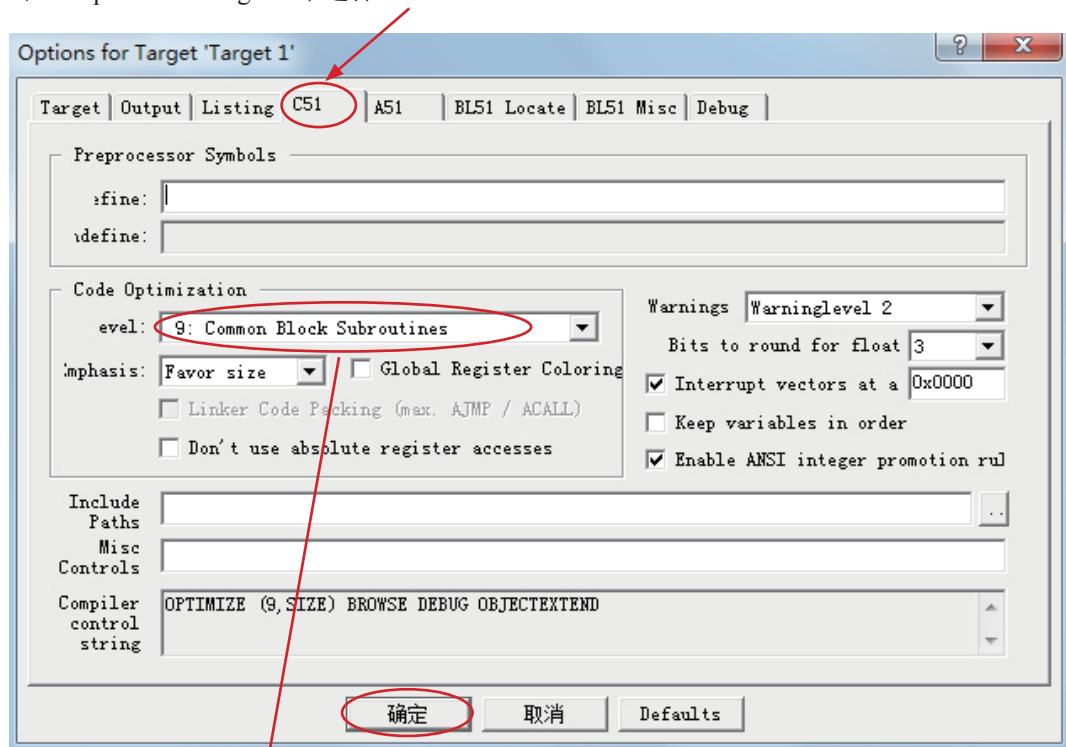
| 助记符                  | 功能说明                 | 字节数 | 传统8051单片机所需时钟 | STC15F104ESW系列单片机所需时钟<br>(采用STC-Y5 CPU内核指令集) | 效率提升 |
|----------------------|----------------------|-----|---------------|----------------------------------------------|------|
| ACALL addr11         | 绝对（短）调用子程序           | 2   | 24            | 4                                            | 6倍   |
| LCALL addr16         | 长调用子程序               | 3   | 24            | 4                                            | 6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 rel             | 相对转移                 | 2   | 24            | 3                                            | 8倍   |
| JMP @A+DPTR          | 相对于DPTR的间接转移         | 1   | 24            | 5                                            | 4.8倍 |
| JZ rel               | 累加器为零转移              | 2   | 24            | 4                                            | 6倍   |
| JNZ rel              | 累加器非零转移              | 2   | 24            | 4                                            | 6倍   |
| CJNE A, direct, rel  | 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, rel   | 累加器与立即数比较，不相等则转移     | 3   | 24            | 4                                            | 6倍   |
| CJNE Rn, #data, rel  | 寄存器与立即数比较，不相等则转移     | 3   | 24            | 4                                            | 6倍   |
| CJNE @Ri, #data, rel | 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, rel         | 寄存器减1，非零转移           | 2   | 24            | 4                                            | 6倍   |
| DJNZ direct, rel     | 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  |

以上指令集均属于 STC-Y5 CPU内核指令集

## 附录H：如何利用Keil C软件减少代码长度

在Keil C软件中选择作如下设置，能将原代码长度最大减少10K。

1. 在“Project”菜单中选择“Options for Target”
2. 在“Options for Target”中选择“C51”



3. 选择按空间大小，9级优化程序
4. 点击“确定”后，重新编译程序即可。

## 附录I：逻辑代数的基础

### ——无微机原理基础的用户请从本章开始学习

这一章主要讲述的内容有：①在数字设备中进行算术运算的基本知识——数制和编码；②数字电路中一些常用逻辑运算及其图形符号。它们是学习单片机这门课程的基础。对于没有微机原理基础的用户和同学，请从这章开始学习。

## I.1 数制与编码

数制是人们利用符号进行计数的科学方法。数制有很多种，常用的数制有：二进制，十进制和十六进制。

进位计数制是把数划分为不同的位数，逐位累加，加到一定数量之后，再从零开始，同时向高位进位。进位计数制有三个要素：数码符号、进位规律和计数基数。下表是各常用数制的总体介绍。

| 常用的数制 | 表示符号 | 数码符号                                | 进制规律  | 计数基数 |
|-------|------|-------------------------------------|-------|------|
| 二进制   | B    | 0、1                                 | 逢二进一  | 2    |
| 十进制   | D    | 0、1、2、3、4、5、6、7、8、9                 | 逢十进一  | 10   |
| 十六进制  | H    | 0、1、2、3、4、5、6、7、8、9、<br>A、B、C、D、E、F | 逢十六进一 | 16   |

我们日常生活中计数一般采用十进制。计算机中采用的是二进制，因为二进制具有运算简单，易实现且可靠，为逻辑设计提供了有利的途径、节省设备等优点。为区别于其它进制数，二进制数的书写通常在数的右下方注上基数2，或加后面加B表示。二进制数中每一位仅有0和1两个可能的数码，所以计数基数为2。二进制数的加法和乘法运算如下：

$$\begin{array}{lll} 0+0=0 & 0+1=1+0=1 & 1+1=10 \\ 0\times 0=0 & 0\times 1=1\times 0=0 & 1\times 1=1 \end{array}$$

由于二进制数在使用中位数太长，不容易记忆，为了便于描述，又常用十六进制作为二进制的缩写。十六进制通常在表示时用尾部标志H或下标16以示区别。

### I.1.1 数制转换

现在我们来介绍这些常用数制之间的转换。

#### 一：二进制 — 十进制转换

方法：将二进制数按权(如下式)展开，然后将各项的数值按十进制数相加，就得到相应的等值十进制数。

例如：N=(1101.101)B，那么N所对应的十进制数时多少呢？

$$\text{按权展开 } N=1\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}=8+4+0+1+0.5+0+0.125=(13.625)D$$

## 二: 十进制 — 二进制转换

方法: 分两部分进行即整数部分和小数部分。

## ① 整数部分转换(基数除法):

- ★ 把我们要转换的数除以二进制的基数(二进制的基数为2), 把余数作为二进制的最低位;
- ★ 把上一次得的商在除以二进制基数(即2), 把余数作为二进制的次低位;
- ★ 继续上一步, 直到最后的商为零, 这时的余数就是二进制的最高位。

## ② 小数部分转换(基数乘法):

- ★ 把要转换数的小数部分乘以二进制的基数(二进制的基数为2), 把得到的整数部分作为二进制小数部分的最高位;
- ★ 把上一步得的小数部分再乘以二进制的基数(即2), 把整数部分作为二进制小数部分的次高位;
- ★ 继续上一步, 直到小数部分变成零为止。或者达到预定的要求也可以。

例如: 将 $(213.8125)_{10}$ 化为二进制数可按如下进行:

先化整数部分:

$$\begin{array}{rcl}
 2 \overline{) 213} & \cdots \cdots \cdots & \text{余数}=1=k_0 \\
 2 \overline{) 106} & \cdots \cdots \cdots & \text{余数}=0=k_1 \\
 2 \overline{) 53} & \cdots \cdots \cdots & \text{余数}=1=k_2 \\
 2 \overline{) 26} & \cdots \cdots \cdots & \text{余数}=0=k_3 \\
 2 \overline{) 13} & \cdots \cdots \cdots & \text{余数}=1=k_4 \\
 2 \overline{) 6} & \cdots \cdots \cdots & \text{余数}=0=k_5 \\
 2 \overline{) 3} & \cdots \cdots \cdots & \text{余数}=1=k_6 \\
 2 \overline{) 1} & \cdots \cdots \cdots & \text{余数}=1=k_7 \\
 0 & & 
 \end{array}$$

于是整数部分 $(213)_{10}=(11010101)_2$

再化小数部分:

$$\begin{array}{rcl}
 0.8125 & & \\
 \times 2 & & \\
 \hline
 1.6250 & \cdots \cdots \cdots & \text{整数部分}=1=k_1 \\
 0.6250 & & \\
 \times 2 & & \\
 \hline
 1.2500 & \cdots \cdots \cdots & \text{整数部分}=1=k_2 \\
 0.2500 & & \\
 \times 2 & & \\
 \hline
 0.5000 & \cdots \cdots \cdots & \text{整数部分}=0=k_3 \\
 0.5000 & & \\
 \times 2 & & \\
 \hline
 1.0000 & \cdots \cdots \cdots & \text{整数部分}=1=k_4
 \end{array}$$

于是小数部分 $(0.8125)_{10}=(0.1101)_2$

综上所述, 十进制数 $213.8125=(11010101.1101)_2=(11010101.1101)_B$

### 三：二进制 — 十六进制转换

方法：二进制和十六进制之间满足 $2^4$ 的关系，因此把要转换的二进制从低位到高位每4位一组，高位不足时在有效位前面添“0”，然后把每组二进制数转换成十六进制即可。

例如，将(010111011110.11010010)B转换为十六进制数：

$$\begin{array}{ccccccc} (0101 & 1101 & 1110 & . & 1101 & 0010) & \text{B} \\ \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \\ = ( & 5 & & \text{D} & \text{E} & & \text{B} & & 2 & ) & \text{H} \end{array}$$

于是，(010111011110.11010010)B=(5DE.B2)H

### 四：十六进制 — 二进制转换

方法：十六进制转换为二进制时，把上面二进制转换十六进制的过程逆过来，即转换时只需将十六进制的每一位用等值的4位二进制代替就行了。

例如：将(C1B.C6)H转换为二进制数：

$$\begin{array}{ccccccc} ( & \text{C} & & 1 & & \text{B} & . & \text{C} & & 6 & ) & \text{H} \\ \downarrow & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ = ( & 1100 & & 0001 & & 1011 & & 1100 & & 0110 & ) & \text{B} \end{array}$$

于是，(C1B.C6)H=(110000011011.11000110)B

### 五：十六进制 — 十进制转换

方法：将十六进制数按权(如下式)展开，然后将各项的数值按十进制数相加，就得到相应的等值十进制数。

例如：N=(2A.7F)H，那么N所对应的十进制数时多少呢？

$$\text{按权展开 } N = 2 \times 16^1 + 10 \times 16^0 + 7 \times 16^{-1} + 15 \times 16^{-2} = 32 + 10 + 0.4375 + 0.05859375 = (42.49609375) \text{D}$$

于是，(2A.7F)H=(42.49609375)D

### 六：十进制 — 十六进制转换

方法：将十进制数转换为十六进制数时，可以先将十进制数转换为二进制数，然后再将得到的二进制数转换为等值的十六进制数。

### I.1.2 原码、反码及补码

在生活中,数有正负之分,在计算机中是怎样表示数的正负符号呢?

在生活中表示数的时候一般都是把正数前面加一个“+”,负数前面加一个“-”,但是计算机是不认识这些的,通常在二进制数前面增加一位符号位。符号位为“0”表示“+”,符号位为“1”表示“-”。这种形式的二进制数称为原码。如果原码为正数,则原码的反码和补码都与原码相同。如果原码为负数,则将原码(除符号位外)按位取反,所得的新二进制数称为原码的反码,反码加1为其补码。

原码、反码、补码这三种形式的总结如下表所示:

|    | 真值 | 原码 | 反码          | 补码      |
|----|----|----|-------------|---------|
| 正数 | +N | 0N | 0N          | 0N      |
| 负数 | -N | 1N | $(2^n-1)+N$ | $2^n+N$ |

例1:求+18和-18八位原码、反码、补码形式。

| 真值  | 原码       | 反码       | 补码       |
|-----|----------|----------|----------|
| +18 | 00010010 | 00010010 | 00010010 |
| -18 | 10010010 | 11101101 | 11101110 |

### I.1.3 常用编码

指定某一组二进制数去代表某一指定的信息,就称为编码。

一:十进制编码

用二进制码表示的十进制数,称为十进制编码。它具有二进制的形式,还具有十进制的特点它可作为人们与数字系统的联系的一种间表示。十进制编码有很多种,最常用的一种是BCD码,又称8421码。

下面我们用表列出几种常见的十进制编码:

| 十进制数 \ 编码种类 | 8421码<br>(BCD码) | 余3码  | 2421码 | 5211码 | 7321码 |
|-------------|-----------------|------|-------|-------|-------|
| 0           | 0000            | 0011 | 0000  | 0000  | 0000  |
| 1           | 0001            | 0100 | 0001  | 0001  | 0001  |
| 2           | 0010            | 0101 | 0010  | 0100  | 0010  |
| 3           | 0011            | 0110 | 0011  | 0101  | 0011  |
| 4           | 0100            | 0111 | 0100  | 0111  | 0101  |
| 5           | 0101            | 1000 | 1011  | 1000  | 0110  |
| 6           | 0110            | 1001 | 1100  | 1001  | 0111  |
| 7           | 0111            | 1010 | 1101  | 1100  | 1000  |
| 8           | 1000            | 1011 | 1110  | 1101  | 1001  |
| 9           | 1001            | 1100 | 1111  | 1111  | 1010  |
| 权           | 8421            |      | 2421  | 5211  | 7321  |

十进制编码分为有权和无权编码。有权编码是指每一位十进制数符均用一组四位二进制码来表示，而且二进制码的每一位都有固定权值。无权编码是指二进制码中每一位都没有固定的权值。上表中8421码(即BCD码)、2421码、5211码、7321码都是有权编码，而余3码是无权编码。

## 二： 奇偶校验码

在数据的存取、运算和传送过程中，难免会发生错误，把“1”错成“0”或把“0”错成“1”。奇偶校验码是一种能检验这种错误的代码。它分为两部分：信息位和奇偶校验位。有奇数个“1”称为奇校验，有偶数个“1”则称为偶校验。

STC MCU Limited



## I.2 几种常用的逻辑运算及其图形符号

逻辑代数中常用的运算有：与(AND)、或(OR)、非(NOT)、与非(NAND)、或非(NOR)、与或非(AND-NOR)、异或(EXCLUSIVE OR)、同或(EXCLUSIVE NOR)等。其中与(AND)、或(OR)、非(NOT)运算时三种最基本的运算。


### 一：与运算及与门

与运算：决定事件结果的全部条件同时具备时，事件才发生。

逻辑变量A和B进行与运算时可写成： $Y=A \cdot B$

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 0 |
| 0   | 1 | 0 |
| 1   | 0 | 0 |
| 1   | 1 | 1 |

与门：实行与逻辑运算的单元电路。

与门图形符号：


### 二：或运算及或门

或运算：决定事件结果各条件中只要有任何一个满足，事件就会发生。

逻辑变量A和B进行或运算时可写成： $Y=A+B$

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 0 |
| 0   | 1 | 1 |
| 1   | 0 | 1 |
| 1   | 1 | 1 |

或门：实行或逻辑运算的单元电路。

或门图形符号：


### 三：非运算及非门

非运算：条件具备时，事件不会发生；条件不具备时，事件才会发生。

逻辑变量A进行非运算时可写成： $Y=A'$

| 真值表 |   |
|-----|---|
| A   | Y |
| 0   | 1 |
| 1   | 0 |

非门：实行非逻辑运算的单元电路。


非门图形符号：

#### 四：与非运算及与非图形符号

与非运算：先进行与运算，然后将结果求反，最后得到的即为与非运算结果。

逻辑变量A和B进行与非运算时可写成： $Y=(A \cdot B)'$

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 1 |
| 0   | 1 | 1 |
| 1   | 0 | 1 |
| 1   | 1 | 0 |


与非图形符号：

#### 五：或非运算及或非图形符号

或非运算：先进行或运算，然后将结果求反，最后得到的即为或非运算结果。

逻辑变量A和B进行或非运算时可写成： $Y=(A+B)'$

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 1 |
| 0   | 1 | 0 |
| 1   | 0 | 0 |
| 1   | 1 | 0 |

或非图形符号：

#### 六：与或非运算及与或非图形符号

与或非运算：在与或非逻辑运算中有4个逻辑变量A、B、C、D。假设A和B为一组，C和D为一组，A、B之间以及C、D之间都是与的关系，只要A、B或C、D任何一组同时为1，输出Y就是0。只有当每一组输入都不全是1时，输出Y才是1。

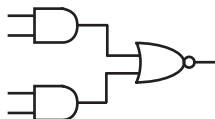
逻辑变量A和B进行或非运算时可写成： $Y=(A \cdot B + C \cdot D)'$

| 真值表 |   |   |   |   |
|-----|---|---|---|---|
| A   | B | C | D | Y |
| 0   | 0 | 0 | 0 | 1 |
| 0   | 0 | 0 | 1 | 1 |
| 0   | 0 | 1 | 0 | 1 |
| 0   | 0 | 1 | 1 | 0 |
| 0   | 1 | 0 | 0 | 1 |
| 0   | 1 | 0 | 1 | 1 |
| 0   | 1 | 1 | 0 | 1 |
| 0   | 1 | 1 | 1 | 0 |
| 1   | 0 | 0 | 0 | 1 |
| 1   | 0 | 0 | 1 | 1 |

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

(接上表)

与或非图形符号:



## 七: 异或运算及异或图形符号

异或运算: 当A、B不同时, 输出Y为1; 而当A、B相同时, 输出Y为0。

逻辑变量A和B进行异或运算时可写成:  $Y = A \oplus B = (A \cdot B') + (A' \cdot B)$ 

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 0 |
| 0   | 1 | 1 |
| 1   | 0 | 1 |
| 1   | 1 | 0 |

异或图形符号:

## 八: 同或运算及同或图形符号

同或运算: 当A、B不同时, 输出Y为0; 而当A、B相同时, 输出Y为1。

逻辑变量A和B进行同或运算时可写成:  $Y = A \odot B = (A \cdot B) + (A' \cdot B')$ 

| 真值表 |   |   |
|-----|---|---|
| A   | B | Y |
| 0   | 0 | 1 |
| 0   | 1 | 0 |
| 1   | 0 | 0 |
| 1   | 1 | 1 |

同或图形符号: